13th International Conference on Typed Lambda Calculi and Applications

TLCA'15, July 1–3, 2015, Warsaw, Poland

Edited by

Thorsten Altenkirch

LIPIcs – Vol. 38 – TLCA 2015 www.dagstuhl.de/lipics

Editor

Thorsten Altenkirch
School of Computer Science
University of Nottingham
txa@cs.nott.ac.uk

ACM Classification 1998
D.1.1 Applicative (Functional) Programming, D.3.2 Language Classifications, D.3.3 Language Constructs and Features, E.1 Data Structures, F.3 Logics and Meanings of Programs, F.4 Mathematical Logic and Formal Languages, I.1 Symbolic and Algebraic Manipulation, I.2.3 Deduction and Theorem Proving

ISBN 978-3-939897-87-3

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-87-3.

Publication date
July, 2015

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.TLCA.2015.i

ISBN 978-3-939897-87-3 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015).
Editor: Thorsten Altenkirch

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the papers of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015), which was held during 1–3 July 2015, in Warsaw, Poland. TLCA 2015 was part of the 8th International Conference on Rewriting, Deduction, and Programming (RDP 2015), together with the 26th International Conference on Rewriting Techniques and Applications (RTA 2015), the Workshop on Higher-Dimensional Rewriting and Applications (HDRA), the Workshop on Homotopy Type Theory / Univalent Foundations (HoTT/UF), the 29th International Workshop on Unification (UNIF 2015), the second International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2015), and the annual meeting of the IFIP Working Group 1.6 on Term Rewriting.

The TLCA series of conferences serves as a forum for presenting original research results that are broadly relevant to the theory and applications of lambda calculus. Previous TLCA conferences were held in Utrecht (1993), Edinburgh (1995), Nancy (1997), L'Aquila (1999), Kraków (2001), Valencia (2003), Nara (2005), Paris (2007), Brasília (2009), Novi Sad (2011), Eindhoven (2013) and Vienna (2014, merged with RTA).

A total of 23 papers were accepted out of 46 submissions for presentation at TLCA 2015 and for inclusion in the proceedings. I would like to thank everyone who submitted a paper and to express my regret that many interesting papers could not be included. Each submitted paper was reviewed by at least three members of the Programme Committee, who were assisted in their work by 71 external reviewers. I thank the members of the Programme Committee and the external reviewers for their review work, as well as Andrei Voronkov for providing the EasyChair system which proved invaluable throughout the review process and the preparation of this volume.

In addition to the contributed papers, the TLCA 2015 programme contained invited talks by Helene Kirchner (joint with RTA 2015), Herman Geuvers and Martin Hofmann.

Many people helped to make TLCA 2015 a success. I would like to thank RDP 2015 Chair Aleksy Schubert and the Organising Committee, the local organising team, TLCA Publicity Chair Luca Paolini, and the TLCA Steering Committee, especially Paweł Urzyczyn.

June 2015
Thorsten Altenkirch

13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015).
Editor: Thorsten Altenkirch
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

TLCA 2015 Organisation

Program Committee

Thorsten Altenkirch (Chair)	University of Nottingham, UK
Steve Awodey	Carnegie Mellon University, USA
Stefano Berardi	University of Torino, Italy
James Chapman	Technical University of Tallinn, Estonia
Gilles Dowek	INRIA, France
Peter Dybjer	Chalmers University, Sweden
Silvia Ghilezan	University of Novi Sad, Serbia
Mauro Jaskelioff	University of Rosario, Argentina
Chantal Keller	Microsoft Research Cambridge, UK
Paul Levy	University of Birmingham, UK
Ralph Matthes	IRIT, CNRS & University of Toulouse, France
Keiko Nakata	FireEye Dresden, Germany
Damian Niwiński	University of Warsaw, Poland
Valeria de Paiva	Nuance Communications, USA
Matthieu Sozeau	INRIA, France
Wouter Swierstra	University of Utrecht, The Netherlands

RDP Conference Chair

Aleksy Schubert	University of Warsaw, Poland

Organising Committee

Marcin Benke	University of Warsaw, Poland
Jacek Chrząszcz	University of Warsaw, Poland
Patryk Czarnik	University of Warsaw, Poland
Łukasz Czajka	University of Warsaw, Poland
Krystyna Jaworska	University of Warsaw, Poland
Paweł Urzyczyn	University of Warsaw, Poland
Daria Walukiewicz-Chrząszcz	University of Warsaw, Poland
Maciej Zielenkiewicz	University of Warsaw, Poland

13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015).
Editor: Thorsten Altenkirch
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

TLCA Steering Committee

Sandra Alves	University of Porto, Portugal
Steve Awodey	Carnegie Mellon University, USA
Pierre-Louis Curien	CNRS and Université Paris Diderot, France
Ugo Dal Lago	University of Bologna, Italy
Gilles Dowek	INRIA, France
Masahito Hasegawa	Kyoto University, Japan
Hugo Herbelin	Université Paris Diderot, France
Martin Hofmann	Ludwig-Maximillians-University, Munich, Germany
Luke Ong	University of Oxford, UK
Michele Pagani	Université Paris Diderot, France
Jens Palsberg	University of California, Los Angeles,, USA
Jakob Rehof	University of Dortmund, Germany
Paweł Urzyczyn (Chair)	University of Warsaw, Poland
Philip Wadler	University of Edinburgh, UK

TLCA Honorary Advisors

Samson Abramsky	University of Oxford, UK
Henk Barendregt	Radboud University Nijmegen, The Netherlands
Roger Hindley	Swansea University, UK
Mariangiola Dezani-Ciancaglini	University of Turin, Italy
Simona Ronchi Della Rocca	University of Turin, Italy

TLCA Publicity Chair

Luca Paolini	University of Turin, Italy

External Reviewers

Andreas Abel
Ki Yung Ahn
Bruno Barras
Alexis Bernadet
Marc Bezem
Katalin Bimbó
James Brotherston
Ulrik Buchholtz
Jacek Chrząszcz
Ranald Clouston
Thierry Coquand
Pierre-Louis Curien
Łukasz Czajka
Nils Anders Danielsson
Alejandro Diaz-Caro
Harley Eades Iii
Didier Galmiche
Ferruccio Guidi
Jennifer Hackett
Hugo Herbelin
Martin Hofmann
Chung-Kil Hur
Danko Ilik
Florent Jacquemard
Alan Jeffrey
Kentaro Kikuchi
Daisuke Kimura
Nicolai Kraus
Keiichirou Kusakari
Jarek Kusmierek
James Laird
Pierre Lescanne
Silvia Likavec
Sam Lindley
Peter LeFanu Lumsdaine
Ian Mackie
Assia Mahboubi
Samuele Maschio
Conor McBride
Henryk Michalewski
Edward Morehouse
J. Garrett Morris
Guillaume Munch-Maccagnoni
Koji Nakazawa
Clive Newstead
Christine Paulin-Mohring
Iosif Petrakis
Elaine Pimentel
Andrew Pitts
Nicolas Pouillard
Jorge A. Parez
Colin Riba
Eike Ritter
Exequiel Rivas
Edmund Robinson
Reuben Rowe
Jorge Luis Sacchini
Alexis Saurin
Bas Spitters
Sam Staton
Martin Sulzmann
Nicolas Tabareau
Tarmo Uustalu
Benno van den Berg
Andrea Vezzosi
Ignas Vysniauskas
Benjamin Werner
Hongwei Xi
Noam Zeilberger
Yu Zhang
Florian Zuleger

Herbrand Disjunctions, Cut Elimination and Context-Free Tree Grammars

Bahareh Afshari, Stefan Hetzl, and Graham E. Leigh

Institute of Discrete Mathematics and Geometry
Vienna University of Technology
Wiedner Hauptstraße 8-10, 1040 Vienna, Austria
{bahareh.afshari, stefan.hetzl, graham.leigh}@tuwien.ac.at

Abstract

Recently a new connection between proof theory and formal language theory was introduced. It was shown that the operation of cut elimination for proofs in first-order predicate logic involving Π_1-cuts corresponds to computing the language of a particular class of regular tree grammars. The present paper expands this connection to the level of Π_2-cuts. Given a proof π of a Σ_1 formula with cuts only on Π_2 formulæ, we show there is associated to π a natural context-free tree grammar whose language is finite and yields a Herbrand disjunction for π.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, F.4.3 Formal Languages

Keywords and phrases Classical logic, Context-free grammars, Cut elimination, First-order logic, Herbrand's theorem, Proof theory

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.1

1 Introduction

The computational content of proofs is a central topic of proof theory. In intuitionistic first-order logic the existential witness property states that the provability of $\exists x F$ entails the existence of some ground term t such that $F(x/t)$ is provable. The analogue of this property in classical logic is Herbrand's theorem [12] (see also [5]). In its simplest version Herbrand's theorem states that if $\exists x F$ is valid and F quantifier-free there exist closed terms $t_1, \ldots, t_k$ such that $\bigvee_{i=1}^{k} F(x/t_i)$ is a tautology. Such formulæ, quantifier-free disjunctions of instances, are hence also called Herbrand disjunctions. Herbrand's theorem applies not only to existential but arbitrary first-order formulæ, providing a tautology by replacing each non-prenex quantifier with a suitable finite disjunction or conjunction of instances. Provided one is willing to speak about provability instead of validity Herbrand's theorem even extends to classical higher-order logic, see for example [25].

A Herbrand disjunction can be read off directly from a cut-free proof though proofs with cut may be non-elementarily smaller than the shortest Herbrand disjunction [30, 26, 27]. Therefore, in general, cut elimination (or an equivalent normalisation process) is necessary in order to obtain a Herbrand disjunction. However, if one is only interested in the witness terms of a Herbrand disjunction and not in the complete cut-free proof then it would be desirable to circumvent the cumbersome process of cut elimination.

For instance, in [13, 14] it was shown that a proof $\pi \vdash \exists x F$ in which all cut formulæ have at most one quantifier induces a (totally rigid acyclic) tree grammar $\mathcal{G}_\pi$, of size no greater than the size of the proof, with a finite language that, when interpreted as a collection of witness terms, forms a Herbrand disjunction for $\exists x F$ (see Figure 2).

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 1–16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

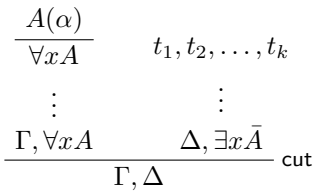


Figure 1 Π_1 cut.

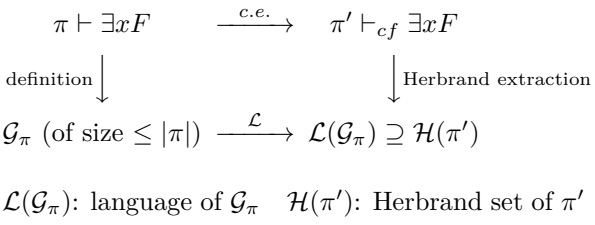


Figure 2 Proof grammars.

Grammars for proofs with only cuts of the form $\exists xA$ or $\forall xA$ with A quantifier free are remarkably simple. Let π be such a proof with end sequent $\exists xF$ where F is quantifier free. Suppose $u_1, u_2, \ldots, u_m$ are the witnesses to the existential quantifier as they appear in π. The induced grammar $\mathcal{G}_\pi$ consists of the production rules i) $\sigma \to F(u_1) \mid F(u_2) \mid \ldots \mid F(u_m)$ where σ is the starting symbol of the grammar, and ii) $\alpha \to t_1 \mid t_2 \mid \cdots \mid t_k$ for every Π_1 cut in π of the form given in Figure 1, where α is the eigenvariable of the cut and $t_1, t_2, \ldots, t_k$ are witness terms of existential quantifier in the right subproof.

1.1 Contributions

In this paper we show how the correspondence between proofs and grammars can be extended to the level of Π_2 cuts. This class of cuts is particularly important for computational applications: a Π_2 formula can be read as a specification of a program and its proof as providing an (non-deterministic) algorithm in line with the Curry–Howard correspondence.

We consider Π_2-proofs in which all cut formulæ have at most one quantifier of each sort i.e. of the form $\forall x \exists y A$ or $\forall xA$ with A quantifier free. It turns out that these proofs correspond to (rigid) context-free tree grammars:

▶ **Theorem 1.** *Let π be a Π_2-proof of a Σ_1 formula F. There is an associated rigid context-free tree grammar $\mathcal{G}_\pi$ (whose number of production rules is bounded by the size of π) with a finite language yielding a Herbrand disjunction for F.*

In fact, if we consider proofs in which every universal introduction rule is immediately followed by a cut or an existential introduction (henceforth called *simple proofs*) we have

▶ **Theorem 2.** *Let $\pi_0, \pi_1, \ldots, \pi_k$ be a sequence of simple Π_2-proofs such that π_{i+1} is obtained from π_i by an application of a cut reduction rule (as in Figure 4) to a subproof of π_i. For every $i \leq k$, $\mathcal{L}(\mathcal{G}_{\pi_0}) \supseteq \mathcal{L}(\mathcal{G}_{\pi_i})$. In particular, if the proof π_k contains only quantifier-free cuts $\mathcal{L}(\mathcal{G}_{\pi_0}) \supseteq \mathcal{H}(\pi_k)$.*

Rigid tree languages were first introduced in [21, 22] with applications to verification in mind. Later in [14] rigid regular grammars were defined by restricting the admissible derivations in the grammar with an equality constraint. In this paper we extend the notion of rigidity to context-free grammars. It plays an important role in obtaining a concise grammar for our proofs. For a Π_2-proof π, we prove the language of the induced rigid context-free grammar $\mathcal{G}_\pi$ has a bound double exponential in the size of π (see Theorem 10). This bound is optimal as it matches the blow up from cut elimination.

Structure of the article. In Section 2 we fix the calculus and cut reduction steps, and define the class of proofs under consideration. Section 3 develops the theory of rigid context-free tree grammars. In Section 4 we present the proof grammars induced by Π_2-proofs: an

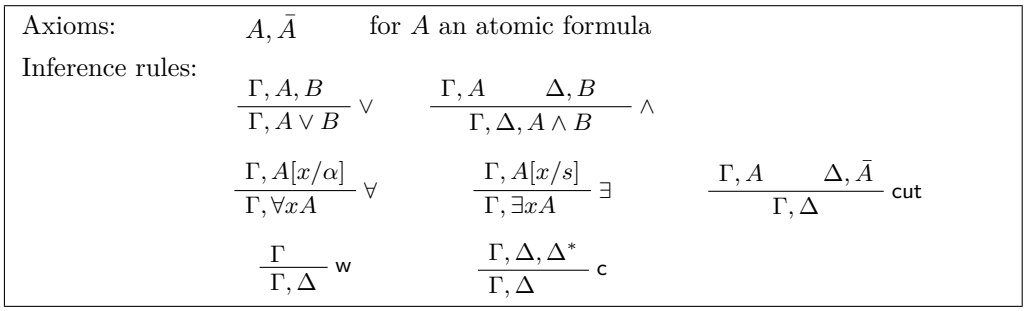


Figure 3 Axioms and rules of sequent calculus.

elementary grammar is given in Section 4.1 to motivate the definition laid out in Section 4.2. Section 5 establishes the preservation of the language of our proof grammars under transitive closure of cut reduction steps. In Section 6 we conclude by describing future work and potential applications of our results and techniques.

1.2 Related work

In [9] Gerhardy and Kohlenbach adapt Shoenfield's variant of Gödel's Dialectica interpretation to a system of pure predicate logic by explicitly adding decision-by-case constants to the target language. The resulting λ-term is first normalised and then used to directly read off a Herbrand disjunction. Heijltjes [10] and McKinley [24] study graphical formalisms of local reductions in classical first-order logic to derive a normal form corresponding to a cut-free proof from which a Herbrand disjunction is obtained. An approach similar to [10, 24] in the formalism of expansion trees [25] can be found in [19]. Historically, Hilbert's ε-calculus [20] is the first formalism which allows a step-wise computation of a Herbrand disjunction in a way that abstracts from the propositional layer of predicate logic.

Like the aforementioned formalisms, the results presented in this paper allow the computation of a Herbrand disjunction in a way that bypasses literal cut elimination. The novelty of our approach lies in the fact that this is achieved via formal grammars. On the one hand this has the consequence that standard problems from formal language theory assume a proof-theoretic meaning and hence standard algorithms can be used to solve the corresponding proof-theoretic problems. For example, an algorithm for solving the membership problem for an adequate class of grammars also solves the following proof-theoretic problem (for the corresponding class of proofs): given a proof π and a term t, is t a witness obtainable by cut-elimination from π? Usually, these 'induced' algorithms have a smaller asymptotic complexity (polynomial to at most exponential time; see e.g. [23, 22]) than the naive proof-theoretic algorithms which rely on explicitly computing the cut-free proof(s) (and need iterated-exponential time). On the other hand, the strong grip on the structure of a Herbrand disjunction afforded by a formal grammar opens the door to interesting theoretical and applied investigations (see Section 6).

2 Proof calculus

We work with a Tait-style (one-sided) sequent calculus for first-order logic with explicit weakening and contraction rules. A *proof* is a finite tree obtained from the axioms and rules laid out in Figure 3. We use capital Greek letters Γ, Δ, etc. for multisets of formulæ, $\bar{A}$ to denote the dual of the formula A obtained by de Morgan laws, and $A[x/s]$ for the formula

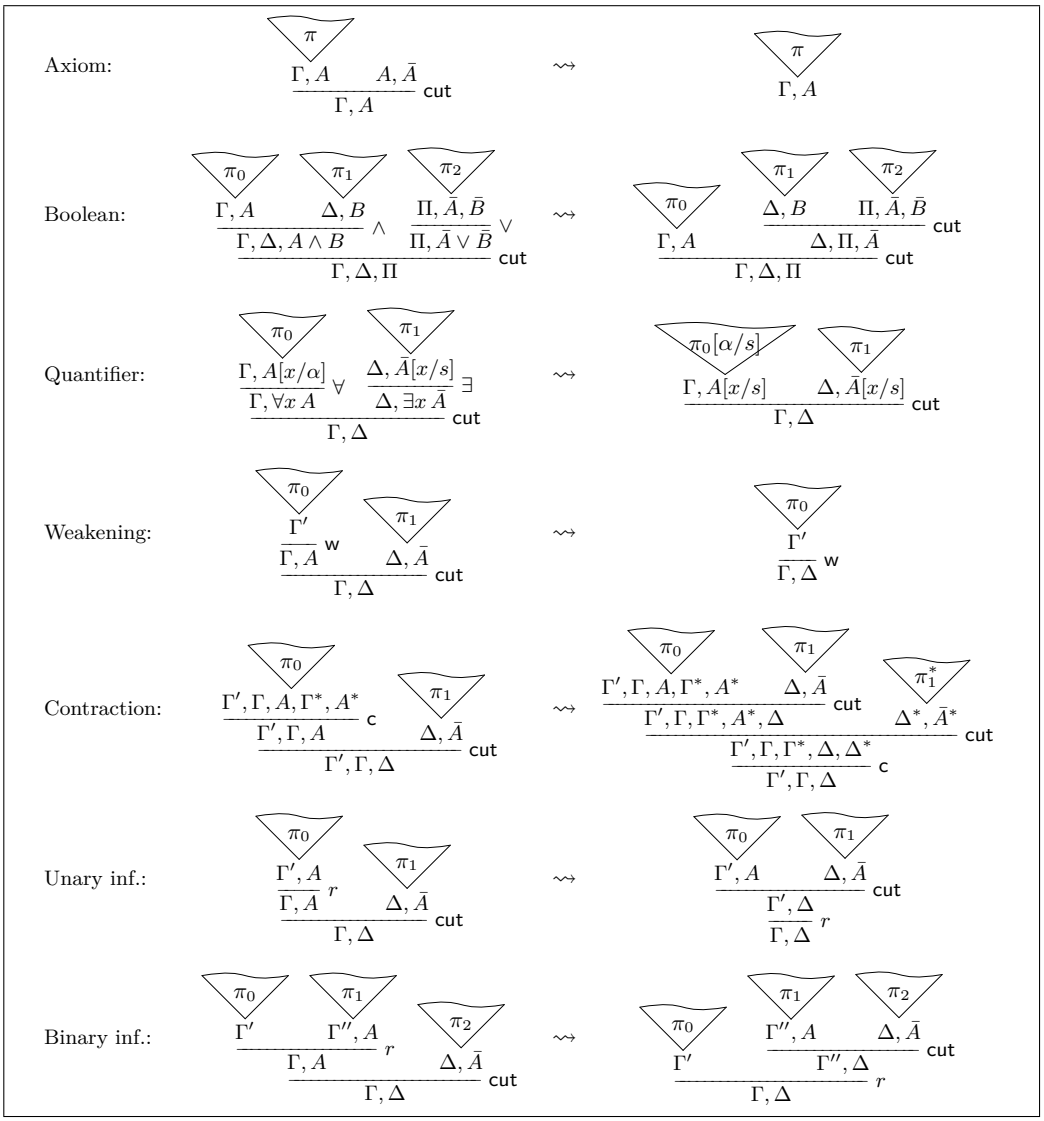


Figure 4 One-step cut reduction and permutation rules.

obtained from A by replacing x with the term s assuming this will not create any variable capture. Γ, Δ and Γ, A denote respectively the disjoint union of Γ, Δ and $\Gamma, \{A\}$.

In the $(\forall)$ rule, α is called the *eigenvariable* and must not appear free in $\Gamma, \forall x A$; in $(\exists)$ rule the term s is assumed to be free for x; and in the contraction rule (c), the set Δ^* denotes a renaming of Δ. Those formulæ which are explicitly mentioned in the premise of an inference rule are said to be *principal* in the rule applied, for example A and B are principal in $(\wedge)$ rule, every formula from Δ^* is principal in (c), and there are no principal formulæ in the weakening rule (w).

We assume all proofs are *regular* namely, all quantifiers' eigenvariables are distinct and different from any free variables. We use the following naming convention for proofs throughout the paper: lower-case Greek letters α, β, γ, etc. represent eigenvariables; x, y for bound variable symbols; and π, π', etc. for proofs. $\pi[\alpha/s]$ is the result of replacing throughout the proof π each occurrence of the variable symbol α by the term s. We write $\pi \vdash \Gamma$ to express

π is a proof of Γ and $\mathrm{EV}(\pi)$ to denote the set of eigenvariables in π. For a position p in (the tree) π, $\pi|p$ denotes the subproof of π at position p with the convention that $\pi|\langle\rangle = \pi$, and $\pi|p0$ and $\pi|p1$ are respectively the left (or only) subproof and right subproof of $\pi|p$. We write $q < p$ if q is a proper prefix of p.

It is useful to isolate a particular class of Π_2-proofs:

▶ **Definition 3** (Simple Π_2-proof)**.** A *simple* Π_2*-proof* is a proof in which i) each sequent is a finite set of prenex Π_2 formulæ, ii) the conclusion is a set of closed Π_1 and Σ_1 formulæ, iii) cut formulæ feature at most one quantifier of each sort, and iv) every universally quantified formula appearing above a cut is principal in the inference directly after its introduction (either a cut or existential introduction).

Condition (ii) above is stipulated primarily for expository purposes. Note that every sequent can be transformed into a sequent containing only Σ_1 formulæ by Skolemization and prenexification. In contrast, it is impossible to Skolemize cut formulæ in a proof since the two (dual) occurrences of the cut formula would yield dual Skolemizations. Also note that condition (iv) does not restrict provability, as any proof satisfying (i)–(iii) can be modified using simple rule permutations to also satisfy (iv). This transformation will not increase the size (number of inference rules) of the proof.

Cut reduction and Herbrand sets

The standard cut reduction rules are given in Figure 4. For proofs π, π' we write $\pi \rightsquigarrow \pi'$ if π' is the result of applying one of the rules to π (and not to a subproof). Notice that in contraction reduction a subproof is duplicated and care is taken to rename the eigenvariables (expressed by annotating the proof/sequent/formula in question by an asterisk) to maintain regularity.

Let π be a cut-free proof of Γ and suppose $A \in \Gamma$ has the form $\exists x_1 \cdots \exists x_k B$ with B quantifier free. If $\bigvee_{(s_1,\ldots,s_k)\in X} B[x_1/s_1, \ldots, x_k/s_k]$ is the Herbrand disjunction for A read off from π, we call X the *Herbrand set of* A *in* π and define $\mathcal{H}(\pi, A) = X$. In addition, $\mathcal{H}(\pi) := \{\{A\} \times \mathcal{H}(\pi, A) \mid A \in \Gamma \cap \Sigma_1\}$.

Running example

We consider a formal proof of the pigeonhole principle for two boxes via the infinite pigeonhole principle. The question of the computational content of this proof is attributed to G. Stolzenberg in [6]. A variety of analytic methods have since been applied to this proof [11, 33, 3, 2] and its generalisations [29, 28]. Despite its relatively short and symmetric nature it allows us to adequately demonstrate grammars for proofs with Π_2 cuts.

Let $f\colon \mathbb{N} \to \{0,1\}$ be a total Boolean function, let I_i express that there are infinitely many $m \in \mathbb{N}$ for which $f(m) = i$ and T express that there exists $m < n$ such that $f(m) = f(n)$. A consequence of the law of excluded middle is $I_0 \vee I_1$. Moreover I_i implies T for each i: assuming I_i, there exists $m \geq 0$ and $n \geq m + 1$ for which $f(m) = f(n) = i$. Combining these observations we conclude that T holds.

The following formalises the above argument. The formal language, Σ, consist of two unary function symbols f, s, one binary function symbol M, a constant symbol 0 and a binary relation $\leq$. We make the following definitions and abbreviations:

- $T = \exists x \exists y (x < y \wedge \mathsf{f}x = \mathsf{f}y)$,
- $I_i = \forall x \exists y (x \leq y \wedge \mathsf{f}y = \underline{i})$ for $i \in \{0,1\}$ where $\underline{0} = 0$ and $\underline{1} = \mathsf{s}0$,
- $\Gamma = \{\forall x \forall y (x \leq \mathsf{M}xy \wedge y \leq \mathsf{M}xy), \forall x (\mathsf{f}x = 0 \vee \mathsf{f}x = \mathsf{s}0)\}$,

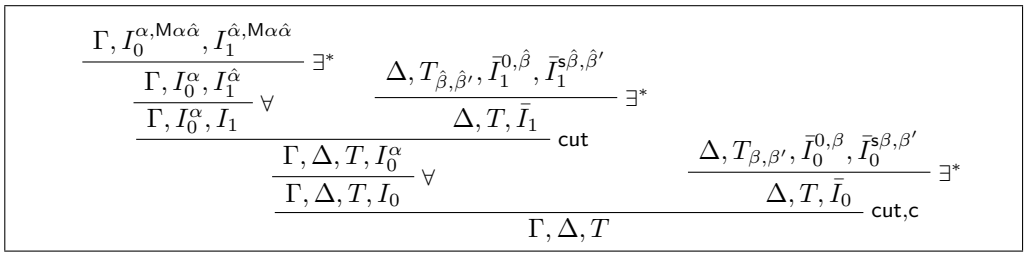


Figure 5 Simple Π_2-proof π_∞ of pigeonhole principle. The inference rules labelled $\exists^*$ represent finitely many existential introduction rules (the order of which is unimportant).

- $\Delta = \{\overline{\forall x \forall y \forall z (x = y \wedge z = y \to x = z)}, \overline{\forall x \forall y (sx \leq y \to x < y))}\}$,
- I_i^s and $I_i^{s,t}$ denote, respectively, $\exists y (s \leq y \wedge \mathsf{f}y = \underline{i})$ and $(s \leq t \wedge \mathsf{f}t = \underline{i})$,
- $T_{s,t}$ denotes $(s < t \wedge \mathsf{f}s = \mathsf{f}t)$.

The intended interpretation of the symbols is: f represents the (arbitrary) function f, s the successor function on $\mathbb{N}$, $\leq$ the standard ordering and M the binary max function.

A formal proof of the pigeonhole principle is given by the simple Π_2-proof in Figure 5 which we name π_∞. For brevity only eigenvariables and witnesses of the quantifiers and instances of the existential formula T are displayed in π_∞. The proof uses about 50 application of the axioms and rules of the calculus but the only cuts in π_∞ are the two Π_2 cuts shown in the figure. Two normal forms of the proof (of size $\sim$200) have been computed in a case study [33] from which one can read off the Herbrand set $\{(0,1),(1,2),(2,3),(0,2),(1,3)\}$ (up to interpretation of the logical symbols) relative to the formula T. Once we have introduced our grammars we shall see how this Herbrand set can be directly computed from π_∞.

3 Context-free tree grammars

In [14] and elsewhere a refinement of regular tree grammars was studied that mimics the construction of terms appearing in cut-elimination for first-order logic with Π_1 cuts. These grammars were called *rigid* tree grammars and are equivalent to the notion of rigid automata introduced and explored in [21, 22]. In this section we provide a generalisation to the class of context-free tree grammars corresponding to Π_2-proofs.

Given a ranked alphabet Σ, we let $\mathrm{Terms}(\Sigma)$ denote the set of terms in the simply-typed λ-calculus built from Σ. For a $T \in \mathrm{Terms}(\Sigma)$ we write $\mathrm{Pos}(T)$ for the set of positions in term (tree) T. For $p \in \mathrm{Pos}(T)$, $T|p$ is the subterm of T at position p.

A *context-free tree grammar* (CFG) is a tuple $\mathcal{G} = \langle N, \Sigma, \sigma, \mathrm{Pr} \rangle$ where N is a set of typed *non-terminals* of order at most 1, $\sigma \in N$ is a designated *start symbol* (of base-type ι), Σ is a ranked alphabet, called *terminals*, disjoint from N, and Pr consists of pairs (a, T) (called *production rules* and written $a \to T$) where $a \in N$ and $T \in \mathrm{Terms}(\Sigma \cup N)$ that has the same type as a. Given a CFG $\mathcal{G}$ we assume $\mathcal{G} = \langle N_\mathcal{G}, \Sigma_\mathcal{G}, \sigma_\mathcal{G}, \mathrm{Pr}_\mathcal{G} \rangle$. If the set $N_\mathcal{G}$ of non-terminals contains only symbols of order 0, $\mathcal{G}$ is a *regular tree grammar*.

Let d be a sequence $\langle \rho_i, p_i \rangle_{i<k}$ of pairs of production rules of a CFG $\mathcal{G}$ and positions, and S and T terms. We call d a *derivation from S to T*, written $d\colon S \to T$, if there exist terms $(N_i)_{i \leq k}$ such that $N_0 = S$, $N_k = T$, and for each $0 \leq i < k$,

1. ρ_i is a production rule of $\mathcal{G}$ and $p_i \in \mathrm{Pos}(N_i)$,
2. For $\rho_i = (a \to S)$, we have $N_i|p_i = a$ and $N_{i+1} = N_i[p_i/S]$, namely the result of replacing the sub-term of N_i at position p_i by S (renaming bound variables if necessary).

The sequence of terms $(N_i)_{i\leq k}$ is uniquely determined by d and S, whence we may write $d(i)$ for N_i. The *length* of d, $\mathrm{lh}(d)$, is k. We say T *is derivable from* S if there exists a derivation $d\colon S \to T$. $\mathcal{G}$ is *acyclic* if for every non-terminal $a \in N_\mathcal{G}$ and every derivation $d\colon a \to T$ with $\mathrm{lh}(d) > 0$, a does not appear in T.

The *language* of a CFG $\mathcal{G}$ is defined as $\mathcal{L}(\mathcal{G}) = \{T \in \mathrm{Terms}(\Sigma_\mathcal{G}) \mid \exists d\colon \sigma_\mathcal{G} \to T\}$. When comparing languages of CFGs it is convenient to work modulo β-convertibility. Thus for grammars $\mathcal{G}$, $\mathcal{H}$, we write $\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\mathcal{H})$ to express that for every $S \in \mathcal{L}(\mathcal{G})$ there exists a β-equivalent $T \in \mathcal{L}(\mathcal{H})$.

3.1 Rigidity

Let $\mathcal{G}$ be a CFG, suppose $\lhd$ is a transitive binary relation on $N_\mathcal{G}$, and R is a designated set of non-terminals. A derivation $d = \langle (a_i \to S_i), p_i\rangle_{i<\mathrm{lh}(d)}\colon S \to T$ induces a natural equivalence relation on the positions in T corresponding to connectedness in parse trees: for $j_0, j_1 < \mathrm{lh}(d)$, let $j_0 \sim_d j_1$ iff there exist $i_0 < j_0, j_1$ such that

1. $p_{i_0} \leq p_{j_0}, p_{j_1}$,
2. $a_{j_0} = a_{j_1} \in R$,
3. for every $k \in \{0,1\}$ and $i_0 < i < j_k < \mathrm{lh}(d)$, if $p_i \leq p_{j_k}$ then $a_{j_k} \not\lhd a_i$.

In other words, two occurrences of a non-terminal $a \in R$ in (the natural tree representation of) the derivation d are considered connected if there is no non-terminal of higher priority between them and their closest common ancestor.

Of particular interest to us is the class of derivations that respect their own $\sim$ relation.

▶ **Definition 4.** Let $\mathcal{G}$ be a CFG and suppose $\lhd$ is a transitive ordering on $N_\mathcal{G}$ and $R \subseteq N_\mathcal{G}$. A derivation $d = \langle \rho_i, p_i\rangle_{i<k}\colon S \to T$ in $\mathcal{G}$ is *rigid with respect to* $(\lhd, R)$ if for every $i, j < k$, $i \sim_d j$ implies $T|p_i = T|p_j$.

A *rigid context-free tree grammar* is a structure $\mathcal{G} = \langle N_\mathcal{G}, R_\mathcal{G}, \lhd_\mathcal{G}, \Sigma_\mathcal{G}, \sigma_\mathcal{G}, \mathrm{Pr}_\mathcal{G}\rangle$ such that $\langle N_\mathcal{G}, \Sigma_\mathcal{G}, \sigma_\mathcal{G}, \mathrm{Pr}_\mathcal{G}\rangle$ is a CFG, $R_\mathcal{G} \subseteq N_\mathcal{G}$ and $\lhd_\mathcal{G}$ is a transitive order on $N_\mathcal{G}$. $R_\mathcal{G}$ is the set of *rigid* non-terminals of $\mathcal{G}$ and $\lhd_\mathcal{G}$ is the *priority ordering of* $\mathcal{G}$. If $R_\mathcal{G} = N_\mathcal{G}$ then $\mathcal{G}$ is called *totally rigid*.

Given a rigid CFG $\mathcal{G}$ and a derivation d in its underlying CFG, we call d *rigid* if it is rigid with respect to $(\lhd_\mathcal{G}, R_\mathcal{G})$. The *language* of a rigid CFG $\mathcal{G}$, $\mathcal{L}(\mathcal{G})$, is the collection of terms derivable from rigid derivations starting from $\sigma_\mathcal{G}$:

$$\mathcal{L}(\mathcal{G}) = \{T \in \mathrm{Terms}(\Sigma_\mathcal{G}) \mid \exists d\colon \sigma_\mathcal{G} \to T \text{ and } d \text{ is } (\lhd_\mathcal{G}, R_\mathcal{G})\text{-rigid}\}.$$

Note the language of a rigid CFG is a set of closed (well-typed) base-type λ-terms.

▶ **Example 5.** Let $\mathcal{G}$ be the rigid CFG with start symbol σ, non-terminals σ, α and γ, rigid non-terminals R, ordering $\lhd$, terminal symbols of appropriate arity, and production rules $\sigma \to f(\alpha, \gamma, \gamma)$, $\gamma \to g(\gamma) \mid a$, and $\alpha \to \gamma$. If $\gamma \notin R$ we have, unsurprisingly, $\mathcal{L}(\mathcal{G}) = \{f(g^m(a), g^n(a), g^o(a)) \mid m, n, o \geq 0\}$. Otherwise,

1. if $\gamma \lhd \gamma \lhd \alpha$, $\mathcal{L}(\mathcal{G}) = \{f(g^m(a), g^n(a), g^n(a)) \mid m, n \geq 0\}$;
2. if $\gamma \lhd \gamma \not\lhd \alpha$, $\mathcal{L}(\mathcal{G}) = \{f(g^m(a), g^m(a), g^m(a)) \mid m \geq 0\}$;
3. if $\gamma \not\lhd \gamma \not\lhd \alpha$, $\mathcal{L}(\mathcal{G}) = \{f(a, a, a)\}$.

In contrast to the grammar presented in Example 5, acyclic tree grammars must have a finite language. If the grammar is also totally rigid its language has size essentially double exponential in the size of the grammar.

▶ **Lemma 6.** *If $\mathcal{G}$ is an acyclic totally rigid CFG then* $|\mathcal{L}(\mathcal{G})| \leq |\mathrm{Pr}_\mathcal{G}|^{2^{|N_\mathcal{G}|-1}}$.

Proof. Suppose $\mathcal{G}$ satisfies the above requirements; let $m = |\mathrm{Pr}_\mathcal{G}|$ and $n = |N_\mathcal{G}| - 1$. Since $\mathcal{G}$ is acyclic we may further assume that $\lhd_\mathcal{G}$ is irreflexive. We argue, by induction on n, that the set $\{T \in \mathrm{Terms}(\Sigma_\mathcal{G}) \mid \exists d\colon \sigma_\mathcal{G} \to T \text{ and } d \text{ is rigid}\}$ has size bounded by m^{2^n}.

The base case is $n = 0$. By the main assumption of the lemma every derivation $d\colon \sigma_\mathcal{G} \to T \in \mathrm{Terms}(\Sigma_\mathcal{G})$ has length 1, of which there are no more than m. For the induction step, suppose $n = n_0 + 1$. Let $N = N_\mathcal{G} \setminus \{a\}$ where $a \neq \sigma_\mathcal{G}$ is chosen such that $a \not\lhd_\mathcal{G} b$ for all $b \in N_\mathcal{G} \setminus \{\sigma_\mathcal{G}\}$. Suppose $d\colon \sigma_\mathcal{G} \to T \in \mathrm{Terms}(\Sigma_\mathcal{G})$ is a rigid derivation in $\mathcal{G}$. Since $\mathcal{G}$ is acyclic d can be re-ordered to have the form $d_0 d_1$ where $d_0\colon \sigma_\mathcal{G} \to S$ and $d_1\colon S \to T = S[a/S']$ for appropriate terms S and S', such that the non-terminal a is not rewritten in d_0 and not introduced by a production rule in d_1. The induction hypothesis implies there is no more than $m^{2^{n_0}}$ possibilities for each of S and S', whence there are $\leq m^{2^n}$ possibilities for T. ◀

4 Proof grammars

In this section we present rigid context-free grammars that arise from simple Π_2-proofs. We first define *elementary proof grammars* which are a natural extension of the rigid regular grammars arising from Π_1-proofs introduced in [14]. As we shall see, elementary proof grammars can have an infinite language and are not immediately suitable for producing Herbrand disjunctions. A proper form of *proof grammars* is then defined in Section 4.2 by adding further structure.

4.1 Elementary proof grammars

Let $\pi \vdash \Gamma$ be a simple Π_2-proof and Γ a set of closed Σ_1 and Π_1 formulæ. The *elementary grammar for* π is a rigid CFG, denoted $\mathcal{E}_\pi$, of the form $\langle N_\pi, R_\pi, \lhd_\pi, \Sigma_\pi, \sigma, \mathrm{Pr}_\pi\rangle$ where

- N_π consists of the eigenvariables appearing in π (of base-type) as well as a *starting symbol* σ and a symbol κ^p (of function type $\iota \to \iota$) for each position p in π at which the rule cut is applied;
- $R_\pi = \mathrm{EV}(\pi)$;
- Σ_π is the term language of first-order logic expanded by
  - a symbol $\tau_{i,F}$ of base-type for each formula $\forall x_0 \cdots x_k F \in \Gamma \cap \Pi_1$ and each $i \leq k$,
  - a symbol F of function type with k arguments for each $\exists x_0 \cdots \exists x_k F \in \Gamma \cap \Sigma_1$;
- $\lhd_\pi$ is the transitive ordering on non-terminals and Pr_π the set of production rules specified below.

Each cut occurring in π, as well as each quantified formula in the conclusion Γ, yields production rules in $\mathcal{E}_\pi$. In addition, the relative order of the quantifier introduction rules and each cut on a genuine Π_2 formula influence the rigidity ordering $\lhd_\pi$. We begin by specifying the rules induced by Γ.

For each formula $\exists x_0 \cdots \exists x_k F \in \Gamma$ with F quantifier-free, and each sequence of terms $\langle s_i\rangle_{i\leq k}$ that appear in π (together) as the witnesses of the sequence of existential quantifiers $\langle \exists x_i\rangle_{i\leq k}$, $\mathcal{E}_\pi$ features a production rule $\sigma \to \mathsf{F} s_0 \cdots s_k$. For each formula $\forall x_0 \cdots \forall x_k F \in \Gamma$ with F quantifier-free and each $i \leq k$, $\mathcal{E}_\pi$ contains the production rule $\alpha_i \to \tau_{i,F}$ where α_i is the (unique) eigenvariable for the quantifier $\forall x_i$ appearing in π. See Figure 6.

The remaining non-terminals attain their production rules based on the structure of the cut in which they are used. Let p be a position in π, A a quantifier-free formula, α, δ eigenvariables and s, t terms. Suppose $\pi|p$ has the form of either cuts given in Figure 7. In the proof on the left the two distinguished appearances of the formula $\exists y A[x/\alpha]$ as well as

$\pi \vdash \Gamma$	$\dfrac{\dfrac{\Pi', F[x_0/s_0,\dots,x_k/s_k]}{\Pi', \exists x_k F[x_0/s_0,\dots,x_{k-1}/s_{k-1}]}\exists}{\vdots}$ $\Gamma = \Pi, \exists x_0 \cdots \exists x_k F$	$\dfrac{\dfrac{\Pi', \forall x_{i+1} \cdots \forall x_k F[x_0/\alpha_0,\dots,x_i/\alpha_i]}{\Pi', \forall x_i \cdots \forall x_k F[x_0/\alpha_0,\dots,x_{i-1}/\alpha_{i-1}]}\forall}{\vdots}$ $\Gamma = \Pi, \forall x_0 \cdots \forall x_k F$
$\mathcal{E}_\pi$	$\sigma \to \mathsf{F} s_0 \cdots s_k$	$\alpha_i \to \tau_{i,F}$
$\mathcal{G}_\pi$	$\sigma \to \mathsf{F} s_0^\sigma \cdots s_k^\sigma$	$\alpha_i \to \tau_{i,F}$

Figure 6 Start and terminal production rules in $\mathcal{E}_\pi$ and $\mathcal{G}_\pi$.

Subproof $\pi\|p$	$\dfrac{\dfrac{\dfrac{\Pi'', A[x/\alpha, y/s]}{\Pi'', \exists y A[x/\alpha]}\exists}{\vdots}}{\dfrac{\Pi', \exists y A[x/\alpha]}{\Pi', \forall x \exists y A}\forall}$ $\dfrac{\dfrac{\dfrac{\Delta', \bar{A}[x/t, y/\beta]}{\vdash \Delta', \forall y \bar{A}[x/t]\ (\dagger)}\forall}{\Delta', \exists x \forall y \bar{A}}\exists}{\vdots}$ $\Delta, \exists x \forall y \bar{A}$ Π, Δ cut	$\dfrac{\dfrac{\Pi', A[x/\alpha]}{\Pi', \forall x A}\forall}{\Pi, \Delta}$ cut $\quad \dfrac{\dfrac{\Delta', \bar{A}[x/t]}{\Delta', \exists x \bar{A}}\exists}{\vdots}$ $\Delta, \exists x \bar{A}$
Rules in $\mathcal{E}_\pi$	$\alpha \to t \qquad \beta \to \kappa^p t \qquad \kappa^p \to \lambda\alpha\, s$	$\alpha \to t$
Rules in $\mathcal{G}_\pi$	$\alpha \to \lambda x_1 \cdots \lambda x_{k_\alpha}\, t^\alpha$ $\beta \to \lambda x_1 \cdots \lambda x_{k_\beta}.\ \kappa^p x_1 \cdots x_{k_\alpha} t^\beta$ $\kappa^p \to \lambda x_1 \cdots \lambda x_{k_\alpha+1}\, s^\alpha$	$\alpha \to \lambda x_1 \cdots \lambda x_{k_\alpha}\, t^\alpha$

Figure 7 Internal production rules in $\mathcal{E}_\pi$ and $\mathcal{G}_\pi$.

the two distinguished occurrences of $\exists x \forall y \bar{A}$ are assumed to be on the same trace, as are, in the right proof, the two occurrences of $\exists x \bar{A}$. The grammar includes the production rules

$$\alpha \to t \qquad \beta \to \kappa^p t \qquad \kappa^p \to \lambda\alpha\, s$$

and we set $a \lhd_\pi \alpha \lhd_\pi \kappa^p \lhd_\pi b$ for each $a \in \mathrm{EV}(\pi|q_\alpha)$ and $b \in \mathrm{EV}(\pi|q_\beta)$ where q_α and q_β are, respectively, the positions in π at which the variables α and β were eliminated (that is $p0$ and the position marked by $(\dagger)$ in Figure 7). In the scenario pictured on the right in which the cut is performed on a Π_1 formula, only a single production rule for α is needed.

▶ **Example 7** (Elementary proof grammar for π_∞). Let $\mathcal{E}_{\pi_\infty} = \langle N_\infty, R_\infty, \lhd_\infty, \Sigma_\infty, \sigma, \mathrm{Pr}_\infty \rangle$. The non-terminals of $\mathcal{E}_{\pi_\infty}$ comprise: starting symbol σ; rigid non-terminals: α, $\hat{\alpha}$, β, β', $\hat{\beta}$ and $\hat{\beta}'$; non-rigid non-terminals: κ^0 for the topmost cut on I_1 and κ $(=\kappa^{\langle\rangle})$ for the lower cut on I_0. In Σ_∞ we have the term symbols and also terminal symbols for each formula in the conclusion (note there are no universals in the conclusion of π_∞). The induced priority ordering is $\hat{\alpha} \lhd_\infty \kappa^0 \lhd_\infty \{\hat{\beta}', \hat{\beta}\} \lhd_\infty \alpha \lhd_\infty \kappa^{\langle\rangle} \lhd_\infty \{\beta', \beta\}$.

In this example we will focus only on the highlighted formula T and the corresponding symbol T of type $\iota \to \iota \to \iota$ in Σ_∞. Pr_∞ has the following rules. The production rules on the left (right) hand side are read from the cut on I_0 (I_1).

$$\sigma \to \mathsf{T}\beta\beta' \qquad\qquad \sigma \to \mathsf{T}\hat{\beta}\hat{\beta}'$$

$$\alpha \to 0 \mid \mathsf{s}\beta \qquad \beta \to \kappa 0 \qquad \hat{\alpha} \to 0 \mid \mathsf{s}\hat{\beta} \qquad \hat{\beta} \to \kappa^0 0$$

$$\kappa \to \lambda\alpha.\ \mathsf{M}\alpha\hat{\alpha} \qquad \beta' \to \kappa(\mathsf{s}\beta) \qquad \kappa^0 \to \lambda\hat{\alpha}.\ \mathsf{M}\alpha\hat{\alpha} \qquad \hat{\beta}' \to \kappa^0(\mathsf{s}\hat{\beta})$$

In general, the production rules of the elementary proof grammar $\mathcal{E}_\pi$ may be cyclic and therefore permit infinite derivations. In the case of $\mathcal{E}_{\pi_\infty}$, for example, this is demonstrated by the sequence of non-terminals $\beta, \kappa, \hat{\alpha}, \hat{\beta}, \kappa^0, \alpha, \beta$.

To avoid enumerating unnecessary terms into the language of the grammar certain derivations should be disallowed. While it is possible to provide a characterisation of the derivations that yield Herbrand sets, the work is beyond the scope of this paper and will not be presented here. Instead, in the following section, we define acyclic proof grammars for which standard (rigid) derivations suffice. This is achieved by raising the types of non-terminals to make the necessary dependencies between non-terminals explicit.

4.2 Typed proof grammars

As before, let $\pi \vdash \Gamma$ be a simple Π_2-proof and Γ a set of closed prenex Σ_1 and Π_1 formulæ. The *(proof) grammar for* π is a rigid CFG denoted by $\mathcal{G}_\pi$. $\mathcal{G}_\pi = \langle N_\pi, R_\pi, \lhd_\pi, \Sigma_\pi, \sigma, \mathrm{Pr}_\pi \rangle$ has the same definition as the elementary grammar but with two essential differences:

1. Eigenvariables are no longer necessarily base-type symbols and their type depends on their relationship to other eigenvariables;
2. Production rules are modified accordingly by type-raising operations.

Types of non-terminals in $\mathcal{G}_\pi$

The type of a given non-terminal will be determined by the relation of its position in π to other eigenvariables in π. This relation will be defined as a well-founded ordering $\prec_\pi$ on the elements of N_π and formalises the (potential) dependency of one non-terminal on another. As well as fixing the type of a non-terminal, the ordering can be seen as the basis of the priority ordering $\lhd_\pi$ used in recognising rigid derivations.

For each position p in π, if $\pi|p$ has the form

$$\dfrac{\dfrac{\Pi', A[x/\alpha]}{\Pi', \forall x A}\,\forall \qquad \Delta, \exists x \bar{A}}{\pi|p \vdash \Pi, \Delta}\,\mathrm{cut}$$

we set $\alpha \prec_\pi \kappa^p$ and $\alpha \prec_\pi a$ for every $a \in \mathrm{EV}(\pi|p0) \cup \{\kappa^q \mid p0 \leq q\}$. Notice that for $a, b \in \mathrm{EV}(\pi)$, $a \prec_\pi b$ implies $b \lhd_\pi a$, and if we write $\prec_a$ for the set $\{b \in N_\pi \mid b \prec_\pi a\}$, $\prec_a$ is a set of eigenvariables linearly ordered by $\prec_\pi$. Moreover, for α appearing as above $\prec_{\kappa^p} = \prec_\alpha \cup \{\alpha\}$.

The type of a non-terminal is chosen to be its order-type in $\prec_\pi$. Let $k_a = |\prec_a|$. The *type* of $a \in N_\pi$ is that of a function over the base-type taking k_a arguments. So in particular σ is of base-type, as is any eigenvariable relating to a universal quantifier in the conclusion.

▶ **Example 8** (Types of non-terminals in π_∞)**.** In $\mathcal{G}_{\pi_\infty}$ we have the same set of non-terminals symbols as in $\mathcal{E}_{\pi_\infty}$ but they are now assigned the following types. The ordering $\prec_\infty$ on N_∞ gives $\prec_\infty = \{(\alpha, \hat{\alpha}), (\hat{\alpha}, \kappa^0), (\alpha, \kappa^0), (\alpha, \hat{\beta}), (\alpha, \hat{\beta}'), (\alpha, \kappa^{\langle\rangle})\}$ so α, β, β' and σ all have base-type, $\hat{\alpha}$, $\hat{\beta}$, $\hat{\beta}'$ and $\kappa^{\langle\rangle}$ have type $\iota \to \iota$ and κ^0 has type $\iota \to \iota \to \iota$. Note, the ordering $\lhd_\infty$ specifying rigidity is unchanged from $\mathcal{E}_{\pi_\infty}$.

Production rules in $\mathcal{G}_\pi$

As mentioned earlier, the production rules of $\mathcal{G}_\pi$ have the form as in $\mathcal{E}_\pi$. We now explain how the change in the type of non-terminals is to be taken into account. This is achieved by means of a type-lifting operation that either lifts an occurrence of a non-terminal to its appropriate type or replaces it with a variable symbol in case it should be abstracted.

For each $\gamma \in \mathrm{EV}(\pi)$, let $\gamma_1 \prec_\pi \cdots \prec_\pi \gamma_{k_\gamma+1} = \gamma$ be the sequence of eigenvariables enumerating the elements of $\{\xi \mid \xi \preceq_\pi \gamma\}$. Also let $\{x_i\}$ be a fresh set of variable symbols.

Given $\alpha \in \mathrm{EV}(\pi) \cup \{\sigma\}$, we define an operation $S \mapsto S^\alpha$ on terms in $\Sigma_\pi \cup N_\pi$ that lifts the occurrence of non-terminals to their required type: we set $(ST)^\alpha = S^\alpha T^\alpha$, $(\lambda y.S)^\alpha = \lambda y.S^\alpha$, $c^\alpha = c$ for $c \in \Sigma_\pi$, and for $\gamma \in \mathrm{EV}(\pi) \cup \{\kappa^q \mid q \text{ a position in } \pi\}$,

$$\gamma^\alpha = \begin{cases} x_i, & \text{if } \gamma = \alpha_i \preceq_\pi \alpha, \\ \gamma\gamma_1^\alpha\gamma_2^\alpha \cdots \gamma_{k_\gamma}^\alpha, & \text{otherwise.} \end{cases}$$

The operation $S \mapsto S^\alpha$ replaces each non-terminal $\alpha_i \preceq_\pi \alpha$ by the variable x_i and to $\gamma \not\preceq_\pi \alpha$ makes explicit the dependence of γ on $\prec_\gamma$. For example, we have

$$\begin{aligned} \gamma^{\gamma_{k_\gamma}} &= \gamma x_1 x_2 \cdots x_{k_\gamma} \\ \gamma^\sigma &= \gamma\gamma_1\big(\gamma_2\gamma_1\big)\big(\gamma_3\gamma_1(\gamma_2\gamma_1)\big) \cdots \big(\gamma_{k_\gamma}\gamma_1(\gamma_2\gamma_1) \cdots (\gamma_{k_\gamma-1}\gamma_1 \cdots (\gamma_{k_\gamma-2}\gamma_1 \cdots))\big) \end{aligned}$$

Let p be a position in π, suppose $\pi|p$ has the form of either cuts in Figure 7. Then the grammar $\mathcal{G}_\pi$ includes the production rules

$$\alpha \to \lambda x_1 \cdots \lambda x_{k_\alpha}\, t^\alpha \qquad \beta \to \lambda x_1 \cdots \lambda x_{k_\beta}.\, \kappa^p x_1 \cdots x_{k_\alpha} t^\beta \qquad \kappa^p \to \lambda x_1 \cdots \lambda x_{k_\alpha+1}\, s^\alpha$$

Observe that $k_\alpha \leq k_\beta$, so $\alpha_i = \beta_i$ for each $0 < i \leq k_\alpha$ and the term in the central production rule is indeed closed. Also for α and κ^p above notice $k_{\kappa^p} = k_\alpha + 1$, hence the production rule for κ^p is well-typed. In the scenario pictured on the right side of Figure 7 in which the cut is performed on a Π_1 formula, naturally only the production rule for α is required.

Concerning the start symbol and formulæ in the conclusion Γ we add, in analogy to $\mathcal{E}_\pi$, production rules $\sigma \to \mathsf{F} s_0^\sigma \cdots s_k^\sigma$ and $\alpha_i \to \tau_{i,F}$ for appropriate formulæ F, terms $\langle s_i \rangle_{i \leq k}$ and eigenvariable α_i. This completes the definition of $\mathcal{G}_\pi$.

▶ **Example 9** (Proof grammar for π_∞)**.** It is now possible to complete the definition of $\mathcal{G}_{\pi_\infty}$. As in Example 7 we will focus on parts of the grammar that are relevant to the formula T and the computation of its language, which we denote $\mathcal{L}(\mathcal{G}_\infty, T)$. The non-terminals and their types were described in the previous example. Notice for instance $\xi^\sigma = \xi$ for $\xi \in \{\alpha, \beta, \beta'\}$, $\xi^\sigma = \xi\alpha$ and $\xi^\alpha = \xi x_1$ for $\xi \in \{\kappa, \hat{\alpha}, \hat{\beta}, \hat{\beta}'\}$, and $(\mathsf{M}\alpha\hat{\alpha})^{\hat{\alpha}} = \mathsf{M}x_1x_2$. The production rules of the grammar (relating to the formula T) are therefore

$$\begin{array}{llll} \sigma \to \mathsf{T}\beta\beta' & & \sigma \to \mathsf{T}(\hat{\beta}\alpha)(\hat{\beta}'\alpha) & \\ \alpha \to 0 \mid \mathsf{s}\beta & \beta \to \kappa 0 & \hat{\alpha} \to \lambda x.\ 0 \mid \lambda x.\ \mathsf{s}(\hat{\beta}x) & \hat{\beta} \to \lambda x.\ \kappa^0 x 0 \\ \kappa \to \lambda x.\ \mathsf{M}x(\hat{\alpha}x) & \beta' \to \kappa(\mathsf{s}\beta) & \kappa^0 \to \lambda x\lambda y.\ \mathsf{M}xy & \hat{\beta}' \to \lambda x.\ \kappa^0 x(\mathsf{s}(\hat{\beta}x)) \end{array}$$

The computation of the language of $\mathcal{L}(\mathcal{G}_\infty, T)$ is not complicated. Nevertheless, for space considerations it is necessary to make a few simplifications. In particular, in accordance with the informal proof presented in Section 2, various terms will be evaluated according to the intended semantics, so $\mathsf{s}S$ will be written as $S+1$ and $\mathsf{M}ST$ will be replaced by $\max\{S,T\}$. Also, $\mathsf{T}ST$ will be presented as $\mathsf{T}(S,T)$. In addition, implicit β-conversion of terms will be performed as this has no effect on rigidity of the considered derivations.

We begin by calculating the terms derivable from $\hat{\beta}$ and $\hat{\beta}'$ (with implicit β-conversion):

$$\hat{\beta}x \to \kappa^0 x 0 \to \mathsf{M}x0 = x \qquad \hat{\beta}'x \to^* \mathsf{M}x(\mathsf{s}(\hat{\beta}x)) \to^* \mathsf{M}x(\mathsf{s}(\mathsf{M}x0)) = x+1$$

Regarding $\hat{\alpha}$, modulo β-conversion it is easy to see we have $\hat{\alpha}x \to^* 0 \mid x+1$. Thus we can also compute the derivations starting from β, β' and α:

$$\begin{array}{ll} \beta \to^* \mathsf{M}0(\hat{\alpha}0) \to^* 0 \mid 1 & \alpha \to^* 0 \mid 1 \mid 2 \\ \beta' \to^* \mathsf{M}(\mathsf{s}\beta)(\hat{\alpha}(\mathsf{s}\beta)) \to \beta+1 \mid \beta+2 & \end{array}$$

Thus $\mathsf{T}(\beta,\beta{+}1)$ and $\mathsf{T}(\beta,\beta{+}2)$ are the two terms derivable from $\mathsf{T}(\beta,\beta')$ without rewriting β. Combining these with the terms obtained from β above yields a total of eight derivations in the underlying non-rigid grammar. However, as $\kappa,\beta',\hat{\alpha}$ are the sole non-terminals used in deriving β and $\beta \not\lhd \kappa,\beta',\hat{\alpha}$, only four of the derivations are rigid, leaving

$$\sigma \to \mathsf{T}(\beta,\beta') \to^* \mathsf{T}(0,1) \mid \mathsf{T}(1,2) \mid \mathsf{T}(0,2) \mid \mathsf{T}(1,3)$$

Concerning the terms derivable from σ via $\mathsf{T}(\hat{\beta}\alpha,\hat{\beta}'\alpha)$, rigid derivations yield

$$\sigma \to \mathsf{T}(\hat{\beta}\alpha,\hat{\beta}'\alpha) \to^* \mathsf{T}(0,1) \mid \mathsf{T}(1,2) \mid \mathsf{T}(2,3)$$

Thus we conclude $\mathcal{L}(\mathcal{G}_\infty,T) = \{\mathsf{T}(0,1),\mathsf{T}(1,2),\mathsf{T}(2,3),\mathsf{T}(0,2),\mathsf{T}(1,3)\}$. The reader may check that $\Gamma \cup \Delta \cup \{T_{m,n} \mid \mathsf{T}(m,n) \in \mathcal{L}(\mathcal{G}_\infty,T)\}$ is derivable.

▶ **Theorem 10** (Language bound). *Let π be a simple Π_2-proof. The number of production rules in $\mathcal{G}_\pi$ is bounded by the number of quantifier inferences in π and $|\mathcal{L}(\mathcal{G}_\pi)| < 2^{2^{|\pi|}}$.*

Proof. Let $\mathcal{G}'{=}\langle N_\pi, N_\pi, \lhd_\pi, \Sigma_\pi, \sigma, \mathrm{Pr}_\pi\rangle$ be the modification of $\mathcal{G}_\pi$ in which all non-terminals are marked as rigid. We observe $\mathcal{L}(\mathcal{G}') = \mathcal{L}(\mathcal{G}_\pi)$. Moreover, a study of paths in π reveals that $\mathcal{G}'$ is acyclic, whereby $|\mathcal{L}(\mathcal{G}_\pi)| \leq |\mathrm{Pr}_\pi|^{2^{|N_\pi|}}$ by Lemma 6. Let $k = \lfloor \frac{|\mathrm{Pr}_\pi|}{2} \rfloor$. Then $N_\pi + k < |\pi|$ and so $|\mathcal{L}(\mathcal{G}_\pi)| \leq 2^{2^{|N_\pi|+k}} < 2^{2^{|\pi|}}$ as required. ◀

5 Language containment

▶ **Lemma 11** (Local reduction). *If $\pi \rightsquigarrow \pi'$ is a local one-step cut reduction between regular simple Π_2-proofs then $\mathcal{L}(\mathcal{G}_{\pi'}) \subseteq \mathcal{L}(\mathcal{G}_\pi)$.*

Proof. We present the argument for two of the interesting cases, the case of Contraction Reduction and Quantifier Reduction; the remaining cases follow by a simple argument mirroring that of the former case.

Suppose, to begin, that the reduction $\pi \rightsquigarrow \pi'$ is an instance of Contraction Reduction and that the cut formula is principal in the contraction. Thus π and π' can be assumed to take the form given below, where $A = \forall x B$ is Π_2 and π_0^* is a renaming of π_0 so that π' is regular.

$$\dfrac{\dfrac{\pi_0 \quad \Gamma, A}{\pi \vdash \Gamma, \Delta', \Delta}\,\mathsf{cut} \qquad \dfrac{\pi_1 \quad \Delta', \Delta, \bar{A}, \Delta^*, \bar{A}^*}{\Delta', \Delta, \bar{A}}\,\mathsf{c}}{\pi \vdash \Gamma, \Delta', \Delta}\,\mathsf{cut} \qquad\qquad \dfrac{\dfrac{\dfrac{\pi_0^* \quad \Gamma^*, A^*}{\Gamma, \Gamma^*, \Delta', \Delta, \Delta^*}\,\mathsf{cut} \qquad \dfrac{\pi_0 \quad \Gamma, A \qquad \pi_1 \quad \Delta', \Delta, \bar{A}, \Delta^*, \bar{A}^*}{\Gamma, \Delta', \Delta, \Delta^*, \bar{A}^*}\,\mathsf{cut}}{\Gamma, \Gamma^*, \Delta', \Delta, \Delta^*}\,\mathsf{cut}}{\pi' \vdash \Gamma, \Delta', \Delta}\,\mathsf{c}$$

Note that we may assume the contraction occurs in the 'right' sub-proof as π is simple and A is a universal formula.

We argue that every rigid derivation in $\mathcal{G}_{\pi'}$ starting from σ can be transformed into a rigid derivation in $\mathcal{G}_\pi$ beginning at σ. Consider the function $f\colon N_{\pi'} \to N_\pi$ defined by

$$\begin{array}{lll} f(\sigma) = \sigma & & f(\kappa^0) = f(\kappa^{01}) = \kappa^{\langle\rangle} \\ f(\gamma) = f(\gamma^*) = \gamma & \text{for } \gamma \in \mathrm{EV}(\pi_0) & f(\kappa^{00p}) = f(\kappa^{010p}) = \kappa^{0p} \\ f(\delta) = \delta & \text{for } \delta \in \mathrm{EV}(\pi_1) & f(\kappa^{011p}) = \kappa^{10p} \end{array}$$

We observe that for all $a, b \in N_{\pi'}$ we have $a \prec_{\pi'} b$ iff $f(a) \prec_\pi f(b)$, thus f is type-preserving and uniquely extends to a function mapping terms in the language of $\Sigma_{\pi'} \cup N_{\pi'}$ to (well-typed) terms in $\Sigma_\pi \cup N_\pi$. Moreover, if $a \to S$ is a production rule in $\mathcal{G}_{\pi'}$, $f(a) \to f(S)$ is a production rule in $\mathcal{G}_\pi$. So f transforms derivations in the former grammar to derivations in the latter grammar; all that remains is to check the operation preserves rigidity.

Rigidity is not immediate as f is not injective. Indeed, let $d = \langle (a_i \to S_i), p_i \rangle_{i<\mathrm{lh}(d)} \colon \sigma \to S$ be a rigid derivation in $\mathcal{G}_{\pi'}$, let $d^f \colon \sigma \to f(S)$ be the derivation in $\mathcal{G}_\pi$ induced by f and suppose $j_0, j_1 < \mathrm{lh}(d)$ are such that $j_0 \not\sim_d j_1$ but $j_0 \sim_{d^f} j_1$, so $f(a_{j_0}) = f(a_{j_1}) \in R_{\mathcal{G}_\pi}$. Since f preserves the priority ordering, it follows that $a_{j_0} \neq a_{j_1}$ and so we may assume $a_{j_0} \in \mathrm{EV}(\pi_0)$ and $a_{j_1} \in \mathrm{EV}(\pi_0^*)$. But then d must utilise a production rule for a rigid non-terminal $\delta \in \mathrm{EV}(\pi_1)$ at a position q such that i) $q < p_{j_0}$ iff $q \not< p_{j_1}$ and ii) either $a_{j_0} \lhd \delta$ or $a_{j_1} \lhd \delta$, contradicting $j_0 \sim_{d^f} j_1$.

Before proceeding with the second case, we remark that in all the other local reduction steps, the natural choice of the 'renaming' function f is injective and preservation of rigidity is immediate.

Suppose now $\pi \rightsquigarrow \pi'$ is an instance of Quantifier Reduction. We may assume π and π' have the form below.

$$\dfrac{\dfrac{\overset{\pi_0}{\Gamma, A[x/\alpha]}}{\Gamma, \forall x\, A}\,\forall \qquad \dfrac{\overset{\pi_1}{\Delta, \bar{A}[x/s]}}{\Delta, \exists x\, \bar{A}}\,\exists}{\pi \vdash \Gamma, \Delta}\,\mathsf{cut} \qquad\qquad \dfrac{\overset{\pi_0[\alpha/s]}{\Gamma, A[x/s]} \qquad \overset{\pi_1}{\Delta, \bar{A}[x/s]}}{\pi' \vdash \Gamma, \Delta}\,\mathsf{cut}$$

As in the previous case we define a function mapping rigid derivations in $\mathcal{G}_{\pi'}$ to rigid derivations in $\mathcal{G}_\pi$. Although every rigid non-terminal of $\mathcal{G}_\pi$ is a non-terminal in $\mathcal{G}_{\pi'}$, the non-terminals arising from $\mathrm{EV}(\pi_0)$ have a higher type than their counterpart in π' as we have $\alpha \prec_\pi \xi$ for every $\xi \in \mathrm{EV}(\pi_0)$. Notice, however, $k_\alpha = 0$ and $k_{\kappa\langle\rangle} = 1$ in $\mathcal{G}_\pi$.

Let $d \colon \sigma \to S \in \mathcal{L}(\mathcal{G}_{\pi'})$ be a rigid derivation in $\mathcal{G}_{\pi'}$. As the formula $A[x/s]$ is Σ_1 it follows that d has one of following two forms.

In the first case, d is (up to simple renaming of non-terminals) a derivation in $\pi_0[\alpha/s]$ and $S \in \mathcal{L}(\mathcal{G}_{\pi_0[\alpha/s]})$. d induces a rigid derivation in $\mathcal{G}_{\pi_0}$ which when augmented by the production rule $\alpha \to s$ (present in $\mathcal{G}_\pi$) provides a derivation of S in $\mathcal{G}_\pi$.

In the second case, (a permutation of) d has the form $d_0 d_1 d_2$ where $d_0 \colon \sigma \to S'$ is a derivation in $\mathcal{G}_{\pi_1}$; $d_1 \colon S' \to S'[\beta/t^\beta]$ is a derivation using the single production rule $\beta \to t^\beta$ (note $k_\beta = 0$ in both $\mathcal{G}_{\pi'}$ and $\mathcal{G}_\pi$) where $\beta \in \mathrm{EV}(\pi_1)$ is the unique eigenvariable for the universal quantifier in $\bar{A}[x/s]$ (if there is one) in $\mathcal{G}_{\pi'}$; and $d_2 \colon S'[\beta/t^\beta] \to S$ is a derivation in $\mathcal{G}_{\pi_0[\alpha/s]}$. We observe that the production rule $\beta \to t^\beta$ becomes, in $\mathcal{G}_\pi$, the derivation $\beta \to \kappa^{\langle\rangle} s \to (\lambda x_1 u^\alpha) s$ where $u[\alpha/s] = t$. Using these derivations in place of d_1 and making the appropriate modifications to the derivation d_2 yields a derivation $e \colon \sigma \to S''$ in $\mathcal{G}_\pi$ such that S'' β-reduces to S. In both cases rigidity is also easily checked. ◀

Using the previous lemma it is not difficult to establish that the language of proof grammars respect also non-local cut reductions, provided simplicity is maintained.

▶ **Theorem 12** (Global Reduction). *Suppose π and π' are regular simple Π_2-proofs such that π differs from π' only in its subproof at position p. If $\pi'|p \rightsquigarrow \pi|p$ then $\mathcal{L}(\mathcal{G}_\pi) \subseteq \mathcal{L}(\mathcal{G}_{\pi'})$.*

We now re-state and prove our main results from the introduction. The first of these is a restatement of Theorem 2 and follows from Theorem 12; the second is a generalisation of Theorem 1.

▶ **Theorem 13.** *Let $\langle\pi\rangle_{i\leq k}$ be a sequence of simple Π_2-proofs such that for each $i < k$, π_{i+1} is obtained by application of one of the reduction rules of Figure 4 to a sub-proof of π_i. Then for every term $T \in \mathcal{L}(\mathcal{G}_{\pi_k})$ there exists a term $S \in \mathcal{L}(\mathcal{G}_{\pi_0})$ that β-reduces to T.*

▶ **Theorem 14.** *Let $\pi \vdash \Gamma, \Delta$ be a Π_2-proof where $\Gamma \subseteq \Pi_1$ and $\Delta \subseteq \Sigma_1$ are sets of prenex formulæ and suppose $|\pi|$ denotes the number of inference rules occurring in π. There exists a totally rigid acyclic context-free tree grammar $\mathcal{G}$ such that i) $|\mathrm{Pr}_{\mathcal{G}}| \leq |\pi|$, ii) $|\mathcal{L}(\mathcal{G})| \leq 2^{2^{|\pi|}}$ and iii) there is a quantifier-free form of Γ, Γ', such that the formula*

$$\bigvee \Gamma' \vee \bigvee \{F[x_1/s_1, \ldots, x_k/s_k] \mid (\exists x_1 \cdots \exists x_k F) \in \Delta \wedge \mathsf{F} s_1 \cdots s_k \in \mathcal{L}(\mathcal{G})\} \tag{1}$$

is a tautology.

Proof. By applying quantifiers inversion if necessary, π can be turned into a simple Π_2-proof π' without an increase in size. Let $\mathcal{G} = \langle N_{\pi'}, N_{\pi'}, \lhd_{\pi'}, \Sigma_{\pi'}, \sigma, \mathrm{Pr}_{\pi'}\rangle$ be the totally rigid grammar derived from $\mathcal{G}_{\pi'}$. Items (i) and (ii) follow from Theorem 10. Regarding (iii), let $\langle\pi_i\rangle_{i\leq k}$ be a reduction sequence of simple Π_2-proofs (for example any obtained from the standard cut elimination algorithms of [31, 32]) starting from $\pi_0 = \pi'$ and leading to a cut-free proof π_k of Γ, Δ. By the previous theorem, $\mathcal{L}(\mathcal{G}_{\pi_k}) \subseteq \mathcal{L}(\mathcal{G}_{\pi})$.

Suppose Γ and Δ have the forms $\forall x_1 \cdots \forall x_{k_0} G_0, \ldots, \forall x_1 \cdots \forall x_{k_m} G_m$ and $\exists x_1 \cdots \exists x_{l_0} F_0$, $\ldots$, $\exists x_1 \cdots \exists x_{l_n} F_n$ respectively where $G_0, \ldots, G_m, F_0 \ldots, F_n$ are quantifier-free. The Herbrand disjunction read from π_k is the formula

$$X = \bigvee_{j\leq m} G_j(\alpha_j^1, \ldots, \alpha_j^{k_j}) \vee \bigvee_{j\leq n} \bigvee_{(s_1,\ldots,s_{l_j})\in\mathcal{H}(\pi_k,F_j)} F_j(s_1, \ldots, s_{l_j})$$

for appropriate choice of variables α_j^i. Since the formula in (1) is the result of replacing every α_j^i by τ_{i,G_j} in X we are done. ◀

6 Conclusion

This work provides an abstraction of proofs which focuses only on the aspects relevant to the extraction of Herbrand disjunctions. Compared to other approaches in the literature, including Herbrand nets [24], proof forests [10], expansion trees with cut [19] and functional interpretation [9], proof grammars offer a representation of Herbrand's theorem suitable for the following exploitation.

As carried out in [18], the result for Π_1-proofs can be strengthened to the following: let π' be any cut-free proof obtained from a Π_1-proof π via standard cut elimination, then $\mathcal{H}(\pi') \subseteq \mathcal{L}(\mathcal{G}_{\pi})$. Moreover, if π' is obtained from π by non-erasing reduction (which corresponds to the λI-calculus, see [4, Section 9]) then we even have $\mathcal{H}(\pi') = \mathcal{L}(\mathcal{G}_{\pi})$. Therefore all (possibly infinitely many) normal forms of the non-erasing reduction have the same Herbrand disjunction. This property of classical logic has been called *Herbrand-confluence* in [18] and provides a general way of defining the computational content of a classical proof in the sense that no witness is ruled out by a choice of reduction strategy. A deeper analysis of Π_2-proofs carried out in [1] has also yielded a Herbrand-confluence result analogous to [18].

A notable application of proof grammars is in *cut introduction*, which is motivated by the aim to structure and compress automatically generated analytic proofs. As shown in [17, 16], the arrows of Figure 2 can be inverted in the sense that a grammar can be computed from a Herbrand disjunction and that from such a grammar one can compute cut formulæ which realise the compression of the grammar. This method has been implemented and empirically

evaluated with good results in [15]. An extension of these techniques to the case of proofs with Π_1-induction has led to a new technique for inductive theorem proving [8]. A natural continuation of the present work is to find an analogous characterisation for proofs with Π_2-induction as well as the development of techniques for the systematic introduction of Π_2 cuts.

An application of proof grammars to proof complexity consists in proving a lower bound on the length of proofs with cuts (which is notoriously difficult to control) by transferring a lower bound on the size of the corresponding grammar. This has been carried out for Π_1 cuts in [7] based on [14], and can potentially be extended to Π_2 cuts based on the results of this paper.

Acknowledgements. The authors' research was supported by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF) project VRG12-04. The authors also wish to thank the anonymous referees for their helpful suggestions.

References

1 Bahareh Afshari, Stefan Hetzl, and Graham E. Leigh. On Herbrand confluence for first-order logic. Submitted, 2015.

2 Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Cut-Elimination: Experiments with CERES. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 481–495. Springer, 2005.

3 Franco Barbanera, Stefano Berardi, and Massimo Schivalocchi. "Classical" programming-with-proofs in λ_{PA}^{Sym}: An analysis of non-confluence. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 365–390. Springer Berlin Heidelberg, 1997.

4 Hendrik Pieter Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1984.

5 Samuel R. Buss. On Herbrand's Theorem. In *Logic and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 1995.

6 Thierry Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60(1):325–337, 1995.

7 Sebastian Eberhard and Stefan Hetzl. Compressibility of finite languages by grammars. preprint, available at `http://www.logic.at/people/hetzl/research/`, 2015.

8 Sebastian Eberhard and Stefan Hetzl. Inductive theorem proving based on tree grammars. to appear in the Annals of Pure and Applied Logic, preprint available at `http://www.logic.at/people/hetzl/research/`, 2015.

9 Philipp Gerhardy and Ulrich Kohlenbach. Extracting Herbrand Disjunctions by Functional Interpretation. *Archive for Mathematical Logic*, 44:633–644, 2005.

10 Willem Heijltjes. Classical proof forestry. *Annals of Pure and Applied Logic*, 161(11):1346–1366, 2010.

11 Hugo Herbelin. *Séquents qu'on calcule*. PhD thesis, Université Paris 7, 1995.

12 Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris, 1930.

13 Stefan Hetzl. On the form of witness terms. *Archive for Mathematical Logic*, 49(5):529–554, 2010.

14 Stefan Hetzl. Applying Tree Languages in Proof Theory. In Adrian-Horia Dediu and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications (LATA) 2012*, volume 7183 of *Lecture Notes in Computer Science*, pages 301–312. Springer, 2012.

15 Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai, and Daniel Weller. Introducing Quantified Cuts in Logic with Equality. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR*, volume 8562 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2014.

16 Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. Algorithmic introduction of quantified cuts. *Theoretical Computer Science*, 549:1–16, 2014.

17 Stefan Hetzl, Alexander Leitsch, and Daniel Weller. Towards Algorithmic Cut-Introduction. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR-18)*, volume 7180 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2012.

18 Stefan Hetzl and Lutz Straßburger. Herbrand-Confluence for Cut-Elimination in Classical First-Order Logic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL) 2012*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 320–334. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.

19 Stefan Hetzl and Daniel Weller. Expansion trees with cut. preprint, available at `http://arxiv.org/abs/1308.0428`, 2013.

20 David Hilbert and Paul Bernays. *Grundlagen der Mathematik II*. Springer, 1939.

21 Florent Jacquemard, Francis Klay, and Camille Vacher. Rigid tree automata. In Adrian Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications (LATA) 2009*, volume 5457 of *Lecture Notes in Computer Science*, pages 446–457. Springer, 2009.

22 Florent Jacquemard, Francis Klay, and Camille Vacher. Rigid tree automata and applications. *Information and Computation*, 209:486–512, 2011.

23 Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

24 Richard McKinley. Proof nets for Herbrand's Theorem. *ACM Transactions on Computational Logic*, 14(1):5:1–5:31, 2013.

25 Dale Miller. A Compact Representation of Proofs. *Studia Logica*, 46(4):347–370, 1987.

26 V.P. Orevkov. Lower bounds for increasing complexity of derivations after cut elimination. *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta*, 88:137–161, 1979.

27 Pavel Pudlák. The Lengths of Proofs. In Sam Buss, editor, *Handbook of Proof Theory*, pages 547–637. Elsevier, 1998.

28 Diana Ratiu and Trifon Trifonov. Exploring the Computational Content of the Infinite Pigeonhole Principle. *Journal of Logic and Computation*, 22(2):329–350, 2012.

29 Monika Seisenberger. *On the Constructive Content of Proofs*. PhD thesis, Ludwig-Maximilians-Universität München, 2003.

30 Richard Statman. Lower bounds on Herbrand's theorem. *Proceedings of the American Mathematical Society*, 75:104–107, 1979.

31 G. Takeuti. *Proof Theory*. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2013.

32 Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.

33 Christian Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, 2000.

Non-Wellfounded Trees in Homotopy Type Theory*

Benedikt Ahrens[1], Paolo Capriotti[2], and Régis Spadotti[1]

1 **Institut de Recherche en Informatique de Toulouse**
Université Paul Sabatier, Toulouse, France
{ahrens,spadotti}@irit.fr

2 **Functional Programming Laboratory, School of Computer Science**
University of Nottingham, UK
pvc@cs.nott.ac.uk

Abstract

We prove a conjecture about the constructibility of *coinductive types* – in the principled form of *indexed* M*-types* – in Homotopy Type Theory. The conjecture says that in the presence of *inductive* types, coinductive types are derivable. Indeed, in this work, we construct coinductive types in a subsystem of Homotopy Type Theory; this subsystem is given by Intensional Martin-Löf type theory with natural numbers and Voevodsky's Univalence Axiom. Our results are mechanized in the computer proof assistant Agda.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Homotopy Type Theory, coinductive types, computer theorem proving, Agda

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.17

1 Introduction

Coinductive data types are used in functional programming to represent infinite data structures. Examples include the ubiquitous data type of streams over a given base type, but also more sophisticated types; as an example we present an alternative definition of equivalence of types (Example 22).

From a categorical perspective, coinductive types are characterized by a *universal property*, which specifies the object with that property *uniquely* in a suitable sense. More precisely, a coinductive type is specified as the *terminal coalgebra* of a suitable endofunctor. In this category-theoretic viewpoint, coinductive types are dual to *inductive* types, which are defined as initial algebras.

Inductive, resp. coinductive, types are usually considered in the principled form of the family of W-types, resp. M-types, parametrized by a type A and a dependent type family B over A, that is, a family of types $(B(a))_{a:A}$. Intuitively, the elements of the coinductive type $\mathsf{M}(A, B)$ are trees with nodes labeled by elements of A such that a node labeled by $a : A$ has $B(a)$-many subtrees, given by a map $B(a) \to \mathsf{M}(A, B)$; see Figure 1 for an example. The *inductive* type $\mathsf{W}(A, B)$ contains only trees where any path within that tree eventually leads to a *leaf*, that is, to a node $a : A$ such that $B(a)$ is empty.

* The work of Benedikt Ahrens was partially supported by the CIMI (Centre International de Mathématiques et d'Informatique) Excellence program ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02 during a postdoctoral fellowship.

$a, b, c : A$
$B(a) = 0$
$B(b) = 2$
$B(c) = 3$

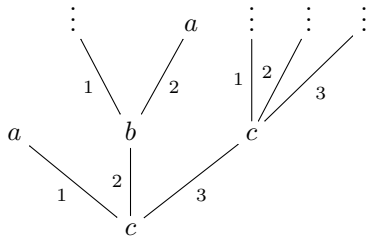


Figure 1 Example of a tree (adapted from [14]).

In this work, we study coinductive types in Homotopy Type Theory (HoTT), an extension of intensional Martin-Löf type theory [11]; we give a brief overview in Section 2.

The universal properties defining inductive and coinductive types, respectively, can be expressed internally to intensional Martin-Löf type theory (and thus internally to HoTT). Awodey, Gambino, and Sojakova [6] use this facility when proving, within a subtheory $\mathcal{H}$ of HoTT, a logical equivalence between

1. the existence of W-types (a.k.a. the existence of a universal object) and
2. the addition of a set of type-theoretic rules to their "base theory" $\mathcal{H}$.

We might call the W-types defined internally "internal W-types", and those specified via type-theoretic rules "external" ones. In that sense, Awodey, Gambino, and Sojakova [6] prove a logical equivalence between the existence of internal and external W-types.

The universal property defining (internal) coinductive types in HoTT is dual to the one defining (internal) inductive types. One might hence assume that their existence is equivalent to a set of type-theoretic rules dual (in a suitable sense) to those given for external W-types as in Item 2 above. However, the rules for external W-types cannot be dualized in a naïve way, due to some asymmetry of HoTT related to dependent types as maps into a "type of types" (a *universe*), see the discussion in [10].

In this work, we show instead that coinductive types in the form of M-types can be derived from certain inductive types. (More precisely, only one specific W-type is needed: the type of natural numbers, which is readily specified as a W-type [6].)

The result presented in this work is not surprising; indeed, the constructibility of coinductive types from inductive types has been shown in extensional type theory (see Section 1.1) and was conjectured to work in HoTT during a discussion on the HoTT mailing list [10]. In this work, we give a formal proof of the constructibility of a class of coinductive types from inductive types, with a proof of correctness of the construction.

The theorem we prove here is actually more general than described above: instead of plain M-types as described above, we construct *indexed* M-types, which can be considered as a form of "(simply-)typed" trees, typed over a type of indices I. Plain M-types then correspond to the mono-typed indexed M-types, that is, to those for which $I = 1$. Since all the ideas are already contained in the case of plain M-types, we describe the construction of those extensively, and only briefly state the definitions and the main result for the indexed case. The formalisation in Agda, however, is done for the more general, indexed, case. An example illustrates the need for these more general *indexed* M-types.

1.1 Related work

Inductive types in the form of W-types in HoTT have been studied by Awodey, Gambino, and Sojakova [6]. The content of that work is described above.

Van den Berg and De Marchi [14] study the existence of *plain* M-types in models of *extensional* type theory, that is, of type theory with a reflection rule identifying propositional and judgmental equality. They prove the derivability of M-types from W-types in such models, see Corollary 2.5 of the arXiv version of that article. A construction in extensional type theory of M-types from W-types is given by Abbott, Altenkirch, and Ghani [1].

Martin-Löf type theory without identity reflection, but with the principle of *Uniqueness of Identity Proofs* (Axiom K) can be identified with the 0-truncated fragment of HoTT (modulo the assumption of univalence and HITs). For such a type theory, a construction of (indexed) M-types from W-types is described by Altenkirch et al. [4], internalizing a standard result in 1-category theory [7]. The present work thus generalizes the construction described in [4] by extending it from the 0-truncated fragment to the whole of HoTT. More specifically, the main work in this generalization is to develop higher-categorical variants of the 1-categorical constructions used in [4] that are compatible with the higher-categorical structure (the coherence data) of types.

1.2 Synopsis

The paper is organized as follows: In Section 2 we present the type theory we are working in – a "subsystem" of HoTT as presented in [13]. In Section 3 we define signatures for *plain* M-types and, via a universal property, the M-type associated to a given signature. In Section 4 we construct the M-type of a given signature. In Section 5 we state the main result for the case of *indexed* M-types. Finally, in Section 6 we give an overview of the formalisation of our result in the proof assistant Agda.

2 The type theory under consideration

The present work takes place within a type theory that is a subsystem of the type theory presented in the HoTT book [13]. The latter is often referred to as Homotopy Type Theory (HoTT); it is an extension of intensional Martin-Löf type theory (IMLTT) [11]. The extension is given by two data: firstly, the *Univalence Axiom*, introduced by Vladimir Voevodsky and proven consistent with IMLTT in the simplicial set model [9]. The second extension is given by *Higher Inductive Types* (HITs), the precise theory of which is still subject to active research. Preliminary results on HITs have been worked out by Sojakova [12] and Lumsdaine and Shulman – see [13, Chap. 6] for an introduction. In the present work, we use the Univalence Axiom, but do not make use of HITs.

The syntax of HoTT is extensively described in a book [13]; we only give a brief summary of the type constructors used in the present work, thus fixing notation. The fundamental objects are *types*, which have *elements* ("inhabitants"), written $a : A$. Types can be dependent on terms, which we write as $x : A \vdash B(x) : \mathcal{U}$. In the preceding judgment, we use a special type $\mathcal{U}$, the "universe" or "type of types". In this work we assume any type being an element of $\mathcal{U}$ in the sense of "typical ambiguity" [13, Chap. 1.3], without worrying about universe levels. The formalisation in Agda ensures that everything works fine in that respect: as we will see later, the universe $\mathcal{U}$ is closed under the construction of M-types.

We use the following type constructors: dependent products $\prod_{(x:A)} B(x)$, with non-dependent variant written $A \to B$, dependent sums $\sum_{(x:A)} B(x)$ with non-dependent variant

written $A \times B$, the identity type $x =_A y$ and the coproduct type $A + B$. In particular, we assume the empty type 0 and the singleton type 1. Furthermore, we assume the existence of a type of natural numbers, given as an inductive type according to the rules given in [13, Chap. 1.9]. Finally, we assume the univalence axiom for the universe $\mathcal{U}$ as presented in [13, Chap. 2.10].

Concerning terms, function application is denoted by parentheses as in $f(x)$ or, occasionally, simply by juxtaposition. We write dependent pairs as (a, b) for $b : B(a)$. Projections are indicated by a subscript, that is, for $x : \sum_{(a:A)} B(a)$ we have $x_0 : A$ and $x_1 : B(x_0)$. Indices are also used occasionally to specify earlier arguments of a function of several arguments; e.g., we write $B_i(a)$ instead of $B(i)(a)$.

We conclude this brief introduction by recalling two important internally definable properties of types: we call the type X **contractible**, if X is inhabited by a unique element, that is, if the following type is inhabited:

$$\mathsf{isContr}(X) := \sum_{(x:X)} \prod_{(x':X)} x' = x \ .$$

We call the type Y **a proposition** if for all $y, y' : Y$, we have $y = y'$. Note that a type X is contractible iff X is a proposition and there is an element $x : X$ (see also [13, Lemma 3.11.3]).

3 Definition of M-types via universal property

Coinductive types represent *potentially infinite* data structures, such as streams or infinite lists. As such, they have to be contrasted to *inductive* datatypes, which represent structures that are *necessarily finite*, such as unary natural numbers or finite lists.

3.1 Signatures, a.k.a. containers

In order to analyze inductive and coinductive types systematically, one usually fixes a notion of "signature": a signature specifies, in an abstract way, the rules according to which the instances of a data structure are built. In the following, we consider signatures to be given by "containers" [4]:

▶ **Definition 1.** A **container** (or **signature**) is a pair (A, B) of a type A and a dependent type $x : A \vdash B(x) : \mathcal{U}$ over A.

The container (A, B) then determines a type of "trees" built as follows: such a tree consists of a *root node*, labeled by an element $a : A$, and a family of trees – "subtrees" of the original tree – indexed by the type $B(a)$. A tree is *well-founded* if it does not have an infinite chain of subtrees.

To the container (A, B) one associates two types of trees built according to those rules: the type $\mathsf{W}(A, B)$ of well-founded trees, and the type $\mathsf{M}(A, B)$ of all trees, i.e., not necessarily well-founded.

The description of the inhabitants of $\mathsf{W}(A, B)$ and $\mathsf{M}(A, B)$ in terms of trees gives a suitable intuition; formally, those types are defined in terms of a *universal property*. Indeed, $\mathsf{M}(A, B)$ will be defined as (the carrier of) a terminal object in a suitable sense.

3.2 Coalgebras for a signature

Any container (A, B) specifies an endomorphism on types as follows:

▶ **Definition 2.** Given a container (A, B), define the **polynomial functor** $P : \mathcal{U} \to \mathcal{U}$ associated to (A, B) as

$$P(X) := P_{A,B}(X) := \sum_{a:A} (B(a) \to X) \ .$$

Given a map $f : X \to Y$, define $Pf : PX \to PY$ as the map

$$Pf(a, g) := (a, f \circ g) \ .$$

Note that Definition 2 does not really define a functor, and, more fundamentally, the universe $\mathcal{U}$ is not a (pre-)category in the sense of [3]. Instead, the appropriate notion for P would be an ∞-(endo)functor on the $(\infty, 1)$-category $\mathcal{U}$ [8]. However, we do not attempt to make any of these notions precise, and do not make use of any "functorial" properties of the defined maps. Our use of the word "functor" merely indicates an analogy to the 1-categorical case.

To any signature $S = (A, B)$ we associate a type of *coalgebras* Coalg_S, and a family of types of morphisms between them:

▶ **Definition 3.** Given a signature $S = (A, B)$ as in Definition 2, an **S-coalgebra** is defined to be a pair (C, γ) consisting of a type $C : \mathcal{U}$ and a map $\gamma : C \to P_S C$. A map of coalgebras from (C, γ) to (D, δ) is defined to be a pair (f, p) of a map $f : C \to D$ and a path $p : \delta \circ f = P_S f \circ \gamma$. Put differently, we set

$$\mathsf{Coalg}_S := \sum_{C:\mathcal{U}} C \to PC$$

and

$$\mathsf{Coalg}_S\Big((C, \gamma), (D, \delta)\Big) := \sum_{f:C\to D} \delta \circ f = Pf \circ \gamma \ .$$

There is an obvious composition of coalgebra morphisms, and the identity map $C \to C$ is the carrier of a coalgebra endomorphism on $(\mathcal{C}, \gamma)$. We also write $(C, \gamma) \Rightarrow (D, \delta)$ for the type of coalgebra morphisms from (C, γ) to (D, δ).

3.3 What is an M-type?

In this section we define (internal) M-types in HoTT via a universal property.

▶ **Definition 4.** Given a container (A, B), **the (internal) M-type $\mathsf{M}_{A,B}$ associated to** (A, B) is defined to be the pair $(M, \mathsf{out} : M \to P_{A,B}M)$ with the following universal property: for any coalgebra $(C, \gamma) : \mathsf{Coalg}_{(A,B)}$ of (A, B), the type of coalgebra morphisms $(C, \gamma) \Rightarrow (M, \mathsf{out})$ from (C, γ) to (M, out) is contractible.

The use of the definite article in Definition 4 is justified by the following lemma:

▶ **Lemma 5.** *The type*

$$\mathsf{Final}_S := \sum_{((X,\rho):\mathsf{Coalg}_S)} \prod_{((C,\gamma):\mathsf{Coalg}_S)} \mathsf{isContr}\big((C, \gamma) \Rightarrow (X, \rho)\big)$$

is a proposition.

Proof. The proof that any two final coalgebras (L, out) and (L', out') have equivalent carriers is standard. The Univalence Axiom then implies that the carriers are (propositionally) equal, $L = L'$. It then remains to show that the coalgebra structure out, when transported along this identity, is equal to out'. We refer to the formalized proof for details. ◀

That is, any inhabitant of Final_S is necessarily unique up to propositional equality. We refer to this inhabitant as the "final coalgebra". In Section 4 we construct the final coalgebra.

In the introduction, we use the adjectives "internal" and "external" to distinguish between types specified via universal properties and type-theoretic rules, respectively. Since we do not consider rules for M-types (that is, external M-types) in this work, we drop the adjective "internal" in what follows.

In the following example, we anticipate the result of the next section, namely the existence of a final coalgebra for any signature (A, B):

▶ **Example 6.** The coinductive type $\mathsf{Stream}(A_0)$ of streams over a base type A_0 is given by $\mathsf{M}(A, B)$ with $A = A_0$ and $B(a) := 1$ for any $a : A_0$. The corresponding polynomial functor P satisfies $P(X) = A_0 \times X$.

Using finality of $\mathsf{M}(A, B)$ we can define maps into streams and prove that they have the expected computational behaviour. For example, the zip function

$$\mathsf{zip} : \mathsf{Stream}(A) \times \mathsf{Stream}(B) \to \mathsf{Stream}(A \times B)$$

can be obtained from the universal property applied to the coalgebra

$$\theta : \mathsf{Stream}(A) \times \mathsf{Stream}(B) \to (A \times B) \times (\mathsf{Stream}(A) \times \mathsf{Stream}(B))$$
$$\theta(xs, ys) := ((\mathsf{head}(xs), \mathsf{head}(ys)), (\mathsf{tail}(xs), \mathsf{tail}(ys))$$

where $\mathsf{head} : \mathsf{Stream}(X) \to X$ and $\mathsf{tail} : \mathsf{Stream}(X) \to \mathsf{Stream}(X)$ are the two components of the final coalgebra out. The computational behaviour of zip is expressed by the fact that zip is a coalgebra morphism

$$\mathsf{zip}(xs, ys) = \mathsf{cons}((\mathsf{head}(xs), \mathsf{head}(ys)), (\mathsf{zip}(\mathsf{tail}(xs), \mathsf{tail}(ys)))),$$

where $\mathsf{cons} = \mathsf{out}^{-1}$.

4 Derivability of M-types

In Section 3 we defined the type Final_S of final coalgebras of a signature S, and showed that this type is a proposition (Lemma 5). In this section, we construct an element of Final_S, which, combined with Lemma 5, proves the following theorem per the remark at the end of Section 2:

▶ **Theorem 7.** *The type*

$$\mathsf{Final}_S = \sum_{((X,\rho))} \prod_{((C,\gamma))} \mathsf{isContr}\big((C, \gamma) \Rightarrow (X, \rho)\big)$$

is contractible.

The construction of the final coalgebra is done in several steps, inspired by a construction of M-types from W-types in a type theory satisfying Axiom K by Altenkirch et al. [4]. Its carrier is defined as the limit of a *chain*:

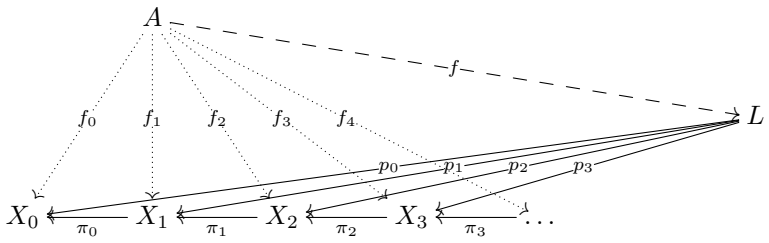


Figure 2 Universal property of L.

▶ **Definition 8.** A **chain** is a pair (X, π) of a family of types $X : \mathbb{N} \to \mathcal{U}$ and a family of functions $\pi_n : X_{n+1} \to X_n$. Here and below we write $X_n := X(n)$ for the nth component of the family X.

The (homotopy) limit of such a chain is given by the type of "compatible tuples":

▶ **Definition 9.** The **limit** of the chain (X, π) is given by the type

$$L := \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} \prod_{(n:\mathbb{N})} \pi_n x_{n+1} = x_n \quad .$$

The limit is equipped with projections $p_n : L \to X_n$, and $\beta_n : \pi_n \circ p_{n+1} = p_n$. Sometimes, we simply write

$$L = \lim X,$$

when the maps π are clear.

Note that this limit (we drop the adjective "homotopy") is an instance of the general construction of homotopy limits by Avigad, Kapulkin, and Lumsdaine [5].

▶ **Lemma 10.** *The type L satisfies the following universal property: for all types A, we have an equivalence of types between maps into L and "cones" over X:*

$$A \to L \quad \simeq \sum_{(f:\prod_{(n:\mathbb{N})} A \to X_n)} \prod_{(n:\mathbb{N})} \pi_n \circ f_{n+1} = f_n \quad =: \mathsf{Cone}(A) \quad .$$

The equivalence, from left to right, maps a function $f : A \to L$ to its projections $p_n \circ f$, as shown in Figure 2.

The next lemma is about tuples in **co**chains, that is, tuples in chains with inverted arrows. Those tuples are determined by their first element:

▶ **Lemma 11.** *Let $X : \mathbb{N} \to \mathcal{U}$ be a family of types, and $l : \prod_{(n:\mathbb{N})} X_n \to X_{n+1}$ a family of functions. Let*

$$Z := \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} \prod_{(n:\mathbb{N})} x_{n+1} = l_n(x_n) \quad .$$

Then the projection $Z \to X_0$ is an equivalence.

Proof. Let G be the functor defined by $GY = 1 + Y$. Fix an element $z : Z$. Then z and l together define a G-algebra structure on X, regarded as a fibration over $\mathbb{N}$. Since $\mathbb{N}$ is the homotopy initial algebra of G, the type Sz of algebra sections of X is contractible. But Z is equivalent to

$$\sum_{(z:X_0)} \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} (x_0 = z) \times \left(\prod_{n:\mathbb{N}} x_{n+1} = l_n(x_n) \right) \ ,$$

which is exactly $\sum_{(z:X_0)} Sz \simeq X_0$. ◀

▶ **Lemma 12.** *Let (X, π) be a chain, and let (X', π') be the* shifted *chain, defined by $X'_n := X_{n+1}$ and $\pi'_n := \pi_{n+1}$. Then the two chains have equivalent limits.*

Proof. Let L and L' be the limits of (X, π) and (X', π'), respectively. We have

$$\begin{aligned}
L' &\overset{(\mathbf{1})}{\simeq} \sum_{(y:\prod_{(n:\mathbb{N})} X_{n+1})} \prod_{(n:\mathbb{N})} \pi_{n+1} y_{n+1} = y_n \\
&\overset{(\mathbf{2})}{\simeq} \sum_{(x_0:X_0)} \sum_{(y:\prod_{(n:\mathbb{N})} X_{n+1})} (\pi_0 y_0 = x_0) \times \left(\prod_{n:\mathbb{N}} \pi_{n+1} y_{n+1} = y_n \right) \\
&\overset{(\mathbf{3})}{\simeq} \sum_{x:\prod_{(n:\mathbb{N})} X_n} (\pi_0 x_0 = x_0) \times \left(\prod_{n:\mathbb{N}} \pi_{n+1} x_{n+2} = x_{n+1} \right) \\
&\overset{(\mathbf{4})}{\simeq} \sum_{(x:\prod_{(n:\mathbb{N})} X_n)} \prod_{(n:\mathbb{N})} \pi_n x_{n+1} = x_n \\
&\overset{(\mathbf{5})}{\simeq} L \ ,
\end{aligned}$$

where (**1**) and (**5**) are by definition. Equivalence (**2**) is given by multiplying with the contractible type $\sum_{(x_0:X_0)} \pi_0 y_0 = x_0$ [13, Lem. 3.11.8] and subsequent swapping of components in a direct product. Equivalence (**3**) is given by joining the first two components, and similarly in (**4**) the last two components are joined. ◀

The next lemma says that polynomial functors (see Definition 2) commute with limits of chains. Let (A, B) be a container with associated polynomial functor $P = P_{A,B}$. Let (X, π) be a chain with limit (L, p). Define the chain $(PX, P\pi)$ with $PX_n := P(X_n)$ and likewise for $P\pi$, and let L^P be its limit. The family of maps $Pp_n : PL \to PX_n$ determines a function $\alpha : PL \to L^P$.

▶ **Lemma 13.** *The function α is an isomorphism.*

Proof. By "equational" reasoning we have

$$
\begin{aligned}
L^P &\overset{(6)}{\simeq} \sum_{(w:\prod_{(n:\mathbb{N})} \sum_{(a:A)} B(a)\to X_n)} \prod_{(n:\mathbb{N})} (P\pi_n)w_{n+1} = w_n \\
&\overset{(7)}{\simeq} \sum_{(a:\prod_{(n:\mathbb{N})} A)} \sum_{(u:\prod_{(n:\mathbb{N})} B(a_n)\to X_n)} \prod_{(n:\mathbb{N})} (a_{n+1}, \pi_n \circ u_{n+1}) = (a_n, u_n) \\
&\overset{(8)}{\simeq} \sum_{(a:\prod_{(n:\mathbb{N})} A)} \sum_{(p:\prod_{(n:\mathbb{N})} a_{n+1}=a_n)} \sum_{(u:\prod_{(n:\mathbb{N})} B(a_n)\to X_n)} \prod_{(n:\mathbb{N})} (p_n)_*(\pi_n \circ u_{n+1}) = u_n \\
&\overset{(9)}{\simeq} \sum_{(a:A)} \sum_{(u:\prod_{(n:\mathbb{N})} B(a)\to X_n)} \prod_{(n:\mathbb{N})} \pi_n \circ u_{n+1} = u_n \\
&\overset{(10)}{\simeq} \sum_{a:A} B(a) \to L \\
&\overset{(11)}{\simeq} PL
\end{aligned}
$$

where (**6**) and (**11**) are by definition, (**7**) is by swapping Π and Σ, (**8**) by expanding equality of pairs, (**9**) by applying Lemma 11 and (**10**) by universal property of L. Verifying that the composition of these isomorphisms is α is straightforward. ◀

Proof of Theorem 7. We now construct a terminal coalgebra for a container (A, B). Let $P = P_{A,B}$ the polynomial functor associated to the container. By recursion on $\mathbb{N}$ we define the chain

$$1 \xleftarrow{!} P1 \xleftarrow{P!} P^2 1 \xleftarrow{P^2!} P^3 \xleftarrow{P^3!} \dots$$

which for brevity we call (W, π), that is, $W_n := P^n 1$ and $\pi_n := P^n !$.

Let (L, p) be the limit of (W, π). If L' is the limit of the shifted chain, we have a sequence of equivalences

$$PL \overset{(12)}{\simeq} L' \overset{(13)}{\simeq} L$$

where (**12**) is given by Lemma 13 and (**13**) by Lemma 12. We denote this equivalence by $\mathsf{in} : PL \to L$, and its inverse by $\mathsf{out} : L \to PL$.

It is worth noting that the construction of L "does not raise the universe level", i.e., if A and B are contained in some universe $\mathcal{U}$, then L is contained in $\mathcal{U}$ as well. In other words, we only need one universe to carry out our construction of the final coalgebra.

We will now show that (L, out) is a final (A, B)-coalgebra. For this, let (C, γ) be any coalgebra, i.e., $\gamma : C \to PC$. The type of coalgebra morphisms $(C, \gamma) \Rightarrow (L, \mathsf{out})$ is given by

$$U := \sum_{f:C\to L} \mathsf{out} \circ f = Pf \circ \gamma \ .$$

We need to show that U is contractible. We compute as follows (see below the math display for intermediate definitions):

$$
\begin{aligned}
U &\overset{(\mathbf{14})}{\simeq} \sum_{f:C\to L} \mathsf{out}\circ f = Pf\circ\gamma \\
&\overset{(\mathbf{15})}{\simeq} \sum_{f:C\to L} \mathsf{out}\circ f = \mathsf{step}(f) \\
&\overset{(\mathbf{16})}{\simeq} \sum_{f:C\to L} \mathsf{in}\circ\mathsf{out}\circ f = \mathsf{in}\circ\mathsf{step}(f) \\
&\overset{(\mathbf{17})}{\simeq} \sum_{f:C\to L} f = \Psi(f) \\
&\overset{(\mathbf{18})}{\simeq} \sum_{c:\mathsf{Cone}} e(c) = \Psi(e(c)) \\
&\overset{(\mathbf{19})}{\simeq} \sum_{c:\mathsf{Cone}} e(c) = e(\Phi(c)) \\
&\overset{(\mathbf{20})}{\simeq} \sum_{c:\mathsf{Cone}} c = \Phi(c) \\
&\overset{(\mathbf{21})}{\simeq} \sum_{((u,q):\mathsf{Cone})} \sum_{(p:u=\Phi_0(u))} p_*(q) = \Phi_1(u)(q) \\
&\overset{(\mathbf{22})}{\simeq} \sum_{(u:\mathsf{Cone}_0)} \sum_{(p:u=\Phi_0 u)} \sum_{q:\mathsf{Cone}_1 u} p_*(q) = \Phi_1(u)(q) \\
&\overset{(\mathbf{23})}{\simeq} \sum_{t:1} 1 \\
&\simeq 1
\end{aligned}
$$

where we use the following definitions: The function $\mathsf{step}_Y : (C \to Y) \to (C \to PY)$ is defined as $\mathsf{step}_Y(f) := Pf \circ \gamma$ and $\Psi : (C \to L) \to (C \to L)$ is defined as $\Psi(f) := \mathsf{in} \circ \mathsf{step}_L(f)$. The map $\Phi : \mathsf{Cone} \to \mathsf{Cone}$ is the counterpart of Ψ on the side of cones. We define $\Phi(u, g) = (\Phi_0 u, \Phi_1 u(g)) : \mathsf{Cone} \to \mathsf{Cone}$ with

$$
\begin{aligned}
(\Phi_0 u)_0 &:= x \mapsto tt : C \to 1 = W_0 \\
(\Phi_0 u)_{n+1} &:= \mathsf{step}_{W_n}(u_n) : C \to W_{n+1} = PW_n
\end{aligned}
$$

and analogously for Φ_1 on paths. By e we denote the equivalence of Lemma 10 from right to left, and $\mathsf{Cone} = \sum_{(u:\mathsf{Cone}_0)} \mathsf{Cone}_1(u)$ is short for $\mathsf{Cone}(C)$. The equivalence (**16**) follows from in being an equivalence, and (**17**) follows from in and out being inverse to each other. We pass from maps into L to cones in (**18**), using the equivalence of Lemma 10, while (**19**) uses the commutativity of the following square:

$$
\begin{array}{ccc}
\mathsf{Cone} & \xrightarrow{e} & (C \to L) \\
{\scriptstyle\Phi}\downarrow & & \downarrow{\scriptstyle\Psi} \\
\mathsf{Cone} & \xrightarrow{e} & (C \to L).
\end{array}
$$

In (**21**), identity in a sigma type is reduced to identity of the components, and in (**22**) the components are rearranged. Finally, step (**23**) consists of two applications of Lemma 11.

Altogether, this shows that for any coalgebra (C, γ), the type of coalgebra morphisms $(C, \gamma) \Rightarrow (L, \mathsf{out})$ is contractible. This concludes the construction of a final coalgebra for the (polynomial functor of the) signature (A, B) and thus, combined with Lemma 5, the proof of Theorem 7. ◀

From the construction of L we get the following corollary about the homotopy level of M-types:

▶ **Lemma 14.** *The homotopy level of the (carrier of the)* M*-type associated to the signature* (A, B) *is bounded by that of the type of nodes* A*, that is,*

$$\mathsf{isofhlevel}_n(A) \to \mathsf{isofhlevel}_n(\mathsf{M}(A, B)) \ .$$

▶ **Example 15.** We continue the example of streams of Example 6, with $A = A_0$ the type of nodes. In that case, the chain considered in the proof of Theorem 7 is given by $W_n = P^n(1) = A^n$, and the map $\pi_n : A^{n+1} \to A^n$ chops of the $(n+1)$th element of any $(n+1)$-tuple. The limit L is hence given by $A^{\mathbb{N}} = (\mathbb{N} \to A)$. The type of streams over A has the same homotopy level as the type A of nodes.

We conclude this section with a proof of the *principle of coinduction*:

▶ **Definition 16** (Bisimulation)**.** Let (C, γ) be a coalgebra for some signature S with associated polynomial functor P and let $\mathcal{R} : C \to C \to \mathcal{U}$ be a binary relation. Define

$$\overline{\mathcal{R}} := \sum_{(a:C)} \sum_{(b:C)} \mathcal{R}(a)(b)$$

along with two projections $\pi_1^{\overline{\mathcal{R}}}(a, b, p) := a$ and $\pi_2^{\overline{\mathcal{R}}}(a, b, p) := b$.

An S*-bisimulation* is a relation $\mathcal{R}$ together with a map $\alpha_{\mathcal{R}} : \overline{\mathcal{R}} \to P(\overline{\mathcal{R}})$ such that both $\pi_1^{\overline{\mathcal{R}}}$ and $\pi_2^{\overline{\mathcal{R}}}$ are P-coalgebra morphisms:

$$\begin{array}{ccccc}
C & \xleftarrow{\pi_1^{\overline{\mathcal{R}}}} & \overline{R} & \xrightarrow{\pi_2^{\overline{\mathcal{R}}}} & C \\
\downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \alpha_{\mathcal{R}}} & & \downarrow{\scriptstyle \gamma} \\
P(C) & \xleftarrow[P(\pi_1^{\overline{\mathcal{R}}})]{} & P(\overline{R}) & \xrightarrow[P(\pi_2^{\overline{\mathcal{R}}})]{} & P(C)
\end{array}$$

We say that a bisimulation is an *equivalence bisimulation* when the underlying relation is an equivalence relation.

▶ **Lemma 17.** *The identity relation* $\cdot = \cdot$ *over an* S*-coalgebra* C *is an equivalence bisimulation. We write* Δ_C *for* $\overline{\cdot = \cdot}$*.*

▶ **Theorem 18** (Coinduction proof principle)**.** *Let* (L, out) *be the final coalgebra for* S*. For any bisimulation* $\overline{\mathcal{R}}$ *over* L*, we have* $\overline{\mathcal{R}} \subseteq \Delta_L$*. That is, for any* $m, m' : L$*,*

$$\mathcal{R}(m)(m') \to m = m'.$$

Proof. Since (L, out) is the final coalgebra, for any coalgebra (C, γ) there exists a unique coalgebra morphism $\mathsf{unfold}_C : C \to L$. It follows that $\pi_1^{\overline{\mathcal{R}}} = \mathsf{unfold}_{\overline{\mathcal{R}}} = \pi_2^{\overline{\mathcal{R}}}$. Finally, given $r : \mathcal{R}(m)(m')$, we obtain $m = \pi_1^{\overline{\mathcal{R}}}(m, m', r) = \pi_2^{\overline{\mathcal{R}}}(m, m', r) = m'$. ◀

5 Indexed M-types

In this section, we briefly state the main definitions for *indexed* M-types. The difference to plain M-types is that the type of nodes of an indexed M-type is actually given by a family of types, indexed by a type of sorts I.

▶ **Definition 19.** An **indexed container** is given by a quadruple (I, A, B, r) such that $I : \mathcal{U}$ is a type, $i : I \vdash A(i) : \mathcal{U}$ is a family of types dependent on I, B is a family $i : I, a : A(i) \vdash B_i(a) : \mathcal{U}$ and r specifies the "sort" of the subtrees, i.e., $r : \prod_{(i:I)} \prod_{(a:A(i))} B_i(a) \to I$.

▶ **Definition 20.** The polynomial functor P associated to an indexed container (I, A, B, r) is an endofunction on the type $I \to \mathcal{U}$:

$$(PX)(i) := \sum_{(a:A(i))} \prod_{(b:B_i(a))} X(r_{i,a}(b)) \ .$$

The functorial action on morphisms is, analogously to Definition 2, given by postcomposition.

Coalgebras for indexed containers, and their morphisms, are defined completely analogously to Definition 3. Again, we prove that terminal coalgebras for indexed containers exist uniquely:

▶ **Theorem 21.** *Let (I, A, B, r) be an indexed container. Then the associated indexed* M*-type is uniquely specified and can be constructed in the type theory described in Section 2.*

An example of a coinductive type that needs indices to be expressed as an M-type is a coinductive formulation of *equivalence of types*, to our knowledge due to T. Altenkirch:

▶ **Example 22.** Let $I := \mathcal{U} \times \mathcal{U}$, and let $A : I \to \mathcal{U}$ be defined by $A(X, Y) := (X \to Y) \times (Y \to X)$. Define B as $B(X, Y)(f, g) := X \times Y$ and $r(X, Y)(f, g)(x, y) := (f(x) = y) \times (x = g(y))$. Then the associated M-type is a family $\mathsf{M} : I \to \mathcal{U}$ and $M(A, B)$ is equivalent to $A \simeq B$.

6 Formalization

The proofs contained in this paper have been formalised in the proof assistant Agda in a self-contained development. The proof has been type-checked by version 2.4.2.2 of Agda. The source code as well as HTML documentation can be found on `https://github.com/HoTT/m-types/`. The source code is also archived with the arXiv version of this article [2].

The formalised proofs deal with the indexed case (Section 5) directly, but apart from that, they correspond closely to the informal proofs presented here. In particular, they make heavy use of the "equational reasoning" technique to prove equivalences between types.

In fact, it is often the case that proving an equivalence between types A and B "directly", i.e. by defining functions $A \to B$ and $B \to A$, and then proving that they compose to identities in both directions, is unfeasibly hard, due to the complexity of the terms involved.

However, in most cases, we can construct an equivalence between A and B by composition of several simple equivalences. Those simple building blocks range from certain ad-hoc equivalences that make specific use of the features of the two types involved, to very general and widely applicable "rewriting rules", like the fact that we can swap a Σ-type with a Π-type (sometimes called the *constructive axiom of choice* [13]).

By assembling elementary equivalences, then, we automatically get both a function $A \to B$ *and* a proof that it is an equivalence. However, sometimes care is needed to ensure that the resulting function has the desired computational properties.

An important consideration during the formalisation of the proof of Theorem 7 was keeping the size of the terms reasonably short. For example, in one early attempt, the innocent-looking term in was being normalised into a term spanning more than 12000 lines.

The explosion in term size was clearly causing performance issues during type-checking, which resulted in Agda running out of memory while checking apparently trivial proofs.

We solved this problem by moving certain definitions (like that of in itself) into an *abstract* block, thereby preventing Agda from expanding it at all. Of course, this means that we lost all the computational properties of certain functions, so we had to abstract out their computational behaviour in the form of propositional equalities, and manually use them in the proof of Theorem 7. This work-around is the source of most of the complications in the formal proof.

7 Conclusion and future work

We have shown how to construct a class of coinductive types from the basic type constructors and natural numbers in Homotopy Type Theory. Our construction follows a well-known pattern, that is known to work in type theory with identity reflection rule.

We work in a univalent universe, but we make minimal use of univalence itself: Lemma 5 is the only result that uses it directly, and elsewhere univalence only appears indirectly through the use of functional extensionality. We intend to make these dependencies more explicit in future developments of the formalisation.

Finally, the coinductive types we construct do not satisfy the expected computation rules *judgmentally*, but only *propositionally*. This fact would justify adding coinduction as a primitive rather than a derived notion – provided that judgmental computation rules are validated by the intended semantics. Those semantic questions are left for future work.

Acknowledgments. We are grateful to many people for helpful discussions about coinductive types, online as well as offline: Thorsten Altenkirch, Steve Awodey, Martín Escardó, Nicolai Kraus, Peter LeFanu Lumsdaine, Ralph Matthes, Paige North, Mike Shulman, and Vladimir Voevodsky. We thank Nicolai Kraus, Peter LeFanu Lumsdaine and Paige North for suggesting improvements and clarifications for this paper.

References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.

2 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in Homotopy Type Theory, 2015. arXiv:1504.02949.

3 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, FirstView:1–30, 1 2015. doi:10.1017/S0960129514000486.

4 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed Containers. available at `http://www.cs.nott.ac.uk/~txa/publ/jcont.pdf`.

5 Jeremy Avigad, Krzysztof Kapulkin, and Peter LeFanu Lumsdaine. Homotopy limits in type theory. *Mathematical Structures in Computer Science*, FirstView:1–31, 1 2015. doi:10.1017/S0960129514000498.

6 Steve Awodey, Nicola Gambino, and Kristina Sojakova. Inductive types in homotopy type theory. In *Proceedings of Symposium on Logic In Computer Science*, pages 95–104, 2012. doi:10.1109/LICS.2012.21.

7 Michael Barr. Terminal coalgebras in well-founded set theory. *Theor. Comput. Sci.*, 114(2):299–315, 1993. doi:10.1016/0304-3975(93)90076-6.

8 James Cranch. Concrete Categories in Homotopy Type Theory, 2013. arXiv:1311.1852.

9 Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The Simplicial Model of Univalent Foundations, 2014. arXiv:1211.2851.

10 HoTT mailing list. Discussion on coinductive types on HoTT mailing list, `https://groups.google.com/d/msg/homotopytypetheory/tYRTcI2Opyo/PIrI6t5me-oJ`.

11 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

12 Kristina Sojakova. Higher inductive types as homotopy-initial algebras. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 31–42, 2015. doi:10.1145/2676726.2676983.

13 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

14 Benno van den Berg and Federico De Marchi. Non-well-founded trees in categories. *Ann. Pure Appl. Logic*, 146(1):40–59, 2007. arXiv:math/0409158.

Conservativity of Embeddings in the $\lambda\Pi$ Calculus Modulo Rewriting

Ali Assaf[1,2]

1 **Inria, Paris, France**
2 **École polytechnique, Palaiseau, France**

Abstract

The $\lambda\Pi$ calculus can be extended with rewrite rules to embed any functional pure type system. In this paper, we show that the embedding is conservative by proving a relative form of normalization, thus justifying the use of the $\lambda\Pi$ calculus modulo rewriting as a logical framework for logics based on pure type systems. This result was previously only proved under the condition that the target system is normalizing. Our approach does not depend on this condition and therefore also works when the source system is not normalizing.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases $\lambda\Pi$ calculus modulo rewriting, pure type systems, logical framework, normalization, conservativity

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.31

1 Introduction

The $\lambda\Pi$ *calculus modulo rewriting* is a logical framework that extends the $\lambda\Pi$ calculus [10] with rewrite rules. Through the Curry-de Bruijn-Howard correspondence, it can express properties and proofs of various logics. Cousineau and Dowek [6] introduced a general embedding of *functional pure type systems* (FPTS), a large class of typed λ-calculi, in the $\lambda\Pi$ calculus modulo rewriting: for any FPTS λS, they constructed the system $\lambda\Pi/S$ using appropriate rewrite rules, and defined two translation functions $|M|$ and $\|A\|$ that translate respectively the terms and the types of λS to $\lambda\Pi/S$. This embedding is complete, in the sense preserves typing: if $\Gamma \vdash_{\lambda S} M : A$ then $\|\Gamma\| \vdash_{\lambda\Pi/S} |M| : \|A\|$. From the logical point of view, it preserves provability. The converse property, called *conservativity*, was only shown partially: assuming $\lambda\Pi/S$ is strongly normalizing, if there is a term N such that $\|\Gamma\| \vdash_{\lambda\Pi/S} N : \|A\|$ then there is a term M such that $\Gamma \vdash_{\lambda S} M : A$.

1.1 Normalization and conservativity

Not much is known about normalization in $\lambda\Pi/S$. Cousineau and Dowek [6] showed that the embedding preserves reduction: if $M \longrightarrow M'$ then $|M| \longrightarrow^+ |M'|$. As a consequence, if $\lambda\Pi/S$ is strongly normalizing (i.e. every well-typed term normalizes) then so is λS, but the converse might not be true *a priori*. This was not enough to show the conservativity of the embedding, so the proof relied on the unproven assumption that $\lambda\Pi/S$ is normalizing. This result is insufficient if one wants to consider the $\lambda\Pi$ calculus modulo rewriting as a general logical framework for defining logics and expressing proofs in those logics, as proposed in [4, 5]. Indeed, if the embedding turns out to be inconsistent then checking proofs in the logical framework has very little benefit.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 31–44

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Consider the PTS λHOL that corresponds to higher order logic [1]:

$$\begin{array}{lcl} \mathcal{S} & = & \mathsf{Prop}, \mathsf{Type}, \mathsf{Kind} \\ \mathcal{A} & = & (\mathsf{Prop} : \mathsf{Type}), (\mathsf{Type} : \mathsf{Kind}) \\ \mathcal{R} & = & (\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Type}, \mathsf{Type}) \end{array}$$

This PTS is strongly normalizing, and therefore consistent. A polymorphic variant of λHOL is specified by $U^- = HOL + (\mathsf{Kind}, \mathsf{Type}, \mathsf{Type})$. It turns out that λU^- is inconsistent: there is a term ω such that $\vdash_{\lambda U^-} \omega : \Pi\alpha : \mathsf{Prop}.\, \alpha$ and which is not normalizing [1]. We motivate the need for a proof of conservativity with the following example.

▶ **Example 1.** The polymorphic identity function $I = \lambda\alpha : \mathsf{Type}.\, \lambda x : \alpha.\, x$ is *not* well-typed in λHOL, but it is well-typed in λU^- and so is its type:

$$\vdash_{\lambda U^-} I : \Pi\alpha : \mathsf{Type}.\, \alpha \to \alpha$$

$$\vdash_{\lambda U^-} \Pi\alpha : \mathsf{Type}.\, \alpha \to \alpha : \mathsf{Type}$$

However, the translation $|I| = \lambda\alpha : u_{\mathsf{Type}}.\, \lambda x : \varepsilon_{\mathsf{Type}}\, \alpha.\, x$ *is* well-typed in $\lambda\Pi/HOL$:

$$\vdash_{\lambda\Pi/HOL} |I| : \Pi\alpha : u_{\mathsf{Type}}.\, \varepsilon_{\mathsf{Type}}\, \alpha \to \varepsilon_{\mathsf{Type}}\, \alpha$$

$$\vdash_{\lambda\Pi/HOL} \Pi\alpha : u_{\mathsf{Type}}.\, \varepsilon_{\mathsf{Type}}\, \alpha \to \varepsilon_{\mathsf{Type}}\, \alpha : \mathsf{Type}$$

It seems that $\lambda\Pi/HOL$, just like λU^-, allows more functions than λHOL, even though the type of $|I|$ is not the translation of a λHOL type. Does that make $\lambda\Pi/HOL$ inconsistent?

1.2 Absolute normalization vs relative normalization

One way to answer the question is to prove strong normalization of $\lambda\Pi/S$ by constructing a model, for example in the algebra of *reducibility candidates* [9]. Dowek [7] recently constructed such a model for the embedding of higher-order logic (λHOL) and of the calculus of constructions (λC). However, this technique is still very limited. Indeed, proving such a result is, by definition, at least as hard as proving the consistency of the original system. It requires specific knowledge of λS and the construction of such a model can be very involved, such as for the calculus of constructions with an infinite universe hierarchy (λC^∞).

In this paper, we take a different approach and show that $\lambda\Pi/S$ is conservative in all cases, even when λS is *not* normalizing. Instead of showing that $\lambda\Pi/S$ is strongly normalizing, we show that it is weakly normalizing *relative to* λS, meaning that proofs in the target language can be reduced to proofs in the source language. That way we prove only what is needed to show conservativity, without having to prove the consistency of λS all over again. After identifying the main difficulties, we characterize a *PTS completion* [17, 16] S^* containing S, and define an inverse translation from $\lambda\Pi/S$ to λS^*. We then prove that λS^* is a conservative extension of λS using the *reducibility method* [18].

1.3 Outline

In Section 2, we recall the theory of pure type systems. In Section 3, we present the framework of the λΠ calculus modulo rewriting. In Section 4, we introduce Cousineau and Dowek's embedding of functional pure type systems in the λΠ calculus modulo rewriting. In Section 5, we prove the conservativity of the embedding using the techniques mentioned above. In Section 6, we summarize the results and discuss future work. Some long proofs have been omitted and can be found in the long version of this paper[1].

[1] Available online at arXiv:1504.05038.

$$\textsc{Empty}\ \frac{}{\mathrm{WF}_{\lambda S}(\cdot)} \qquad \textsc{Declaration}\ \frac{\Gamma \vdash_{\lambda S} A : s \qquad x \notin \Gamma}{\mathrm{WF}_{\lambda S}(\Gamma, x : A)} \qquad \textsc{Variable}\ \frac{\mathrm{WF}_{\lambda S}(\Gamma) \qquad (x : A) \in \Gamma}{\Gamma \vdash_{\lambda S} x : A}$$

$$\textsc{Sort}\ \frac{\mathrm{WF}_{\lambda S}(\Gamma) \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash_{\lambda S} s_1 : s_2} \qquad \textsc{Product}\ \frac{\Gamma \vdash_{\lambda S} A : s_1 \qquad \Gamma, x : A \vdash_{\lambda S} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\lambda S} \Pi x{:}A.\, B : s_3}$$

$$\textsc{Abstraction}\ \frac{\Gamma, x : A \vdash_{\lambda S} M : B \qquad \Gamma \vdash_{\lambda S} \Pi x{:}A.\, B : s}{\Gamma \vdash_{\lambda S} \lambda x{:}A.\, M : \Pi x{:}A.\, B} \qquad \textsc{Application}\ \frac{\Gamma \vdash_{\lambda S} M : \Pi x{:}A.\, B \qquad \Gamma \vdash_{\lambda S} N : A}{\Gamma \vdash_{\lambda S} M\, N : B[x \backslash N]}$$

$$\textsc{Conversion}\ \frac{\Gamma \vdash_{\lambda S} M : A \qquad \Gamma \vdash_{\lambda S} B : s \qquad A \equiv_\beta B}{\Gamma \vdash_{\lambda S} M : B}$$

Figure 1 Typing rules of λS.

2 Pure type systems

Pure type systems [1] are a general class of typed λ-calculi parametrized by a specification.

▶ **Definition 2.** A PTS *specification* is a triple $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

- $\mathcal{S}$ is a set of of symbols called *sorts*
- $\mathcal{A} \subseteq \mathcal{S} \times S$ is a set of *axioms* of the form $(s_1 : s_2)$
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *rules* of the form (s_1, s_2, s_3)

We write (s_1, s_2) as a short-hand for the rule (s_1, s_2, s_2). The specification S is *functional* if the relations $\mathcal{A}$ and $\mathcal{R}$ are functional, that is $(s_1, s_2) \in \mathcal{A}$ and $(s_1, s_2') \in \mathcal{A}$ imply $s_2 = s_2'$, and $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1, s_2, s_3') \in \mathcal{R}$ imply $s_3 = s_3'$. The specification is *full* if for all $s_1, s_2 \in \mathcal{S}$, there is a sort s_3 such that $(s_1, s_2, s_3) \in \mathcal{R}$.

▶ **Definition 3.** Given a PTS specification $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ and a countably infinite set of variables $\mathcal{V}$, the abstract syntax of λS is defined by the following grammar:

$$\begin{array}{lrcl} \text{(terms)} & \mathcal{T} & ::= & \mathcal{S} \mid \mathcal{V} \mid \mathcal{T}\,\mathcal{T} \mid \lambda \mathcal{V}{:}\mathcal{T}.\,\mathcal{T} \mid \Pi \mathcal{V}{:}\mathcal{T}.\,\mathcal{T} \\ \text{(contexts)} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}, \mathcal{V} : \mathcal{T} \end{array}$$

We use lower case letters $x, y, \alpha, \beta, \ldots$ to denote variables, uppercase letters such as M, N, $A, B, \ldots$ to denote terms, and uppercase Greek letters such as $\Gamma, \Delta, \Sigma, \ldots$ to denote contexts. The set of free variables of a term M is denoted by $\mathrm{FV}(M)$. We write $A \to B$ for $\Pi x{:}A.\, B$ when $x \notin \mathrm{FV}(B)$.

The typing rules of λS are presented in Figure 1. We write $\Gamma \vdash M : A$ instead of $\Gamma \vdash_{\lambda S} M : A$ when the context is unambiguous. We say that M is a Γ-*term* when $\mathrm{WF}(\Gamma)$ and $\Gamma \vdash M : A$ for some A. We say that A is a Γ-*type* when $\mathrm{WF}(\Gamma)$ and either $\Gamma \vdash A : s$ or $A = s$ for some $s \in \mathcal{S}$. We write $\Gamma \vdash M : A : s$ as a shorthand for $\Gamma \vdash M : A \wedge \Gamma \vdash A : s$.

▶ **Example 4.** The following well-known systems can all be expressed as functional pure type systems using the same set of sorts $\mathcal{S} = \mathsf{Type}, \mathsf{Kind}$ and the same set of axioms $\mathcal{A} = (\mathsf{Type} : \mathsf{Kind})$:

- Simply-typed λ calculus ($\lambda\!\to$):
  $\mathcal{R} = (\mathsf{Type}, \mathsf{Type})$
- System F ($\lambda 2$):
  $\mathcal{R} = (\mathsf{Type}, \mathsf{Type}), (\mathsf{Kind}, \mathsf{Type})$
- $\lambda\Pi$ calculus (λP):
  $\mathcal{R} = (\mathsf{Type}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Kind})$
- Calculus of constructions (λC):
  $\mathcal{R} = (\mathsf{Type}, \mathsf{Type}), (\mathsf{Kind}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Kind}), (\mathsf{Kind}, \mathsf{Kind})$

▶ **Example 5.** Let $I = \lambda\alpha : \mathsf{Type}.\, \lambda x : \alpha.\, x$ be the polymorphic identity function. The term I is not well-typed in the simply typed λ calculus but it is well-typed in the calculus of constructions λC:

$$\vdash_{\lambda C} I : \Pi\alpha : \mathsf{Type}.\, \alpha \to \alpha$$

The following properties hold for all pure type systems [1].

▶ **Theorem 6** (Correctness of types). *If* $\Gamma \vdash_{\lambda S} M : A$ *then* $\mathrm{WF}_{\lambda S}(\Gamma)$ *and either* $\Gamma \vdash_{\lambda S} A : s$ *or* $A = s$ *for some* $s \in \mathcal{S}$*, i.e.* A *is a* Γ*-type.*

The reason why we don't always have $\Gamma \vdash_{\lambda S} A : s$ is that some sorts do not have an associated axiom, such as Kind in Example 4, which leads to the following definition.

▶ **Definition 7** (Top-sorts). A sort $s \in \mathcal{S}$ is called a *top-sort* when there is no sort $s' \in \mathcal{S}$ such that $(s : s') \in \mathcal{A}$.

The following property is useful for proving properties about systems with top-sorts.

▶ **Theorem 8** (Top-sort types). *If* $\Gamma \vdash_{\lambda S} A : s$ *and* s *is a top-sort then either* $A = s'$ *for some sort* $s' \in \mathcal{S}$ *or* $A = \Pi x : B.\, C$ *for some terms* B, C*.*

▶ **Theorem 9** (Confluence). *If* $M_1 \longrightarrow_\beta^* M_2$ *and* $M_1 \longrightarrow_\beta^* M_3$ *then there is a term* M_4 *such that* $M_2 \longrightarrow_\beta^* M_4$ *and* $M_3 \longrightarrow_\beta^* M_4$*.*

▶ **Theorem 10** (Product compatibility). *If* $\Pi x : A.\, B \equiv_\beta \Pi x : A'.\, B'$ *then* $A \equiv_\beta A'$ *and* $B \equiv_\beta B'$*.*

▶ **Theorem 11** (Subject reduction). *If* $\Gamma \vdash_{\lambda S} M : A$ *and* $M \longrightarrow_\beta^* M'$ *then* $\Gamma \vdash_{\lambda S} M' : A$*.*

Finally, we state the following property for functional pure type systems.

▶ **Theorem 12** (Uniqueness of types). *Let* $\mathcal{S}$ *be a functional specification. If* $\Gamma \vdash_{\lambda S} M : A$ *and* $\Gamma \vdash_{\lambda S} M : B$ *then* $A \equiv_\beta B$*.*

In the rest of the paper, all the pure type systems we will consider will be functional.

3 The λΠ calculus modulo rewriting

The $\lambda\Pi$ *calculus*, also known as *LF* and as λP, is one of the simplest forms of λ calculus with dependent types, and corresponds through the Curry-de Bruijn-Howard correspondence to a minimal first-order logic of higher-order terms. As mentioned in Example 4, it can be defined as the functional pure type system λP with the following specification:

$$\begin{array}{lcl} \mathcal{S} & = & \mathsf{Type}, \mathsf{Kind} \\ \mathcal{A} & = & \mathsf{Type} : \mathsf{Kind} \\ \mathcal{R} & = & (\mathsf{Type}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Kind}) \end{array}$$

$$\textsc{Empty}\ \frac{}{\mathrm{WF}_{\lambda\Pi/}(\cdot)} \qquad \textsc{Declaration}\ \frac{\Gamma \vdash_{\lambda\Pi/} A : s \qquad x \notin \Sigma, \Gamma}{\mathrm{WF}_{\lambda\Pi/}(\Gamma, x : A)} \qquad \textsc{Variable}\ \frac{\mathrm{WF}_{\lambda\Pi/}(\Gamma) \qquad (x : A) \in \Sigma, \Gamma}{\Gamma \vdash_{\lambda\Pi/} x : A}$$

$$\textsc{Sort}\ \frac{\mathrm{WF}_{\lambda\Pi/}(\Gamma) \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash_{\lambda\Pi/} s_1 : s_2} \qquad \textsc{Product}\ \frac{\Gamma \vdash_{\lambda\Pi/} A : s_1 \qquad \Gamma, x : A \vdash_{\lambda\Pi/} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash_{\lambda\Pi/} \Pi x{:}A.\, B : s_3}$$

$$\textsc{Abstraction}\ \frac{\Gamma, x : A \vdash_{\lambda\Pi/} M : B \qquad \Gamma \vdash_{\lambda\Pi/} \Pi x{:}A.\, B : s}{\Gamma \vdash_{\lambda\Pi/} \lambda x{:}A.\, M : \Pi x{:}A.\, B} \qquad \textsc{Application}\ \frac{\Gamma \vdash_{\lambda\Pi/} M : \Pi x{:}A.\, B \qquad \Gamma \vdash_{\lambda\Pi/} N : A}{\Gamma \vdash_{\lambda\Pi/} M\, N : B[x \backslash N]}$$

$$\textsc{Conversion}\ \frac{\Gamma \vdash_{\lambda\Pi/} M : A \qquad \Gamma \vdash_{\lambda\Pi/} B : s \qquad A \equiv_{\beta R} B}{\Gamma \vdash_{\lambda\Pi/} M : B}$$

Figure 2 Typing rules of $\lambda\Pi/(\Sigma, R)$.

The $\lambda\Pi$ *calculus modulo rewriting* extends the $\lambda\Pi$ calculus with rewrite rules. By equating terms modulo a set of rewrite rules R in addition to α and β equivalence, it can type more terms using the conversion rule, and therefore express theories that are more complex. The calculus can be seen as a variant of Martin-Löf's logical framework [13, 11] where equalities are expressed as rewrite rules.

We recall that a rewrite rule is a triple $[\Delta]\ M \rightsquigarrow N$ where Δ is a context and M, N are terms such that $\mathrm{FV}(N) \subseteq \mathrm{FV}(M)$. A set of rewrite rules R induces a reduction relation on terms, written $\longrightarrow_R$, defined as the smallest contextual closure such that if $[\Delta]\ M \rightsquigarrow N \in R$ then $\sigma(M) \longrightarrow_R \sigma(N)$ for any substitution σ of the variables in Δ. We define the relation $\longrightarrow_{\beta R}$ as $\longrightarrow_\beta \cup \longrightarrow_R$, the relation $\equiv_R$ as the smallest congruence containing $\longrightarrow_R$, and the relation $\equiv_{\beta R}$ as the smallest congruence containing $\longrightarrow_{\beta R}$.

▶ **Definition 13.** A rewrite rule $[\Delta]\ M \rightsquigarrow N$ is *well-typed in a context* Σ when there is a term A such that $\Sigma, \Delta \vdash_{\lambda\Pi} M : A$ and $\Sigma, \Delta \vdash_{\lambda\Pi} N : A$.

▶ **Definition 14.** Let Σ be a well-formed $\lambda\Pi$ context and R a set of rewrite rules that are well-typed in Σ. The $\lambda\Pi$ *calculus modulo* (Σ, R), written $\lambda\Pi/(\Sigma, R)$, is defined with the same syntax as the $\lambda\Pi$ calculus, but with the typing rules of Figure 2. We write $\lambda\Pi/$ instead of $\lambda\Pi/(\Sigma, R)$ when the context is unambiguous.

▶ **Example 15.** Let Σ be the context

$$\alpha : \mathsf{Type}, c : \alpha, f : \alpha \to \mathsf{Type}$$

and R be the following rewrite rule

$$[\cdot]\ f\, c \rightsquigarrow \Pi y{:}\alpha.\, f\, y \to f\, y$$

Then the term

$$\delta = \lambda x{:}f\, c.\, x\, c\, x$$

is well-typed in $\lambda\Pi/(\Sigma, R)$:

$$\vdash_{\lambda\Pi/(\Sigma,R)} \delta : f\, c \to f\, c$$

Note that the term δ would not be well-typed without the rewrite rule, even if we replace all the occurrences of $f\, c$ in δ by $\Pi y{:}\alpha.\, f\, y \to f\, y$.

The system $\lambda\Pi$ is a pure type system and therefore enjoys all the properties mentioned in Section 2. The behavior of $\lambda\Pi/\,(\Sigma, R)$ however depends on the choice of (Σ, R). In particular, some properties analogous to those of pure type systems depend on the confluence of the relation $\longrightarrow_{\beta R}$.

▶ **Theorem 16** (Correctness of types). *If* $\Gamma \vdash_{\lambda\Pi/} M : A$ *then* $\mathrm{WF}_{\lambda\Pi/}(\Gamma)$ *and either* $\Gamma \vdash_{\lambda\Pi/} A : s$ *for some* $s \in \{\mathsf{Type}, \mathsf{Kind}\}$ *or* $A = \mathsf{Kind}$.

▶ **Theorem 17** (Top-sort types). *If* $\Gamma \vdash_{\lambda\Pi/} A : \mathsf{Kind}$ *then either* $A = \mathsf{Type}$ *or* $A = \Pi x{:}B.\, C$ *for some terms* B, C *such that* $\Gamma, x : B \vdash_{\lambda\Pi/} C : \mathsf{Kind}$.

Assuming $\longrightarrow_{\beta R}$ is confluent, the following properties hold [3].

▶ **Theorem 18** (Product compatibility). *If* $\Pi x : A.\, B \equiv_{\beta R} \Pi x : A'.\, B'$ *then* $A \equiv_{\beta R} A'$ *and* $B \equiv_{\beta R} B'$.

▶ **Theorem 19** (Subject reduction). *If* $\Gamma \vdash_{\lambda\Pi/} M : A$ *and* $M \longrightarrow^*_{\beta R} M'$ *then* $\Gamma \vdash_{\lambda\Pi/} M' : A$.

▶ **Theorem 20** (Uniqueness of types). *If* $\Gamma \vdash_{\lambda\Pi/} M : A$ *and* $\Gamma \vdash_{\lambda\Pi/} M : B$ *then* $A \equiv_{\beta R} B$.

4 Embedding FPTS's in the λΠ calculus modulo

In this section, we present the embedding of functional pure type systems in the $\lambda\Pi$ calculus modulo rewriting as introduced by Cousineau and Dowek [6]. In this embedding, sorts are represented as *universes à la Tarski*, as introduced by Martin-Löf [12] and later developed by Luo [11] and Palmgren [14]. The embedding is done in two steps. First, given a pure type system λS, we construct $\lambda\Pi/S$ by giving an appropriate signature and rewrite system. Second, we define a translation from the terms and types of λS to the terms and types of $\lambda\Pi/S$. The proofs of the theorems in this section can be found in the original paper [6].

▶ **Definition 21** (The system $\lambda\Pi/S$). Consider a functional pure type system specified by $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$. Define Σ_S to be the well-formed context containing the declarations:

$u_s : \mathsf{Type}$	$\forall s \in \mathcal{S}$
$\varepsilon_s : u_s \to \mathsf{Type}$	$\forall s \in \mathcal{S}$
$\dot{s_1} : u_{s_2}$	$\forall s_1 : s_2 \in \mathcal{A}$
$\dot{\pi}_{s_1 s_2 s_3} : \Pi\alpha{:}u_{s_1}.\, (\varepsilon_{s_1}\, \alpha \to u_{s_2}) \to u_{s_3}$	$\forall (s_1, s_2, s_3) \in \mathcal{R}$

Let R_S be the well-typed rewrite system containing the rules

$$[\cdot]\ \varepsilon_{s_2}\, \dot{s_1} \rightsquigarrow u_{s_1}$$

for all $s_1 : s_2 \in \mathcal{A}$, and

$$[\Delta_{s_1 s_2 s_3}]\ \varepsilon_{s_3}\, (\dot{\pi}_{s_1 s_2 s_3}\, A\, B) \rightsquigarrow \Pi x{:}(\varepsilon_{s_1}\, A).\, \varepsilon_{s_2}\, (B\, x)$$

for all $(s_1, s_2, s_3) \in \mathcal{R}$, where $\Delta_{s_1 s_2 s_3} = (A : u_{s_1}, B : (\varepsilon_{s_1}\, \alpha \to u_{s_2}))$. The system $\lambda\Pi/S$ is defined as the $\lambda\Pi$ calculus modulo (Σ_S, R_S), that is, $\lambda\Pi/(\Sigma_S, R_S)$.

▶ **Theorem 22** (Confluence). *The relation* $\longrightarrow_{\beta R}$ *is confluent.*

The translation is composed of two functions, one from the terms of λS to the terms of $\lambda\Pi/S$, the other from the types of λS to the types of $\lambda\Pi/S$.

▶ **Definition 23.** The translation $|M|_\Gamma$ of Γ-terms and the translation $\|A\|_\Gamma$ of Γ-types are mutually defined as follows.

$$\begin{array}{rcl}
|s|_\Gamma & = & \dot{s} \\
|x|_\Gamma & = & x \\
|M\,N|_\Gamma & = & |M|_\Gamma\ |N|_\Gamma \\
|\lambda x\!:\!A.\,M|_\Gamma & = & \lambda x\!:\!\|A\|_\Gamma\,.\ |M|_{\Gamma,x:A} \\
|\Pi x\!:\!A.\,B|_\Gamma & = & \dot{\pi}_{s_1s_2s_3}\ |A|_\Gamma\ (\lambda x\!:\!\|A\|_\Gamma\,.\ |B|_{\Gamma,x:A}) \\
 & & \text{where } \Gamma \vdash A : s_1 \\
 & & \text{and } \Gamma, x : A \vdash B : s_2 \\
 & & \text{and } (s_1, s_2, s_3) \in \mathcal{R}
\end{array}$$

$$\begin{array}{rcl}
\|s\|_\Gamma & = & u_s \\
\|\Pi x\!:\!A.\,B\|_\Gamma & = & \Pi x\!:\!\|A\|_\Gamma\,.\ \|B\|_{\Gamma,x:A} \\
\|A\|_\Gamma & = & \varepsilon_s\ |A|_\Gamma \ \text{ where } \Gamma \vdash A : s
\end{array}$$

Note that this definition is redundant but it is well-defined up to $\equiv_{\beta R}$. In particular, because some Γ-types are also Γ-terms, there are two ways to translate them, but they are equivalent:

$$\begin{array}{rcl}
\varepsilon_{s_2}\,\dot{s_1} & \equiv_{\beta R} & u_{s_1} \\
\varepsilon_{s_3}\ |\Pi x\!:\!A.\,B|_\Gamma & \equiv_{\beta R} & \Pi x\!:\!\|A\|_\Gamma\,.\ \|B\|_{\Gamma,x:A}
\end{array}$$

This definition is naturally extended to well-formed contexts as follows.

$$\begin{array}{rcl}
\|\cdot\| & = & \cdot \\
\|\Gamma, x : A\| & = & \|\Gamma\|\,, x : \|A\|_\Gamma
\end{array}$$

▶ **Example 24.** The polymorphic identity function of the Calculus of constructions λC is translated as

$$|I| = \lambda\alpha\!:\!u_{\mathsf{Type}}.\,\lambda x\!:\!\varepsilon_{\mathsf{Type}}\,\alpha.\,x$$

and its type $A = \Pi\alpha\!:\!\mathsf{Type}.\,\alpha \to \alpha$ is translated as:

$$|A| = \dot{\pi}_{\mathsf{Kind},\mathsf{Type},\mathsf{Type}}\ \dot{\mathsf{Type}}\ (\lambda\alpha\!:\!u_{\mathsf{Type}}.\ |A_\alpha|)$$

where $A_\alpha = \alpha \to \alpha$ and

$$|A_\alpha| = \dot{\pi}_{\mathsf{Type},\mathsf{Type},\mathsf{Type}}\,\alpha\,(\lambda x\!:\!\varepsilon_{Type}\,\alpha.\,\varepsilon_{Type}\,\alpha)$$

The identity function applied to itself is translated as:

$$|I\,A\,I| = |I|\ |A|\ |I|$$

The embedding is complete, in the sense that all the typing relations of λS are preserved by the translation.

▶ **Theorem 25** (Completeness). *For any context* Γ *and terms* M *and* A*, if* $\Gamma \vdash_{\lambda S} M : A$ *then* $\|\Gamma\| \vdash_{\lambda\Pi/S} |M|_\Gamma : \|A\|_\Gamma$.

5 Conservativity

In this section, we prove the converse of the completeness property. One could attempt to prove that if $\|\Gamma\| \vdash_{\lambda\Pi/S} |M|_\Gamma : \|A\|_\Gamma$ then $\Gamma \vdash_{\lambda S} M : A$. However, that would be too weak because the translation $|M|_\Gamma$ is only defined for well-typed terms. A second attempt would be to define inverse translations $\varphi(M)$ and $\psi(A)$ and prove that if $\Gamma \vdash_{\lambda\Pi/S} M : A$ then $\psi(\Gamma) \vdash_{\lambda S} \varphi(M) : \psi(A)$, but that would not work either because not all terms and types of $\lambda\Pi/S$ correspond to valid terms and types of λS, as was shown in Example 1. Therefore the property that we want to prove is: if there is a term N such that $\|\Gamma\| \vdash_{\lambda\Pi/S} N : \|A\|_\Gamma$ then there is a term M such that $\Gamma \vdash_{\lambda S} M : A$.

The main difficulty is that some of these *external* terms can be involved in witnessing valid λS types, as illustrated by the following example.

▶ **Example 26.** Consider the context nat : Type. Even though the polymorphic identity function I and its type are not well-typed in λHOL, they can be used in $\lambda\Pi/HOL$ to construct a witness for $nat \to nat$.

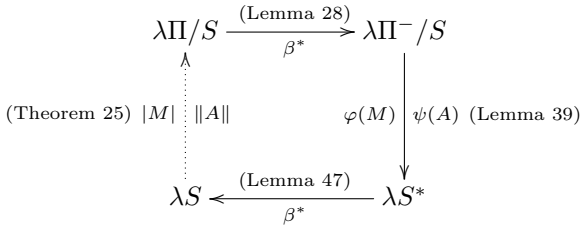
$$nat : u_{\mathsf{Type}} \vdash_{\lambda\Pi/HOL} (|I|\ nat) : (\varepsilon_{\mathsf{Type}}\, nat \to \varepsilon_{\mathsf{Type}}\, nat)$$

We can normalize the term $|I|\ nat$ to $\lambda x : \varepsilon_{\mathsf{Type}}\, nat.\, x$ which is a term that corresponds to a valid λHOL term: it is the translation of the term $\lambda x : nat.\, x$. However, as discussed previously, we cannot restrict ourselves to normal terms because we do not know if $\lambda\Pi/S$ is normalizing.

To prove conservativity, we will therefore need to address the following issues:

1. The system $\lambda\Pi/S$ can type more terms than λS.
2. These terms can be used to construct proofs for the translation of λS types.
3. The $\lambda\Pi/S$ terms that inhabit the translation of λS types can be reduced to the translation of λS terms.

We will proceed as follows. First, we will eliminate β-redexes at the level of Kind by reducing $\lambda\Pi/S$ to a subset $\lambda\Pi^-/S$. Then, we will extend λS to a *minimal completion* λS^* that can type more terms than λS, and show that $\lambda\Pi^-/S$ corresponds to λS^* using inverse translations $\varphi(M)$ and $\psi(A)$. Finally, we will show that λS^* terms inhabiting λS types can be reduced to λS terms. The procedure is summarized in the following diagram.



5.1 Eliminating β-redexes at the level of Kind

In $\lambda\Pi/S$, we can have β-redexes at the level of Kind such as $(\lambda x : A.\, u_s)\, M$. These redexes are artificial and are never generated by the forward translation of any PTS. We show here that they can always be safely eliminated.

▶ **Definition 27.** A Γ-term M of type C is at the level of Kind (resp. Type) if $\Gamma \vdash C$: Kind (resp. $\Gamma \vdash C$: Type). We define $\lambda\Pi^-/S$ terms as the subset of well-typed $\lambda\Pi/S$ terms that do not contain any Kind-level β-redexes.

▶ **Lemma 28.** *For any $\lambda\Pi/S$ context Γ and Γ-term M, there is a $\lambda\Pi^-/S$ term M^- such that $M \longrightarrow_\beta^* M^-$.*

Proof. Reducing a Kind-level β-redex $(\lambda x : A.\, B)\, N$ does not create other Kind-level β-redexes because N is at the level of Type. Indeed, in the $\lambda\Pi$ calculus modulo rewriting the only Kind rule is (Type, Kind, Kind). Therefore $N : A :$ Type. If N reduces to a λ-abstraction then the only redexes it can create are at the level of Type. Therefore, the number of Kind-level β-redexes strictly decreases, so any Kind-level β-reduction strategy will terminate. ◀

▶ **Example 29.** The term

$$I_1 = \lambda\alpha{:}u_{\mathsf{Type}}.\, \lambda x{:}\varepsilon_{\mathsf{Type}}\, ((\lambda\beta{:}u_{\mathsf{Type}}.\, \beta)\, \alpha).\, x$$

is in $\lambda\Pi^-/HOL$. The term

$$I_2 = \lambda\alpha{:}u_{\mathsf{Type}}.\, \lambda x{:}((\lambda\beta{:}u_{\mathsf{Type}}.\, \varepsilon_{\mathsf{Type}}\, \beta)\, \alpha).\, x$$

is not in $\lambda\Pi^-/HOL$ but

$$I_2 \longrightarrow_\beta \lambda\alpha{:}u_{\mathsf{Type}}.\, \lambda x{:}\varepsilon_{Type}\alpha.\, x$$

which is in $\lambda\Pi^-/HOL$.

5.2 Minimal completion

To simplify our reducibility proof in the next section, we will translate $\lambda\Pi/S$ back to a pure type system, but since it cannot be λS we will define a slightly larger PTS called λS^* that contains λS and that will be easier to manipulate than $\lambda\Pi/S$.

The reason we need a larger PTS is that we have types that do not have a type, such as top-sorts because there is no associated axiom. Similarly, we can sometimes prove $\Gamma, x : A \vdash_{\lambda S} M : B$ but cannot abstract over x because there is no associated product rule. Completions of pure type systems were originally introduced by Severi [17, 16] to address these issues by injecting λS into a larger pure type system.

▶ **Definition 30** (Completion [16])**.** A specification $S' = (\mathcal{S}', \mathcal{A}', \mathcal{R}')$ is a *completion of* S if

1. $\mathcal{S} \subseteq \mathcal{S}'$,$\mathcal{A} \subseteq \mathcal{A}'$, $\mathcal{R} \subseteq \mathcal{R}'$, and
2. for all sorts $s_1 \in \mathcal{S}$, there is a sort $s_2 \in \mathcal{S}'$ such that $(s_1 : s_2) \in \mathcal{A}'$, and
3. for all sorts $s_1, s_2 \in \mathcal{S}'$, there is a sort $s_3 \in \mathcal{S}'$ such that $(s_1, s_2, s_3) \in \mathcal{R}'$.

Notice that all the top-sorts of λS are typable in $\lambda S'$ and that $\lambda S'$ is full, meaning that all products are typable. These two properties reflect exactly the discrepancy between λS and $\lambda\Pi^-/S$. Not all completions are conservative though, so we define the following completion.

▶ **Definition 31** (Minimal completion)**.** We define the *minimal completion of* S, written S^*, to be the following specification:

$$\begin{aligned}
\mathcal{S}^* &= \mathcal{S} \cup \{\tau\} \\
\mathcal{A}^* &= \mathcal{A} \cup \{(s_1 : \tau) \mid s_1 \in \mathcal{S}, \nexists s2, (s_1 : s_2) \in \mathcal{A}\} \\
\mathcal{R}^* &= \mathcal{R} \cup \{(s_1, s_2, \tau) \mid s_1, s_2 \in \mathcal{S}^*, \nexists s_3, (s_1, s_2, s_3) \in \mathcal{R}\}
\end{aligned}$$

where $\tau \notin \mathcal{S}$.

We add a new top-sort τ and axioms $s : \tau$ for all previous top-sorts s, and complete the rules to obtain a PTS full. The new system is a completion by Definition 30 and it is minimal in the sense that we generically added the smallest number of sorts, axioms, and rules so that the result is guaranteed to be conservative. Any well-typed term of λS is also well-typed in λS^*, but just like $\lambda\Pi^-/S$, this system allows more functions than λS.

▶ **Example 32.** The polymorphic identity function is well-typed in λHOL^*.

$$\vdash_{\lambda HOL^*} I : \Pi\alpha\!:\!\mathsf{Type}.\, \alpha \to \alpha$$

$$\vdash_{\lambda HOL^*} \Pi\alpha\!:\!\mathsf{Type}.\, \alpha \to \alpha : \tau$$

Next, we define inverse translations that translate the terms and types of $\lambda\Pi^-/S$ to the terms and types of λS^*.

▶ **Definition 33** (Inverse translations)**.** The inverse translation of terms $\varphi(M)$ and the inverse translation of types $\psi(A)$ are mutually defined as follows.

$$\begin{array}{rcl}
\varphi(\dot{s}) & = & s \\
\varphi(\dot{\pi}_{s_1 s_2 s_3}) & = & \lambda\alpha\!:\!s_1.\, \lambda\beta\!:\!(\alpha \to s_2).\, \Pi x\!:\!\alpha.\, \beta\, x \\
\varphi(x) & = & x \\
\varphi(M\, N) & = & \varphi(M)\, \varphi(N) \\
\varphi(\lambda x\!:\!A.\, M) & = & \lambda x\!:\!\psi(A).\, \varphi(M)
\end{array}$$

$$\begin{array}{rcl}
\psi(u_s) & = & s \\
\psi(\varepsilon_s\, M) & = & \varphi(M) \\
\psi(\Pi x\!:\!A.\, B) & = & \Pi x\!:\!\psi(A).\, \psi(B)
\end{array}$$

Note that this is only a partial definition, but it is total for $\lambda\Pi^-/S$ terms. In particular, it is an inverse of the forward translation in the following sense.

▶ **Lemma 34.** *For any* Γ*-term* M *and* Γ*-type* A*,*
1. $\varphi(|M|_\Gamma) \equiv_\beta M$*,*
2. $\psi(\|A\|_\Gamma) \equiv_\beta A$*.*

Proof. By induction on M or A. We show the product case where $M = \Pi x : A.\, B$. By induction hypothesis, $\varphi(|A|) \equiv_\beta A$ and $\varphi(|B|) \equiv_\beta B$. Therefore

$$\begin{array}{rcl}
\varphi(|M|) & = & (\lambda\alpha.\, \lambda\beta.\, \Pi x\!:\!\alpha.\, \beta\, x)\ \varphi(|A|)\, (\lambda x.\, \varphi(|B|)) \\
 & \longrightarrow_\beta^* & \Pi x\!:\!\varphi(|A|).\, \varphi(|B|) \\
 & \equiv_\beta & \Pi x\!:\!A.\, B
\end{array}$$

◀

Next we show that the inverse translations preserve typing.

▶ **Lemma 35.**
1. $\varphi(M[x\backslash N]) = \varphi(M)[x\backslash\varphi(N)]$
2. $\psi(A[x\backslash N]) = \psi(A)[x\backslash\varphi(N)]$

Proof. By induction on M or A. We show the product case $A = \Pi y\!:\!B.\, C$. Without loss of generality, $y \neq x$ and $y \notin N$ and $y \notin \varphi(N)$. Then $\Pi y\!:\!B.\, C[x\backslash N] = \Pi y\!:\!B[x\backslash N].\, C[x\backslash N]$.

By induction hypothesis, $\psi(B[x\backslash N]) = \psi(B)[x\backslash\varphi(N)]$ and $\psi(C[x\backslash N]) = \psi(C)[x\backslash\varphi(N)]$. Therefore

$$\begin{array}{rcl}\psi(A[x\backslash N]) & = & \Pi y{:}\psi(B)[x\backslash\varphi(N)].\,\psi(C)[x\backslash\varphi(N)] \\ & = & \Pi x{:}\psi(B).\,\psi(C)[x\backslash\varphi(N)] \\ & = & \psi(\Pi x{:}B.\,C)[x\backslash\varphi(N)]\end{array}$$

◀

▶ **Lemma 36.**
1. *If* $M \longrightarrow_{\beta R} N$ *then* $\varphi(M) \longrightarrow_\beta^* \varphi(N)$
2. *If* $A \longrightarrow_{\beta R} B$ *then* $\psi(A) \longrightarrow_\beta^* \psi(B)$

Proof. By induction on M or A. We show the base cases.
- Case $M = (\lambda x{:}A_1.\,M_1)\,N_1$, $N = M_1[x\backslash N_1]$. Then $\varphi(M) = (\lambda x{:}\psi(A_1).\,\varphi(M_1))\,\varphi(N_1)$. Therefore $\varphi(M) \longrightarrow_\beta \varphi(M_1)[x\backslash\varphi(N_1)]$ which is equal to $\varphi(M_1[x\backslash N_1])$ by Lemma 35.
- Case $A = \varepsilon_s\,\dot{s}$, $B = u_s$. Then $\psi(A) = s = \psi(B)$.
- Case $A = \varepsilon_{s_1}\,(\dot{\pi}_{s_1s_2s_3}\,A_1\,B_1)$, $B = \Pi x{:}\varepsilon_{s_1}\,A_1.\,\varepsilon_{s_2}\,(B_1\,x)$. Then

$$\begin{array}{rcl}\psi(A) & = & (\lambda\alpha.\,\lambda\beta.\,\Pi x{:}\alpha.\,\beta\,x)\;\varphi(A_1)\,\varphi(B_1) \\ & \longrightarrow_\beta^* & \Pi x{:}\varphi(A_1).\,\varphi(B_1)\,x \\ & = & \psi(\Pi x{:}A_1.\,B_1\,x)\end{array}$$

◀

▶ **Lemma 37.**
1. *If* $M \equiv_{\beta R} N$ *then* $\varphi(M) \equiv_\beta \varphi(N)$
2. *If* $A \equiv_{\beta R} B$ *then* $\psi(A) \equiv_\beta \psi(B)$

Proof. Follows from Lemma 36. ◀

Because the forward translation of contexts does not introduce any type variable, we define the following restriction on contexts.

▶ **Definition 38** (Object context)**.** We say that Γ is an *object context* if $\Gamma \vdash_{\lambda\Pi/S} A : \mathsf{Type}$ for all $x : A \in \Gamma$. If $\Gamma = (x_1 : A_1, \ldots, x_n : A_n)$ is an object context, we define $\psi(\Gamma)$ as $(x_1 : \psi(A_1), \ldots, x_n : \psi(A_n))$.

▶ **Lemma 39.** *For any* $\lambda\Pi^-/S$ *object context* Γ *and terms* M, A*:*
1. *If* $\mathrm{WF}_{\lambda\Pi/S}(\Gamma)$ *then* $\mathrm{WF}_{\lambda S^*}(\psi(\Gamma))$.
2. *If* $\Gamma \vdash_{\lambda\Pi/S} M : A : \mathsf{Type}$ *then* $\psi(\Gamma) \vdash_{\lambda S^*} \varphi(M) : \psi(A)$.
3. *If* $\Gamma \vdash_{\lambda\Pi/S} A : \mathsf{Type}$ *then* $\psi(\Gamma) \vdash_{\lambda S^*} \psi(A) : s$ *for some sort* $s \in \mathcal{S}^*$.

Proof. By induction on the derivation. The details of the proof can be found in the long version of this paper. ◀

5.3 Reduction to λS

In order to show that λS^* is a conservative extension of λS, we prove that β-reduction at the level of τ terminates. A straightforward proof by induction would fail because contracting a τ-level β-redex can create other such redexes. To solve this, we adapt Tait's *reducibility method* [18]. The idea is to strengthen the induction hypothesis of the proof by defining a predicate by induction on the type of the term.

▶ **Definition 40.** The predicate $\Gamma \models_S M : A$ is defined as $\mathrm{WF}_{\lambda S}(\Gamma)$ and $\Gamma \vdash_{\lambda S^*} M : A : s$ for some sort s and:

- if $s \neq \tau$ or $A = s'$ for some $s' \in \mathcal{S}$ then $\Gamma \models_S M : A$ iff $M \longrightarrow_\beta^* M'$ and $A \longrightarrow_\beta^* A'$ for some M', A' such that $\Gamma \vdash_{\lambda S} M' : A'$,
- if $s = \tau$ and $A = \Pi x : B.\, C$ for some B, C then $\Gamma \models_S M : A$ iff for all N such that $\Gamma \models_S N : B$, $\Gamma \models_S M\, N : C[x \backslash N]$.

Note that recursive definition covers all cases thanks to Theorem 8. To show that it is well-founded, we define the following measure of A.

▶ **Definition 41.** If $\mathrm{WF}_{\lambda S}(\Gamma)$ and $\Gamma \vdash_{\lambda S^*} A : s$ then $\mathcal{H}_\tau(A)$ is defined as:

$$\begin{array}{rcll} \mathcal{H}_\tau(A) & = & 0 & \text{if } s \neq \tau \\ \mathcal{H}_\tau(s') & = & 0 & \text{if } s = \tau \\ \mathcal{H}_\tau(\Pi x : B.\, C) & = & 1 + max(\mathcal{H}_\tau(B) + \mathcal{H}_\tau(C)) & \text{if } s = \tau \end{array}$$

▶ **Lemma 42.** *If $\Gamma, x : B \vdash_{\lambda S^*} C : \tau$ and $\Gamma \vdash_{\lambda S^*} N : B$ then $\mathcal{H}_\tau(C[x \backslash N]) = \mathcal{H}_\tau(C)$.*

Proof. By induction on C. ◀

▶ **Corollary 43.** *Definition 40 is well-founded.*

Proof. The measure $\mathcal{H}_\tau(A)$ strictly decreases in the definition. ◀

The predicate we defined is compatible with β-equivalence.

▶ **Lemma 44.** *If $\Gamma \models_S M : A$ and $\Gamma \vdash_{\lambda S^*} M' : A$ and $M \equiv_\beta M'$ then $\Gamma \models_S M' : A$.*

Proof. By induction on the height of A.

- If $s \neq \tau$ or $A = s'$ for some $s' \in \mathcal{S}$ then $M \longrightarrow_\beta^* M''$ and $A \longrightarrow_\beta^* A'$ for some M'', A' such that $\Gamma \vdash_{\lambda S} M'' : A'$. By confluence and subject reduction, $M' \longrightarrow_\beta^* M'''$ such that $\Gamma \vdash_{\lambda S} M''' : A'$.
- If $s = \tau$ and $A = \Pi x : B.\, C$ for some B, C then for all N such that $\Gamma \models_S N : B$, $\Gamma \models_S M\, N : C[x \backslash N]$. By induction hypothesis, $\Gamma \models_S M'\, N : C[x \backslash N]$. Therefore $\Gamma \models_S M' : \Pi x : B.\, C$. ◀

▶ **Lemma 45.** *If $\Gamma \models_S M : A$ and $\Gamma \vdash_{\lambda S^*} A' : s$ and $A \equiv_\beta A'$ then $\Gamma \models_S M : A'$.*

Proof. By induction on the height of A.

- If $s \neq \tau$ or $A = s'$ for some $s' \in \mathcal{S}$ then $M \longrightarrow_\beta^* M'$ and $A \longrightarrow_\beta^* A''$ for some M', A'' such that $\Gamma \vdash_{\lambda S} M' : A''$. By conversion, $\Gamma \vdash_{\lambda S^*} M : A'$, so by subject reduction $\Gamma \vdash_{\lambda S^*} M' : A'$. By confluence, subject reduction, and conversion, $A' \longrightarrow_\beta^* A'''$ such that $\Gamma \vdash_{\lambda S} M' : A'''$.
- If $s = \tau$ and $A = \Pi x : B.\, C$ for some B, C then for all N such that $\Gamma \models_S N : B$, $\Gamma \models_S M\, N : C[x \backslash N]$. By product compatibility, $A' = \Pi x : B'.\, C'$ such that $B \equiv_\beta B'$ and $C \equiv_\beta C'$. By induction hypothesis, $\Gamma \models_S M\, N : C'[x \backslash N]$. Therefore $\Gamma \models_S M : \Pi x : B'.\, C'$. ◀

We extend the definition of the inductive predicate to contexts and substitutions before proving the main general lemma.

▶ **Definition 46.** If $\mathrm{WF}_{\lambda S^*}(\Gamma)$, $\mathrm{WF}_{\lambda S}(\Gamma')$, and σ is a substitution for the variables of Γ, then $\Gamma' \models_S \sigma : \Gamma$ when $\Gamma' \models_S \sigma(x) : \sigma(A)$ for all $(x : A) \in \Gamma$.

▶ **Lemma 47.** *If* $\Gamma \vdash_{\lambda S^*} M : A : s$ *then for any context* Γ' *and substitution* σ *such that* $\mathrm{WF}_{\lambda S}(\Gamma')$ *and* $\Gamma' \models_S \sigma : \Gamma$, $\Gamma' \models_S \sigma(M) : \sigma(A)$.

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda S^*} M : A$. The details of the proof can be found in the long version of this paper. ◀

▶ **Corollary 48.** *Suppose* $\mathrm{WF}_{\lambda S}(\Gamma)$ *and either* $\Gamma \vdash_{\lambda S} A : s$ *or* $A = s$ *for some* $s \in \mathcal{S}$. *If* $\Gamma \vdash_{\lambda S^*} M : A$ *then* $M \longrightarrow_\beta^* M'$ *such that* $\Gamma \vdash_{\lambda S} M' : A$.

Proof. Taking σ as the identity substitution, there are terms M' and A' such that $M \longrightarrow_\beta^* M'$ and $A \longrightarrow_\beta^* A'$ and $\Gamma \vdash_{\lambda S} M' : A'$. If $A = s \in S$ then $A' = s$ and we are done. Otherwise by conversion we get $\Gamma \vdash_{\lambda S} M' : A$. ◀

We now have all the tools to prove the main theorem.

▶ **Theorem 49** (Conservativity). *For any* Γ*-type* A *of* λS, *if there is a term* N *such that* $\|\Gamma\| \vdash_{\lambda\Pi/S} N : \|A\|_\Gamma$ *then there is a term* M *such that* $\Gamma \vdash_{\lambda S} M : A$.

Proof. By Lemma 28, there is a $\lambda\Pi^-/S$ term N^- such that $N \longrightarrow_\beta^* N^-$. By subject reduction, $\|\Gamma\| \vdash_{\lambda\Pi/S} N^- : \|A\|_\Gamma$. By Lemmas 39 and 34, $\Gamma \vdash_{\lambda S^*} \varphi(N^-) : A$. By Corollary 48, there is a term M such that $\varphi(N^-) \longrightarrow_\beta^* M$ and $\Gamma \vdash_{\lambda S} M : A$. ◀

6 Conclusion

We have shown that $\lambda\Pi/S$ is conservative even when λS is not normalizing. Even though $\lambda\Pi/S$ can construct more functions than λS, it preserves the semantics of λS. This effect is similar to various conservative extensions of pure type systems such as *pure type systems with definitions* [17], *pure type systems without the* Π*-condition* [16], or *predicative (ML) polymorphism* [15]. Inconsistency in pure type systems usually does not come from the ability to type more functions, but from the possible impredicativity caused by assigning a sort to the type of these functions. It is clear that no such effect arises in $\lambda\Pi/S$ because there is no constant $\dot{\pi}_{s_1 s_2 s_3}$ associated to the type of illegal abstractions.

One could ask whether the techniques we used are adequate. While the construction of λS^* is not absolutely necessary, we feel that it simplifies the proof and that it helps us better understand the behavior of $\lambda\Pi/S$ by reflecting it back into a pure type system. The relative normalization steps of Section 5.3 correspond to the normalization of a simply typed λ calculus. Therefore, it is not surprising that we had to use Tait's reducibility method. However, our proof can be simplified in some cases. A PTS is *complete* when it is a completion of itself. In that case, the construction of S^* is unnecessary. The translations $\varphi(M)$ and $\psi(A)$ translate directly into λS, and Section 5.3 can be omitted. This is the case for example for the calculus of constructions with *infinite type hierarchy* (λC^∞) [17], which is the basis for proof assistants such as Coq and Matita.

The results of this paper can be extended in several directions. They could be adapted to show the conservativity of other embeddings, such as that of the *calculus of inductive constructions* (CIC) [4]. They also indirectly imply that $\lambda\Pi/S$ is weakly normalizing when λS is weakly normalizing because the image of a λS term is normalizing [6]. The strong normalization of $\lambda\Pi/S$ when λS is strongly normalizing is still an open problem. The Barendregt-Geuvers-Klop conjecture states that any weakly normalizing PTS is also strongly normalizing [8]. There is evidence that this conjecture is true [2], in which case we hope that its proof could be adapted to prove the strong normalization of $\lambda\Pi/S$. Weak normalization could also be used as an intermediary step for constructing models by induction on types in order to prove strong normalization.

Acknowledgments. We thank Gilles Dowek and Guillaume Burel for their support and feedback, as well as Frédéric Blanqui, Raphaël Cauderlier, and the various anonymous referees for their comments and suggestions on previous versions of this paper.

References

1 H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
2 Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1-2):317–361, 2001.
3 Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(01):37–92, 2005.
4 M. Boespflug and G. Burel. CoqInE: translating the calculus of inductive constructions into the λΠ-calculus modulo. In *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, pages 44–50, 2012.
5 M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-calculus modulo as a universal proof language. In *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, pages 28–43, 2012.
6 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, number 4583 in Lecture Notes in Computer Science, pages 102–117. Springer Berlin Heidelberg, 2007.
7 Gilles Dowek. Models and termination of proof-reduction in the λΠ-calculus modulo theory. arXiv:1501.06522, hal-01101834, 2014.
8 Herman Geuvers. *Logics and type systems*. PhD thesis, University of Nijmegen, 1993.
9 Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat, Université Paris VII, 1972.
10 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.
11 Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., New York, NY, USA, 1994.
12 Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, 1984.
13 Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.
14 Erik Palmgren. On universes in type theory. In *Twenty-five years of constructive type theory*, pages 191–204. Oxford University Press, 1998.
15 Cody Roux and Floris van Doorn. The structural theory of pure type systems. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi*, number 8560 in Lecture Notes in Computer Science, pages 364–378. Springer International Publishing, 2014.
16 Paula Severi. Pure type systems without the Pi-condition. *Proceedings of 7th Nordic Workshop on Programming Theory*, 1995.
17 Paula Severi and Erik Poll. Pure type systems with definitions. In Anil Nerode and Yu V. Matiyasevich, editors, *Logical Foundations of Computer Science*, number 813 in Lecture Notes in Computer Science, pages 316–328. Springer Berlin Heidelberg, 1994.
18 W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

Models for Polymorphism over Physical Dimensions

Robert Atkey[1], Neil Ghani[1], Fredrik Nordvall Forsberg[1], Timothy Revell[1], and Sam Staton[2]

1 **University of Strathclyde, UK**
2 **University of Oxford, UK**

Abstract

We provide a categorical framework for models of a type theory that has special types for physical quantities. The types are indexed by the physical dimensions that they involve. Fibrations are used to organize this index structure in the models of the type theory. We develop some informative models of this type theory: firstly, a model based on group actions, which captures invariance under scaling, and secondly, a way of constructing new models using relational parametricity.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Category Theory, Units of Measure, Dimension Types, Type Theory

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.45

1 Introduction

This paper is about semantic models of programs that manipulate physical quantities. Physical quantities are organized into dimensions, such as *Length* or *Time*. A fundamental principle is that it is not meaningful to add or compare quantities of different dimensions, but they can be multiplied. To measure a physical quantity we use units, such as metres for length and seconds for time. We understand these units as chosen constant quantities of given dimensions.

Here is a simple polymorphic program that is defined for all dimensions; it takes a quantity x of a given dimension X, and returns its double, which has the same dimension.

$$f \coloneqq (\Lambda X.\, \lambda x : \mathsf{Quantity}(X).\, x + x) : \ \forall X.\, \mathsf{Quantity}(X) \to \mathsf{Quantity}(X) \tag{1}$$

To illustrate, we can use the polymorphic function f to double a length of 5 metres.

$$f_{\mathit{Length}}\,(5\mathrm{m}) = 10\mathrm{m} : \mathsf{Quantity}(\mathit{Length}) \tag{2}$$

There are a few key points that are worth emphasising about examples (1) and (2) above:

- There are two kinds of variable, X and x. The first variable X stands for a dimension whereas x stands for an inhabitant of a type. To emphasise this distinction, we use different abstraction symbols (λ and Λ) for the two kinds of variable.
- The type $\mathsf{Quantity}(X)$ depends on a dimension X, and it is inhabited by quantities of that dimension. For example, the standard unit of measurement for length, the metre, is a quantity of that dimension, i.e. a constant $\mathrm{m} : \mathsf{Quantity}(\mathit{Length})$.

Several authors have developed programming languages with type systems that support physical quantities [8, 12, 17, 10, 6]. The type systems are often motivated as static analyses that help to prevent disasters by accommodating dimensions. For example, the Mars Climate Orbiter was lost as a result of a unit mismatch in the software [13].

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 45–59

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our starting point in this paper is the work of Kennedy [10] who developed techniques for reasoning about these kinds of programs. However, we take a different approach by developing a general categorical notion of model for a programming language of this form, and by developing ways of building models. The main contributions of this paper are:

1. We provide a general notion of a model for a programming language with physical dimension types by introducing the concept of a λD-*model* (Section 3). The basic idea is that for each context of dimension variables, there is a model of the simply typed λ-calculus extended with types of quantities of the dimensions definable in the context ($\mathsf{Quantity}(D)$ etc.). Moreover these models of the simply typed λ-calculus are related by substituting for dimension variables, and this also defines a universal property for polymorphic quantification over dimension variables.
2. An important example of a λD-model is built from group actions (Example 9). A difficulty with set-theoretic models of dimension polymorphism is as follows: how does one understand $\mathsf{Quantity}(X)$ as a set, if the dimension X is not specified and we have no fixed units of measure for X? We resolve this by interpreting $\mathsf{Quantity}(X)$ as the set of magnitudes, i.e. positive real numbers, thought of as quantities of some unspecified unit of measure, but then by equipping $\mathsf{Quantity}(X)$ with an action of the scaling group, to explain how to change the units of measure. We can then ask that any function $\mathsf{Quantity}(X) \to \mathsf{Quantity}(X)$ is invariant under changing that unspecified unit of measure, more precisely, invariant under scaling.
3. We show how the λD-model built from group actions supports a diverse range of parametricity-like theorems, without the need to define a separate relational semantics (Section 4). This results in simple proofs of theorems that would otherwise require more heavy machinery.
4. We explore the relationship between the parametricity-like theorems of the λD-model built from group actions, and a natural notion of a relational model (Section 5). Formally, we show that when interpreting the syntax these two notions coincide.

2 Types with Physical Dimensions

We begin by recalling a simple type theory, which we call λD, indexed by dimensions based on Kennedy's work [10]. Within this type theory we can express programs such as (1) and (2). Since there are two kinds of variable, we have two kinds of context.

Dimensions and Dimension Contexts. A dimension context Δ is a finite list of distinct dimension variables. A dimension-expression-in-context $\Delta \vdash D\ \mathsf{dim}$ is a monomial D in the variables Δ. More precisely, if $\Delta = X_1, \ldots, X_n$ and $k_i \in \mathbb{Z}$ then $\Delta \vdash X_1^{k_1} \ldots X_n^{k_n}\ \mathsf{dim}$. We can make the set $\{D \mid \Delta \vdash D\ \mathsf{dim}\}$ an Abelian group under addition of exponents, and indeed this is the free Abelian group on Δ. This universal property gives a notion of substitution on dimension expressions. For example, $X, Z \vdash (X^2Y^3)[{}^{(XZ^2)}/_Y] = X^5Z^6\ \mathsf{dim}$.

Types. Well-formed types are given by judgements of the form $\Delta \vdash T\ \mathsf{type}$ where Δ is a dimension context. The judgements are generated by the following rules.

$$\frac{\Delta \vdash D\ \mathsf{dim}}{\Delta \vdash \mathsf{Quantity}(D)\ \mathsf{type}} \qquad \frac{\Delta, X \vdash T\ \mathsf{type}}{\Delta \vdash \forall X.T\ \mathsf{type}} \qquad \frac{\Delta \vdash T\ \mathsf{type} \quad \Delta \vdash U\ \mathsf{type}}{\Delta \vdash T \to U\ \mathsf{type}}$$

$$\frac{}{\Delta \vdash 1\ \mathsf{type}} \qquad \frac{\Delta \vdash T\ \mathsf{type} \quad \Delta \vdash U\ \mathsf{type}}{\Delta \vdash T \times U\ \mathsf{type}} \qquad \frac{}{\Delta \vdash 0\ \mathsf{type}} \qquad \frac{\Delta \vdash T\ \mathsf{type} \quad \Delta \vdash U\ \mathsf{type}}{\Delta \vdash T + U\ \mathsf{type}}$$

Notice that we do not have System-F-style polymorphism, but instead dimension polymorphism: types can be parameterised by dimensions, but they cannot be parameterised by types, since we do no have type variables. From the Curry-Howard perspective this is a first-order-logic where the domain of discourse is the theory of Abelian groups and where there is a single atomic predicate, $\mathsf{Quantity}$.

Terms and Typing Contexts. Well-formed typing contexts are given by judgements $\Delta \vdash \Gamma\ \mathsf{ctx}$ where Δ is a dimension context, Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$ and there is a well-formed typing judgement $\Delta \vdash T_i\ \mathsf{type}$ for every i. Well-formed terms are given by judgements $\Delta; \Gamma \vdash t : T$ where there is a well-formed typing context $\Delta \vdash \Gamma\ \mathsf{ctx}$ and a well-formed type $\Delta \vdash T\ \mathsf{type}$. The rules for the type formers 1, $\_ \times \_$, 0, $\_ + \_$ and $\_ \to \_$ are the usual ones from simply typed λ-calculus.

$$\frac{\Delta \vdash \Gamma, \Gamma'\ \mathsf{ctx} \quad \Delta \vdash T\ \mathsf{type}}{\Delta; \Gamma, x : T, \Gamma' \vdash x : T} \qquad \frac{\Delta \vdash \Gamma\ \mathsf{ctx} \quad (\mathrm{op} : T) \in \mathrm{Ops}}{\Delta; \Gamma \vdash \mathrm{op} : T}$$

$$\frac{\Delta; \Gamma, x : T \vdash t : U}{\Delta; \Gamma \vdash \lambda x.t : T \to U} \qquad \frac{\Delta; \Gamma \vdash t : T \to U \quad \Delta; \Gamma \vdash u : T}{\Delta; \Gamma \vdash t\, u : U}$$

$$\frac{\Delta \vdash \Gamma\ \mathsf{ctx}}{\Delta; \Gamma \vdash () : 1} \qquad \frac{\Delta; \Gamma \vdash t_1 : T_1 \qquad \Delta; \Gamma \vdash t_2 : T_2}{\Delta; \Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \frac{\Delta; \Gamma \vdash t : T_1 \times T_2}{\Delta; \Gamma \vdash \mathsf{pr}_i(t) : T_i}$$

$$\frac{\Delta; \Gamma \vdash t : 0 \quad \Delta \vdash T\ \mathsf{type}}{\Delta; \Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \{\} : T} \qquad \frac{\Delta; \Gamma \vdash t : T_i}{\Delta; \Gamma \vdash \mathsf{inj}_i\, t : T_1 + T_2}$$

$$\frac{\Delta; \Gamma \vdash t : T_1 + T_2 \quad \left(\Delta; \Gamma, x_i : T_i \vdash u_i : U\right)_{i \in \{1,2\}}}{\Delta; \Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \{\mathsf{inj}_1\, x_1 \mapsto u_1; \mathsf{inj}_2\, x_2 \mapsto u_2\} : U}$$

In addition, we have the introduction and elimination rules for quantification over a dimension variable.

$$\frac{\Delta, X; \Gamma \vdash t : T}{\Delta; \Gamma \vdash \Lambda X.t : \forall X.T} \qquad \frac{\Delta \vdash D\ \mathsf{dim} \quad \Delta; \Gamma \vdash t : \forall X.T}{\Delta; \Gamma \vdash t_D : T[D/X]}$$

We use Bool as an abbreviation for $1 + 1$. Our calculus is parameterised by a collection Ops of primitive operation typings $(\mathrm{op} : T_{\mathrm{op}})$ where for each primitive operation $\mathrm{op} : T_{\mathrm{op}}$, its type T_{op} is closed (i.e., $\vdash T_{\mathrm{op}}\ \mathsf{type}$). An example set of primitive operations includes dimension-polymorphic arithmetic and test operations on quantities:

$$\begin{aligned}\mathrm{Ops} = (&+ : \forall X.\mathsf{Quantity}(X) \times \mathsf{Quantity}(X) \to \mathsf{Quantity}(X),\\ &\times : \forall X_1.\forall X_2.\mathsf{Quantity}(X_1) \times \mathsf{Quantity}(X_2) \to \mathsf{Quantity}(X_1 \cdot X_2),\ 1 : \mathsf{Quantity}(1),\\ &\mathsf{inv} : \forall X.\mathsf{Quantity}(X) \to \mathsf{Quantity}(X^{-1}),\\ &< : \forall X.\mathsf{Quantity}(X) \times \mathsf{Quantity}(X) \to \mathsf{Bool}).\end{aligned}$$

One could also define a type of signed/zero quantities $\mathsf{Real}(X) := \mathsf{Quantity}(X) + 1 + \mathsf{Quantity}(X)$, and then extend the language with further arithmetic term constants such as signed addition $+ : \forall X.\mathsf{Real}(X) \times \mathsf{Real}(X) \to \mathsf{Real}(X)$.

To write terms that make use of common sets of dimensions and units, we judge terms in a context $(\Delta_{dim}, \Gamma_{units})$. For example, $\Delta_{dim} = (\mathit{Length}, \mathit{Time})$ and $\Gamma_{units} = (\mathrm{m} : \mathsf{Quantity}(\mathit{Length}), \mathrm{ft} : \mathsf{Quantity}(\mathit{Length}), s : \mathsf{Quantity}(\mathit{Time}))$.

3 Categorical Semantics of Dimension Types

Next up we give a general categorical semantics for the λD type theory. Central to this is the notion of a $\lambda\forall$-fibration.

▶ **Definition 1.** A $\lambda\forall$-*fibration* is a bicartesian closed fibration with simple products.

It is well-known that $\lambda\forall$-fibrations give a categorical model of the fragment of first-order logic without existential quantifiers. Nevertheless, we briefly introduce the basic notions now, since they are central to our development. We refer to Jacobs [9] for the full details.

A fibration $p : \mathcal{E} \to \mathcal{B}$ is a functor between categories satisfying certain conditions. These conditions (along with the structure in Definition 2) allow us to model the λD type theory. The basic idea is that dimension contexts will be interpreted as objects in a category $\mathcal{B}$. For each $B \in \mathcal{B}$, we consider the fibre $\mathcal{E}_B$, i.e. the subcategory of $\mathcal{E}$ with objects $E \in \mathcal{E}$ for which $p(E) = B$. The objects of $\mathcal{E}_B$ will be used to interpret types in dimension context B, and the morphisms in $\mathcal{E}_B$ will be used to interpret terms. We can substitute dimension expressions for dimension variables, and this substitution will be interpreted using morphisms in $\mathcal{B}$. Since p is a fibration, one can form a reindexing functor $f^* : \mathcal{E}_{B'} \to \mathcal{E}_B$ for each morphism $f : B \to B'$ in $\mathcal{B}$, which describes substitution for dimension variables in types and terms.

A fibration is said to be bicartesian closed if $\mathcal{E}_B$ is a Cartesian closed category with coproducts for all B, and each reindexing functor $f^* : \mathcal{E}_{B'} \to \mathcal{E}_B$ preserves products, exponentials and coproducts. This bicartesian closed structure is needed to interpret the product, function and coproduct types.

Concatenation of dimension contexts will be interpreted using products in the category $\mathcal{B}$. The reindexing functors $\pi^* : \mathcal{E}_B \to \mathcal{E}_{B\times B'}$ for the product projections $\pi : B \times B' \to B$ correspond to context-weakening. A fibration $p : \mathcal{E} \to \mathcal{B}$ is said to have simple products if $\mathcal{B}$ has products and the reindexing functors for the product projections have right adjoints $\forall : E_{B\times B'} \to E_B$ that are compatible with reindexing ('Beck-Chevalley'). A fibration is said to have products if this condition holds for all morphisms in the base, not just projections. These right adjoints are needed to interpret universal quantification of dimension variables in types.

▶ **Definition 2.** A λD-model (p, G, Q) is a $\lambda\forall$-fibration $p : \mathcal{E} \to \mathcal{B}$, an Abelian group object G in $\mathcal{B}$, and an object Q in the fibre $\mathcal{E}_G$.

Recall that an Abelian group object in a category $\mathcal{B}$ with products is given by an object G together with maps $e : 1 \to G$, $m : G \times G \to G$ and $i : G \to G$ satisfying the laws of Abelian groups. This group structure is needed to interpret dimension expressions: for each vector of n integers we have a morphism $G^n \to G$.

An equivalent way to define Abelian group objects if $\mathcal{B}$ has chosen products is as follows. Recall that the Lawvere theory for Abelian groups is the category L_{Ab} whose objects are natural numbers, and where a morphism $m \to n$ is an $m \times n$ matrix of integers. Composition of morphisms is given by matrix multiplication, and categorical products are given by arithmetic addition of natural numbers. An Abelian group object in $\mathcal{B}$ is an object G of $\mathcal{B}$ together with a strictly-product-preserving functor $F : \mathsf{L}_{\mathsf{Ab}} \to \mathcal{B}$ such that $F(1) = G$.

We remark that the object Q in a λD-model is analogous to the generic object in a model of System F.

In order to ascertain the value of Definition 2, we now do three things: i) we show that a λD-model in fact does provide categorical models of dimension types, ii) we give examples of λD-models, and iii) we prove theorems that show the viability of reasoning at this level of abstraction.

3.1 Modelling Dimension Types

To show that λD-models provide a categorical semantics for dimension types, we must show how to interpret the syntax given in Section 2 in any given λD-model. We will use the $\lambda\forall$-fibration to separate the indexing information (the dimensions) from the indexed information (the types and terms). This means that the base category of the fibration will be used to interpret dimension contexts, and types and terms will be interpreted as objects and morphisms in the fibres above the dimension contexts in which they are defined. Bicartesian closure of the fibres will allow us to inductively interpret types built from $\mathsf{1}$, $\times$, $\mathsf{0}$, $+$ and $\to$, and we will take the standard approach in categorical logic to interpret quantification of dimensions by using right adjoints. Finally, since dimension expressions for a dimension context are defined as elements of the free Abelian group on that dimension context, we will use the Abelian group object structure to interpret such expressions. Formally, we interpret the syntax as follows.

- Dimension contexts $\Delta = X_1, \ldots, X_n$ are interpreted as the product of the Abelian group object $[\![\Delta]\!] = G^n$ in $\mathcal{B}$.
- Dimension expressions $\Delta \vdash D\ \mathsf{dim}$ are interpreted as morphisms $G^n \to G$ in the base $\mathcal{B}$, by using the structure of the Abelian group object G. For example, $[\![X, Y \vdash X \cdot Y^{-1}]\!] = G \times G \xrightarrow{\mathrm{id}_G \times i} G \times G \xrightarrow{m} G$.
- Well-formed types $\Delta \vdash T\ \mathsf{type}$ are interpreted as objects $[\![T]\!]$ in the fibre above $[\![\Delta]\!]$, defined by induction on the structure of T. We interpret $\mathsf{1}$, $\times$, $+$ and $\to$ using the bicartesian closed structure of the fibres, and quantification of a dimension variable $[\![\Delta \vdash \forall X.T]\!]$ is defined by right adjoint to reindexing along the projection $\pi : [\![\Delta \vdash \Gamma, X]\!] \to [\![\Delta \vdash \Gamma]\!]$. Quantities $\mathsf{Quantity}(D)$ are interpreted by reindexing the object Q along the interpretation of D, i.e. $[\![\Delta \vdash \mathsf{Quantity}(D)]\!] = [\![\Delta \vdash D\ \mathsf{dim}]\!]^*(Q)$.
- Well-formed typing contexts $\Delta \vdash \Gamma\ \mathsf{ctx}$ are interpreted as products in the fibre above $[\![\Delta]\!]$, i.e. $[\![\Delta \vdash x_1 : T_1, \ldots, x_n : T_n]\!] = [\![\Delta \vdash T_1]\!] \times \ldots \times [\![\Delta \vdash T_n]\!]$.
- Well-formed terms $\Delta; \Gamma \vdash t : T$ are interpreted as morphisms $[\![t]\!] : [\![\Gamma]\!] \to [\![T]\!]$ in the fibre above $[\![\Delta]\!]$. We assume that there is an interpretation $[\![\mathrm{op}]\!] : 1 \to [\![T]\!]$ for each primitive operation $(\mathrm{op} : T) \in \mathrm{Ops}$.

In this paper we have only considered universal quantification of units but existential quantification can be given just as easily. Existential quantification is interpreted as the left adjoint to reindexing along a projection. Properties of existential quantification can be proven by dualising the relevant proofs of properties about universal quantification.

Care is needed to ensure that the denotational semantics properly respects substitution. This can be done either by requiring that the fibrations are *split*, or by adding explicit coercions to the language [5]. All the examples of fibrations in this paper are split.

3.2 First Examples of λD-Models

We now give some examples of λD-models. We begin by noting that in Kennedy's paper [10], a simpler approach is taken to the semantics of dimensions: the dimensions are simply thrown away in a *dimension-erasure semantics*. From the categorical perspective, this means that the calculus is stripped of its fibred structure leaving only a simply typed λ-calculus which Kennedy models, as is to be expected, within a Cartesian closed category. In particular, he chooses the Cartesian closed category of complete partial orders which he needs for recursion. Nevertheless, Kennedy's model can be viewed as a λD-model.

▶ **Example 3** (Dimension-Erasure Models)**.** Let $\mathcal{C}$ be a bicartesian closed category. Then the functor $\mathcal{C} \to 1$ is a $\lambda\forall$-fibration. The unique object of 1 is a trivial Abelian group object. By taking $\mathcal{C}$ to be the category of complete partial orders and continuous functions, and by choosing the flat pointed cpo $\mathbb{Q}_\perp$ to interpret $\mathsf{Quantity}$ we obtain a model corresponding to Kennedy's dimension-erasure model. This model supports a plethora of primitive operations, including all the standard arithmetical ones. However, the model also contains many functions which are not dimensionally invariant, i.e. they do not scale appropriately under change of units – Kennedy uses relational parametricity [15] to remove these unwanted elements; we will come back to his relational model in Section 5.

▶ **Example 4** (Syntactical Models)**.** We can construct a $\lambda\forall$-fibration $\mathcal{C}\ell(\lambda D)$ from the syntax in a standard way. The base category $\mathcal{B}$ is the Lawvere theory of Abelian groups $\mathsf{L_{Ab}}$. The fibre $\mathcal{C}\ell(\lambda D)_n$ over n is the category whose objects are types with n dimension variables, and whose morphisms are terms in context, modulo a standard notion of conversion. The object 1 in $\mathsf{L_{Ab}}$ is an Abelian group object, and $(\mathcal{C}\ell(\lambda D) \to \mathsf{L_{Ab}}, 1, (X \vdash \mathsf{Quantity}(X)\ \mathsf{type}))$ is a λD-model.

▶ **Example 5** (The Dimension-Indexed Families Model)**.** Let $\mathsf{Fam}(\mathsf{Set})$ be the category whose objects are pairs $(I, \{X_i\}_{i\in I})$ of a set I and an I-indexed family of sets $\{X_i\}_{i\in I}$. A morphism $(I, \{X_i\}_{i\in I}) \to (J, \{Y_j\}_{j\in J})$ is a pair $(f, \{\phi_i\}_{i\in I})$ where f is a function $f : I \to J$ and ϕ_i is a function $\phi_i : X_i \to Y_{f(i)}$ for all $i \in I$. It is well known that the forgetful functor $(I, \{X_i\}_{i\in I}) \mapsto I : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$, taking a family to its index set, is a $\lambda\forall$-fibration (see e.g. Jacobs [9, Lemma 1.9.5]). For any given set B of fundamental dimensions (e.g. *Length*, *Time*, *Mass* etc.), let G be the free Abelian group on B. Suppose that we also have a set Q_d of quantities for each dimension $d \in G$ (for instance, we can choose $Q_d = \mathbb{R}^+ \times \{\overline{d}\}$ where $\overline{d}$ is a unit of measure for the dimension d, e.g. $\overline{\mathit{Length}} = \mathrm{m}$, $\overline{\mathit{Time}} = \mathrm{s}$, $\overline{d \cdot d'} = \overline{d} \cdot \overline{d'}$ etc). We then have a λD-model with $\mathsf{Quantity}$ interpreted as $(G, \{Q_d\}_{d\in G})$.

In this model, a dimension expression $X_1, \ldots, X_n \vdash D\ \mathsf{dim}$ is interpreted as a function $G^n \to G$ using the free Abelian group structure on G: for each valuation of the dimension variables as physical dimensions, we have an interpretation of the expression as a physical dimension. A type with a free dimension variable $X \vdash T\ \mathsf{type}$ is interpreted as a family of sets, indexed by the dimensions in G. Similarly a term with a free dimension variable is interpreted as a family of functions, one for each dimension in G. This model does support many primitive operations, but it does not support dimension invariant polymorphism. For instance, the model supports adding a term $\mathsf{eq} : \forall X_1.\forall X_2.\mathsf{Bool}$ which tests whether two dimensions are the same, which is clearly not invariant under change of representation.

Related examples include the relations fibration $\mathsf{Rel} \to \mathsf{Set}$ and the subobject fibration $\mathsf{Sub}(\mathsf{Set}) \to \mathsf{Set}$. This example can also be generalised to the fibration $\mathsf{Fam}(\mathcal{C}) \to \mathsf{Set}$, which is a $\lambda\forall$-fibration if $\mathcal{C}$ is bicartesian closed.

A Source of Fibrations with Simple Products

We next look at a particular class of λD models, where the fibres in the $\lambda\forall$-fibration are functors. We prove a general theorem for such fibrations, and instantiate it to construct several examples. We first introduce some notation. Let $\mathcal{S}$ be a category (typically $\mathcal{S} = \mathsf{Set}$), and consider the category $\mathsf{Cat}//\mathcal{S}$. The objects are pairs $(\mathcal{C}, P : \mathcal{C} \to \mathcal{S})$, where $\mathcal{C}$ is a small category and $P : \mathcal{C} \to \mathcal{S}$ a functor. Morphisms $(F, \phi) : (\mathcal{C}, P) \to (\mathcal{D}, Q)$ are pairs of a functor $F : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\phi : P \to Q \circ F$. The obvious projection functor $(\mathcal{C}, P) \mapsto \mathcal{C} : \mathsf{Cat}//\mathcal{S} \to \mathsf{Cat}$ is a fibration. The fibre over a small category $\mathcal{C}$ is the category

$\mathcal{S}^{\mathcal{C}}$ of functors $[\mathcal{C} \to \mathcal{S}]$ and natural transformations between them. Reindexing is given by precomposition of functors.

▶ **Theorem 6.** *If $\mathcal{S}$ has all small limits then the fibration $\mathsf{Cat}//\mathcal{S} \to \mathsf{Cat}$ has simple products.*

This result appears to be fairly well-known folklore (see e.g. Lawvere [11, end of §3], Melliès and Zeilberger [14]), but since it is important in what follows we sketch a proof.

Proof Sketch. For any functor $F : \mathcal{C} \to \mathcal{D}$, the reindexing functor $F^* : \mathcal{S}^{\mathcal{D}} \to \mathcal{S}^{\mathcal{C}}$ has a right adjoint $F_* : \mathcal{S}^{\mathcal{C}} \to \mathcal{S}^{\mathcal{D}}$, known as the 'right Kan extension along F', which always exists when $\mathcal{S}$ has limits. For simple products, we are only interested in a right adjoint to weakening, i.e. in the functor $\forall_{\mathcal{C}} : \mathcal{S}^{\mathcal{C}\times\mathcal{D}} \to \mathcal{S}^{\mathcal{C}}$ which is the right Kan extension along along the projection functor $\pi_{\mathcal{C}} : \mathcal{C} \times \mathcal{D} \to \mathcal{C}$. Expanding the definitions, we see that $\forall_{\mathcal{C}}(P) : \mathcal{C} \to \mathcal{S}$ is a point-wise limit:

$$(\forall_{\mathcal{C}}\, P)(c) = \lim_{d\in\mathcal{D}} P(c,d) \ . \tag{3}$$

The Beck-Chevalley condition requires that the canonical map $F^*\forall_{\mathcal{C}'} \to \forall_{\mathcal{C}}(F \times \mathrm{id}_{\mathcal{D}})^*$ is a natural isomorphism for all functors $F : \mathcal{C} \to \mathcal{C}'$. Indeed, for any $P : \mathcal{C} \times \mathcal{D} \to \mathcal{S}$, $c \in \mathcal{C}$:

$$\begin{aligned}(F^*(\forall_{\mathcal{C}'}\, P))(c) = (\forall_{\mathcal{C}}\, P)(F(c)) \cong \lim_{d\in\mathcal{D}}(F(c),d) &= \lim_{d\in\mathcal{D}}(((F\times \mathrm{id}_{\mathcal{D}})^*(P))(c,d)) \\ &\cong (\forall_{\mathcal{C}}((F\times \mathrm{id}_{\mathcal{D}})^*(P))(c) \ .\end{aligned}$$ ◀

A Source of Models by Change of Base

In general, a useful way of building fibrations is by changing the base. If $p : \mathcal{E} \to \mathcal{B}$ is a fibration, and $F : \mathcal{A} \to \mathcal{B}$ is a functor, then the pullback of p along F in Cat, denoted $F^*p : F^*\mathcal{E} \to \mathcal{A}$, is again a fibration. The same is true of λD-models.

▶ **Theorem 7.** *Let $p : \mathcal{E} \to \mathcal{B}$ be a fibration, and let $F : \mathcal{A} \to \mathcal{B}$ be a functor.*

(i) *If p has simple products and F preserves products, then $F^*p : F^*\mathcal{E} \to \mathcal{A}$ has simple products.*

(ii) *If p is bicartesian closed then $F^*p : F^*\mathcal{E} \to \mathcal{A}$ is bicartesian closed.*

(iii) *If G is an Abelian group object in $\mathcal{A}$ and $(p, F(G), Q)$ is a λD-model, then also $(F^*p, G, (G, Q))$ is a λD-model.*

Proof. For item (i): for any $A \in \mathcal{A}$, reindexing along a projection $\pi_A : A \times A' \to A$ in $\mathcal{A}$ is by construction reindexing along $F(\pi_A)$ in $\mathcal{B}$, which (as F preserves finite products) is the same as reindexing along a projection $\pi_{FA} : FA \times FA' \to FA$, which has a right adjoint and satisfies the Beck-Chevalley condition, since p has simple products.

For item (ii): F^*p is a bicartesian closed fibration since each fibre $(F^*\mathcal{E})_A$ is by construction of the form $\mathcal{E}_{FA}$ and hence bicartesian closed, and reindexing by f in $\mathcal{A}$ is by construction defined to be reindexing by Ff in $\mathcal{B}$, which preserves the structure.

Item (iii) is an immediate corollary. ◀

For a simple illustration of the change of base result, notice that the dimension-erasure fibration $\mathcal{C} \to 1$ arises from pulling back the families fibration $\mathsf{Fam}(\mathcal{C}) \to \mathsf{Set}$ along the unique product-preserving functor $1 \to \mathsf{Set}$.

▶ **Example 8** (Models over the Lawvere theory L_{Ab})**.** Let $(p : \mathcal{E} \to \mathcal{B}, G, Q)$ be a λD-model. Recall that the Abelian group object G in $\mathcal{B}$ gives rise to a unique product-preserving functor $F : \mathsf{L}_{\mathsf{Ab}} \to \mathcal{B}$ such that $F(1) = G$. By Theorem 7, we have a λD-model $(F^*p, 1, (1, Q))$.

▶ **Example 9** (A Model Built from Group Actions)**.** Let G be a group. Recall that a G-set consists of a set A together with a group action, i.e. a function $\cdot_A : G \times A \to A$ such that $e \cdot_A a = a$ and $(gh) \cdot_A a = g \cdot_A (h \cdot_A a)$. The category $\mathsf{Grp}//\mathsf{Set}$ has as objects pairs (G, A) where G is a group and A is a G-set. A morphism $(G, A) \to (H, B)$ in $\mathsf{Grp}//\mathsf{Set}$ is given by a group homomorphism $\phi : G \to H$ and a function $f : A \to B$ such that for any $g \in G$ and $a \in A$ we have $f(g \cdot_A a) = (\phi g) \cdot_B (fa)$. Let Grp be the category of groups and homomorphisms. We call the forgetful functor $p : \mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}$ the $\mathsf{Grp}//\mathsf{Set}$ fibration.

▶ **Proposition 10.** *Let G be an Abelian group, and let Q be a G-set. Then $(p : \mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}, G, Q)$ is a λD-model.*

Proof. For any group H, the fibre above H is the category of H-sets and equivariant functions. This is isomorphic to the the functor category Set^H, where we consider the group H as a category with one object $\star$ and a morphism for each element of H. Indeed, there is a product-preserving, full and faithful functor $\mathsf{Grp} \to \mathsf{Cat}$, taking a group to the corresponding one-object category. The fibration $\mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}$ is thus the pullback of the fibration $\mathsf{Cat}//\mathsf{Set} \to \mathsf{Cat}$ along this embedding $\mathsf{Grp} \to \mathsf{Cat}$. Thus, by Theorem 6 and Theorem 7(i), $\mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}$ has simple products.

Each fibre is bicartesian closed. Products and coproducts are inherited from Set. For the function space, let A and B be G-sets; then the set of functions $(A \to B)$ is also a G-set, with action given by $(g \cdot_{(A \to B)} f)(x) \coloneqq g \cdot_B (f(g^{-1} \cdot_A x))$. It follows that reindexing preserves the bicartesian closed structure. (This is not the case more generally for $\mathsf{Cat}//\mathsf{Set} \to \mathsf{Cat}$, so Theorem 7(ii) does not apply.) Finally, an Abelian group object in Grp is the same thing as an Abelian group. Hence (p, G, Q) is a λD-model. ◀

In the $\mathsf{Grp}//\mathsf{Set}$ fibration, the Abelian group G can be thought of as a group of scaling factors, and the G-set Q is a set of quantities together with a scaling action. For instance, let $Q = G = (\mathbb{R}^+, \times, 1)$, the positive reals. We model a type with a free dimension variable $X \vdash T\ \mathsf{type}$ as a G-set. A term with a free dimension variable is interpreted as a function that is invariant under G. We explore this model in more detail in Section 4.

The $\mathsf{Grp}//\mathsf{Set}$ model supports several primitive operations, which we discuss after Theorem 13.

More generally, instead of having sets and group actions, we also have λD-models built from actions of groupoids.

▶ **Example 11** (A Model Built from Groupoid Actions)**.** Recall that a *groupoid* is a small category $\mathcal{C}$ where every morphism is an isomorphism, and that a functor $\mathcal{C} \to \mathsf{Set}$ is called a groupoid action (or presheaf). The category $\mathsf{Gpd}//\mathsf{Set}$ has as objects pairs $(\mathcal{A}, \phi)$ where $\mathcal{A}$ is a groupoid and $\phi : \mathcal{A} \to \mathsf{Set}$ is a functor. A morphism $(\mathcal{A}, \phi) \to (\mathcal{B}, \psi)$ in $\mathsf{Gpd}//\mathsf{Set}$ is given by a functor $F : \mathcal{A} \to \mathcal{B}$ and a natural transformation $\eta : \phi \to \psi \circ F$ between functors $\mathcal{A} \to \mathsf{Set}$. Let Gpd be the category of groupoids and functors. Then the forgetful functor $\mathsf{Gpd}//\mathsf{Set} \to \mathsf{Gpd}$, which we call the $\mathsf{Gpd}//\mathsf{Set}$ fibration, is a $\lambda\forall$-fibration, and the proof of this is very similar to the proof in Example 9.

On theme with this subsection, the $\mathsf{Gpd}//\mathsf{Set}$ fibration is related to the other fibrations by change of base:

- The families fibration $\mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ from Example 5 arises from pulling back the groupoid action fibration $\mathsf{Gpd}//\mathsf{Set} \to \mathsf{Gpd}$ along the discrete-groupoid-functor $\mathsf{Set} \to \mathsf{Gpd}$.
- The group action fibration $\mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}$ from Example 9 arises from pulling back the groupoid action fibration $\mathsf{Gpd}//\mathsf{Set} \to \mathsf{Gpd}$ along the functor $\mathsf{Grp} \to \mathsf{Gpd}$ that regards each group as a groupoid with one object.

We now discuss λD-models in $\mathsf{Gpd}//\mathsf{Set}$. Let $f : G \to H$ be a homomorphism of Abelian groups. This induces a groupoid whose objects are the elements of H, and where the hom-sets are $\mathrm{mor}(h, h') = \{g \in G \mid f(g) \cdot_H h = h'\}$. The group operation in G provides composition of morphisms. This groupoid can be given the structure of an Abelian group object in Gpd, and, moreover, every Abelian group in Gpd arises in this way [3].

We have already seen that the $\mathsf{Gpd}//\mathsf{Set}$ fibration subsumes the families and group actions fibrations. It also subsumes them as λD-models. To recover group actions (Example 9), let G be an Abelian group of scale factors. The Abelian group object induced by the unique homomorphism $G \to 1$ is a one-object groupoid, and hence we build the λD-models of group actions. To recover the families example (Example 5), fix a set of dimension constants and let H be the free Abelian group on that set. The unique homomorphism $1 \to H$ induces the discrete groupoid whose objects are H, and hence we build the λD-models of families of sets.

4 Group Actions and Dimension Types

In this section we will look in greater detail at the λD-model given by the $\mathsf{Grp}//\mathsf{Set}$ fibration. It turns out that many interesting theorems can be proven in this model, and so to aid us in this task we first concretely spell out the reindexing and simple product structure.

Given a group G, we write $\mathcal{G}$ (with a different font) for the corresponding one-element category, which has morphisms given by elements of G and composition given by group multiplication. Suppose that $\phi : \mathcal{G} \to \mathsf{Set}$ is a G-set (considered as a functor). We write $|\phi| \coloneqq \phi(\star)$ for the underlying carrier set of ϕ. Reindexing along $\pi : \mathcal{G} \times \mathcal{H} \to \mathcal{G}$ yields the $G \times H$-set given by $\phi \circ \pi$. In other words, $\pi^*\phi$ is a $G \times H$-set with the same underlying carrier $|\pi^*\phi| = |\phi|$ as the G-set ϕ, and action given by $(g, h) \cdot_{\pi^*\phi} x = g \cdot_\phi x$.

Now suppose that $\psi : \mathcal{G} \times \mathcal{H} \to \mathsf{Set}$ is a $G \times H$-set. According to Theorem 6 (equation (3)), the underlying set of $\forall_\pi \psi$ is given by $|\forall_\pi \psi| = \lim_{y \in \mathcal{H}} \psi(\star, y)$. By the universal property of limits,

$$\lim_{y \in \mathcal{H}} \psi(\star, y) \cong \mathsf{Set}\big(1, \lim_{y \in \mathcal{H}} \psi(\star, y)\big) \cong [\mathcal{H}, \mathsf{Set}]\big(K1, \psi(\star, \_)\big) \ ,$$

hence $|\forall_\pi \psi| = \{y \in |\psi| \mid \forall h \in H \,.\, (e_A, h) \cdot_\psi y = y\}$, and the action is given by $g \cdot_{\forall_\pi \psi} x = (g, e_H) \cdot_\psi x$. Notice that to give the group action of $\forall_\pi \psi$, we had to make a particular choice of an element in H, namely the identity element e_H. However, any element of H would have given the same result, since for all $y \in |\forall_\pi \psi|$,

$$(g, h) \cdot_\psi y = ((g, e_H)(e_G, h)) \cdot_\psi y = (g, e_H) \cdot_\psi ((e_G, h) \cdot_\psi y) = (g, e_H) \cdot_\psi y.$$

Many of the properties of dimension types that Kennedy proves using parametricity can be shown to hold in the $\mathsf{Grp}//\mathsf{Set}$-fibration, without having to define a separate relational semantics and this is the content of Theorems 13–18. Before we formally state and prove these we introduce a substitution lemma, which holds in any model.

▶ **Lemma 12** (Substitution Lemma). *Suppose that* $\Delta, X \vdash T$ type *and that* $\Delta \vdash D$ dim *denotes a dimension expression. Then* $[\![T[D/X]]\!] \cong (\mathrm{id}_{[\![\Delta]\!]}, [\![D]\!])^* [\![T]\!]$.

Proof. By induction on the structure of T. ◀

Explicitly, Lemma 12 says that the semantics of substituting a dimension expression for a dimension variable is given by reindexing along the identity paired with the dimension expression. Since reindexing is given by precomposition we have that

$$(\mathrm{id}_{[\![\Delta]\!]}, [\![D]\!])^* [\![T]\!] \cong [\![T]\!](\mathrm{id}_{[\![\Delta]\!]}, [\![D]\!]) \ ,$$

i.e., substitution of the n^{th} dimension variable is given by precomposition at the n^{th} component.

For the rest of this section, we will use semantic brackets $[\![\ \_\ ]\!]$ to refer only to the Grp//Set interpretation.

▶ **Theorem 13.** *Suppose that* $X_1, \ldots, X_n, X \vdash S, T$ type. *Then*

$$|[\![\forall X.S \to T]\!]| \cong [\mathcal{G}, \mathsf{Set}]([\![S]\!](\underbrace{\star, \ldots, \star}_{n \text{ times}}, \_), [\![T]\!](\underbrace{\star, \ldots, \star}_{n \text{ times}}, \_))$$

Proof. By the Kan extension formula and Yoneda. ◀

This theorem says that in the Grp//Set model a universally quantified variable over an arrow type can be considered as a natural transformation between the domain and codomain of the arrow type, with the first n components fixed. In other words, it is interpreted as the set of functions that are equivariant in the last argument.

In particular, if $X_1 \ldots X_n \vdash S, T$ type then the type $(\forall \vec{X}.\, S \to T)$ is interpreted as the set of all homomorphisms $[\mathcal{G}^n, \mathsf{Set}]([\![S]\!], [\![T]\!])$. We use this fact to conclude that the group actions model supports several primitive operations. For any $q \in Q$, we can accommodate a term constant $q : \mathsf{Quantity}(1)$, which is interpreted by $[\![q]\!] = q$. When $Q = G$, we can also accommodate a term constant for multiplication

$$\times : \forall X.\forall Y.\, \mathsf{Quantity}(X) \times \mathsf{Quantity}(Y) \to \mathsf{Quantity}(X \cdot Y)$$

which is interpreted as the group operation. When $Q = G = (\mathbb{R}^+, \times, 1)$, the positive reals, we also have addition, $+ : \forall X.\, \mathsf{Quantity}(X) \times \mathsf{Quantity}(X) \to \mathsf{Quantity}(X)$, which is equivariant since $q(r + s) = qr + qs$.

▶ **Theorem 14.** *Suppose that* $\Delta, X \vdash T$ type. *Then* $|[\![\forall X.\mathsf{Quantity}(X) \to T]\!]| \cong |[\![T[1/X]]\!]|$.

Proof. By Theorem 13, Lemma 12 and Yoneda. ◀

We now prove some theorems about the Grp//Set fibration that are parametricity results in Kennedy's original paper. The proofs here involve applications of Lemma 12, Theorem 13 and Theorem 14. First, we take a look at the interplay between scaling factors and polymorphic functions.

▶ **Theorem 15** (Scaling Factors). *Suppose* $\Delta_{dim}; \Gamma_{units} \vdash t : \forall X.\mathsf{Quantity}(X) \to \mathsf{Quantity}(X^n)$, *where* $n \in \mathbb{N}$. *Then for all* $g \in G$ *and* $x \in |[\![\mathsf{Quantity}(X)]\!]|$, *we have* $[\![t]\!](g \cdot x) = g^n \cdot [\![t]\!]x$.

Proof. We know from Theorem 13 that $[\![t]\!] \in [\mathcal{G}, \mathsf{Set}]([\![\mathsf{Quantity}(X)]\!], [\![\mathsf{Quantity}(X^n)]\!])$. In other words, $[\![t]\!](g \cdot x) = g^n \cdot [\![t]\!]x$ for all $x \in |[\![\mathsf{Quantity}(X)]\!]|$, as required. ◀

This theorem tells us that polymorphic functions are *invariant under scaling*. Intuitively we see that scaling factors must be changed in an appropriately polymorphic way. If we apply Theorem 14 to the type $\forall X.\mathsf{Quantity}(X) \to \mathsf{Quantity}(X^n)$, we see that

$$|[\![\forall X.\mathsf{Quantity}(X) \to \mathsf{Quantity}(X^n)]\!]| \cong |[\![\mathsf{Quantity}(1^n)]\!]| \cong |[\![\mathsf{Quantity}(1)]\!]| \cong Q,$$

Putting $Q = G$, we conclude that all the terms of type $\forall X.\mathsf{Quantity}(X) \to \mathsf{Quantity}(X^n)$ are of the form $\Lambda X.\, \lambda q : \mathsf{Quantity}(X).\ r \times q^n$ for $r \in G$.

▶ **Theorem 16.** *There is no ground term* $\vdash t : \forall X.\mathsf{Quantity}(X^2) \to \mathsf{Quantity}(X)$., *i.e., we cannot write a polymorphic square root function.*

Proof. To see this we exhibit a model where the existence of such a term is impossible. Consider the λD-model $(p : \mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}, \mathbb{Z}_2, \mathbb{Z}_2)$ where the Abelian group $\mathbb{Z}_2 = (\{-1, 1\}, \cdot, 1)$ is used to interpret both dimensions and units. Theorem 13 says that the interpretation of the type $\forall X.\mathsf{Quantity}(X^2) \to \mathsf{Quantity}(X)$ is given by

$$|[\![\forall X.\mathsf{Quantity}(X^2) \to \mathsf{Quantity}(X)]\!]| \cong [\mathbb{Z}_2, \mathsf{Set}]([\![\mathsf{Quantity}(X^2)]\!], [\![\mathsf{Quantity}(X)]\!])$$

i.e. any element f of $|[\![\forall X.\mathsf{Quantity}(X^2) \to \mathsf{Quantity}(X)]\!]|$, satisfies for all $g, x \in \mathbb{Z}_2$

$$f(g^2 \cdot x) = g \cdot (fx) \qquad (*)$$

If f exists, then either $f(-1) = -1$ or $f(-1) = 1$, but both lead to contradictions. To this end suppose that $f(-1) = -1$, then by $(*)$ we have $f((-1)^2 \cdot -1) = (-1) \cdot f(-1)$, which is a contradiction since the left-hand side is equal to -1 and the right-hand side is equal to 1. A similar argument shows that $f(-1) = 1$ is also not possible, and hence there exists no such f. ◀

This result can be extended to also include terms t using primitive operations Ops, as long as these operations can be interpreted in the model in question. For example, the result holds in the presence of multiplication

$$\times : \forall X.\forall Y.\, \mathsf{Quantity}(X) \times \mathsf{Quantity}(Y) \to \mathsf{Quantity}(X \cdot Y).$$

However, this model does not support a polymorphic zero constant $0 : \forall X.\, \mathsf{Quantity}(X)$, as such a primitive would of course gives rise to a trivial counterexample to the theorem.

Next, we can prove a theorem that relates a dimensionally invariant function to a dimensionless one. This is a simplified version of the *Buckingham Pi Theorem* of dimensional analysis [4] (for a more modern introduction, see Sonin [16]).

▶ **Theorem 17.** *We have a bijection*

$$|[\![\forall X.\mathsf{Quantity}(X) \times \mathsf{Quantity}(X) \to \mathsf{Quantity}(1)]\!]| \cong |[\![\mathsf{Quantity}(1) \to \mathsf{Quantity}(1)]\!]|$$

Proof. This is a consequence of Theorem 14, after currying. ◀

We finish this section with another uninhabitedness result, this time about a higher order type.

▶ **Theorem 18.** *There is no term*

$$\vdash t : \forall X_1.\forall X_2.(\mathsf{Quantity}(X_1) \to \mathsf{Quantity}(X_2)) \to \mathsf{Quantity}(X_1 \cdot X_2)\ .$$

Proof. Choose G and Q to be $\mathbb{Z}_2$. Interpreting the type of t, we have

$$\begin{aligned}&[\![\forall X_1.\forall X_2.(\mathsf{Quantity}(X_1) \to \mathsf{Quantity}(X_2)) \to \mathsf{Quantity}(X_1 \cdot X_2)]\!] \\ &= \{t \in (\mathbb{Z}_2 \to \mathbb{Z}_2) \to \mathbb{Z}_2 \mid \\ &\qquad \forall g_1, g_2 \in \mathbb{Z}_2,\ f : \mathbb{Z}_2 \to \mathbb{Z}_2.\ (g_1 g_2) \cdot (t(f)) = t(\lambda q \in \mathbb{Z}_2.\, g_2 \cdot (f(g_1^{-1} \cdot q)))\}\end{aligned}$$

Hence for any $t \in [\![\forall X_1.\forall X_2.(\mathsf{Quantity}(X_1) \to \mathsf{Quantity}(X_2)) \to \mathsf{Quantity}(X_1 \cdot X_2)]\!]$, instantiating $f = \mathrm{id}_Q$ we get that $(g_1 g_2) \cdot (t(\mathrm{id}_Q)) = t(\lambda q \in \mathbb{Z}_2.\, g_2 \cdot (g_1^{-1} \cdot q))$ for all g_1 and g_2, but this is not possible. If $g_1 = 1$ and $g_2 = -1$, then the equation reduces to $-1 \cdot t(\mathrm{id}_Q) = t(\mathrm{id}_Q)$, which is a contradiction since $t(\mathrm{id}_Q) \in \mathbb{Z}_2 = \{-1, 1\}$. ◀

Again, the result can be extended to terms that use a set of primitive operations Ops, as long as all primitive operations in Ops can be interpreted in the model used in the proof.

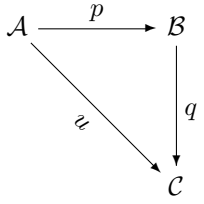
5 Relational Models

In Section 4 we pointed out that many of the results that we proved in the $\mathsf{Grp}//\mathsf{Set}$ λD-model are results that Kennedy [10] proves using parametricity. It is curious how the parametricity-style proofs in the $\mathsf{Grp}//\mathsf{Set}$ λD-model are simple and slick and do not require a separate relational semantics. One cannot help but wonder, is the $\mathsf{Grp}//\mathsf{Set}$ λD-model really as good as having full-blown parametricity at one's finger tips?

To answer this question we look at a general method of attaching a (fibrational) logic to a λD-model to give a notion of a *relational* λD-model. This allows us to reconstruct Kennedy's relational parametricity in our setting (Example 21), as well as to talk about a relational version of the $\mathsf{Grp}//\mathsf{Set}$ λD-model (Example 22).

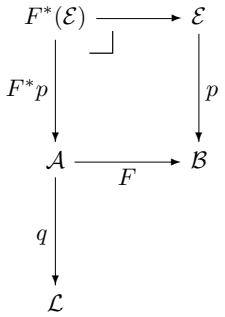
To begin this section, we first recall a theorem about the composition of fibred structure.

▶ **Theorem 19.** *Suppose that $p : \mathcal{A} \to \mathcal{B}$ and $q : \mathcal{B} \to \mathcal{C}$ are fibrations and let $u : \mathcal{A} \to \mathcal{C}$ denote the composite $q \circ p$ (hence u is also a fibration). Suppose further that q has simple products. For any projection map $\pi_{q(B)} : q(B) \times Y \to q(B)$ in $\mathcal{C}$, denote the Cartesian morphism in $\mathcal{B}$ above it by $\pi^{\S}_{q(B)} : \pi^*B \to B$. Then u has simple products that are preserved by p if and only if for any projection map $\pi : q(B) \times Y \to q(B)$ in $\mathcal{C}$, the functor $(\pi^{\S}_{q(B)})^* : \mathcal{A}_B \to \mathcal{A}_{\pi^*B}$ has right adjoints for all $B \in \mathcal{B}$, satisfying the Beck-Chevalley condition.*



Proof. This theorem is proven by using the factorisation and lifting properties of the 2-category Fib as outlined by Hermida [7]. Though the proof is not too difficult, it does require 2-categorical technology, which we do not introduce here. Hence, we leave the proof as an exercise for the 2-category-savvy reader. ◀

We now put this theorem to use. Given a λD-model $q : \mathcal{A} \to \mathcal{L}$ and a logic $p : \mathcal{E} \to \mathcal{B}$, there is a natural way to glue them together to provide a relational semantics.



▶ **Theorem 20.** *Let $(q : \mathcal{A} \to \mathcal{L},\ G,\ Q_0)$ be a λD-model, $F : \mathcal{A} \to \mathcal{B}$ a product preserving functor and $p : \mathcal{E} \to \mathcal{B}$ a bicartesian closed fibration with products. Consider the pullback of p along F, and let Q_R denote an object in the fibre $\mathcal{E}_{F(Q_0)}$. Then $(q \circ F^*p : F^*\mathcal{E} \to \mathcal{L}, G, (Q_0, Q_R))$ is a λD-model.*

Proof. Clearly G is an Abelian group object in $\mathcal{L}$, and (Q_0, Q_R) is in the fibre $(F^*\mathcal{E})_G$. To check that $q \circ F^*p$ is a bicartesian closed fibration is a simple exercise. Finally, since p has all products, so does F^*p. Hence, $q \circ F^*p$ has simple products by Theorem 19. ◀

Next, we look at an example that uses Theorem 20 to generate Kennedy's original relationally parametric model of dimension types [10] from essentially the dimension-erasure model back in Example 3.

▶ **Example 21.** Let G be an Abelian group. Then using the notation from Theorem 20, let $\mathcal{L}$ be the Lawvere theory of Abelian groups L_{Ab}, $\mathcal{A}$ be the category $\mathsf{L}_{\mathsf{Ab}} \times \mathsf{Set}$, $q : \mathsf{L}_{\mathsf{Ab}} \times \mathsf{Set} \to \mathsf{L}_{\mathsf{Ab}}$ be the fibration given by the first projection, and $p : \mathsf{Sub}(\mathsf{Set}) \to \mathsf{Set}$ be the subset fibration. Define $F : \mathsf{L}_{\mathsf{Ab}} \times \mathsf{Set} \to \mathsf{Set}$ to be the product preserving functor defined on objects

$(n, X) \in \mathsf{L_{Ab}} \times \mathsf{Set}$ by $F(n, X) = G^n \times X \times X$, and on morphisms $(f, g) : (n, X) \to (m, Y)$ by $F(f, g) = (G^f, g, g)$. Finally, let $Q_0 = G$, and $Q_R = \{(g, g_1, g_2) \mid gg_1 = g_2\} \subseteq G \times G \times G$.

In this model, each type $\Delta \vdash T$ is interpreted as a triple $(|\Delta|, [\![T]\!]_o, [\![T]\!]_r) \in \mathsf{L_{Ab}} \times \mathsf{Set} \times \mathsf{Sub}(\mathsf{Set})$, where $[\![T]\!]_r \subseteq G^n \times [\![T]\!]_o \times [\![T]\!]_o$. Spelling this out explicitly, we have the following interpretations, which are equivalent to Kennedy's original relationally parametric model for dimension types:

$$
\begin{aligned}
[\![\Delta \vdash \mathsf{Quantity}(D)]\!] &= (|\Delta|, G, \{(g, g_1, g_2) \mid ([\![D]\!]g)g_1 = g_2\}) \\
[\![\Delta \vdash T \times U]\!] &= (|\Delta|, [\![T]\!]_o \times [\![U]\!]_o, \\
&\quad \{(g, (t_1, u_1), (t_2, u_2)) \mid (g, t_1, t_2) \in [\![T]\!]_r, (g, u_1, u_2) \in [\![U]\!]_r\}) \\
[\![\Delta \vdash T + U]\!] &= (|\Delta|, [\![T]\!]_o + [\![U]\!]_o, \\
&\quad \{(g, \mathsf{inj}_1\, t, \mathsf{inj}_1\, t') \mid (g, t, t') \in [\![T]\!]_r\} \\
&\quad\quad \cup \{(g, \mathsf{inj}_2\, u, \mathsf{inj}_2\, u') \mid (g, u, u') \in [\![U]\!]_r\}) \\
[\![\Delta \vdash T \to U]\!] &= (|\Delta|, [\![T]\!]_o \to [\![U]\!]_o, \\
&\quad \{(g, f_1, f_2) \mid \forall t_1, t_2.\ (g, t_1, t_2) \in [\![T]\!]_r \implies (g, f_1 t_1, f_2 t_2) \in [\![U]\!]_r\}) \\
[\![\Delta \vdash \forall X.\ T]\!] &= (|\Delta|, [\![T]\!]_o, \{(g, t_1, t_2) \mid \forall g' \in G.\ ((g, g'), t_1, t_2) \in [\![T]\!]_r\})
\end{aligned}
$$

Note that in the interpretation of $\forall X.\ T$, the "carrier" (i.e., the second component) is exactly the carrier of the interpretation of T.

We can also apply Theorem 20 to obtain a natural relational model for the $\mathsf{Grp}//\mathsf{Set}$ λD-model (Example 9).

▶ **Example 22.** As before, let G be an Abelian group and $\mathcal{L}$ be the Lawvere theory of Abelian groups $\mathsf{L_{Ab}}$. Let $q : \mathcal{A} \to \mathsf{L_{Ab}}$ be the pullback of the fibration $\mathsf{Grp}//\mathsf{Set} \to \mathsf{Grp}$ along the unique product-preserving functor $M : \mathsf{L_{Ab}} \to \mathsf{Grp}$ with $M(1) = G$, as in Example 8, so that the objects of $\mathcal{A}$ are triples (n, X, ϕ) with (X, ϕ) a G^n-set. Let $p : \mathsf{Sub}(\mathsf{Set}) \to \mathsf{Set}$ be the subset fibration. Define $F : \mathcal{A} \to \mathsf{Set}$ to be the product preserving functor defined on objects by $F(n, X, \phi) = G^n \times X \times X$ and on morphisms $(f, \alpha) : (n, X, \phi) \to (m, Y, \psi)$ by $F(f, \alpha) = (\alpha, f, f)$. Finally, we let $Q_0 = (G, \phi)$, where ϕ denotes group multiplication, and $Q_R = \{(g, g_1, g_2) \mid gg_1 = g_2\} \subseteq G \times G \times G$.

Then each type $\Delta \vdash T$ is again interpreted as a triple $(|\Delta|, [\![T]\!]_o, [\![T]\!]_r) \in \mathsf{L_{Ab}} \times \mathsf{Sub}(\mathsf{Set})$, with $[\![T]\!]_r \subseteq G^n \times [\![T]\!]_o \times [\![T]\!]_o$. The only difference between the interpretation of types in this example and Example 21 is the second component of the interpretation of dimension quantification:

$$
\begin{aligned}
[\![\Delta \vdash \forall X.\ T]\!]_r = (|\Delta|, &\{t \in |[\![T]\!]_r| \mid \forall g \in G.\ ((e_{G^{|\Delta|}}, g), t, t) \in [\![T]\!]_r\}, \\
&\{(g, t_1, t_2) \mid \forall g' \in G.\ ((g, g'), t_1, t_2) \in [\![T]\!]_r\})
\end{aligned}
$$

This interpretation, in contrast to the interpretation in Example 21, has "cut-down" the carrier of the interpretation of $\forall$-types to only include the "parametric" elements. As a consequence, this interpretation satisfies an analogue of the *Identity Extension* lemma from relationally parametric models of System F [15].

▶ **Proposition 23.** *For all type interpretations* $(|\Delta|, [\![T]\!]_o, [\![T]\!]_r)$*, we have:*

$$\forall x_1, x_2 \in [\![T]\!]_o.\ (e, x_1, x_2) \in [\![T]\!]_r \Leftrightarrow x_1 = x_2$$

Compare this to the identity extension property for System F models, which states that if we instantiate the relational interpretation of a type with the equality relation for all of its free variables, then the resulting relation is the equality relation. In the current setting, equality relations for the free variables are replaced by the unit element of the groups $G^{|\Delta|}$. Indeed, this model is equivalent to the restriction to one-dimensional scalings of the reflexive graph model for System Fω with geometric symmetries presented by Atkey [1].

We end this discussion of relational models by showing the relationships between the models in Examples 21 and 22 and the Grp//Set model we considered in detail in Section 4. By construction, the carriers of the interpretations of each type in the model in Example 22 and the Grp//Set model are identical. Moreover, the relational interpretation in Example 22 and the group action in the Grp//Set model are related as follows.

▶ **Theorem 24.** *For all types $\Delta \vdash T$ type, if the interpretation of T in the model of Example 22 is $(|\Delta|, A, P \subseteq G^{|\Delta|} \times A \times A)$ and the* Grp//Set *model interpretation is (G^n, A, ψ), then $(g, a_1, a_2) \in P \Leftrightarrow g \cdot_\psi a_1 = a_2$.*

Proof. By induction on the derivation of $\Delta \vdash T$ type. ◀

Using Theorem 24, we can see that we could have used the relationally parametric model to derive the results in Section 4. There is literally no difference between the two models for the purposes of interpreting the types of our calculus.

We can also relate the relationally parametric model from Example 22 to the dimension-erasure semantics in Example 3. By constructing a logical relation between the two models, we can show:

▶ **Theorem 25.** *For any closed term $\vdash t$: Bool, the interpretation of t in the dimension-erasure model of Example 3 is equal to the interpretation of t in the relationally parametric model of Example 22.*

By the compositionality of both interpretations, this theorem means that if we can show that two open terms s and t are equal in the model of Example 22 (and equivalently, the Grp//Set model), then they will be contextually equivalent for the dimension erasure model.

It remains to discuss the relationship between Kennedy's original relational model (Example 21), and the relational model in Example 22 that satisfies the identity extension property. As noted above, the difference between these interpretations lies in the semantics of the $\forall$-type. Kennedy's model does not restrict the carrier of the interpretation to just the "parametric" elements, i.e., the elements that preserve all relations. Therefore, the interpretations of types that contain nested $\forall$s are not directly comparable. We might expect that we could observe a difference between the two models when proving statements about terms whose types contain negatively nested $\forall$-types. However, Kennedy's original work does not present any results involving terms with such types, and we have not found any natural examples. This is in contrast with the situation with relationally parametric models of System F, where the proof that final coalgebras can be represented crucially relies on the restriction of the interpretation of quantified types to the parametric elements [2].

Therefore, our Grp//Set model and the equivalent relational model in Example 22 practically coincides with Kennedy's original model, but offer the advantage of not requiring a separate relational semantics to prove important theorems. This in many cases makes proofs of these theorems clearer. Additionally, the Grp//Set model offers an interpretation that directly links the semantics to symmetry.

6 Concluding Remarks

To conclude, we have studied a typed λ-calculus with polymorphism over physical dimensions, which we called λD (Section 2) and we have developed a model theory for the calculus. Under the Curry-Howard correspondence, the λD-calculus is a fragment of first-order logic where the domain of discourse is an unspecified Abelian group, and so our notion of model (Definition 2) is based on the standard fibrational techniques in categorical logic.

One particular model turned out to be particularly straightforward and yet informative – the model based on group actions (Example 9). Of course, automorphisms and group actions play a key role in the classical model theory of first order logic, but in this paper we have shown that these techniques are also useful on the other side of the Curry-Howard correspondence. Many arguments about the λD-calculus, including type isomorphisms and definability arguments, can be made in this model (Section 4).

Parametricity is most often studied using relational techniques, and in this paper we have developed a method for building relational λD-models (Theorem 20). Using this method we were able to reconstruct two particular relational models: a relational model due to Kennedy (Example 21, [10]) and a restriction of a relational model due to Atkey (Example 22, [1]). Although the group-actions model is different in style, we showed (formally) that it is actually closely related to the two relational models (Theorems 24 and 25).

Acknowledgements. We are grateful to participants in a discussion on the categories mailing list about the Grp//Set fibration. Research supported by the EPSRC Grant EP/K023837/1 (NG, FNF) and ERC Grant QCLS and a Royal Society Fellowship (SS).

References

1 Robert Atkey. From parametricity to conservation laws, via Noether's theorem. In *Proc. POPL 2014*, pages 491–502, 2014.
2 Lars Birkedal and Rasmus E Møgelberg. Categorical models for Abadi and Plotkin's logic for parametricity. *Mathematical Structures in Computer Science*, 15(04):709–772, 2005.
3 Ronald Brown and Christopher B Spencer. G-groupoids, crossed modules and the fundamental groupoid of a topological group. *Proc. Indag. Math.*, 79(4):296–302, 1976.
4 Edgar Buckingham. On physically similar systems; illustrations of the use of dimensional equations. *Physical Review*, 4(4):345–376, 1914.
5 P-L Curien, Richard Garner, and Martin Hofmann. Revisiting the categorical interpretation of dependent type theory. *Theoret. Comput. Sci.*, 546:99–119, 2014.
6 Martin Erwig and Margaret Burnett. Adding apples and oranges. In *Practical Aspects of Declarative Languages*, pages 173–191. Springer, 2002.
7 Claudio Hermida. Some properties of Fib as a fibred 2-category. *Journal of Pure and Applied Algebra*, 134(1):83–109, 1999.
8 Ronald T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, 1983.
9 Bart Jacobs. *Categorical logic and type theory*, volume 141. Elsevier, 1999.
10 Andrew J. Kennedy. Relational parametricity and units of measure. In *POPL'97*, 1997.
11 F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3-4):281–296, 1969.
12 R Männer. Strong typing and physical units. *ACM Sigplan Notices*, 21(3):11–20, 1986.
13 Mars Climate Orbiter Mishap Investigation Board. Phase I report. NASA, 1999.
14 Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. In *Proc. POPL 2015*, pages 3–16, 2015.
15 John Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, 1983.
16 Ain A Sonin. The physical basis of dimensional analysis. Department of Mechanical Engineering, MIT, 2001.
17 Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference. In *Computational Logic – Essays in Honor of Alan Robinson*, pages 479–483, 1991.

MALL Proof Equivalence is Logspace-Complete, via Binary Decision Diagrams

Marc Bagnol

Department of Mathematics and Statistics, University of Ottawa, Canada

Abstract

Proof equivalence in a logic is the problem of deciding whether two proofs are equivalent *modulo* a set of permutation of rules that reflects the commutative conversions of its cut-elimination procedure. As such, it is related to the question of proofnets: finding canonical representatives of equivalence classes of proofs that have good computational properties. It can also be seen as the word problem for the notion of free category corresponding to the logic.

It has been recently shown that proof equivalence in MLL (the multiplicative with units fragment of linear logic) is Pspace-complete, which rules out any low-complexity notion of proofnet for this particular logic.

Since it is another fragment of linear logic for which attempts to define a fully satisfactory low-complexity notion of proofnet have not been successful so far, we study proof equivalence in MALL$^-$ (multiplicative-additive without units fragment of linear logic) and discover a situation that is totally different from the MLL case. Indeed, we show that proof equivalence in MALL$^-$ corresponds (under AC_0 reductions) to equivalence of binary decision diagrams, a data structure widely used to represent and analyze Boolean functions efficiently.

We show these two equivalent problems to be Logspace-complete. If this technically leaves open the possibility for a complete solution to the question of proofnets for MALL$^-$, the established relation with binary decision diagrams actually suggests a negative solution to this problem.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, F.4.1 Mathematical Logic

Keywords and phrases linear logic, proof equivalence, additive connectives, proofnets, binary decision diagrams, logarithmic space, AC_0 reductions

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.60

1 Introduction

Proofnets: from commutative conversions to canonicity

From the perspective of the Curry-Howard (or propositions-as-types) correspondence [5], a proof of $A \Rightarrow B$ in a logic enjoying a `cut`-elimination procedure can be seen as a program that inputs (through the `cut` rule) a proof of A and outputs a `cut`-free proof of B.

Coming from this dynamic point of view, linear logic [6] makes apparent the distinction between data that can or cannot be copied/erased via its exponential modalities and retains the symmetry of classical logic: the linear negation $(\cdot)^\star$ is an involutive operation. The study of `cut`-elimination is easier in this setting thanks to the linearity constraint. However, in its sequent calculus presentation, the `cut`-elimination procedure of linear logic still suffers from the common flaw of these type of calculi: commutative conversions.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 60–75

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$$\dfrac{\dfrac{\dfrac{\langle\pi\rangle}{A^\star, B^\star, C, D, \Gamma}}{A^\star \parr B^\star, C, D, \Gamma}\parr}{A^\star \parr B^\star, C \parr D, \Gamma}\parr \qquad \dfrac{\langle\mu\rangle \quad \langle\nu\rangle}{A \quad B}\otimes \qquad \dfrac{}{C \parr D, \Gamma}\text{cut} \quad \to \quad \dfrac{\dfrac{\dfrac{\langle\pi\rangle}{A^\star, B^\star, C, D, \Gamma}}{A^\star \parr B^\star, C, D, \Gamma}\parr \qquad \dfrac{\langle\mu\rangle \quad \langle\nu\rangle}{A \quad B}\otimes}{C, D, \Gamma}\text{cut}}{C \parr D, \Gamma}\parr$$

In the above reduction, one of the two formulas related by the `cut` rule is introduced deeper in the proof, making it impossible to perform an actual elimination step right away: one needs first to *permute* the rules in order to be able to go on.

This type of step is called a *commutative conversion* and their presence complexify a lot the study of the `cut`-elimination procedure, as one needs to work *modulo* an equivalence relation on proofs that is not orientable into a rewriting procedure in an obvious way: there are for instance situations of the form

$$\dfrac{\dfrac{\langle\pi_1\rangle \quad \langle\pi_2\rangle}{A^\star, B^\star, \Gamma \qquad A}\text{cut}}{\vdash B^\star, \Gamma} \qquad \dfrac{\langle\pi_3\rangle}{B}\qquad \dfrac{}{\vdash \Gamma}\text{cut} \quad \leftrightarrow \quad \dfrac{\dfrac{\langle\pi_1\rangle \quad \langle\pi_3\rangle}{A^\star, B^\star, \Gamma \qquad B}\text{cut}}{\vdash A^\star, \Gamma} \qquad \dfrac{\langle\pi_2\rangle}{A}\qquad \dfrac{}{\vdash \Gamma}\text{cut}$$

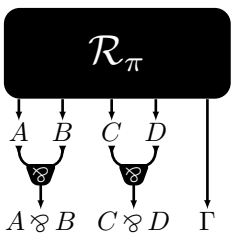
where it is not possible to favor one side of the equivalence without further non-local knowledge of the proof. The point here is that, as a language for describing proofs, sequent calculus is somewhat *too explicit*. For instance, the fact that the two proofs

$$\dfrac{\dfrac{\dfrac{\langle\pi\rangle}{A, B, C, D, \Gamma}}{A \parr B, C, D, \Gamma}\parr}{A \parr B, C \parr D, \Gamma}\parr \qquad \text{and} \qquad \dfrac{\dfrac{\dfrac{\langle\pi\rangle}{A, B, C, D, \Gamma}}{A, B, C \parr D, \Gamma}\parr}{A \parr B, C \parr D, \Gamma}\parr$$

are different objects from the point of view of sequent calculus generates the first commutative conversion we saw above.

A possible solution to this issue is to look for more intrinsic description of proofs, to find a language that is more *synthetic*; if possible to the point where we have no commutative conversions to perform anymore.

Introduced at the same time as linear logic, the theory of *proofnets* [6, 7] partially addresses this issue. The basic idea is to describe proofs as graphs rather than trees, where application of logical rules become local graph construction, thus erasing some inessential sequential informations. Indeed, the two proofs above would translate into the same proofnet:



(where $\mathcal{R}_\pi$ is the proofnet translation of the rest of the proof) and the corresponding commutative conversion disappears.

For the multiplicative without units fragment of linear logic (MLL$^-$), proofnets yield an entirely satisfactory solution to the problem, and constitute a low-complexity canonical representation of proofs based on local operations on graphs.

By canonical, we mean here that two proofs are equivalent *modulo* the permutations of rules induced by the commutative conversions if and only if they have the same proofnet

translation. From a categorical perspective, this means that proofnets constitute a syntactical presentation of the free semi-$*$-autonomous category and a solution to the associated word problem [10].

Contrastingly, the linear logic community has struggled to extend the notion of proofnets to wider fragment: even the question of MLL (that is, MLL$^-$ plus the multiplicative units) could not find a satisfactory answer. A recent result [9] helps to understand this situation: proof equivalence of MLL is actually a Pspace-complete problem. Hence, there is no hope for a satisfactory notion of low-complexity proofnet for this fragment[1].

In this article, we consider the same question, but in the case of MALL$^-$: the multiplicative-additive without units fragment of linear logic. Indeed, this fragment has so far also resisted the attempts to build a notion of proofnet that at the same time characterizes proof equivalence and has basic operations of tractable complexity: we have either canonical nets of exponential size [13] or tractable nets that are not canonical [7]. Therefore, it would have not been too surprising to have a similar result of completeness for some untractable complexity class. An obvious candidate in that respect would be coNP: as we will see, one of these two approaches to proofnets for MALL$^-$ is related to Boolean formulas, which equivalence problem is coNP-complete.

It turns out in the end that this is not the case: our investigation concludes that the equivalence problem in MALL$^-$ is Logspace-complete under AC_0 reductions. But maybe more importantly, we uncover in the course of the proof an unexpected connexion of this theoretical problem with a very practical issue: indeed we show that MALL$^-$ proofs are closely related to binary decision diagrams.

Binary decision diagrams

The problem of the representation of Boolean functions is of central importance in circuit design and has a large range of practical applications. Over the years, binary decision diagrams (BDD) [2] became the most widely used data structure to treat this question.

Roughly speaking, a BDD is a binary tree with nodes labeled by Boolean variables and leaves labeled by values 0 and 1. Such a tree represents a Boolean function in the sense that once an assignment of the variable is chosen, then following the left or right path at each node according to the value 0 or 1 chosen for its variable eventually leads to a leave, which is the output of the function.

This representation has many advantages which justify its popularity [15]: most basic operations (negation and other logical connectives) on BDD can be implemented efficiently, in many practical cases the size of the BDD representing a Boolean function remains compact (thanks to the possibility to have shared subtrees) and once a variable ordering is chosen they enjoy a notion of normal form.

In this article, we consider both BDD and ordered BDD with no sharing of subtrees and write them as special kinds of Boolean functions by introducing an `IfThenElse` constructor. However, when manipulating them from a complexity point of view we will keep the binary tree presentation in mind.

[1] Of course, this applies only to the standard formulation of units: the equivalence problem for any notion of multiplicative units enjoying less permutations of rules could potentially still be tractable via proofnets: see for instance the work of S. Guerrini and A. Masini [8] and D. Hughes [11].

AC_0 reductions

To show that a problem is complete for some complexity class C, one needs to specify the notion of reduction functions considered, and of course this needs to be a class of functions supposed to be smaller than C itself (indeed *any* problem in C is complete under C reductions).

A standard notion of reduction for the class Logspace is (uniform) AC_0 reduction [3], formally defined in terms of uniform circuits of fixed depth and unbounded fan-in. We will not be getting in the details about this complexity class and, as we will consider only graph transformations, we will rely on the following intuitive principle: if a graph transformation locally replaces each vertex by a bounded number of vertices and the replacement depends *only* on the vertex considered and eventually its direct neighbors, then the transformation is in AC_0. Typical examples of such a transformation are certain simple cases of so-called "gadget" reductions used in complexity theory to prove hardness results.

Outline of the paper

Section 2 covers some background material on MALL$^-$ and notions of proofnet for this fragment: monomial proofnets and the associated vocabulary for Boolean formulas and BDD in Section 2.1 and the notion of slicing in Section 2.2. Then, we introduce in Section 2.3 an intermediary notion of proof representation that will help us to relate proofs in MALL$^-$ and BDD. In Section 3, we prove that proof equivalence in MALL$^-$ and equivalence of BDD relate to each other through AC_0 reductions and that they are both Logspace-complete.

2 Proof equivalence in MALL$^-$

▶ **Notation 1.** *The formulas of* MALL$^-$ *are built inductively from atoms which we write* $\alpha, \beta, \gamma, \dots$ *their duals* $\alpha^\star, \beta^\star, \gamma^\star, \dots$ *and the binary connectives* $\parr, \otimes, \&_x, \oplus$ (we consider that the $\&$ connectives carry a label x to simplify some reasonings, but we will omit it when it is not relevant). *We write formulas as uppercase letters* $A, B, C, \dots$ *unless we want to specify they are atoms.* Sequents *are sequences of formulas, written as greek uppercase letters* $\Gamma, \Delta, \Lambda, \dots$ *such that all occurrences of the connective* $\&$ *in a sequent carry a different label. The concatenation of two sequents* Γ *and* Δ *is simply written* Γ, Δ.

Let us recall the rules of MALL$^-$[2]. We do not include the `cut` rule in our study, since in a static situation (we are not looking at the `cut`-elimination procedure of MALL$^-$) it can always be encoded w.l.o.g. using the $\otimes$ rule.

$$\alpha, \alpha^\star \qquad \frac{\Gamma, A, B}{\Gamma, A \parr B}\parr \qquad \frac{\Gamma, A \qquad \Delta, B}{\Gamma, \Delta, A \otimes B}\otimes \qquad \frac{\Gamma, A}{\Gamma, A \oplus B}\oplus_1 \qquad \frac{\Gamma, B}{\Gamma, A \oplus B}\oplus_r \qquad \frac{\Gamma, A \qquad \Gamma, B}{\Gamma, A \,\&_x\, B}\&_x$$

(by convention, we leave the axiom rule implicit to lighten notations. Also, we will use the notation $\begin{matrix}\langle\pi\rangle\\ \Gamma\end{matrix}$ *for "the proof* π *of conclusion* Γ*".)*

▶ Remark 2. Any time we will look at a MALL$^-$ proof from a complexity perspective, we will consider they are represented as trees with nodes corresponding to rules, labeled by the connective introduced and the sequent that is the conclusion of the rule.

[2] We consider a η-expanded version of MALL$^-$, which simplifies proofs and definitions, but the extension of our results to a version with non-atomic axioms would be straightforward. Also, we work *modulo* the exchange rule.

Two MALL⁻ proofs π and ν are said to be *equivalent* (notation $\pi \sim \nu$) if one can pass from one to the other via permutations of rules [14]. We have an associated decision problem.

▶ **Definition 3** (MALL⁻**equ**). MALL⁻**equ** is the decision problem:

"Given two MALL⁻ *proofs π and ν with the same conclusion, do we have $\pi \sim \nu$?"*

We will not go through all the details about this syntactic way to define proof equivalence in MALL⁻. The reason for this is that we already have an available equivalent characterization in terms of *slicing* [14] which we review in Section 2.2. Instead, let us focus only on the most significant case.

$$\dfrac{\dfrac{\langle\pi\rangle \quad \langle\mu\rangle}{\Gamma, A, C \quad \Gamma, B, C}\,\&}{\Gamma, A \,\&\, B, C} \quad \langle\nu\rangle \;\; \Delta, D \;\otimes}{\Gamma, \Delta, A \,\&\, B, C \otimes D} \quad \sim \quad \dfrac{\dfrac{\langle\pi\rangle \quad \langle\nu\rangle}{\Gamma, A, C \quad \Delta, D}\,\otimes \qquad \dfrac{\langle\mu\rangle \quad \langle\nu\rangle}{\Gamma, B, C \quad \Delta, D}\,\otimes}{\Gamma, \Delta, A, C \otimes D \qquad \Gamma, \Delta, B, C \otimes D}\,\&}{\Gamma, \Delta, A \,\&\, B, C \otimes D}$$

In the above equivalence, the $\otimes$ rule gets lifted above the $\&$ rule. But doing so, notice that we created two copies of ν instead of one, therefore the size of the prooftree has grown. Iterating on this observation, it is not hard to build pairs of proofs that are equivalent, but one of which is exponentially bigger than the other. This is indeed where the difficulty of proof equivalence in MALL⁻ lies. As a matter of fact, this permutation of rules *alone* would be enough to build the encoding of the equivalence problem of binary decision diagrams presented in Section 3.1.

A way to attack proof equivalence in a logic, as we exposed in Section 1, is to try to setup a notion of proofnet for this logic. In the following, we will review the main two approaches to this idea in the case of MALL⁻: *monomial proofnets* by J.-Y. Girard [7, 16] and *slicing proofnets* by D. Hughes and R. van Glabbeek [13, 14]. We will then design an intermediate notion of BDD *slicing* that will be more suited to our needs.

2.1 Monomial proofnets

The first attempt in the direction of a notion of proofnet for MALL⁻ is due to J.-Y.Girard [7], followed by a version with a full `cut`-elimination procedure by O. Laurent and R. Maielli [16].

While proofnets for multiplicative linear logic without units were introduced along linear logic itself [6], extending the notion to the multiplicative-additive without units fragment proved to be a true challenge, mainly because of the *superposition* at work in the $\&$ rule.

Girard's idea was to represent the superposed "versions" of the proofnet by attributing a Boolean formula (called a *weight*) to each link, with one Boolean variable for each $\&$ connective in the conclusion Γ. To retrieve the version of the proofnet corresponding to some selection of the left/right branches of each $\&$, one then just needs to evaluate the Boolean formulas with the corresponding valuation of their variables.

This is the occasion to introduce the vocabulary to speak about Boolean formulas.

▶ **Definition 4** (Boolean formula). Given a finite set of variables $V = \{x_1, \ldots, x_n\}$, a *Boolean formula* over V is inductively defined from the elements of V; the constants 0 (*"false"*) and 1(*"true"*); the unary symbol $\bar{\cdot}$ (*"negation"*); the binary symbols $+$ and $.$ (*"sum/or/disjuction"* and *"product/and/conjunction"* respectively).

We consider a syntactic notion of *equality* of Boolean formulas: for instance $0.x \neq 0$. The real important notion, that we therefore state separately, is *equivalence*: the fact that if we replace the variables with actual values, we gets the same output.

▶ **Definition 5** (equivalence)**.** A *valuation* v of V is a choice of 0 or 1 for any element of V. A valuation induces a an *evaluation* function $v(\cdot)$ from Boolean formulas over V to $\{0,1\}$ in the obvious way. Two Boolean formulas ϕ and ψ over V are *equivalent* (notation $\phi \sim \psi$) when for any valuation v of V, we have $v(\phi) = v(\psi)$.

▶ **Definition 6** (monomial)**.** We write $\overline{V} = \{\overline{x_1}, \dots, \overline{x}_n\}$. A *monomial* over V is a Boolean formula of the form $y_1. \dots .y_k$ with $\{y_1, \dots, y_k\} \subseteq V \cup \overline{V}$.

Two monomials m and m' are in *conflict* if $m.m' \sim 0$.

▶ Remark 7. Two monomials are in conflict if and only if there is a variable x such that x appears in one of them and $\overline{x}$ appears in the other.

While monomials are a specific type of Boolean formula, the *binary decision diagrams* we are about to introduce are not defined directly as Boolean formulas. Of course, they relate to each other in an obvious way, but having a specific syntax for binary decision diagrams will prove more convenient to solve the problems we will be facing.

▶ **Definition 8** (BDD)**.** A *binary decision diagram* (BDD) is defined inductively as:

- The constants 0 and 1 are BDD
- If ϕ, ψ are BDD and X is either a variable, 0 or 1, `If` X `Then` ϕ `Else` ψ is a BDD
- If ϕ is a BDD and X is either a variable, 0 or 1, `DontCare` X `Then` ϕ is a BDD

Moreover, suppose we have an ordered set of variables $V = \{x_1, \dots, x_n\}$ with the convention that variables are listed in the reverse order: x_n is first, then x_{n-1}, etc. We define a subclass of BDD which we call *ordered binary decision diagrams over* V (°BDD/V) by restricting to the following inductive cases (we let $V' = \{x_1, \dots, x_{n-1}\}$)

- The constants 0 and 1 are °BDD/$\varnothing$
- If ϕ and ψ are °BDD/V', `If` x_n `Then` ϕ `Else` ψ is a °BDD/V
- If ϕ is a °BDD/V', `DontCare` x_n `Then` ϕ is a °BDD/V

The notions of valuation and equivalence are extended to BDD and °BDD the obvious way so that `DontCare` X `Then` $\phi \sim \phi$ and `If` X `Then` ϕ `Else` $\psi \sim X.\phi + \overline{X}.\psi$.

▶ Remark 9. Any time we will look at BDD and °BDD from a complexity perspective, we will consider they are represented as labeled trees. The cases of `If` 0, `DontCare` 1,... will be useful to obtain AC_0 reductions in Section 2.3 and Section 3.2, since erasing a whole subpart of a graph of which we do not know the address in advance is not something that is doable in this complexity class. The absence of sharing implied by the tree representation is also crucial to get low-complexity reductions.

▶ **Example 10.** The Boolean formula $x.y.\overline{z}$ is a monomial, while $x.y + z$ and $x.1$ are not.

The BDD `If` x_2 `Then` (`If` x_1 `Then` 0 `Else` 1) `Else` 1 (which is *not* a °BDD/$\{x_1, x_2\}$ by the way) is equivalent to the Boolean formula $x_2.\overline{x_1} + \overline{x_2}$: both evaluate to 1 only for the valuations $\{x_1 \mapsto 1, x_2 \mapsto 0\}$, $\{x_1 \mapsto 0, x_2 \mapsto 0\}$ and $\{x_1 \mapsto 0, x_2 \mapsto 1\}$. There exist an equivalent °BDD/$\{x_1, x_2\}$: `If` x_2 `Then` (`If` x_1 `Then` 0 `Else` 1) `Else` (`DontCare` x_1 `Then` 1).

▶ **Definition 11.** BDD**equ** is the following decision problem:

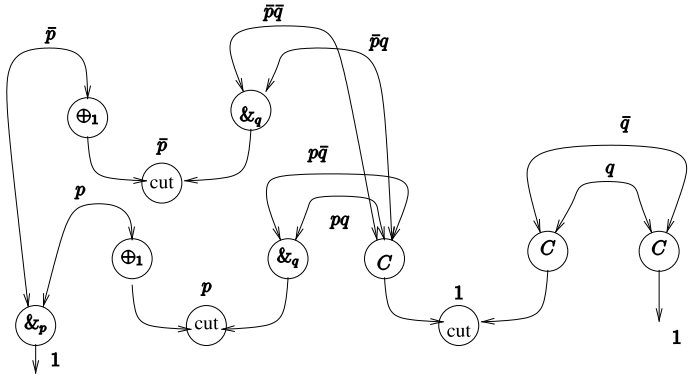
"given two BDD ϕ *and* ψ*, do we have* $\phi \sim \psi$*?"*

°BDD**equ** is the following decision problem:

"given two °BDD/V ϕ *and* ψ*, do we have* $\phi \sim \psi$*?"*

Girard's proofnets are called monomial because the only Boolean formulas that are allowed are monomials. This is, as of the state of the art, the only known way to have a notion of proofnet that enjoys a satisfying correctness criterion.

For our purposes, we do not need to get into the technical details of monomial proofnets. Still, let us end this section with an example of proofnet from the article by Laurent and Maielli, where the monomial weight of a link is pictured just above it:



2.2 Slicings and proof equivalence

The idea of slicing dates back to J.-Y. Girard's original article on proofnets for MALL⁻ [7], and was even present in the original article on linear logic [6]. It amounts to the natural point of view already evoked in Section 2.1, seeing the & rule as introducing superposed variants of the proof, which are eventually to be selected from in the course of `cut`-elimination. If we have two alternative *slices* for each & connective of a sequent Γ and all combinations of slices can be selected independently, we readily see that the global number of slices will be exponential in the number of & connectives in Γ.

This is indeed the major drawback of the representation of proofs as set of slices: the size of objects representing proofs may grow exponentially in the size of the original proofs. This of course impairs any fine-grained analysis in terms of complexity.

▶ **Definition 12** (slicing). Given a MALL⁻ sequent Γ, a *linking* of Γ is a subset

$$\big\{[\alpha_1, \alpha_1^\star], \ldots, [\alpha_n, \alpha_n^\star]\big\}$$

of the set of (unordered) pairs of occurrences of dual atoms in Γ.

Then, a *slicing* of Γ is a finite set of linkings of Γ.

To any MALL⁻ proof π, we associate a slicing $\mathcal{S}_\pi$ by induction:

- If $\pi = \alpha, \alpha^\star$ then $\mathcal{S}_\pi$ is the set containing only the linking $\big\{[\alpha, \alpha^\star]\big\}$
- If $\pi = \dfrac{\langle\mu\rangle}{\Gamma, A, B}\,{}^{\parr}$ then $\mathcal{S}_\pi = \mathcal{S}_\mu$, where we see atoms of $A \parr B$ as the corresponding atoms of A and B
- If $\pi = \dfrac{\langle\mu\rangle \quad \langle\nu\rangle}{\Gamma, A \quad \Delta, B}\,{}^{\otimes}$ then $\mathcal{S}_\pi = \{\, \lambda \cup \lambda' \mid \lambda \in \mathcal{S}_\mu\,,\ \lambda' \in \mathcal{S}_\nu \,\}$
- If $\pi = \dfrac{\langle\mu\rangle}{\Gamma, A}\,{}^{\oplus_1}$ then $\mathcal{S}_\pi = \mathcal{S}_\mu$, and likewise for $\oplus_{\mathtt{r}}$

- If $\pi = \dfrac{\begin{array}{cc}\langle\mu\rangle & \langle\nu\rangle \\ \Gamma, A & \Gamma, B\end{array}}{\Gamma, \Delta, A \,\&\, B}\,\&$ then $\mathcal{S}_\pi = \mathcal{S}_\mu \cup \mathcal{S}_\nu$

▶ Remark 13. In the $\otimes$ rule, it is clearly seen that the number of slices are multiplied. This is just what is needed in order to have a combinatorial explosion: for any n, a proof π_n of

$$\underbrace{\alpha^\star \otimes \cdots \otimes \alpha^\star}_{n \text{ times}}, \underbrace{\alpha \,\&\, \alpha, \ \ldots\ , \alpha \,\&\, \alpha}_{n \text{ times}}$$

obtained by combining with the $\otimes$ rule n copies of the proof

$$\frac{\alpha^\star, \alpha \qquad \alpha^\star, \alpha}{\alpha^\star, \alpha \,\&\, \alpha}\,\&$$

will be of linear size in n, but with a slicing $\mathcal{S}_n$ containing 2^n linkings.

Slicings (associated to a proof) correspond exactly to the notion of proofnets elaborated by D. Hughes and R. van Glabbeek [13]. While their study was mainly focused on the problems of finding a correctness criterion and designing a `cut`-elimination procedure for these, it also covers the proof equivalence problem. The proof that their notion of proofnet characterizes MALL⁻ proof equivalence can be found in an independent note [14].

▶ **Theorem 14** (slicing equivalence [14, Theorem 1]). *Let π and ν be two* MALL⁻ *proofs. We have that $\pi \sim \nu$ if and only if $\mathcal{S}_\pi = \mathcal{S}_\nu$.*

Let us also end this section with a graphical representation of an example of proofnet from the article of Hughes and van Glabbeek, encoding the proof on the left with three linkings $\lambda_1, \lambda_2, \lambda_3$:

$$\dfrac{\dfrac{\dfrac{P^\star, P}{P^\star \oplus Q^\star, P}\oplus_1}{(P^\star \oplus Q^\star) \oplus R^\star, P}\oplus_1 \quad \dfrac{\dfrac{Q^\star, Q}{P^\star \oplus Q^\star, Q}\oplus_\mathrm{r}}{(P^\star \oplus Q^\star) \oplus R^\star, P}\oplus_1}{(P^\star \oplus Q^\star) \oplus R^\star, P \,\&\, Q}\,\& \quad \dfrac{R^\star, R}{(P^\star \oplus Q^\star) \oplus R^\star, P}\oplus_\mathrm{r}}{(P^\star \oplus Q^\star) \oplus R^\star, (P \,\&\, Q) \,\&\, R}\,\&$$

λ_1 : $(P^\perp \oplus Q^\perp) \oplus R^\perp,\ P \,\&\, (Q \,\&\, R)$

λ_2 : $(P^\perp \oplus Q^\perp) \oplus R^\perp,\ P \,\&\, (Q \,\&\, R)$

λ_3 : $(P^\perp \oplus Q^\perp) \oplus R^\perp,\ P \,\&\, (Q \,\&\, R)$

2.3 BDD slicings

We finally introduce an intermediate notion of representation of proofs which will be a central tool in the next section. In a sense, it is a synthesis of monomial proofnets and slicings: acknowledging the fact that slicing makes the size of the representation explode, we rely on BDD to keep things more compact.

Of course, the canonicity property is lost. But this is exactly the point! Indeed, deciding whether two "BDD slicings" are equivalent is the reformulation of proof equivalence in MALL⁻ we rely on in the reductions between MALL⁻**equ** (Definition 3) and BDD**equ** (Definition 11).

▶ **Definition 15** (BDD slicing). Given a MALL⁻ sequent Γ, a BDD *slicing* of Γ is a function $\mathcal{B}$ that associates a BDD to every element $[\gamma, \gamma^\star]$ of the set of (unordered) pairs of occurrences of dual atoms in Γ.

We say that two BDD slicings M, N of the same Γ are *equivalent* (notation $M \sim N$) if for any pair $[\gamma, \gamma^\star]$, we have $M[\gamma, \gamma^\star] \sim N[\gamma, \gamma^\star]$ in the sense of Definition 5.

To any MALL⁻ proof π, we associate a BDD slicing $\mathcal{B}_\pi$ by induction:

- If $\pi = \alpha, \alpha^\star$ then $\mathcal{B}_\pi[\alpha, \alpha^\star] = 1$.

- If $\pi = \dfrac{\langle\mu\rangle}{\dfrac{\Gamma, A, B}{\Gamma, A \parr B}\parr}$ then $\mathcal{B}_\pi[\gamma, \gamma^\star] = \mathcal{B}_\mu[\gamma, \gamma^\star]$ where we see atoms of $A \parr B$ as the corresponding atoms of A and B
- If $\pi = \dfrac{\dfrac{\langle\mu\rangle}{\Gamma, A} \quad \dfrac{\langle\nu\rangle}{\Delta, B}}{\Gamma, \Delta, A \otimes B}\otimes$ then $\mathcal{B}_\pi[\gamma, \gamma^\star] = \begin{cases} \mathcal{B}_\mu[\gamma, \gamma^\star] & \text{if } \gamma, \gamma^\star \text{ are atoms of } \Gamma, A \\ \mathcal{B}_\nu[\gamma, \gamma^\star] & \text{if } \gamma, \gamma^\star \text{ are atoms of } \Delta, B \\ 0 & \text{otherwise}^3 \end{cases}$
- If $\pi = \dfrac{\dfrac{\langle\mu\rangle}{\Gamma, A}}{\Gamma, A \oplus B}\oplus_1$ then $\mathcal{B}_\pi[\gamma, \gamma^\star] = \begin{cases} \mathcal{B}_\mu[\gamma, \gamma^\star] & \text{if } \gamma, \gamma^\star \text{ are atoms of } \Gamma, A \\ 0 & \text{otherwise} \end{cases}$ and likewise for $\oplus_{\mathtt{r}}$.
- If $\pi = \dfrac{\dfrac{\langle\mu\rangle}{\Gamma, A} \quad \dfrac{\langle\nu\rangle}{\Gamma, B}}{\Gamma, \Delta, A \,\&_x B}\&_x$ then

$$\mathcal{B}_\pi[\gamma, \gamma^\star] = \begin{cases} \mathtt{If}\ x\ \mathtt{Then}\ \mathcal{B}_\mu[\gamma, \gamma^\star]\ \mathtt{Else}\ 0 & \text{if } \gamma \text{ or } \gamma^\star \text{ is an atom of } A \\ \mathtt{If}\ x\ \mathtt{Then}\ 0\ \mathtt{Else}\ \mathcal{B}_\nu[\gamma, \gamma^\star] & \text{if } \gamma \text{ or } \gamma^\star \text{ is an atom of } B \\ \mathtt{If}\ x\ \mathtt{Then}\ \mathcal{B}_\mu[\gamma, \gamma^\star]\ \mathtt{Else}\ \mathcal{B}_\nu[\gamma, \gamma^\star] & \text{otherwise} \end{cases}$$

▶ Remark 16. The BDD we obtain this way are actually of a specific type: they are usually called *read-once* BDD: from the root to any leave, one never crosses two `IfThenElse` nodes asking for the value of the same variable.

▶ **Example 17.** The weight of the pairs $[\alpha, \alpha^\star]$ and $[\delta, \delta^\star]$ in the BDD slicing of the proof

$$\pi = \dfrac{\dfrac{\dfrac{\alpha, \alpha^\star}{\alpha \oplus \beta, \alpha^\star}\oplus_1 \quad \dfrac{\beta, \beta^\star}{\alpha \oplus \beta, \beta^\star}\oplus_{\mathtt{r}}}{\alpha \oplus \beta, \alpha^\star \,\&_x \beta^\star}\&_x \quad \delta, \delta^\star}{\alpha \oplus \beta, (\alpha^\star \,\&_x \beta^\star) \otimes \delta, \delta^\star}\otimes$$

are $\mathcal{B}_\pi[\alpha, \alpha^\star] = \mathtt{If}\ x\ \mathtt{Then}\ 1\ \mathtt{Else}\ 0$ and $\mathcal{B}_\pi[\delta, \delta^\star] = 1$.

It is not hard to see that proof equivalence matches the equivalence of BDD slicings by relating them to slicings from the previous section.

▶ **Theorem 18** (BDD slicing equivalence)**.** *Let π and ν be two* MALL$^-$ *proofs. We have that $\pi \sim \nu$ if and only if $\mathcal{B}_\pi \sim \mathcal{B}_\nu$.*

Proof. We show in fact that $\mathcal{S}_\pi = \mathcal{S}_\nu$ if and only if $\mathcal{B}_\pi \sim \mathcal{B}_\nu$, with Theorem 14 in mind.

To a BDD slicing $\mathcal{B}$, we can associate a linking $v(\mathcal{B})$ for each valuation v of the variables occurring in $\mathcal{B}$ by setting $v(\mathcal{B}) = \{\ [\alpha, \alpha^\star] \mid v(\mathcal{B}[\alpha, \alpha^\star]) = 1\ \}$ and then a slicing $f(\mathcal{B}) = \{\ v(\mathcal{B}) \mid v \text{ valuation}\ \}$. By definition, it is clear that if $\mathcal{B}$ and $\mathcal{B}'$ involve the same variables and $\mathcal{B} \sim \mathcal{B}'$ then $f(\mathcal{B}) = f(\mathcal{B}')$.

Conversely, suppose $\mathcal{B}_\pi \not\sim \mathcal{B}_\nu$, so that there is a v such that $v(\mathcal{B}_\pi) \neq v(\mathcal{B}_\nu)$. To conclude that $f(\mathcal{B}_\pi) \neq f(\mathcal{B}_\nu)$, we must show that there is no other v' such that $v'(\mathcal{B}_\nu) = v(\mathcal{B}_\pi)$.

To do this, we can extend the notion of valuation to proofs: if v is a valuation of the labels x of the $\&_x$ in π, $v(\pi)$ is defined by keeping only the left or right branch of $\&_x$ according to the value of x. Now we can consider the set $v^{\mathtt{ax}}(\pi)$ of axiom rules in $v(\pi)$ and we can show by

[3] Remember we consider *occurrences* of atoms, and as the $\otimes$ rule splits the context into two independent parts that and no axiom rule can cross this splitting.

induction that $v^{\mathtt{ax}}(\pi)$ must contain at least one pair with one atom which is a subformula of the side of each $\&_x$ that has been kept. Therefore for any π and ν with the same conclusion, if $v \neq v'$ we have $v^{\mathtt{ax}}(\nu) \neq v'^{\mathtt{ax}}(\pi)$ no matter what. Then we can remark that $v^{\mathtt{ax}}(\pi)$ is just another name for $v(\mathcal{B}_\pi)$ so that in the end, there cannot be $v \neq v'$ such that $v'(\mathcal{B}_\nu) = v(\mathcal{B}_\pi)$.

Finally, an easy induction shows that $f(\mathcal{B}_\pi) = \mathcal{S}_\pi$ and therefore we are done. ◀

Also, a BDD equivalent to the BDD associated to a pair can be computed in AC_0.

▶ **Lemma 19** (computing BDD slicings). *For any* MALL⁻ *proof* π *and any pair* $[\gamma, \gamma^\star]$*, we can compute in* AC_0 *a* BDD ϕ *such that* $\phi \sim \mathcal{B}_\pi[\gamma, \gamma^\star]$.

Proof. As we see proofs as labeled trees (Remark 2), we will only locally replace the rules of the proof the following way to obtain the corresponding BDD ϕ:

- Replace axiom rules $\gamma, \gamma^\star$ by 1 and other axiom rules by 0
- Replace all $\parr$ and $\oplus$ rules by `DontCare` 1 `Then` $\cdot$ nodes
- In the $\otimes$ case, test which side the atoms $\gamma, \gamma^\star$ are attributed to *(this can be done locally by looking at the conclusions of the premise of the rule)* and replace it by a `If` 0 `Then` $\cdot$ `Else` $\cdot$ or a `If` 1 `Then` $\cdot$ `Else` $\cdot$ node accordingly
- Replace $\&_x$ rules by a `If` x `Then` $\cdot$ `Else` $\cdot$ nodes

We can see by induction that the resulting BDD is equivalent to $\mathcal{B}_\pi[\gamma, \gamma^\star]$. All these operations can be performed by looking only at the rule under treatment (and its immediate neighbors in the case of $\otimes$) and always replaces one rule by exactly one node. Therefore it is in AC_0. ◀

▶ **Corollary 20** (reduction). MALL⁻**equ** *reduces to* BDD**equ** *in* AC_0.

In the next section we focus on the equivalence of BDD and °BDD, proving first that the case of °BDD can be reduced to proof equivalence in MALL⁻. Then, we will show the problem of equivalence of BDD to be in Logspace, and that of °BDD to be Logspace-hard, thus characterizing the intrinsic complexity of proof equivalence in MALL⁻ as Logspace-complete.

Note that this contrasts with the classical result that equivalence of general Boolean formulas is coNP-complete. It turns out indeed that the classes of BDD we consider enjoy a number of properties that allow to solve equivalence more easily.

3 Equivalence of BDD

3.1 Equivalence of °BDD reduces to proof equivalence in MALL⁻

We now show that the converse of Lemma 19 holds for °BDD.

To do this, we rely on a formula B which will serve as the placeholder of a °BDD/V ϕ we want to encode; and a context Γ which contains one $\&_x$ connective for each variable x in V, organized in a way that allows for an inductive specification of °BDD.

Given an °BDD ϕ, we wish to obtain a proof π_ϕ of B, Γ such that dual pairs with one element in B will receive either the value ϕ or a value equivalent to $\overline{\phi}$ in the BDD slicing of π_ϕ; and on the other hand, the other dual pairs of Γ will receive equivalent values whatever the °BDD we encode is. This will lead to the fact that two such encoding proofs are equivalent if and only if the °BDD they encode are equivalent.

▶ **Notation 21.** *We fix atomic formulas* $\alpha_1, \ldots, \alpha_n \ldots$ *and* β *and write* $B = \beta \oplus \beta$*. In what follows, we will use* $\beta^{\mathtt{l}}$ *and* $\beta^{\mathtt{r}}$ *to refer respectively to the left and right copies of* β *in* B*; and likewise* $\alpha_i^{\mathtt{l}}$ *and* $\alpha_i^{\mathtt{r}}$ *for copies of* α_i *in* $\alpha_i \,\&\, \alpha_i$*.*

We write respectively π_0 *and* π_1 *the proofs*

$$\frac{\beta, \beta^\star}{B, \beta^\star}\oplus_1 \qquad \frac{\beta, \beta^\star}{B, \beta^\star}\oplus_\mathtt{r}$$

and for any n, we write $\pi^n_\&$ the proof

$$\frac{\alpha^\star_n, \alpha_n \qquad \alpha^\star_n, \alpha_n}{\alpha^\star_n, \alpha_n \,\&\, \alpha_n}\&$$

▶ **Definition 22** (encoding a °BDD)**.** Let ϕ be an °BDD over the variables $V = \{x_1, \dots, x_n\}$. We define the sequent

$$\Gamma^n = \ \big(\cdots(\beta^\star \otimes \alpha^\star_1) \otimes \alpha^\star_2) \otimes \cdots\big) \otimes \alpha^\star_n \ , \ \alpha_1 \,\&_{x_1} \alpha_1 \ , \ \alpha_2 \,\&_{x_2} \alpha_2 \ , \ \dots \ , \ \alpha_{n-1} \,\&_{x_{n-1}} \alpha_{n-1}$$

with $\Gamma^0 = \beta^\star$ and set $\Delta^n = \Gamma^n, \alpha_n \,\&_{x_n} \alpha_n$ with also $\Delta^0 = \beta^\star$.

We define the proof π_ϕ of conclusion B, Δ^n by induction on n

- The base cases are 0 and 1, encoded respectively as π_0 and π_1.
- Otherwise, if $\phi = \mathtt{If}\ x_n\ \mathtt{Then}\ \psi\ \mathtt{Else}\ \zeta$, with both ψ and ζ being °BDD$/\{x_1, \dots, x_{n-1}\}$, we have π_ψ and π_ζ defined by induction, and then
$$\pi_\phi = \ \dfrac{\dfrac{\langle\pi_\psi\rangle \qquad \langle\pi_\zeta\rangle}{B, \Gamma^n, \alpha_n}\otimes}{B, \Delta^n}\&_n$$
- The last case is $\phi = \mathtt{DontCare}\ x_n\ \mathtt{Then}\ \psi$, with ψ being a °BDD$/\{x_1, \dots, x_{n-1}\}$ so that we have π_ψ defined by induction, and then
$$\pi_\phi = \ \dfrac{\langle\pi_\psi\rangle \qquad \langle\pi^n_\&\rangle}{B, \Delta^n}\otimes$$

We still need to state in what sense π_ϕ is an encoding of ϕ: we turn the statement about the value of atoms of B we made in the beginning of this section into a precise property.

▶ **Lemma 23** (associated BDD slicing)**.** *Writing $\mathcal{B}$ the* BDD *slicing of π_ϕ, we have*

$$\mathcal{B}[\beta^\mathtt{l}, \beta^\star] = \phi \qquad \mathcal{B}[\beta^\mathtt{r}, \beta^\star] \sim \overline{\phi} \qquad \mathcal{B}[\alpha^\star_i, \alpha^\mathtt{l}_i] \sim x_i \qquad \mathcal{B}[\alpha^\star_i, \alpha^\mathtt{r}_i] \sim \overline{x}_i$$

Proof. By a routine inspection of induction cases. Let us only review $\phi = \mathtt{If}\, x_n\, \mathtt{Then}\, \psi\, \mathtt{Else}\, \zeta$. Let us write $\mathcal{B}_\phi$, $\mathcal{B}_\psi$ and $\mathcal{B}_\zeta$ the BDD slicings respectively associated to the proofs π_ϕ, π_ψ and π_ζ.

By induction we have that $\mathcal{B}_\psi[\beta^\mathtt{l}, \beta^\star] = \psi$ and $\mathcal{B}_\zeta[\beta^\mathtt{l}, \beta^\star] = \zeta$, therefore by definition of the BDD slicing associated to a & rule, we have that $\mathcal{B}_\phi[\beta^\mathtt{l}, \beta^\star] = \mathtt{If}\ x_n\ \mathtt{Then}\ \psi\ \mathtt{Else}\ \zeta = \phi$. The case of $\beta^\mathtt{r}$ is similar, but for its use of Lemma 27 from the next section.

As for the other pairs of occurences of dual atoms in the conclusion, let us have a look at the case of $[\alpha^\star_i, \alpha^\mathtt{l}_i]$: if by induction $\mathcal{B}_\psi[\alpha^\star_i, \alpha^\mathtt{l}_i] \sim x_i$ and $\mathcal{B}_\zeta[\alpha^\star_i, \alpha^\mathtt{l}_i] \sim x_i$, then by definition $\mathcal{B}_\phi[\alpha^\star_i, \alpha^\mathtt{l}_i] \sim \mathtt{If}\ x_n\ \mathtt{Then}\ x_i\ \mathtt{Else}\ x_i \sim x_i$. The case of $\alpha^\mathtt{r}_i$ is similar. ◀

▶ **Corollary 24** (equivalence)**.** *If ϕ and ψ are two* °BDD$/\{x_1, \dots, x_n\}$*, then $\pi_\phi \sim \pi_\psi$ if and only if $\phi \sim \psi$.*

▶ **Lemma 25** (computing the encoding)**.** *Given a* °BDD$/\{x_1, \dots, x_n\}$ *ϕ, π_ϕ can be computed in* AC_0*.*

Proof. As in the proof of Lemma 19, we show that the inductive definition can in fact be seen as a local graph transformation introducing nodes of bounded size:

- Replace each 0 node by $\dfrac{\langle\pi_0\rangle}{\vdots\ B, \beta^\star}$ and 1 node by $\dfrac{\langle\pi_1\rangle}{\vdots\ B, \beta^\star}$.
- Replace each $\texttt{DontCare}\ x_i\ \texttt{Then}\ \cdot$ by $\dfrac{\dfrac{\langle\pi_{\&}^{x_i}\rangle}{\vdots\ \ \alpha_i^\star, \alpha_i \,\&_{x_i}\, \alpha_i}\otimes}{B, \Delta^i}$
- Replace each $\texttt{If}\ x_i\ \texttt{Then}\ \cdot\ \texttt{Else}\ \cdot$ by $\dfrac{\dfrac{\vdots\ \ \alpha_i^\star, \alpha_i}{B, \Gamma^i, \alpha_i}\otimes \quad \dfrac{\vdots\ \ \alpha_i^\star, \alpha_i}{B, \Gamma^i, \alpha_i}\otimes}{B, \Delta^i}\,\&_{x_i}$

For any of these replacements, we see that the choice of the case to apply and the label of the resulting block (of bounded size) of rules replacing a node depends only on the label of the node we are replacing, therefore the transformation is in AC_0. ◀

▶ **Corollary 26** (reduction). °BDD**equ** *reduces to* MALL⁻**equ** *in* AC_0.

To sum up, we have so far the following chain of AC_0 reductions:

$$\text{°BDD}\mathbf{equ} \;\to\; \text{MALL}^{-}\mathbf{equ} \;\to\; \text{BDD}\mathbf{equ}$$

3.2 Logspace-completeness

We prove in this section that all these equivalence problems are Logspace-complete. We begin by listing a few useful properties of BDD that will allow to design a Logspace decision procedure for their equivalence. Then, we prove the Logspace-hardness by reducing to °BDD**equ** a Logspace-complete problem on line graph orderings.

The starting point is the good behavior of BDD with respect to negation. In the following lemma, we consider the negation of a BDD which is not strictly speaking a BDD itself: we think of it as the equivalent Boolean formula, the point being precisely to show that this Boolean formula can be easily expressed as a BDD.

▶ **Lemma 27** (negation). *If ϕ and ψ are* BDD *and X is either 0, 1 or a variable, we have*

$$\overline{\texttt{If}\ X\ \texttt{Then}\ \phi\ \texttt{Else}\ \psi} \sim \texttt{If}\ X\ \texttt{Then}\ \overline{\phi}\ \texttt{Else}\ \overline{\psi}$$

$$\overline{\texttt{DontCare}\ X\ \texttt{Then}\ \phi} \sim \texttt{DontCare}\ X\ \texttt{Then}\ \overline{\phi}$$

Proof. First we can transform our expression by

$$\overline{\texttt{If}\ X\ \texttt{Then}\ \phi\ \texttt{Else}\ \psi} \sim \overline{X.\phi + \overline{X}.\psi} \sim (\overline{X} + \overline{\phi}).(X + \overline{\psi}) \sim X.\overline{\psi} + \overline{X}.\overline{\phi} + \psi.\phi$$

then we can apply the so-called "consensus rule" of Boolean formulas

$$X.\overline{\psi} + \overline{X}.\overline{\phi} + \psi.\phi \sim X.\overline{\phi} + \overline{X}.\overline{\psi} \sim \texttt{If}\ X\ \texttt{Then}\ \overline{\phi}\ \texttt{Else}\ \overline{\psi}$$

The case of `DontCare` is obvious. ◀

▶ **Corollary 28.** *If ϕ is a* BDD*, then there is a* BDD *$\widehat{\phi}$ such that $\overline{\phi} \sim \widehat{\phi}$.*
Moreover $\widehat{\phi}$ can be computed in logarithmic space.

Proof. An induction on the previous lemma shows that we can obtain the negation of a BDD simply by flipping the 0 nodes to 1 nodes and conversely. Hence the transformation is even in AC_0. ◀

Then, we show that a BDD can be seen as a sum of monomials through a Logspace transformation.

▶ **Lemma 29** (BDD as sums of monomials). *If ϕ is a* BDD*, then there is a formula $\phi_{\mathtt{m}}$ which is a sum of monomials and is such that $\phi \sim \phi_{\mathtt{m}}$.*

Moreover $\phi_{\mathtt{m}}$ can be computed in logarithmic space.

Proof. For each 1 node in ϕ, go down to the root of ϕ and output one by one the variables of any $\mathtt{If}\, x\, \mathtt{Then} \cdot \mathtt{Else} \cdot$ encountered: this produces the monomial associated to this 1 node. Then $\phi_{\mathtt{m}}$ is the sum of all the monomials obtained this way and is clearly equivalent to ϕ. The procedure is in Logspace because we only need to remember which 1-leave we are treating and where we are in the tree (when going down) at any point. ◀

Putting all this together, we finally obtain a space-efficient decision procedure. Note however that it is totally sub-optimal in terms of time: to keep with the logarithmic space bound, we have to recompute a lot of things rather than store them.

▶ **Corollary 30** (BDD**equ** is in Logspace). *There is a logarithmic space algorithm that, given two* BDD *ϕ and ψ, decides wether they are equivalent.*

Proof. The BDD ϕ and ψ are equivalent if and only if $\phi \Leftrightarrow \psi = (\overline{\phi} + \psi).(\overline{\psi} + \phi) \sim 1$ that is to say (by passing to the negation) $(\overline{\psi}.\phi) + (\overline{\phi}.\psi) \sim 0$, which holds if and only if both $\overline{\psi}.\phi \sim 0$ and $\overline{\phi}.\psi \sim 0$.

But then, considering the first one (the other being similar) we can rewrite it in logarithmic space using the two above lemmas as $(\widehat{\psi})_{\mathtt{m}}.\phi_{\mathtt{m}} \sim 0$. This holds if and only if for all pairs (m, m') of one monomial in $(\widehat{\psi})_{\mathtt{m}}$ and one monomial in $\phi_{\mathtt{m}}$, m and m' are in conflict; which can be checked in logarithmic space using Remark 7. ◀

Let us now introduce an extremely simple, yet Logspace-complete problem [4], which will ease the Logspace-hardness part of our proof.

▶ **Definition 31** (order between vertices). *Order between vertices* (**ORD**) is the following decision problem:

"Given a directed graph $G = (V, E)$ that is a line [4] *and two vertices $f, s \in V$ do we have $f < s$ in the total order induced by G?"*

▶ **Lemma 32. ORD** *reduces to* °BDD**equ** *in* AC_0.

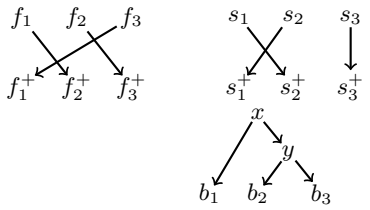
Proof. Again we are going to build a local graph transformation that is in AC_0.

First, we assume w.l.o.g. that the begin b and the exit e vertices of G are different from f and s. We write f^+ and s^+ the vertices immediately after f and s in G.

Then, we perform a first transformation by replacing the graph with three copies of itself (this can be done by locally scanning the graph and create labeled copies of the vertices

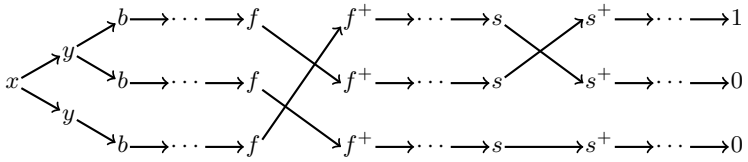
[4] We use the standard definition of graph as a pair (V, E) of sets of vertices and edges (oriented couples of vertices $x \to y$). A graph is a *line* if it is connected and all the vertices have in-degree and out-degree 1, except the *begin* vertex which has in-degree 0 and out-degree 1 and the *exit* vertex which has in-degree 1 and out-degree 0. A line induces a total order on vertices through its transitive closure.

and edges). We write x_i to refer to the copy of the vertex x in the graph i. The second transformation is a rewiring of the graph as follows: erase the edges going out of the f_i and s_i and replace them as pictured in the two first subgraphs:



Let us call G_r the rewired graph and G_n the non-rewired graph. To each of them we add two binary nodes x and y connected to the begin vertices b_i as pictured in the third graph above. Then we can produce two corresponding °BDD ϕ_r and ϕ_n by replacing the exit vertices e_1, e_2, e_3 by 1, 0, 0 respectively, x and y by a `If` x `Then` (`DontCare` y `Then` $\cdot$) `Else` (`If` y `Then` $\cdot$ `Else` $\cdot$) block of nodes; and any other v_i vertex by a `DontCare` v `Then` $\cdot$. It is then easy to see that if $f < s$ in the order induced by G if and only if ϕ_r and ϕ_n are equivalent.

Let us illustrate graphically what happens in the case where $f < s$: we draw the resulting °BDD as a labeled graph with the convention that a node labeled with z with out-degree 1 is a `DontCare` z `Then` $\cdot$ and a node labeled with z with out-degree 2 is a `If` z `Then` $\cdot$ `Else` $\cdot$ node with the upper edge corresponding to the `Then` branch and the lower edge corresponding to the `Else` branch.



◀

▶ Remark 33. The above construction relies on the fact that there are non-commuting permutations on the set of three elements: in a sense we are just attributing two non-commuting σ and τ to f and s and make sure that the order in which they intervene affects the equivalence class of the resulting °BDD. An approach quite similar in spirit with the idea of *permutation branching program* [1].

We can now extend our chain of reductions with the two new elements from this section

ORD (Logspace-hard) $\rightarrow$ °BDD**equ** $\rightarrow$ MALL$^-$**equ** $\rightarrow$ BDD**equ** ($\in$ Logspace)

so in the end we get our main result:

▶ **Theorem 34** (Logspace-completeness). *The decision problems* °BDD**equ**, MALL$^-$**equ** *and* BDD**equ** *are* Logspace*-complete under* AC_0 *reductions.*

4 Conclusion

In this work, we characterized precisely the complexity of proof equivalence in MALL$^-$ as Logspace-complete, contrasting greatly with the situation for the MLL fragment which has a Pspace-complete equivalence problem. We did so by establishing a correspondence between MALL$^-$ proofs and specific classes of BDD.

This path we took for proving our result is interesting in itself since the established correspondence allows a transfer of ideas in both directions. In particular, any progress in the theoretical problem of finding a correct notion of proofnet for MALL$^-$ would yield potential

applications to BDD, a notion of widespread practical use. An idea to explore might be the notion of *conflict net* defined by D. Hughes in an unpublished note [12]. Roughly speaking, the principle is to consider a presentation of proofnets with the information of the links that cannot be present at the same time, rather than giving an explicit formula to compute their presence, as it is the case with monomial proofnets or the BDD slicings we introduced.

On the other hand, since many optimization problems regarding BDD are known to be NP-complete, a finer view at the encoding of Section 3.1 in addition to basic constraints about what we expect from a notion of proofnet should lead to an impossibility result, even though the equivalence problem for MALL^- is only Logspace-complete.

Acknowledgements to people from `cstheory.stackexchange.com` for pointing the author to the notion of BDD; to Dominic Hughes for the live feedback during the redaction of the article; to Clément Aubert, for his help in understanding AC_0 reductions.

References

1 David A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. *Journal of Computer and System Sciences*, 38(1):150 – 164, 1989.

2 Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

3 Ashok K. Chandra, Larry J. Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM J. Comput.*, 13(2):423–439, 1984.

4 Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400 – 411, 1997.

5 Jean Gallier. On the correspondence between proofs and lambda-terms. In Philippe de Groote, editor, *The Curry-Howard isomorphism*, Cahiers du Centre de Logique, pages 55–138. Academia, 1995.

6 Jean-Yves Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1):1–101, 1987.

7 Jean-Yves Girard. Proof-nets: The parallel syntax for proof-theory. *Logic and Algebra*, 180:97–124, May 1996.

8 Stefano Guerrini and Andrea Masini. Parsing MELL Proof Nets. *Theoretical Computer Science*, 254(1-2):317–335, 2001.

9 Willem Heijltjes and Robin Houston. No Proof Nets for MLL with Units: Proof Equivalence in MLL is PSPACE-complete. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 50:1–50:10, New York, NY, USA, 2014. ACM.

10 Willem Heijltjes and Lutz Straßburger. Proof nets and semi-star-autonomous categories. *Mathematical Structures in Computer Science*, FirstView:1–40, 11 2014.

11 Dominic J. D. Hughes. Simple multiplicative proof nets with units. Technical report, 2005.

12 Dominic J. D. Hughes. Abstract p-time proof nets for MALL: Conflict nets. *arXiv: math.LO/0801.2421v1*, 2008.

13 Dominic J. D. Hughes and Rob J. van Glabbeek. Proof nets for unit-free multiplicative-additive linear logic. *ACM Trans. Comput. Log.*, 6(4):784–842, 2005.

14 Dominic J. D. Hughes and Rob J. van Glabbeek. MALL proof nets identify proofs modulo rule commutation. `http://boole.stanford.edu/~dominic/MALL-equiv.pdf`, *to appear*, 2015.

15 Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 12th edition, 2009.

16 Olivier Laurent and Roberto Maieli. Cut elimination for monomial MALL proof nets. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 486–497, 2008.

Mixin Composition Synthesis Based on Intersection Types*

Jan Bessai[1], Andrej Dudenhefner[1], Boris Düdder[1], Tzu-Chun Chen[2], Ugo de'Liguoro[3], and Jakob Rehof[1]

1 **Technical University of Dortmund, Dortmund, Germany**
{jan.bessai, boris.duedder, andrej.dudenhefner, jakob.rehof}@cs.tu-dortmund.de
2 **Technical University of Darmstadt, Darmstadt, Germany**
tcchen@rbg.informatik.tu-darmstadt.de
3 **University of Torino, Torino, Italy**
ugo.deliguoro@unito.it

Abstract

We present a method for synthesizing compositions of mixins using type inhabitation in intersection types. First, recursively defined classes and mixins, which are functions over classes, are expressed as terms in a lambda calculus with records. Intersection types with records and record-merge are used to assign meaningful types to these terms without resorting to recursive types. Second, typed terms are translated to a repository of typed combinators. We show a relation between record types with record-merge and intersection types with constructors. This relation is used to prove soundness and partial completeness of the translation with respect to mixin composition synthesis. Furthermore, we demonstrate how a translated repository and goal type can be used as input to an existing framework for composition synthesis in bounded combinatory logic via type inhabitation. The computed result corresponds to a mixin composition typed by the goal type.

1998 ACM Subject Classification F.4.1 Mathematical Logic – λ Calculus and Related Systems

Keywords and phrases Record Calculus, Combinatory Logic, Type Inhabitation, Mixin, Intersection Type

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.76

1 Introduction

Starting with Cardelli's pioneering work [13], various typed λ-calculi extended with records have been thoroughly studied to model sophisticated features of object-oriented programming languages, like recursive objects and classes, object extension, method overriding and inheritance (see e.g. [1, 11, 23]).

Here, we focus on the synthesis of mixin compositions. In the object-oriented paradigm, mixins [9, 10] have been introduced as an alternative construct for code reuse that improves over the limitations of multiple inheritance, e.g. connecting incompatible base classes and semantic ambiguities caused by the diamond problem. Together with abstract classes and traits, mixins (functions over classes) can be considered as an advanced construct to obtain flexible implementations of module libraries and to enhance code reusability; many popular

* This work was partially supported by EU COST Action IC1201: BETTY and MIUR PRIN CINA Prot. 2010LHT4KM, San Paolo Project SALT.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 76–91

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programming languages miss native support for mixins, but they are an object of intensive study and research (e.g. [8, 24]). In this setting we aim at synthesizing classes from a library of mixins that can be used in programming languages like Java, which do not natively support mixins. Our particular modeling approach is inspired by modern language features (e.g. ECMAScript "`bind`") to preserve contexts in order to prevent programming errors [21].

We formalize synthesis of classes from a library of mixins as an instance of the relativized type inhabitation problem in bounded combinatory logic with intersection types [20]. Relativized type inhabitation is the decision problem: given a combinatory type context Δ and a type τ does there exist an applicative term e such that e has type τ under the type assumptions in Δ? We denote type inhabitation by $\Delta \vdash_{\mathrm{BCL}} ? : \tau$ and implicitly include the problem of constructing a term inhabiting τ.

Relativized type inhabitation, which is undecidable in general [20], is decidable in k-bounded combinatory logic $\mathsf{BCL}_k(\to,\cap)$, that is, combinatory logic typed with arrow and intersection types of depth at most k, and hence an algorithm for semi-deciding type inhabitation for $\mathsf{BCL}(\to,\cap) = \bigcup_k \mathsf{BCL}_k(\to,\cap)$ can be obtained by iterative deepening over k and solving the corresponding decision problem in $\mathsf{BCL}_k(\to,\cap)$ [20]. In the present paper, we enable combinatory synthesis of classes via intersection typed mixin combinators. Intersection types [4] play an important rôle in combinatory synthesis, because they allow for semantic specification of components and synthesis goals [20, 5].

Now, looking at $\{C_1 : \sigma_1, \ldots, C_p : \sigma_p, M_1 : \tau_1, \ldots, M_q : \tau_q\} \subseteq \Delta$ as the abstract specification of a library including classes C_i and mixins M_j with interfaces σ_i and τ_j respectively, and given a type τ specifying an unknown class, we may identify the class synthesis problem with the type inhabitation problem $\Delta \vdash_{\mathrm{BCL}} ? : \tau$. To make this feasible, we have to bridge the gap between the expressivity of highly sophisticated type systems used for typing classes and mixins, for instance F-bounded polymorphism used in [12, 15], and the system of intersection types from [4]. In doing so, we move from the system originally presented in [16], consisting of a type assignment system of intersection and record types to a λ-calculus which we enrich here with record merge operation (called "**with**" in [15]), to allow for expressive mixin combinators. The type system is modified by reconstructing record types $\langle l_i : \sigma_i \mid i \in I\rangle$ as intersection of unary record types $\langle l_i : \sigma_i\rangle$, and considering a subtype relation extending the one in [4]. This is however not enough for typing record merge, for which we consider a type-merge operator +. The problem of typing extensible records and merge, faced for the first time in [27, 26], is notoriously hard; to circumvent difficulties the theory of record subtyping in [15] (where a similar type-merge operator is considered) allows just for "exact" record typing, which involves subtyping in depth, but not in width. Such a restriction, that has limited effects w.r.t. a rich and expressive type system like F-bounded polymorphism, would be too severe in our setting. Therefore, we undertake a study of the type algebra of record types with intersection and type-merge, leading to a type assignment system where exact record typing is required only for the right-hand side operand of the term merge operator, which is enough to ensure soundness of typing.

The next challenge is to show that we can type in a meaningful way in our system classes and mixins, where the former are essentially recursive records and the latter are made of a combination of fixed point combinators and record merge. Such combinators, which usually require recursive types, can be typed in our system by means of an iterative method exploiting the ability of intersection types to represent approximations of the potentially infinite unfolding of recursive definitions.

The final problem we face is the encoding of intersection types with record types and type-merge into the language of $\mathsf{BCL}(\to,\cap)$. For this purpose we consider a conservative extension

of bounded combinatory logic, called $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$, where we allow unary type constructors that are monotonic and distribute over intersection. We show that the (semi) algorithm solving inhabitation for $\mathsf{BCL}(\to,\cap)$ can be adapted to $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$, by proving that the key properties necessary to solve the inhabitation problem in $\mathsf{BCL}(\to,\cap)$ are preserved in $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$ and showing how the type-merge operator can be simulated in $\mathsf{BCL}(\mathbb{T}_\mathbb{C})$. In fact, type-merge is not monotonic in its second argument, due to the lack of negative information caused by the combination of + and $\cap$. Our work culminates in two theorems that ensure soundness and completeness of the so obtained method w.r.t. synthesis of classes by mixins composition.

Related works. This work evolves from the contributions [5, 17, 6] to the workshop ITRS'14. The papers that have inspired our work, mainly by Cook and others, have been cited above. The theme of using intersection types and bounded-polymorphism for typing object-oriented languages and inheritance has been treated in [14, 25]. Type inhabitation has been recently used for synthesis of object oriented code [22, 18], but to our best knowledge the present paper provides, for the first time, a theory of type-safe mixin composition synthesis based on the component-oriented approach of combinatory logic synthesis.

2 Intersection Types for Mixins and Classes

2.1 Intersection and record types

We consider a type-free λ-calculus of extensible records, equipped with a merge operator. The term syntax is defined by the following grammar:

$$\begin{array}{rrll} \Lambda_R \ni M,N,M_i & ::= & x \mid (\lambda x.M) \mid (MN) \mid (M.l) \mid R \mid (M \oplus R) & \text{terms} \\ R & ::= & \langle l_i = M_i \mid i \in I\rangle & \text{records} \end{array}$$

where $x \in \mathbf{Var}$ and $l \in \mathbf{Label}$ range over denumerably many *term variables* and *labels* respectively, and the sets of indexes I are finite. Free and bound variables are defined as usual for ordinary λ-calculus, and we name Λ^0_R the set of all closed terms in Λ_R; terms are identified up to renaming of bound variables and $M\{N/x\}$ denotes capture avoiding substitution of N for x in M. We adopt notational conventions from [3]; in particular application associates to the left and external parentheses are omitted when unnecessary; also the dot notation for record selection takes precedence over λ, so that $\lambda x.\, M.l$ reads as $\lambda x.(M.l)$. If not stated otherwise $\oplus$ also associates to the left, and we avoid external parentheses when unnecessary.

Terms $R \equiv \langle l_i = M_i \mid i \in I\rangle$ (writing $\equiv$ for syntactic identity) represent *records*, with fields l_i and M_i as the respective values; we set $lbl(\langle l_i = M_i \mid i \in I\rangle) = \{l_i \mid i \in I\}$. The term $M.l$ is *field selection* and $M \oplus R$ is record *merge*. In particular if R_1 and R_2 are records then $R_1 \oplus R_2$ is the record with as fields the union of the fields of R_1 and R_2 and as values those of the original records but in case of ambiguity, where the values in R_2 prevail. The syntactic constraint that R is a record in $M \oplus R$ is justified after Definition 8.

The actual meaning of these operations is formalized by the following reduction relation:

▶ **Definition 1** (Λ_R reduction)**.** Reduction $\longrightarrow \subseteq \Lambda^2_R$ is the least compatible relation such that:

$$\begin{array}{lrcll} (\beta) & (\lambda x.M)N & \longrightarrow & M\{N/x\} & \\ (sel) & \langle l_i = M_i \mid i \in I\rangle.l_j & \longrightarrow & M_j & \text{if } j \in I \\ (\oplus) & \langle l_i = M_i \mid i \in I\rangle \oplus \langle l_j = N_j \mid\ j \in J\rangle & \longrightarrow & \langle l_i = M_i,\ l_j = N_j \mid i \in I \setminus J,\ j \in J\rangle & \end{array}$$

We claim that $\longrightarrow^*$ is Church-Rosser, and that $(M \oplus R_1) \oplus R_2$ is equivalent to $M \oplus (R_1 \oplus R_2)$ under any reasonable observational semantics (e.g. by extending to Λ_R applicative bisimulation from the lazy λ-calculus).

Record merge subsumes field update: $M.l := N \equiv M \oplus \langle l = N \rangle$, but merge is not uniformly definable in terms of update as long as labels are not expressions in the calculus.

In the spirit of Curry's assignment of polymorphic types and of intersection types in particular, types are introduced as a syntactical tool to capture semantic properties of terms, rather than as constraints to term formation.

▶ **Definition 2** (Intersection types for Λ_R).

$$\begin{array}{rcll} \mathbb{T} \ni \sigma, \sigma_i & ::= & a \mid \omega \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \cap \sigma_2 \mid \rho & \text{types} \\ \mathbb{T}_{\langle\rangle} \ni \rho, \rho_i & ::= & \langle\rangle \mid \langle l : \sigma \rangle \mid \rho_1 + \rho_2 \mid \rho_1 \cap \rho_2 & \text{record types} \end{array}$$

where a ranges over *type constants*, $l \in \mathbf{Label}$.

We use σ, τ, possibly with sub and superscripts, for types in $\mathbb{T}$ and ρ, ρ_i, possibly with superscripts, for record types in $\mathbb{T}_{\langle\rangle}$ only. Note that $\to$ associates to the right, and $\cap$ binds stronger than $\to$. As with intersection type systems for the λ-calculus, the intended meaning of types are sets, provided a set theoretic interpretation of constants a.

Following [4], type semantics is given axiomatically by means of the subtyping relation $\leq$, that can be interpreted as subset inclusion. It is the least pre-order over $\mathbb{T}$ such that:

▶ **Definition 3** (Type inclusion: arrow and intersection types).

$$\begin{array}{lll} \sigma \leq \omega, & \omega \leq \omega \to \omega, & \\ \sigma \cap \tau \leq \sigma, & \sigma \cap \tau \leq \tau, & \sigma \leq \tau_1 \ \& \ \sigma \leq \tau_2 \Rightarrow \sigma \leq \tau_1 \cap \tau_2, \\ (\sigma \to \tau_1) \cap (\sigma \to \tau_2) \leq \sigma \to \tau_1 \cap \tau_2 & & \sigma_2 \leq \sigma_1 \ \& \ \tau_1 \leq \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2 \end{array}$$

We write $\sigma = \tau$ for $\sigma \leq \tau$ and $\tau \leq \sigma$.

▶ **Definition 4** (Type inclusion: record types).

$$\begin{array}{lll} \langle l : \sigma \rangle \cap \langle l : \tau \rangle \leq \langle l : \sigma \cap \tau \rangle, & \sigma \leq \tau \Rightarrow \langle l : \sigma \rangle \leq \langle l : \tau \rangle, & \\ \langle l : \sigma \rangle \leq \langle\rangle, & \langle l : \sigma \rangle + \langle\rangle = \langle l : \sigma \rangle = \langle\rangle + \langle l : \sigma \rangle, & \\ \langle l : \sigma \rangle + \langle l : \tau \rangle = \langle l : \tau \rangle, & \langle l : \sigma \rangle + \langle l' : \tau \rangle = \langle l : \sigma \rangle \cap \langle l' : \tau \rangle & (l \neq l'), \\ \langle l : \sigma \rangle + (\langle l : \tau \rangle \cap \rho) = \langle l : \tau \rangle \cap \rho, & \langle l : \sigma \rangle + (\langle l' : \tau \rangle \cap \rho) = \langle l' : \tau \rangle \cap (\langle l : \sigma \rangle + \rho) & (l \neq l'). \end{array}$$

While Definition 3 is standard after [4], comments on Definition 4 are in order. Type $\langle\rangle$ is the type of all records. Type $\langle l : \sigma \rangle$ is a unary record type, whose meaning is the set of records having at least a field labeled by l, with value of type σ; therefore $\langle l : \sigma \rangle \cap \langle l : \tau \rangle$ is the type of records having label l with values both of type σ and τ, that is of type $\sigma \cap \tau$. In fact the equation $\langle l : \sigma \rangle \cap \langle l : \tau \rangle = \langle l : \sigma \cap \tau \rangle$ is derivable. On the other hand $\langle l : \sigma \rangle \cap \langle l' : \tau \rangle$, with $l \neq l'$, is the type of records having fields labeled by l and l', with values of type σ and τ respectively. It follows that intersection of record types can be used to express properties of records with arbitrary (though finitely) many fields, which justifies the abbreviation $\langle l_i : \sigma_i \mid i \in I \neq \varnothing \rangle = \bigcap_{i \in I} \langle l_i : \sigma_i \rangle$ and $\langle l_i : \sigma_i \mid i \in \varnothing \rangle = \langle\rangle$, where we assume that the l_i are pairwise distinct. Finally, as it will be apparent from Definition 8 below, $\rho_1 + \rho_2$ is the type of all records obtained by merging a record of type ρ_1 with a record of type ρ_2, which is intended to type $\oplus$ that is at the same time a record extension and field updating operation. Since this is the distinctive feature of the system introduced here, we comment on this by means of a few lemmas, illustrating its properties.

▶ **Lemma 5.**
1. $(\forall j \in J \subseteq I.\ \sigma_j \leq \tau_j) \Rightarrow \langle l_i : \sigma_i \mid i \in I\rangle \leq \langle l_j : \tau_j \mid j \in J\rangle$,
2. $\langle l_i : \sigma_i \mid i \in I\rangle + \langle l_j : \tau_j \mid j \in J\rangle = \langle l_i : \sigma_i, m_j : \tau_j \mid i \in I \setminus J, j \in J\rangle$,
3. $\forall \rho \in \mathbb{T}_{\langle\rangle}.\ \exists\ \langle l_i : \sigma_i \mid i \in I\rangle.\ \rho = \langle l_i : \sigma_i \mid i \in I\rangle$.

Part (1) of Lemma 5 states that subtyping among intersection of unary record types subsumes subtyping in width and depth of ordinary record types from the literature. Part (2) shows that the + type constructor reflects at the level of types the operational behavior of the merge operator $\oplus$. Part (3) says that any record type is equivalent to an intersection of unary record types; this implies that types of the form $\rho_1 + \rho_2$ are eliminable in principle. However they play a key role in typing mixins, motivating the issue of control of negative information in the synthesis process: see sections 2.2 and 4. More properties of subtyping record types w.r.t. + and $\cap$ are listed in the next lemma. Let us preliminary define the map $lbl : \mathbb{T}_{\langle\rangle} \to \wp(\mathbf{Label})$ (where $\wp(\mathbf{Label})$ is the powerset of **Label**) by:

$$lbl(\langle l : \sigma\rangle) = \{l\}, \quad lbl(\rho_1 \cap \rho_2) = lbl(\rho_1 + \rho_2) = lbl(\rho_1) \cup lbl(\rho_2).$$

Then we immediately have:

▶ **Lemma 6.**
1. $\rho_1 = \rho_2 \Rightarrow lbl(\rho_1) = lbl(\rho_2)$,
2. $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing \Rightarrow \rho_1 + \rho_2 = \rho_1 \cap \rho_2$.

In (1) above $\rho_1 = \rho_2$ is $\rho_1 \leq \rho_2 \leq \rho_1$. About (2) note that condition $lbl(\rho_1) \cap lbl(\rho_2) = \varnothing$ is essential, since $\rho_1 + \rho_2 \neq \rho_2 + \rho_1$ in general, as it immediately follows by Lemma 5.2.

▶ **Lemma 7.**
1. $(\rho_1 + \rho_2) + \rho_3 = \rho_1 + (\rho_2 + \rho_3)$,
2. $(\rho_1 \cap \rho_2) + \rho_3 = (\rho_1 + \rho_3) \cap (\rho_2 + \rho_3)$,
3. $\rho_1 \leq \rho_2 \Rightarrow \rho_1 + \rho_3 \leq \rho_2 + \rho_3$,
4. $\rho_1 + \rho_2 = \rho_1 \cap \rho_2 \Leftrightarrow \rho_1 + \rho_2 \leq \rho_1$.

▶ Remark. In general $\rho_1 + (\rho_2 \cap \rho_3) \neq (\rho_1 + \rho_3) \cap (\rho_2 + \rho_3)$: take $\rho_1 \equiv \langle l_1 : \sigma_1, l_2 : \sigma_2\rangle$, $\rho_2 \equiv \langle l_1 : \sigma_1'\rangle$ and $\rho_3 \equiv \langle l_2 : \sigma_2'\rangle$, with $\sigma_1 \neq \sigma_1'$ and $\sigma_2 \neq \sigma_2'$. Then we have: $\rho_1 + (\rho_2 \cap \rho_3) = \rho_1 + \langle l_1 : \sigma_1', l_2 : \sigma_2'\rangle = \langle l_1 : \sigma_1', l_2 : \sigma_2'\rangle$, while $(\rho_1 + \rho_3) \cap (\rho_2 + \rho_3) = \langle l_1 : \sigma_1, l_2 : \sigma_2'\rangle \cap \langle l_1 : \sigma_1', l_2 : \sigma_2\rangle = \langle l_1 : \sigma_1 \cap \sigma_1', l_2 : \sigma_2 \cap \sigma_2'\rangle$. The last example suggests that $(\rho_1 + \rho_3) \cap (\rho_2 + \rho_3) \leq \rho_1 + (\rho_2 \cap \rho_3)$. On the other hand $\rho_2 \leq \rho_3 \not\Rightarrow \rho_1 + \rho_2 \leq \rho_1 + \rho_3$. Indeed:

$$\begin{array}{lcll} \langle l_0 : \sigma_1, l_1 : \sigma_2\rangle + \langle l_1 : \sigma_3, l_2 : \sigma_4\rangle & = & \langle l_0 : \sigma_1, l_1 : \sigma_1', l_2 : \sigma_2\rangle & \\ & \not\leq & \langle l_0 : \sigma_0, l_1 : \sigma_1, l_2 : \sigma_2\rangle & \text{if } \sigma_1' \not\leq \sigma_1 \\ & = & \langle l_0 : \sigma_1, l_1 : \sigma_2\rangle + \langle l_2 : \sigma_4\rangle & \end{array}$$

even if $\langle l_1 : \sigma_1, l_2 : \sigma_2\rangle \leq \langle l_2 : \sigma_2\rangle$. From this and (3) of Lemma 7, we conclude that + is monotonic in its first argument, but not in its second one.

We come now to the type assignment system. A *basis* (also called a context in the literature) is a finite set $\Gamma = \{x_1 : \sigma_n, \ldots, x_n : \sigma_n\}$, where the variables x_i are pairwise distinct; we set $dom(\Gamma) = \{x \mid \exists \sigma.\ x : \sigma \in \Gamma\}$ and we write $\Gamma, x : \sigma$ for $\Gamma \cup \{x : \sigma\}$ where $x \notin dom(\Gamma)$. Then we consider the following extension of the system in [4], also called **BCD** in the literature.

▶ **Definition 8** (Type Assignment). The rules of the assignment system are:

$$\frac{x:\sigma\in\Gamma}{\Gamma\vdash x:\sigma}\,(Ax) \qquad \frac{\Gamma,x:\sigma\vdash M:\tau}{\Gamma\vdash\lambda x.M:\sigma\to\tau}\,(\to I)$$

$$\frac{\Gamma\vdash M:\sigma\to\tau \qquad \Gamma\vdash N:\sigma}{\Gamma\vdash MN:\tau}\,(\to E) \qquad \frac{\Gamma\vdash M:\sigma \quad \Gamma\vdash M:\tau}{\Gamma\vdash M:\sigma\cap\tau}\,(\cap)$$

$$\frac{}{\Gamma\vdash M:\omega}\,(\omega) \qquad \frac{\Gamma\vdash M:\sigma \qquad \sigma\le\tau}{\Gamma\vdash M:\tau}\,(\le)$$

$$\frac{}{\Gamma\vdash\langle l_i=M_i\mid i\in I\rangle:\langle\rangle}\,(\langle\rangle) \qquad \frac{\Gamma\vdash M_k:\sigma \quad k\in I}{\Gamma\vdash\langle l_i=M_i\mid i\in I\rangle:\langle l_k:\sigma\rangle}\,(rec)$$

$$\frac{\Gamma\vdash M:\langle l:\sigma\rangle}{\Gamma\vdash M.l:\sigma}\,(sel) \qquad \frac{\Gamma\vdash M:\rho_1 \qquad \Gamma\vdash R:\rho_2 \quad (*)}{\Gamma\vdash M\oplus R:\rho_1+\rho_2}\,(+)$$

where $(*)$ in rule $(+)$ is the side condition: $lbl(R)=lbl(\rho_2)$.

Using Lemma 5.1, the following rule is easily shown to be admissible:

$$\frac{\Gamma\vdash M_j:\sigma_j \qquad \forall j\in J\subseteq I}{\Gamma\vdash\langle l_i=M_i\mid i\in I\rangle:\langle l_j:\sigma_i\mid j\in J\rangle}\,(rec')$$

Contrary to this, the side condition $(*)$ of rule $(+)$ is equivalent to "exact" record typing in [10], disallowing record subtyping in width. Such a condition is necessary for soundness of typing. Indeed suppose that $\Gamma\vdash M_0:\sigma$ and $\Gamma\vdash M_0':\sigma_0'$ but $\Gamma\not\vdash M_0':\sigma_0$; then without $(*)$ we could derive:

$$\frac{\Gamma\vdash\langle l_0=M_0\rangle:\langle l_0:\sigma_0\rangle \qquad \Gamma\vdash\langle l_0=M_0',l_1:\sigma_1\rangle:\langle l_1:\sigma_1\rangle}{\Gamma\vdash\langle l_0=M_0\rangle\oplus\langle l_0=M_0',l:_1:\sigma_1\rangle:\langle l_0:\sigma_0,l_1:\sigma_1\rangle}$$

from which we obtain that $\Gamma\vdash(\langle l_0=M_0\rangle\oplus\langle l_0=M_0',l:_1:\sigma_1\rangle).l_0:\sigma_0$ breaking subject reduction, since $(\langle l_0=M_0\rangle\oplus\langle l_0=M_0',l:_1:\sigma_1\rangle).l_0\longrightarrow^*M_0'$. The essential point is that proving that $\Gamma\vdash N:\langle l:\sigma\rangle$ doesn't imply that $l'\notin lbl(R')$ for any $l'\neq l$, which follows only by the uncomputable (not even r.e.) statement that $\Gamma\not\vdash N:\langle l':\omega\rangle$, a negative information.

This explains the restriction to record terms as the second argument of $\oplus$: in fact allowing $M\oplus N$ to be well formed for an arbitrary N we might have $N\equiv x$ in $\lambda x.(M\oplus x)$. But extending lbl to all terms in Λ_R is not possible without severely limiting the expressiveness of the assignment system. In fact to say that $lbl(N)=lbl(R)$ if $N\longrightarrow^*R$ would make the lbl function non computable; on the other hand putting $lbl(x)=\varnothing$, which is the only reasonable and conservative choice as we do not know of possible substitutions for x in $\lambda x.(M\oplus x)$, implies that the latter term has type $\omega\to\omega=\omega$ at best.

As a final remark, let us observe that we do not adopt exact typing of records in general, but only for typing the right-hand side of $\oplus$-terms, a feature that will be essential when typing mixins.

▶ **Lemma 9.** *Let $\sigma\neq\omega$:*

1. $\Gamma\vdash x:\sigma\iff\exists\tau.\,x:\tau\in\Gamma\ \&\ \tau\le\sigma$,
2. $\Gamma\vdash\lambda x.M:\sigma\iff\exists I,\sigma_i,\tau_i.\ \Gamma,x:\sigma_i\vdash M:\tau_i\ \&\ \bigcap_{i\in I}\sigma_i\to\tau_i\le\sigma$,
3. $\Gamma\vdash MN:\sigma\iff\exists\tau.\,\Gamma\vdash M:\tau\to\sigma\ \&\ \Gamma\vdash N:\tau$,
4. $\Gamma\vdash\langle l_i=M_i\mid i\in I\rangle:\sigma\iff\forall i\in I\ \exists\sigma_i.\,\Gamma\vdash M_i:\sigma_i\ \&\ \langle l_i:\sigma_i\mid i\in I\rangle\le\sigma$,
5. $\Gamma\vdash M.l:\sigma\iff\Gamma\vdash M:\langle l:\sigma\rangle$,
6. $\Gamma\vdash M\oplus R:\sigma\iff\exists\rho_1,\rho_2.\,\Gamma\vdash M:\rho_1\ \&\ \Gamma\vdash R:\rho_2\ \&\ lbl(R)=lbl(\rho_2)\ \&\ \rho_1+\rho_2\le\sigma$.

▶ **Theorem 10** (Subject reduction). $\Gamma \vdash M : \sigma \;\&\; M \longrightarrow N \Rightarrow \Gamma \vdash N : \sigma$.

Proof Sketch. The proof is by cases of reduction rules, using Lemma 9. The only relevant case is when $M \equiv R_1 \oplus R_2$, with $R_1 \equiv \langle l_i = M_i \mid i \in I\rangle$ and $R_2 \equiv \langle l_j = N_j \mid \; j \in J\rangle$, and $N \equiv \langle l_i = M_i,\; l_j = N_j \mid i \in I \setminus J,\; j \in J\rangle$. By Lemma 9.6 we may suppose w.l.o.g. that $\sigma = \rho_1 + \rho_2$, and that $\rho_1 = \langle l_i : \sigma_i \mid i \in I'\rangle$ for some $I' \subseteq I$, and $\rho_2 = \langle l_j : \tau_j \mid j \in J'\rangle$ for some $J' \subseteq J$; but also we know that $J' = J$, because of condition $(*)$.

Now by Lemma 5.2, $\rho_1 + \rho_2 = \langle l_i : \sigma_i, l_j : \tau_j \mid i \in I' \setminus J, j \in J\rangle$; on the other hand by Lemma 9.4, we know that $\Gamma \vdash M_i : \sigma_i$ for all $i \in I'$, $\Gamma \vdash N_j : \tau_j$ for all $j \in J$, and therefore we conclude that $\Gamma \vdash N : \langle l_i : \sigma_i, l_j : \tau_j \mid i \in I' \setminus J, j \in J\rangle$ by multiple applications of rules (rec) and $(\cap)$. ◀

2.2 Class and Mixin combinators

The following definition of classes and mixins is inspired by [15] and [10] respectively, though with some departures to be discussed below. To make the description more concrete, in the examples we add constants to Λ_R.

Recall that a *combinator* is a term in Λ_R^0, namely a closed term. Let $\mathbf{Y}$ be Curry's fixed point combinator: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ (the actual definition of $\mathbf{Y}$ is immaterial, since all fixed point combinators have the same types in $\mathbf{BCD}$).

▶ **Definition 11.** Let $\mathsf{myClass}, \mathsf{state}$ and $\mathsf{argClass}$ be (term) variables and $\mathbf{Y}$ be a fixed point combinator; then we define the following sets of combinators:

$$\begin{array}{lrcl} \text{Class:} & C & ::= & \mathbf{Y}(\lambda\,\mathsf{myClass}\;\lambda\,\mathsf{state}.\,\langle l_i = N_i \mid i \in I\rangle) \\ \text{Mixin:} & M & ::= & \lambda\,\mathsf{argClass}.\,\mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\,(\mathsf{argClass}\;\mathsf{state}) \oplus \langle l_i = N_i \mid i \in I\rangle) \end{array}$$

We define $\mathcal{C}$ and $\mathcal{M}$ as the sets of classes and mixins respectively.

To illustrate this definition let us use the abbreviation $\texttt{let}\;\; x = N \;\;\texttt{in}\;\; M \equiv M\{N/x\}$. Then a class combinator $C \in \mathcal{C}$ can be written in a more perspicuous way as follows:

$$C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\;\lambda\,\mathsf{state}.\;\texttt{let}\;\;\mathsf{self} = (\mathsf{myClass}\;\mathsf{state})\;\;\texttt{in}\;\;\langle l_i = N_i \mid i \in I\rangle). \tag{1}$$

A *class* is the fixed point of a function, the *class definition*, mapping a recursive definition of the class itself and a state S, that is the value or a record of values in general, for the instance variables of the class, into a record $\langle l_i = N_i \mid i \in I\rangle$ of *methods*. A class C is instantiated to an *object* $O \equiv C\,S$ by applying the class C to a state S. Hence we have:

$$O \equiv C\,S \longrightarrow^* \texttt{let}\;\;\mathsf{self} = (C\,S)\;\;\texttt{in}\;\;\langle l_i = N_i \mid i \in I\rangle,$$

where the variable self is used in the method bodies N_i to call other methods from the same object. Note that the recursive parameter $\mathsf{myClass}$ might occur in the N_i in subterms other than $(\mathsf{myClass}\;\mathsf{state})$, and in particular $N_i\{C/\mathsf{myClass}\}$ might contain a subterm $C\,S'$, where S' is a state possibly different than S; even C itself might be returned as the value of a method. Classes are the same as in [15] §4, but for the explicit identification of self with $(\mathsf{myClass}\;\mathsf{state})$.

We come now to typing of classes. Let $R = \langle l_i = N_i \mid i \in I\rangle$, and suppose that $C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\;\lambda\,\mathsf{state}.\,R) \in \mathcal{C}$. To type C we must find a type σ (a type of its state) and a sequence of types $\rho_1, \ldots, \rho_n \in \mathbb{T}_{\langle\rangle}$ such that for all $i < n$:

$$\mathsf{myClass} : \sigma \to \rho_i, \mathsf{state} : \sigma \vdash R : \rho_{i+1}.$$

Note that this is always possible for any n: in the worst case, we can take $\rho_i = \langle l_i : \omega \mid i \in I \rangle$ for all $0 < i \leq n$. In general one has more expressive types, depending on the typings of the N_i in R (see example 12 below). It follows that:

$$\vdash \lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\, R : (\omega \to \rho_1) \cap \bigcap_{1 \leq i < n} (\sigma \to \rho_i) \to (\sigma \to \rho_{i+1}),$$

and therefore, by using the fact that $\vdash \mathbf{Y} : (\omega \to \tau_1) \cap \cdots \cap (\tau_{n-1} \to \tau_n) \to \tau_n$ for arbitrary types $\tau_1, \ldots, \tau_n$, we conclude that the typing of classes has the following shape (where $\rho = \rho_n$):

$$\vdash C \equiv \mathbf{Y}(\lambda\,\mathsf{myClass}\ \lambda\,\mathsf{state}.\, \langle l_i = N_i \mid i \in I \rangle) : \sigma \to \rho \tag{2}$$

In conclusion the type of a class C is the arrow from the type of the state σ to a type ρ of its instances.

▶ **Example 12.** The class Point has an integer state and contains the method *get* to retrieve the state, *set* to update the current state and *shift* to add a value to the current state.

$$\begin{aligned}\mathsf{Point} = \mathbf{Y}(&\lambda\mathsf{myClass}.\lambda\mathsf{state}.\texttt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \texttt{in}\\ &\langle \mathit{get} = \mathsf{state}, \mathit{set} = \lambda\mathsf{state}'.\mathsf{state}', \mathit{shift} = \lambda d.\mathsf{self}.\mathit{set}(\mathsf{self}.\mathit{get} + d)\rangle)\end{aligned}$$

Note that in a setting without references, we rely on purely functional state updates. Therefore, every function returns a new state that can be used to construct the new object. An example for this is set, which just returns the new state.

To type Point we have $\vdash \mathsf{Point} : \mathrm{Int} \to \rho_{\mathrm{Point}}$ where $\rho_{\mathrm{Point}} = \rho_2$ and

$$\begin{aligned}\rho_1 &= \langle \mathit{get} : \mathrm{Int}, \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \omega \rangle\\ \rho_2 &= \langle \mathit{get} : \mathrm{Int}, \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int} \rangle\end{aligned}$$

using $\mathbf{Y} : (\omega \to \mathrm{Int} \to \rho_1) \cap ((\mathrm{Int} \to \rho_1) \to \mathrm{Int} \to \rho_2) \to \mathrm{Int} \to \rho_2$

A *mixin* $M \in \mathcal{M}$ is a combinator such that, if $C \in \mathcal{C}$ then $M\,C$ reduces to a new class $C' \in \mathcal{C}$, inheriting from C. Writing M in a more explicit way we obtain:

$$\begin{aligned}M \equiv \lambda\,\mathsf{argClass}.\, \mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\, &\texttt{let}\ \mathsf{super} = (\mathsf{argClass}\ \mathsf{state})\ \texttt{in}\\ &\texttt{let}\ \mathsf{self} = (\mathsf{myClass}\ \mathsf{state})\ \texttt{in}\\ &\mathsf{super} \oplus \langle l_i = N_i \mid i \in I \rangle)\end{aligned}$$

In words, a mixin merges an instance $C\,S$ of the input class C with a new state S together with a *difference* record $R \equiv \langle l_i = N_i \mid i \in I \rangle$, that would be written $\Delta(C\,S)$ in terms of [10]. Note that our mixins are not the same as class modificators (also called wrappers e.g. in [9]) because the latter do not take the instantiation of a class as the value of super, but the class definition, namely the function defining the class *before* taking its fixed point.

Let $M \equiv \lambda\,\mathsf{argClass}.\, \mathbf{Y}(\lambda\,\mathsf{myClass}\,\lambda\,\mathsf{state}.\, (\mathsf{argClass}\ \mathsf{state}) \oplus R) \in \mathcal{M}$; to type M we have to find types $\sigma^1, \sigma^2, \rho^1$ and a sequence $\rho^2_1, \ldots, \rho^2_n \in \mathbb{T}_{\langle\rangle}$ of record types such that for all $1 \leq i < n$ it is true that $\mathit{lbl}(R) = \mathit{lbl}(\rho^2_i)$ and such that, setting $\Gamma_0 = \{\mathsf{argClass} : \sigma^1 \to \rho^1, \mathsf{myClass} : \omega, \mathsf{state} : \sigma^1 \cap \sigma^2\}$ and $\Gamma_i = \{\mathsf{argClass} : \sigma^1 \to \rho^1, \mathsf{myClass} : (\sigma^1 \cap \sigma^2) \to \rho^1 + \rho^2_i, \mathsf{state} : \sigma^1 \cap \sigma^2\}$ for all $1 \leq i < n$, we may deduce for all $0 \leq i < n$:

$$\dfrac{\dfrac{\dfrac{\Gamma_i \vdash \mathsf{state} : \sigma^1 \cap \sigma^2}{\Gamma_i \vdash \mathsf{state} : \sigma^1}\,(\leq)}{\Gamma_i \vdash \mathsf{argClass}\ \mathsf{state} : \rho^1}\,(\to E) \qquad \Gamma_i \vdash R : \rho^2_{i+1} \qquad \mathit{lbl}(R) = \mathit{lbl}(\rho^2_{i+1})}{\Gamma_i \vdash (\mathsf{argClass}\ \mathsf{state}) \oplus R : \rho^1 + \rho^2_{i+1}}\,(+)$$

Hence for all $0 \leq i < n$ we can derive the typing judgment:

$$\mathsf{argClass} : \sigma^1 \to \rho^1 \vdash \lambda\, \mathsf{myClass}\, \lambda\, \mathsf{state}.\, (\mathsf{argClass}\ \mathsf{state}) \oplus R : \\ ((\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2_i)) \to (\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2_{i+1})$$

and therefore, by reasoning as for classes, we get (setting $\rho^2 = \rho^2_n$):

$$\vdash M \equiv \lambda\, \mathsf{argClass}.\, \mathbf{Y}(\lambda\, \mathsf{myClass}\, \lambda\, \mathsf{state}.\, (\mathsf{argClass}\ \mathsf{state}) \oplus R) : \\ (\sigma^1 \to \rho^1) \to (\sigma^1 \cap \sigma^2) \to (\rho^1 + \rho^2) \quad (3)$$

Spelling out this type, we can say that σ^1 is a type of the state of the argument-class of M; $\sigma^1 \cap \sigma^2$ is the type of the state of the resulting class, that refines σ^1. ρ^1 expresses the requirements of M about the methods of the argument-class, i.e. what is assumed to hold for the usages of super and $\mathsf{argClass}$ in R to be properly typed; $\rho^1 + \rho^2$ is a type of the record of methods of the refined class, resulting from the merge of the methods of the argument-class with those of the difference R; since in general there will be overridden methods, whose types might be incompatible, the + type constructor cannot be replaced by intersection.

▶ **Example 13.** The mixin Movable, provided an argument class that contains a *set* and a *shift* method, creates a new class with (potentially overwritten) methods *set* and *move* along with delegated methods. The method *set* is fixed to set the underlying state to 1 and the method *move* shifts the state by 1.

$$\mathsf{Movable} = \lambda \mathsf{argClass}.\mathbf{Y}(\lambda \mathsf{myClass}.\lambda \mathsf{state}.\mathtt{let}\ \mathsf{super} = \mathsf{argClass}\ \mathsf{state}\ \mathtt{in} \\ \mathtt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \mathtt{in} \\ \mathsf{super} \oplus \langle \mathit{set} = \mathsf{super}.\mathit{set}(1), \mathit{move} = \mathsf{self}.\mathit{shift}(1)\rangle)$$

Note that to update self and super one can use $\mathsf{myClass}$ and $\mathsf{argClass}$. Generalizing for all $\rho \in \mathbb{T}_{\langle\rangle}$ following the argumentation above we can choose:

$$\rho^1 = \rho \cap \langle \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int}\rangle \qquad \rho^2 = \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \mathrm{Int}\rangle \\ \sigma^1 = \mathrm{Int} \qquad \sigma^2 = \omega \\ \rho^2_1 = \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \omega\rangle \qquad \rho^2_2 = \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \mathrm{Int}\rangle$$

Using these choices we can type $\mathbf{Y}$ by

$$\vdash \mathbf{Y} : (\omega \to (\mathrm{Int} \to \rho^1 + \rho^2_1)) \cap ((\mathrm{Int} \to \rho^1 + \rho^2_1) \to (\mathrm{Int} \to \rho^1 + \rho^2_2)) \to (\mathrm{Int} \to \rho^1 + \rho^2_2)$$

and obtain the typings of Movable for all ρ:

$$\dfrac{\dfrac{\dots}{\vdash \mathrm{Movable} : (\mathrm{Int} \to \rho \cap \langle \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int}\rangle) \to (\mathrm{Int} \to \rho^1 + \rho^2_2)}}{\vdash \mathrm{Movable} : (\mathrm{Int} \to \rho \cap \langle \mathit{set} : \mathrm{Int} \to \mathrm{Int}, \mathit{shift} : \mathrm{Int} \to \mathrm{Int}\rangle) \to (\mathrm{Int} \to \rho + \langle \mathit{set} : \mathrm{Int}, \mathit{move} : \mathrm{Int}\rangle)}\ (\leq)$$

3 Encoding of Record Types in Bounded Combinatory Logic

Our main goal is to combine type information given by intersection types for Λ_R and the capabilities of the logical programming language given by BCL inhabitation to synthesize meaningful mixin compositions as terms of a combinatory logic. Such *combinatory terms* are formed by application of combinators from a *repository* (combinatory logic context) Δ.

▶ **Definition 14** (Combinatory Term). $E, E' ::= C \mid (E\ E'),\ C \in dom(\Delta)$

We create repositories of typed combinators that can be considered logic programs for the existing BCL synthesis framework (CL)S [7] to reason about semantics of such compositions. The underlying type system of (CL)S is an extension of the intersection type system **BCD** [4] by covariant constructors. The extended type system $\mathbb{T}_{\mathbb{C}}$, while suited for synthesis, is flexible enough to encode record types and features of +. While (CL)S implements covariant constructors of arbitrary arity, we only use unary constructors, which are sufficient for our encoding.

▶ **Definition 15** (Intersection Types with Constructors $\mathbb{T}_{\mathbb{C}}$). The set $\mathbb{T}_{\mathbb{C}}$ is given by:

$$\mathbb{T}_{\mathbb{C}} \ni \sigma, \tau, \tau_1, \tau_2 \ ::= a \mid \alpha \mid \omega \mid \tau_1 \to \tau_2 \mid \tau_1 \cap \tau_2 \mid c(\tau)$$

where a ranges over constants, α over type variables and c over unary constructors $\mathbb{C}$.

$\mathbb{T}_{\mathbb{C}}$ adds the following two subtyping axioms to the **BCD** system

$$\tau_1 \leq \tau_2 \Rightarrow c(\tau_1) \leq c(\tau_2) \qquad c(\tau_1) \cap c(\tau_2) \leq c(\tau_1 \cap \tau_2)$$

The additional axioms ensure *constructor distributivity*, i.e., $c(\tau_1) \cap c(\tau_2) = c(\tau_1 \cap \tau_2)$.

▶ **Definition 16** (Type Assignment in $\mathbb{T}_{\mathbb{C}}$).

$$\frac{C : \tau \in \Delta \qquad S \text{ Substitution}}{\Delta \vdash_{\mathrm{BCL}} C : S(\tau)}\ (\mathrm{Var}) \qquad \frac{\Delta \vdash_{\mathrm{BCL}} E : \sigma \to \tau \qquad \Delta \vdash_{\mathrm{BCL}} E' : \sigma}{\Delta \vdash_{\mathrm{BCL}} EE' : \tau}\ (\to E)$$

$$\frac{\Delta \vdash_{\mathrm{BCL}} E : \sigma \quad \Delta \vdash_{\mathrm{BCL}} E : \tau}{\Delta \vdash_{\mathrm{BCL}} E : \sigma \cap \tau}\ (\cap) \qquad \frac{\Delta \vdash_{\mathrm{BCL}} E : \sigma \qquad \sigma \leq \tau}{\Delta \vdash_{\mathrm{BCL}} E : \tau}\ (\leq)$$

We extend the necessary property of *beta-soundness* and a notion of *paths* and *organized types* from [20] to constructors in the following way.

▶ **Lemma 17** (Extended Beta-Soundness). *If* $\bigcap_{i \in I}(\sigma_i \to \tau_i) \cap \bigcap_{j \in J} c_j(\tau_j) \cap \bigcap_{k \in K} \alpha_k \cap \bigcap_{k' \in K'} a_{k'} \leq c(\tau)$, *then* $\{j \in J \mid c_j = c\} \neq \varnothing$ *and* $\bigcap\{\tau_j \mid j \in J, c_j = c\} \leq \tau$.

▶ **Definition 18** (Path). A *path* π is a type of the form: $\pi ::= a \mid \alpha \mid \tau \to \pi \mid c(\omega) \mid c(\pi)$, where α is a variable, τ is a type, c is a constructor and a is a constant.

▶ **Definition 19** (Organized Type). A type τ is called organized, if it is an intersection of paths $\tau \equiv \bigcap_{i \in I} \tau_i$, where τ_i for $i \in I$ are paths.

Similarly, we obtain the following property of subtyping w.r.t. organized types.

▶ **Lemma 20.** *Given two organized types* $\tau \equiv \bigcap_{i \in I} \tau_i$ *and* $\sigma \equiv \bigcap_{j \in J} \sigma_j$, *we have* $\tau \leq \sigma$ *iff for all* $j \in J$ *there exists an* $i \in I$ *with* $\tau_i \leq \sigma_j$.

Note that for all intersection types there exists an equivalent organized intersection type coinciding with the notion of strict intersection types [2].

For a set of typed combinators Δ and a type $\tau \in \mathbb{T}_{\mathbb{C}}$ we say τ is inhabitable in Δ, if there exists a combinatory term E such that $\Delta \vdash_{\mathrm{BCL}} E : \tau$. In the following we fix a finite set of labels $\mathcal{L} \subseteq$ **Label** that are used in the particular domain of interest for mixin composition synthesis.

Records as Unary Covariant Distributing Constructors

We define constructors $\langle\!\langle\cdot\rangle\!\rangle$ and $l(\cdot)$ for $l \in \mathcal{L}$ to represent record types using the following partial translation function

$$[\![\cdot]\!]\colon \mathbb{T} \to \mathbb{T}_{\mathbb{C}}, [\![\tau]\!] = \begin{cases} \tau & \text{if } \tau \equiv \omega \text{ or } \tau \equiv a \\ [\![\tau_1]\!] \to [\![\tau_2]\!] & \text{if } \tau \equiv \tau_1 \to \tau_2 \\ [\![\tau_1]\!] \cap [\![\tau_2]\!] & \text{if } \tau \equiv \tau_1 \cap \tau_2 \\ \langle\!\langle l(\tau) \rangle\!\rangle & \text{if } \tau \equiv \langle l : \tau \rangle \\ \langle\!\langle \omega \rangle\!\rangle & \text{if } \tau \equiv \langle\rangle \\ \text{undefined} & \text{else} \end{cases}$$

Since atomic records are covariant and distribute over $\cap$, the presented translation preserves subtyping. We have $[\![\langle l_i : \tau_i \mid i \in I\rangle]\!] = [\![\bigcap_{i\in I} \langle l_i : \tau_i\rangle]\!] = \bigcap_{i\in I} \langle\!\langle l_i([\![\tau_i]\!])\rangle\!\rangle$ if $I \neq \varnothing$.

Note that the translation function $[\![\cdot]\!]$ is not defined for types containing + in $\mathbb{T}$. Additionally, + has non-monotonic properties and therefore cannot be immediately represented by a covariant type constructor. Simply applying Lemma 5(3) is impossible, if the left-hand side of + is all-quantified. There are two possibilities to deal with this situation. The first option is extending the type-system used for inhabitation. Here, the main difficulty is that existing versions of the inhabitation algorithm crucially rely on the separation of intersections into paths [20]. As demonstrated in the remark accompanying Lemma 7, it becomes unclear how to perform such a separation in the presence of the non-monotonic + operation. The second option, pursued in the rest of this section, is to use the expressiveness of the logical programming language given by $\mathsf{BCL}(\mathbb{T}_{\mathbb{C}})$ inhabitation. Specifically, encoding $\mathbb{T}$ types containing + as $\mathbb{T}_{\mathbb{C}}$ types accompanied by following repositories $\Delta_{\mathcal{L}}$ and $\Delta_{\wp(\mathcal{L})}$ suited for $\mathsf{BCL}(\mathbb{T}_{\mathbb{C}})$ inhabitation. We introduce $|\mathcal{L}|$ distinct variables $\alpha_{l'}$ indexed by $l' \in \mathcal{L}$ and $2^{|\mathcal{L}|}$ distinct constructors $w_L(\cdot)$ indexed by $L \subseteq \mathcal{L}$.

$$\Delta_{\mathcal{L}} = \{W_{\{l\}} : w_{\{l\}}(\bigcap_{l'\in\mathcal{L}\setminus\{l\}} \langle\!\langle l'(\alpha_{l'})\rangle\!\rangle) \mid l \in \mathcal{L}\}$$

$$\Delta_{\wp(\mathcal{L})} = \{W_L : w_{\{l_1\}}(\alpha) \to w_{\{l_2\}}(\alpha) \to \ldots \to w_{\{l_k\}}(\alpha) \to w_L(\alpha) \mid k \geq 2, \{l_1, \ldots, l_k\} = L \subseteq \mathcal{L}\}\,.$$

These repositories are purely logical in a sense that they do not represent terms in Λ_R but encode necessary side conditions in the logic program. In particular, we formalize the conditions for the absence of a label in a record type by the following Lemma 21. This encoding of negative information is crucial to encode non-monotonic properties of +.

▶ **Lemma 21.** *Let $l \in \mathcal{L}$ be a label and let $\tau \in \mathbb{T}$ be a type such that $[\![\tau]\!]$ is defined. $w_{\{l\}}([\![\tau]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ iff $\tau \in \mathbb{T}_{\langle\rangle} \cup \{\omega\}$ with $l \notin lbl(\tau)$.*

Proof Sketch. Let $l \in \mathcal{L}$ be a label.

(⇒) Let wlog. $[\![\tau]\!] \equiv \bigcap_{i\in I} \tau_i$ be an organized intersection type such that $w_{\{l\}}([\![\tau]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$. The only combinator with a matching target to inhabit $w_{\{l\}}([\![\tau]\!])$ is $W_{\{l\}}$. Due to distributivity of type constructors there exists a substitution S such that $S(w_{\{l\}}(\bigcap_{l'\in\mathcal{L}\setminus\{l\}} \langle\!\langle l'(\alpha_{l'})\rangle\!\rangle)) \leq w_{\{l\}}([\![\tau]\!])$. By Lemma 20 followed by Lemma 17 we obtain for each $i \in I$ that there exists an $l' \in \mathcal{L} \setminus \{l\}$ such that $\tau_i = \langle\!\langle l'([\![\sigma_i]\!])\rangle\!\rangle$ for some type $\sigma_i \in \mathbb{T}$. By definition of $[\![\cdot]\!]$ and distributivity we obtain $\tau \in \mathbb{T}_{\langle\rangle} \cup \{\omega\}$ with $l \notin lbl(\tau)$.

(⇐) Let $\tau = \bigcap_{l'\in L} \langle l' : \tau_{l'}\rangle$ (resp. $\langle\rangle$) for some $L \subseteq \mathcal{L} \setminus \{l\}$ and types $\tau_{l'} \in \mathbb{T}$ for $l' \in L$. We have $W_{\{l\}} : S(w_{\{l\}}(\bigcap_{l'\in\mathcal{L}\setminus\{l\}} \langle\!\langle l'(\alpha_{l'})\rangle\!\rangle)) \leq w_{\{l\}}([\![\tau]\!])$ for a substitution $S(\alpha_{l'}) = \begin{cases} [\![\tau_{l'}]\!] & \text{if } l' \in L \\ \omega & \text{else} \end{cases}$

◀

▶ **Lemma 22.** *Let $L \subseteq \mathcal{L}$ be a non-empty set of labels and let $\tau \in \mathbb{T}$ be a type such that $[\![\tau]\!]$ is defined. $w_L([\![\tau]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ iff $\tau \in \mathbb{T}_{\langle\rangle} \cup \{\omega\}$ with $L \cap lbl(\tau) = \varnothing$.*

Proof Sketch. Using W_L and Lemma 21 for each argument of W_L. ◀

Our intermediate goal is to use inhabitation in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ to translate types of the shape $\rho + \bigcap_{l \in L} \langle l : \tau_l \rangle$ into $\rho \cap \bigcap_{l \in L} \langle l : \tau_l \rangle$, which in general is incorrect. The following Lemma 23 describes sufficient circumstances where this translation holds.

▶ **Lemma 23** ($\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ Translation Soundness)**.** *Let $L \subseteq \mathcal{L}$ be a non-empty set of labels, let $\rho \in \mathbb{T}_{\langle\rangle}$ be a type such that $[\![\rho]\!]$ is defined and let $\tau_l \in \mathbb{T}$ for $l \in L$ be types. If $w_L([\![\rho]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ then $\rho + \bigcap_{l \in L} \langle l : \tau_l \rangle = \rho \cap \bigcap_{l \in L} \langle l : \tau_l \rangle$.*

Proof Sketch. Since $w_L([\![\rho]\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ we have $lbl(\rho) \cap L = \varnothing$. The result follows from Lemma 6 (2). ◀

Next, we show by the following Lemma 24 that inhabitation in $\Delta_{\mathcal{L}}$ is not too restrictive.

▶ **Lemma 24** ($\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ Translation Completeness)**.** *Let $L \subseteq \mathcal{L}$ be a non-empty set of labels, let $\tau_l \in \mathbb{T}$ for $l \in L$ be types, let $\rho \in \mathbb{T}_{\langle\rangle}$ be a type such that $[\![\rho]\!]$ is defined. There exists a type $\rho' \in \mathbb{T}_{\langle\rangle}$ such that $w_L([\![\rho']\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ and $\rho' \cap \bigcap_{l \in L} \langle l : \tau_l \rangle \leq \rho + \bigcap_{l \in L} \langle l : \tau_l \rangle$.*

Proof Sketch. Given $\rho = \bigcap_{l' \in L'} \langle l' : \tau'_{l'} \rangle$ choose $\rho' = \bigcap_{l' \in L' \setminus L} \langle l' : \tau'_{l'} \rangle$ (resp. $\rho' = \langle\rangle$ if $L' \setminus L = \varnothing$). By Lemma 22 $w_L([\![\rho']\!])$ is inhabitable in $\Delta_{\mathcal{L}} \cup \Delta_{\wp(\mathcal{L})}$ and $\rho' \cap \bigcap_{l \in L} \langle l : \tau_l \rangle \leq \rho + \bigcap_{l \in L} \langle l : \tau_l \rangle$. ◀

Note that in Lemma 24 the type ρ' can be chosen greater than ρ only due to the non-monotonic properties of $+$.

4 Mixin Composition Synthesis by Type Inhabitation

In this section we denote type assignment in Λ_R by $\vdash_{\langle\rangle}$ and fix the following ingredients:

- A finite set of classes $\mathcal{C}$.
- For each $C \in \mathcal{C}$ types $\sigma_C \in \mathbb{T}, \rho_C \in \mathbb{T}_{\langle\rangle}$ such that $[\![\sigma_C \to \rho_C]\!]$ is defined and $\varnothing \vdash_{\langle\rangle} C : \sigma_C \to \rho_C$.
- A finite set of mixins $\mathcal{M}$.
- For each $M \in \mathcal{M}$ types $\sigma_M \in \mathbb{T}$ and $\rho^1_M, \rho^2_M \in \mathbb{T}_{\langle\rangle}$ such that $[\![\sigma_M]\!], [\![\rho^1_M]\!], [\![\rho^2_M]\!]$ are defined and for all types $\rho \in \mathbb{T}_{\langle\rangle}$ we have $\varnothing \vdash_{\langle\rangle} M : (\sigma_M \to \rho \cap \rho^1_M) \to (\sigma_M \to \rho + \rho^2_M)$.
- For each $M \in \mathcal{M}$ the non-empty set of labels $L_M = lbl(\rho^2_M) \subseteq \mathcal{L}$.

We translate given classes and mixins to the following a repository $\Delta^{\mathcal{C},\mathcal{M}}_{\mathcal{L}}$ of combinators

$$\begin{aligned}\Delta^{\mathcal{C},\mathcal{M}}_{\mathcal{L}} =& \{C : [\![\sigma_C \to \rho_C]\!] \mid C \in \mathcal{C}\} \\ & \cup \{M : w_{L_M}(\alpha) \to ([\![\sigma_M]\!] \to \alpha \cap [\![\rho^1_M]\!]) \to ([\![\sigma_M]\!] \to \alpha \cap [\![\rho^2_M]\!]) \mid M \in \mathcal{M}\} \\ & \cup \Delta_{\mathcal{L}} \cup \{W_{L_M} \in \Delta_{\wp(\mathcal{L})} \mid M \in \mathcal{M}, |L_M| > 1\}\end{aligned}$$

To simplify notation, we introduce the infix metaoperator $\gg$ such that $x \gg f = f\ x$. It is right associative and has the lowest precedence. Accordingly, $x \gg f \gg g = g\ (f\ x)$.

Although types in $\Delta^{\mathcal{C},\mathcal{M}}_{\mathcal{L}}$ do not contain record-merge, we show by following Theorem 25 that types of mixin compositions in $\mathsf{BCL}(\to, \cap)$ are sound.

▶ **Theorem 25** (Soundness). *Let $M_1, \ldots, M_n \in \mathcal{M}$ be mixins, let $L_1, \ldots, L_n \subseteq \mathcal{L}$ be sets of labels, let $C \in \mathcal{C}$ be a class and let $\sigma \in \mathbb{T}, \rho \in \mathbb{T}_{\langle\rangle}$ be types such that $[\![\sigma \to \rho]\!]$ is defined.*
If $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} \vdash_{BCL} C \gg (M_1\ W_{L_1}) \gg (M_2\ W_{L_2}) \gg \ldots \gg (M_n\ W_{L_n}) : [\![\sigma \to \rho]\!]$,
then $\varnothing \vdash_{\langle\rangle} C \gg M_1 \gg M_2 \gg \ldots \gg M_n : \sigma \to \rho$.

Proof Sketch. Induction on n proving $L_i = L_{M_i}$ followed by Lemma 23 and $(\to E)$. ◀

Complementary, we show by the following Theorem 26 that typing of mixin compositions in $\mathsf{BCL}(\to, \cap)$ is complete with respect to previously described typing in $\mathbb{T}$.

▶ **Theorem 26** (Partial Completeness). *Let $\Gamma \subseteq \{x_C : \sigma_C \to \rho_C \mid C \in \mathcal{C}\} \cup \{x_M^\rho : (\sigma_M \to \rho \cap \rho_M^1) \to (\sigma_M \to \rho + \rho_M^2) \mid M \in \mathcal{M}, \rho \in \mathbb{T}_{\langle\rangle}, [\![\rho]\!]$ is defined$\}$ be a finite context and let $\sigma \in \mathbb{T}, \rho \in \mathbb{T}_{\langle\rangle}$ be types such that $[\![\sigma \to \rho]\!]$ is defined.*
If $\Gamma \vdash_{\langle\rangle} x_C \gg x_{M_1}^{\rho_1} \gg x_{M_2}^{\rho_2} \gg \ldots \gg x_{M_n}^{\rho_n} : \sigma \to \rho$,
then $\Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}} \vdash_{BCL} C \gg (M_1\ W_{L_{M_1}}) \gg (M_2\ W_{L_{M_2}}) \gg \ldots \gg (M_n\ W_{L_{M_n}}) : [\![\sigma \to \rho]\!]$.

Proof Sketch. Induction on n choosing for each x_M^ρ where $\rho = \bigcap_{l \in L} \langle l : \tau_l \rangle$ (resp. $\rho = \langle\rangle$) the substitution $S_i(\alpha) = \bigcap_{l \in L \setminus L_M} \langle\!\langle l(\tau_l) \rangle\!\rangle$ to type $M \in \Delta_{\mathcal{L}}^{\mathcal{C},\mathcal{M}}$ and using $(\to E)$ and Lemma 24. ◀

Coming back to our running example, we obtain

$$
\begin{aligned}
\Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{get,set,shift,move\}} = \{\ \text{Point} :\ & \text{Int} \to \langle\!\langle get(\text{Int}) \cap set(\text{Int} \to \text{Int}) \cap shift(\text{Int} \to \text{Int}) \rangle\!\rangle, \\
\text{Movable} :\ & w_{\{set,move\}}(\alpha) \\
& \to (\text{Int} \to \alpha \cap \langle\!\langle set(\text{Int} \to \text{Int}) \cap shift(\text{Int} \to \text{Int}) \rangle\!\rangle) \\
& \to (\text{Int} \to \alpha \cap \langle\!\langle set(\text{Int}) \cap move(\text{Int}) \rangle\!\rangle), \\
W_{\{get\}} :\ & w_{\{get\}}(\langle\!\langle set(\alpha_1) \cap shift(\alpha_2) \cap move(\alpha_3) \rangle\!\rangle), \\
W_{\{set\}} :\ & w_{\{set\}}(\langle\!\langle get(\alpha_1) \cap shift(\alpha_2) \cap move(\alpha_3) \rangle\!\rangle), \\
W_{\{shift\}} :\ & w_{\{shift\}}(\langle\!\langle get(\alpha_1) \cap set(\alpha_2) \cap move(\alpha_3) \rangle\!\rangle), \\
W_{\{move\}} :\ & w_{\{move\}}(\langle\!\langle get(\alpha_1) \cap set(\alpha_2) \cap shift(\alpha_3) \rangle\!\rangle), \\
W_{\{set,move\}} :\ & w_{\{set\}}(\alpha) \to w_{\{move\}}(\alpha) \to w_{\{set,move\}}(\alpha) \}
\end{aligned}
$$

We may ask inhabitation questions such as

$$\Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{get,set,shift,move\}} \vdash_{\text{BCL}} ? : [\![\text{Int} \to \langle shift : \text{Int} \to \text{Int}, move : \text{Int} \rangle]\!]$$

and obtain the combinatory term "Movable $W_{\{set,move\}}$ Point" as a synthesized result. From Theorem 25 we know

$$\varnothing \vdash_{\langle\rangle} \text{Movable Point} : \text{Int} \to \langle shift : \text{Int} \to \text{Int}, move : \text{Int} \rangle$$

On the other hand, if we want to inhabit $[\![\text{Int} \to \langle set : \text{Int} \to \text{Int}, move : \text{Int} \rangle]\!]$ we obtain no results. From Lemma 26 we know that, restricted to the previously described typing in $\mathbb{T}$, there is no mixin composition applied to a class with the resulting type $\text{Int} \to \langle set : \text{Int} \to \text{Int}, move : \text{Int} \rangle$.

The presented encoding has several benefits with respect to scalability. First, the size of the presented repositories is polynomial in $|\mathcal{L}| * |\mathcal{C}| * |\mathcal{M}|$. Second, expanding the label set $\mathcal{L}$ requires only to update combinators in $\Delta_{\mathcal{L}}$ leaving existing types of classes and mixins untouched. Third, adding a class/mixin to an existing repository is as simple as adding one typed combinator for the class/mixin and at most one logical combinator. Again, it is important that the existing combinators in the repository remain untouched. As an example, we add the following mixin MovableBy to $\Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{get,set,shift,move\}}$.

$$\begin{aligned}\text{MovableBy} = \lambda\mathsf{argClass}.\mathbf{Y}(\lambda\mathsf{myClass}.\lambda\mathsf{state}.\\ &\texttt{let}\ \mathsf{super} = \mathsf{argClass}\ \mathsf{state}\ \texttt{in}\\ &\texttt{let}\ \mathsf{self} = \mathsf{myClass}\ \mathsf{state}\ \texttt{in}\ \mathsf{super} \oplus \langle \mathit{move} = \mathsf{super}.\mathit{shift}\rangle)\end{aligned}$$

In Λ_R for all types $\rho \in \mathbb{T}_{\langle\rangle}$ we have

$$\varnothing \vdash_{\langle\rangle} \text{MovableBy}: (\text{Int} \to \rho \cap \langle \mathit{shift} : \text{Int} \to \text{Int}\rangle) \to (\text{Int} \to \rho + \langle \mathit{move} : \text{Int} \to \text{Int}\rangle)$$

We obtain the following extended repository

$$\begin{aligned}\Delta^{\{\text{Point}\},\{\text{Movable},\text{MovableBy}\}}_{\{\mathit{get},\mathit{set},\mathit{shift},\mathit{move}\}} = \Delta^{\{\text{Point}\},\{\text{Movable}\}}_{\{\mathit{get},\mathit{set},\mathit{shift},\mathit{move}\}} \cup \{&\text{MovableBy} : w_{\{\mathit{move}\}}(\alpha)\\ &\to (\text{Int} \to \alpha \cap \langle\!\langle \mathit{shift}(\text{Int} \to \text{Int})\rangle\!\rangle) \to (\text{Int} \to \alpha \cap \langle\!\langle \mathit{move}(\text{Int} \to \text{Int})\rangle\!\rangle)\}\end{aligned}$$

Asking the inhabitation question

$$\Delta^{\{\text{Point}\},\{\text{Movable},\text{MovableBy}\}}_{\{\mathit{get},\mathit{set},\mathit{shift},\mathit{move}\}} \vdash_{\text{BCL}} ? : [\![\text{Int} \to \langle \mathit{set} : \text{Int}, \mathit{move} : \text{Int} \to \text{Int}\rangle]\!]$$

synthesizes "Point $\gg$ (Movable $W_{\{\mathit{set},\mathit{move}\}}$) $\gg$ (MovableBy $W_{\{\mathit{move}\}}$)". Note that even in such a simplistic scenario the order in which mixins are applied can be crucial mainly because $\oplus$ is not commutative. Moreover, the early binding of self and the associated preservation of overwritten methods may make multiple applications of a single mixin meaningful.

5 Conclusion and Future Work

We presented a theory for automatic compositional construction of object oriented classes by combinatory synthesis. This theory is based on the λ-calculus with records and $\oplus$ typed by intersection types with records and $+$. It is capable of modeling classes as states to records (i.e. objects), and mixins as functions from classes to classes. Mixins can be assigned meaningful types using $+$ expressing their compositional character. However, non-monotonic properties of $+$ are incompatible with the existing well-studied theory of $\mathsf{BCL}(\to,\cap)$ synthesis. Therefore, we designed a translation to repositories of combinators typed in $\mathsf{BCL}(\mathbb{T}_{\mathbb{C}})$. We have proven this translation to be sound (Theorem 25) and partially complete (Theorem 26). A notable feature is the encoding of negative information (the absence of labels). It exploits the logic programming capabilities of inhabitation, by adding sets of combinators serving as witnesses for the non-presence of labels. In section 4 we also showed that this encoding scales wrt. extension of repositories.

Future work includes further studies on the possibilities to encode predicates exploiting patterns similar to the negative information encoding. The partial completeness result indicates a more expressive power of type constructors compared to records. Another direction of future work is to extend types of mixins and classes by semantic as well as modal types [19], a development initiated in [5]. In particular, the expressiveness of semantic types can be used to assign meaning to multiple applications of a single mixin and allow to reason about object oriented code on a higher abstraction level as well as higher semantic accuracy.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments.

References

1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2 Steffen Van Bakel. Strict intersection types for the lambda calculus. *ACM Comput. Surv.*, 43(3):20:1–20:49, April 2011.
3 H. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
4 H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
5 Jan Bessai, Boris Düdder, Andrej Dudenhefer, and Moritz Martens. Delegation-based mixin composition synthesis. `http://www-seal.cs.tu-dortmund.de/seal/downloads/papers/paper-ITRS2014.pdf`, 2014.
6 Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. *arXiv preprint arXiv:1503.04911*, 2015.
7 Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. Combinatory Logic Synthesizer. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA'14*, volume 8802, pages 26–40, 2014.
8 Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66, 1999.
9 Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Univeristy of Utha, 1992.
10 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, 1990.
11 Kim B. Bruce. *Foundations of Object-Oriented Languages – Types and Semantics*. MIT Press, 2002.
12 Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280, 1989.
13 Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173, pages 51–67, 1984.
14 Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.
15 William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL'90*, pages 125–135. ACM Press, 1990.
16 Ugo de'Liguoro. Characterizing convergent terms in object calculi via intersection types. In *TLCA*, pages 315–328, 2001.
17 Ugo de'Liguoro and Tzu chun Chen. Semantic Types for Classes and Mixins. `http://www.di.unito.it/~deligu/papers/UdLTC14.pdf`, 2014.
18 Boris Düdder. *Automatic Synthesis of Component & Connector-Software Architectures with Bounded Combinatory Logic*. Dissertation, TU Dortmund, 2014.
19 Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 67–86, 2014.
20 Boris Düdder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded Combinatory Logic. In *Proceedings of CSL'12*, volume 16, pages 243–258. Schloss Dagstuhl, 2012.
21 Standard Ecma. ECMA-262 ECMAScript Language Specification, 2011.
22 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete Completion using Types and Weights. *SIGPLAN Notices*, 48(6):27–38, 2013.
23 Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005.
24 Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57. ACM, 2005.

25 Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.

26 Didier Rémy. Typing record concatenation for free. In *POPL'92*, pages 166–176, 1992.

27 Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991.

Non-Constructivity in Kan Simplicial Sets

Marc Bezem[1], Thierry Coquand[2], and Erik Parmann[3]

1 **Department of Informatics, University of Bergen, Norway,** `bezem@ii.uib.no`
2 **Department of Computer Science and Engineering, Chalmers/University of Gothenburg, Sweden,** `thierry.coquand@cse.gu.se`
3 **Department of Informatics, University of Bergen, Norway,** `Erik.Parmann@ii.uib.no`

Abstract

We give an analysis of the non-constructivity of the following basic result: if X and Y are simplicial sets and Y has the Kan extension property, then Y^X also has the Kan extension property. By means of Kripke countermodels we show that even simple consequences of this basic result, such as edge reversal and edge composition, are not constructively provable. We also show that our unprovability argument will have to be refined if one strengthens the usual formulation of the Kan extension property to one with explicit horn-filler operations.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Constructive logic, simplicial sets, semantics of simple types

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.92

1 Introduction

Brouwer's Programme is the constructive reformulation of (as much as possible of) classical mathematics. In [2] it has been shown that the following theorem, though classically true (cf. [10, Corollary 7.11]), cannot be proved constructively.

▶ **Theorem 1** (classical). *The fibers of* 0 *and* 1 *of a Kan fibration* $p : E \to \Delta^1$ *are homotopy equivalent.*

In this paper we show that the following basic theorems cannot be proved constructively.

▶ **Theorem 2** (classical). *If* X *and* Y *are Kan simplicial sets, then any edge in* Y^X *can be reversed.*

▶ **Theorem 3** (classical). *If* X *and* Y *are Kan simplicial sets, then compatible edges in* Y^X *can be composed.*

The above two theorems follow immediately and constructively from the following.

▶ **Theorem 4** (classical). *If* X *and* Y *are Kan simplicial sets, then also* Y^X *is so.*

Hence we obtain that also Theorem 4, though classically true even without requiring that X is Kan (cf. [10, Theorem 6.9]), cannot be proved constructively.

The importance of these results is twofold. First, it is of evident importance for Brouwer's Programme to understand which results of classical mathematics already are constructive and which results are not. Second, Theorem 4 plays a crucial role in the construction of models of type theory with the Univalence Axiom, see [7]. The use of classical logic in proving this crucial property implies in particular that the model construction cannot be used to give a computational interpretation of univalence. Actually, Theorem 4 is a necessary step in the

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 92–106

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

semantics of the simply typed λ-calculus based on Kan simplicial sets. In what follows we expand on these points; for more motivation we refer to [2].

We would like to use the occasion to say a few not-too-technical words on the role of the Kan extension property of simplicial sets in relation to univalence. Let MLTT be Martin-Löf type theory with universe U and inductive equality $=_U$ on U. Assume we have two distinct copies of the natural numbers, inductively defined by constructors $0 : N$ $(0' : N')$ and $S : N \to N$ $(S' : N' \to N')$. MLTT proves $N =_U N$ and $N' =_U N'$, but not $N =_U N'$. The Univalence Axiom (UA) implies that (homotopy) equivalent types are equal, and in particular $N =_U N'$. By the Leibniz property of inductive equality, this implies that N and N' have the same properties and that all structure on N can be transported to N' and vice versa. This holds even uniformly, for example, $\Pi P : U \to U.\ (PN \to PN')$ is inhabited under UA. On the other hand, without UA, $\Pi P : U \to U.\ (PN \to PN')$ is not inhabited in MLTT. (One reason is that MLTT has models in which $N \neq N'$, so one can take $P \equiv \lambda X : U.\ (N =_U X)$ and get PN but not PN').

The above observation concerns not only the rather artificial type N' but also any other type that is equivalent to N, such as the type of lists over a unit type with one object. In fact the observation concerns all equivalent types. A less artificial example is perhaps the equivalence of the unit type to $\Sigma x : A.\ (a =_A x)$ for given $a : A : U$. The upshot is that validating UA requires an interpretation of $=_U$ that carries much more information than in MLTT without UA, since the elimination rule for $=_U$ (roughly, the Leibniz property, or substitutivity of equals for equals) has to be much stronger. In our simple example, the interpretation of $=_U$ must be leveraged to give an inhabitant of $\Pi P : U \to U.\ (PN \to PN')$.

Simplicial sets can be used to build a presheaf-style [6] model of MLTT. In this model the interpretation of $=_U$ does not validate UA. It turns out that if one builds a model of MLTT based on Kan simplicial sets, then it is possible to validate UA. The crucial notion here is that of a Kan fibration. A Kan *fibration* $p : E \to B$ is a map of simplicial sets with a specific lifting property. This lifting property *lifts* a path from b_0 to b_1 in B to a transport function from the fiber $p^{-1}(b_0)$ to the fiber $p^{-1}(b_1)$. In the model based on Kan simplicial sets, an inhabitant of $N =_U N'$ is interpreted as a path from N to N' in U. (Here and below we omit the correct but tedious phrase *the interpretation of* $N, N', U, \ldots$). Any $P : U \to U$ is interpreted as a Kan fibration with fibers PT for any $T : U$. (NB the fibration, being a projection on the base type, has a direction opposite to the arrow in $P : U \to U$). Then the transport function obtained from the lifting property is the desired function $PN \to PN'$. In short, one can say that the transport functions interpret substitutivity of equals by equals.

Finally, to come back to the topic of this paper: if all types are to have Kan structure, one has to prove this inductively following the rules of type formation. One of the induction steps is Theorem 4. The unprovablility of Theorem 4 shows that, from the constructive point of view, there is a problem with using the exponent Y^X in the category of Kan simplicial sets to interpret function types $X \to Y$.

The type theoretic (synthetic) formulation of homotopy equivalence and the Univalence Axiom, as well as the model of MLTT plus UA using Kan simplicial sets are all due to Voevodsky [14, 7]. This model confirms the homotopical interpretation proposed by Awodey and Warren [1].

Theorem 4 (without requiring that X is Kan) has an interesting history. The first appearance seems to be [12, Appendix A, p. 1A-8, Theorem 3]. Moore credits A. Heller for the definition of the function space Y^X on page 1A-4. Moore's proof is combinatorial, using the excluded middle in distinguishing the cases a non/degenerate on page 1A-9, l. 17ff. (Typo: on page 1A-7, l. 12 and 15, the map F is missing on the rhs; evidently $F_{(\mu,\nu)}$ was

intended to depend on F.) The proof in [10, Theorem 6.9] is much the same as the one by Moore (with the F's in place). Several variations of this argument can be found in the literature.

An essentially more abstract proof using anodyne extensions is given by Gabriel and Zisman in [4, Chapter Four, 3.1.2] (take $B = \Delta^0$). Here the classical reasoning shows up when in 2.1.2 amalgamated sums over sets of non-degenerate simplices are taken.

The results of Moore and Heller imply that Kan simplicial sets form a cartesian closed category, which can be seen as a germ of the fact that they model dependent type theory.

The rest of the paper is structured as follows. In Section 2 we give an introduction to simplicial sets, and in Section 3 we provide several examples of simplicial sets which will be in use in the rest of the article. In Section 4 we take a closer look at Theorem 2, and provide a Kripke model showing that a constructive consequence, Lemma 14 cannot be proven constructively. Section 5 deals with edge compostion, much in the same way as Section 4 deals with edge reversal. A summary and evaluation of the results obtained so far is given in Section 6. In Section 7 we strengthen the Kan condition and prove constructively a weak version of Lemma 14. This shows that our unprovability argument will have to be refined for the stronger Kan condition. We sum up our findings and discuss further research in Section 8.

2 Preliminaries

▶ **Definition 5** (Simplicial set)**.** A *simplicial set* A is a collection of sets $A[i]$ for $i \in \mathbb{N}$ such that for every $0 < n$ and $j \leq n$ we have a function (*face map*) $d^n_j : A[n] \to A[n-1]$, and for every $0 \leq n$ and $j \leq n$ we have a function (*degeneracy map*) $s^n_j : A[n] \to A[n+1]$, satisfying the following *simplicial identities* for all suitable superscripts, which we happily omit:

$$d_i d_j = d_{j-1} d_i \qquad \text{if } i < j \tag{1}$$
$$d_i s_j = s_{j-1} d_i \qquad \text{if } i < j \tag{2}$$
$$d_i s_j = id \qquad \text{for } i = j, j+1 \tag{3}$$
$$d_i s_j = s_j d_{i-1} \qquad \text{if } i > j+1 \tag{4}$$
$$s_i s_j = s_j s_{i-1} \qquad \text{if } i > j \tag{5}$$

An element of $A[i]$ is called an *i-simplex*, or just simplex when we don't wish to stipulate the dimension. A *degenerate* element is any element $a \in A[i+1]$ in the image of a degeneracy map.

Note that a simplicial identity like, e.g., $d^n_i d^{n+1}_j = d^n_{j-1} d^{n+1}_i$ actually means

$$\forall x \in A[n+1].\ d^n_i(d^{n+1}_j(x)) = d^n_{j-1}(d^{n+1}_i(x)).$$

With a countably infinite signature, the above definition can be expressed completely in many-sorted first-order logic. That means that we can see first-order models which satisfy the above requirement as simplicial sets, and instead of simplicial sets we could talk about first-order models satisfying the above requirements.

Simplicial sets form a category. For two simplicial sets A and B, $Hom_S(A, B)$ is the set of all natural transformations from A to B. A natural transformation is a collection of maps $g[n] : A[n] \to B[n]$ commuting with the face and degeneracy maps of A and B: $g[n]s_i = s_i g[n-1]$ for all $0 \leq i < n$ and $g[n+1]d_i = d_i g[n]$ for all $0 \leq i \leq n+1$. We freely omit the dimension $[n]$ when it can be inferred from the other arguments. For more information on simplicial sets we refer to, for example, [10, 5, 3].

▶ **Definition 6** (Kan simplicial set). A simplicial set Y satisfies the Kan condition if for any collection of simplices $y_0, \ldots, y_{k-1}, y_{k+1}, \ldots, y_n$ in $Y[n-1]$ such that $d_i y_j = d_{j-1} y_i$ for any $i < j$ with $i \neq k$ and $j \neq k$, there is an n-simplex y in Y such that $d_i y = y_i$ for all $i \neq k$. The Kan condition is also called the Kan extension property, and a simplicial set is called a *Kan simplicial set* if it satisfies the Kan condition.

▶ **Definition 7** (Kan graph). A *reflexive multigraph* consists of C_1, C_0, d_0, d_1, s where C_0 is a set of points, C_1 a set of edges, $d_i : C_1 \to C_0$, d_1 the *source* and d_0 the *target* function, and $s : C_0 \to C_1$ the function mapping each $c \in C_0$ to a selfloop of c. We write $e : a \to b$ if e is in C_1 such that $d_1(e) = a$ and $d_0(e) = b$ (note the direction!). In particular we have $d_i(s(c)) = c$ for all $c \in C_0$. A *Kan graph* is a reflexive multigraph having the property that for all a, b, c in C_0, if $e : a \to b$ and $f : a \to c$, then there exists an edge $g : b \to c$ in C_1.

Kan graphs can be viewed as truncated Kan simplicial sets, modelling a truncated proof-relevant equality relation. Note that we don't require the Kan graph to have explicit functions giving the required edges like in [2], we merely require that the edges exists. We discuss this distinction further in Section 7. The special requirement of the edges for the Kan graph is in the literature often called *Euclidean*. Euclidean combined with reflexivity gives both transitivity and symmetry.

3 Examples of simplicial sets

We give some examples of simplicial sets that are used in the sequel.

3.1 Standard simplicial k-simplex Δ^k

Δ^k is the simplicial set with $\Delta^k[j]$ consisting of all non-decreasing sequences of numbers $0, \ldots, k$ of length $j+1$. Equivalently, $\Delta^k[j]$ is the set of order-preserving functions $[j] \to [k]$, where $[i]$ denotes $0, \ldots, i$ with the natural ordering. Examples are $\Delta^1[0] = \{0, 1\}$, $\Delta^1[1] = \{00, 01, 11\}$, $\Delta^2[1] = \{00, 01, 02, 11, 12, 22\}$ and

$$\Delta^2[2] = \{000, 001, 002, 011, 012, 022, 111, 112, 122, 222\}.$$

The degeneracy map $s^j_k : \Delta^i[j] \to \Delta^i[j+1]$ duplicates the k-th element in its input. So, $s^j_k(x_0 \ldots x_k \ldots x_{j+1}) = x_0 \ldots x_k x_k \ldots x_{j+1}$. The face map $d^j_k : \Delta^i[j] \to \Delta^i[j-1]$ deletes the k-th element. So, $d^j_k(x_0 \ldots x_j) = x_0 \ldots x_{k-1} x_{k+1} \ldots x_j$.

3.2 The k-horns Λ^k_j

Λ^k_j is the j'th horn of the standard k-simplex Δ^k, and defined by $\Lambda^k_j[n] = \{f \in \Delta^k[n] \mid [k] - \{j\} \not\subseteq Im(f)\}$. Alternatively, it is $\Delta^k[n]$ except every element must avoid some element not equal to j. For example, $\Lambda^2_0[1] = \{00, 01, 02, 11, \cancel{12}, 22\} = \Delta^2[1] - \{12\}$ (excluding 12, since 12 does not avoid any element not equal to 0). We also have:

$$\Lambda^2_0[2] = \{000, 001, 002, 011, \cancel{012}, 022, 111, \cancel{112}, \cancel{122}, 222\}.$$

The Kan extension condition for a simplicial set Y can also be formulated as: every map $F : \Lambda^k_j \to Y$ can be extended to a map $F' : \Delta^k \to Y$. This is equivalent to Definition 6.

3.3 Cartesian products

For two simplicial sets A and B, $A \times B$ is the simplicial set given by $(A \times B)[i] = A[i] \times B[i]$, and the structural maps d and s use d^A and d^B component-wise (and likewise for s^A and s^B). So if $a \in A[i]$ and $b \in B[i]$ then $(a, b) \in (A \times B)[i]$, and $d_i((a, b)) = (d_i^A(a), d_i^B(b))$. In particular, the degenerate simplices of $A \times B$ are pairs $(s_j^A(a), s_j^B(b)) \in (A \times B)[i + 1]$. (Caveat: this is stronger than both components being degenerate.)

3.4 Function spaces

We give the standard definition [12, p. 1A-4]: Y^X is the simplicial set given by $Y^X[i] = Hom_S(\Delta^i \times X, Y)$, where Hom_S denotes morphisms (natural transformations) of simplicial sets, and structural maps as follows. The face maps $d_k[i] : Y^X[i] \to Y^X[i-1]$ need to map elements of $Hom_S(\Delta^i \times X, Y)$ to $Hom_S(\Delta^{i-1} \times X, Y)$ and the degeneracy maps vice versa. For their definition it is convenient to view a k-simplex in Δ^i as an order-preserving function $a : [k] \to [i]$. Let d_k^* be the strictly increasing function on natural numbers such that $d_k^*(n) = n$ if $n < k$ and $d_k^*(n) = n + 1$ otherwise (d_k^* 'jumps' over k). Given $F \in Hom_S(\Delta^i \times X, Y)$, define $(d_k F)[i](a, x) = F[i](d_k^* a, x)$. For the degeneracy maps, let s_k^* be the weakly increasing function on natural numbers such that $s_k^*(n) = n$ if $n \leq k$ and $s_k^*(n) = n - 1$ otherwise (s_k^* 'duplicates' k). Then define $(s_k F)[i](a, x) = F[i](s_k^* a, x)$.

3.5 The simplicial set defined by a reflexive multigraph

The following definition from [2] gives the general construction of a simplicial set from a reflexive multigraph. It is important to note that, even if the reflexive multigraph is transitive, its simplicial set is not the same as the nerve [5, Example 1.4] of the category defined by the multigraph. The difference is subtle: if we have edges $f : x \to y,\ g : y \to z,\ h, h' : x \to z$, where the composition $gf = h$, then the nerve does not contain the 2-simplex with f, g, h', in contrast to below.

▶ **Definition 8.** Given a reflexive multigraph C we define the simplicial set $S(C)$ as follows. $S(C)[0] = C_0$, $S(C)[1] = C_1$ and $S(C)[n]$, for $n \geq 2$, consisting of all tuples of the form $(u_0, \ldots, u_n; \ldots, e_{ij}, \ldots)$ such that

$$e_{ij} : u_i \to u_j \text{ in } C_1 \text{ for all } 0 \leq i < j \leq n.$$

The maps d_k in $S(C)$ are defined by removing from $(u_0, \ldots, u_n; \ldots, e_{ij}, \ldots)$ the point u_k and all edges e_{ik} and e_{kj}. The maps s_k in $S(C)$ are defined by duplicating the point u_k in $(u_0, \ldots, u_n; \ldots, e_{ij}, \ldots)$, adding an edge $e_{k(k+1)} = s(u_k)$, and duplicating edges and incrementing indices of edges as appropriate. This completes the construction of the simplicial set $S(C)$.

We now see why Kan graphs are named as they are: the S construction above turns them into Kan simplicial sets.

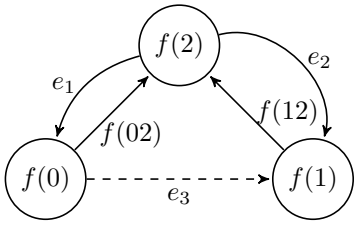
▶ **Lemma 9.** *$S(Y)$ is a Kan simplicial set whenever Y is a Kan graph.*

Proof. Consider Λ_k^n for some $n \geq 1$ and $0 \leq k \leq n$ and let $f : \Lambda_k^n \to S(Y)$. We have to define a lifting $h : \Delta^n \to S(Y)$. Δ^n consists of elements in every dimension, but we only need to specify h for every element in $\Delta^n[n]$. This since both the higher and lower dimensional objects are the (possibly repeated) s_i or d_j images of objects in $\Delta^n[n]$, and h must commute with both s_i and d_j, which determines h.

If $n = 1$, note that that Λ^1_k only consists of one point, and degenerations of that point in the higher dimensions. E.g., if $k = 0$ then $\Lambda^1_0[0] = \{0\}$, $\Lambda^1_0[1] = \{00\}$ etc. In that case we extend f to $h : \Delta^1 \to S(Y)$ by mapping $h(1) = h(0) = f(0)$, which determines h in higher dimensions.

If $n = 2$ we use the fact that Y is a Kan graph, so for any two edges $f : a \to b$ and $g : a \to c$ there is an edge from b to c. The 2-horn gives two edges in the graph with at least one common point, and the fact that the graph is both reflexive, symmetric and transitive (because of the Kan property) enables us to find a third edge with compatible endpoints. The procedure depends on the value of k. We will here give the procedure for $k = 2$; $k = 0, 1$ are just simple adaptations.

Given $f : \Lambda^2_2 \to S(Y)$ we have edges $f(02) : f(0) \to f(2)$ and edges $f(12) : f(1) \to f(2)$, and we need to find a value for $h(01) : f(0) \to f(1)$ such that $d_1 h(01) = d_1 f(02)$ and $d_0 h(01) = d_1 f(12)$. In other words, we need to find that the dotted edge in the diagram actually exists (self-loops are not displayed).



Recall that $S(Y)[1] = Y[1]$, so both $f(01)$ and $f(12)$ are actual edges in Y. By applying the Kan property on $s(f(0))$ and $f(02)$ we get an edge $e_1 : f(2) \to f(0)$. Similarly we get an edge $e_2 : f(2) \to f(1)$. Now, by using the Kan property on e_1 and e_2 we get an edge $e_3 : f(0) \to f(1)$, and we put $h(01) = e_3$.

Finally, if $n \geq 3$ we observe that the horn Λ^n_k contains all points and edges of Δ^n, and we define the lifting by

$$h(q) = (f_{[0]}(q(0)), \ldots, f_{[0]}(q(m)); \ldots, f_{[1]}(e_{ij}), \ldots).$$

Here $q : [m] \to [n]$ is order-preserving and e_{ij} is the edge from $q(i)$ to $q(j)$ in $\Delta^n[1] = \Lambda^n_k[1]$. ◀

4 Edge reversal

In this section we give the classical proof of Theorem 2 and show that there is no constructive proof.

4.1 Edge reversal, definition and classical proof

▶ **Definition 10** (Edge reversal)**.** A simplicial set Y is said to have *edge reversal* when for every edge $e \in Y[1]$ there exists an edge $f \in Y[1]$ with $d_1(f) = d_0(e)$ and $d_0(f) = d_1(e)$.

▶ **Lemma 11.** *Kan simplicial sets have edge reversal.*

Proof. Given an arbitrary Kan simplicial set Y and an edge $e \in Y[1]$ we can make a map $G : \Lambda^2_0 \to Y$ by letting $G(0) = G(2) = d_1(e)$, $G(1) = d_0(e)$, $G(01) = e$ and $G(02) = s(d_1(e))$. Since Y is Kan we can extend G to $G : \Delta^2 \to Y$, giving us a value for $G(12) \in Y[1]$, which must be an edge between $G(1)$ and $G(2) = G(0)$, giving the reverse edge. ◀

We introduce some convenient ad-hoc terminology for later use.

▶ **Definition 12** (Y^X*-good*)**.** Let X and Y are reflexive multigraphs and $F_{01} : X[1] \to Y[1]$. Define $F_0 = d_1 F_{01} s : X[0] \to Y[0]$ and $F_1 = d_0 F_{01} s : X[0] \to Y[0]$. We say that F_{01} is Y^X*-good* when the following two requirements hold for $i = 0, 1$:

- For all $e, e' \in X[1]$, if $d_i(e) = d_i(e')$ then $d_i F_{01}(e) = d_i F_{01}(e')$;
- For all $e \in X[1]$, $F_i d_0(e) = F_i d_1(e)$.

The first requirement expresses that F_{01}, F_0, F_1 respect endpoints, that is, if $e : a \to b$ in $X[1]$, then $F_{01}(e) : F_0(a) \to F_1(b)$ in $Y[1]$. The second requirement ensures that F_0 and F_1 are constant on each weakly connected component of X. (Notice that $F_{01}s(y)$ for $y \in Y[0]$ does not need to map to a degenerate edge, so F_0 and F_1 are not necessarily identical.)

▶ **Lemma 13.** *If X and Y are reflexive multigraphs and $F_{01} : X[1] \to Y[1]$ is Y^X-good, then we can extend F_{01} to a 1-simplex in $S(Y)^{S(X)}$.*

Proof. To be a map in $S(Y)^{S(X)}$ we need to extend $F_{01} : X[1] \to Y[1]$ to a family of maps $F'_{01}[n] : (\Delta^1 \times S(X))[n] \to S(Y)[n]$ which commute with d_i and s_j. Recall the definitions $F_0 = d_1 F_{01} s$ and $F_1 = d_0 F_{01} s$. We define $F'_{01}[n]$ depending on n. If $n = 0$ then the input will have the form (i, x) where $0 \le i \le 1$ and $x \in X[0]$, and we put $F'_{01}[0](i, x) = F_i(x)$. If $n = 1$ the input will have the form (ij, e) where $0 \le i \le j \le 1$ and $e \in X[1]$. If $i = j$ we put $F'_{01}(ij, e) = sF_i(d_0(e))$. Note that since F_{01} is Y^X-good, we know that $F_i(d_0(e)) = F_i(d_1(e))$, justifying our choice of the degenerate edge as the output. If $i < j$ we let $F'_{01}(01, e) = F_{01}(e)$. If $n > 1$ any input to $F'_{01}[n]$ will have the form $(0^a 1^b, (x_0, \dots, x_n; \dots e_{ij}, \dots))$ such that $a + b = n + 1$. We let $F'_{01}[n]$ map this element to the tuple

$$(F_0(x_0), \dots, F_0(x_{a-1}), F_1(x_a) \dots, F_1(x_{a+b-1}); \dots e'_{ij}, \dots),$$

where $e'_{ij} = s(F_0(x_a))$ if $i < j < a$, $e'_{ij} = F_{01}(e_{ij})$ if $i < a \le j$, and $e'_{ij} = s(F_1(x_a))$ if $a \le i < j$. That is, the $F'_{01}[n]$ images are sequences of a number of F_0 images followed by b number of F_1 images, with all edges being degenerate, except the bridges between the two nodes. Since each of the derived F_i functions are constant on each connected component, and the input consists exactly of sequences of nodes in the same connected component, all of the elements $F_0(x_0), \dots, F_0(x_{a-1})$ are the same element in $Y[0]$, and likewise for $F_1(x_a) \dots, F_1(x_{a+b-1})$. This justifies our choice of e'_{ij} as the degenerate edges.

It should be clear that this map does indeed commute with d_i and s_j, completing the proof. ◀

▶ **Lemma 14** (classical)**.** *For all Kan graphs Y and X, if $F_{01} : X[1] \to Y[1]$ is Y^X-good, then there is an $F_{10} : X[1] \to Y[1]$ such that $d_0 F_{01} = d_1 F_{10}$ and $d_1 F_{01} = d_0 F_{10}$.*

Proof. Let X and Y be Kan graphs. The $S(Y)$ and $S(X)$ are Kan simplicial sets by Lemma 9. By applying the classical Theorem 2 we get that $S(Y)^{S(X)}$ has edge reversal. Since F_{01} is Y^X-good we extend F_{01} to an edge $F'_{01} \in S(Y)^{S(X)}[1]$ as defined in the proof of Lemma 13. By edge reversal in $S(Y)^{S(X)}$ we get an $F'_{10} \in S(Y)^{S(X)}[1]$ satisfying $d_1(F'_{10}) = d_0(F'_{01})$ and $d_0(F'_{10}) = d_1(F'_{01})$. We put $F_{10}(x) = F'_{10}(01, x)$. By expanding the definition of d_k from Section 3.4, we get the following properties: $F'_{10}(00, e) = F'_{01}(11, e)$ and $F'_{10}(11, e) = F'_{01}(00, e)$, giving $F'_{10}(0, d_i(e)) = F'_{01}(1, d_i(e))$ and $F'_{10}(1, d_i(e)) = F'_{01}(0, d_i(e))$. We calculate $d_0 F_{01}(e) = d_0 F'_{01}(01, e) = F'_{01}(1, d_0(e)) = F_1 d_0(e)$. Since F_{01} is Y^X-good (2nd requirement) we have $F_1 d_0(e) = F_1 d_1(e)$. We continue the calculation: $F_1 d_1(e) = F'_{01}(1, d_1(e)) = F'_{10}(0, d_1(e))$ where the last step is justified above. We continue: $F'_{10}(0, d_1(e)) = d_1 F'_{10}(01, e) = d_1 F_{10}(e)$. In total we have proved $d_0 F_{01}(e) = d_1 F_{10}(e)$ for all $e \in X[1]$. Hence $d_0 F_{01} = d_1 F_{10}$. The other equation is proved symmetrically. ◀

Table 1 Kripke (counter)model for edge reversal.

	Day 1
X_0	$\{x, x'\}$
X_1	$\{s(x), s(x')\}$
Y_0	$\{y_0, y_1, y'_0, y'_1\}$
Y_1	$\{s(y_0), s(y_1), s(y'_0), s(y'_1), y_0y_1 : y_0 \to y_1, y'_0y'_1 : y'_0 \to y'_1, a : y_1 \to y_0, b : y'_1 \to y'_0\}$

	Day 2
X_0	$\{x{=}x'\}$
X_1	$\{s(x){=}s(x')\}$
Y_0	$\{y_0 = y'_0, y_1 = y'_1\}$
Y_1	$\{s(y_0) = s(y'_0), s(y_1) = s(y'_1), y_0y_1 = y'_0y'_1, a, b\}$

	Input	Output
F_0	x	y_0
F_0	x'	y'_0
F_1	x	y_1
F_1	x'	y'_1
F_{01}	$s(x)$	y_0y_1
F_{01}	$s(x')$	$y'_0y'_1$

Kripke [8] showed that constructive logic is sound for Kripke models, so the existence of a Kripke countermodel of a statement gives the non-existence of a constructive proof of that statement. We will now, by the means of a Kripke model, see that Lemma 14 does not hold constructively.

4.2 Edge reversal, the Kripke countermodel

We describe a Kripke model containing a Y^X-good F_{01} such that there cannot be a function $F_{10} : X[1] \to Y[1]$ with $d_0F_{01} = d_1F_{10}$ and $d_1F_{01} = d_0F_{10}$, even though X and Y are Kan graphs.

For clarity, the functions $F_0 = d_1F_{01}s$ and $F_1 = d_0F_{01}s$ as defined in Definition 12 are also made explicit in this model. Face maps are part of the model, but not made explicit.

The model consists of two days, with an X and a Y part each. On day 1 both X and Y consist of two separate components, which get merged on day 2. We give the model both in Table 1 and, graphically, in Figure 1 and 2.

It is easy to see that both X and Y are Kan graphs by simply observing that each of their two components are strongly connected. It is also clear that we cannot define a consistent F_{10}. In day 1 we would have to set $F_{10}(s(x)) = a$ and $F_{10}(s(x')) = b$ to satisfy the requirement that $d_0F_{01} = d_1F_{10}$ and $d_1F_{01} = d_0F_{10}$. The problem occurs in day 2, where we have that $s(x) = s(x')$, but $a \neq b$, making it impossible for F_{10} to respect equality. Note that all other functions, F_0, F_1, F_{01}, s, d_0, and d_1 remain consistent after collapsing, that is, they still map equal elements to equal elements.

5 Edge composition

In this section we give the classical proof of Theorem 3 and show that there is no constructive proof.

▶ **Definition 15** (Edge composition). A simplicial set Y is said to have *edge composition* when for every edge $e_1, e_2 \in Y[1]$, if $d_0(e_1) = d_1(e_2)$ then there exists an edge $f \in Y[1]$ with $d_1(f) = d_1(e_1)$ and $d_0(f) = d_0(e_2)$.

▶ **Lemma 16.** *Kan simplicial sets have edge composition.*

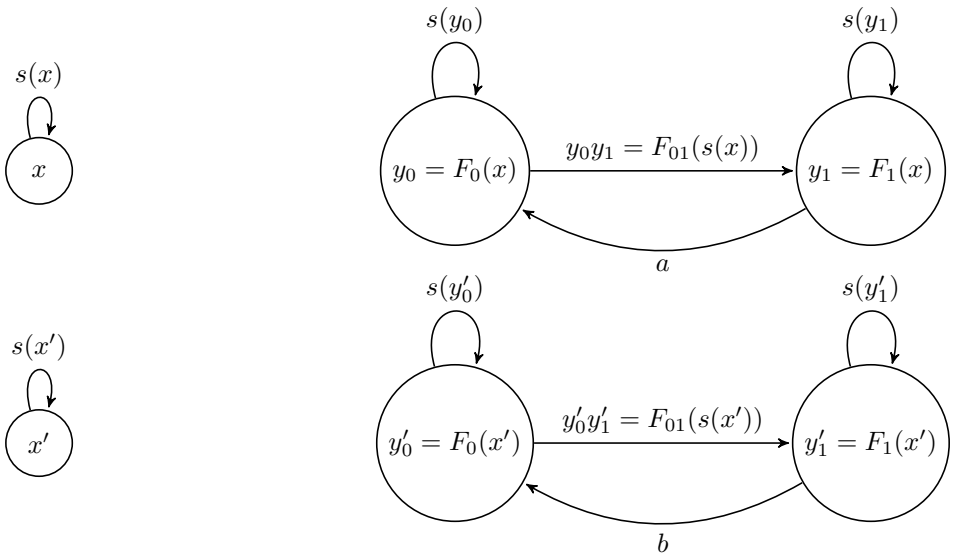


Figure 1 Kripke (counter)model for edge reversal, day 1.

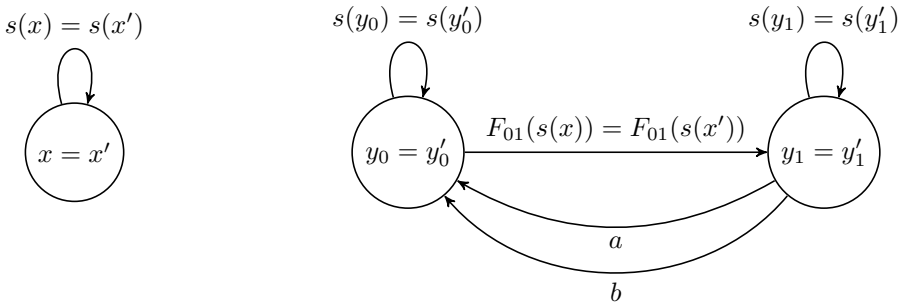


Figure 2 Kripke (counter)model for edge reversal, day 2.

Proof. Given an arbitrary Kan simplicial set Y and edges $e_1, e_2 \in Y[1]$ with $d_0(e_1) = d_1(e_2)$, we can make a map $G : \Lambda^2_1 \to Y$ by putting $G(0) = d_1(e_1)$, $G(1) = d_0(e_1)$, $G(2) = d_0(e_2)$ $G(01) = e_1$ and $G(12) = e_2$. Since Y is Kan we can extend G to $G : \Delta^2 \to Y$, giving us a simplex $G(02) : G(0) \to G(2)$ in $Y[1]$, the composition of e_1 and e_2. ◀

By a proof essentially identical to the proof of Lemma 14 we get the following lemma.

▶ **Lemma 17** (classical). *For all Kan graphs Y and X, if $F_{01} : X[1] \to Y[1]$ and $F_{12} : X[1] \to Y[1]$ are Y^X-good maps satisfying $d_0F_{01} = d_1F_{12}$, then there is an $F_{02} : X[1] \to Y[1]$ such that $d_0F_{01} = d_0F_{02}$ and $d_1F_{12} = d_1F_{02}$.*

In Figure 3 and 4 we see that Lemma 17 is not constructively provable. We have two Y^X-good functions F_{01} and F_{12}, satisfying the requirement, and both X and Y are Kan graphs. If $S(Y)^{S(X)}$ had edge composition we would get a function F_{02} that $d_1F_{01} = d_1F_{02}$ and $d_0F_{12} = d_0F_{02}$. However, such a function is not definable in the Kripke model. The reason is analogous to the case of edge-reversal: from day 1 to day 2 we have equated objects in the domain of F_{02} while keeping the images distinct. Specifically, on day 1 we are forced to set $F_{02}(s(x)) = a$ and $F_{02}(s(x')) = b$, but on day 2 we have $s(x) = s(x')$, but $a \neq b$.

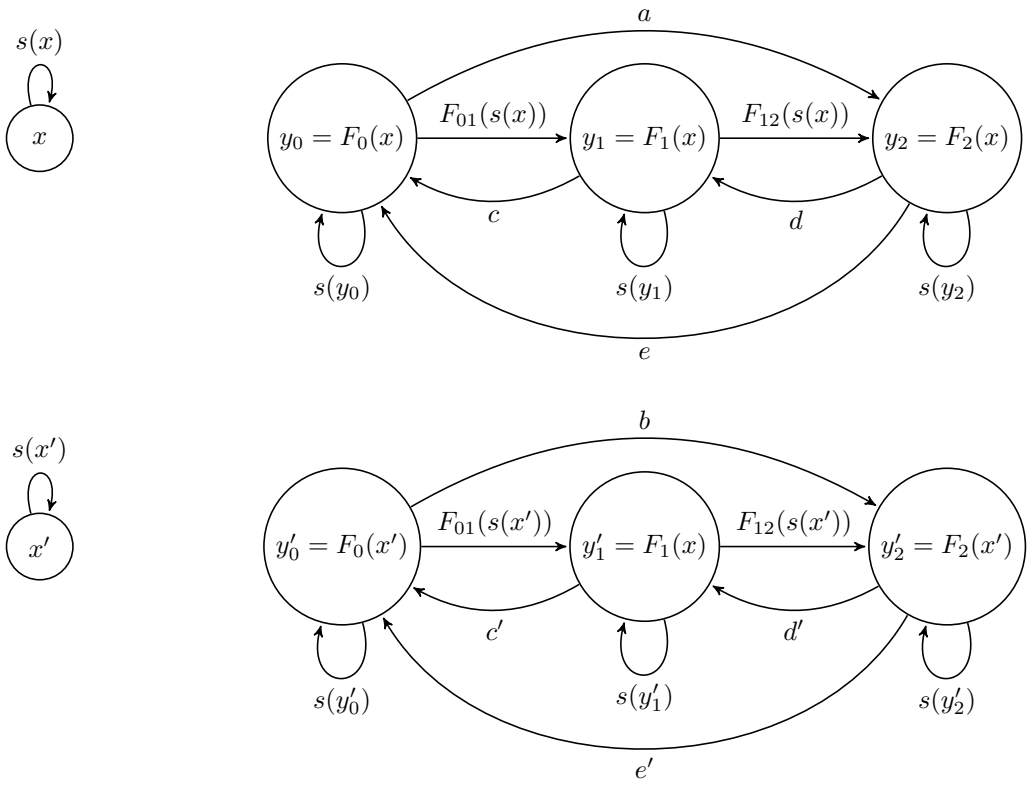


Figure 3 Kripke (counter)model for edge composition, day 1.

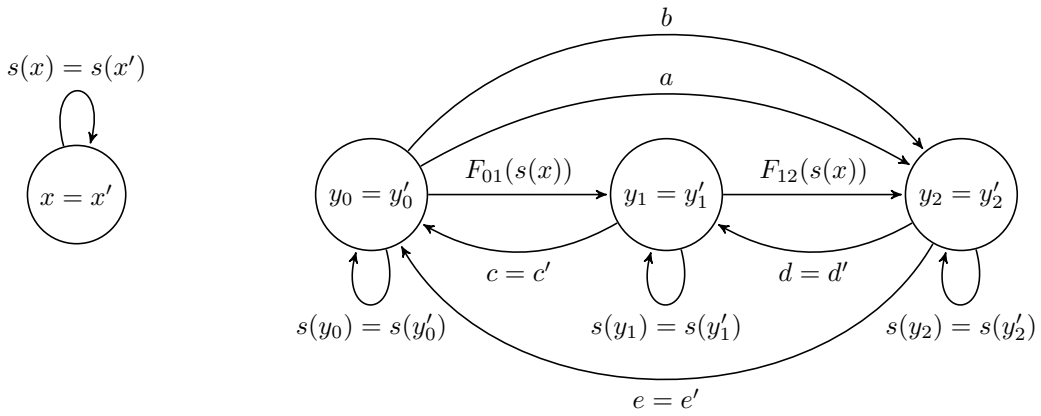


Figure 4 Kripke (counter)model for edge composition, day 2.

6 Evaluation of the results

The results up to now are summarized in Figure 5.

Having concrete, finite Kripke countermodels against Lemma 14 and 17 allows for a further simplification: everything remains valid under the condition that X has at most two points. Likewise, explicit bounds read off from the Kripke models can be imposed on the number of points of Y and on the number of edges in X and in Y. The simplified results are denoted by postfixing the number of the result by a 'b' for bounded, so Lemma 14b is the bounded version of Lemma 14.

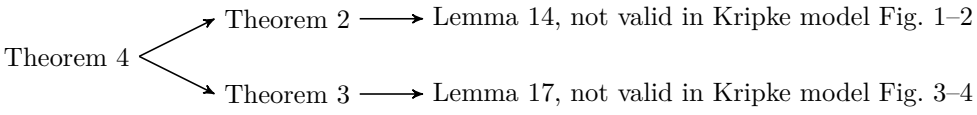


Figure 5 Summary of results, all implications constructive.

With explicit bounds on the size of the domain, functions are completely determined by a finite number of function values. For example, if we have $\forall z \in X.\ (z = x \vee z = x')$ for $x, x' \in X$, then the binary predicate $\mathrm{fun}(y, y') \equiv (x = x' \to y = y')$ on Y completely describes all functions $X \to Y$, in evidence $x \mapsto y,\ x' \mapsto y'$. With this in mind it is not difficult to express Lemma 14b as a first-order classical tautology Φ that is not true in all Kripke models.

Now fix a constructive framework that is sufficiently expressive for the results in Figure 5. For example, IZF (Zermelo-Fraenkel set theory in IPL, intuitionistic predicate logic) will do. Let $[|\Phi|]$ be the Tarski interpretation of Φ expressed in IZF. The following fundamental property of IZF could be called the semantic conservativity of IZF over IPL:

If $[|\Phi|]$ is provable in IZF, then Φ is true in all Kripke models.

Lubarsky [9] and McCarty [11] independently provided constructive proofs of the above conservativity property of IZF. We gratefully acknowledge their prompt answers to our question.[1]

Empowered by the proofs of Lubarsky and McCarty we can now conclude that Lemma 14b cannot be proved in IZF. The same is true for Lemma 17b, and for all other results in Figure 5, as well as for their bounded versions.

7 Kan graphs with explicit filler functions

Let us first give an intuitive explanation of our countermodels. They actually exploit the undecidability of equality: on day 1 we don't know what will be equal on day 2. (This is different from the decidability of degeneracy, but the two are related: for example, an edge e is degenerate iff $e = s_0(d_1(e))$.) In Figure 1 and 2, the point is that $y_0 \neq y_0'$ on day 1, so one cannot put $F_{10}(s(x)) = F_{10}(s(x')) = a$ since this conflicts with $d_0 F_{10} = d_1 F_{01}$. One is thus forced to a choice that turns out to be wrong on day 2.

One attempt to deal with this lack of information is to give Kan simplicial sets more structure. One could for example change Definition 6 of a Kan simplicial set into one where we not only know that the required n-simplex exists, but actually have *functions* producing them. In the formulation using horns as in Section 3.2 this would amount to a dependent function $\mathrm{fill}(k, j, F)$ such that $\mathrm{fill}(k, j, F) : \Delta^k \to Y$ extends $F : \Lambda^k_j \to Y$, for any k, j, F. This form of Kan simplicial set has been introduced by Nikolaus in [13] under the name of *algebraic Kan complex*. The definition with explicit fill-functions has certain advantages, both classically and constructively, as we will see below. However, one should be careful in defining Y^X: morphisms in the category of algebraic Kan complexes are required to map chosen fillers in X to chosen fillers in Y. As a consequence, there are less maps from X to Y

[1] Strengthening the semantic conservativity to syntactic conservativity, that is, concluding that Φ is provable in intuitionistic predicate logic, by using the completeness of the Kripke semantics implicates some classical logic. Although not needed for this paper, we think there is some general interest in a constructive proof that IPL $\vdash \Psi$ whenever IZF $\vdash [|\Psi|]$, for any first-order sentence Ψ.

as algebraic Kan complexes than as just simplicial sets. What we propose could be called a *functional* Kan simplicial set, with explicit fill-functions but with maps as for ordinary simplicial sets. As a consequence the exponential Y^X of simplicial sets can be used.

To be able to prove an analogue of Lemma 9 we have to strengthen the notion of Kan graph to also include such *filler functions*, cf. [2].

▶ **Definition 18** (Kan fill-graph)**.** A *Kan fill-graph* is a reflexive multigraph with a partial function $\mathrm{fill} : Y[1] \times Y[1] \to Y[1]$ such that for all $e_1, e_2 \in Y[1]$, if $e_1 : a \to b$ and $e_2 : a \to c$, then $\mathrm{fill}(e_1, e_2) : b \to c$.

As noted earlier, the Kan property together with reflexivity implies symmetry and transitivity. We can now define the corresponding functions.

▶ **Definition 19** (Edge reversal)**.** For all $e \in Y[1]$ where Y is a Kan fill-graph let

$$e^{-1} = \mathrm{fill}(e, sd_1(e)).$$

If $e : a \to b$, then $sd_1(e) : a \to a$, and $\mathrm{fill}(e, sd_1(e)) : b \to a$.

Note that we in general don't have $(e^{-1})^{-1} = e$, but we do have that $d_i((e^{-1})^{-1}) = d_i(e)$.

▶ **Definition 20** (Edge composition)**.** Using the inverse for edges in Y we define the composition of two edges $e_1 : a \to b$ and $e_2 : b \to c$ as

$$\mathrm{trans}(e_1, e_2) = \mathrm{fill}(e_1^{-1}, e_2).$$

Again we are in no way guaranteed that $\mathrm{trans}(e_1, s(b)) = e_1$ or $\mathrm{trans}(s(x), s(x)) = s(x)$.

We immediately see that the addition of explicit functions adds power, as we can now prove constructively and trivially an analogue of Lemma 14.

▶ **Lemma 21.** *For all Kan fill-graphs Y, X and for every $F : X[1] \to Y[1]$, the function $F^{-1} : X[1] \to Y[1]$ defined by $F^{-1}(e) = F(e)^{-1}$ satisfies $d_0F = d_1F^{-1}$ and $d_1F = d_0F^{-1}$.*

Note how using explicit functions rules out the Kripke counter-example we gave of Lemma 14. If $s(x) = s(x')$ on day 2, then we immediately get $a = F_{01}^{-1}(s(x)) = F_{01}^{-1}(s(x')) = b$ since equality has to be preserved.

We can even use the above fact to show that:

▶ **Lemma 22.** *For any reflexive multigraph X and Kan fill-graph Y, $S(Y)^{S(X)}$ has edge reversal.*

Proof. Assume an edge $F \in S(Y)^{S(X)}[1]$, we proceed to define F^{-1} such that $d_0(F) = d_1(F^{-1})$ and $d_1(F) = d_0(F^{-1})$. As $F \in S(Y)^{S(X)}[1]$ we have $F[n] : \Delta^1[n] \times X[n] \to Y[n]$. We start with $n = 0$, defining $F^{-1}[0] : (\Delta^1 \times X)[0] \to Y[0]$ by letting $F^{-1}[0](0, x) = F[0](1, x)$ and $F^{-1}[0](1, x) = F[0](0, x)$. Likewise for $n = 1$ we define $F^{-1}(00, e) = F(11, e)$ and $F^{-1}(11, e) = F(00, e)$, these are directly enforced by $d_0(F) = d_1(F^{-1})$ and $d_1(F) = d_0(F^{-1})$. For the case of $F^{-1}(01, e)$ we need to find an edge $F^{-1}(01, e) : F^{-1}(0, d_1e) \to F^{-1}(1, d_0e)$, which from the way we defined $F^{-1}[0]$ is the same as an edge

$$F^{-1}(01, e) : F(1, d_1e) \to F(0, d_0e).$$

The diagram in Figure 6 shows $e \in S(X)[1]$ with its endpoints on the left, and the nodes and edges we have directly reachable in $S(Y)$ using only F on the right.

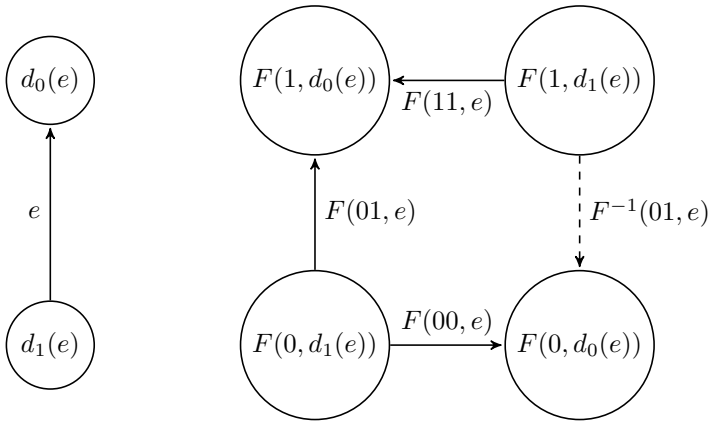


Figure 6 Reversing F.

Reading off the figure we can define $F^{-1}(01, e)$ as follows:

$$F^{-1}(01, e) = \text{trans}(F(11, e)), \text{fill}(F(01, e), F(00, e)))$$

Note that F^{-1} is well-defined since the functions involved in the definition are. Moreover, F^{-1} commutes with s_0, d_0, d_1 by construction.

Having defined F^{-1} for dimension 0 and 1, F^{-1} is also determined in higher dimensions, because of the truncation in $S(X), S(Y)$. In the case of $n > 1$ any input to $F^{-1}[n]$ will have the form

$$F^{-1}(0^a 1^b, (x_0, \dots, x_n; \dots e_{ij}, \dots))$$

where $a + b = n + 1$. We let $F^{-1}[n]$ map this element to the tuple

$$(F^{-1}(0, x_0), \dots, F^{-1}(0, x_{a-1}), F^{-1}(1, x_a) \dots, F^{-1}(1, x_{a+b-1}); \dots e'_{ij}, \dots),$$

where $e'_{ij} = F^{-1}(00, e_{ij})$ if $i < j < a$, $e'_{ij} = F^{-1}(01, e_{ij})$ if $i < a \leq j$, and $e'_{ij} = F^{-1}(11, e_{ij})$ if $a \leq i < j$. This commutes with face and degeneracy maps. ◀

Using the same techniques we can constructively prove the following variant of Lemma 17.

▶ **Lemma 23.** *For any Kan graph X and Kan fill-graph Y, if $F_{01} : X[1] \to Y[1]$ and $F_{12} : X[1] \to Y[1]$ satisfy $d_0 F_{01} = d_1 F_{12}$, then there is a $F_{02} : X[1] \to Y[1]$ such that $d_1 F_{01} = d_1 F_{02}$ and $d_0 F_{12} = d_0 F_{02}$.*

▶ **Lemma 24.** *For any reflexive multigraph X and Kan fill-graph Y, $S(Y)^{S(X)}$ has edge composition.*

Proof. Assume edges $F_{01} \in S(Y)^{S(X)}$, $F_{12} \in S(Y)^{S(X)}$ such that $d_0(F_{01}) = d_1(F_{12})$, and we proceed to define $F_{02} \in S(Y)^{S(X)}$ such that $d_1(F_{02}) = d_1(F_{01})$ and $d_0(F_{02}) = d_0(F_{12})$.

As was the case in the proof of Lemma 22, we are forced on $F_{02}(0, x) = F_{01}(0, x)$, $F_{02}(1, x) = F_{12}(1, x)$, $F_{02}(00, e) = F_{01}(00, e)$, and $F_{02}(11, e) = F_{12}(11, e)$.

For the case of $F_{02}(01, e)$ we need to find an edge $F_{02}(01, e) : F_{02}(0, d_1 e) \to F_{02}(1, d_0 e)$, which from the way we defined $F_{02}[0]$ is the same as an edge

$$F_{02}(01, e) : F_{01}(0, d_1 e) \to F_{12}(1, d_0 e).$$

We note that $d_0(F_{01}) = d_1(F_{12})$ enforces $F_{01}(11, e) = F_{12}(00, e)$, which again enforces $F_{01}(1, d_i(e)) = F_{12}(0, d_i(e))$. This gives the diagram in Figure 7, enabling us to read off:

$$F_{02}(01, e) = \text{fill}(\text{trans}(F_{01}(11, e), F_{01}(01, e)^{-1}), F_{12}(01, e)).$$ ◀

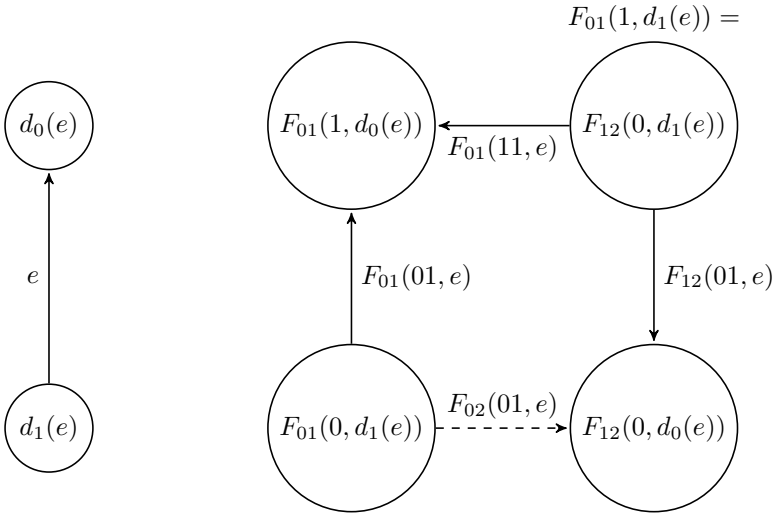


Figure 7 Filling the horn Λ^2_1.

8 Conclusions and Future Research

We have given a thorough analysis of the non-constructivity of the basic result that the Kan extension property is preserved under the usual operation of exponentiation of simplicial sets. An important step in this analysis, also employed in [2], is the truncation of simplicial sets to dimension 1. This allows us to study the basic result in the simplified situation of Kan graphs. Once one has shown the constructive unprovability of the basic result in the situation of Kan graphs, one obtains *a fortiori* its unprovability for Kan simplicial sets.

The much simpler notion of Kan graph (as compared to Kan simplicial set) invites to further thought experiments. One of those is the study of simple, constructive consequences of the Kan extension property, such as edge reversal and edge composition. It turns out that already these consequences cannot be proven constructively.

Another experiment is to strengthen the Kan extension property from existence of an n-simplex as in Definition 6 to having a function, called a *filler*, yielding these n-simplices. This makes quite a difference. None of the Kripke models we have introduced is able to deal with such fillers, since equating objects in X and Y implies that filler-values such as a and b in Figure 1 also have to be equal. The question arises whether this is necessary so, or just coincidental in the particular Kripke model. This question is answered in Section 7, where we prove constructively that, if X is a graph and Y a Kan-fill graph, then $S(Y)^{S(X)}$ has edge reversal and edge composition. This result may be of independent interest. It suggests that showing the (expected) constructive unprovability of Theorem 4 for *algebraic Kan complexes* as in [13] will require more complicated structures than graphs. The above expectation is based on an analysis of filling a 2-horn in Y^X, which requires defining $F(001, t)$. As F has to commute with s_0, one must know whether the 2-simplex t is an s_0-image or not. This can in general only be decided by an appeal to classical logic. We have to leave this to future research.

References

1 Steve Awodey and Michael Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146:45–55, 2009.

2 Marc Bezem and Thierry Coquand. A Kripke model for simplicial sets. *TCS*, 2015. doi:10.1016/j.tcs.2015.01.035.

3 Greg Friedman. An elementary illustrated introduction to simplicial sets. Preprint, `http://arxiv.org/abs/0809.4221`, 2008.

4 Peter Gabriel and Michel Zisman. *Calculus of fractions and homotopy theory*. Springer, 1967.

5 Paul Goerss and J.F. Jardine. *Simplicial Homotopy Theory*. Modern Birkhäuser Classics. Birkhauser Verlag GmbH, 2009. Reprint of Vol. 174 of Progress in Mathematics, 1999.

6 Martin Hofmann. Syntax and semantics of dependent types. In A.M. Pitts and P. Dybjer, editors, *Semantics and logics of computation*, volume 14 of *Publ. Newton Inst.*, pages 79–130. Cambridge University Press, Cambridge, 1997.

7 Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. Preprint, `http://arxiv.org/abs/1211.2851`, 2012.

8 Saul Kripke. Semantical analysis of intuitionistic logic I. In M. Dummett and J.N. Crossley, editors, *Formal Systems and Recursive Functions*, pages 92–130. North–Holland, Amsterdam, 1965.

9 Bob Lubarsky, March 2015. Personal communication.

10 Jon Peter May. *Simplicial Objects in Algebraic Topology*. Chicago Lectures in Mathematics. University of Chicago Press, 2nd edition, 1993.

11 Charles McCarty. Two questions about IZF and intuitionistic validity. Pdf file, personal communication, March 2015.

12 John C. Moore. Algebraic homotopy theory. Lectures at Princeton, `http://faculty.tcu.edu/gfriedman/notes/aht1.pdf`, 1956.

13 Thomas Nikolaus. Algebraic models for higher categories. Preprint, `http://arxiv.org/abs/1003.1342`, 2010.

14 Vladimir Voevodsky. Notes on type systems. `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/expressions_current.pdf`, 2009.

Logical Relations for Coherence of Effect Subtyping

Dariusz Biernacki and Piotr Polesiuk

Institute of Computer Science, University of Wrocław
Joliot-Curie 15, 50-383 Wrocław, Poland
{dabi,ppolesiuk}@cs.uni.wroc.pl

Abstract

A coercion semantics of a programming language with subtyping is typically defined on typing derivations rather than on typing judgments. To avoid semantic ambiguity, such a semantics is expected to be coherent, i.e., independent of the typing derivation for a given typing judgment. In this article we present heterogeneous, biorthogonal, step-indexed logical relations for establishing the coherence of coercion semantics of programming languages with subtyping. To illustrate the effectiveness of the proof method, we develop a proof of coherence of a type-directed, selective CPS translation from a typed call-by-value lambda calculus with delimited continuations and control-effect subtyping. The article is accompanied by a Coq formalization that relies on a novel shallow embedding of a logic for reasoning about step-indexing.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.3 Studies of Program Constructs

Keywords and phrases type system, coherence of subtyping, logical relation, control effect, continuation-passing style

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.107

1 Introduction

Programming languages that allow for subtyping, i.e., a mechanism facilitating coercions of expressions of one type to another, are usually given either a subset semantics, where one type is considered a subset of another type, or – a coercion semantics, where expressions are explicitly converted from one type to another. In the presence of subtyping, typing derivations depend on the occurrences of the subtyping judgments and, therefore, typing judgments do not have unique typing derivations. Consequently, a coercion semantics that interprets subtyping judgment by introducing explicit type coercions is defined on typing derivations rather than on typing judgments. But then a natural question arises as to whether such a semantics is coherent, i.e., whether it does not depend on the typing derivation.

The problem of coherence has been considered in a variety of typed lambda calculi. Reynolds proved the coherence of the denotational semantics for intersection types [22]. Breazu-Tannen et al. proved the coherence of a coercion translation from the lambda calculus with polymorphic, recursive and sum types to system F [8], by showing that any two derivations of the same judgment are normalizable to a unique normal derivation where the correctness of the normalization steps is justified by an equational theory in the target calculus. Curien and Ghelli introduced a translation from system $F_{\leq}$ to a calculus with explicit coercions and showed that any two derivations of the same judgment are translated to terms that are normalizable to a unique normal form [9]. Finally, Schwinghammer followed Breazu-Tannen et al.'s approach to prove the coherence of coercion translation from Moggi's computational lambda calculus with subtyping [24].

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 107–122

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The normalization-based proofs consist in finding a normal form for a representation of the derivation and they hinge on showing that such normal forms are unique for a given typing judgment. When the source calculus under consideration is presented in the spirit of the lambda calculus à la Church, i.e., the lambda abstractions are type annotated, as is the case in all the aforementioned articles that follow the normalization-based approach, the term and the typing context indeed determine the shape of the normal derivation (modulo a top level coercion that depends on the type of the term) [18]. However, in calculi à la Curry this is no longer the case and the method cannot be directly applied. Still, if the calculus is at least weakly normalizing, one can hope to recover the uniqueness property for normal typing derivations for source terms in normal form, assuming that term normalization preserves the coercion semantics. For instance, in the simply typed λ-calculus the typing context uniquely determines the type of the term in the function position in applications building a β-normal form, and, hence, derivations in normal form for such terms are unique. This line of reasoning cannot be used when the calculus includes recursion.

In this article, we consider the coherence problem in simply-typed lambda calculi with general recursion, control effects and with no type annotations. The coercion semantics we study translate typing derivations in the source calculus to a corresponding target calculus with explicit type coercions (that in most cases can be further replaced with equivalent lambda-term representations) and our criterion for coherence of the translation is contextual equivalence [19] in the target calculus.

The main result of this work is a construction of logical relations for establishing so construed coherence of coercion semantics, applicable in a variety of calculi. In particular, we address the problem of coherence of a type-directed CPS translation from the call-by-value λ-calculus with delimited-control operators and control-effect subtyping introduced by Materzok and the first author [16], extended with recursion. While the translation for the calculus with explicit type annotations has been shown to be coherent in terms of an equational theory in a target calculus [15], no CPS coercion translation for the original version, let alone extended with recursion, has been proven coherent.

The reasons why coherence in this calculus is important are twofold. First of all, it is very expressive and therefore interesting from the theoretical point of view. In particular, the calculus has been shown to generalize the canonical type-and-effect system for Danvy and Filinski's shift and reset control operators [10, 11], and, furthermore, that it is strictly more expressive than the CPS hierarchy of Danvy and Filinski [17]. These results heavily rely on the effect subtyping relation that, e.g., allows to coerce pure expressions to effectful ones. From a more practical point of view, the selective CPS translation, that leaves pure (i.e., control-effect free) expressions in direct style and introduces explicit coercions to interpret effect subtyping in the source calculus, is a good candidate for embedding the control operators in an existing programming language, such as Scala [23].

In order to deal with the complexity of the source calculus and of the translation itself, we introduce binary logical relations on terms of the target calculus that are: heterogeneous, biorthogonal [14, 20, 13], and step-indexed [3, 2, 1]. Heterogeneity allows us to relate terms of different types, and in particular those in continuation-passing style with those in direct style. This is a crucial property, since the same term can have a pure type, resulting in a direct-style term through the translation and another, impure type, resulting in a term in continuation-passing style. Relating such terms requires quantification over types and to assure well-foundedness of the construction, we need to use step-indexing, which also supports reasoning about recursion, even if not in a critical way. We follow Dreyer et al. [12] in using logical step-indexed logical relations in our presentation of step-indexing. Biorthogonality, by

imposing a particular order of evaluation on expressions, simplifies the construction of the logical relations. It also facilitates reasoning about continuations represented as evaluation contexts.

Apart from the calculus with effect subtyping, we have used the ideas presented in this article to show coherence of subtyping in several other calculi, including the simply typed lambda calculus with subtyping [18] extended with recursion, the calculus of intersection types [22], and the lambda calculus with subtyping and the control operator call/cc.

The article is accompanied by a Coq development that presently consists of a library that provides a new shallow embedding of the logic for reasoning about step-indexed logical relations, and complete formalization of the proofs presented in the rest of the article. The code is available at `http://www.ii.uni.wroc.pl/~ppolesiuk/lrcoherence`.

The rest of this article is structured as follows. In Section 2, we briefly present Dreyer et al.'s logic for reasoning about step indexing [12] on which we base our presentation. We also describe the main ideas behind our Coq formalization of the logic. In Section 3, we introduce the construction of the logical relations in a simple yet sufficiently interesting scenario – the simply typed lambda calculus à la Curry with natural numbers, type Top, general recursion and standard subtyping. The goal of this section is to introduce the basic ingredients of the proof method before embarking on a considerably more challenging journey in the subsequent section. In Section 4, we present the main result of the article – the logical relations for establishing the coherence of the CPS translation from the calculus of delimited control with effect subtyping. In Section 5, we summarize the article.

2 Reasoning about step-indexed logical relations

Step-indexed logical relations [3, 2, 1] are a powerful tool for reasoning about programming languages. Instead of describing a general behavior of program execution, they focus on the first n computation steps, where the step index n is an additional parameter of the relation. This additional parameter makes it possible to define logical relations inductively not only on the structure of types, but also on the number of computation steps that are allowed for a program to make and, therefore, they provide an elegant way to reason about features that introduce non-termination to the programming language, including recursive types [2] and references [1].

However, reasoning directly about step-indexed logical relations is tedious because proofs become obscured by step-index arithmetic. Dreyer et al. [12] proposed logical step-indexed logical relations (LSLR) to avoid this problem. The LSLR logic is an intuitionistic logic for reasoning about one particular Kripke model: where possible worlds are natural numbers (step-indices) and where future worlds have smaller indices than the present one. All formulas are interpreted as monotone (non-increasing) sequences of truth values, whereas the connectives are interpreted as usual. In particular, in the case of implication we quantify over all future worlds to ensure monotonicity, so the formula $\varphi \Rightarrow \psi$ is valid at index n (written $n \models \varphi \Rightarrow \psi$) iff $k \models \varphi$ implies $k \models \psi$ for every $k \leq n$. In contrast to Dreyer et al. we do not assume that all formulas are valid in world 0, because it is not necessary.

The LSLR logic is also equipped with a modal operator $\triangleright$ (later), to provide access to strictly future worlds. The formula $\triangleright\varphi$ means φ *holds in any future world*, or formally $\triangleright\varphi$ is always valid at world 0, and $n+1 \models \triangleright\varphi$ iff φ is valid at n (and other future worlds by monotonicity). The later operator comes with two inference rules:

$$\frac{\Gamma, \Sigma \vdash \varphi}{\Gamma, \triangleright\Sigma \vdash \triangleright\varphi}\ \triangleright\text{-intro} \qquad \frac{\Gamma, \triangleright\varphi \vdash \varphi}{\Gamma \vdash \varphi}\ \text{Löb}$$

$$\begin{array}{lcll} \tau & ::= & \mathsf{Nat} \mid \mathsf{Top} \mid \tau \to \tau & \text{(types)} \\ e & ::= & x \mid \lambda x.e \mid e\,e \mid \mathsf{fix}\,x(x).e \mid n & \text{(expressions)} \end{array}$$

$$\frac{}{\tau \le \tau}\ \text{S-Refl} \qquad \frac{\tau_2 \le \tau_3 \qquad \tau_1 \le \tau_2}{\tau_1 \le \tau_3}\ \text{S-Trans} \qquad \frac{}{\tau \le \mathsf{Top}}\ \text{S-Top}$$

$$\frac{\tau_2' \le \tau_1' \qquad \tau_1 \le \tau_2}{(\tau_1' \to \tau_1) \le (\tau_2' \to \tau_2)}\ \text{S-Arr} \qquad \frac{(x:\tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \text{T-Var} \qquad \frac{\Gamma, x:\tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \lambda x.e\ :\ \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1\ :\ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2\ :\ \tau_2}{\Gamma \vdash e_1\,e_2\ :\ \tau_1}\ \text{T-App} \qquad \frac{\Gamma, f:\tau_1 \to \tau_2, x:\tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \mathsf{fix}\,f(x).e\ :\ \tau_1 \to \tau_2}\ \text{T-Fix}$$

$$\frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \text{T-Const} \qquad \frac{\Gamma \vdash e\ :\ \tau \qquad \tau \le \tau'}{\Gamma \vdash e\ :\ \tau'}\ \text{T-Sub}$$

Figure 1 The source language – λ-calculus with subtyping.

The first rule allows one to shift reasoning to a future world, making the assumptions about the future world available. The Löb rule expresses an induction principle for indices. Note that the premise of the rule also captures the base case, because the assumption $\triangleright\varphi$ is trivial in the world 0. The later operator comes with no general elimination rule.

Predicates in LSLR logic as well as step-indexed logical relations can be defined inductively on indices. More generally, we can define a recursive predicate $\mu r.\varphi(r)$, provided all occurrences of r in φ are guarded by the later operator, to guarantee well-foundedness of the definition. For the sake of readability, in this paper we define recursive predicates and relations by giving a set of clauses instead of using the μ operator.

Since the logic is developed for reasoning about one particular model, we can freely add new inference rules for the logic if we prove they are valid in the model. We can also add new relations or predicates to the logic if we provide their monotone interpretation. In particular, constant functions are monotone, so we can safely use predicates defined outside of the logic, such as typing or reduction relations.

Our Coq formalization accompanying this article is built on our IxFree library that contains a shallow embedding of the LSLR logic similar to Appel et al.'s formalization of the "very modal model" [4]. Logical connectives including the later operator are functions on a special type `IProp` of "indexed propositions" defined as a type of monotone functions from `nat` to `Prop`. The library provides tactics representing the most important inference rules. One of the main differences between our library and Appel et al.'s formalization is a way of keeping track of the assumptions. Instead of interpreting a sequent $\varphi_1, \ldots, \varphi_n \vdash \psi$ directly, we treat it as $k \models \psi$ with the standard Coq assumptions $k \models \varphi_1$, …, $k \models \varphi_n$. This approach is very convenient since it allows for reusing a number of existing Coq tactics.

3 Introducing the logical relations

In this section we prove the coherence of subtyping in the simply-typed call-by-value lambda calculus extended with recursion, where the coercion semantics is given by a standard translation to the simply-typed lambda calculus with explicit coercions [9]. Our goal here is to introduce the proof method in a simple scenario, so that in Section 4 we can focus on issues specific to control effects. The logical relations we present in this section are biorthogonal and step-indexed, which is not strictly necessary but it makes the development more elegant. Furthermore, biorthogonality and step-indexing become crucial in handling more complicated calculi such as the one of Section 4 and, therefore, are essential for the method to scale.

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{Nat} \mid \mathsf{Unit} \mid \tau \to \tau & \text{(types)} \\
c & ::= & \mathsf{id} \mid c \circ c \mid \mathsf{top} \mid c \to c & \text{(coercions)} \\
e & ::= & x \mid \lambda x.e \mid e\ e \mid c\ e \mid \mathsf{fix}\, x(x).e \mid n \mid \langle\rangle & \text{(expressions)} \\
v & ::= & x \mid \lambda x.e \mid \mathsf{fix}\, x(x).e \mid (c \to c)\ v \mid n \mid \langle\rangle & \text{(values)} \\
E & ::= & \Box \mid E\ e \mid v\ E \mid c\ E & \text{(evaluation contexts)}
\end{array}
$$

$$
\frac{}{\mathsf{id} :: \tau \rhd \tau}\ \textsc{S-Refl} \qquad \frac{c_1 :: \tau_2 \rhd \tau_3 \qquad c_2 :: \tau_1 \rhd \tau_2}{c_1 \circ c_2 :: \tau_1 \rhd \tau_3}\ \textsc{S-Trans} \qquad \frac{}{\mathsf{top} :: \tau \rhd \mathsf{Unit}}\ \textsc{S-Top}
$$

$$
\frac{c_1 :: \tau_2' \rhd \tau_1' \qquad c_2 :: \tau_1 \rhd \tau_2}{c_1 \to c_2 :: (\tau_1' \to \tau_1) \rhd (\tau_2' \to \tau_2)}\ \textsc{S-Arr} \qquad \frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \textsc{T-Const} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \textsc{T-Var}
$$

$$
\frac{}{\Gamma \vdash \langle\rangle\ :\ \mathsf{Unit}}\ \textsc{T-Unit} \qquad \frac{\Gamma, x : \tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \lambda x.e\ :\ \tau_1 \to \tau_2}\ \textsc{T-Abs} \qquad \frac{c :: \tau \rhd \tau' \qquad \Gamma \vdash e\ :\ \tau}{\Gamma \vdash c\ e\ :\ \tau'}\ \textsc{T-CApp}
$$

$$
\frac{\Gamma \vdash e_1\ :\ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2\ :\ \tau_2}{\Gamma \vdash e_1\ e_2\ :\ \tau_1}\ \textsc{T-App} \qquad \frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \mathsf{fix}\, f(x).e\ :\ \tau_1 \to \tau_2}\ \textsc{T-Fix}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
E[(\lambda x.e)\ v] & \to_\beta & E[e\{v/x\}] & E[\mathsf{id}\ v] & \to_\iota & E[v] \\
E[(\mathsf{fix}\, f(x).e)\ v] & \to_\beta & E[e\{\mathsf{fix}\, f(x).e/f, v/x\}] & E[(c_1 \circ c_2)\ v] & \to_\iota & E[c_1\ (c_2\ v)] \\
 & & & E[\mathsf{top}\ v] & \to_\iota & E[\langle\rangle] \\
 & & & E[(c_1 \to c_2)\ v_1\ v_2] & \to_\iota & E[c_2\ (v_1\ (c_1\ v_2))]
\end{array}
$$

Figure 2 The target language – λ-calculus with explicit coercions.

3.1 The simply-typed lambda calculus with subtyping

The syntax and typing rules for the source language are given in Figure 1. The language is the simply-typed lambda calculus with recursive functions ($\mathsf{fix}\, f(x).e$) and natural numbers (n). For brevity we do not consider any primitive operations on natural numbers or other basic types, but they could seamlessly be added to the language. We include the type Top, to make the subtyping relation interesting. The typing and subtyping rules are standard [18].

3.2 Coercion semantics

The semantics of the source language is given by a translation of the typing derivations to a target language that extends the source language with explicit type coercions (and replaces Top with Unit). The coercions express conversion of a term from one type to another, according to the subtyping relation. Figure 2 contains syntax, typing rules and reduction rules of the target language. The type coercions c and their typing rules correspond exactly to the subtyping rules of the source language. The grammar of terms contains explicit coercion application of the form $c\ e$ and it is worth noting that terms of the form $(c \to c)\ v$ are considered values, since they represent a coercion expecting another value as argument (witness the last reduction rule).

The operational semantics of the target language distinguishes between β-rules that perform actual computations and ι-rules that rearrange coercions. Both of them are used during program evaluation. We say that program e terminates (written $e{\downarrow}$) when it can be reduced to a value using both sorts of reduction rules, according to the evaluation strategy determined by the evaluation contexts.

General contexts are closed terms with one hole (possibly under some binders), and are denoted by the metavariable C. We write $\vdash C\ :\ (\Gamma; \tau_1)\ \rightsquigarrow\ \tau_2$ if for any e with $\Gamma \vdash e\ :\ \tau_1$ we have $\Gamma \vdash C[e]\ :\ \tau_2$. Contextual approximation, written $\Gamma \vdash e_1\ \precsim_{ctx}\ e_2\ :\ \tau$, means that

$$[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!] \qquad [\![\mathsf{Nat}]\!] = \mathsf{Nat} \qquad [\![\mathsf{Top}]\!] = \mathsf{Unit}$$

$$\begin{aligned}
\mathcal{S}[\![\tau \le \tau]\!]_{\text{S-Refl}} &= \mathsf{id} \\
\mathcal{S}[\![\mathsf{Top} \le \tau]\!]_{\text{S-Top}} &= \mathsf{top} \\
\mathcal{S}[\![\tau_1 \le \tau_3]\!]_{\text{S-Trans}(D_1,D_2)} &= \mathcal{S}[\![\tau_2 \le \tau_3]\!]_{D_1} \circ \mathcal{S}[\![\tau_1 \le \tau_2]\!]_{D_2} \\
\mathcal{S}[\![\tau_1' \to \tau_1 \le \tau_2' \to \tau_2]\!]_{\text{S-Arr}(D_1,D_2)} &= \mathcal{S}[\![\tau_2' \le \tau_1']\!]_{D_1} \to \mathcal{S}[\![\tau_1 \le \tau_2]\!]_{D_2}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\![x]\!]_{\text{T-Var}} &= x & \mathcal{T}[\![\mathsf{fix}\, f(x).e]\!]_{\text{T-Fix}(D)} &= \mathsf{fix}\, f(x).\mathcal{T}[\![e]\!]_D \\
\mathcal{T}[\![\lambda x.e]\!]_{\text{T-Abs}(D)} &= \lambda x.\mathcal{T}[\![e]\!]_D & \mathcal{T}[\![e]\!]_{\text{T-Sub}(D_1,D_2)} &= \mathcal{S}[\![\tau \le \tau']\!]_{D_2}\, \mathcal{T}[\![e]\!]_{D_1} \\
\mathcal{T}[\![e_1\ e_2]\!]_{\text{T-App}(D_1,D_2)} &= \mathcal{T}[\![e_1]\!]_{D_1}\, \mathcal{T}[\![e_2]\!]_{D_2} & \mathcal{T}[\![n]\!]_{\text{T-Const}} &= n
\end{aligned}$$

Figure 3 Coercion semantics for the λ-calculus with subtyping.

for any context C and type τ', such that $\vdash C : (\Gamma;\tau) \rightsquigarrow \tau'$ if $C[e_1]$ terminates, then so does $C[e_2]$. If $\Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$ and $\Gamma \vdash e_2 \precsim_{ctx} e_1 : \tau$, then we say that e_1 and e_2 are contextually equivalent.

The coercion semantics of the source language is given in Figure 3. The function $\mathcal{S}[\![.]\!]$ translates subtyping proofs into coercions, and function $\mathcal{T}[\![.]\!]$ translates typing derivations into terms of the target language.

▶ **Lemma 1.** *Coercion semantics preserves types.*
1. *If* $D :: \tau_1 \le \tau_2$ *then* $\mathcal{S}[\![\tau_1 \le \tau_2]\!]_D :: [\![\tau_1]\!] \triangleright [\![\tau_2]\!]$.
2. *If* $D :: \Gamma \vdash e : \tau$ *then* $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_D : [\![\tau]\!]$.

3.3 Logical relations

In order to reason about contextual equivalence in the target language, we define logical relations (Figure 4). Relations are expressed in the LSLR logic described in Section 2, so they are implicitly step-indexed.

We call these relations heterogeneous because they are parameterized by two types, one for each of the arguments. This property is important for our coherence proof, since it makes it possible to relate the results of the translation of two typing derivations which assign different types to the same term. When both types τ_1 and τ_2 are Nat or both are arrow types, the value relation $\mathcal{V}[\![\tau_1;\tau_2]\!]$ is standard. Two values are related for type Nat if they are the same constant, and two functions are related when they map related arguments to related results. The most interesting are the cases when type parameters of the relation are different. When one of these types is Unit, then any values are in the relation, because we do not expect them to carry any information – Unit is the result of translating the Top type. Functions are never related with natural numbers.

The relation $\mathcal{E}[\![\tau_1;\tau_2]\!]$ for closed terms is defined by biorthogonality. Two terms are related if they behave the same in related contexts, and contexts are related (relation $\mathcal{K}[\![\tau_1;\tau_2]\!]$) if they yield the same observations when plugged with related values. Yielding the same observations (relation $\precsim$) is defined for each step-index separately: $e_1 \precsim e_2$ is valid at k iff termination of e_1 using at most k β-steps (and any number of ι-steps) implies termination of e_2 in any number of steps. This interpretation is monotone, so the relation $\precsim$ can be added to the LSLR logic. The relation $\mathcal{E}[\![\tau_1;\tau_2]\!]$ is extended to open terms as usual: two open

$$
\begin{array}{rcl}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Nat}; \mathsf{Nat}]\!] & \iff & \exists n, v_1 = v_2 = n \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1' \to \tau_1; \tau_2' \to \tau_2]\!] & \iff & \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1'; \tau_2']\!].(v_1\ a_1, v_2\ a_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] & \iff & \top \qquad \text{if } \tau_1 = \mathsf{Unit} \text{ or } \tau_2 = \mathsf{Unit} \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] & \iff & \bot \qquad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] & \iff & \forall (E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!].E_1[e_1] \precsim E_2[e_2] \\
(E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!] & \iff & \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].E_1[v_1] \precsim E_2[v_2] \\
k \models e_1 \precsim e_2 & \iff & e_1{\downarrow^k} \implies e_2{\downarrow} \\
(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!] & \iff & \forall x, (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\Gamma_1(x); \Gamma_2(x)]\!] \\
\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2 & \iff & \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!].(e_1\gamma_1, e_2\gamma_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!]
\end{array}
$$

Figure 4 Logical relations for the λ-calculus with explicit coercions.

terms are related (written $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2$) when every pair of related closing substitutions makes them related.

Notice that we do not assume that related terms have valid types. Our relations may include some "garbage", e.g., $(1, \lambda x.x) \in \mathcal{V}[\![\mathsf{Unit}; \mathsf{Nat}]\!]$, but it is non-problematic. One can mechanically prune these relations to well-typed terms, but this change complicates formalization and we did not find it useful.

In this presentation we consider languages with only one base type. Adding more base types and some subtyping between them will not change the general shape of the proof, but defining logical relations for such a case is a little trickier. We would stipulate that two values v_1 and v_2 are related for base types b_1 and b_2 iff for every common supertype b of b_1 and b_2, coercing v_1 and v_2 to b yields the same constant.

The relation $\precsim$ is preserved by reductions in the following sense, where the third assertion expresses an elimination rule of the later modality that is crucial in the subsequent proofs.

▶ **Lemma 2.** *The following assertions hold:*

1. *If* $e_1 \to_\iota e_1'$ *and* $e_1' \precsim e_2$ *then* $e_1 \precsim e_2$.
2. *If* $e_2 \to_\iota e_2'$ *and* $e_1 \precsim e_2'$ *then* $e_1 \precsim e_2$.
3. *If* $e_1 \to_\beta e_1'$ *and* $\triangleright e_1' \precsim e_2$ *then* $e_1 \precsim e_2$.
4. *If* $e_2 \to_\beta e_2'$ *and* $e_1 \precsim e_2'$ *then* $e_1 \precsim e_2$.

The proof of soundness of the logical relations follows closely the standard technique for biorthogonal logical relations [20, 13]. First, we need to show compatibility lemmas, which state that the relation is preserved by every language construct. Most of them are standard and we omit them due to lack of space. The only compatibility lemma specific to our relations is the following lemma for coercion application.

▶ **Lemma 3** (Coercion compatibility). *The logical relation is preserved by coercion application.*

1. *If* $c :: \tau_1 \triangleright \tau_2$ *and* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_0$ *then* $\Gamma_1; \Gamma_2 \vdash c\, e_1 \precsim_{log} e_2 \ : \ \tau_2; \tau_0$.
2. *If* $c :: \tau_1 \triangleright \tau_2$ *and* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_0; \tau_1$ *then* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} c\, e_2 \ : \ \tau_0; \tau_2$.

Proof. We prove both cases by induction on the typing derivation of the coercion c. ◀

▶ **Theorem 4** (Fundamental property). *If* $\Gamma \vdash e \ : \ \tau$ *then* $\Gamma; \Gamma \vdash e \precsim_{log} e \ : \ \tau; \tau$.

Proof. By induction on the derivation $\Gamma \vdash e \ : \ \tau$. In each case we apply the corresponding compatibility lemma. ◀

▶ **Lemma 5** (Precongruence). *If* $\vdash C \ : \ (\Gamma;\tau) \ \rightsquigarrow \ \tau_0$ *and* $\Gamma;\Gamma \vdash e_1 \precsim_{log} e_2 \ : \ \tau;\tau$ *then* $(C[e_1], C[e_2]) \in \mathcal{E}[\![\tau_0;\tau_0]\!]$.

Proof. By induction on the derivation of context typing, using the appropriate compatibility lemma in each case. For contexts containing subterms we also need the fundamental property. For the empty context we use the empty substitution, since the empty substitutions are in relation $\mathcal{G}[\![\varnothing;\varnothing]\!]$. ◀

▶ **Lemma 6** (Adequacy). *If* $(e_1, e_2) \in \mathcal{E}[\![\tau;\tau]\!]$ *then* $e_1 \precsim e_2$.

Proof. Let us show $\Box[e_1] \precsim \Box[e_2]$. Using the assertion $(e_1, e_2) \in \mathcal{E}[\![\tau;\tau]\!]$, it suffices to show $(\Box, \Box) \in \mathcal{K}[\![\tau;\tau]\!]$, which is trivial, since values always terminate. ◀

▶ **Theorem 7** (Soundness). *If* $k \models \Gamma;\Gamma \vdash e_1 \precsim_{log} e_2 \ : \ \tau;\tau$ *holds for every* k, *then* $\Gamma \vdash e_1 \precsim_{ctx} e_2 \ : \ \tau$.

Proof. Suppose $\vdash C \ : \ (\Gamma;\tau) \ \rightsquigarrow \ \tau_0$ and $C[e_1]{\downarrow}$, we need to show $C[e_2]{\downarrow}$. By Lemma 5 and Lemma 6 we know $k \models C[e_1] \precsim C[e_2]$ for every k. Taking k to be the number of steps in which $C[e_1]$ terminates, we have that $C[e_2]$ also terminates, by the definition of $\precsim$. ◀

3.4 Coherence of the coercion semantics

Having established soundness of the logical relations, we are in a position to prove the main coherence lemma, phrased in terms of the logical relations, and the coherence theorem.

▶ **Lemma 8.** *If* $D_i :: \Gamma_i \vdash e \ : \ \tau_i$ *for* $i = 1, 2$ *are two typing judgments for the same term* e *of the source language, then* $[\![\Gamma_1]\!];[\![\Gamma_2]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{log} \mathcal{T}[\![e]\!]_{D_2} \ : \ [\![\tau_1]\!];[\![\tau_2]\!]$.

Proof. The proof follows by induction on the structure of both derivations D_1 and D_2. At least one of these derivations is decreased in every case. When one of derivations starts with subsumption rule (T-Sub), we apply Lemma 3. The coercion that we get after translation is well-typed by Lemma 1. In other cases we just apply the appropriate compatibility lemma. ◀

▶ **Theorem 9** (Coherence). *If* D_1 *and* D_2 *are derivations of the same typing judgment* $\Gamma \vdash e \ : \ \tau$, *then* $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{ctx} \mathcal{T}[\![e]\!]_{D_2} \ : \ [\![\tau]\!]$.

Proof. Immediately from Lemma 8 and Theorem 7. ◀

Coercion semantics described here translates the source language into the language with explicit coercions. We chose coercions to be a separate syntactic category, because we found it very convenient, especially for proving Lemma 3. However, one can define a coercion semantics which translates subtyping proofs directly to λ-expressions. Our result can be easily extended for such a translation. Let $|e|$ be a term e with all the coercions replaced by the corresponding expressions. To prove that for any contextually equivalent terms e_1 and e_2 in the language with coercions, terms $|e_1|$ and $|e_2|$ are contextually equivalent in the language without coercions, we need three simple facts that can be easily verified:

1. every well-typed term in the language without coercions is well typed in the language with coercions,
2. term e terminates iff $|e|$ terminates,
3. if context C does not contain coercions then $C[|e|] = |C[e]|$.

In the next section we show that the results presented in this section can be adapted to a considerably more complex calculus – a calculus of delimited control with control-effect subtyping.

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{Nat} \mid \tau \to T & \text{(pure types)} \\
T & ::= & \tau \mid \tau[T]T & \text{(types)} \\
e & ::= & x \mid \lambda x.e \mid e\ e \mid \mathsf{fix}\, x(x).e \mid \mathcal{S}_0 x.e \mid \langle e \rangle \mid n & \text{(expressions)}
\end{array}
$$

$$\frac{}{T \le T}\ \textsc{S-Refl} \qquad \frac{T_2 \le T_3 \qquad T_1 \le T_2}{T_1 \le T_3}\ \textsc{S-Trans} \qquad \frac{\tau_1 \le \tau_2 \qquad T_2 \le T_1 \qquad U_1 \le U_2}{\tau_1[T_1]U_1 \le \tau_2[T_2]U_2}\ \textsc{S-Cons}$$

$$\frac{T_1 \le T_2}{\tau \le \tau[T_1]T_2}\ \textsc{S-Lift} \qquad \frac{\tau_2 \le \tau_1 \qquad T_1 \le T_2}{(\tau_1 \to T_1) \le (\tau_2 \to T_2)}\ \textsc{S-Arr} \qquad \frac{\Gamma \vdash e\ :\ T \qquad T \le U}{\Gamma \vdash e\ :\ U}\ \textsc{T-Sub}$$

$$\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \textsc{T-Var} \qquad \frac{\Gamma, x:\tau \vdash e\ :\ T}{\Gamma \vdash \lambda x.e\ :\ \tau \to T}\ \textsc{T-Abs} \qquad \frac{\Gamma \vdash e_1\ :\ \tau \to T \qquad \Gamma \vdash e_2\ :\ \tau}{\Gamma \vdash e_1\ e_2\ :\ T}\ \textsc{T-PApp}$$

$$\frac{\Gamma \vdash e_1\ :\ (\tau_2 \to \tau_1[U_4]U_3)[U_2]U_1 \qquad \Gamma \vdash e_2\ :\ \tau_2[U_3]U_2}{\Gamma \vdash e_1\ e_2\ :\ \tau_1[U_4]U_1}\ \textsc{T-App} \qquad \frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \textsc{T-Const}$$

$$\frac{\Gamma, f:\tau \to T, x:\tau \vdash e\ :\ T}{\Gamma \vdash \mathsf{fix}\, f(x).e\ :\ \tau \to T}\ \textsc{T-Fix} \qquad \frac{\Gamma, x:\tau \to T \vdash e\ :\ U}{\Gamma \vdash \mathcal{S}_0 x.e\ :\ \tau[T]U}\ \textsc{T-Sft} \qquad \frac{\Gamma \vdash e\ :\ \tau[\tau]T}{\Gamma \vdash \langle e \rangle\ :\ T}\ \textsc{T-Rst}$$

Figure 5 The source language – λ-calculus with delimited control and effect subtyping.

4 Coherence of a CPS translation of control-effect subtyping

The calculus of delimited control studied in this section is the λ-calculus extended with recursion and the control operators shift_0 ($\mathcal{S}_0$) and reset_0 ($\langle\cdot\rangle$) [11] that can explore and reorganize an arbitrary portion of the stack of delimited continuations. It was built around a central idea that types for such a calculus should contain information about the stack and that the amount of that information should be governed by a subtyping mechanism [16]. We will define the semantics of the calculus by a CPS translation to a target calculus endowed with a reduction semantics, but if we were to directly give a reduction rule for shift_0, it would be:

$$F[\langle E[\mathcal{S}_0 x.e]\rangle] \quad \to \quad F[e\{\lambda y.\langle E[y]\rangle / x\}]$$

where E is a pure evaluation context representing the current delimited continuation (delimited by $\langle\cdot\rangle$ and captured by $\mathcal{S}_0$) and F is a metacontext, i.e., a list of such pure evaluation contexts separated by control delimiters, representing the current metacontinuation [6]. In terms of abstract-machine semantics E together with F represent the entire control stack. We can see from the reduction rule that nothing prevents the expression e from capturing another delimited continuation from F and, therefore, the types of the calculus express requirements on the shape of the control stack, so that expressions can safely perform control operations exploring the stack. However, under some conditions, an expression that imposes certain requirements on the stack can be used with a stack of which more is known or assumed. Such coercions are possible thanks to the subtyping relation that lies at the heart of the calculus.

4.1 The lambda calculus with delimited control and effect subtyping

The syntax and typing rules of the calculus of delimited control are shown in Figure 5. Types are either pure (τ) or effect annotated ($\tau[T_1]T_2$). An expression of type $\tau[T_1]T_2$ can be used with the top-most delimited continuation that when given a value of type τ behaves as specified by its answer type T_1, and with the rest of the stack whose type is T_2.

The calculus comprises the simply typed lambda calculus with the standard subtyping rules, extended with typing and subtyping rules for effectful computations. The most interesting from the point of view of effect subtyping are the rules S-Cons and S-Lift. The

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{Nat} \mid \tau \to T & \text{(pure types)} \\
T & ::= & \tau \mid (\tau \to T) \Rightarrow T & \text{(types)} \\
c & ::= & \mathsf{id} \mid c \circ c \mid c \to c \mid \uparrow c \mid c[c]c & \text{(coercions)} \\
e & ::= & x \mid \lambda x.e \mid e\ e \mid c\ e \mid \mathsf{fix}\ x(x).e \mid n & \text{(expressions)} \\
v & ::= & x \mid \lambda x.e \mid \mathsf{fix}\ x(x).e \mid (c \to c)\ v \mid \uparrow c\ v \mid (c[c]c)\ v \mid n & \text{(values)} \\
E & ::= & \Box \mid E\ e \mid v\ E \mid c\ E & \text{(evaluation contexts)}
\end{array}
$$

$$
\frac{}{\mathsf{id} :: T \triangleright T}\ \textsc{S-Refl} \qquad \frac{c_1 :: T_2 \triangleright T_3 \qquad c_2 :: T_1 \triangleright T_2}{c_1 \circ c_2 :: T_1 \triangleright T_3}\ \textsc{S-Trans}
$$

$$
\frac{c_1 :: \tau_2 \triangleright \tau_1 \qquad c_2 :: T_1 \triangleright T_2}{c_1 \to c_2 :: (\tau_1 \to T_1) \triangleright (\tau_2 \to T_2)}\ \textsc{S-Arr}
$$

$$
\frac{c :: T_1 \triangleright T_2}{\uparrow c :: \tau \triangleright ((\tau \to T_1) \Rightarrow T_2)}\ \textsc{S-Lift} \qquad \frac{c :: \tau_1 \triangleright \tau_2 \qquad c_1 :: T_2 \triangleright T_1 \qquad c_2 :: U_1 \triangleright U_2}{c[c_1]c_2 :: ((\tau_1 \to T_1) \Rightarrow U_1) \triangleright ((\tau_2 \to T_2) \Rightarrow U_2)}\ \textsc{S-Cons}
$$

$$
\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \textsc{T-Var} \qquad \frac{\Gamma, x : \tau \vdash e\ :\ T}{\Gamma \vdash \lambda x.e\ :\ \tau \to T}\ \textsc{T-Abs} \qquad \frac{\Gamma \vdash e_1\ :\ \tau \to T \qquad \Gamma \vdash e_2\ :\ \tau}{\Gamma \vdash e_1\ e_2\ :\ T}\ \textsc{T-App}
$$

$$
\frac{\Gamma, x : \tau \to T \vdash e\ :\ U}{\Gamma \vdash \lambda x.e\ :\ (\tau \to T) \Rightarrow U}\ \textsc{T-KAbs} \qquad \frac{\Gamma \vdash e\ :\ (\tau \to T) \Rightarrow U \qquad \Gamma \vdash v\ :\ \tau \to T}{\Gamma \vdash e\ v\ :\ U}\ \textsc{T-KApp}
$$

$$
\frac{c :: T \triangleright U \qquad \Gamma \vdash e\ :\ T}{\Gamma \vdash c\ e\ :\ U}\ \textsc{T-CApp} \qquad \frac{\Gamma, f : \tau \to T, x : \tau \vdash e\ :\ T}{\Gamma \vdash \mathsf{fix}\ f(x).e\ :\ \tau \to T}\ \textsc{T-Fix} \qquad \frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \textsc{T-Const}
$$

$$
\begin{array}{rcl}
E[(\lambda x.e)\ v] & \to_\beta & E[e\{v/x\}] \\
E[(\mathsf{fix}\ f(x).e)\ v] & \to_\beta & E[e\{\mathsf{fix}\ f(x).e/f, v/x\}] \\
E[\uparrow c\ v_1\ v_2] & \to_\beta & E[c\ (v_2\ v_1)]
\end{array}
\qquad
\begin{array}{rcl}
E[\mathsf{id}\ v] & \to_\iota & E[v] \\
E[(c_1 \circ c_2)\ v] & \to_\iota & E[c_1\ (c_2\ v)] \\
E[(c_1 \to c_2)\ v_1\ v_2] & \to_\iota & E[c_2\ (v_1\ (c_1\ v_2))] \\
E[(c[c_1]c_2)\ v_1\ v_2] & \to_\iota & E[c_2\ (v_1\ ((c \to c_1)\ v_2))]
\end{array}
$$

Figure 6 The target language – λ-calculus with explicit coercions of control effects.

rule S-Cons is a direct consequence of the interpretation of effect-annotated types given above that actually follows from the CPS interpretation of delimited continuations – a type $\tau[T_1]T_2$ is interpreted in CPS as $(\tau \to T_1) \Rightarrow T_2$, where $\Rightarrow$ means an effectful function space (see Section 4.2). The rule S-Lift is more interesting and it says that a pure computation can be considered impure, provided the answer type of the top-most delimited continuation can be coerced into the type of the rest of the stack. We have, for an example, $\mathsf{Nat} \leq \mathsf{Nat}[\mathsf{Nat}]\mathsf{Nat}$. It is worth noting that the calculus provides two rules for function application. They are necessary for expressivity reasons, but at the same time they are an additional source of difficulty when it comes to establishing the coherence of the subtyping. For more detailed presentation of the calculus we refer the reader to [16]. (The presentation of the type system in [16] differs from the one shown in Figure 5 in some inessential details, but the two type systems are equally expressive.)

4.2 Coercion semantics: a type-directed selective CPS translation

The syntax and typing rules of the target language are presented in Figure 6. There are two kinds of arrow type: the usual one $\tau \to T$ for regular functions and the effectful one $(\tau \to T) \Rightarrow U$ for expressions in CPS. We make this distinction to express the fact that the CPS translation (see Figure 7) yields expressions with strong restrictions on the occurrence of terms in CPS: they are never passed as arguments (typing environment consists of only pure types) and they can be applied only to values (witness the rule T-KApp) representing continuations.

$$\begin{array}{rclcrcl}
[\![\mathsf{Nat}]\!]_p & = & \mathsf{Nat} & & [\![\tau]\!] & = & [\![\tau]\!]_p \\
[\![\tau \to T]\!]_p & = & [\![\tau]\!]_p \to [\![T]\!] & & [\![\tau[T]U]\!] & = & ([\![\tau]\!]_p \to [\![T]\!]) \Rightarrow [\![U]\!]
\end{array}$$

$$\begin{array}{rcl}
\mathcal{S}[\![T \le T]\!]_{\text{S-Refl}} & = & \mathsf{id} \\
\mathcal{S}[\![T_1 \le T_3]\!]_{\text{S-Trans}(D_1,D_2)} & = & \mathcal{S}[\![T_2 \le T_3]\!]_{D_1} \circ \mathcal{S}[\![T_1 \le T_2]\!]_{D_2} \\
\mathcal{S}[\![\tau_1 \to T_1 \le \tau_2 \to T_2]\!]_{\text{S-Arr}(D_1,D_2)} & = & \mathcal{S}[\![\tau_2 \le \tau_1]\!]_{D_1} \to \mathcal{S}[\![T_1 \le T_2]\!]_{D_2} \\
\mathcal{S}[\![\tau \le \tau[T]U]\!]_{\text{S-Lift}(D)} & = & \uparrow \mathcal{S}[\![T \le U]\!]_D \\
\mathcal{S}[\![\tau_1[T_1]U_1 \le \tau_2[T_2]U_2]\!]_{\text{S-Cons}(D,D_1,D_2)} & = & \mathcal{S}[\![\tau_1 \le \tau_2]\!]_D[\mathcal{S}[\![T_2 \le T_1]\!]_{D_1}]\mathcal{S}[\![U_1 \le U_2]\!]_{D_2}
\end{array}$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{T}[\![x]\!]_{\text{T-Var}} & = & x & \mathcal{T}[\![\mathcal{S}_0 x.e]\!]_{\text{T-Sft}(D)} & = & \lambda x.\mathcal{T}[\![e]\!]_D \\
\mathcal{T}[\![\lambda x.e]\!]_{\text{T-Abs}(D)} & = & \lambda x.\mathcal{T}[\![e]\!]_D & \mathcal{T}[\![\langle e \rangle]\!]_{\text{T-Rst}(D)} & = & \mathcal{T}[\![e]\!]_D\ (\lambda x.x) \\
\mathcal{T}[\![\mathsf{fix}\, f(x).e]\!]_{\text{T-Fix}(D)} & = & \mathsf{fix}\, f(x).\mathcal{T}[\![e]\!]_D & \mathcal{T}[\![n]\!]_{\text{T-Const}} & = & n \\
\mathcal{T}[\![e]\!]_{\text{T-Sub}(D_1,D_2)} & = & \mathcal{S}[\![T \le U]\!]_{D_2}\, \mathcal{T}[\![e]\!]_{D_1} & & & \\
\mathcal{T}[\![e_1\ e_2]\!]_{\text{T-PApp}(D_1,D_2)} & = & \mathcal{T}[\![e_1]\!]_{D_1}\, \mathcal{T}[\![e_2]\!]_{D_2} & & & \\
\mathcal{T}[\![e_1\ e_2]\!]_{\text{T-App}(D_1,D_2)} & = & \lambda k.\mathcal{T}[\![e_1]\!]_{D_1}\, (\lambda f.\mathcal{T}[\![e_2]\!]_{D_2}\, (\lambda x.f\ x\ k)) & & &
\end{array}$$

Figure 7 Type-directed selective CPS translation.

Again, the operational semantics distinguishes between β-rules and ι-rules. We classified the last β-rule as "actual computation" because it does not only rearrange coercions. It translates back a lifted value v_1 and applies to it a given continuation v_2. This rule and the last ι-rule reduce a coerced value applied to a continuation, so terms of the form $(\uparrow c\, v)$ and $(c[c]c\, v)$ are considered values. Notice that these values have effectful types. We extend the notion of ι-reduction to evaluation contexts: $E_1 \to_\iota E_2$ holds iff $E_1[v] \to_\iota E_2[v]$ for every value v.

As in Section 3, the metavariable C denotes general closed contexts. We also define typing of general contexts $\vdash C\ :\ (\Gamma;T) \rightsquigarrow T_0$ as before. The definition of contextual approximation $\Gamma \vdash e_1 \precsim_{ctx} e_2\ :\ T$ is slightly weaker, because we allow only contexts with pure answer type. Indeed, an expression that requires a continuation to trigger computation can hardly be considered a complete program.

The coercion semantics of the source language is given by the type-directed selective CPS translation presented in Figure 7. The translation is selective because it leaves terms of pure type in direct style – witness, e.g, the equations for variable or pure application. Effectful applications are translated according to Plotkin's call-by-value CPS translation [21], whereas the translation of shift_0 and reset_0 is surprisingly straightforward – shift_0 is turned into a lambda-abstraction expecting a delimited continuation, and reset_0 is interpreted by providing its subexpression with the reset delimited continuation, represented by the identity function.

▶ **Example 10.** Consider the program $(\mathsf{fix}\, f(x).f\ x)\ 1$ in the source language. We derive the type $\mathsf{Nat}[T]T$ for it in two ways: let D_1 be the derivation

$$\dfrac{\dfrac{\dfrac{\vdots}{f : \mathsf{Nat} \to \mathsf{Nat}[T]T, x : \mathsf{Nat} \vdash f\ x\ :\ \mathsf{Nat}[T]T}}{\vdash \mathsf{fix}\, f(x).f\ x\ :\ \mathsf{Nat} \to \mathsf{Nat}[T]T}\ \text{T-Fix} \qquad \dfrac{}{\vdash 1\ :\ \mathsf{Nat}}\ \text{T-Const}}{\vdash (\mathsf{fix}\, f(x).f\ x)\ 1\ :\ \mathsf{Nat}[T]T}\ \text{T-PApp}$$

and D_2 be the derivation

$$
\begin{array}{rcl}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Nat}; \mathsf{Nat}]\!] & \iff & \exists n, v_1 = v_2 = n \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to T_1; \tau_2 \to T_2]\!] & \iff & \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].(v_1\ a_1, v_2, a_2\ ) \in \mathcal{E}[\![T_1; T_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] & \iff & \bot \qquad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![T_1; T_2]\!] & \iff & \forall (E_1, E_2) \in \mathcal{K}[\![T_1; T_2]\!].E_1[e_1] \precsim E_2[e_2] \\
(E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!] & \iff & \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].E_1[v_1] \precsim E_2[v_2] \\
(E_1, E_2) \in \mathcal{K}[\![\tau_1; (\tau_2 \to T_2) \Rightarrow U_2]\!] & \iff & \exists T, (E, k) \in \mathcal{KV}[\![\tau_1 \rightsquigarrow T; \tau_2 \to T_2]\!], \\
 & & \quad (E_1', E_2') \in \mathcal{K}[\![T; U_2]\!]. \\
 & & \quad\quad E_1 \to_\iota^* E_1'[E[\,]] \wedge E_2 \to_\iota^* E_2'[[\,]\ k] \\
(E_1, E_2) \in \mathcal{K}[\![(\tau_1 \to T_1) \Rightarrow U_1; \tau_2]\!] & \iff & \exists T, (k, E) \in \rhd\mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \rightsquigarrow T]\!], \\
 & & \quad (E_1', E_2') \in \rhd\mathcal{K}[\![U_1; T]\!]. \\
 & & \quad\quad E_1 \to_\iota^* E_1'[[\,]\ k] \wedge E_2 \to_\iota^* E_2'[E[\,]] \\
(E_1, E_2) \in \mathcal{K}[\![(\tau_1 \to T_1) \Rightarrow U_1; & & \\
\quad (\tau_2 \to T_2) \Rightarrow U_2]\!] & \iff & \exists (k_1, k_2) \in \mathcal{V}[\![\tau_1 \to T_1; \tau_2 \to T_2]\!], \\
 & & \quad (E_1', E_2') \in \mathcal{K}[\![U_1; U_2]\!]. \\
 & & \quad\quad E_1 \to_\iota^* E_1'[[\,]\ k_1] \wedge E_2 \to_\iota^* E_2'[[\,]\ k_2] \\
(E, v) \in \mathcal{KV}[\![\tau_1 \rightsquigarrow T_1; \tau_2 \to T_2]\!] & \iff & \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].(E[a_1], v\ a_2) \in \mathcal{E}[\![T_1; T_2]\!] \\
(v, E) \in \mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \rightsquigarrow T_2]\!] & \iff & \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].(v\ a_1, E[a_2]) \in \mathcal{E}[\![T_1; T_2]\!] \\
(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!] & \iff & \forall x.(\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\Gamma_1(x); \Gamma_2(x)]\!] \\
\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2\ :\ T_1; T_2 & \iff & \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!].(e_1\gamma_1, e_2\gamma_2) \in \mathcal{E}[\![T_1; T_2]\!]
\end{array}
$$

Figure 8 Logical relations for the λ-calculus with explicit coercions of control effects.

$$
\dfrac{\dfrac{\dfrac{\vdots}{\vdash \mathsf{fix}\ f(x).f\ x\ :\ \mathsf{Nat} \to \mathsf{Nat}[T]T} \quad \cdots}{\vdash \mathsf{fix}\ f(x).f\ x\ :\ (\mathsf{Nat} \to \mathsf{Nat}[T]T)[T]T}\ \text{T-Sub} \quad \dfrac{\dfrac{}{\vdash 1\ :\ \mathsf{Nat}}\ \text{T-Const} \quad \cdots}{\vdash 1\ :\ \mathsf{Nat}[T]T}\ \text{T-Sub}}{\vdash (\mathsf{fix}\ f(x).f\ x)\ 1\ :\ \mathsf{Nat}[T]T}\ \text{T-App}
$$

Then

$$
\begin{array}{rcl}
\mathcal{T}[\![(\mathsf{fix}\ f(x).f\ x)\ 1]\!]_{D_1} & = & (\mathsf{fix}\ f(x).f\ x)\ 1 \\
\mathcal{T}[\![(\mathsf{fix}\ f(x).f\ x)\ 1]\!]_{D_2} & = & \lambda k.(\uparrow \mathsf{id}\ (\mathsf{fix}\ f(x).f\ x))\ (\lambda g.(\uparrow \mathsf{id}\ 1)\ (\lambda y.g\ y\ k))
\end{array}
$$

By the results of the next sections, these terms are contextually equivalent.

▶ **Lemma 11.** *Coercion semantics preserves types.*
1. *If* $D :: T_1 \le T_2$ *then* $\mathcal{S}[\![T_1 \le T_2]\!]_D :: [\![T_1]\!] \rhd [\![T_2]\!]$.
2. *If* $D :: \Gamma \vdash e\ :\ T$ *then* $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_D\ :\ [\![T]\!]$.

4.3 Logical relations

The logical relations are defined in Figure 8. The relation $\mathcal{V}[\![\tau_1; \tau_2]\!]$ for pure values and the relation $\mathcal{E}[\![T_1; T_2]\!]$ for expressions are similar to the relations defined in Section 3.3. All information about control effects is captured in the relation $\mathcal{K}[\![T_1; T_2]\!]$ for contexts. If T_1 and T_2 are computation arrow types, then two contexts are related iff they can be decomposed as applications to related continuations in related contexts. In general, this application to a continuation does not have to be the top-most element of the context and some rearrangement of coercions is needed, so such a decomposition is defined by ι-reduction of contexts.

The most interesting are the cases that relate pure and impure contexts. As previously, the impure context should be decomposed to a continuation k and the rest of the context. Then the pure context should be decomposed in such a way that the continuation k is

related with some portion E of the pure context. The answer type of E cannot be retrieved from the type of the initial pure context, so we quantify over all possible types. In order to make the construction well-founded, the relations are defined by nested induction on step indices and on the structure of the second type. Notice that step indices play a role only in one case – when we quantify over the second type and the later operator guards the non-structural use of the relations $\mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \rightsquigarrow T]\!]$ and $\mathcal{K}[\![U_1; T]\!]$. The auxiliary relations $\mathcal{KV}[\![\tau_1 \rightsquigarrow T_1; \tau_2 \to T_2]\!]$ and $\mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \rightsquigarrow T_2]\!]$ relate a portion of an evaluation context with a value of an arrow type and they are defined analogously to the value relation for functions.

The relations of this section possess properties analogous to the ones of Section 3.3, in particular the relation $\precsim$ is preserved by reduction (Lemma 2) and the compatibility lemmas (including Lemma 3) hold. However, the proof of the compatibility lemmas requires the following results that establish the preservation of relations with respect to ι-reductions of evaluation contexts.

▶ **Lemma 12.** *The following assertions hold:*
1. *If* $E \to_\iota^* E'$ *and* $E'[e_1] \precsim e_2$ *then* $E[e_1] \precsim e_2$.
2. *If* $E \to_\iota^* E'$ *and* $e_1 \precsim E'[e_2]$ *then* $e_1 \precsim E[e_2]$.
3. *If* $E_1 \to_\iota^* E_1'$ *and* $(E_1', E_2) \in \mathcal{K}[\![T_1; T_2]\!]$ *then* $(E_1, E_2) \in \mathcal{K}[\![T_1; T_2]\!]$.
4. *If* $E_2 \to_\iota^* E_2'$ *and* $(E_1, E_2') \in \mathcal{K}[\![T_1; T_2]\!]$ *then* $(E_1, E_2) \in \mathcal{K}[\![T_1; T_2]\!]$.

The rest of the soundness proof follows the same lines as in Section 3.3. Interestingly, the adequacy lemma can be proved only for pure types, which is in harmony with the notion of contextual equivalence in the target calculus.

▶ **Theorem 13** (Fundamental property). *If* $\Gamma \vdash e : T$ *then* $\Gamma; \Gamma \vdash e \precsim_{log} e : T; T$.

▶ **Lemma 14** (Precongruence). *If* $\vdash C : (\Gamma; T) \rightsquigarrow \tau$ *and* $\Gamma; \Gamma \vdash e_1 \precsim_{log} e_2 : T; T$, *then* $(C[e_1], C[e_2]) \in \mathcal{E}[\![\tau; \tau]\!]$.

▶ **Lemma 15** (Adequacy). *If* $(e_1, e_2) \in \mathcal{E}[\![\tau; \tau]\!]$ *then* $e_1 \precsim e_2$.

▶ **Theorem 16** (Soundness). *If* $k \models \Gamma; \Gamma \vdash e_1 \precsim_{log} e_2 : T; T$ *holds for every* k, *then* $\Gamma \vdash e_1 \precsim_{ctx} e_2 : T$.

4.4 Coherence of the CPS translation

Although standard compatibility lemmas and coercion compatibility suffice to prove soundness of logical relations, we need another kind of compatibility to prove coherence, since there is another source of ambiguity. Two typing derivations in the source language can be different not only because of the subsumption rule, but also because of two rules for application.

▶ **Lemma 17** (Mixed application compatibility). *The following assertions hold:*
1. *If* $\Gamma_1; \Gamma_2 \vdash f_1 \precsim_{log} f_2 : ((\tau_1' \to (\tau_1 \to U_4) \Rightarrow U_3) \to U_2) \Rightarrow U_1; \tau_2' \to T_2$
   and $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 : (\tau_1' \to U_3) \Rightarrow U_2; \tau_2'$
   then $\Gamma_1; \Gamma_2 \vdash \lambda k.f_1\,(\lambda f.e_1\,(\lambda x.f\ x\ k)) \precsim_{log} f_2\,e_2 : (\tau_1' \to U_4) \Rightarrow U_1; T$.
2. *If* $\Gamma_1; \Gamma_2 \vdash f_1 \precsim_{log} f_2 : \tau_1' \to T_1; ((\tau_2' \to (\tau_2 \to U_4) \Rightarrow U_3) \to U_2) \Rightarrow U_1$
   and $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 : \tau_1'; (\tau_2' \to U_3) \Rightarrow U_2$
   then $\Gamma_1; \Gamma_2 \vdash f_1\,e_1 \precsim_{log} \lambda k.f_2\,(\lambda f.e_2\,(\lambda x.f\ x\ k)) : T; (\tau_2' \to U_4) \Rightarrow U_1$.

Proof. Both cases are similar, so we show only the first one. We have to show that both terms closed by substitutions have the same observations in related contexts $(E_1, E_2) \in$

$\mathcal{K}[\![(\tau_1' \to U_4) \Rightarrow U_1; T]\!]$. Since context E_1 is in relation for effectful type, by the definition of logical relations and Lemma 12, it can be decomposed as a continuation k and the rest of the context. Now we have the missing continuation k that can trigger computation in the first term, so the rest of the proof consists in simple context manipulations, applying definitions and performing reductions. ◀

▶ **Lemma 18.** *If $D_i :: \Gamma_i \vdash e \ : \ T_i$ for $i = 1, 2$ are two typing judgments for the same term e of the source language, then $[\![\Gamma_1]\!]; [\![\Gamma_2]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{log} \mathcal{T}[\![e]\!]_{D_2} \ : \ [\![T_1]\!]; [\![T_2]\!]$.*

▶ **Theorem 19** (Coherence)**.** *If D_1 and D_2 are derivations of the same typing judgment $\Gamma \vdash e \ : \ T$, then $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \ \precsim_{ctx} \ \mathcal{T}[\![e]\!]_{D_2} \ : \ [\![T]\!]$.*

In contrast to the calculus considered in Section 3.4, such coherence theorem does not imply coherence of translation directly to the simply typed λ-calculus (where coercions are expressed as λ-terms). As a counterexample, the derivations from Example 10 give us terms that can be distinguished by the context $C = (\lambda x.1)\,[\,]$. This is because types in the target language carry more information than simple types, and in particular, an expression of a type $(\tau \to T) \Rightarrow U$ is not a usual function, but a computation waiting for a continuation, as explained in Section 4.2.

But still we can prove some interesting properties of a direct translation to the simply typed λ-calculus in two cases: when control effects do not leak to the context or when we relate only whole programs. Let $|e|$ be a term e with all coercions replaced by corresponding expressions.

▶ **Corollary 20.** *If $D_1, D_2 :: \Gamma \vdash e \ : \ \tau$ and τ does not contain any type of the form $\tau'[T]U$, then $|\mathcal{T}[\![e]\!]_{D_1}|$ and $|\mathcal{T}[\![e]\!]_{D_2}|$ are contextually equivalent.*

▶ **Corollary 21.** *If $D_1, D_2 :: \Gamma \vdash e \ : \ \tau$ then $|\mathcal{T}[\![e]\!]_{D_1}|$ terminates iff $|\mathcal{T}[\![e]\!]_{D_2}|$ terminates. Moreover, if $\tau = \mathsf{Nat}$ and one of the expressions terminates to a constant, then the other term evaluates to the same constant.*

5 Conclusion

We have shown that the technique of logical relations can be used for establishing the coherence of subtyping, when it is phrased in terms of contextual equivalence in the target of the coercion translation. In particular, we have demonstrated that a combination of heterogeneity, biorthogonality and step-indexing provides a sufficiently powerful tool for establishing coherence of effect subtyping in a calculus of delimited control with the coercion semantics given by a type-directed selective CPS translation. However, we have successfully applied the presented approach also to other calculi with subtyping, e.g., as demonstrated in this article for the simply-typed λ-calculus with recursion. The Coq development accompanying this paper is based on a new embedding of Dreyer et al.'s logic for reasoning about step-indexing that, we believe, considerably improves the presentation and formalization of the logical relations.

Regarding logical relations for type-and-effect systems, there has been a work on proving correctness of a partial evaluator for shift and reset by Asai [5], and on termination of evaluation of the λ-calculi with delimited-control operators by Biernacka et al. [6, 7] and by Materzok and the first author [16]. Unsurprisingly, all these results, like ours, are built on the notion of biorthogonality, even if not mentioned explicitly. The distinctive feature of our construction is a combination of heterogeneity and step-indexing that supports reasoning about the observational equivalence of terms of different types whose structure is very distant from each other, e.g., about direct-style and continuation-passing-style terms.

Acknowledgments. We thank Andrés A. Aristizábal, Małgorzata Biernacka, and the anonymous reviewers for helpful comments on the presentation of this work.

References

1 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL'09*, Savannah, GA, USA, 2009, pp. 340–353.

2 Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP'06*, Vienna, Austria, March 2006, *LNCS* 3924, pp. 69–83.

3 Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM TOPLAS*, 23(5):657–683, 2001.

4 Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL'07*, Nice, France, 2007, pp. 109–122.

5 Kenichi Asai. Logical relations for call-by-value delimited continuations. In *Trends in Functional Programming 2005*, Tallinn, Estonia, 2005, pp. 413–428.

6 Małgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. In *PPDP'09*, Coimbra, Portugal, 2009, pp. 289–300.

7 Małgorzata Biernacka, Dariusz Biernacki, and Sergueï Lenglet. Typing control operators in the CPS hierarchy. In *PPDP'11*, Odense, Denmark, 2011, pp. 149–160.

8 Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

9 Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

10 Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, 1989.

11 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Lisp and Functional Programming 1990*, Nice, France, 1990, pp. 151–160.

12 Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2:16):1–37, 2011.

13 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

14 Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68(1):53–78, 1994.

15 Marek Materzok. Axiomatizing subtyped delimited continuations. In *CSL'13*, Torino, Italy, 2013, *LIPIcs* 23, pp. 521–539.

16 Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In *ICFP'11*, Tokyo, Japan, 2011, pp. 81–93.

17 Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In *APLAS'12*, LNCS 7705, Kyoto, Japan, 2012, pp. 296–311.

18 John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

19 James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

20 Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pp. 227–273. 1998.

21 Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

22 John C. Reynolds. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software 1991*, Sendai, Japan, 1991, *LNCS* 526, pp. 675–700.

23 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP'09*, Edinburgh, UK, 2009, pp. 317–328.

24 Jan Schwinghammer. Coherence of subsumption for monadic types. *Journal of Functional Programming*, 19(2):157–172, 2009.

Observability for Pair Pattern Calculi

Antonio Bucciarelli[1], Delia Kesner[1], and Simona Ronchi Della Rocca[2]

1 **Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, Paris, France**
2 **Dipartimento di Informatica, Università di Torino, Italy**

Abstract

Inspired by the notion of solvability in the λ-calculus, we define a notion of observability for a calculus with pattern matching. We give an intersection type system for such a calculus which is based on non-idempotent types. The typing system is shown to characterize the set of terms having canonical form, which properly contains the set of observable terms, so that typability alone is not sufficient to characterize observability. However, the inhabitation problem associated with our typing system turns out to be decidable, a result which – together with typability – allows to obtain a full characterization of observability.

1998 ACM Subject Classification F.4.1 Lambda calculus and related systems, F.3.2 Operational Semantics, F.4.1 Proof theory

Keywords and phrases Solvability, pattern calculi, intersection types, inhabitation

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.123

1 Introduction

In these last years there has been a growing interest in *pattern* λ*-calculi* [16, 11, 6, 12, 10, 15] which are used to model the pattern-matching primitives of functional programming languages (*e.g.* OCAML, ML, Haskell) and proof assistants (*e.g.* Coq, Isabelle). These calculi are extensions of λ-calculus, where abstractions are written as $\lambda\mathtt{p}.\mathtt{t}$, where $\mathtt{p}$ is a *pattern* specifying the expected structure of the argument. In this paper we restrict our attention to *pair* patterns, which are expressive enough to illustrate the challenging notion of solvability/observability in the framework of pattern λ-calculi.

In order to implement different *evaluation strategies*, the use of *explicit pattern-matching* becomes appropriate, giving rise to different languages with explicit pattern-matching [6, 7, 1]. In all of them, an application $(\lambda\mathtt{p}.\mathtt{t})\mathtt{u}$ reduces to $\mathtt{t}[\mathtt{p}/\mathtt{u}]$, where $[\mathtt{p}/\mathtt{u}]$ is an explicit matching, defined by means of suitable reduction rules, which are used to decide if the argument $\mathtt{u}$ matches the pattern $\mathtt{p}$. If the matching is possible, the evaluation proceeds by computing a substitution which is applied to the body $\mathtt{t}$. Otherwise, two cases arise: either a successful matching is not possible at all, and then the term $\mathtt{t}[\mathtt{p}/\mathtt{u}]$ reduces to a *failure*, denoted by the constant $\mathtt{fail}$, or it could become possible after the application of some pertinent substitution to the argument $\mathtt{u}$, in which case the reduction is simply *blocked*. An example of failure is caused by the term $(\lambda\langle\mathtt{z}_1,\mathtt{z}_2\rangle.\mathtt{z}_1)(\lambda\mathtt{y}.\mathtt{y})$, while a blocked reduction is caused by the term $(\lambda\langle\mathtt{z}_1,\mathtt{z}_2\rangle.\mathtt{z}_1)\mathtt{y}$.

Inspired by the notion of solvability in the λ-calculus, we define a notion of *observability* for a pair pattern calculus with explicit matching. A term $\mathtt{t}$ is said to be observable if there is a *head-context* $\mathtt{C}$ such that $\mathtt{C}[\mathtt{t}]$ reduces to a pair, which is the only data structure of the language. This notion is conservative with respect to the notion of solvability in the λ-calculus, *i.e.* $\mathtt{t}$ is solvable in the λ-calculus if and only if $\mathtt{t}$ is observable in our calculus.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 123–137

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Solvability in the λ-calculus is of course undecidable, but it has been characterized at least in three different ways: syntactically by the notion of head-normal form [2], operationally by the notion of head-reduction [2], and logically by an intersection type assignment system [3, 13]. The problem becomes harder when changing from the call-by-name to the call-by-value setting. Indeed, in the call-by-value λ-calculus, there are normal forms that are unsolvable, like the term $(\lambda \mathtt{z}.\Delta)(\mathtt{x}I)\Delta$, where $\Delta = \lambda \mathtt{x}.\mathtt{xx}$. The problem for the pair pattern calculus is similar to that for the call-by-value, but even harder. As in the call-by-value setting, an argument needs to be partially evaluated before being consumed. Indeed, in order to evaluate an application $(\lambda \mathtt{p}.\mathtt{t})\mathtt{u}$, it is necessary to verify if $\mathtt{u}$ matches the pattern $\mathtt{p}$, and thus the subterm $\mathtt{u}$ can be forced to be partially evaluated. However, while only discrimination between values and non-values are needed in the call-by-value setting, the possible shapes of patterns are infinite here.

The difficulty of the problem depends on two facts. First, there is no simple *syntactical* characterization of observability: indeed, we supply a notion of *canonical form* such that reducing to some canonical form is a necessary condition for being observable. But this is not sufficient: canonical forms may contain blocking explicit matchings, so that we need to know whether or not there exists a substitution being able to unblock simultaneously all these blocked forms.

This theoretical complexity is reflected in the logical characterization we supply for observability: a term $\mathtt{t}$ turns out to be observable if and only if it is typable, say with a type of the shape $\mathtt{A}_1 \to \mathtt{A}_2 \to ... \to \mathtt{A}_n \to \alpha$ (where α is a product type), and all the types $\mathtt{A}_i$ $(1 \leq i \leq n)$ are *inhabited*. The inhabitation problem for idempotent intersection types is known to be undecidable [17], but it has recently been proved that it is decidable in the non-idempotent case [5]. More precisely, there is a sound and complete algorithm solving the inhabitation problem of non-idempotent intersection types for the λ-calculus. In this paper, we supply a type assignment system, based on non-idempotent intersection, which assigns types to terms of our pair pattern calculus. We then extend the inhabitation algorithm given in [5] to this framework, that is substantially more complicated, due to the explicit pattern matching and the use of structural information of patterns in the typing rules. However, the paper does not only show decidability of inhabitation for the pair pattern calculus, but it *uses* the decidability result to derive a full characterization of observability, which is the main result of the paper. We thus combine typability with inhabitation in order to obtain an interesting characterization of the set of *meaningful* terms of the pair pattern calculus.

The paper is organized as follows. Sec. 2 introduces the pattern calculus. Sec. 3 presents the type system and proves a characterization of terms having canonical forms by means of typability. Sec. 4 discusses the relationship between observability and inhabitation and Sec. 5 presents a sound and complete algorithm for the inhabitation problem associated to our typing system. Sec. 6 shows a complete characterization of observability, and Sec. 7 concludes by discussing some future work.

2 The Pair Pattern Calculus

We now introduce the Λ_{p}-calculus, a generalization of the λ-calculus where abstraction is extended to *patterns* and terms to pairs. Pattern matching is specified by means of an *explicit* operation. Reduction is performed only if the argument matches the abstracted pattern.

Terms and contexts of the Λ_{p}-calculus are defined as follows:

$$\begin{array}{llll} \textbf{(Patterns)} & \mathtt{p},\mathtt{q} & ::= & \mathtt{x} \mid \langle \mathtt{p},\mathtt{p}\rangle \\ \textbf{(Terms)} & \mathtt{t},\mathtt{u},\mathtt{v} & ::= & \mathtt{x} \mid \lambda\mathtt{p}.\mathtt{t} \mid \langle \mathtt{t},\mathtt{t}\rangle \mid \mathtt{t}\mathtt{t} \mid \mathtt{t}[\mathtt{p}/\mathtt{t}] \mid \mathtt{fail} \\ \textbf{(Contexts)} & \mathtt{C} & ::= & \Box \mid \lambda\mathtt{p}.\mathtt{C} \mid \langle \mathtt{C},\mathtt{t}\rangle \mid \langle \mathtt{t},\mathtt{C}\rangle \mid \mathtt{C}\mathtt{t} \mid \mathtt{t}\mathtt{C} \end{array}$$

where $\mathtt{x},\mathtt{y},\mathtt{z}$ range over a countable set of variables, and every pattern $\mathtt{p}$ is *linear*, *i.e.* every variable appears at most once in $\mathtt{p}$. We denote by $\mathtt{Id}$ the **identity function** $\lambda\mathtt{x}.\mathtt{x}$. As usual we use the abbreviation $\lambda\mathtt{p}_1 \dots \mathtt{p}_n.\mathtt{t}_1 \dots \mathtt{t}_m$ for $\lambda\mathtt{p}_1(\dots(\lambda\mathtt{p}_n.((\mathtt{t}_1\mathtt{t}_2)\dots\mathtt{t}_m))\dots)$, $n \geq 0$, $m \geq 1$. Remark that every λ-term is a $\Lambda_{\mathtt{p}}$-term.

The operator $[\mathtt{p}/\mathtt{t}]$ is called an **explicit matching**. The constant $\mathtt{fail}$ denotes the failure of the matching operation. **Free** and **bound variables** of terms are defined as expected, in particular $\mathtt{fv}(\lambda\mathtt{p}.\mathtt{t}) := \mathtt{fv}(\mathtt{t}) \setminus \mathtt{fv}(\mathtt{p})$ and $\mathtt{fv}(\mathtt{t}[\mathtt{p}/\mathtt{u}]) := (\mathtt{fv}(\mathtt{t}) \setminus \mathtt{fv}(\mathtt{p})) \cup \mathtt{fv}(\mathtt{u})$. We write $\mathtt{p}\#\mathtt{q}$ iff $\mathtt{fv}(\mathtt{p})$ and $\mathtt{fv}(\mathtt{q})$ are disjoint. As usual, terms are considered modulo α-conversion. Given a context $\mathtt{C}$ and a term $\mathtt{t}$, $\mathtt{C}[\mathtt{t}]$ denotes the term obtained by replacing the unique occurrence of $\Box$ in $\mathtt{C}$ by $\mathtt{t}$, allowing the capture of free variables of $\mathtt{t}$. A **head-context** is a context of the shape $(\lambda\mathtt{p}_1 ... \mathtt{p}_n.\Box)\mathtt{t}_1 ... \mathtt{t}_m$ $(n, m \geq 0)$.

The **reduction relation** of the $\Lambda_{\mathtt{p}}$-calculus, denoted by $\to$, is the contextual closure of the following reduction rules:

$$\begin{array}{llll} (r_1) & (\lambda\mathtt{p}.\mathtt{t})\mathtt{u} & \mapsto & \mathtt{t}[\mathtt{p}/\mathtt{u}] \\ (r_2) & \mathtt{t}[\mathtt{x}/\mathtt{u}] & \mapsto & \mathtt{t}\{\mathtt{x}/\mathtt{u}\} \\ (r_3) & \mathtt{t}[\langle \mathtt{p}_1,\mathtt{p}_2\rangle/\langle \mathtt{u}_1,\mathtt{u}_2\rangle] & \mapsto & \mathtt{t}[\mathtt{p}_1/\mathtt{u}_1][\mathtt{p}_2/\mathtt{u}_2] \\ (r_4) & \mathtt{t}[\mathtt{p}/\mathtt{v}]\mathtt{u} & \mapsto & (\mathtt{t}\mathtt{u})[\mathtt{p}/\mathtt{v}] \\ (r_5) & \mathtt{t}[\langle \mathtt{p}_1,\mathtt{p}_2\rangle/\mathtt{u}[\mathtt{q}/\mathtt{v}]] & \mapsto & \mathtt{t}[\langle \mathtt{p}_1,\mathtt{p}_2\rangle/\mathtt{u}][\mathtt{q}/\mathtt{v}] \end{array}$$

$$\begin{array}{llll} (r_6) & \mathtt{t}[\langle \mathtt{p}_1,\mathtt{p}_2\rangle/\lambda\mathtt{y}.\mathtt{u}] & \mapsto & \mathtt{fail} \\ (r_7) & \mathtt{t}[\langle \mathtt{p}_1,\mathtt{p}_2\rangle/\mathtt{fail}] & \mapsto & \mathtt{fail} \\ (r_8) & \mathtt{fail}\ \mathtt{t} & \mapsto & \mathtt{fail} \\ (r_9) & \mathtt{fail}[\mathtt{p}/\mathtt{t}] & \mapsto & \mathtt{fail} \\ (r_{10}) & \lambda\mathtt{p}.\mathtt{fail} & \mapsto & \mathtt{fail} \\ (r_{11}) & \langle \mathtt{t},\mathtt{u}\rangle\mathtt{v} & \mapsto & \mathtt{fail} \end{array}$$

where $\mathtt{t}\{\mathtt{x}/\mathtt{u}\}$ denotes the substitution of all the free occurrences of $\mathtt{x}$ in $\mathtt{t}$ by $\mathtt{u}$. By α-conversion, and without loss of generality, no reduction rule captures free variables. Thus for example in rule r_4 the bound and free variables of the term $\mathtt{t}[\mathtt{p}/\mathtt{v}]\mathtt{u}$ are supposed to be disjoint, so that the variables of $\mathtt{p}$ (which are bound in the whole term) cannot be free in $\mathtt{u}$. The reflexive and transitive closure of $\to$ is written $\to^*$.

The rule (r_1) triggers the pattern operation while rule (r_2) performs substitution, rules (r_3), (r_6) and (r_7) implement (successful or unsuccessful) pattern matching. Rules (r_8), (r_9) and (r_{10}) deal with propagation of failure. Rules (r_4) and (r_5) may seem unnecessary, and the calculus would be also confluent without them, but they are particularly useful for the design of the inhabitation algorithm (see Sec. 5). Indeed, rule (r_4) pushes *head* explicit matchings out, and rule (r_5) eliminates *nested* explicit matchings, *i.e.* matchings of the form $\mathtt{t}[\langle \mathtt{p}_1,\mathtt{p}_2\rangle/\mathtt{u}[\mathtt{q}/\mathtt{v}]]$. Notice that confluence would be lost if we allow (r_5) on the more general form: $\mathtt{t}[\mathtt{p}/\mathtt{u}[\mathtt{q}/\mathtt{v}]] \mapsto \mathtt{t}[\mathtt{p}/\mathtt{u}][\mathtt{q}/\mathtt{v}]$. Indeed, the following critical pair could not be closed: $\mathtt{y}[\langle \mathtt{z}_1,\mathtt{z}_2\rangle/\mathtt{z}]$ $_{r_5,r_2}{}^{*}\!\leftarrow$ $\mathtt{y}[\mathtt{x}/\mathtt{u}[\langle \mathtt{z}_1,\mathtt{z}_2\rangle/\mathtt{z}]] \to_{r_2} \mathtt{y}$.

▶ **Lemma 1.**

1. *The reduction relation $\to$ is confluent.*
2. *Every infinite $\to$-reduction sequence contains an infinite number of $\to_{r_2}$-reduction steps.*

The proof of the first item relies on the decreasing diagram technique [18]; that of the second one is by induction on a suitable syntactic measure.

Canonical forms are terms defined by the following grammar:

$$\mathcal{J} ::= \lambda\mathtt{p}.\mathcal{J} \mid \langle \mathtt{t},\mathtt{t}\rangle \mid \mathcal{K} \mid \mathcal{J}[\langle \mathtt{p},\mathtt{q}\rangle/\mathcal{K}] \qquad\qquad \mathcal{K} ::= \mathtt{x} \mid \mathcal{K}\mathtt{t}$$

A term $\mathtt{t}$ **is in canonical form** (or it is **canonical**), written cf, if it is generated by $\mathcal{J}$, and it **has a canonical form** if it reduces to a term in cf. Note that the cf of a term is not unique, *e.g.* both $\langle \mathtt{Id}, \mathtt{Id}\ \mathtt{Id}\rangle$ and $\langle \mathtt{Id}, \mathtt{Id}\rangle$ are cfs of $(\lambda \mathtt{xy}.\langle \mathtt{x}, \mathtt{y}\rangle)\ \mathtt{Id}\ (\mathtt{Id}\ \mathtt{Id})$. It is worth noticing that cfs and normal forms do not coincide. For example, the terms $\lambda\langle \mathtt{x}, \mathtt{y}\rangle.(\mathtt{x}(\Delta\Delta))[\langle \mathtt{z}_1, \mathtt{z}_2\rangle/\mathtt{yId}]$ and $\langle \mathtt{Id}, \mathtt{Id}\ \mathtt{Id}\rangle$ are in cf, but not in normal form, while $\mathtt{fail}$ is in normal form but not in cf. Every head normal-form in the λ-calculus is a cf in the $\Lambda_{\mathtt{p}}$-calculus.

On the pathway towards the definition of an adequate notion of *solvability* for the $\Lambda_{\mathtt{p}}$-calculus, we first recall the notion of solvability for the λ-calculus. A term $\mathtt{t}$ is solvable iff there is a head-context $\mathtt{C}$ such that $\mathtt{C}[\mathtt{t}]$ reduces to $\mathtt{Id}$. It is clear that pairs have to be taken into account in order to extend the notion of solvability to the pair pattern calculus. When should a pair be considered as meaningful? At least two choices are possible: the *lazy* semantics considers a pair as meaningful in itself, the *strict* one requires both of its components to be meaningful. The first choice is adopted in this paper, since being a pair is already an observable property, particularly sufficient to unblock an explicit matching, independently from the observability of its components.

Thus, a term $\mathtt{t}$ is said to be **observable** iff there is a head-context $\mathtt{C}$ such that $\mathtt{C}[\mathtt{t}]$ reduces to a pair, *i.e.* $\mathtt{C}[\mathtt{t}] \to^* \langle \mathtt{t}_1, \mathtt{t}_2\rangle$, for some terms $\mathtt{t}_1, \mathtt{t}_2 \in \Lambda_{\mathtt{p}}$. Thus for example, the term $\langle \Delta\Delta, \Delta\Delta\rangle$, consisting of a pair of unsolvable terms $\Delta\Delta$, is observable. This notion of observability turns out to be conservative with respect to that of solvability for the λ-calculus (see Theorem 23).

3 The Type System $\mathcal{P}$

In this section we present a type system for the $\Lambda_{\mathtt{p}}$-calculus, and we show that it characterizes terms having canonical form.

The set $\mathcal{T}$ of types is generated by the following grammar:

$$\begin{array}{lcll} \alpha & ::= & o \mid \times_1(\tau) \mid \times_2(\tau) & \text{(product types)} \\ \sigma, \tau, \pi, \rho & ::= & \alpha \mid \mathtt{A} \to \sigma & \text{(strict types)} \\ \mathtt{B} & ::= & [\sigma_i]_{i\in I}\ (I \neq \emptyset) & \text{(non-empty multiset types)} \\ \mathtt{A} & ::= & [\,] \mid \mathtt{B} & \text{(multiset types)} \end{array}$$

where I is a finite set of indices. The arrow constructor is right associative. We consider a unique type constant o, which can be assigned to any pair.

We write $\mathtt{supp}(\mathtt{A})$ to denote the *support set* of the multiset $\mathtt{A}$, $\sqcup$ for multiset union and $\in$ to denote multiset membership. The **product** operation $\mathbb{X}$ on multisets is defined as follows:

$$\begin{array}{rcrcll} [\,] & \mathbb{X} & [\,] & := & [o] & \\ [\sigma_i]_{i\in I} & \mathbb{X} & [\rho_j]_{j\in J} & := & [\times_1(\sigma_i)]_{i\in I} \sqcup [\times_2(\rho_j)]_{j\in J} & \text{if } I \neq \emptyset \text{ or } J \neq \emptyset \end{array}$$

Remark that $\sqcup_{i\in I}\mathtt{A}_i \mathbb{X} \sqcup_{i\in I} \mathtt{A}'_i \sqsubseteq \sqcup_{i\in I}(\mathtt{A}_i \mathbb{X} \mathtt{A}'_i)$, the multiset inclusion being strict for example in the following case: $([\,] \sqcup [\,])\mathbb{X}([\,] \sqcup [\,]) = [o] \sqsubset [o, o] = ([\,]\mathbb{X}[\,]) \sqcup ([\,]\mathbb{X}[\,])$.

The **structure** of a pattern describes its shape, it is defined as follows:

$$\begin{array}{rcl} \mathcal{S}(\mathtt{x}) & := & [\,] \\ \mathcal{S}(\langle \mathtt{p}_1, \mathtt{p}_2\rangle) & := & \mathcal{S}(\mathtt{p}_1)\mathbb{X}\mathcal{S}(\mathtt{p}_2) \end{array}$$

E.g. $\mathcal{S}(\langle \mathtt{x}, \mathtt{y}\rangle) = [o]$, $\mathcal{S}(\langle \mathtt{x}, \langle \mathtt{y}, \mathtt{z}\rangle\rangle) = [\times_2(o)]$ and $\mathcal{S}(\langle\langle \mathtt{x}, \mathtt{w}\rangle, \langle \mathtt{y}, \mathtt{z}\rangle\rangle) = [\times_1(o), \times_2(o)]$. Notice that $\mathcal{S}(\mathtt{p})$ is nothing but a description of $\mathtt{p}$ seen as a binary tree whose leaves are distinct variables, and whose nodes are labeled by the pair constructor. Indeed, each element of $\mathcal{S}(\mathtt{p})$ specifies a maximal branch of such a tree, *i.e.* a branch whose last node is a pair constructor, and whose children are both leaves (*i.e.* variables). $\mathcal{S}(\mathtt{p})$ should be understood as the multiset

$$\frac{}{\mathtt{x}:\mathtt{B}\Vdash\mathtt{x}:\mathtt{B}}\ (\mathtt{varpat}) \qquad \frac{}{\Vdash\mathtt{x}:[\,]}\ (\mathtt{weakpat}) \qquad \frac{\Gamma\Vdash\mathtt{p}:\mathtt{A}_1 \quad \Delta\Vdash\mathtt{q}:\mathtt{A}_2 \quad \mathtt{p}\#\mathtt{q}}{\Gamma+\Delta\Vdash\langle\mathtt{p},\mathtt{q}\rangle:\mathtt{A}_1\mathbb{X}\mathtt{A}_2}\ (\mathtt{pairpat})$$

$$\frac{}{\mathtt{x}:[\pi]\vdash\mathtt{x}:\pi}\ (\mathtt{var}) \qquad \frac{\Gamma\vdash\mathtt{t}:\pi \quad \Gamma|_{\mathtt{p}}\Vdash\mathtt{p}:[\sigma_i]_{i\in I} \quad (\sigma_j\in\mathcal{S}(\mathtt{p}))_{j\in J} \quad I\cap J=\emptyset}{\Gamma\setminus\Gamma|_{\mathtt{p}}\vdash\lambda\mathtt{p}.\mathtt{t}:[\sigma_k]_{k\in I\cup J}\to\pi}\ (\to\mathtt{i})$$

$$\frac{\Gamma\vdash\mathtt{t}:[\sigma_i]_{i\in I}\to\pi \quad (\Delta_i\vdash\mathtt{u}:\sigma_i)_{i\in I}}{\Gamma+_{i\in I}\Delta_i\vdash\mathtt{tu}:\pi}\ (\to\mathtt{e})$$

$$\frac{}{\vdash\langle\mathtt{t},\mathtt{u}\rangle:o}\ (\mathtt{emptypair}) \qquad \frac{\Gamma\vdash\mathtt{t}:\sigma}{\Gamma\vdash\langle\mathtt{t},\mathtt{u}\rangle:\times_1(\sigma)}\ (\mathtt{pair1}) \qquad \frac{\Gamma\vdash\mathtt{u}:\tau}{\Gamma\vdash\langle\mathtt{t},\mathtt{u}\rangle:\times_2(\tau)}\ (\mathtt{pair2})$$

$$\frac{\Gamma\vdash\mathtt{t}:\sigma \quad \Gamma|_{\mathtt{p}}\Vdash\mathtt{p}:[\sigma_i]_{i\in I} \quad (\sigma_j\in\mathcal{S}(\mathtt{p}))_{j\in J} \quad (\Delta_k\vdash\mathtt{u}:\sigma_k)_{k\in I\cup J} \quad I\cap J=\emptyset}{(\Gamma\setminus\Gamma|_{\mathtt{p}})+_{k\in I\cup J}\Delta_k\vdash\mathtt{t}[\mathtt{p}/\mathtt{u}]:\sigma}\ (\mathtt{sub})$$

Figure 1 The type assignment system $\mathcal{P}$.

of *non depletable* resources associated with $\mathtt{p}$; the persistent character of these resources is highlighted in the forthcoming typing system.

Typing environments, written Γ,Δ, are functions from variables to multiset types, assigning the empty multiset to almost all the variables. The **domain** of Γ, written $\mathtt{dom}(\Gamma)$, is the set of variables whose image is different from $[\,]$. We write $\Gamma\#\Delta$ iff $\mathtt{dom}(\Gamma)\cap\mathtt{dom}(\Delta)=\emptyset$.

▶ Notation 2. Given the environments $\{\Gamma_i\}_{i\in I}$, we write $+_{i\in I}\Gamma_i$ for the environment which maps $\mathtt{x}$ to $\sqcup_{i\in I}\Gamma_i(\mathtt{x})$. If $I=\emptyset$, the resulting environment is the one having an empty domain. Note that $\Gamma+\Delta$ and $\Gamma+_{i\in I}\Delta_i$ are just particular cases of the previous general definition. When $\Gamma\#\Delta$ we write $\Gamma;\Delta$ instead of $\Gamma+\Delta$. We write $\Gamma\setminus\mathtt{x}$ for the environment assigning $[\,]$ to $\mathtt{x}$, and acting as Γ otherwise; $\mathtt{x}_1:\mathtt{A}_1;\ldots;\mathtt{x}_n:\mathtt{A}_n$ is the environment assigning $\mathtt{A}_i$ to $\mathtt{x}_i$, for $1\leq i\leq n$, and $[\,]$ to any other variable; $\Gamma|_{\mathtt{p}}$ denotes the environment such that $\Gamma|_{\mathtt{p}}(\mathtt{x})=\Gamma(\mathtt{x})$, if $\mathtt{x}\in\mathtt{fv}(\mathtt{p})$, $[\,]$ otherwise.

The **type assignment system** $\mathcal{P}$ (see Fig. 1) is a set of typing rules assigning strict types of $\mathcal{T}$ to terms of $\Lambda_{\mathtt{p}}$. We write $\Pi\triangleright\Gamma\vdash\mathtt{t}:\sigma$ (resp. $\Pi\triangleright\Gamma\Vdash\mathtt{p}:\mathtt{A}$) to denote a **typing derivation** ending in the sequent $\Gamma\vdash\mathtt{t}:\sigma$ (resp. $\Gamma\Vdash\mathtt{p}:\mathtt{A}$), in which case $\mathtt{t}$ (resp. $\mathtt{p}$) is called the **subject** of Π; by abuse of notation, $\Gamma\vdash\mathtt{t}:\sigma$ (resp. $\Gamma\Vdash\mathtt{p}:\mathtt{A}$) also denotes the existence of some typing derivation ending in this sequent, in which case $\mathtt{t}$ (resp. $\mathtt{p}$) is said to be **typable**. The **measure** of a typing derivation Π, written $\mathtt{meas}(\Pi)$, is the number of typing rules in Π.

Rules ($\mathtt{var}$) and ($\to\mathtt{e}$) are those used for λ-calculus in [5, 8]. Linearity of patterns is guaranteed by the clause $\mathtt{p}\#\mathtt{q}$ in rule ($\mathtt{pairpat}$). Rule ($\mathtt{weakpat}$) is essential to type *erasing* functions such as for example $\lambda\mathtt{x}.\mathtt{Id}$. The rule ($\mathtt{emptypair}$) types for example $\langle\Delta\Delta,\Delta\Delta\rangle$, and thus $(\lambda\langle\mathtt{x},\mathtt{y}\rangle.\mathtt{Id})\langle\Delta\Delta,\Delta\Delta\rangle$. Rules ($\mathtt{pair1}$) and ($\mathtt{pair2}$) type pairs having just one typed component, whereas standard typed calculi with pairs (e.g. [6]) requires both components to be typed. This is necessary to type terms like $(\lambda\langle\mathtt{x},\mathtt{y}\rangle.\mathtt{x})\langle\mathtt{Id},\Delta\Delta\rangle$. Moreover, the standard policy can be easily recovered from ours by typing a pair whose components are both typed using ($\mathtt{pair1}$) and ($\mathtt{pair2}$) successively.

The rules ($\to$ i) and (sub) are the most subtle ones [1]. Here is where the structural types come into play: they can be used *ad libitum* (whence the notation $(\sigma_j \in \mathcal{S}(\mathtt{p}))_{j\in J}$), thanks to *non depletable* nature of the information provided by the structure of patterns (whereas the type information of variables should be understood as *depletable*). Concerning more specifically the rule (sub): in order to type $\mathtt{t}[\mathtt{p}/\mathtt{u}]$, on one hand we need to type t and on the other one we need to check that p and u can be assigned the same types. Since the system is relevant, we need to collect the environments used in all the premises typing p and u. Remark however that there is a lack of symmetry between patterns and terms: while the only information we can use about terms is the one concerning their types, a pattern p has not only a type (description of its depletable resources), but also an intrinsic shape that is completely described by the structural (non depletable) types in the set $\mathcal{S}(\mathtt{p})$.

Actually the structural information on patterns is necessary, in particular, to guarantee subject reduction for rule (r_5). Indeed, given $\mathtt{t} = \lambda\mathtt{w}.(\mathtt{z}\mathtt{z}')[\langle \mathtt{z},\mathtt{z}'\rangle/(\mathtt{y}\mathtt{x})[\langle \mathtt{x},\mathtt{x}'\rangle/\mathtt{w}]] \to_{r_5} \lambda\mathtt{w}.(\mathtt{z}\mathtt{z}')[\langle \mathtt{z},\mathtt{z}'\rangle/(\mathtt{y}\mathtt{x})][\langle \mathtt{x},\mathtt{x}'\rangle/\mathtt{w}] = \mathtt{t}'$, and $\Gamma = \mathtt{y} : [[\,] \to \times_1(\tau), [\pi] \to \times_2(\sigma)]$, we have that $\Gamma \vdash \mathtt{t} : [o, \times_1(\pi)] \to \sigma$, but $\Gamma \vdash \mathtt{t}' : [o, \times_1(\pi)] \to \sigma$ holds only by using the fact that $o \in \mathcal{S}(\langle \mathtt{x},\mathtt{x}'\rangle)$. This counterexample shows that a clear tension appears between the rewriting rule (r_5) and the use of the structural set $\mathcal{S}(\mathtt{p})$ in the typing rules ($\to$ i) and (sub). Eliminating (r_5) from the reduction system would certainly simplify the typing system, but would significantly complicate the inhabitation algorithm that will be presented in Sec. 5.

▶ **Example 3.** The following (partially described) derivation is valid:

$$\dfrac{\begin{array}{c}(a)\ \mathtt{x} : [\alpha] \vdash \mathtt{x} : \alpha \qquad (b)\ \mathtt{x} : [\alpha] \Vdash \langle \mathtt{x},\mathtt{y}\rangle : [\times_1(\alpha)] \\ (c)\ (o \in \mathcal{S}(\langle \mathtt{x},\mathtt{y}\rangle)) \qquad (d)\ \mathtt{z} : [o] \vdash \mathtt{z} : o \qquad (e)\ \mathtt{z} : [\times_1(\alpha)] \vdash \mathtt{z} : \times_1(\alpha)\end{array}}{\mathtt{z} : [o, \times_1(\alpha)] \vdash \mathtt{x}[\langle \mathtt{x},\mathtt{y}\rangle/\mathtt{z}] : \alpha}\ (\mathtt{sub})$$

Using only the hypothesis (a), (b) and (e) we get another valid typing derivation ending in $\mathtt{z} : [\times_1(\alpha)] \vdash \mathtt{x}[\langle \mathtt{x},\mathtt{y}\rangle/\mathtt{z}] : \alpha$ which does not use structural information about the pattern $\langle \mathtt{x},\mathtt{y}\rangle$.

The system is relevant, in the sense that only the used premises are registered in the typing environments. This property, formally stated in the following lemma, will be an important technical tool used to develop the inhabitation algorithm.

▶ **Lemma 4** (Relevance)**.**

- *If* $\Gamma \Vdash \mathtt{p} : \mathtt{A}$*, then* $\mathtt{dom}(\Gamma) \subseteq \mathtt{fv}(\mathtt{p})$.
- *If* $\Gamma \vdash \mathtt{t} : \sigma$*, then* $\mathtt{dom}(\Gamma) \subseteq \mathtt{fv}(\mathtt{t})$.

Proof. By induction on the typing derivations. ◀

Some useful properties will be needed in the sequel. In particular, the next technical lemma says that, given different types $\mathtt{A}_i$ for a given pattern p, it is always possible to split $\sqcup_{i\in I}\mathtt{A}_i$ into a bunch of resource types A and another one of structural types $\mathtt{A}'$.

▶ **Lemma 5.** *Let* $I \neq \emptyset$*. If* $(\Gamma_i \Vdash \mathtt{p} : \mathtt{A}_i)_{i\in I}$*, then there exist* $\mathtt{A}, \mathtt{A}'$ *such that*

1. $\mathtt{A} \sqcup \mathtt{A}' = \sqcup_{i\in I}\mathtt{A}_i$,
2. $+_{i\in I}\Gamma_i \Vdash \mathtt{p} : \mathtt{A}$

[1] Notice that "Γ", "$\Gamma|_p$" and "$\Gamma \setminus \Gamma|_p$" in rules ($\to$ i) and (sub) could be replaced by "$\Gamma_1;\Gamma_2$", "Γ_2" and "Γ_1", respectively, only if $\mathtt{dom}(\Gamma_1) \cap \mathtt{fv}(\mathtt{p}) = \emptyset$. Otherwise, for instance, $\lambda\mathtt{x}.\mathtt{x}$ would be typable with type $[\,] \to \sigma$.

3. $\mathtt{A} = [\,]$ *implies* $\mathtt{A}' = [\,]$,
4. $\mathtt{supp}(\mathtt{A}') \subseteq \mathcal{S}(\mathtt{p})$,
5. $\mathtt{meas}(+_{i\in I}\Gamma_i \Vdash \mathtt{p} : \mathtt{A}) \leq \Sigma_{i\in I}\mathtt{meas}(\Gamma_i \Vdash \mathtt{p} : \mathtt{A}_i)$.

Proof. By induction on $\mathtt{p}$. ◀

The following lemma can be shown by induction on typing derivations; it is used in the forthcoming subject reduction property.

▶ **Lemma 6** (Substitution Lemma). *If* $\Pi \rhd \Gamma; \mathtt{x} : [\rho_i]_{i\in I} \vdash \mathtt{t} : \tau$, *and* $(\Theta_i \rhd \Delta_i \vdash \mathtt{u} : \rho_i)_{i\in I}$ *then* $\Pi' \rhd \Gamma +_{i\in I} \Delta_i \vdash \mathtt{t}\{\mathtt{x}/\mathtt{u}\} : \tau$ *where* $\mathtt{meas}(\Pi') < \mathtt{meas}(\Pi) + \sum_{i\in I}\mathtt{meas}(\Theta_i)$.

Notice that, in the process of assigning a type to a term $\mathtt{t}$, some subterms of $\mathtt{t}$ may be left untyped. Typically, this happens when $\mathtt{t}$ contains occurrences of non typable terms, like in $\lambda\mathtt{x}.\mathtt{x}(\Delta\Delta)$. We are then going to define the notion of **typed occurrence** of a typing derivation, which plays an essential role in the rest of this paper: indeed, thanks to the use of non-idempotent intersection types, a combinatorial argument based on a measure on typing derivations (*cf.* Lem. 9.1), allows to prove the termination of reduction of redexes occurring in typed occurrences of their respective typing derivations.

Let us then define an **occurrence** of a subterm $\mathtt{u}$ in a term $\mathtt{t}$ as a context $\mathtt{C}$ such that $\mathtt{C}[\mathtt{u}] = \mathtt{t}$. Then, given a typing derivation $\Pi \rhd \Gamma \vdash \mathtt{t} : \sigma$, an occurrence of a subterm of $\mathtt{t}$ is a typed occurrence of Π if and only if it is the subject of a subderivation of Π. More precisely:

▶ **Definition 7.** Given a type derivation Π, the set of **typed occurrences** of Π, written $\mathtt{toc}(\Pi)$, by induction on the last rule of Π.

- If Π ends with (var), then $\mathtt{toc}(\Pi) := \{\Box\}$.
- If Π ends with (pair1) with subject $\langle \mathtt{u}, \mathtt{v}\rangle$ and premise Π', then $\mathtt{toc}(\Pi) := \{\Box\} \cup \{\langle \mathtt{C}, \mathtt{v}\rangle \mid \mathtt{C} \in \mathtt{toc}(\Pi')\}$.
- If Π ends with (pair2) with subject $\langle \mathtt{u}, \mathtt{v}\rangle$ and premise Π' then $\mathtt{toc}(\Pi) := \{\Box\} \cup \{\langle \mathtt{u}, \mathtt{C}\rangle \mid \mathtt{C} \in \mathtt{toc}(\Pi')\}$.
- If Π ends with $(\to \mathtt{i})$ with subject $\lambda\mathtt{p}.\mathtt{u}$ and premise Π' then $\mathtt{toc}(\Pi) := \{\Box\} \cup \{\lambda\mathtt{p}.\mathtt{C} \mid \mathtt{C} \in \mathtt{toc}(\Pi')\}$.
- If Π ends with $(\to \mathtt{e})$ with subject $\mathtt{tu}$ and premises Π_1 and Π_k $(k \in K)$ with subjects $\mathtt{t}$ and $\mathtt{u}$ respectively, then $\mathtt{toc}(\Pi) := \{\Box\} \cup \{\mathtt{tC} \mid \mathtt{C} \in \mathtt{toc}(\Pi_k), k \in K\} \cup \{\mathtt{Cu} \mid \mathtt{C} \in \mathtt{toc}(\Pi_1)\}$.
- If Π ends with (sub) with subject $\mathtt{t}[\mathtt{p}/\mathtt{u}]$ and premises Π_1 and Π_k $(k \in K)$ with subjects $\mathtt{t}$ and $\mathtt{u}$ respectively, then $\mathtt{toc}(\Pi) := \{\Box\} \cup \{\mathtt{C}[\mathtt{p}/\mathtt{u}] \mid \mathtt{C} \in \mathtt{toc}(\Pi_1)\} \cup \{\mathtt{t}[\mathtt{p}/\mathtt{C}] \mid \mathtt{C} \in \mathtt{toc}(\Pi_k), k \in K\}$.

▶ **Example 8.** Given the following derivations Π and Π', the occurrences $\Box$ and $\Box\mathtt{y}$ belong to both $\mathtt{toc}(\Pi)$ and $\mathtt{toc}(\Pi')$ while $\mathtt{x}\Box$ belongs to $\mathtt{toc}(\Pi)$ but not to $\mathtt{toc}(\Pi')$.

$$\Pi \rhd \frac{\mathtt{x} : [[\tau] \to \tau] \vdash \mathtt{x} : [\tau] \to \tau \qquad \mathtt{y} : [\tau] \vdash \mathtt{y} : \tau}{\mathtt{x} : [[\tau] \to \tau], \mathtt{y} : [\tau] \vdash \mathtt{xy} : \tau} \qquad \Pi' \rhd \frac{\mathtt{x} : [[\,] \to \tau] \vdash \mathtt{x} : [\,] \to \tau}{\mathtt{x} : [[\,] \to \tau] \vdash \mathtt{xy} : \tau}$$

Given $\Pi \rhd \Gamma \vdash \mathtt{t} : \tau$, $\mathtt{t}$ is said to be in **Π-normal form**, also written **Π-nf**, if for every typed occurrence $\mathtt{C} \in \mathtt{toc}(\Pi)$ such that $\mathtt{t} = \mathtt{C}[\mathtt{u}]$, the subterm $\mathtt{u}$ is not a redex.

The system $\mathcal{P}$ enjoys both subject reduction and subject expansion. In particular, thanks to the use of multisets, subject reduction decreases the measure of the derivation, in case a substitution is performed by rule (r_2) and the redex is typed. This property allows for a simple proof of the "only if" part of the characterization theorem.

▶ **Lemma 9.**

1. **(Weighted Subject Reduction)** *If* $\Pi \triangleright \Gamma \vdash \mathtt{t} : \tau$ *and* $\mathtt{t} \to \mathtt{v}$*, then* $\Pi' \triangleright \Gamma \vdash \mathtt{v} : \tau$ *and* $\mathtt{meas}(\Pi') \leq \mathtt{meas}(\Pi)$*. Moreover, if the reduced redex is* (r_2) *and it occurs in a typed occurrence of* Π*, then* $\mathtt{meas}(\Pi') < \mathtt{meas}(\Pi)$*.*
2. **(Subject Expansion)** *If* $\Gamma \vdash \mathtt{v} : \sigma$ *and* $\mathtt{t} \to \mathtt{v}$*, then* $\Gamma \vdash \mathtt{t} : \sigma$*.*

Proof. 1. By induction on $\mathtt{t} \to \mathtt{v}$ using Lemmas 5, 6 and 4.
2. By induction on $\mathtt{t} \to \mathtt{v}$.

◀

We are now ready to provide the logical characterization of terms having canonical form.

▶ **Theorem 10** (Characterization)**.** *A term* $\mathtt{t}$ *is typable iff* $\mathtt{t}$ *has a canonical form.*

Proof. ▬ (if) We reason by induction on the grammar defining the canonical forms. We first prove that for all type σ and for all $\mathcal{K}$-canonical form $\mathtt{t}$, $\mathtt{t}$ can be typed by σ. In fact every $\mathcal{K}$-canonical form is of the shape $\mathtt{x}\mathtt{t}_1 ... \mathtt{t}_n$, for $n \geq 0$. It is easy to check that $\mathtt{x} : \underbrace{[] \to ... \to []}_{n} \to \sigma \vdash \mathtt{x}\mathtt{t}_1 ... \mathtt{t}_n : \sigma$. Let $\mathtt{t}$ be a $\mathcal{J}$-canonical form. If $\mathtt{t} = \langle \mathtt{u}, \mathtt{v} \rangle$ then by rule $(\mathtt{emptypair}) \vdash \langle \mathtt{u}, \mathtt{v} \rangle : o$. If $\mathtt{t} = \lambda \mathtt{p}.\mathtt{u}$, then by induction $\mathtt{u}$ can be typed and the result follows from rule $(\to \mathtt{I})$. Let $\mathtt{t} = \mathtt{t}'[\langle \mathtt{p}, \mathtt{q} \rangle / \mathtt{v}]$, where $\mathtt{t}'$ (resp. $\mathtt{v}$) is a $\mathcal{J}$ (resp. $\mathcal{K}$) canonical form. By the *i.h.* there are Γ, σ such that $\Gamma \vdash \mathtt{t}' : \sigma$. Moreover, it is easy to see that $\Gamma|_{\langle \mathtt{p},\mathtt{q} \rangle} \Vdash \langle \mathtt{p}, \mathtt{q} \rangle : [\sigma_i]_{i \in I}$, for some $[\sigma_i]_{i \in I}$. Since $\mathtt{v}$ is a $\mathcal{K}$-canonical form, then $\Delta_i \vdash \mathtt{v} : \sigma_i$ for all $i \in I$, as shown above. Thus $\Gamma +_{i \in I} \Delta_i \vdash \mathtt{t}'[\langle \mathtt{p}, \mathtt{q} \rangle / \mathtt{v}] : \sigma$ by rule $(\mathtt{sub})$ with $J = \emptyset$.

▬ (only if) Let $\mathtt{t}$ be a typable term, *i.e.* $\Pi \triangleright \Gamma \vdash \mathtt{t} : \sigma$. Consider a reduction strategy $\mathcal{ST}$ that always chooses a *typed* redex occurrence. By Lem. 9.1 and Lem. 1.2 the strategy $\mathcal{ST}$ always terminates. Let $\mathtt{t}'$ be a normal-form of $\mathtt{t}$ for the strategy $\mathcal{ST}$, *i.e.* $\mathtt{t}$ reduces to $\mathtt{t}'$ using $\mathcal{ST}$, and $\mathcal{ST}$ applied to $\mathtt{t}'$ is undefined. We know that $\Pi' \triangleright \Gamma \vdash \mathtt{t}' : \sigma$ by Lem. 9.1. Then, by definition of $\mathcal{ST}$, $\mathtt{t}'$ has no typed redex occurrence. A simple induction on $\mathtt{t}'$ allows to conclude that it is a canonical form. ◀

4 From canonicity to observability

We proved in the previous section that system $\mathcal{P}$ gives a complete characterization of terms having canonical forms. The next theorem proves that system $\mathcal{P}$ is complete with respect to observability.

▶ **Theorem 11.** *Observability implies typability.*

Proof. If $\mathtt{t}$ is observable, then there is a head context $\mathtt{C}$ such that $\mathtt{C}[\mathtt{t}]$ reduces to $\langle \mathtt{u}, \mathtt{v} \rangle$, for some $\mathtt{u}$ and $\mathtt{v}$. Since all pairs are typable, the term $\mathtt{C}[\mathtt{t}]$ is typable by Lem. 9.2. Remember that $\mathtt{C}[\mathtt{t}] = (\lambda \mathtt{p}_1 \ldots \mathtt{p}_n.\mathtt{t})\mathtt{t}_1 ... \mathtt{t}_m$ so that $\mathtt{t}$ is typable too, by easy inspection of the typing system. ◀

Unfortunately, soundness does not hold, *i.e.* the set of observable terms is strictly included in the set of terms having canonical form, as shown below.

▶ **Example 12.** The term $\mathtt{t}_1 = \lambda \mathtt{x}.\mathtt{Id}[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{x}][\langle \mathtt{y}', \mathtt{z}' \rangle / \mathtt{xId}]$ is canonical, hence typable (by Thm. 10), but not observable. In fact, it is easy to see that there is no term $\mathtt{u}$ such that both $\mathtt{u}$ and $\mathtt{uId}$ reduce to pairs. A less trivial example is the term $\mathtt{t}_2 = \lambda \mathtt{x}.\mathtt{Id}[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{x}\langle \mathtt{Id}, \mathtt{Id} \rangle][\langle \mathtt{y}', \mathtt{z}' \rangle / \mathtt{xIdId}]$, which is canonical, hence typable, but not observable, as proved in the next lemma.

▶ **Lemma 13.** *There is no closed term* $\mathtt{u}$ *s.t. both* $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ *and* $\mathtt{uIdId}$ *reduce to pairs.*

Proof. By contradiction. Indeed, assume that there exist a closed term $\mathtt{u}$ such that both $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ and $\mathtt{uIdId}$ reduce to pairs. Since pairs are always typable, then $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ and $\mathtt{uIdId}$ are typable by Lem. 9.2. In any of the typing derivations of such terms, $\mathtt{u}$ occurs in a typed position, so that $\mathtt{u}$ turns out to be also typable.
Now, since $\mathtt{u}$ is typable and closed, then it reduces to a (typable and closed) canonical form $\mathtt{v} \in \mathcal{J}$ by Thm. 10. But $\mathtt{v}$ cannot be in $\mathcal{K}$, which only contains open terms. Moreover, $\mathtt{v}$ cannot be a pair, otherwise $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle \to^* \mathtt{v}\langle \mathtt{Id}, \mathtt{Id}\rangle \to^* \langle \mathtt{v}_1, \mathtt{v}_2\rangle\langle \mathtt{Id}, \mathtt{Id}\rangle \to^* \mathtt{fail}$ which contradicts (by Lem. 1) the fact that $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ reduces to a pair. We then have two possible forms for $\mathtt{v}$.
If $\mathtt{v} = \mathtt{s}[\langle \mathtt{p}_1, \mathtt{p}_2\rangle/\mathtt{k}]$, where $\mathtt{s} \in \mathcal{J}$ and $\mathtt{k} \in \mathcal{K}$. Then $\mathtt{k}$ is an open term which implies $\mathtt{v}$ is an open term. Contradiction.
If $\mathtt{v} = \lambda\mathtt{p}.\mathtt{s}$, where $\mathtt{s} \in \mathcal{J}$, then $\mathtt{p}$ is necessarily a variable, say $\mathtt{z}$, since otherwise $\mathtt{vId}$ reduces to $\mathtt{fail}$, and hence $\mathtt{uIdId} \to^* \mathtt{vIdId} \to^* \mathtt{fail}$, which contradicts (by Lem. 1) the fact that $\mathtt{uIdId}$ reduces to a pair. We analyze the possible forms of $\mathtt{s}$.

- If $\mathtt{s}$ is a pair, then $\mathtt{uIdId} \to^* (\lambda\mathtt{z}.\mathtt{s})\mathtt{IdId} \to^* \mathtt{fail}$, which contradicts (by Lem. 1) the fact that $\mathtt{uIdId}$ reduces to a pair.
- If $\mathtt{s}$ is an abstraction, then $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle \to^* (\lambda\mathtt{z}.\mathtt{s})\langle \mathtt{Id}, \mathtt{Id}\rangle$ which reduces to an abstraction, contradicting (by Lem. 1) the fact that $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ reduces to a pair.
- If $\mathtt{s}$ is in $\mathcal{K}$, then $\mathtt{s} = \mathtt{x}\mathtt{t}_1 \dots \mathtt{t}_n$ with $n \geq 0$. Remark that $\mathtt{z} \neq \mathtt{x}$ is not possible since $\mathtt{v} = \lambda\mathtt{z}.\mathtt{s}$ is closed. Then $\mathtt{z} = \mathtt{x}$. If $\mathtt{s} = \mathtt{z}$, then $\mathtt{uIdId}$ reduces to $\mathtt{Id}$ which contradicts (by Lem. 1) the fact that $\mathtt{uIdId}$ reduces to a pair. Otherwise, $\mathtt{s} = \mathtt{z}\mathtt{t}_1 \dots \mathtt{t}_n$ with $n \geq 1$, and thus $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ reduces to $\langle \mathtt{Id}, \mathtt{Id}\rangle\mathtt{t}_1 \dots \mathtt{t}_n \to^* \mathtt{fail}$, which contradicts again (by Lem. 1) the fact that $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ reduces to a pair.
- If $\mathtt{s}$ is $\mathtt{s}'[\langle \mathtt{p}_1, \mathtt{p}_2\rangle/\mathtt{k}]$, with $\mathtt{k} \in \mathcal{K}$, then $\mathtt{k} = \mathtt{z}\mathtt{t}_1 \dots \mathtt{t}_n$ with $n \geq 0$, since any other head variable for $\mathtt{k}$ would contradict $\mathtt{v}$ closed. Now, in the first case we have $\mathtt{uIdId}$ reduces to $\mathtt{fail}$ which contradicts (by Lem. 1) the fact that $\mathtt{uIdId}$ reduces to a pair. Otherwise, $\mathtt{k} = \mathtt{z}\mathtt{t}_1 \dots \mathtt{t}_n$ with $n \geq 1$ implies $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ reduces to $\mathtt{fail}$ which contradicts (by Lem. 1) the fact that $\mathtt{u}\langle \mathtt{Id}, \mathtt{Id}\rangle$ reduces to a pair. ◀

The first non-observable term $\mathtt{t}_1$ in Ex. 12 could be ruled out by introducing a notion of *compatibility* between types and requiring multiset types to be composed only by compatible strict types. Unfortunately, we claim that a compatibility relation defined *syntactically*, let us call it $\mathtt{comp}$, cannot lead to a sound and complete characterization of observability. By "defined syntactically" we mean that the value of $\mathtt{comp}(\sigma \to \sigma', \rho \to \rho')$ should only depend on the values of $\mathtt{comp}(\sigma, \rho)$ and $\mathtt{comp}(\sigma', \rho')$. Another basic requirement of $\mathtt{comp}$ would be that every product type is incompatible with any functional type. The second non-observable term $\mathtt{t}_2$ in Ex. 12 is appropriate to illustrate our claim, by keeping in mind that any pair of types assignable to $\mathtt{x}$ in any typing derivation for $\mathtt{t}_2$ need to be incompatible.

Indeed, the shortest typing for $\mathtt{t}_2$ above is obtained by assigning to $\mathtt{x}$ the two types $[\,] \to o$ and $[\,] \to [\,] \to o$, and in order to state the incompatibility between them it would be necessary to define that $\mathtt{comp}(\sigma, \rho)$ and $\neg\mathtt{comp}(\sigma', \rho')$ imply $\neg\mathtt{comp}([\sigma] \to \sigma', [\rho] \to \rho')$. Another typing for $\mathtt{t}_2$ is obtained by assigning to $\mathtt{x}$ the two types $[o] \to o$ and $[\tau] \to [\tau] \to o$ respectively, where $\tau = [o] \to o$, so that $\neg\mathtt{comp}(\sigma, \rho)$ and $\neg\mathtt{comp}(\sigma', \rho')$ should imply $\neg\mathtt{comp}([\sigma] \to \sigma', [\rho] \to \rho')$. We conclude that $\neg\mathtt{comp}(\sigma', \rho')$ alone should imply $\neg\mathtt{comp}([\sigma] \to \sigma', [\rho] \to \rho')$. However, arrow types $[\sigma] \to \sigma'$ and $[\rho] \to \rho'$ having incompatible right-hand sides may very well be compatible. For instance, letting $\sigma = \sigma' = o$ and $\rho = \rho' = [o] \to o$, one gets two types for $\mathtt{Id}$

which need of course to be compatible. Hence, a *syntactic* characterization of such a notion of compatibility seems out of reach.

Fortunately, there exists a sound and complete *semantical* notion of compatibility between types, obtained *a posteriori* as follows: given two strict types π_1 and π_2, build the corresponding sets of inhabitants $\mathtt{T}(\emptyset, \pi_1)$ and $\mathtt{T}(\emptyset, \pi_2)$, using the inhabitation algorithm presented in Sec. 5. Then π_1 and π_2 are *semantically* compatible if and only if $\mathtt{T}(\emptyset, \pi_1) \cap \mathtt{T}(\emptyset, \pi_2)$ is non-empty.

While the inhabitation problem for (idempotent) intersection types is undecidable [17], it becomes decidable for non-idempotent intersection types [5], which is just a subsystem of our typing system $\mathcal{P}$ introduced in Sec. 3. We will prove in the following that inhabitation is also decidable for the non-trivial extension $\mathcal{P}$. We will then use this result for characterizing observability in the pattern calculus without referring to a complete syntactic characterization, which is not possible in this framework, as illustrated by Example 12.

5 Inhabitation for System $\mathcal{P}$

We now show a sound and complete algorithm to solve the inhabitation problem for System $\mathcal{P}$. Given a strict type σ, the inhabitation problem consists in finding a closed term $\mathtt{t}$ such that $\vdash \mathtt{t} : \sigma$ is derivable. We extend the problem to multiset types by defining $\mathtt{A}$ to be inhabited if and only if there is a closed term $\mathtt{t}$ such that $\vdash \mathtt{t} : \sigma_i$ for every $\sigma_i \in \mathtt{A}$. These notions will naturally be generalized later to non-closed terms.

We already noticed that the system $\mathcal{P}$ allows to type terms containing untyped subterms through the rule $(\to \mathtt{e})$ with $I = \emptyset$ and the rule $(\mathtt{sub})$ with $I = J = \emptyset$. In order to identify inhabitants in such cases we introduce a term constant Ω to denote a generic untyped subterm. Our inhabitation algorithm produces **approximate normal forms** $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, also written anf, defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{a}, \mathtt{b}, \mathtt{c} & ::= & \Omega \mid \mathcal{N} & \mathcal{N} & ::= & \lambda \mathtt{p}.\mathcal{N} \mid \langle \mathtt{a}, \mathtt{b} \rangle \mid \mathcal{L} \mid \mathcal{N}[\langle \mathtt{p}, \mathtt{q} \rangle / \mathcal{L}] \\ & & & \mathcal{L} & ::= & \mathtt{x} \mid \mathcal{L}\mathtt{a} \end{array}$$

Note that $anfs$ do not contain redexes, differently from canonical forms. In particular, thanks to the reduction rule (r_4) (resp. (r_5)), they do not contain *head* (resp. *nested*) explicit matchings. This makes the inhabitation algorithm much more intuitive and simpler.

▶ **Example 14.** the term $\lambda\langle \mathtt{x}, \mathtt{y} \rangle.(\mathtt{x}(\mathtt{IdId}))[\langle \mathtt{z}_1, \mathtt{z}_2 \rangle / \mathtt{yId}]$ is canonical but not an anf, while $\lambda\langle \mathtt{x}, \mathtt{y} \rangle.(\mathtt{x}\Omega)[\langle \mathtt{z}_1, \mathtt{z}_2 \rangle / \mathtt{yId}]$ is an anf.

$Anfs$ are ordered by the smallest contextual order $\leq$ such that $\Omega \leq \mathtt{a}$, for any $\mathtt{a}$. We also write $\mathtt{a} \leq \mathtt{t}$ when the term $\mathtt{t}$ is obtained from $\mathtt{a}$ by replacing each occurrence of Ω by a term of $\Lambda_{\mathtt{p}}$: For example $\mathtt{x}\Omega\Omega \leq \mathtt{x}(\mathtt{Id}\Delta)(\Delta\Delta)$ is obtained by replacing the first (resp. second) occurrence of Ω by $\mathtt{Id}\Delta$ (resp. $\Delta\Delta$).

Let $\mathcal{A}(\mathtt{t}) = \{\mathtt{a} \mid \exists \mathtt{u}\ \mathtt{t} \to^* \mathtt{u} \text{ and } \mathtt{a} \leq \mathtt{u}\}$ be the set of **approximants** of the term $\mathtt{t}$, and let $\bigvee$ denote the least upper bound with respect to $\leq$. We write $\uparrow_{i \in I} \mathtt{a}_i$ to denote the fact that $\bigvee\{\mathtt{a}_i\}_{i \in I}$ does exist. It is easy to check that, for every $\mathtt{t}$ and $\mathtt{a}_1, \ldots \mathtt{a}_n \in \mathcal{A}(\mathtt{t})$, $\uparrow_{i \in \{1,\ldots,n\}} \mathtt{a}_i$. An anf $\mathtt{a}$ is a **head subterm** of $\mathtt{b}$ if either $\mathtt{b} = \mathtt{a}$ or $\mathtt{b} = \mathtt{cc}'$ and $\mathtt{a}$ is a head subterm of $\mathtt{c}$. System $\mathcal{P}$ can also be trivially extended to give types to $anfs$, simply assuming that no type can be assigned to the constant Ω. It is easy to check that, if $\Gamma \vdash \mathtt{a} : \sigma$ and $\mathtt{a} \leq \mathtt{b}$ (resp. $\mathtt{a} \leq \mathtt{t}$) then $\Gamma \vdash \mathtt{b} : \sigma$ (resp. $\Gamma \vdash \mathtt{t} : \sigma$).

Given $\Pi \triangleright \Gamma \vdash \mathtt{t} : \tau$, where $\mathtt{t}$ is in Π-nf (*cf.* Sec. 3), $\mathcal{A}(\Pi)$ is the minimal approximant $\mathtt{b}$ of $\mathtt{t}$ such that $\Pi \triangleright \Gamma \vdash \mathtt{b} : \tau$. Formally, given $\Pi \triangleright \Gamma \vdash \mathtt{t} : \sigma$, where $\mathtt{t}$ is in Π-nf, the **minimal approximant of** Π, written $\mathcal{A}(\Pi)$, is defined by induction on $\mathtt{meas}(\Pi)$ as follows:

- $\mathcal{A}(\Gamma \vdash \mathtt{x} : \rho) = \mathtt{x}$; $\mathcal{A}(\Gamma \vdash \langle \mathtt{t}, \mathtt{u} \rangle : o) = \langle \Omega, \Omega \rangle$.
- If $\Pi \triangleright \Gamma \vdash \lambda\mathtt{p}.\mathtt{t} : \mathtt{A} \to \rho$ follows from $\Pi' \triangleright \Gamma' \vdash \mathtt{t} : \rho$, then $\mathcal{A}(\Pi) = \lambda\mathtt{p}.\mathcal{A}(\Pi')$, $\mathtt{t}$ being in Π'-nf.
- If $\Pi \triangleright \Gamma \vdash \langle \mathtt{t}, \mathtt{u} \rangle : \times_1(\tau)$ follows from $\Pi' \triangleright \Gamma \vdash \mathtt{t} : \tau$, then $\mathcal{A}(\Pi) = \langle \mathcal{A}(\Pi'), \Omega \rangle$, $\mathtt{t}$ being in Π'-nf. Similarly for a pair of type $\times_2(\tau)$.
- If $\Pi \triangleright \Gamma = \Gamma' +_{i\in I} \Delta_i \vdash \mathtt{tu} : \rho$ follows from $\Pi' \triangleright \Gamma' \vdash \mathtt{t} : [\sigma_i]_{i\in I} \to \rho$ and $(\Pi'_i \triangleright \Delta_i \vdash \mathtt{u} : \sigma_i)_{i\in I}$, then $\mathcal{A}(\Pi) = \mathcal{A}(\Pi')(\bigvee_{i\in I} \mathcal{A}(\Pi'_i))$
- If $\Pi \triangleright \Gamma = \Gamma' +_{i\in I} \Delta_i \vdash \mathtt{t}[\mathtt{p}/\mathtt{u}] : \tau$ follows from $\Pi' \triangleright \Gamma'' \vdash \mathtt{t} : \tau$ and $(\Psi_i \triangleright \Delta_i \vdash \mathtt{u} : \rho_i)_{i\in I}$, then $\mathcal{A}(\Pi) = \mathcal{A}(\Pi')[\mathtt{p}/\bigvee_{i\in I} \mathcal{A}(\Psi_i)]$

Remark that, in the application case of the definition above, the anf corresponding to $I = \emptyset$ is $\mathcal{A}(\Pi')\Omega$. Moreover, in the last case, $\mathtt{p}$ cannot be a variable, $\mathtt{t}$ being in Π-nf. A simple inspection of the typing rules for $\Vdash$ shows that in this case $I \neq \emptyset$.

▶ **Example 15.** Consider the following derivation Π:

$$\dfrac{\dfrac{\dfrac{\mathtt{y} : [[\,] \to o] \vdash \mathtt{y} : [\,] \to o}{\mathtt{y} : [[\,] \to o] \vdash \mathtt{y}(\Delta\Delta) : o} \quad \Vdash \langle \mathtt{z}_1, \mathtt{z}_2 \rangle : o \quad \dfrac{\mathtt{x} : [[\,] \to o] \vdash \mathtt{x} : [\,] \to o}{\mathtt{x} : [[\,] \to o] \vdash \mathtt{xId} : o}}{\mathtt{x} : [[\,] \to o]; \mathtt{y} : [[\,] \to o] \vdash \mathtt{y}(\Delta\Delta)[\langle \mathtt{z}_1, \mathtt{z}_2 \rangle / \mathtt{xId}] : o}}{\vdash \lambda\mathtt{xy}.\mathtt{y}(\Delta\Delta)[\langle \mathtt{z}_1, \mathtt{z}_2 \rangle / \mathtt{xId}] : [[\,] \to o] \to [[\,] \to o] \to o}$$

The minimal approximant of Π is $\lambda\mathtt{xy}.\mathtt{y}\Omega[\langle \mathtt{z}_1, \mathtt{z}_2 \rangle / \mathtt{x}\Omega]$.

A simple induction on $\mathtt{meas}(\Pi)$ allows to show the following:

▶ **Lemma 16.** *If $\Pi \triangleright \Gamma \vdash \mathtt{t} : \sigma$ and $\mathtt{t}$ is in Π-nf, then $\Pi \triangleright \Gamma \vdash \mathcal{A}(\Pi) : \sigma$.*

5.1 The inhabitation algorithm

The inhabitation algorithm is presented in Fig. 2. As usual, in order to solve the problem for closed terms, it is necessary to extend the algorithm to open ones, so, given an environment Γ and a strict type σ, the algorithm builds the set $\mathtt{T}(\Gamma, \sigma)$ containing *all* the $anfs$ $\mathtt{a}$ such that there exists a derivation $\Pi \triangleright \Gamma \vdash \mathtt{a} : \sigma$, with $\mathtt{a} = \mathcal{A}(\Pi)$, then stops[2]. Thus, our algorithm is not an extension of the classical inhabitation algorithm for simple types [4, 9]. In particular, when restricted to simple types, it constructs all the $anfs$ inhabiting a given type, while the original algorithm reconstructs just the *long η-normal forms*. The algorithm uses four auxiliary predicates, namely

- $\mathtt{P}_{\mathcal{V}}(\mathtt{A})$, where $\mathcal{V}$ is a finite set of variables, contains the pairs $(\Gamma, \mathtt{p})$ such that (i) $\Gamma \Vdash \mathtt{p} : \mathtt{A}$, and (ii) $\mathtt{p}$ does not contain any variable in $\mathcal{V}$.
- $\mathtt{TI}(\Gamma, [\sigma_i]_{i\in I})$, contains all the $anfs$ $\mathtt{a} = \bigvee_{i\in I} \mathtt{a}_i$ such that $\Gamma = +_{i\in I}\Gamma_i$, $\mathtt{a}_i \in \mathtt{T}(\Gamma_i, \sigma_i)$ for all $i \in I$, and $\uparrow_{i\in I} \mathtt{a}_i$.
- $\mathtt{H}^{\Delta}_{\mathtt{b}}(\Gamma, \sigma) \triangleright \tau$ contains all the $anfs$ $\mathtt{a}$ such that $\mathtt{b}$ is a head subterm of $\mathtt{a}$, and such that if $\mathtt{b} \in \mathtt{T}(\Delta, \sigma)$ then $\mathtt{a} \in \mathtt{T}(\Gamma + \Delta, \tau)$.
- $\mathtt{HI}^{\Delta}_{\mathtt{b}}(\Gamma, [\sigma_i]_{i\in I}) \triangleright [\rho_i]_{i\in I}$ contains all the anf $\mathtt{a} = \bigvee_{i\in I} \mathtt{a}_i$ such that $\Delta = +_{i\in I}\Delta_i$, $\Gamma = +_{i\in I}\Gamma_i$, $\mathtt{a}_i \in \mathtt{H}^{\Delta}_{\mathtt{b}}(\Gamma, \sigma_i) \triangleright \rho_i$ and $\uparrow_{i\in I} \mathtt{a}_i$.

[2] It is worth noticing that, given Γ and σ, the set of $anfs$ $\mathtt{a}$ such that there exists a derivation $\Pi \triangleright \Gamma \vdash \mathtt{a} : \sigma$ is possibly infinite. However, the subset of those verifying $\mathtt{a} = \mathcal{A}(\Pi)$ is finite; they are the minimal ones, those generated by the inhabitation algorithm (this is proved in Lem. 19).

Note that the algorithm has different kinds of non-deterministic behaviours, *i.e.* different choices of rules can produce different results. Indeed, given an input (Γ, σ), the algorithm may apply a rule like (Abs) in order to decrease the type σ, or a rule like (Head) in order to decrease the environment Γ. Moreover, every rule (R) which is based on some decomposition of the environment and/or the type, like (Subs), admits different applications. In what follows we illustrate the non-deterministic behaviour of the algorithm. For that, we represent a **run of the algorithm** as a tree whose nodes are labeled with the name of the rule applied.

▶ **Example 17.** We consider different inputs of the form $(\emptyset, \sigma)$, for different strict types σ. For every such input, we give an output and the corresponding run.

1. $\sigma = [[\alpha] \to \alpha] \to [\alpha] \to \alpha$.
   a. output: $\lambda \mathtt{xy.xy}$, run: $\mathtt{Abs}(\mathtt{Abs}(\mathtt{Head}(\mathtt{Prefix}(\mathtt{TUn}(\mathtt{Head}(\mathtt{Final})), \mathtt{Final})), \mathtt{Varp}), \mathtt{Varp})$.
   b. output: $\lambda \mathtt{x.x}$, run: $\mathtt{Abs}(\mathtt{Head}(\mathtt{Final}), \mathtt{Varp})$.
2. $\sigma = [[\,] \to \alpha] \to \alpha$. output: $\lambda \mathtt{x.x}\Omega$, run: $\mathtt{Abs}(\mathtt{Head}(\mathtt{Prefix}(\mathtt{TUn}, \mathtt{Final})), \mathtt{Varp})$.
3. $\sigma = [[o] \to o, o] \to o$.
   a. output: $\lambda \mathtt{x.xx}$, run: $\mathtt{Abs}(\mathtt{Head}(\mathtt{Prefix}(\mathtt{TUn}(\mathtt{Head}(\mathtt{Final})), \mathtt{Final})), \mathtt{Varp})$.
   b. Explicit substitutions may be used to consume some, or all, the resources in $[[o] \to o, o]$ output: $\lambda \mathtt{x.x}[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{x}\langle \Omega, \Omega \rangle]$, run:
   $\mathtt{Abs}(\mathtt{Subs}(\mathtt{HUn}(\mathtt{Prefix}(\mathtt{TUn}(\mathtt{Pair}), \mathtt{Final})), \mathtt{Pairp}(\mathtt{Weakp}, \mathtt{Weakp}), \mathtt{Head}(\mathtt{Final})), \mathtt{Varp})$.
   c. There are four additional runs, producing the following outputs:
   $\lambda \mathtt{x.x}\langle \Omega, \Omega \rangle[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{x}]$,
   $\lambda \mathtt{x}.\langle \Omega, \Omega \rangle[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{xx}]$,
   $\lambda \mathtt{x}.\langle \Omega, \Omega \rangle[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{x}][\langle \mathtt{w}, \mathtt{s} \rangle / \mathtt{x}\langle \Omega, \Omega \rangle]$,
   $\lambda \mathtt{x}.\langle \Omega, \Omega \rangle[\langle \mathtt{y}, \mathtt{z} \rangle / \mathtt{x}\langle \Omega, \Omega \rangle][\langle \mathtt{w}, \mathtt{s} \rangle / \mathtt{x}]$.

Along the recursive calls of the inhabitation algorithm, the parameters (type and/or environment) decrease strictly, for a suitable notion of measure, so that every run is finite:

▶ **Lemma 18.** *The inhabitation algorithm terminates.*

5.2 Soundness and completeness

We now prove soundness and completeness of our inhabitation algorithm.

▶ **Lemma 19.** $\mathtt{a} \in \mathtt{T}(\Gamma, \sigma) \Leftrightarrow \exists \Pi \triangleright \Gamma \vdash \mathtt{a} : \sigma$ *such that* $\mathtt{a} = \mathcal{A}(\Pi)$.

Proof. The "only if" part is proved by induction on the rules in Fig. 2, and the "if" part is proved by induction on the definition of $\mathcal{A}(\Pi)$. In both parts, additional statements concerning the predicates of the inhabitation algorithm other than T are required, in order to strenghten the inductive hypothesis. ◀

▶ **Theorem 20** (Soundness and Completeness)**.**

1. *If* $\mathtt{a} \in \mathtt{T}(\Gamma, \sigma)$ *then, for all* $\mathtt{t}$ *such that* $\mathtt{a} \leq \mathtt{t}$, $\Gamma \vdash \mathtt{t} : \sigma$.
2. *If* $\Pi \triangleright \Gamma \vdash \mathtt{t} : \sigma$ *then there exists* $\Pi' \triangleright \Gamma \vdash \mathtt{t}' : \sigma$ *such that* $\mathtt{t}'$ *is in* Π'*-nf, and* $\mathcal{A}(\Pi') \in \mathtt{T}(\Gamma, \sigma)$.

Proof. Soundness follows from Lem. 19 ($\Rightarrow$) and the fact that $\Gamma \vdash \mathtt{a} : \sigma$ and $\mathtt{a} \leq \mathtt{t}$ imply $\Gamma \vdash \mathtt{t} : \sigma$. For completeness we first apply Lem. 9.1 that guarantees the existence of $\Pi' \triangleright \Gamma \vdash \mathtt{t}' : \sigma$ such that $\mathtt{t}'$ is in Π'-nf, and then Lem. 16 and Lem. 19 ($\Leftarrow$). ◀

$$\frac{\mathtt{x} \notin \mathcal{V}}{(\emptyset;\mathtt{x}) \in_0 \mathtt{P}_{\mathcal{V}}([\,])}\ (\mathtt{Weakp}) \qquad \frac{\mathtt{x} \notin \mathcal{V}}{(\mathtt{x}:\mathtt{B};\mathtt{x}) \in_0 \mathtt{P}_{\mathcal{V}}(\mathtt{B})}\ (\mathtt{Varp})$$

$$\frac{(\Gamma;\mathtt{p}) \in_i \mathtt{P}_{\mathcal{V}}(\mathtt{A}_1) \qquad (\Delta;\mathtt{q}) \in_j \mathtt{P}_{\mathcal{V}}(\mathtt{A}_2) \qquad \mathtt{p}\#\mathtt{q}}{(\Gamma;\Delta;\langle \mathtt{p},\mathtt{q}\rangle) \in_1 \mathtt{P}_{\mathcal{V}}(\mathtt{A}_1 \mathbb{X} \mathtt{A}_2)}\ (\mathtt{Pairp})$$

$$\frac{\mathtt{a} \in \mathtt{T}(\Gamma;\Delta,\tau) \qquad \mathtt{A} = \mathtt{A}_1 \sqcup \mathtt{A}_2 \qquad (\Delta;\mathtt{p}) \in_i \mathtt{P}_{\mathtt{dom}(\Gamma)}(\mathtt{A}_1) \qquad \mathtt{supp}(\mathtt{A}_2) \subseteq \mathcal{S}(\mathtt{p})}{\lambda \mathtt{p}.\mathtt{a} \in \mathtt{T}(\Gamma, \mathtt{A} \to \tau)}\ (\mathtt{Abs})$$

$$\frac{(\mathtt{a}_i \in \mathtt{T}(\Gamma_i,\sigma_i))_{i\in I} \qquad \uparrow_{i\in I} \mathtt{a}_i}{\bigvee_{i\in I} \mathtt{a}_i \in \mathtt{TI}(+_{i\in I}\Gamma_i, [\sigma_i]_{i\in I})}\ (\mathtt{TUn}) \qquad \frac{(\mathtt{a}_i \in \mathtt{H}_{\mathtt{b}}^{\Delta_i}(\Gamma_i,\sigma_i) \triangleright \rho_i)_{i\in I} \qquad \uparrow_{i\in I} \mathtt{a}_i}{\bigvee_{i\in I} \mathtt{a}_i \in \mathtt{HI}_{\mathtt{b}}^{+_{i\in I}\Delta_i}(+_{i\in I}\Gamma_i, [\sigma_i]_{i\in I}) \triangleright [\rho_i]_{i\in I}}\ (\mathtt{HUn})$$

$$\frac{}{\langle \Omega,\Omega\rangle \in \mathtt{T}(\emptyset, o)}\ (\mathtt{Pair}) \qquad \frac{\mathtt{a} \in \mathtt{T}(\Gamma,\tau)}{\langle \mathtt{a},\Omega\rangle \in \mathtt{T}(\Gamma, \times_1(\tau))}\ (\mathtt{Prod1}) \qquad \frac{\mathtt{a} \in \mathtt{T}(\Gamma,\tau)}{\langle \Omega,\mathtt{a}\rangle \in \mathtt{T}(\Gamma, \times_2(\tau))}\ (\mathtt{Prod2})$$

$$\frac{\mathtt{a} \in \mathtt{H}_{\mathtt{x}}^{\mathtt{x}:[\sigma]}(\Gamma,\sigma) \triangleright \tau}{\mathtt{a} \in \mathtt{T}(\Gamma + (\mathtt{x}:[\sigma]),\tau)}\ (\mathtt{Head}) \qquad \frac{\sigma = \tau}{\mathtt{a} \in \mathtt{H}_{\mathtt{a}}^{\Delta}(\emptyset,\sigma) \triangleright \tau}\ (\mathtt{Final})$$

$$\frac{\Gamma = \Gamma_0 + \Gamma_1 \qquad \mathtt{b} \in \mathtt{TI}(\Gamma_0, \mathtt{A}) \qquad \mathtt{a} \in \mathtt{H}_{\mathtt{cb}}^{\Delta+\Gamma_0}(\Gamma_1,\sigma) \triangleright \tau}{\mathtt{a} \in \mathtt{H}_{\mathtt{c}}^{\Delta}(\Gamma, \mathtt{A} \to \sigma) \triangleright \tau}\ (\mathtt{Prefix})$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_0 + \Gamma_1 \qquad \mathtt{c} \in \mathtt{HI}_{\mathtt{x}}^{\mathtt{x}:\mathtt{B}}(\Gamma_0, \mathtt{B}) \triangleright \mathtt{F}(\mathtt{B}) \qquad \mathtt{F}(\mathtt{B}) = \mathtt{A}_1 \sqcup \mathtt{A}_2(*) \\ (\Delta,\mathtt{p}) \in_1 \mathtt{P}_{\mathtt{dom}(\Gamma_0+\Gamma_1+(\mathtt{x}:\mathtt{B}))}(\mathtt{A}_1) \qquad \mathtt{supp}(\mathtt{A}_2) \subseteq \mathcal{S}(\mathtt{p}) \qquad \mathtt{b} \in \mathtt{T}(\Gamma_1;\Delta,\tau)\end{array}}{\mathtt{b}[\mathtt{p}/\mathtt{c}] \in \mathtt{T}(\Gamma + (\mathtt{x}:\mathtt{B}),\tau)}\ (\mathtt{Subs})$$

(*) where the operator $\mathtt{F}()$ is defined as follows:

$$\mathtt{F}(\alpha) := \alpha \qquad \mathtt{F}(\mathtt{A} \to \tau) := \mathtt{F}(\tau) \qquad \mathtt{F}([\,]) := [\,] \qquad \mathtt{F}([\sigma_i]_{i\in I}) := [\mathtt{F}(\sigma_i)]_{i\in I}$$

Figure 2 The inhabitation algorithm.

6 Characterizing Observability

We are now able to state the main result of this paper, *i.e.* the characterization of observability for the pattern calculus. The following lemma assures that types reflect correctly the structure of the data types.

▶ **Lemma 21.** *Let* $\mathtt{t}$ *be a closed and typable term, then*

- *If* $\mathtt{t}$ *has functional type, then* $\mathtt{t}$ *reduces to an abstraction.*
- *If* $\mathtt{t}$ *has product type, then* $\mathtt{t}$ *reduces to a pair.*

Proof. Let $\mathtt{t}$ be a closed and typable term. By Thm. 10 we know that $\mathtt{t}$ reduces to a (closed) canonical form in $\mathcal{J}$. The proof is by induction on the maximal length of such reduction sequences.

If $\mathtt{t}$ is already a canonical form, we analyze all the cases.

- If $\mathtt{t}$ is a variable, then this gives a contradiction with $\mathtt{t}$ closed.
- If $\mathtt{t}$ is a function, then the property trivially holds.

- If $\mathtt{t}$ is a pair, then the property trivially holds.
- If $\mathtt{t}$ is an application, then $\mathtt{t}$ has the form $\mathtt{x}\mathtt{t}_1 \dots \mathtt{t}_n$. Therefore at least $\mathtt{x}$ belongs to the set of free variables of $\mathtt{t}$, which leads to a contradiction with $\mathtt{t}$ closed.
- If $\mathtt{t}$ is a closure, *i.e.* $\mathtt{t} = \mathtt{u}[\langle \mathtt{p}_1, \mathtt{p}_2 \rangle / \mathtt{v}]$, where $\mathtt{v} \in \mathcal{K}$ has the form $\mathtt{x}\mathtt{t}_1 \dots \mathtt{t}_n$, then at least $\mathtt{x}$ belongs to the set of free variables of $\mathtt{t}$, which leads to a contradiction with $\mathtt{t}$ closed.

Otherwise, $\mathtt{t} \to \mathtt{t}' \to^* \mathtt{u}$, where $\mathtt{u}$ is in $\mathcal{J}$. The term $\mathtt{t}'$ is also closed and typable (Lem. 9.1), then the *i.h.* gives the desired result for $\mathtt{t}'$, so the property holds also for $\mathtt{t}$. ◀

▶ **Theorem 22** (Characterizing Observability). *A term* $\mathtt{t}$ *is observable iff* $\Pi \rhd \mathtt{x}_1 : \mathtt{A}_1; \dots; \mathtt{x}_n : \mathtt{A}_n \vdash \mathtt{t} : \mathtt{B}_1 \to \dots \to \mathtt{B}_m \to \alpha$*, where* $n \geq 0, m \geq 0$*,* α *is a product type and all* $\mathtt{A}_1, \dots \mathtt{A}_n, \mathtt{B}_1, \dots \mathtt{B}_m$ *are inhabited.*

Proof. The left-to-right implication: if $\mathtt{t}$ is observable, then there exists a head-context $\mathtt{C}$ such that $\mathtt{C}[\mathtt{t}] \to^* \langle \mathtt{u}, \mathtt{v} \rangle$. Since $\vdash \langle \mathtt{u}, \mathtt{v} \rangle : o$, we get $\Pi' \rhd \vdash \mathtt{C}[\mathtt{t}] : o$ by Lem. 9.2. By definition $\mathtt{C}[\mathtt{t}] = (\lambda \mathtt{p}_1 ... \lambda \mathtt{p}_n.\mathtt{t})\mathtt{u}_1 ... \mathtt{u}_m$, so Π has a subderivation $\Pi' \rhd \vdash \lambda \mathtt{p}_1 ... \lambda \mathtt{p}_n.\mathtt{t} : \mathtt{B}_1 \to \dots \to \mathtt{B}_m \to o$ (by rule $(\to \mathtt{e})$), where $\mathtt{B}_i$ is inhabited by $\mathtt{u}_i$ $(1 \leq i \leq m)$. Since $n \leq m$, Π' has a subderivation $\Pi'' \rhd \Gamma \vdash \mathtt{t} : \mathtt{B}_{n+1} \to \dots \to \mathtt{B}_m \to o$ (by rule $(\to \mathtt{i})$), where $\Gamma|_{\mathtt{p}_i} \Vdash \mathtt{p}_i : \mathtt{B}_i$ $(1 \leq i \leq n)$. The result follows since $\mathtt{x}_1 : A_1, \dots, .\mathtt{x}_l : A_l \Vdash \mathtt{p} : B$ and B is inhabited implies that all the A_i are inhabited. The right-to-left implication: if $\mathtt{A}_1, \dots \mathtt{A}_n, \mathtt{B}_1, \dots \mathtt{B}_m$ are all inhabited, then there exist $\mathtt{u}_1, \dots \mathtt{u}_n, \mathtt{v}_1, \dots \mathtt{v}_m$ such that $\vdash \mathtt{u}_i : \sigma_i^j$ for every type σ_i^j of $\mathtt{A}_i$ $(1 \leq i \leq n)$ and $\vdash \mathtt{v}_i : \rho_i^j$ for every type ρ_i^j of $\mathtt{B}_i$ $(1 \leq i \leq m)$. Let $\mathtt{C} = (\lambda \mathtt{x}_1 \dots \mathtt{x}_n.\Box)\mathtt{u}_1 \dots \mathtt{u}_n \mathtt{v}_1 \dots \mathtt{v}_n$ be a head-context. We have $\vdash \mathtt{C}[\mathtt{t}] : \alpha$, which in turn implies that $\mathtt{C}[\mathtt{t}]$ reduces to a pair, by Lem. 21. Then the term $\mathtt{t}$ is observable by definition. ◀

The notion of observability is conservative with respect to that of solvability in λ-calculus.

▶ **Theorem 23** (Conservativity). *A* λ*-term* $\mathtt{t}$ *is solvable in the* λ*-calculus if and only if* $\mathtt{t}$ *is observable in* $\Lambda_{\mathtt{p}}$*.*

Proof.
- (if) Take an unsolvable λ-term $\mathtt{t}$ so that $\mathtt{t}$ does not have head normal-form. Then $\mathtt{t}$ (seen as a term of our calculus) has no canonical form, and thus $\mathtt{t}$ is not typable by Thm. 10. It turns out that $\mathtt{t}$ is not observable in $\Lambda_{\mathtt{p}}$ by Thm. 22.
- (only if) Take a solvable λ-term $\mathtt{t}$ so that there exist a head-context $\mathtt{C}$ such that $\mathtt{C}[\mathtt{t}]$ reduces to $\mathtt{Id}$, then it is easy to construct a head context $\mathtt{C}'$ such that $\mathtt{C}'[\mathtt{t}]$ reduces to a pair (just take $\mathtt{C}' = \mathtt{C}\ \langle \mathtt{t}_1, \mathtt{t}_2 \rangle$ for some terms $\mathtt{t}_1, \mathtt{t}_2$). ◀

7 Conclusion and Further Work

We propose a notion of observability for pair pattern calculi which is conservative with respect to the notion of solvability for λ-calculus.

We provide a logical characterization of observable terms by means of typability *and* inhabitation.

Further work will be developed in different directions. As we already discussed in Sec. 2, different definitions of observability would be possible. We explored the one based on a lazy semantics, but it would be also interesting to obtain a full characterization based on a strict semantics. Another point to be developed is the definition of a suitable notion of head reduction, which, despite its relative simplicity, turn out to be quite cumbersome. On the semantical side, it is well known that non-idempotent intersection types can be used to supply a logical description of the relational semantics of λ-calculus [8, 14]. We would like to start from our type assignment system for building a denotational model of the pattern

calculus. Last but not least, a challenging question is related to the characterization of observability in a more general framework of pattern λ-calculi allowing the patterns to be dynamic [10].

References

1 Thibaut Balabonski. On the implementation of dynamic patterns. In Eduardo Bonelli, editor, *HOR*, volume 49 of *EPTCS*, pages 16–30, 2010.

2 Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics*. North-Holland, Amsterdam, revised edition, 1984.

3 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983.

4 C. Ben-Yelles. *Type-assignment in the lambda-calculus; syntax and semantics*. PhD thesis, University of Wales Swansea, 1979.

5 Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. The inhabitation problem for non-idempotent intersection types. In Josep Díaz, Ivan Lanese, and Davide Sangiorgi, editors, *TCS*, LNCS. Springer, 2014. To appear.

6 Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323(1-3):71–127, 2004.

7 Horatiu Cirstea, Germain Faure, and Claude Kirchner. A rho-calculus of explicit constraint application. *Higher-Order and Symbolic Computation*, 20(1-2):37–72, 2007.

8 Daniel de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *CoRR*, abs/0905.4251, 2009.

9 J. Roger Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Amsterdam, 2008.

10 Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.

11 Wolfram Kahl. Basic pattern matching calculi: A fresh view on matching failure. In Yukiyoshi Kameyama and Peter Stuckey, editors, *FLOPS*, volume 2998 of *LNCS*, pages 276–290. Springer, 2004.

12 Jan-Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1-3):16–31, 2008.

13 Jean Louis Krivine. *Lambda-Calculus, Types and Models*. Masson, Paris, and Ellis Horwood, Hemel Hempstead, 1993.

14 Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Logical relational lambda-models. To appear in Mathematical Structures in Computer Science.

15 Barbara Petit. A polymorphic type system for the lambda-calculus with constructors. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2009.

16 Simon Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

17 Pawel Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

18 Vincent van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(2):259–280, 1994.

Undecidability of Equality in the Free Locally Cartesian Closed Category

Simon Castellan[1], Pierre Clairambault[1], and Peter Dybjer[2]

1 **ENS de Lyon, CNRS, Inria, UCBL, Université de Lyon**
LIP, 46 allée d'Italie, 69364 Lyon, France
2 **Chalmers University of Technology, Sweden**

Abstract

We show that a version of Martin-Löf type theory with extensional identity, a unit type $\mathrm{N}_1, \Sigma, \Pi$, and a base type is a free category with families (supporting these type formers) both in a 1- and a 2-categorical sense. It follows that the underlying category of contexts is a free locally cartesian closed category in a 2-categorical sense because of a previously proved biequivalence. We then show that equality in this category is undecidable by reducing it to the undecidability of convertibility in combinatory logic.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.2 Semantics of Programming Languages

Keywords and phrases Extensional type theory, locally cartesian closed categories, undecidability

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.138

1 Introduction

In previous work [5, 6] we showed the biequivalence of locally cartesian closed categories (lcccs) and the I, Σ, Π-fragment of extensional Martin-Löf type theory. More precisely, we showed the biequivalence of the following two 2-categories.

- The first has as *objects* lcccs, as *arrows* functors which preserve the lccc-structure (up to isomorphism), and as *2-cells* natural transformations.
- The second has as *objects* categories with families (cwfs) [8] which support extensional identity types (I-types), Σ-types, Π-types, and are *democratic*, as *arrows* pseudo cwf-morphisms (preserving structure up to isomorphism), and as *2-cells* pseudo cwf-transformations. A cwf is democratic iff there is an equivalence between its category of contexts and its category of closed types.

This result is a corrected version of a result by Seely [13] concerning the equivalence of the category of lcccs and the category of Martin-Löf type theories. Seely's paper did not address the coherence problem caused by the interpretation of substitution as pullbacks [7]. As Hofmann showed [9], this coherence problem can be solved by extending a construction of Bénabou [2]. Our biequivalence is based on this construction.

Cwfs are models of the most basic rules of dependent type theory; those dealing with substitution, assumption, and context formation, the rules which come before any rules for specific type formers. The distinguishing feature of cwfs, compared to other categorical notions of model of dependent types, is that they are formulated in a way which makes the connection with the ordinary syntactic formulation of dependent type theory transparent. They can be defined purely equationally [8] as a generalised algebraic theory (gat) [3], where each sort symbol corresponds to a judgment form, and each operator symbol corresponds to

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 138–152

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

an inference rule in a variable free formulation of Martin-Löf's explicit substitution calculus for dependent type theory [11, 15].

Cwfs are not only models of dependent type theory, but also suggest an answer to the question what dependent type theory is as a mathematical object. Perhaps surprisingly, this is a non-trivial question, and Voevodsky has remarked that "a type system is not a mathematical notion". There are numerous variations of Martin-Löf type theory in the literature, even of the formulation of the most basic rules for dependent types. There are systems with explicit and implicit substitutions, variations in assumption, context formation, and substitution rules. There are formulations with de Bruijn indices and with ordinary named variables, etc. In fact, there are so many rules that most papers do not try to provide a complete list; and if you do try to list all of them how can you be sure that you haven't forgotten any? Nevertheless, there is a tacit assumption that most variations are equivalent and that a complete list of rules could be given if needed. However, from a mathematical point of view this is neither clear nor elegant.

To remedy this situation we suggest to define Martin-Löf type theory (and other dependent type theories) abstractly as the initial cwf (with extra structure). The category of cwfs and morphisms which preserve cwf-structure on the nose was defined by Dybjer [8]. We suggest that the correctness of a definition or an implementation of dependent type theory means that it gives rise to an initial object in this category of cwfs (with extra structure). Here we shall construct the initial object in this category explicitly in the simplest possible way following closely the definition of the generalised algebraic theory of cwfs. Note however that the notion of a generalised algebraic theory is itself based on dependent type theory, that is, on cwf-structure. So just defining the initial cwf as the generalised algebraic theory of cwfs would be circular. Instead we construct the initial cwf explicitly by giving grammar and inference rules which follow closely the operators of the gat of cwfs. However, we must also make equality reasoning explicit. To decrease the number of rules, we present a "per-style" system rather than an ordinary one. We will mutually define four partial equivalence relations (pers): for the judgments of context equality $\Gamma = \Gamma'$, substitution equality $\Delta \vdash \gamma = \gamma' : \Gamma$, type equality $\Gamma \vdash A = A'$, and term equality $\Gamma \vdash a = a' : A$. The ordinary judgments will be defined as the reflexive instances, for example, $\Gamma \vdash a : A$ will be defined as $\Gamma \vdash a = a : A$.

Our only optimisation is the elimination of some redundant arguments of operators. For example, the composition operator in the gat of cwfs has five arguments: three objects and two arrows. However, the three object arguments can be recovered from the arrows, and can hence be omitted. This method is also used in *D-systems*, the essentially algebraic formulation of cwfs by Voevodsky.

The goal of the present paper is to prove the undecidability of equality in the free lccc. To this end we extend our formal system for cwfs with rules for extensional I-types, $\mathrm{N}_1, \Sigma, \Pi$, and a base type. Now we want to show that this yields a free lccc on one object, by appealing to our biequivalence theorem. (Since the empty context corresponds to the unit type N_1 and context extension to Σ, it follows that our free cwf is democratic.) However, it does not suffice to show that we get a free cwf in the 1-category of cwfs and strict cwf-morphism, but we must show that it is also free ("bifree") in the 2-category of cwfs and pseudo cwf-morphisms. Although informally straightforward, this proof is technically more involved because of the complexity of the notion of pseudo cwf-morphism.

Once we have constructed the free lccc (as a cwf-formulation of Martin-Löf type theory with extensional I-types, $\mathrm{N}_1, \Sigma, \Pi$, and one base type) we will be able to prove undecidability. It is a well-known folklore result that extensional Martin-Löf type theory with one universe

has undecidable equality, and we only need to show that a similar construction can be made without a universe, provided we have a base type. We do this by encoding untyped combinatory logic as a context, and use the undecidability of equality in this theory.

Related work. Palmgren and Vickers [12] show how to construct free models of essentially algebraic theories in general. We could use this result to build a free cwf, but this only shows freeness in the 1-categorical sense. We also think that the explicit construction of the free (and bifree) cwf is interesting in its own right.

Plan. In Section 2 we prove a few undecidability theorems, including the undecidability of equality in Martin-Löf type theory with extensional I-types, Π, and one base type. In Section 3 we construct a free cwf on one base type. We show that it is free both in a 1-categorical sense (where arrows preserve cwf-structure on the nose) and in a 2-categorical sense (where arrows preserve cwf-structure up to isomorphism). In Section 4 we construct a free cwf with extensional identity types, $\mathrm{N}_1, \Sigma, \Pi$, and one base type. We then use the biequivalence result to conclude that this yields a free lccc in a 2-categorical sense.

2 Undecidability in Martin-Löf type theory

Like any other single-sorted first order equational theory, combinatory logic can be encoded as a context in Martin-Löf type theory with I-types, Π-types, and a base type o. The context Γ_{CL} for combinatory logic is the following:

$$\begin{array}{lcllcl} k & : & o, & ax_k & : & \Pi xy : o.\ \mathrm{I}(o,\ k \cdot x \cdot y,\ x), \\ s & : & o, & ax_s & : & \Pi xyz : o.\ \mathrm{I}(o,\ s \cdot x \cdot y \cdot z,\ x \cdot z \cdot (y \cdot z)) \\ \cdot & : & o \to o \to o, & & & \end{array}$$

Here we have used the left-associative binary infix symbol "$\cdot$" for application. Note that $k, s, \cdot, ax_k, ax_s$ are all variables.

▶ **Theorem 1.** *Type-inhabitation in Martin-Löf type theory with (intensional or extensional) identity-types, Π-types and a base type is undecidable.*

This follows from the undecidability of convertibility in combinatory logic, because the type

$$\Gamma_{\mathrm{CL}} \vdash \mathrm{I}(o,\ M,\ M')$$

is inhabited iff the closed combinatory terms M and M' are convertible. Clearly, if the combinatory terms are convertible, it can be formalized in this fragment of type theory. For the other direction we build a model of the context Γ_{CL} where o is interpreted as the set of combinatory terms modulo convertibility.

▶ **Theorem 2.** *Judgmental equality in Martin-Löf type theory with extensional identity-types, Π-types and a base type is undecidable.*

With extensional identity types [10] the above identity type is inhabited iff the corresponding equality judgment is valid:

$$\Gamma_{\mathrm{CL}} \vdash M = M' : o$$

This theorem also holds if we add N_1 and Σ-types to the theory. The remainder of the paper will show that the category of contexts for the resulting fragment of Martin-Löf type theory is bifree in the 2-category of lcccs (Theorem 20). Our main result follows:

▶ **Theorem 3.** *Equality of arrows in the bifree lccc on one object is undecidable.*

We would like to remark that the following folklore theorem can be proved in the same way.

▶ **Theorem 4.** *Judgmental equality in Martin-Löf type theory with extensional identity-types,* Π*-types and a universe* U *is undecidable.*

If we have a universe we can instead work in the context

$$\begin{array}{rcl@{\qquad}rcl} X & : & \mathrm{U} & \cdot & : & X \to X \to X, \\ k & : & X, & ax_k & : & \Pi xy : X.\ \mathrm{I}(X,\ k \cdot x \cdot y,\ x), \\ s & : & X, & ax_s & : & \Pi xyz : X.\ \mathrm{I}(X,\ s \cdot x \cdot y \cdot z,\ x \cdot z \cdot (y \cdot z)) \end{array}$$

and prove undecidability for this theory (without a base type) in the same way as above.

Note that we don't need any closure properties at all for U – only the ability to quantify over small types. Hence we prove a slightly stronger theorem than the folklore theorem which assumes that U is closed under function types, and then uses the context

$$\begin{array}{rcl} X & : & \mathrm{U}, \\ x & : & \mathrm{I}(\mathrm{U}, X, X \to X) \end{array}$$

so that X is a model of the untyped lambda calculus.

3 A free category with families

In this section we define a free cwf syntactically, as a *term model* consisting of derivable contexts, substitutions, types and terms modulo derivable equality. To this end we give syntax and inference rules for a cwf-calculus, that is, a variable free explicit substitution calculus for dependent type theory.

We first prove that this calculus yields a free cwf in the category where morphisms preserve cwf-structure on the nose. The free cwf on one object is a rather degenerate structure, since there are no non-trivial dependent types. However, we have nevertheless chosen to present this part of the construction separately. Cwfs model the common core of dependent type theory, including all generalised algebraic theories, pure type systems [1], and fragments of Martin-Löf type theory. The construction of a free pure cwf is thus the common basis for constructing free and initial cwfs with appropriate extra structure for modelling specific dependent type theories.

In Section 3.4 we prove that our free cwf is also bifree. We then extend this result to cwfs supporting N_1, Σ, and Π-types. By our biequivalence result [5, 6] it also yields a bifree lccc.

3.1 The 2-category of categories with families

The 2-category of cwfs and pseudo-morphisms which preserve cwf-structure up to isomorphism was defined in [5, 6]. Here we only give an outline.

▶ **Definition 5** (Category with families)**.** A cwf $\mathcal{C}$ is a pair $(\mathcal{C}, T)$ of a category $\mathcal{C}$ and a functor $T : \mathcal{C}^{\mathrm{op}} \to \mathbf{Fam}$ where **Fam** is the category of families of sets. We write $\mathrm{Ctx}_{\mathcal{C}} = |\mathcal{C}|$ and $\mathrm{Sub}_{\mathcal{C}}(\Delta, \Gamma) = \mathrm{Hom}_{\mathcal{C}}(\Delta, \Gamma)$. For $\Gamma \in \mathrm{Ctx}_{\mathcal{C}}$ we write $T\Gamma = (\mathrm{Tm}_{\mathcal{C}}(\Gamma, A))_{A \in \mathrm{Ty}_{\mathcal{C}}\Gamma}$. The functorial action of T on a type A is written $A[\_]$: if $\gamma : \mathrm{Sub}_{\mathcal{C}}(\Gamma, \Delta)$ and $A \in \mathrm{Ty}_{\mathcal{C}}(\Delta)$, $A[\gamma] \in \mathrm{Ty}_{\mathcal{C}}(\Gamma)$. Similarly if $a \in \mathrm{Tm}_{\mathcal{C}}(\Delta, A)$, we write $a[\gamma] \in \mathrm{Tm}_{\mathcal{C}}(\Gamma, A[\gamma])$ for the functorial action of T on a.

We assume that $\mathcal{C}$ has a terminal object $1_{\mathcal{C}}$. Moreover we assume that for each $\Gamma \in \mathrm{Ctx}_{\mathcal{C}}$ and $A \in \mathrm{Ty}_{\mathcal{C}}(\Gamma)$ there exists $\Gamma.A \in \mathrm{Ctx}_{\mathcal{C}}$ with a map $\mathtt{p}_A : \mathrm{Sub}_{\mathcal{C}}(\Gamma.A, \Gamma)$ and a term $\mathtt{q}_A \in \mathrm{Tm}_{\mathcal{C}}(\Gamma.A, A[\mathtt{p}_A])$, such that for every pair $\gamma : \mathrm{Sub}_{\mathcal{C}}(\Delta, \Gamma)$ and $a \in \mathrm{Tm}_{\mathcal{C}}(\Delta, A[\gamma])$ there exists a unique map $\langle \gamma, a \rangle : \mathrm{Sub}_{\mathcal{C}}(\Delta, \Gamma.A)$ such that $\mathtt{p}_A \circ \langle \gamma, a \rangle = \gamma$ and $\mathtt{q}_A[\langle \gamma, a \rangle] = a$.

Note that with the notation $\mathrm{Ty}_{\mathcal{C}}$ and $\mathrm{Tm}_{\mathcal{C}}$ there is no need to explicitly mention the functor T when working with categories with families, and we will often omit it. Given a substitution $\gamma : \Gamma \to \Delta$, and $A \in \mathrm{Ty}_{\mathcal{C}}(\Delta)$, we write $\gamma \uparrow A$ or γ^+ (when A can be inferred from the context) for the lifting of γ to A: $\langle \gamma \circ \mathtt{p}, \mathtt{q} \rangle : \Gamma.A[\gamma] \to \Delta.A$.

The indexed category. In [5, 6] it is shown that any cwf $\mathcal{C}$ induces a functor $\mathbf{T} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ assigning to each context Γ the category whose objects are types in $\mathrm{Ty}_{\mathcal{C}}(\Gamma)$ and morphisms from A to B are substitutions $\gamma : \Gamma.A \to \Gamma.B$ such that $\mathtt{p} \circ \gamma = \mathtt{p}$. (They are in bijection with terms of type $\Gamma \cdot A \vdash B[\mathtt{p}]$.) Any morphism γ in $\mathbf{T}\Gamma$ from a type A to B induces a function on terms of that type written $\{\gamma\} : \mathrm{Tm}_{\mathcal{C}}(\Gamma, A) \to \mathrm{Tm}_{\mathcal{C}}(\Gamma, B)$ defined by $\{\gamma\}(a) = \mathtt{q}[\gamma \circ \langle \mathtt{id}, a \rangle]$. We will write $\theta : A \cong_{\Gamma} B$ for an isomorphism in $\mathbf{T}\Gamma$.

The functorial action is given by $\mathbf{T}(\gamma)(\varphi) = \langle \mathtt{p}, \mathtt{q}[\varphi \circ \gamma \uparrow A] \rangle : \Gamma.A[\gamma] \to \Gamma.B[\gamma]$, from which we deduce the action on terms $\{\mathbf{T}(\gamma)(\varphi)\}(a) = \mathtt{q}[\varphi \circ \langle \gamma, a \rangle]$.

▶ **Definition 6** (Pseudo cwf-morphisms)**.** A pseudo-cwf morphism from a cwf $(\mathcal{C}, T)$ to a cwf $(\mathcal{C}', T')$ is a pair (F, σ) where $F : \mathcal{C} \to \mathcal{C}'$ is a functor and for each $\Gamma \in \mathcal{C}$, σ_Γ is a **Fam**-morphism from $T\Gamma$ to $T'F\Gamma$ preserving the structure up to isomorphism. For example, there are isomorphisms

- $\rho_{\Gamma,A} : F(\Gamma.A) \cong F\Gamma.FA$
- $\theta_{A,\gamma} : F\Gamma.FA[F\gamma] \cong F\Gamma.F(A[\gamma])$ for $\Gamma \vdash \gamma : \Delta$.

satisfying some coherence diagrams, see [6] for the complete definition.

As σ_Γ is a **Fam**-morphism from $(\mathrm{Tm}_{\mathcal{C}}(\Gamma, A))_{A \in \mathrm{Ty}_{\mathcal{C}}(\Gamma)}$ to $(\mathrm{Tm}_{\mathcal{D}}(F\Gamma, B))_{B \in \mathrm{Ty}_{\mathcal{D}}(F\Gamma)}$, we write FA for the image of A by $\mathrm{Ty}_{\mathcal{C}}(\Gamma) \to \mathrm{Ty}_{\mathcal{D}}(F\Gamma)$ induced by σ_Γ and Fa for the image of $\Gamma \vdash a : A$ through $\mathrm{Tm}_{\mathcal{C}}(\Gamma, A) \to \mathrm{Tm}_{\mathcal{D}}(F\Gamma, FA)$ induced by σ_Γ.

A pseudo cwf-morphism is strict whenever $\theta_{A,\gamma}$ and $\rho_{\Gamma,A}$ are both identities and $F1 = 1$. Cwfs and strict cwf-morphisms form a category $\mathbf{CwF}_s$.

▶ **Definition 7** (Pseudo cwf-transformation)**.** A pseudo cwf-transformation between functors (F, σ) and (G, τ) is a pair (φ, ψ) where $\varphi : F \Rightarrow G$ is a natural transformation, and for each $\Gamma \in \mathcal{C}$ and $A \in \mathrm{Ty}_{\mathcal{C}}(\Gamma)$ $\psi_{\Gamma,A}$ is a type isomorphism $FA \cong_{F\Gamma} GA[\varphi_\Gamma]$ satisfying:

$$\varphi_{\Gamma.A} = F(\Gamma.A) \xrightarrow{\rho_F} F\Gamma.FA \xrightarrow{\psi_{\Gamma,A}} F\Gamma.GA[\varphi_\Gamma] \xrightarrow{\varphi_\Gamma^+} G\Gamma.GA \xrightarrow{\rho_G^{-1}} G(\Gamma.A)$$

We will write **CwF** for the resulting 2-category.

3.2 Syntax and inference rules for the free category with families

3.2.1 Raw terms

In this section we define the syntax and inference rules for a minimal dependent type theory with one base type o. This theory is closely related to the generalised algebraic theory of cwfs [8], but here we define it as a usual logical system with a grammar and a collection of inference rules. The grammar has four syntactic categories: contexts `Ctx`, substitutions `Sub`, types `Ty` and terms `Tm`:

$$\begin{array}{lcllcl} \Gamma & ::= & 1 \mid \Gamma.A & \qquad A & ::= & o \mid A[\gamma] \\ \gamma & ::= & \gamma \circ \gamma \mid \mathtt{id}_\Gamma \mid \langle\rangle_\Gamma \mid \mathtt{p}_A \mid \langle \gamma, a \rangle_A & \qquad a & ::= & a[\gamma] \mid \mathtt{q}_A \end{array}$$

These terms have as few annotations as possible, only what is needed to recover the domain and codomain of a substitution, the context of a type, and the type of a term:

$$\begin{aligned}
\mathrm{dom}(\gamma \circ \gamma') &= \mathrm{dom}(\gamma') & \mathrm{cod}(\gamma \circ \gamma') &= \mathrm{cod}(\gamma) \\
\mathrm{dom}(\mathtt{id}_\Gamma) &= \Gamma & \mathrm{cod}(\mathtt{id}_\Gamma) &= \Gamma \\
\mathrm{dom}(\langle\rangle_\Gamma) &= \Gamma & \mathrm{cod}(\langle\rangle_\Gamma) &= 1 \\
\mathrm{dom}(\mathtt{p}_A) &= \text{ctx-of}(A).A & \mathrm{cod}(\mathtt{p}_A) &= \text{ctx-of}(A) \\
\mathrm{dom}(\langle\gamma, a\rangle_A) &= \mathrm{dom}(\gamma) & \mathrm{cod}(\langle\gamma, a\rangle_A) &= \mathrm{cod}(\gamma).A
\end{aligned}$$

$$\begin{aligned}
\text{ctx-of}(o) &= 1 & \text{type-of}(a[\gamma]) &= (\text{type-of}(a))[\gamma] \\
\text{ctx-of}(A[\gamma]) &= \mathrm{dom}(\gamma) & \text{type-of}(\mathtt{q}_A) &= A[\mathtt{p}_A]
\end{aligned}$$

These functions will be used in the freeness proof.

3.2.2 Inference rules

We simultaneously inductively define four families of partial equivalence relations (pers) for the four forms of equality judgment:

$$\Gamma = \Gamma' \vdash \qquad \Gamma \vdash A = A' \qquad \Delta \vdash \gamma = \gamma' : \Gamma \qquad \Gamma \vdash a = a' : A$$

In the inference rules which generate these pers we will use the following abbreviations for the basic judgment forms: $\Gamma \vdash$ abbreviates $\Gamma = \Gamma \vdash$, $\Gamma \vdash A$ abbreviates $\Gamma \vdash A = A$, $\Delta \vdash \gamma : \Gamma$ abbreviates $\Delta \vdash \gamma = \gamma : \Gamma$, and $\Gamma \vdash a : A$ abbreviates $\Gamma \vdash a = a : A$.

Per-rules for the four forms of judgments:

$$\frac{\Gamma = \Gamma' \vdash \quad \Gamma' = \Gamma'' \vdash}{\Gamma = \Gamma'' \vdash} \qquad \frac{\Gamma = \Gamma' \vdash}{\Gamma' = \Gamma \vdash} \qquad \frac{\Delta \vdash \gamma = \gamma' : \Gamma \quad \Delta \vdash \gamma' = \gamma'' : \Gamma}{\Delta \vdash \gamma = \gamma'' : \Gamma} \qquad \frac{\Delta \vdash \gamma = \gamma' : \Gamma}{\Delta \vdash \gamma' = \gamma : \Gamma}$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma \vdash A' = A''}{\Gamma \vdash A = A''} \qquad \frac{\Gamma \vdash A = A'}{\Gamma \vdash A' = A} \qquad \frac{\Gamma \vdash a = a' : A \quad \Gamma \vdash a' = a'' : A}{\Gamma \vdash a = a'' : A}$$

$$\frac{\Gamma \vdash a = a' : A}{\Gamma \vdash a' = a : A}$$

Preservation rules for judgments:

$$\frac{\Gamma = \Gamma' \vdash \quad \Delta = \Delta' \vdash \quad \Gamma \vdash \gamma = \gamma' : \Delta}{\Gamma' \vdash \gamma = \gamma' : \Delta'} \qquad \frac{\Gamma = \Gamma' \vdash \quad \Gamma \vdash A = A'}{\Gamma' \vdash A = A'}$$

$$\frac{\Gamma = \Gamma' \vdash \quad \Gamma \vdash A = A' \quad \Gamma \vdash a = a' : A}{\Gamma' \vdash a = a' : A'}$$

Congruence rules for operators:

$$\frac{\Gamma \vdash \delta = \delta' : \Delta \quad \Delta \vdash \gamma = \gamma' : \Theta}{\Gamma \vdash \gamma \circ \delta = \gamma' \circ \delta' : \Theta} \qquad \frac{\Gamma = \Gamma' \ \vdash}{\Gamma \ \vdash \ \mathtt{id}_\Gamma = \mathtt{id}_{\Gamma'} : \Gamma} \qquad \frac{\Gamma \vdash A = A' \quad \Delta \vdash \gamma = \gamma' : \Gamma}{\Delta' \vdash A[\gamma] = A'[\gamma']}$$

$$\frac{\Gamma \vdash a = a' : A \quad \Delta \vdash \gamma = \gamma' : \Gamma}{\Delta \vdash a[\gamma] = a'[\gamma'] : A[\gamma']} \qquad \frac{}{1 = 1 \vdash} \qquad \frac{\Gamma = \Gamma' \ \vdash}{\Gamma \vdash \langle\rangle_\Gamma = \langle\rangle_{\Gamma'} : 1} \qquad \frac{\Gamma = \Gamma' \vdash \quad \Gamma \vdash A = A'}{\Gamma.A = \Gamma'.A' \vdash}$$

$$\frac{\Gamma \vdash A = A'}{\Gamma.A \vdash \mathtt{p}_A = \mathtt{p}_{A'} : \Gamma} \qquad \frac{\Gamma \vdash A = A'}{\Gamma.A \vdash \mathtt{q}_A = \mathtt{q}_{A'} : A[\mathtt{p}_A]}$$

$$\frac{\Gamma \vdash A = A' \quad \Delta \vdash \gamma = \gamma' : \Gamma \quad \Delta \vdash a = a' : A[\gamma]}{\Delta \vdash \langle \gamma, a \rangle_A = \langle \gamma', a' \rangle_{A'} : \Gamma.A}$$

Conversion rules:

$$\frac{\Delta \vdash \theta : \Theta \quad \Gamma \vdash \delta : \Delta \quad \Xi \vdash \gamma : \Gamma}{(\theta \circ \delta) \circ \gamma = \theta \circ (\delta \circ \gamma)} \qquad \frac{\Gamma \ \vdash \ \gamma : \Delta}{\Gamma \vdash \gamma = \mathtt{id}_\Delta \circ \gamma : \Delta} \qquad \frac{\Gamma \ \vdash \ \gamma : \Delta}{\Gamma \vdash \gamma = \gamma \circ \mathtt{id}_\Gamma : \Delta}$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash \gamma : \Gamma \quad \Theta \vdash \delta : \Delta}{\Theta \vdash A[\gamma \circ \delta] = (A[\gamma])[\delta]} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A[\mathtt{id}_\Gamma] = A} \qquad \frac{\Gamma \vdash a : A \quad \Delta \vdash \gamma : \Gamma \quad \Theta \vdash \delta : \Delta}{\Theta \vdash a[\gamma \circ \delta] = (a[\gamma])[\delta] : (A[\gamma])[\delta]}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a[\mathtt{id}_\Gamma] = a : A} \qquad \frac{\Gamma \vdash \gamma : 1}{\Gamma \vdash \gamma = \langle\rangle_\Gamma : 1} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathtt{p}_A \circ \langle \gamma, a \rangle_A = \gamma : \Gamma}$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash \gamma : \Gamma \quad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathtt{q}_A[\langle \gamma, a \rangle_A] = a : A[\gamma]} \qquad \frac{\Delta \vdash \gamma : \Gamma.A}{\Delta \vdash \gamma = \langle \mathtt{p}_A \circ \gamma, \mathtt{q}_A[\gamma] \rangle_A : \Gamma.A}$$

Rule for the base type:

$$\frac{}{1 \vdash o = o}$$

3.2.3 The syntactic cwf $\mathcal{T}$

We can now define a term model as the syntactic cwf obtained by the well-formed contexts, etc, modulo judgmental equality. We use brackets for equivalence classes in this definition. (Note that brackets are also used for substitution in types and terms, but this should not cause confusion since we will soon drop the equivalence class brackets.)

▶ **Definition 8.** The term model $\mathcal{T}$ is given by:

- $\mathrm{Ctx}_\mathcal{T} = \{\Gamma \mid \Gamma \vdash \}/{=^c}$, where $\Gamma =^c \Gamma'$ if $\Gamma = \Gamma' \vdash$ is derivable.
- $\mathrm{Sub}_\mathcal{T}([\Gamma], [\Delta]) = \{\gamma \mid \Gamma \vdash \gamma : \Delta\}/{=^\Gamma_\Delta}$ where $\gamma =^\Gamma_\Delta \gamma'$ iff $\Gamma \vdash \gamma = \gamma' : \Delta$ is derivable. Note that this makes sense since it only depends on the equivalence class of Γ (morphisms and morphism equality are preserved by object equality).
- $\mathrm{Ty}_\mathcal{T}([\Gamma]) = \{A \mid \Gamma \vdash A\}/{=^\Gamma}$ where $A =^\Gamma B$ if $\Gamma \vdash A = B$.
- $\mathrm{Tm}_\mathcal{T}([\Gamma], [A]) = \{a \mid \Gamma \vdash a : A\}/{=^\Gamma_A}$ where $a =^\Gamma_A a'$ if $\Gamma \vdash a = a' : A$.

The cwf-operations on $\mathcal{T}$ can now be defined in a straightforward way. For example, if $\Delta \vdash \theta : \Theta$, $\Gamma \vdash \delta : \Delta$, we define $[\theta] \circ_\mathcal{T} [\delta] = [\theta \circ \delta]$, which is well-defined since composition preserves equality.

3.3 Freeness of $\mathcal{T}$

We shall now show that $\mathcal{T}$ is the free cwf on one base type, in the sense that given a cwf $\mathcal{C}$ and a type $A \in \mathrm{Ty}_{\mathcal{C}}(1)$, there exists a unique strict cwf morphism $\mathcal{T} \to \mathcal{C}$ which maps $[o]$ to A. Such a morphism can be defined by first defining a partial function for each sort of raw terms (where `Ctx` denotes the set of raw contexts, `Sub` the set of raw substitutions, and so on defined by the grammar of Section 3.2.1), cf Streicher [14].

$$\begin{aligned}
[\![-]\!] &: \texttt{Ctx} \to \mathrm{Ctx}_{\mathcal{C}} \\
[\![-]\!]_{\Gamma,\Delta} &: \texttt{Sub} \to \mathrm{Sub}_{\mathcal{C}}([\![\Gamma]\!],[\![\Delta]\!]) \\
[\![-]\!]_{\Gamma} &: \texttt{Ty} \to \mathrm{Ty}_{\mathcal{C}}([\![\Gamma]\!]) \\
[\![-]\!]_{\Gamma,A} &: \texttt{Tm} \to \mathrm{Tm}_{\mathcal{C}}([\![\Gamma]\!],[\![A]\!]_{\Gamma})
\end{aligned}$$

These functions are defined by mutual induction on the structure of raw terms:

$$\begin{aligned}
[\![1]\!] &= 1_{\mathcal{C}} & [\![\Gamma.A]\!] &= [\![\Gamma]\!].{}_{\mathcal{C}}[\![A]\!] \\
[\![\gamma' \circ \gamma]\!]_{\Gamma,\Theta} &= [\![\gamma']\!]_{\Delta,\Theta} \circ_{\mathcal{C}} [\![\gamma]\!]_{\Gamma,\Delta} & [\![\mathtt{id}_{\Gamma}]\!]_{\Gamma,\Gamma} &= \mathrm{id}_{\mathcal{C}[\![\Gamma]\!]} \\
[\![\langle \gamma, a\rangle_A]\!]_{\Gamma,\Delta.A} &= \langle [\![\gamma]\!]_{\Gamma,\Delta}, [\![a]\!]_{\Gamma,A[\gamma]}\rangle & [\![\langle\rangle_{\Gamma}]\!]_{\Gamma,1} &= \langle\rangle_{\Gamma} \\
[\![A[\gamma]]\!]_{\Gamma} &= [\![A]\!]_{\Delta}[[\![\gamma]\!]_{\Gamma,\Delta}] & [\![a[\gamma]]\!]_{\Gamma,A[\gamma]} &= [\![a]\!]_{\Delta,A}[[\![\gamma]\!]_{\Gamma,\Delta}] \\
[\![\mathtt{p}_A]\!]_{\Gamma.A,\Gamma} &= \mathtt{p}_A & [\![\mathtt{q}_A]\!]_{\Gamma.A,A[p]} &= \mathtt{q}_A \\
[\![o]\!]_1 &= A
\end{aligned}$$

Note that $\Delta = \mathrm{dom}(\gamma') = \mathrm{cod}(\gamma)$ in the equation for $\circ$, etc.

We then prove by induction on the inference rules that:

▶ **Lemma 9.** *If $\Gamma = \Gamma' \vdash$, then $[\![\Gamma]\!] = [\![\Gamma']\!]$ and both are defined.*
- *If $\Gamma \vdash \gamma = \gamma' : \Delta$, then $[\![\gamma]\!]_{\Gamma,\Delta} = [\![\gamma']\!]_{\Gamma,\Delta}$ and both are defined.*
- *If $\Gamma \vdash A = A'$, then $[\![A]\!]_{\Gamma} = [\![A']\!]_{\Gamma}$ and both are defined.*
- *If $\Gamma \vdash a = a' : A$, then $[\![a]\!]_{\Gamma,A} = [\![a']\!]_{\Gamma,A}$ and both are defined.*

Hence the partial interpretation lifts to the quotient of syntax by judgmental equality:

$$\begin{aligned}
\overline{[\![-]\!]} &: \mathrm{Ctx}_{\mathcal{T}} \to \mathrm{Ctx}_{\mathcal{C}} \\
\overline{[\![-]\!]}_{[\Gamma],[\Delta]} &: \mathrm{Sub}_{\mathcal{T}}([\Gamma],[\Delta]) \to \mathrm{Sub}_{\mathcal{C}}(\overline{[\![\Gamma]\!]},\overline{[\![\Delta]\!]}) \\
\overline{[\![-]\!]}_{[\Gamma]} &: \mathrm{Ty}_{\mathcal{T}}([\Gamma]) \to \mathrm{Ty}_{\mathcal{C}}(\overline{[\![\Gamma]\!]}) \\
\overline{[\![-]\!]}_{[\Gamma],[A]} &: \mathrm{Tm}_{\mathcal{T}}([\Gamma],[A]) \to \mathrm{Tm}_{\mathcal{C}}(\overline{[\![\Gamma]\!]},\overline{[\![A]\!]}_{[\Gamma]})
\end{aligned}$$

This defines a strict cwf morphism $\mathcal{T} \to \mathcal{C}$ which maps $[o]$ to A. It is easy to check that it is the unique such strict cwf-morphism. Hence we have proved:

▶ **Theorem 10.** *$\mathcal{T}$ is the free cwf on one object, that is, for every other cwf $\mathcal{C}$ and $A \in \mathrm{Ty}_{\mathcal{C}}(1)$ there is a unique strict cwf morphism $\mathcal{T} \to \mathcal{C}$ which maps $[o]$ to A.*

From now on we will uniformly drop the equivalence class brackets and for example write Γ for $[\Gamma]$. There should be no risk of confusion, but we remark that proofs by induction on syntax and inference rules are on representatives rather than equivalence classes.

3.4 Bifreeness of $\mathcal{T}$

We recall that an object I is bi-initial in a 2-category iff for any object A there exists an arrow $I \to A$ and for any two arrows $f, g : I \to A$ there exists a unique 2-cell $\theta : f \Rightarrow g$. It follows that θ is invertible. It also follows that bi-initial objects are equivalent.

▶ **Definition 11.** A cwf $\mathcal{C}$ is bifree on one base type iff it is bi-initial in the 2-category $\mathbf{CwF}^o$:

- *Objects*: Pairs $(\mathcal{C}, o_\mathcal{C})$ of a CwF and a chosen $o_\mathcal{C} \in \mathrm{Ty}_\mathcal{C}(1)$,
- *1-cells from* $(\mathcal{C}, o_\mathcal{C})$ *to* $(\mathcal{D}, o_\mathcal{D})$: pseudo cwf-morphisms $F : \mathcal{C} \to \mathcal{D}$ such that there exists $\varphi_F : F(o_\mathcal{C}) \cong o_\mathcal{D}$ in $\mathrm{Ty}_\mathcal{D}(1)$,
- *2-cells from* F *to* G *with type* $(\mathcal{C}, o_\mathcal{C}) \to (\mathcal{D}, o_\mathcal{D})$: pseudo cwf-transformations (φ, ψ) from F to G satisfying $\psi_{o_\mathcal{C}} = \alpha_G^{-1} \circ \alpha_F : F(o_\mathcal{C}) \cong G(o_\mathcal{C})$.

▶ **Theorem 12.** *$\mathcal{T}$ is a bifree cwf on one base type.*

We have showed that for every cwf $\mathcal{C}$, $A \in \mathrm{Ty}_\mathcal{C}(1)$, the interpretation $\overline{[\![-]\!]}$ is a strict cwf-morphism mapping o to A. Hence it is a morphism in $\mathbf{CwF}^o$. It remains to show that for any other morphism $F : \mathcal{T} \to \mathcal{C}$ in $\mathbf{CwF}^o$, there is a unique 2-cell (pseudo cwf-transformation) $(\varphi, \psi) : \overline{[\![-]\!]} \to F$, which happens to be an isomorphism. This asymmetric version of bi-initiality is equivalent to that given above because the 2-cell we build is an isomorphism.

Existence of (φ, ψ)

We construct (φ, ψ) by induction on the inference rules and simultaneously prove their naturality properties:

- If $\Gamma = \Gamma' \vdash$, then there exist $\varphi_\Gamma = \varphi_{\Gamma'} : \overline{[\![\Gamma]\!]} \cong F\Gamma$.
- If $\Gamma \vdash A = A'$, then there exist $\psi_A = \psi_{A'} : \overline{[\![\Gamma.A]\!]} \cong_{\overline{[\![\Gamma]\!]}} \overline{[\![\Gamma]\!]}.FA[\varphi_\Gamma]$.
- If $\Gamma \vdash \gamma = \gamma' : \Delta$, then $F\gamma \circ \varphi_\Gamma = \varphi_\Delta \circ \overline{[\![\gamma]\!]}$
- If $\Gamma \vdash a = a' : A$, then $\{\psi_A\}(\overline{[\![a]\!]}) = Fa[\varphi_\Gamma]$

Since it also follows that $\varphi_{\Gamma.A} = \rho^{-1} \circ \varphi_\Gamma^+ \circ \psi_A$ we conclude that (φ, ψ) is a pseudo cwf-transformation. For space reasons we only present the proofs of the first two items and refer the reader to the long version of the paper [4] for the other two.

Empty context. F preserves terminal objects, thus we let $\phi_1 : \overline{[\![1]\!]} = 1_\mathcal{C} \cong F1$.

Context extension. By induction, we have $\psi_A : \overline{[\![A]\!]} \cong_\Gamma FA[\varphi_\Gamma]$. We define $\varphi_{\Gamma.A}$ as the following composition of isomorphisms:

$$\varphi_{\Gamma.A} = \overline{[\![\Gamma.A]\!]} \xrightarrow{\psi_A} \overline{[\![\Gamma]\!]}.FA[\varphi_\Gamma] \xrightarrow{\langle \varphi_\Gamma, \mathtt{q} \rangle} F\Gamma.FA \xrightarrow{\rho^{-1}_{\Gamma,A}} F(\Gamma.A)$$

Type substitution. Let $\Gamma \vdash \gamma : \Delta$ and $\Delta \vdash A$. By induction we get $\varphi_\Delta \circ \overline{[\![\gamma]\!]} = F\gamma \circ \varphi_\Gamma$ and $\psi_A : \overline{[\![A]\!]} \cong_\Delta FA[\varphi_\Delta]$. Since $\mathbf{T}$ is a functor, $\mathbf{T}\gamma$ is a functor from $\mathbf{T}\Delta$ to $\mathbf{T}\Gamma$ thus,

$$\mathbf{T}(\overline{[\![\gamma]\!]})(\psi_A) : \overline{[\![A[\gamma]]\!]} \cong_\Gamma FA[\varphi_\Delta \circ \gamma] = FA[F\gamma][\varphi_\Gamma]$$

by induction hypothesis on γ. So we define

$$\psi_{A[\gamma]} = \mathbf{T}(\varphi_\Gamma)(\theta_{A,\gamma}) \circ \mathbf{T}(\overline{[\![\gamma]\!]})(\psi_A) : \overline{[\![A[\gamma]]\!]} \cong_{\overline{[\![\Gamma]\!]}} (F(A[\gamma]))[\varphi_\Gamma]$$

Using the previous case we can get a simpler equation for $\varphi_{\Gamma.A[\gamma]}$:

$$\varphi_{\Gamma.A[\gamma]} = \langle \varphi_\Gamma \circ \mathtt{p}, \mathtt{q}[\rho \circ \varphi_{\Delta.A} \circ \gamma \uparrow A] \rangle : \overline{[\![\Gamma.A[\gamma]]\!]} \to F(\Gamma.A[\gamma])$$

Base type. By definition, F is equipped with $\alpha_F : \overline{[\![o]\!]} \cong F(o)$. We define $\psi_o = \alpha_F^{-1} : \overline{[\![o]\!]} \cong F(o)$ in $\mathrm{Ty}_\mathcal{C}(1)$.

Uniqueness of (φ, ψ)

Let $(\varphi', \psi') : \overline{[\![\cdot]\!]} \to F$ be another pseudo cwf-transformation in $\mathbf{CwF}^o$. We prove the following by induction:

- If $\Gamma \vdash$, then $\varphi_\Gamma = \varphi'_\Gamma$
- If $\Gamma \vdash A$, then $\psi_A = \psi'_A$

Empty context. There is a unique morphism between the terminal objects $\overline{[\![1]\!]}$ and $F1$, so $\varphi_1 = \varphi'_1$.

Context extension. Assume by induction $\varphi_\Gamma = \varphi'_\Gamma$ and $\psi_A = \psi'_A$. By the coherence law of pseudo cwf-transformations, we have $\varphi'_{\Gamma.A} = \rho^{-1} \circ {\varphi'_\Gamma}^+ \circ \psi'_A$ from which the equality $\varphi'_{\Gamma.A} = \varphi_{\Gamma.A}$ follows.

Type substitution. Assume we have $\Delta \vdash A$ and $\Gamma \vdash \gamma : \Delta$, and consider $\psi_{A[\gamma]}$ and $\psi'_{A[\gamma]}$. By definition of pseudo cwf-transformations, one has $\mathbf{T}(\varphi'_\Gamma)(\theta^{-1}_{A,\gamma}) \circ \psi'_{A[\gamma]} = \mathbf{T}(F\gamma)(\psi'_A)$. Since we know $\varphi_\Gamma = \varphi'_\Gamma$ we know φ'_Γ is an isomorphism and thus $\psi'_{A[\gamma]}$ depends only on φ'_Γ and ψ'_A from which it follows that $\psi'_{A[\gamma]} = \psi'_{A[\gamma]}$.

Base type. The definition of 2-cells in $\mathbf{CwF}^o$ entails $\psi'_o = \alpha_F^{-1} : \overline{[\![o]\!]} \to F([\![o]\!])$.

4 A free lccc

4.1 From cwfs to lcccs

We now extend our cwf-calculus with extensional I-types, N_1, Σ, and Π and prove that it yields a free cwf supporting these type formers. To show that this yields a free lccc we apply the biequivalence [6] between lcccs and *democratic* cwfs supporting these type formers.

▶ **Definition 13** (Democratic cwfs). A cwf $\mathcal{C}$ is democratic when for each context Γ there is a type $\bar{\Gamma} \in \mathrm{Ty}(1)$ such that $\Gamma \cong 1.\bar{\Gamma}$. A pseudo cwf morphism $F : \mathcal{C} \to \mathcal{D}$ between democratic cwf preserves democracy when there is an isomorphism $F(\bar{\Gamma}) \cong \overline{F\Gamma}[\langle\rangle_\Gamma]$ satisfying a coherence diagram stated in [6] (Definition 8).

The free cwf with N_1, Σ, and Π-types is democratic since the empty context can be represented by the unit type N_1 and context extension by a Σ-type.

4.2 Cwfs with support for type constructors

A cwf supports a certain type former when it has extra structure and equations corresponding to the to formation, introduction, elimination, and equality rules for the type former in question. We only spell out what it means for a cwf to support and preserve extensional identity types and refer the reader to [6] for the definitions wrt Σ- and Π-types. The definition of what it means to support and preserve N_1 is analogous.

▶ **Definition 14** (Cwf with identity types). A cwf $\mathcal{C}$ is said to support extensional identity types when for each $a, a' \in \mathrm{Tm}_\mathcal{C}(\Gamma, A)$ there is a type $\mathrm{I}(A, a, a') \in \mathrm{Ty}_\mathcal{C}(\Gamma)$ satisfying the following condition:

- $\mathrm{I}(A, a, a')[\gamma] = \mathrm{I}(A[\gamma], a[\gamma], a'[\gamma])$ for any $\gamma : \Delta \to \Gamma$
- For $a \in \mathrm{Tm}_\mathcal{C}(\Gamma, A)$, there exists $\mathrm{r}(a) \in \mathrm{Tm}_\mathcal{C}(\Gamma, \mathrm{I}(A, a, a))$. Moreover, if $c \in \mathrm{Tm}_\mathcal{C}(\Gamma, \mathrm{I}(A, a, a'))$ then $a = a'$ and $c = \mathrm{r}(a)$.

A pseudo cwf morphism F preserves identity types when

$$\mathrm{I}(FA, Fa, Fa') \cong_\Gamma F(\mathrm{I}(A, a, a')).$$

We write $\mathbf{CwF}_d^{\mathrm{I},\Sigma,\Pi}$ for the 2-category of democratic cwfs supporting I, Σ, and Π with morphisms preserving them, and $\mathbf{CwF}_{s,d}^{\Sigma,\Pi,I}$ for the strict version. Note that by democracy, any democratic cwf has a unit type representing the empty context.

Σ and Π are functorial and preserve isomorphisms:

▶ **Lemma 15** (Functoriality of Σ). *Let $f_A : A \cong_\Gamma A'$ and $f_B : B \cong_{\Gamma.A} B'[f_A]$ with $\Gamma.A \vdash B$ and $\Gamma.A' \vdash B'$. Then $\Sigma(A,B) \cong \Sigma(A',B')$ functorially: if $g_A : A' \cong A''$ and $g_B : B' \cong B''[g_A]$, then $\Sigma(g_A \circ f_A, g_B \circ f_A^+ \circ f_B) = \Sigma(g_A, g_B) \circ \Sigma(f_A, f_B) : \Sigma(A,B) \to \Sigma(A'',B'')$.*

▶ **Lemma 16** (Functoriality of Π). *Let $f_A : A \cong_\Gamma A'$ and $f_B : B \cong_{\Gamma.A} B'[f_A]$. Then there is a type isomorphism $\Pi(f_A, f_B) : \Pi(A,B) \cong_\Gamma \Pi(A',B')$ functorially.*

4.3 The syntactic cwf with extensional I, N_1, Σ, and Π

We extend the grammar and the set of inference rules with rules for $\mathrm{I}, \mathrm{N}_1, \Sigma$, and Π-types:

$$\begin{aligned} A &::= \cdots \mid \mathrm{I}(a,a) \mid \mathrm{N}_1 \mid \Sigma(A,A) \mid \Pi(A,A) \\ a &::= \cdots \mid \mathrm{r}(a) \mid 0_1 \mid \mathrm{fst}(A,a) \mid \mathrm{snd}(A,A,a) \mid \mathrm{pair}(A,A,a,a) \mid \mathrm{ap}(A,A,a,a) \mid \lambda(A,a) \end{aligned}$$

For each type we define its context:

$$\begin{aligned} \text{ctx-of}(\mathrm{I}(a,a')) &= \text{ctx-of}(\text{type-of}(a)) \qquad \text{ctx-of}(\Sigma(A,B)) = \text{ctx-of}(A) \\ \text{ctx-of}(\Pi(A,B)) &= \text{ctx-of}(A) \end{aligned}$$

For each term we define its type:

$$\begin{aligned} \text{type-of}(\mathrm{fst}(A,c)) &= A \\ \text{type-of}(\mathrm{snd}(A,B,c)) &= B\,\langle \mathtt{id}_{\text{ctx-of}(A)}, \mathrm{fst}(A,c)\rangle \qquad \text{type-of}(\mathrm{r}(a)) = \mathrm{I}(a,a) \\ \text{type-of}(\mathrm{pair}(A,B,a,b)) &= \Sigma(A,B) \qquad \text{type-of}(\lambda(A,c)) = \Pi(A, \text{type-of}(c)) \\ \text{type-of}(\mathrm{pair}(A,B,a,b)) &= \Sigma(A,B) \qquad \text{type-of}(\mathrm{ap}(A,B,c,a)) = B\,\langle \mathtt{id}_{\text{ctx-of}(A)}, a\rangle \end{aligned}$$

4.3.1 Inference rules

Rules for I-types:

$$\frac{\Gamma \vdash a = a' : A \qquad \Gamma \vdash b = b' : A}{\Gamma \vdash \mathrm{I}(a,b) = \mathrm{I}(a',b')} \qquad \frac{\Gamma \vdash a = a' : A}{\Gamma \vdash \mathrm{r}(a) = \mathrm{r}(a') : \mathrm{I}(a,a')} \qquad \frac{\Gamma \vdash c : \mathrm{I}(a,a')}{\Gamma \vdash a = a' : \text{type-of}(a)}$$

$$\frac{\Gamma \vdash c : \mathrm{I}(a,a')}{\Gamma \vdash c = \mathrm{r}(a) : \mathrm{I}(a,a')} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash a' : A \qquad \Delta \vdash \gamma : \Gamma}{\Gamma \vdash \mathrm{I}(a,a')[\gamma] = \mathrm{I}(a[\gamma], a'[\gamma])}$$

Rules for N_1:

$$\frac{}{\vdash \mathrm{N}_1} \qquad \frac{}{\vdash 0_1 : \mathrm{N}_1} \qquad \frac{\vdash a : \mathrm{N}_1}{\vdash a = 0_1 : \mathrm{N}_1}$$

Rules for Σ-types:

$$\frac{\Gamma \vdash A = A' \quad \Gamma.A \vdash B = B'}{\Gamma \vdash \Sigma(A,B) = \Sigma(A',B')} \qquad \frac{\Gamma \vdash A = A' \quad \Gamma \vdash c = c' : \Sigma(A,B)}{\Gamma \vdash \mathrm{fst}(A,c) = \mathrm{fst}(A',c') : A}$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma.A \vdash B = B' \quad \Gamma \vdash c = c' : \Sigma(A,B)}{\Gamma \vdash \mathrm{snd}(A,B,c) = \mathrm{snd}(A',B',c') : B\,\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle}$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma.A \vdash B = B' \quad \Gamma \vdash a = a' : A' \quad \Gamma \vdash b = b' : B\,\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle}{\Gamma \vdash \mathrm{pair}(A,B,a,b) = \mathrm{pair}(A',B',a',b') : \Sigma(A,B)}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Gamma \vdash a : A' \quad \Gamma \vdash b : B\,\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle}{\Gamma : \mathrm{fst}(A, \mathrm{pair}(A,B,a,b)) = a : A}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Gamma \vdash a : A' \quad \Gamma \vdash b : B\,\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle}{\Gamma : \mathrm{snd}(A,B, \mathrm{pair}(A,B,a,b)) = b : B\,\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle]}{\Gamma \vdash c = \mathrm{pair}(A,B,\mathrm{fst}(A,c),\mathrm{snd}(A,B,c)) : \Sigma(A,B)}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Delta \vdash \gamma : \Gamma}{\Gamma \vdash \Sigma(A,B)[\gamma] = \Sigma(A[\gamma], B[\gamma^+])} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash c : \Sigma(A,B) \quad \Delta \vdash \gamma : \Gamma}{\Gamma \vdash \mathrm{fst}(A,c)[\gamma] = \mathrm{fst}(A[\gamma], c[\gamma]) : A}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Gamma \vdash c : \Sigma(A,B) \quad \Delta \vdash \gamma : \Gamma}{\Gamma \vdash \mathrm{snd}(A,B,c)[\gamma] = \mathrm{snd}(A[\gamma], B[\gamma^+], c[\gamma]) : B[\langle \gamma, \mathrm{fst}(A,c)[\gamma]\rangle]}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[\langle \mathtt{id}_\Gamma, \mathrm{fst}(A,c)\rangle] \quad \Delta \vdash \gamma : \Gamma}{\Gamma \vdash \mathrm{pair}(A,B,a,b)[\gamma] = \mathrm{pair}(A[\gamma], B[\gamma^+], a[\gamma], b[\gamma]) : \Sigma(A,B)[\gamma]}$$

Rules for Π-types:

$$\frac{\Gamma \vdash A = A' \quad \Gamma.A \vdash B = B'}{\Gamma \vdash \Pi(A,B) = \Pi(A',B')} \qquad \frac{\Gamma \vdash A = A' \quad \Gamma.A \vdash b = b' : B}{\Gamma \vdash \lambda(A,b) = \lambda(A',b') : \Pi(A,B)}$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma.A \vdash B = B' \quad \Gamma \vdash c = c' : \Pi(A,B) \quad \Gamma \vdash a = a' : A}{\Gamma \vdash \mathrm{app}(A,B,c,a) = \mathrm{app}(A',B',c',a') : B[\langle \mathtt{id}, a\rangle]}$$

$$\frac{\Gamma \vdash c : \Pi(A,B) \quad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{app}(A,B,\lambda(A,b),a) = b[\langle \mathtt{id}, a\rangle] : B[\langle \mathtt{id}, a\rangle]} \qquad \frac{\Gamma \vdash c : \Pi(A,B)}{\Gamma \vdash \lambda(\mathrm{app}(c[p], q)) = c : \Pi(A,B)}$$

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \Pi(A,B)[\gamma] = \Pi(A[\gamma], A[\gamma^+])} \qquad \frac{\Gamma \vdash c : \Pi(A,B) \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \lambda(b)[\gamma] = \lambda(b[\gamma^+]) : \Pi(A,B)[\gamma]}$$

$$\frac{\Gamma \vdash c : \Pi(A,B) \quad \Gamma \vdash a : A \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathrm{app}(c,a)[\gamma] = \mathrm{app}(c[\gamma], a[\gamma]) : B[\langle \gamma, a[\gamma]\rangle]}$$

4.3.2 The syntactic cwf supporting $\mathrm{I}, \mathrm{N}_1, \Sigma$, and Π

It is straightforward to extend the definition of the term model $\mathcal{T}$ with $\mathrm{I}, \mathrm{N}_1, \Sigma$, and Π-types, to form a cwf $\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$ supporting these type constructors. As we already explained it is democratic.

We want to show that $\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$ is free, not only in the 2-category of cwfs supporting I,Σ,Π but in the subcategory of the democratic ones. (Democracy entails that N_1 is also supported.) It is straightforward to extend the interpretation functor and prove its uniqueness. It is also easy to check that it preserves democracy.

▶ **Theorem 17.** *$\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$ is the free democratic cwf supporting I,Σ,Π on one object.*

4.4 Bifreeness of $\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$

We now prove the key result:

▶ **Theorem 18.** *$\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$ is the bifree democratic cwf supporting I,Σ,Π on one object.*

This means that $\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$ is bi-initial in the 2-category $\mathbf{CwF}_d^{\mathrm{I},\Sigma,\Pi,o}$ where objects are democratic cwfs which support I,Σ,Π, and a base type o, and where morphisms preserve these type formers up to isomorphism.

4.4.1 Existence

We resume our inductive proof from Section 3.4 with the cases for $\mathrm{I},\mathrm{N}_1,\Sigma$, and Π.

Unit type. Since F preserves democracy and terminal object it follows that $1.F(\mathrm{N}_1) = 1.F(\bar{1}) \cong \overline{F1} \cong \bar{1}$.

Identity type. Assume $\Gamma \vdash a, a' : A$. By induction, we have $\psi_A : \overline{[\![A]\!]} \cong_{\overline{[\![\Gamma]\!]}} FA[\varphi_\Gamma]$. We know I-types preserve isomorphisms in the indexed category (Lemma 10, page 35 of [6]) yielding (over $\overline{[\![\Gamma]\!]}$):

$$\begin{aligned}\psi_{\mathrm{I}(a,a')} : \overline{[\![\mathrm{I}(a,a')]\!]} &\cong \mathrm{I}_{\mathcal{C}}(FA[\varphi_\Gamma], \{\psi_A\}(\overline{[\![a]\!]}), \{\psi_A\}(\overline{[\![a']\!]}))\\ &= \mathrm{I}_{\mathcal{C}}(FA[\varphi_\Gamma], F(a)[\varphi_\Gamma], F(a')[\varphi_\Gamma])\,.\end{aligned}$$

Σ-types. Assume we have $\Gamma \vdash A$ and $\Gamma.A \vdash B$. By induction we have the isomorphisms $\psi_A : \overline{[\![A]\!]} \cong_\Gamma FA[\varphi_\Gamma]$ and $\psi_B : \overline{[\![B]\!]} \cong_{\Gamma.A} FB[\varphi_{\Gamma.B}]$. We let

$$\begin{aligned}\psi_{\Sigma(A,B)} = \ \Gamma.\Sigma(A,B) &\xrightarrow{\Sigma(\psi_A,\psi_B)} \Gamma.\Sigma(FA[\varphi_\Gamma], FB[\rho^{-1}\circ {\varphi_\Gamma^{-1}}^{+}])\\ &\xrightarrow{\mathbf{T}(\varphi_\Gamma)(s_{A,B}^{-1})} \Gamma.F(\Sigma(A,B))[\varphi_\Gamma]\end{aligned}$$

$\psi_{\Sigma(A,B)}$ can be related to $\varphi_{\Gamma.A.B}$:

$$\begin{aligned}\psi_{\Sigma(A,B)} &= \mathbf{T}(\varphi_\Gamma)(s_{A,B}^{-1}) \circ \Sigma(\psi_A,\psi_B)\\ &= {\varphi_\Gamma^{-1}}^{+} \circ s_{A,B}^{-1} \circ {\varphi_\Gamma}^{+} \circ \chi^{-1} \circ \psi_A^{+} \circ \psi_B \circ \chi_{A,B}\\ &= {\varphi_\Gamma^{-1}}^{+} \circ s_{A,B}^{-1} \circ \chi_{A,B} \circ {\varphi_\Gamma}^{++} \circ \psi_A^{+} \circ \psi_B \circ \chi_{A,B}\\ &= {\varphi_\Gamma^{-1}}^{+} \circ \rho \circ F(\chi_{A,B}) \circ \rho^{-1} \circ {\rho^{-1}}^{+} \circ {\varphi_\Gamma}^{++} \circ \psi_A^{+} \circ \psi_B \circ \chi_{A,B}\\ &= {\varphi_\Gamma^{-1}}^{+} \circ \rho \circ F(\chi_{A,B}^{-1}) \circ \rho^{-1} \circ \varphi_{\Gamma.A}^{+} \circ \psi_B \circ \chi_{A,B}\\ &= {\varphi_\Gamma^{-1}}^{+} \circ \rho \circ F(\chi_{A,B}^{-1}) \circ \varphi_{\Gamma.A.B} \circ \chi_{A,B}\end{aligned}$$

From that calculation, we deduce $\varphi_{\Gamma.\Sigma(A,B)} = F(\chi_{A,B}^{-1}) \circ \varphi_{\Gamma.A.B} \circ \chi_{A,B}$.

Π-types. Define $\psi_{\Pi(A,B)}$ as follows

$$\begin{aligned}\overline{[\![\Gamma.\Pi(A,B)]\!]} &\xrightarrow{\Pi(\psi_A,\psi_B)} \overline{[\![\Gamma]\!]}.\Pi(FA[\varphi_\Gamma], FB[\rho\circ\varphi_{\Gamma\uparrow FA}])\\ &= \overline{[\![\Gamma]\!]}.\Pi(FA, FB[\rho])[\varphi_\Gamma]\\ &\xrightarrow{T(\varphi_\Gamma)(\xi_{A,B}^{-1})} \overline{[\![\Gamma]\!]}.F(\Pi(A,B))[\varphi_\Gamma]\end{aligned}$$

4.4.2 Uniqueness

We resume the uniqueness proof from 3.4.

The unit type. It follows from the preservation of democracy of F.

Identity types. We need to show $\psi'_{\mathrm{I}(a,a')} = \psi_{\mathrm{I}(a,a')} : \Gamma.\mathrm{I}(a,a') \to \Gamma.F(\mathrm{I}(a,a'))[\varphi_\Gamma]$. Let $A = \text{type-of}(a)$. By post-composing with the coherence isomorphism $F(\mathrm{I}(a,a')) \cong_{F\Gamma} \mathrm{I}(FA, Fa, Fa')$, we get a morphism between identity types. In an extensional type theory, identity types are either empty or singletons, thus there is at most one morphism between two identity types (which is an isomorphism). This implies that $\psi_{\mathrm{I}(a,a')} = \psi'_{\mathrm{I}(a,a')}$.

Σ-types. By induction, we assume that $\varphi_{\Gamma.A.B} = \varphi'_{\Gamma.A.B}$. By naturality of φ', we have $\varphi'_{\Sigma(A,B)} = F(\chi^{-1}_{A,B}) \circ \varphi'_{\Gamma.A.B} \circ \chi_{A,B} = \varphi_{\Gamma.\Sigma(A,B)}$. Hence $\psi_{\Sigma(A,B)} = \psi'_{\Sigma(A,B)}$.

Π-types. As in the previous section, by induction we assume $\varphi_{\Gamma.A.B} = \varphi'_{\Gamma.A.B}$. Let $\mathtt{ev}$ be the obvious map $\Gamma.A.\Pi(A,B)[\mathtt{p}] \to \Gamma.A.B$. Proposition 11 of [6] entails:

▶ **Lemma 19.** *Assume $\Gamma.A \vdash B$. The only automorphism ω of $\Pi(A,B)$ (in $\boldsymbol{T}\Gamma$) such that $T\mathtt{p}(\omega) : \Gamma.A.\Pi(A,B)[\mathtt{p}] \cong \Gamma.A.\Pi(A,B)[\mathtt{p}]$ satisfies $\mathtt{ev} \circ T\mathtt{p}(\omega) = \mathtt{ev}$ is the identity.*

It remains to show that $\psi^{-1}_{\Pi(A,B)} \circ \psi'_{\Pi(A,B)}$ satisfies the condition. But we have:

$$\begin{aligned}
& F(\mathtt{ev}) \circ \rho^{-1} \circ \theta_{\Pi(A,B),\mathtt{p}} \circ \varphi_{\Gamma.A} \circ T\mathtt{p}(\psi'_{\Pi(A,B)}) \\
= {} & F(\mathtt{ev}) \circ \rho^{-1} \circ \varphi_{\Gamma.A} \circ T(\varphi_{\Gamma.A})(\theta) \circ T\mathtt{p}(\psi'_{\Pi(A,B)}) \\
= {} & F(\mathtt{ev}) \circ \rho^{-1} \circ \varphi_{\Gamma.A} \circ \psi_{\Pi(A,B)[\mathtt{p}]} \\
= {} & F(\mathtt{ev}) \circ \varphi_{\Gamma.A.\Pi(A,B)[\mathtt{p}]} \\
= {} & \varphi'_{\Gamma.A.B} \circ \mathtt{ev}
\end{aligned}$$

By only using naturality conditions on φ' and ψ'. Write $\tau : \Gamma.A.F(\Pi(A,B))[\varphi_\Gamma][\mathtt{p}] \to F(\Gamma.A.B)$ for the map $F(\mathtt{ev}) \circ \rho^{-1} \circ \theta_{\Pi(A,B),\mathtt{p}} \circ \varphi_{\Gamma.A}$. Since φ and ψ are natural, we can do the same reasoning, and have $\tau \circ T\mathtt{p}(\psi_{\Pi(A,B)}) = \varphi_{\Gamma.A.B} \circ \mathtt{ev}$. Thus, we get:

$$\varphi^{-1}_{\Gamma.A.B} \circ \tau = \mathtt{ev} \circ T\mathtt{p}(\psi^{-1}_{\Pi(A,B)})$$

Using our induction hypothesis on B ($\varphi_{\Gamma.A.B} = \varphi'_{\Gamma.A.B}$) we have

$$\mathtt{ev} \circ T\mathtt{p}(\psi^{-1}_{\Pi(A,B)}) \circ T\mathtt{p}(\psi'_{\Pi(A,B)}) = \varphi^{-1}_{\Gamma.A.B} \circ \tau \circ T\mathtt{p}(\psi'_{\Pi(A,B)}) = \mathtt{ev}$$

as desired. Hence $\psi_{\Pi(A,B)} = \psi'_{\Pi(A,B)}$.

4.5 The free lccc

Let **LCC** be the 2-category of lcccs. The biequivalence of [6] yields pseudofunctors:

$$U : \mathbf{CwF}_d^{\Sigma,\Pi,I} \to \mathbf{LCC} \quad H : \mathbf{LCC} \to \mathbf{CwF}_d^{\Sigma,\Pi,\mathrm{I}}$$

such that $UH = \mathrm{I}$ and $HU \cong \mathrm{I}$. In particular there are adjunctions $H \dashv U$ and $U \dashv H$.

▶ **Theorem 20.** *$U\mathcal{T}^{\mathrm{I},\mathrm{N}_1,\Sigma,\Pi}$ is the bifree lccc on one object, that is, it is bi-initial in $\boldsymbol{LCC}^o$.*

Proof. Let $\mathbb{C}$ be an lccc with a chosen object $o_\mathbb{C} \in \mathbb{C}$. By democracy $o_\mathbb{C}$ can be viewed as a type over the empty context in the cwf $H\mathbb{C}$, thus $(H\mathbb{C}, o)$ is in $\mathbf{CwF}_d^{\Sigma,\Pi,I,o}$. Thus we have a pseudo cwf functor $\overline{[\![\cdot]\!]} : \mathcal{T}^{\Sigma,\Pi,\mathrm{N}_1,I} \to H\mathbb{C}$ satisfying $[\![o]\!] \cong o_\mathbb{C}$. Hence $F : U\mathcal{T}^{\Sigma,\Pi,\mathrm{N}_1,I} \to \mathbb{C}$ in **LCC** because of the adjunction.

Assume we have another $G : U\mathcal{T}^{\Sigma,\Pi,\mathrm{N}_1,\mathrm{I}} \to \mathbb{C}$. Because of the adjunction we get $G^* : \mathcal{T}^{\Sigma,\Pi,\mathrm{N}_1,\mathrm{I}} \to H\mathbb{C}$. Thus by bifreeness of $\mathcal{T}^{\Sigma,\Pi,\mathrm{N}_1,I}$ we have $\varphi : \overline{[\![\cdot]\!]} \cong G^*$, thus $F \cong G$. Moreover, any other morphism $F \to G$ yields a morphism $\overline{[\![\cdot]\!]} \to G^*$ and is equal to φ. ◀

References

1 Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992.

2 Jean Benabou. Fibered categories and the foundations of naive category theory. *J. Symb. Log*, 50(1):10–37, 1985.

3 John Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

4 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Undecidability of equality in the free locally cartesian closed category. Available at `http://arxiv.org/abs/1504.03995`, 2015.

5 Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and martin-löf type theories. In *Typed Lambda Calculi and Applications – 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, pages 91–106, 2011.

6 Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6), 2014.

7 Pierre-Louis Curien. Substitution up to isomorphism. *Fundamenta Informaticae*, 19(1,2):51–86, 1993.

8 Peter Dybjer. Internal type theory. In *TYPES'95, Types for Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, pages 120–134. Springer, 1996.

9 Martin Hofmann. Interpretation of type theory in locally cartesian closed categories. In *Proceedings of CSL*. Springer LNCS, 1994.

10 Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

11 Per Martin-Löf. Substitution calculus. Notes from a lecture given in Göteborg, November 1992.

12 Erik Palmgren and Steve J. Vickers. Partial horn logic and cartesian categories. *Annals of Pure and Applied Logic*, 145(3):314 – 353, 2007.

13 Robert Seely. Locally cartesian closed categories and type theory. *Math. Proc. Cambridge Philos. Soc.*, 95(1):33–48, 1984.

14 Thomas Streicher. *Semantics of Type Theory*. Number 12 in Progress in Theoretical Computer Science. Basel: Birkhauser Verlag, 1991.

15 Alvaro Tasistro. Formulation of Martin-Löf's theory of types with explicit substitutions. Technical report, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, 1993. Licentiate Thesis.

The Inconsistency of a Brouwerian Continuity Principle with the Curry–Howard Interpretation

Martín Hötzel Escardó and Chuangjie Xu

School of Computer Science, University of Birmingham, UK

Abstract

If all functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are continuous then $0 = 1$. We establish this in intensional (and hence also in extensional) intuitionistic dependent-type theories, with existence in the formulation of continuity expressed as a Σ type via the Curry–Howard interpretation. But with an intuitionistic notion of anonymous existence, defined as the propositional truncation of Σ, it is consistent that all such functions are continuous. A model is Johnstone's topological topos. On the other hand, any of these two intuitionistic conceptions of existence give the same, consistent, notion of uniform continuity for functions $(\mathbb{N} \to \mathbb{2}) \to \mathbb{N}$, again valid in the topological topos. It is open whether the consistency of (uniform) continuity extends to homotopy type theory. The theorems of type theory informally proved here are also formally proved in Agda, but the development presented here is self-contained and doesn't show Agda code.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Dependent type, intensional Martin-Löf type theory, Curry–Howard interpretation, constructive mathematics, Brouwerian continuity axioms, anonymous existence, propositional truncation, function extensionality, homotopy type theory, topos theory

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.153

1 Introduction

We show that a continuity principle that holds in Brouwerian intuitionistic mathematics becomes false when we move to its Curry–Howard interpretation. We formulate and prove this in an intensional version of intuitionistic type theory (Section 2). Another Brouwerian (uniform) continuity principle, however, is logically equivalent to its Curry–Howard interpretation (Section 4).

In order to be able to formulate and prove this logical equivalence, we need a type theory in which both a formula and its Curry–Howard interpretation can be expressed (Section 3). For example, toposes admit both $\forall, \exists$ (via the subobject classifier) and Π, Σ (via their local cartesian closedness) and hence qualify. We adopt the HoTT-book [17] approach of working with propositional truncation $\|-\|$ to express $\exists(x : X).A(x)$ as the propositional truncation of $\Sigma(x : X).A(x)$. This is related to NuPrl's squash types [14], Maietti's mono-types [13], and Awodey–Bauer bracket types in extensional type theory [1]. Here by a proposition we mean a type whose elements are all equal in the sense of the identity type, as in the HoTT book. In a topos with identity types understood as equalizers, the propositions are the truth values (subterminal objects), and the propositional truncation of an object X is its support, namely the image of the unique map $X \to 1$ to the terminal object. This gives the truth value of the inhabitedness of X, without necessarily revealing an inhabitant of X, and we have that

$$(\exists(x : X).A(x)) = \|\Sigma(x : X).A(x)\|\,.$$

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 153–164

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In HoTT, this is taken as the definition of $\exists$, with truncation taken as a primitive notion. But we don't (need to) work with the homotopical understanding of type theory or the univalence axiom here.

1.1 The continuity of all functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$

In Brouwerian intuitionistic mathematics, all functions $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ on the Baire space $\mathbb{N}^{\mathbb{N}} = (\mathbb{N} \to \mathbb{N})$ are continuous [2, 19]. This means that, for any sequence $\alpha : \mathbb{N}^{\mathbb{N}}$ of natural numbers, the value $f\alpha$ of the function depends only on a finite prefix of the argument $\alpha : \mathbb{N}^{\mathbb{N}}$. If we write $\alpha =_n \beta$ to mean that the sequences α and β agree at their first n positions, a precise formulation of this continuity principle is

$$\forall(f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\ \forall(\alpha : \mathbb{N}^{\mathbb{N}}).\ \exists(n : \mathbb{N}).\ \forall(\beta : \mathbb{N}^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta.$$

It is well known that this statement cannot be proved in higher-type Heyting arithmetic (HA^ω), but that is consistent and validated by the model of Kleene–Kreisel continuous functionals, and also realizable with Kleene's second combinatory algebra K_2 [2].

We show that, in intensional Martin-Löf type theory, the Curry–Howard interpretation of the above continuity principle is false: It is a theorem of intensional MLTT, even without universes, that

$$\big(\Pi(f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\Pi(\alpha : \mathbb{N}^{\mathbb{N}}).\Sigma(n : \mathbb{N}).\Pi(\beta : \mathbb{N}^{\mathbb{N}}).\alpha =_n \beta \to f\alpha = f\beta\big) \to 0 = 1.$$

We prove this by adapting Kreisel's well-known argument that, e.g. in HA^ω, extensionality, choice and continuity are together impossible [12][18][2, page 267]. The difference here is that

1. We work in *intensional* type theory.
2. Choice for the Σ interpretation of existence is a theorem of type theory.

So what is left to understand is that extensionality is not needed in Kreisel's argument when it is rendered in type theory (Section 2).

The above two versions of the notion of continuity can be usefully compared by considering the interpretations of HA^ω and MLTT in Johnstone's *topological topos* [9]. The point of this topos is that it fully embeds a large cartesian closed category of continuous maps of topological spaces, the sequential topological spaces, and the larger locally cartesian closed category of Kuratowski limit spaces [15]. As discussed above, any topos has $\exists, \forall, \Sigma, \Pi$ and therefore models both intuitionistic predicate logic and dependent type theory. We have that

1. The formula

   $$\forall(f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\ \forall(\alpha : \mathbb{N}^{\mathbb{N}}).\ \exists(n : \mathbb{N}).\ \forall(\beta : \mathbb{N}^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

   is true in the topological topos.

   The informal reading of this is "all functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ are continuous".
2. There is a function

   $$(\Pi(f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\Pi(\alpha : \mathbb{N}^{\mathbb{N}}).\Sigma(n : \mathbb{N}).\Pi(\beta : \mathbb{N}^{\mathbb{N}}).\alpha =_n \beta \to f\alpha = f\beta) \to 0 = 1$$

   in the topological topos, or indeed in any topos whatsoever, by our version of Kreisel's argument.

   The informal reading of this is "not all functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ are continuous".

But there is no contradiction in the formal versions of the above statements: they simultaneously hold in the same world, the topological topos. From a hypothetical inhabitant of

$$\Pi(f\colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\Pi(\alpha : \mathbb{N}^{\mathbb{N}}).\Sigma(n : \mathbb{N}).\Pi(\beta : \mathbb{N}^{\mathbb{N}}).\alpha =_n \beta \to f\alpha = f\beta$$

we get a modulus-of-continuity functional

$$M : (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}) \times \mathbb{N}^{\mathbb{N}} \to \mathbb{N},$$

by projection (rather than by choice in the topos-logic sense), which gives a modulus of continuity $n = M(f, \alpha)$ of the function $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ at the point $\alpha : \mathbb{N}^{\mathbb{N}}$. Kreisel's argument derives a contradiction from the existence of M. What this shows, then, is that although every function *is* continuous, there is no continuous way of finding a modulus of continuity of a given function f at a given point α. There is no continuous M. Perhaps the difference between the seemingly contradictory statements becomes clearer if we formulate them type theoretically with and without propositional truncation. In the topological topos, the object

$$\Pi(f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\ \Pi(\alpha : \mathbb{N}^{\mathbb{N}}).\ \|\Sigma(n : \mathbb{N}).\ \Pi(\beta : \mathbb{N}^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta\|$$

is inhabited, but

$$\Pi(f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\ \Pi(\alpha : \mathbb{N}^{\mathbb{N}}).\ \Sigma(n : \mathbb{N}).\ \Pi(\beta : \mathbb{N}^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

is not.

1.2 The uniform continuity of all functions $2^{\mathbb{N}} \to \mathbb{N}$

The above situation changes radically when we move from the Baire space to the Cantor space, and from continuous functions to uniformly continuous functions. Another Brouwerian continuity principle is that all functions from the Cantor space $2^{\mathbb{N}} = (\mathbb{N} \to 2)$ to the natural numbers are uniformly continuous:

$$\forall(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \exists(n\colon \mathbb{N}).\ \forall(\alpha, \beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta.$$

Again this is not provable in HA^{ω} but consistent and validated by the model of continuous functionals, by realizability over K_2, and by the topological topos. We have also constructively developed a model analogous to the topological topos in [20].

By the above discussion, the above principle is equivalent to

$$\Pi(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \|\Sigma(n\colon \mathbb{N}).\ \Pi(\alpha, \beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta\|.$$

We show that this, in turn, is logically equivalent to its untruncated version

$$\Pi(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \Sigma(n\colon \mathbb{N}).\ \Pi(\alpha, \beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta.$$

In particular, it follows that this object is inhabited (by a global point) in the topological topos. Each inhabitant gives, by projection, a "fan functional" $(2^{\mathbb{N}} \to \mathbb{N}) \to \mathbb{N}$ that continuously assigns a modulus of uniform continuity to its argument. There is a canonical one, which assigns the least modulus of uniform continuity.

In order to establish the above logical equivalence, we prove the following general principle for "exiting truncations": If A is a family of types indexed by natural numbers such that

1. $A(n)$ is a proposition for every $n\colon \mathbb{N}$, and
2. $A(n)$ implies that $A(m)$ is decidable for every $m < n$,

then

$$\|\Sigma(n{:}\mathbb{N}).A(n)\| \to \Sigma(n{:}\mathbb{N}).A(n).$$

From anonymous existence one gets explicit existence in this case.

1.3 A question regarding Church's Thesis

Troelstra [18] also shows that extensionality, choice and Church's Thesis (CT) are together impossible, and Beeson [2, page 268] adapts this argument to conclude that extensional Martin-Löf type theory refutes CT, with existence expressed by Σ. But CT with existence expressed as the truncation of Σ is consistent with MLTT, and validated by Hyland's *effective topos* [8]. What seems to be open is whether CT formulated with Σ is already refuted by *intensional* MLTT (including the ξ-rule). This question has been popularized by Maria Emilia Maietti.

2 Continuity of functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$

We reason informally, but rigorously, in type theory, where, as above, we use the equality sign to denote identity types, unless otherwise indicated. A formal proof, written in Agda [3, 4, 16], is available at [6], but the development here is self-contained and doesn't show Agda code.

The following says that the Curry–Howard interpretation of "all functions $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ are continuous" is false.

▶ **Theorem 1.** *If*

$$\Pi(f\colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\Pi(\alpha\colon \mathbb{N}^{\mathbb{N}}).\Sigma(n\colon \mathbb{N}).\Pi(\beta\colon \mathbb{N}^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

then $0 = 1$.

We take the conclusion to be $0 = 1$ rather than the empty type because we are not assuming a universe for the sake of generality. The argument below gives $0 = 1$, and, as is well known, to get to the empty type from $0 = 1$ a universe is needed.

Proof. Let 0^ω denote the infinite sequence of zeros, that is, $\lambda i.0$, and let $0^n k^\omega$ denote the sequence of n many zeros followed by infinitely many k's. Then

$$(0^n k^\omega) =_n 0^\omega \quad \text{and} \quad (0^n k^\omega)(n) = k.$$

Assume $\Pi(f\colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\Pi(\alpha\colon \mathbb{N}^{\mathbb{N}}).\Sigma(n\colon \mathbb{N}).\Pi(\beta\colon \mathbb{N}^{\mathbb{N}}).\alpha =_n \beta \to f\alpha = f\beta$. By projection, with $\alpha = 0^\omega$, this gives a modulus-of-continuity function

$$M\colon (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}) \to \mathbb{N}$$

such that

$$\Pi(f\colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}).\Pi(\beta\colon \mathbb{N}^{\mathbb{N}}).0^\omega =_{Mf} \beta \to f(0^\omega) = f\beta. \tag{1}$$

We use M to define a function $f\colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ such that $M(f)$ cannot be a modulus of continuity of f and hence get a contradiction. Let

$$m = M(\lambda\alpha.0),$$

and define $f\colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ by

$$f\beta = M(\lambda\alpha.\beta(\alpha m)).$$

The crucial observation is that, by simply expanding the definitions, we have the judgemental equalities

$$f(0^\omega) = M(\lambda\alpha.0^\omega(\alpha m)) = M(\lambda\alpha.0) = m,$$

because $0^\omega(\alpha m) = 0$. By the defining property (1) of M, and the crucial observation,

$$\Pi(\beta\colon \mathbb{N}^{\mathbb{N}}).0^\omega =_{Mf} \beta \to m = f\beta. \tag{2}$$

For any $\beta\colon \mathbb{N}^{\mathbb{N}}$, by the continuity of $\lambda\alpha.\beta(\alpha m)$, by the definition of f, and by the defining property (1) of M, we have that

$$\Pi(\alpha\colon \mathbb{N}^{\mathbb{N}}).0^\omega =_{f\beta} \alpha \to \beta 0 = \beta(\alpha m).$$

If we choose $\beta = 0^{Mf+1}1^\omega$, we have $0^\omega =_{Mf+1} \beta$, and so $0^\omega =_{Mf} \beta$, and hence $f(\beta) = m$ by (2). This gives

$$\Pi(\alpha\colon \mathbb{N}^{\mathbb{N}}).0^\omega =_{m} \alpha \to \beta 0 = \beta(\alpha m).$$

Considering $\alpha = 0^m(Mf+1)^\omega$, we have $0^\omega =_m \alpha$, and therefore

$$0 = \beta 0 = \beta(\alpha m) = \beta(Mf+1) = 1. \qquad \blacktriangleleft$$

▶ Remark (Thomas Streicher, personal communication). The conversion

$$f(0^\omega) = M(\lambda\alpha.0^\omega(\alpha m)) = M(\lambda\alpha.0) = m$$

in the above proof relies on the ξ-rule (reduction under λ), which is not available in a system based on the combinators S and K rather than the λ-calculus. Usually HA^ω is taken in combinatory form, in which case one needs some form of extensionality to conclude that $f(0^\omega) = m$, and this explains how we avoid the extensionality hypothesis in Kreisel's original argument. But notice that the ξ-rule holds in categorical models.

Therefore the argument of the above proof shows that:

▶ **Theorem 2.** *In* HA^ω*, the* ξ*-rule, the axiom of choice, and the continuity of all functions* $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ *are together impossible.*

Another observation, offered independently by Thorsten Altenkirch, Thierry Coquand and Per Martin-Löf (personal communication), is that the continuity of a function $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ implies that it is extensional in the sense that it maps pointwise equal arguments to equal values, and so the continuity axiom has some amount of extensionality built into it.

The above formulation and proof of Theorem 1 assumes natural numbers, identity types, Π and Σ types, and no universes. But it uses only the identity type of the natural numbers. If we assume a universe U, this identity type doesn't need to be assumed, because it can be defined by induction. We first define a U-valued equality relation, where $\mathbb{0}$ is the empty type and $\mathbb{1}$ is the unit type with element $\star$,

$$(0 = 0) = \mathbb{1}, \quad (m+1 = 0) = (0 = n+1) = \mathbb{0}, \quad (m+1 = n+1) = (m = n).$$

Then we define $\mathrm{refl} : \Pi(n : \mathbb{N}).n = n$ by induction as

$$\mathrm{refl}(0) = \star, \qquad \mathrm{refl}(n+1) = \mathrm{refl}(n),$$

and $J : \Pi(A : \Pi(m,n).m = n \to U).(\Pi n.A\,n\,n\,(\mathrm{refl}(n))) \to \Pi m, n, p.A\,m\,n\,p$ by

$$\begin{array}{llllllll} J & A & r & 0 & 0 & \star & = & r\,0, \\ J & A & r & (m+1) & 0 & p & = & \mathbb{0}\text{-rec}\,(A\,(m+1)\,0)\,p, \\ J & A & r & 0 & (n+1) & p & = & \mathbb{0}\text{-rec}\,(A\,0\;\;(n+1))\,p, \\ J & A & r & (m+1) & (n+1) & p & = & J\,(\lambda mn.A(m+1)(n+1))\,(\lambda n.r(n+1))\,m\,n\,p. \end{array}$$

where $\mathbb{0}$-rec : $\Pi(X : U).\mathbb{0} \to X$ is the recursion combinator of the empty type. The usual computation rule, or judgemental equality, for J when it is given as primitive doesn't hold here, but the above J is enough to define transport (substitution) and hence symmetry, transitivity and application (congruence), which are enough to carry out the above proof formally (and we have checked this in Agda [6]). Hence the theorem and its proof can be expressed in a type theory without a primitive equality type. All is needed to formulate and prove Theorem 1 is a type theory with $\mathbb{0}, \mathbb{1}, \mathbb{N}, \Pi, \Sigma, U$.

3 Propositional truncation and existential quantification

We recall the notion of propositional truncation from the HoTT book and use it to define the quantifiers $\exists, \forall$, in a slightly different way from that in the HoTT book, so that they satisfy the Lawvere's adjointness conditions that correspond to their intuitionistic introduction and elimination rules. Another difference is that, instead of adding propositional truncations for all types to our type theory, we define what a propositional truncation for a given type is. For some types, their propositional truncation already exist, including the types needed in our discussion of uniform continuity in Section 4.

3.1 Propositional truncation

We adopt the terminology of the HoTT book, which clashes with the terminology of the Curry–Howard interpretation of (syntactical) propositions as types. For us, a *proposition* is a subsingleton, or a type whose elements are all equal, in the sense of the identity type, here written "=" again as in the HoTT book:

$$\mathrm{isProp}\, X = \Pi(x, y\colon X).x = y.$$

Perhaps a better terminology, compatible with that of topos theory, would be *truth value*, in order to avoid the clash. But we will stick to the terminology *proposition*, and occasionally use *truth value* synonymously, for emphasis.

A propositional truncation of a type X, if it exists, is a proposition $\|X\|$ together with a map $|-| : X \to \|X\|$ such that for any proposition P and $f : X \to P$ we can find $\bar{f} : \|X\| \to P$. Because P is a proposition, this map $\bar{f}$ is automatically unique up to pointwise equality, and we have $\bar{f}|x| = f(x)$, and hence a propositional truncation is a reflection in the categorical sense, giving a universal map of X into a proposition. This can also be understood as a recursion principle, or elimination rule,

$$\mathrm{isProp}\, P \to (X \to P) \to \|X\| \to P,$$

for any types P and X. The induction principle, in this case, can be derived from the recursion principle, but in practice it is seldom needed.

In HoTT, propositional truncations for all types are given as higher-inductive types, with the judgemental equality $\bar{f}|x| = f(x)$. From the existence of the truncation of the two-point type $\mathbb{2}$ with this judgemental equality, one can prove function extensionality (any two pointwise equal functions are equal) [11]. The assumption that $\|X\| \to X$ for every type X gives a constructive taboo (and also contradicts univalence) [10].

However, for some types X, not only can a propositional truncation $\|X\|$ be constructed in MLTT, but also there is a map $\|X\| \to X$:

1. If $P = \mathbb{0}$ or $P = \mathbb{1}$, or more generally if P is any proposition, we can take $\|P\| = P$, of course. In particular, if $X \to \mathbb{0}$, we can take $\|X\| = X$; but also we can take $\|X\| = \mathbb{0}$, even though we can't say $X = \mathbb{0}$ without univalence.

2. If we have an inhabitant of X then we can take $\|X\| = \mathbb{1}$. The map $\|X\| \to X$ simply picks the given inhabitant.
3. More generally, if X is logically equivalent to a proposition P, then we can take $\|X\| = P$, and we make profitable use of this simple fact.
4. If X is any type and $g : X \to X$ is a constant map in the sense that any two of its values are equal, we can take $\|X\|$ to be the type $\Sigma(x : X).g(x) = x$ of fixed points of g, together with the function $X \to \|X\|$ that maps x to $(g(x), p)$, where p is an inhabitant of the type $g(g(x)) = g(x)$ coming from the constancy witness [10]. In this case the first projection gives a map $\|X\| \to X$. Given a map $f : X \to P$ into a proposition, we let $\bar{f} : \|X\| \to P$ be the first projection followed by $f : X \to P$ (and we don't use the fact that P is a proposition).
5. For any $f : \mathbb{N} \to \mathbb{N}$, the type $\Sigma(n : \mathbb{N}).f(n) = 0$, which may well be empty, has a constant endomap that sends (n, p) to (n', p'), where we take the least $n' \leq n$ with $p' : f(n') = 0$, using the decidability of equality of $\mathbb{N}$ and bounded search. Hence not only $\|\Sigma(n : \mathbb{N}).f(n) = 0\|$ exists, but also $\|\Sigma(n : \mathbb{N}).f(n) = 0\| \to \Sigma(n : \mathbb{N}).f(n) = 0$.

3.2 Quantification

For a universe U, let Prop be the type of propositions in U:

$$\mathrm{Prop} = \Sigma(X : U).\, \mathrm{isProp}\, X.$$

If we assume that all types in U come with designated propositional truncations, then we have a reflection

$$r : U \to \mathrm{Prop}$$

that sends $X : U$ to the pair $(\|X\|, p)$ with $p : \mathrm{isProp}\, \|X\|$ coming from the assumption. In the other direction, we have an embedding

$$s : \mathrm{Prop} \to U,$$

given by the projection. For $X : U$ we have

$$s(r(X)) = \|X\|.$$

(We also have that s is a section of r if propositional univalence holds.) For a fixed type $X : U$, the type constructors Σ and Π can be regarded as having type

$$\Sigma, \Pi : (X \to U) \to U.$$

We define

$$\exists, \forall : (X \to \mathrm{Prop}) \to \mathrm{Prop},$$

by, for any $A : X \to \mathrm{Prop}$,

$$\exists(A) = r(\Sigma(s \circ A)), \quad \forall(A) = r(\Pi(s \circ A)),$$

which we also write more verbosely as

$$\begin{aligned} (\exists(x : X).A(x)) &= r(\Sigma(x : X).s(A(x))), \\ (\forall(x : X).A(x)) &= r(\Pi(x : X).s(A(x))). \end{aligned}$$

This is essentially the same as the definition in the HoTT book, except that we give different types to $\exists, \forall$. With the type given in the book, $\forall$ gets confused with Π, because, with function extensionality, a product of propositions is a proposition, and so there is no need to distinguish $\forall$ from Π in the book.

The point of choosing the above types is that now it is easy to justify that these quantifiers do satisfy the intuitionistic rules for quantification. It is enough to show that they satisfy Lawvere's adjointness conditions. For $P, Q : \mathrm{Prop}$, define

$$(P \leq Q) = (s(P) \to s(Q)).$$

This is a pre-order (and a partial order if propositional univalence holds). Now endow the function type $(X \to \mathrm{Prop})$ with the pointwise order (using Π to define it). Then the quantifiers $\exists, \forall : (X \to \mathrm{Prop}) \to \mathrm{Prop}$ are the left and right adjoints to the exponential transpose $\mathrm{Prop} \to (X \to \mathrm{Prop})$ of the projection

$$\mathrm{Prop} \times X \to \mathrm{Prop},$$

using the universal property of truncation. The exponential transpose maps P to $\lambda x.P$. Hence the adjointness condition for the existential quantifier amounts to

$$\exists(A) \leq P \iff A \leq \lambda x.P.$$

Expanding the definitions, this amounts to

$$\begin{aligned} \|\Sigma(x : X).s(A(x))\| \to s(P) &\iff \Pi(x : X).s(A(x)) \to s(P), \\ &\iff (\Sigma(x : X).s(A(x))) \to s(P). \end{aligned}$$

So we need to check that

$$\|\Sigma(x : X).s(A(x))\| \to s(P) \iff (\Sigma(x : X).s(A(x))) \to s(P)$$

holds, but this is the case by the defining property of propositional truncation. For the sake of completeness, we also check the adjointness condition

$$P \leq \forall(A) \iff \lambda x.P \leq A.$$

For this we need function extensionality (which follows from the assumption of truncations supporting the judgemental equality discussed above). By definition, this amounts to

$$s(P) \to \|\Pi(x : X).s(A(x))\| \iff \Pi(x : X).s(P) \to s(A(x)).$$

But

$$\Pi(x : X).s(P) \to s(A(x)) \iff s(P) \to \Pi(x : X).s(A(x)),$$

and so the above is equivalent to

$$s(P) \to \|\Pi(x : X).s(A(x))\| \iff s(P) \to \Pi(x : X).s(A(x)).$$

But this again holds by the defining property of truncation, because, by function extensionality, a product of propositions is a proposition, and each $s(A(x))$ is a proposition. This explains why $\forall$ is identified with Π in the HoTT book.

Having established that the quantifiers $\exists, \forall : (X \to \mathrm{Prop}) \to \mathrm{Prop}$ defined from Σ and Π with truncation (via the reflection $r : U \to \mathrm{Prop}$) do satisfy the adjointness conditions corresponding to the introduction and elimination rules of intuitionistic logic, in practice we prefer to use the notation of the HoTT book, with $\exists(x : X).A(x)$ defined as $\|\Sigma(x : X).A(x)\|$ for propositionally-valued $A : X \to U$, or even avoid $\exists$ altogether, and just use truncation explicitly, as in the next section.

4 Uniform continuity of functions $2^{\mathbb{N}} \to \mathbb{N}$

We now compare the untruncated formulation of the uniform continuity principle

$$\Pi(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \Sigma(n\colon \mathbb{N}).\ \Pi(\alpha,\beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

with its truncated version

$$\Pi(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \|\Sigma(n\colon \mathbb{N}).\ \Pi(\alpha,\beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta\|.$$

A formal counter-part in Agda of this section is available at [6].

We work in a type theory with $\mathbb{0}, \mathbb{1}, \mathbb{2}, \mathbb{N}, \Pi, \Sigma, \mathrm{Id}$. This time, identity types for types other than $\mathbb{N}$ are needed, but universes are not. But we need more:

1. In principle, we would have to assume the presence of truncations, for example as defined in the HoTT book and explained in the previous section, from which function extensionality follows [11].
2. However, in turns out that function extensionality alone suffices, because it implies the existence of the propositional truncation mentioned above, and hence we can omit propositional truncations from our type theory. (But it doesn't seem to be possible to remove the assumption of function extensionality in the theorem proved here.)

Hence we don't assume propositional truncations in our type theory.

▶ **Theorem 3.** *Assuming function extensionality, for every $f : 2^{\mathbb{N}} \to \mathbb{N}$ the type*

$$\Sigma(n\colon \mathbb{N}).\ \Pi(\alpha,\beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

has a propositional truncation, and the proposition

$$\Pi(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \|\Sigma(n\colon \mathbb{N}).\ \Pi(\alpha,\beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta\|$$

is logically equivalent to the type

$$\Pi(f\colon 2^{\mathbb{N}} \to \mathbb{N}).\ \Sigma(n\colon \mathbb{N}).\ \Pi(\alpha,\beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta.$$

▶ **Lemma 4.** *Function extensionality implies that, for any $f : 2^{\mathbb{N}} \to \mathbb{N}$, the type family*

$$A(n) = \Pi(\alpha,\beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

satisfies the following conditions:

1. *$A(n)$ is a proposition for every $n\colon \mathbb{N}$, and*
2. *$A(n)$ implies that $A(m)$ is decidable for every $m < n$,*

Proof. By Hedberg's Theorem [7], equality of natural numbers is a proposition. Hence so is $A(n)$, because, by function extensionality, a product of a family of propositions is a proposition. To conclude that for all n, if $A(n)$ holds then $A(m)$ is decidable for all $m < n$, it is enough to show that for all n, (1) $\neg A(n+1)$ implies $\neg A(n)$, and (2) if $A(n+1)$ holds then $A(n)$ is decidable. (1) This follows from $A(n) \to A(n+1)$, which says that any number bigger than a modulus of uniform continuity is also a modulus, which is immediate. (2): For every n, the type

$$B(n) = \Pi(s\colon 2^n).\ f(s0^\omega) = f(s1^\omega),$$

is decidable, because $\mathbb{N}$ has decidable equality and finite products of decidable types are also decidable. Now let $n : \mathbb{N}$ and assume $A(n+1)$. To show that $A(n)$ is decidable, it is

enough to show that $A(n)$ is logically equivalent to $B(n)$, because then $B(n) \to A(n)$ and $\neg B(n) \to \neg A(n)$ and hence we can decide $A(n)$ by reduction to deciding $B(n)$.

The implication $A(n) \to B(n)$ holds without considering the assumption $A(n+1)$. To see this, assume $A(n)$ and let $s : 2^n$. Taking $\alpha = s0^\omega$ and $\beta = s1^\omega$, we conclude from $A(n)$ that $f(s0^\omega) = f(s1^\omega)$, which is the conclusion of $B(n)$.

Now assume $A(n+1)$ and $B(n)$. To establish $A(n)$, let $\alpha, \beta : 2^{\mathbb{N}}$ with $\alpha =_n \beta$. We need to conclude that $f(\alpha) = f(\beta)$. By the decidability of equality of 2, either $\alpha(n) = \beta(n)$ or not. If $\alpha(n) = \beta(n)$, then $\alpha =_{n+1} \beta$, and hence $f(\alpha) = f(\beta)$ by the assumption $A(n+1)$. If $\alpha(n) \neq \beta(n)$, we can assume w.l.o.g. that $\alpha(n) = 0$ and $\beta(n) = 1$. Now take the finite sequence $s = \alpha(0), \alpha(1), \dots, \alpha(n-1) (= \beta(0), \beta(1), \dots, \beta(n-1))$. Then $\alpha =_{n+1} s0^\omega$ and $s1^\omega =_{n+1} \beta$, which together with $A(n+1)$ imply $f(\alpha) = f(s0^\omega)$ and $f(s1^\omega) = f(\beta)$. But $f(s0^\omega) = f(s1^\omega)$ by $B(n)$, and hence $f(\alpha) = f(\beta)$ by transitivity. ◀

▶ **Lemma 5.** *If a type X is logically equivalent to a proposition Q, then*
1. *X has the propositional truncation $\|X\| = Q$, and*
2. *$\|X\| \to X$.*

Proof. We have $X \to \|X\|$ because this is the assumption $X \to Q$. If $X \to P$ for some proposition P, then also $\|X\| \to P$, because this means $Q \to P$, which follows from the assumption $Q \to X$ and transitivity of implication. This shows that our definition of $\|X\|$ has the required property for truncations. And $\|X\| \to X$ is the assumption that $Q \to X$. ◀

▶ **Lemma 6.** *Function extensionality implies that for any family A of types indexed by natural numbers such that*
1. *$A(n)$ is a proposition for every $n\colon \mathbb{N}$, and*
2. *$A(n)$ implies that $A(m)$ is decidable for every $m < n$,*

the type $\Sigma(n : \mathbb{N}).A(n)$ is logically equivalent to the proposition

$$P = \Sigma(k : \mathbb{N}).B(k)$$

where

$$B(k) = A(k) \times \Pi(i : \mathbb{N}).A(i) \to k \leq i.$$

Proof. By function extensionality, the product of a family of propositions is a proposition, and hence the type $\Pi(n : \mathbb{N}).A(n) \to k \leq n$ is a proposition, because the type $k \leq n$ is a proposition. Because the product of two propositions is a proposition, the type $B(k)$ is a proposition. But now if $B(k)$ and $B(k')$ then, by construction, $k = k'$. Hence any two inhabitants of P are equal, using the fact that $B(k)$ is a proposition, which means that P is indeed a proposition. By projection, $P \to \Sigma(n : \mathbb{N}).A(n)$. Conversely, if we have $(n, a) : \Sigma(n : \mathbb{N}).A(n)$, then we can find, by the decidability of $A(m)$ for $m < n$, the minimal k such that there is $b : A(k)$, by search bounded by n, and this gives an element $(k, b, \mu) : P$ where $\mu : \Pi(i : \mathbb{N}).A(i) \to k \leq i$ is the minimality witness. This shows that $\Sigma(n : \mathbb{N}).A(n) \to P$ and concludes the proof. ◀

▶ Remark. Function extensionality in the above lemma can be avoided using the fact that the type of fixed points of a constant endomap is a proposition [10], where a map is constant if any two of its values are equal. Given $(n, a)\colon \Sigma(n\colon\mathbb{N}).A(n)$, we know that $A(m)$ is decidable for all $m < n$ and thus can find the minimal m such that $A(m)$, by search bounded by n, which gives an endomap of $\Sigma(n\colon\mathbb{N}).A(n)$. This map is constant, because any two minimal witnesses are equal, and because $A(n)$ is a proposition. Then we instead take P to be the type of fixed points of this constant map.

By Lemmas 4, 5, and 6, for any $f : 2^{\mathbb{N}} \to \mathbb{N}$, the truncation of the type

$$\mathrm{UC}(f) = \Sigma(n\colon \mathbb{N}).\ \Pi(\alpha, \beta\colon 2^{\mathbb{N}}).\ \alpha =_n \beta \to f\alpha = f\beta$$

exists and implies $\mathrm{UC}(f)$, which establishes Theorem 3. ◀

Unfolding the above construction of the truncation, the truncated version of uniform continuity says that there is, using Σ to express existence, a minimal modulus of uniform continuity, making this use of Σ into a proposition, and, by function extensionality, the statement of uniform continuity into a proposition too. Then the theorem says that this proposition is logically equivalent to the existence, using Σ again, of some modulus of uniform continuity. This statement is not a proposition, because any number bigger than a modulus of uniform continuity is itself a modulus of uniform continuity.

The situation here is analogous to that of quasi-inverses and equivalences in the sense of the HoTT book. The type expressing that a function has a quasi-inverse is not a proposition in general, but it is equivalent to the type expressing that the function is an equivalence, which is always a proposition. Hence being an equivalence is the propositional truncation of having a quasi-inverse.

Acknowledgements. The material of this paper was presented at the Institute Henri Poicaré in Paris in June 2014 during the research programme *Semantics of proofs and certified mathematics* organized by Pierre-Louis Curien, Hugo Herbelin, Paul-André Mellies, at the workshop *Semantics of proofs and programs* organized by Thomas Ehrhard and Alex Simpson. We are grateful to them for the invitation to take part of such a wonderful programme and workshop, and to the participants, for useful discussions and input. In particular, we mention Thorsten Altenkirch, Andrej Bauer, Thierry Coquand, Nicolai Kraus, Per Martin-Löf, Paulo Oliva, Bas Spitters and Thomas Streicher.

We also thank the anonymous referees for various suggestions for improvement, including a simplification of the proof of Theorem 1 by removing a case distinction (the updated Agda files [6] contain both our original proof and the simplified proof presented here).

References

1 Steve Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

2 Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

3 Ana Bove and Peter Dybjer. Dependent types at work. Lecture Notes for the LerNet Summer School, Piriápolis, Uruguay, 2008.

4 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda—a functional language with dependent types. In *Theorem proving in higher order logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

5 Martín Hötzel Escardó and Chuangjie Xu. A constructive manifestation of the Kleene–Kreisel continuous functionals. To appear in APAL, 2013.

6 Martín Hötzel Escardó and Chuangjie Xu. The inconsistency of a Brouwerian continuity principle with the Curry–Howard interpretation (Agda development). School of Computer Science, University of Birmingham, UK. Available at `http://www.cs.bham.ac.uk/~mhe/continuity-false/`, 2015.

7 Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *J. Functional Programming*, pages 413–436, 1998.

8 J Martin E Hyland. The effective topos. *Studies in Logic and the Foundations of Mathematics*, 110:165–216, 1982.

9 Peter T. Johnstone. On a topological topos. *Proceedings of the London Mathematical Society*, 38(3):237–271, 1979.

10 Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Generalizations of hedberg's theorem. In *Typed Lambda Calculi and Applications*, volume 7941 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2013.

11 Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf Type Theory. Submitted for publication, 2014.

12 G. Kreisel. On weak completeness of intuitionistic predicate logic. *J. Symbolic Logic*, 27:139–158, 1962.

13 Maria Emilia Maietti. The internal type theory of a Heyting pretopos. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs*, volume 1512 of *Lecture Notes in Computer Science*, pages 216–235. Springer Berlin Heidelberg, 1998.

14 N. P. Mendler. Quotient types via coequalizers in Martin-Löf type theory. In G. Huet and G. Plotkin, editors, Informal Proceedings of the First Workshop on Logical Frameworks, Antibes, May 1990, pages 349–360, 1990, May 1990.

15 M. Menni and A. Simpson. Topological and limit-space subcategories of countably-based equilogical spaces. *Math. Struct. Comput. Sci.*, 12(6):739–770, 2002.

16 U. Norell. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 1–2, 2009.

17 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

18 A. S. Troelstra. A note on non-extensional operations in connection with continuity and recursiveness. *Indag. Math.*, 39(5):455–462, 1977.

19 Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics : An Introduction*. North-Holland, 1988.

20 Chuangjie Xu and Martín Hötzel Escardó. A constructive model of uniform continuity. In *Typed Lambda Calculi and Applications*, volume 7941 of *Lecture Notes in Computer Science*, pages 236–249. Springer Berlin Heidelberg, 2013.

Curry-Howard for Sequent Calculus at Last!

José Espírito Santo

Centro de Matemática, Universidade do Minho, Portugal

Abstract

This paper tries to remove what seems to be the remaining stumbling blocks in the way to a full understanding of the Curry-Howard isomorphism for sequent calculus, namely the questions: What do variables in proof terms stand for? What is co-control and a co-continuation? How to define the dual of Parigot's mu-operator so that it is a co-control operator? Answering these questions leads to the interpretation that sequent calculus is a formal vector notation with first-class co-control. But this is just the "internal" interpretation, which has to be developed simultaneously with, and is justified by, an equivalent, "external" interpretation, offered by natural deduction: the sequent calculus corresponds to a bi-directional, agnostic (w.r.t. the call strategy), computational lambda-calculus. Next, the formal duality between control and co-control is studied, in the context of classical logic. The duality cannot be observed in the sequent calculus, but rather in a system unifying sequent calculus and natural deduction.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases co-control, co-continuation, vector notation, let-expression, formal substitution, context substitution, computational lambda-calculus, classical logic, de Morgan duality

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.165

1 Introduction

Despite the anathema *"From an algorithmic viewpoint, the sequent calculus has no* Curry-Howard isomorphism*, because of the multitude of ways of writing the same proof"* [9], more than two decades of research have been dedicated to extend the Curry-Howard isomorphism to the sequent calculus. In its purest form, this is the question: if systems of combinators correspond to Hilbert systems, and the ordinary λ-calculus corresponds to natural deduction, what variant of the λ-calculus does correspond to the sequent calculus? Many computational aspects have been shown to be relevant: pattern matching [2, 19], explicit substitutions[1, 18], abstract machines [4]. But we miss a clear-cut answer to the question in its pure form. The textbook [18] says that the sequent calculus corresponds to explicit substitutions, but it immediately admits "this is just the beginning of the story", as we will see yet again.

We might dismiss the question as closed: maybe the sequent calculus is too complex to admit such a clear-cut computational explanation. But we will not give up, because of the following reason: there are very basic questions, at the bottom of any attempt to understand the sequent calculus computationally, which remain barely uttered and scandalously unanswered; these question we may now give an answer; and this answer opens the way to the desired clear-cut interpretation of the sequent calculus. The questions are:

(i) What does a variable stand for in a sequent calculus proof-term?
(ii) What is the related substitution operation?
(iii) What is co-control?

What do we mean? Something very simple. Suppose the proof terms L_2 and L_3 represent derivations of $\Gamma \vdash A$ and $\Gamma, y : B \vdash C$ respectively; and suppose we now infer $\Gamma, x : A \supset B \vdash C$

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 165–179

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

by left-introduction, building a derivation represented by some term $\mathsf{L}(x, L_2, y.L_3)$. What does this x stand for? What can it be substituted for? Certainly not for a proof-term, as $\mathsf{L}(L_1, L_2, y.L_3)$ corresponds to no derivation.

Of course we can survive by avoiding (not answering) the questions. We may say $\mathsf{L}(x, L_2, y.L_3)$ is a particular form $\langle xL_2/y\rangle L_3$ of "explicit substitution", the general form of which corresponds to cut, we can write cut-elimination rules with this syntax, many of them consisting of permutations of substitutions, all this even without breaking strong normalization [18]. Left introduction is reified with a cut that will to be eliminated, interpreted as a substitution that will not be executed. After all this, we still don't know what that x stands for; and we are busy doing explicitly "substitution", but we do not really know what substitution we mean.

Variables in proof-terms correspond to formulas in the l.h.s. of sequents; and the explicit handling of formulas in the l.h.s. of sequents is typical of the sequent calculus. Now, we have a model of what the handling of formulas in the *r.h.s.* of sequents means computationally: it is the $\lambda\mu$-calculus, which proves that the r.h.s. handling is related to *control operation*. So, by mere formal duality, "co-control" operation has to be involved in the computational interpretation of the sequent calculus. But what is co-control? This is question III.

Toward the system. Of course, our starting point for a co-control operator is the $\tilde{\mu}$-operator of the $\overline{\lambda}\mu\tilde{\mu}$-calculus. But in $\overline{\lambda}\mu\tilde{\mu}$: variables are (and stand for) proof-terms – which allows one to represent left-introductions $\mathsf{L}(x, L_2, y.L_3)$ as certain cuts that cannot be eliminated; and the operational meaning of $\tilde{\mu}$ is given by a reduction rule that triggers an ordinary term-substitution.

We propose to integrate the $\tilde{\mu}$-operator in a system keeping the cut=redex paradigm, where the treatment of variables is very much like in the $\overline{\lambda}$-calculus [10]: variables are not terms, but rather show in a construction that Herbelin writes xl and sees as corresponding to the structural inference of contraction. We prefer to interpret this construction logically as an inference that makes a formula passive on the l.h.s. of the sequent, and computationally as the dual to the "naming" construction aM of the $\lambda\mu$-calculus. Additionally, in our system the operational meaning of $\tilde{\mu}$ is given by a rule that captures some sort of context, triggering some sort of "structural substitution", as in $\lambda\mu$. Variables will stand for that sort of context (in the same way as names in $\lambda\mu$ stand for evaluation contexts or continuations), and the new "structural substitution" is the related kind of substitution. But what sort of context and structural substitution? The co-control question sends us back to the questions I and II.

The answers come from recent work about the isomorphism between $\overline{\lambda}\mu\tilde{\mu}$ and natural deduction [17], where a context-like concept named co-context was introduced in the sequent calculus side. Here we adapt the concept to our intuitionistic setting and rename it *co-continuation*. This is the last ingredient of the sequent calculus we propose, named $\overline{\lambda\tilde{\mu}}$. The system is an extension of $\overline{\lambda}$ with first-class co-control.

Computational interpretation. Having obtained the system, what is its computational interpretation? First-class co-control is a part of the interpretation, but not the whole story – it is only the story that goes beyond $\overline{\lambda}$. Fortunately, all that is needed for the rest of the story is already in place. Surprisingly, we can harvest not one but actually two clear-cut interpretations. Amazingly, the alternative can be seen through the little construction xl.

The first interpretation, the *external* one, is through natural deduction. At the basis of it is the idea, going back to [10], that l represents an evaluation context or continuation (a concept derived from the λ-calculus or natural deduction syntax), and that xl represents a

fill instruction: fill x in the hole of the continuation represented by l. This interpretation, if developed along the lines of [7, 17], is the core of the isomorphism Θ between the system $\overline{\lambda\tilde{\mu}}$ and the natural deduction system (named $\underline{\lambda\mathsf{let}}$) that is designed hand-in-hand with it.

Notice the isomorphism holds at the levels of syntax and rewriting. For this reason, Θ is the ultimate term assignment, that completely side-steps the anathema cited above, and the ultimate realization of the idea, going back to Prawitz [15], of sequent calculus as a meta-system for natural deduction. But, more important, Θ is a computational interpretation, because the target system $\underline{\lambda\mathsf{let}}$ – we will see – has a clear computational meaning itself: it is a computational λ-calculus [12, 16] agnostic w.r.t. the CBN vs CBV alternative.

The second interpretation, the *internal* one, is through the "structural substitution" operation of $\overline{\lambda\tilde{\mu}}$: xl is a fill instruction, but l is the stuff to be filled in the hole of the co-continuation that will land at x. Here l is taken literally, as primitive syntax, not as a continuation. So, we need a good word to describe l computationally. It could be "list" [10] or "spine" [3], but the best is "*vector*", to suggest the (informal) "vector notation" of the λ-calculus [11]. This choice is part of a most needed re-interpretation of a 15-years-old technical result: there is a fragment of $\overline{\lambda}$, here renamed $\overrightarrow{\lambda}$, that is isomorphic to the λ-calculus [6, 3, 5], the isomorphism being essentially the map $\mathcal{P}$ from natural deduction to sequent calculus introduced by Prawitz [15]. But someone has to say loudly that $\overrightarrow{\lambda}$ is a *formal vector notation*. This is the re-interpretation.

Summarizing, we propose two computational interpretations of $\overline{\lambda\tilde{\mu}}$: formal vector notation with first-class co-control, developed in Section 3; and agnostic computational λ-calculus, developed in Section 4. Section 5 looks closely at the duality between control and co-control and extracts unforeseen consequences for logical duality and structural proof theory. Please bear in mind that the words of this introduction just give an approximation of what we want to say. Let the technical development that follows speak for itself.

2 Background

Outside Section 5, we just consider intuitionistic implicational logic. Formulas(=types) are given by: $A, B, C ::= X \mid A \supset B$. In typing systems, contexts Γ are sets of declarations $x : A$ with at most one declaration per variable. In term languages, meta-substitution is denoted with square brackets, as in $[N/x]M$.

2.1 Control operation

Parigot's $\lambda\mu$-calculus [13] is our model for the management of formulas and (co-)variables in the r.h.s. of sequents, when the possibility of a distingushed/active/selected formula exists – a model we wish to "dualize" to the l.h.s. of sequents.

Still we diverge from the original. Let $Q := [b]((\mu a.M)N_1 \cdots N_m)$. In the original formulation of $\lambda\mu$, the reduction of Q proceeds by m applications of "structural reduction", by which μ-abstraction consumes the arguments, one after the other, capturing contexts $[\cdot]N_i$ and triggering a *"structural substitution"*. After m such reduction steps, the resulting term $[b]\mu a.M'$ is reduced by a *renaming* rule, producing $[b/a]M'$. The same effect is obtained with a single, long-step, reduction rule for the μ-operator. Consider the context $\mathcal{C} = [b]([\cdot]N_1 \cdots N_m)$, hence $Q = \mathcal{C}[\mu a.M]$. Now apply the reduction rule $\mathcal{C}[\mu a.M] \to [\mathcal{C}/a]M$, where $[\mathcal{C}/a]M$ is

(Terms)	t, u	$::=$	$x \mid \lambda x.t \mid \mu a.c$	(β)	$\langle \lambda x.t \mid u :: e \rangle$	$\to$	$\langle u \mid \tilde{\mu} x.\langle t \mid e \rangle \rangle$
(Co-terms)	e	$::=$	$a \mid u :: e \mid \tilde{\mu} x.c$	$(\tilde{\mu})$	$\langle t \mid \tilde{\mu} x.c \rangle$	$\to$	$[t/x]c$
(Commands)	c	$::=$	$\langle t \mid e \rangle$	(μ)	$\langle \mu a.c \mid e \rangle$	$\to$	$[e/a]c$

Figure 1 The $\overline{\lambda}\mu\tilde{\mu}$-calculus.

context substitution, in whose definition the critical case reads:[1]

$$[\mathcal{C}/a](aP) = \mathcal{C}[P'] \text{ where } P' = [\mathcal{C}/a]P \ . \tag{1}$$

In this formulation, the single reduction rule still says that the μ-operator captures $\mathcal{C}$, while the definition of context substitution says that aP is a *fill instruction*: fill the hole of the $\mathcal{C}$ that lands here with P.

This style with a single reduction rule is – see [17] – the exact reflection in natural deduction of the reduction rule for the μ-operator in the $\overline{\lambda}\mu\tilde{\mu}$-calculus. We are calling the latter reduction rule μ – see Fig. 1 – and so it is natural to call μ the corresponding natural deduction rule [2]. Beware that, sometimes, in $\lambda\mu$, μ names solely the "structural reduction" – which we may see as the control operation proper. In the style with a single rule, μ comprehends the whole control operation, and we will seek a single rule $\tilde{\mu}$ to comprehend the whole co-control operation – and find something different from the rule $\tilde{\mu}$ of $\overline{\lambda}\mu\tilde{\mu}$, despite the compelling symmetry of the latter.

2.2 Vector notation

The vector notation for the λ-calculus is the following definition of the λ-terms [11]:

$$M, N, P, Q \ ::= \ \lambda x.M \mid x\vec{N} \mid (\lambda x.M)N\vec{N}$$

According to this definition, λ-terms have three forms, which we call *first, second and third forms*. The advantage of this notation is that the head variable/redex is visible, and the β-normal forms are obtained by omitting the third form of terms in the above grammar.

This notation is informal, since many details are left unspecified. For instance: Are vectors a separate syntactic class, or do the second and third forms correspond to families of rules? How is substitution defined? Notice that, in the second form, it does not make sense to replace x by another term.

Let us introduce the *relaxed* vector notation:

$$M, N, P, Q \ ::= \ \lambda x.M \mid x\vec{N} \mid M\vec{N}$$

Some redundancy is now allowed, as the same ordinary λ-term can be represented in many ways. By analyzing the third form, we find four cases. One corresponds to the third form of the original vector notation, the other three may be simplified as follows:

$$\begin{array}{lrcl} (\epsilon) & M[] & = & M \\ (\pi_1) & (x\vec{Q})N\vec{N} & = & x(\vec{Q}N\vec{N}) \\ (\pi_2) & (P\vec{Q})N\vec{N} & = & P(\vec{Q}N\vec{N}) \end{array}$$

Here $[]$ represents the empty vector. The simplifications were given names that will link them with the reduction rules of sequent calculus.

---

[1] Structural substitution is the particular case $[a([\cdot]N)/a]\_$ of context substitution.
[2] In [17] both rules are called σ_μ.

$$Ax\ \frac{}{\Gamma|A\vdash[]:A} \qquad Cut\ \frac{\Gamma\vdash t:A \quad \Gamma|A\vdash k:B}{\Gamma\vdash tk:B} \qquad Pass\ \frac{\Gamma,x:A|A\vdash k:B}{\Gamma,x:A\vdash x\hat{}k:B}$$

$$L\supset\ \frac{\Gamma\vdash u:A \quad \Gamma|B\vdash k:C}{\Gamma|A\supset B\vdash u::k:C} \qquad R\supset\ \frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash\lambda x.t:A\supset B} \qquad Act\ \frac{\Gamma,x:B\vdash v:B}{\Gamma|A\vdash\tilde{\mu}x.v:B}$$

Figure 2 Typing rules of the $\overline{\lambda\tilde{\mu}}$-calculus.

The third form of the original vector notation is a β-redex and hence it means something like "call the head function with the first argument of the vector". The simplifications that have arisen by relaxing the vector notation can be read as rules for vector bookkeeping: garbage collect an empty vector (ϵ), or append chained vector (π_i). Much more difficult is to recognize in the second form $x\vec{N}$ an instruction for action on the vector $\vec{N}$: this is visible in a more general system with co-control, like the sequent calculus we are about to introduce, where the base case for vectors is not just $[]$.

3 Sequent calculus

3.1 The $\overline{\lambda\tilde{\mu}}$-calculus

The proof-expressions of $\overline{\lambda\tilde{\mu}}$ [3] are given by the following grammar:

$$\begin{array}{rrcl} \text{(Terms)} & t,u,v & ::= & \lambda x.t \mid x\hat{}k \mid tk \\ \text{(Generalized vectors)} & k & ::= & [] \mid \tilde{\mu}x.v \mid u::k \end{array}$$

The typing rules are in Fig. 2. They handle two kinds of sequents: $\Gamma\vdash t:A$ and $\Gamma|A\vdash k:B$. The distinguished formula A in the latter is not exactly a "stoup" or a focused formula, because the operator $\tilde{\mu}x.t$ may select an arbitrary formula from the context Γ. The construction $x\hat{}k$ comes from the $\overline{\lambda}$-calculus, but here it forms a pair with $\tilde{\mu}x.t$: logically, these are a activation/passification pair, in the style of the $\lambda\mu$-calculus, but acting on the l.h.s. of sequents.

The notation $\tilde{\mu}x.t$ comes from $\overline{\lambda}\mu\tilde{\mu}$ [4]; but here, contrary to what happens in $\overline{\lambda}\mu\tilde{\mu}$, the reduction rule that defines the behavior of $\tilde{\mu}$ does not trigger a term-substitution, but rather a context-substitution, in the style of the above presentation of the μ-operator. The construction $x\hat{}k$ is easily recognized as the accompanying fill-instruction, and what remains is to pin down the right notion of context that $\tilde{\mu}$ will capture. The notion is this:

$$\mathcal{H}\ ::=\ x\hat{}[\cdot] \mid t([\cdot]) \mid \mathcal{H}[u::[\cdot]]$$

These expressions are called *co-continuations*. Later we will argue they are logically dual to continuations.

The reduction rules of $\overline{\lambda\tilde{\mu}}$ are in Fig. 3. Let $\pi:=\pi_1\cup\pi_2$. The co-control rule $\tilde{\mu}$ triggers the context substitution $[\mathcal{H}/x]\_$, in whose definition the only non-routine case is:

$$[\mathcal{H}/x](x\hat{}k)=\mathcal{H}[k'] \text{ with } k'=[\mathcal{H}/x]k\ .$$

This equation gives the meaning of $x\hat{}k$: fill k in the hole of the $\mathcal{H}$ that will substitute x. The π_i-rules employ concatenation of generalized vectors $k@k'$, defined by the obvious equations $[]@k'=k'$ and $(u::k)@k'=u::(k@k')$, together with $(\tilde{\mu}x.t)@k'=\tilde{\mu}x.tk'$.

[3] We would have adopted the name $\overline{\lambda}\tilde{\mu}$ (to suggest $\overline{\lambda}+\tilde{\mu}$), had it not been already in use [4].

$$\begin{array}{llcl@{\qquad}llcl} (\beta) & (\lambda x.t)(u :: k) & \to & (u(\tilde{\mu}x.t))k & (\epsilon) & t[] & \to & t \\ (\tilde{\mu}) & \mathcal{H}[\tilde{\mu}x.t] & \to & [\mathcal{H}/x]t & (\pi_1) & (x\hat{}k)k' & \to & x\hat{}(k@k') \\ & & & & (\pi_2) & (tk)k' & \to & t(k@k') \end{array}$$

Figure 3 Reduction rules of $\overline{\lambda\tilde{\mu}}$.

Rule $\tilde{\mu}$ eliminates all occurrences of the $\tilde{\mu}$-operator. The remaining rules eliminate all occurrences of cuts tk. So the $\beta\tilde{\mu}\epsilon\pi$-normal forms correspond to a well-known representation of β-normal λ-terms. There is a critical pair generated by rules $\tilde{\mu}$ and π. This is the *call-by-name vs call-by-value* dilemma [4].

Two particular cases of the reduction rule $\tilde{\mu}$ are:

$$(\rho) \qquad y\hat{}(\tilde{\mu}x.t) \to [y\hat{}[\cdot]/x]t \qquad\qquad (\sigma) \quad u(\tilde{\mu}x.t) \to [u([\cdot])/x]t$$

We let $\tau := \tilde{\mu}\backslash(\rho \cup \sigma)$. The particular case ρ may be called the *renaming* rule (as sometime does the "dual" rule in the $\lambda\mu$-calculus). Indeed, the particular case $[y\hat{}[\cdot]/x]t$ of context substitution is almost indistinguishable from a substitution operation that renames variables, since the critical case of its definition reads[4]

$$[y\hat{}[\cdot]/x](x\hat{}k) = y\hat{}k' \text{ with } k' = [y\hat{}[\cdot]/x]k \ .$$

If we wanted a set of small step $\tilde{\mu}$-rules, in the style of the original $\lambda\mu$-calculus, we would have taken ρ and σ, together with $u :: (\tilde{\mu}x.t) \to \tilde{\mu}x.[x\hat{}(u :: [\cdot])/x]t$. The particular case $[x\hat{}(u :: [\cdot])/x]t$ of context substitution gives a form of "structural substitution" dual to that found in $\lambda\mu$.

3.2 The proof theory of vector notation

We consider notorious fragments of $\overline{\lambda\tilde{\mu}}$. First we do a kind of reconstruction of $\overline{\lambda}$. Next we identify two subsystems of our version of $\overline{\lambda}$ that say something about vector notation.

The $\rho\tau$-normal-forms of $\overline{\lambda\tilde{\mu}}$ are given by:

$$t, u, v \ ::= \ \lambda x.t \,|\, x\hat{}l \,|\, tk \qquad k \ ::= \ l \,|\, \tilde{\mu}x.t \qquad l \ ::= \ [] \,|\, u :: l$$

These are the $\overline{\lambda}$-expressions, if we recognize $t(\tilde{\mu}x.v)$ as the "mid-cut", and if we ignore the other forms of explicit substitution or concatenation that are primitive in the original formulation of $\overline{\lambda}$. In terms of logical derivations, $\rho\tau$-normalization ends in the focused fragment LJT [10] of the sequent calculus (the fragment corresponding to $\overline{\lambda}$). In this sense $\rho\tau$-reduction is a focalization process. See the focalization theorem below, saying that $\rho\tau$-normal forms exist and are unique.

The $\tilde{\mu}$-rule is now restricted to the σ-rule, but the critical pair with π remains. This is solved by a syntactical trick, that chooses the call-by-name option: erase the case $\tilde{\mu}x.t$ from k's (so that there is no more a distinction between k's and l's, there is a single class of *vectors* ranger over by l), but break the cut tk into the two cases tl ("head-cut") and $t(\tilde{\mu}x.v)$. A π-redex $(x\hat{}l)l'$ or $(tl)l'$ is no longer a σ-redex, and a mid-cut $(y\hat{}l)\tilde{\mu}x.v$ or $(ul)\tilde{\mu}x.v$ can never be reduced by π (as it could in $\overline{\lambda\tilde{\mu}}$). This concludes the reconstruction of $\overline{\lambda}$.

This version of $\overline{\lambda}$ is equipped with β, σ, ϵ and π. We now consider the R-normal forms, with $R = \sigma$ or $R = \sigma \cup \epsilon \cup \pi$.

[4] We say "almost" because, don't forget, variables are not expressions *per se*.

- In the first case, terms have the forms $\lambda x.t$, $x\hat{}l$ or tl, equipped with a rule β that corresponds to $\overline{\lambda}$'s β-rule followed by σ-normalization. The rules ϵ and π remain. Let us call this system $\overset{\hookrightarrow}{\lambda}$.
- In the second case, terms have the forms $\lambda x.t$, $x\hat{}l$ or $(\lambda x.t)(u :: l)$, equipped solely with a rule β that corresponds to $\overline{\lambda}$'s β-rule followed by $\sigma\epsilon\pi$-normalization. Let us call this system $\overrightarrow{\lambda}$.

Around 15 year ago [6, 3, 5], the system $\overrightarrow{\lambda}$ was identified and proved isomorphic to the ordinary λ-calculus, with the isomorphism comprising a bijection between the sets of terms and an isomorphism of β-reduction relations. In the author's opinion, this little technical fact has a tremendous importance for the Curry-Howard isomorphism that has never been recognized. First, $\overrightarrow{\lambda}$ has a clear and revealing computational interpretation: it is a *formal vector notation*. It is a concrete definition of the notation (it says what vectors are, how substitution is defined, etc.) in perfect correspondence with a logical calculus. Second, we can reconstruct from this interpretation the interpretation of full sequent calculus $\overline{\lambda\tilde{\mu}}$ by walking back the path that led from $\overline{\lambda\tilde{\mu}}$ to $\overrightarrow{\lambda}$. Clearly, $\overset{\hookrightarrow}{\lambda}$ is a formalization of the relaxed vector notation we introduced in Section 2; and, since $\overset{\hookrightarrow}{\lambda}$ is the fragment of normal forms of $\overline{\lambda\tilde{\mu}}$ w.r.t. the co-control rule $\tilde{\mu}$, $\overline{\lambda\tilde{\mu}}$ can be interpreted as a *formal, relaxed vector notation with first-class co-control*.

4 Natural deduction

The interpretation developed before is an internal and literal one: vectors are vectors; and co-control is understood in a formal way, as dual to control. We now develop a natural deduction system $\underline{\lambda\mathsf{let}}$ that is isomorphic to $\overline{\lambda\tilde{\mu}}$. The isomorphism gives a new, external interpretation, which recovers the view that (some) vectors are "evaluation contexts"[10]; it justifies the design of $\overline{\lambda\tilde{\mu}}$, namely its notion of $\mathcal{H}$ and reduction rule $\tilde{\mu}$; finally, it allows the transfer of properties among the two systems.

4.1 The $\underline{\lambda\mathsf{let}}$-calculus

The proof-expressions of the calculus are given by:

$$\begin{array}{lrcl} \text{(Terms)} & M, N, P & ::= & \lambda x.M \mid \mathsf{app}(H) \mid \mathsf{let}\, x := H \,\mathsf{in}\, P \\ \text{(Heads)} & H & ::= & x \mid \mathsf{hd}(M) \mid HN \end{array}$$

Notice that variables x and applications HN are not terms. A head $\mathsf{hd}(M)$ is called a *head term*.

The typing rules are in Fig. 4. They handle two kinds of sequents: $\Gamma \vdash M : A$ and $\Gamma \rhd H : A$. Four rules are standard, with the appropriate kind of sequent determined by the kind of expression being typed. The remaining two rules switch the kind of sequent, and are called *coercions*. However, despite the superficial impression, the two coercions are quite different, one being called weak and the other strong. Normalization will tell them apart radically.

A *normal* derivation is one without occurrences of *Let* and *SCoercion*.

▶ **Theorem 1** (Subformula property). *Every formula (resp. every formula, including A) occurring in a normal derivation of $\Gamma \vdash M : A$ (resp. $\Gamma \rhd H : A$) is subformula of A or of some formula in Γ (resp. is a subformula of some formula in Γ).*

$$\frac{}{\Gamma, x : A \rhd x : A}\ Hyp \qquad \frac{\Gamma \rhd H : A \quad \Gamma, x : A \vdash P : B}{\Gamma \vdash \mathsf{let}\, x := H \,\mathsf{in}\, P : B}\ Let \qquad \frac{\Gamma \rhd H : A}{\Gamma \vdash \mathsf{app}(H) : A}\ WCoercion$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \supset B}\ Intro \qquad \frac{\Gamma \rhd H : A \supset B \quad \Gamma \vdash N : A}{\Gamma \rhd HN : B}\ Elim \qquad \frac{\Gamma \vdash M : A}{\Gamma \rhd \mathsf{hd}(M) : A}\ SCoercion$$

Figure 4 Typing rules of $\underline{\lambda\mathsf{let}}$.

$$\begin{array}{rrcl}
(\mathsf{beta}) & \mathsf{hd}(\lambda x.M)N & \to & \mathsf{hd}(\mathsf{let}\, x := \mathsf{hd}(N) \,\mathsf{in}\, M) \\
(\mathsf{let}) & \mathsf{let}\, x := H \,\mathsf{in}\, P & \to & [H/x]P \\
(\mathsf{triv}) & \mathsf{app}(\mathsf{hd}(M)) & \to & M \\
(\mathsf{head}_1) & \mathsf{hd}(\mathsf{app}(H)) & \to & H \\
(\mathsf{head}_2) & \mathcal{K}[\mathsf{hd}(\mathsf{let}\, x := H \,\mathsf{in}\, P)] & \to & \mathsf{let}\, x := H \,\mathsf{in}\, \mathcal{K}[\mathsf{hd}(P)]
\end{array}$$

Figure 5 Reduction rules of $\underline{\lambda\mathsf{let}}$.

So, the weak coercion loses information regarding the subformula property, while the strong coercion potentially violates that property.

The reduction rules of $\underline{\lambda\mathsf{let}}$ are in Fig. 5. Rule let triggers ordinary substitution $[H/x]P$, while rule head_2 employs certain contexts that we call *continuations*:

$$\mathcal{K} ::= \mathsf{app}([\cdot]) \mid \mathsf{let}\, x := [\cdot] \,\mathsf{in}\, P \mid \mathcal{K}[[\cdot]N]$$

We single out two particular cases of let: ren, when $H = x$; and sub, when $H = \mathsf{hd}(M)$. We put $\mathsf{t} := \mathsf{let} \backslash (\mathsf{ren} \cup \mathsf{sub})$. Let $\mathsf{head} := \mathsf{head}_1 \cup \mathsf{head}_2$. Notice that rules beta and head_1 are relations on heads. The normal forms w.r.t. all reduction rules are given by:

$$M ::= \lambda x.M \mid \mathsf{app}(H) \qquad H ::= x \mid HN$$

That is, these normal forms are characterized by the absence of occurrences of lets and $\mathsf{hd}()$. Lets are eliminated by let whereas all the other rules concur to eliminate the coercion $\mathsf{hd}()$. So, $\mathsf{beta}, \mathsf{triv}, \mathsf{let}, \mathsf{head}$-reduction is *normalization*, that is, the reduction to a form corresponding to normal derivations.

4.2 Isomorphism

See Fig. 6 for the map $\Theta : \overline{\lambda\tilde{\mu}} \longrightarrow \underline{\lambda\mathsf{let}}$. There is actually a function $\Theta : \overline{\lambda\tilde{\mu}}{-}Terms \longrightarrow \underline{\lambda\mathsf{let}}{-}Terms$, together with an auxiliary function $\Theta : \underline{\lambda\mathsf{let}}{-}Heads \times \overline{\lambda\tilde{\mu}}{-}Vectors \longrightarrow \underline{\lambda\mathsf{let}}{-}Terms$. Let $\Theta(t) = M$, $\Theta(u_i) = N_i$ and $\Theta(v) = P$. The idea is to map, say, $t(u_1 :: u_2 :: \tilde{\mu}x.v)$ to $\mathsf{let}\, x := hd(M)N_1N_2 \,\mathsf{in}\, P$, and $x\hat{}(u_1 :: u_2 :: [])$ to $\mathsf{app}(xN_1N_2)$: left-introductions are replaced by applications, inverting the associativity of non-abstractions.

▶ **Theorem 2** (Isomorphism). *Map Θ is a sound bijection between the set of $\overline{\lambda\tilde{\mu}}$-terms and the set of $\underline{\lambda\mathsf{let}}$-terms (whose inverse Ψ is shown in Fig. 7). Moreover, let R be rule β (resp. $\tilde{\mu}$, ϵ, π) of $\overline{\lambda\tilde{\mu}}$, and let R' be rule* beta *(resp.* let, triv, head*) of $\underline{\lambda\mathsf{let}}$. Then, $t \to_R t'$ in $\overline{\lambda\tilde{\mu}}$ iff $\Theta t \to_R \Theta t'$ in $\underline{\lambda\mathsf{let}}$.*

The real action of the isomorphism happens in the translation of non-abstractions. Every non-abstraction $\underline{\lambda\mathsf{let}}$-term has the form $\Theta(H, k)$, and $\Psi\Theta(H, k) = \Psi(H, k)$. Every non-abstraction $\overline{\lambda\tilde{\mu}}$-term has the form $\Psi(H, k)$, and $\Theta\Psi(H, k) = \Theta(H, k)$. So non-abstractions

$$\begin{array}{rclcrcl}
\Theta(\lambda x.t) & = & \lambda x.\Theta t & & \Theta(H,[]) & = & \mathsf{app}(H) \\
\Theta(x\hat{}k) & = & \Theta(x,k) & & \Theta(H,\tilde{\mu}x.t) & = & \mathsf{let}\, x := H \,\mathsf{in}\, \Theta t \\
\Theta(tk) & = & \Theta(hd(\Theta t),k) & & \Theta(H,u::k) & = & \Theta(H\Theta u,k)
\end{array}$$

Figure 6 Map $\Theta : \overline{\lambda\tilde{\mu}} \longrightarrow \underline{\lambda\mathsf{let}}$.

$$\begin{array}{rclcrcl}
\Psi(\lambda x.M) & = & \lambda x.\Psi M & & \Psi(x,k) & = & x\hat{}k \\
\Psi(\mathsf{app}(H)) & = & \Psi(H,[]) & & \Psi(\mathsf{hd}(M),k) & = & (\Psi M)k \\
\Psi(\mathsf{let}\, x := H \,\mathsf{in}\, P) & = & \Psi(H,\tilde{\mu}x.\Psi P) & & \Psi(HN,k) & = & \Psi(H,(\Psi N)::k)
\end{array}$$

Figure 7 Map $\Psi : \underline{\lambda\mathsf{let}} \longrightarrow \overline{\lambda\tilde{\mu}}$

have the form $\Theta(H,k)$ (natural deduction) or $\Psi(H,k)$ (sequent calculus), and the isomorphism action between them is just to interchange Θ and Ψ in these expressions.

Ψ can be extended to continuations, establishing a bijection with vectors: $\Psi(\mathsf{app}([\cdot])) = []$, $\Psi(\mathsf{let}\, x := [\cdot] \,\mathsf{in}\, P) = \tilde{\mu}x.\Psi P$ and $\Psi(\mathcal{K}[[\cdot]N]) = \Psi(N) :: \Psi(\mathcal{K})$. As we knew, continuations are typed "on the left": the sequent calculus rules for typing vectors are derived typing rules in natural deduction for typing continuations.

Similarly, Θ can be extended to co-continuations, establishing a bijection with heads: $\Theta(x\hat{}[\cdot]) = x$, $\Theta(t([\cdot]) = \mathsf{hd}(\Theta t)$ and $\Theta(\mathcal{H}[u :: [\cdot]]) = \Theta(\mathcal{H})\Theta u$. This tells us how to type co-continuations [17]: the natural deduction rules for typing heads are derived typing rules in sequent calculus for typing co-continuations, and so co-continuations are typed "on the right".

Let us call Θ the inverse of the bijection between continuations and vectors. Then it is easy to prove that $\Theta(H,k) = \Theta(k)[H]$. This tells us that non-abstractions in $\overline{\lambda\tilde{\mu}}$ are fill instructions, and that Θ executes these instructions. For instance, $\Theta(tk) = \Theta(\mathsf{hd}(\Theta t),k) = \Theta(k)[\mathsf{hd}(\Theta t)]$; so tk means "fill $\mathsf{hd}(M)$ in the hole of continuation $\mathcal{K}$", with $M = \Theta t$ and $\mathcal{K} = \Theta(k)$; and $\Theta(tk)$ is the result of such filling. Similarly, $x\hat{}k$ means "fill x in the hole of $\mathcal{K}$". So Θ realizes again the idea, going back to Prawitz [15], that sequent calculus derivations are instructions for building natural deduction proofs.

4.3 Forgetfulness

The *forgetful map* $|\cdot|$ translates $\underline{\lambda\mathsf{let}}$-expressions to λ-terms by erasing occurrences of the coercion $\mathsf{hd}(\cdot)$, forgetting the distinction between terms and heads, de-sugaring let-expressions (*i.e.* translating them as β-redexes), and mapping $|\mathsf{app}(H)| = I|H|$, where $I = \lambda x.x$. The following results about $\overline{\lambda\tilde{\mu}}$ are proved through the analysis of this simple translation of the isomorphic calculus $\underline{\lambda\mathsf{let}}$.

▶ **Theorem 3** (Strong normalization). *Every typable term of $\overline{\lambda\tilde{\mu}}$ (resp. of $\underline{\lambda\mathsf{let}}$) is $\beta\epsilon\tilde{\mu}\pi$-SN (resp.* beta triv let head*-SN).*

▶ **Theorem 4** (Focalization). *Every term of $\overline{\lambda\tilde{\mu}}$ has a unique $\rho\tau$-normal form, which is a $\overline{\lambda}$-term (representing a LJT-proof).*

4.4 Computational interpretation

We argue that $\underline{\lambda\mathsf{let}}$ is a *bidirectional, agnostic, computational λ-calculus*. The word "bi-directional" comes from [14], where the organization of a typing system for λ-terms with two kinds of sequents (for synthesis, like $\Gamma \vdash M : A$, and for checking, like $\Gamma \rhd H : A$) is already

found. The word "agnostic" comes from [19] and means coexistence or superimposition of call-by-name (CBN) and call-by-value (CBV).

Let us go back to Fig. 5. Rule β generates a let-expression, which can be executed by the separate rule let. Let-expressions enjoy "associativity": the particular case of head_2 where $\mathcal{K} = \mathsf{let}\, y := [\cdot]\, \mathsf{in}\, Q$ reads $\mathsf{let}\, y := \mathsf{hd}(\mathsf{let}\, x := H\, \mathsf{in}\, P)\, \mathsf{in}\, Q \to \mathsf{let}\, x := H\, \mathsf{in}\, \mathsf{let}\, y := \mathsf{hd}(P)\, \mathsf{in}\, Q$. In addition, there is a pair of reduction rules to cancel a sequence of two coercions. So we might view $\underline{\lambda\mathsf{let}}$ as a sort of computational λ-calculus [12, 16] where rule let does not require the computation of a value prior to substitution triggering, and where a pair of trivial reduction rules (triv and head_1) eliminates odd accumulations of coercions caused by a clumsy syntax.

However, this is not the right view. head_1 works together with head_2 to reduce every non-abstraction term to one of the forms $\mathcal{K}[x]$ or $\mathcal{K}[\mathsf{hd}(\lambda x.M)]$. This is a hint of what we see next: all reduction rules of $\underline{\lambda\mathsf{let}}$ have quite clear roles in CBV and CBN computation, and through these roles we will understand how different rules triv and head_1 are.

Let us make a technical point. Rule head_1 is a relation on heads. As with all other reduction rules, head_1 generates by compatible closure a relation $\to_{\mathsf{head}_1}$ on heads and another on terms. The relation $\to_{\mathsf{head}_1}$ on terms would have been the same, had we taken the rule head_1 as the relation on terms $\mathcal{K}[\mathsf{hd}(\mathsf{app}(H))] \to \mathcal{K}[H]$. A similar remark applies to beta. In the discussion of CBN and CBV that follows, we take head_1 and beta in their alternative formulation, so that it makes sense to speak about head_1- or beta-reduction at root position of a term.

CBN and CBV are defined through priorities among reduction rules [4]:

- CBV strategy: reduction at root position of a closed, non-abstraction term with priority given to head.
- CBN strategy: reduction at root position of a closed, let-free, non-abstraction term with priority given to triv.

We will give an alternative characterization of these strategies. For CBN we need the rule $\mathcal{K}[\mathsf{hd}(\lambda x.M)N] \to \mathcal{K}[\mathsf{hd}([\mathsf{hd}(N)/x]M)]$, which we call $\mathit{CBN} - \mathsf{beta}$, and is obtained by beta followed by let.

▶ **Theorem 5** (Agnosticism). *The following is an equivalent description of the CBN and CBV strategies. In this description, "reduction" means root-position reduction of a closed, non-abstraction term. We assume additionally that the initial term is let-free.*

- *CBV. Do* head*-reduction as long as possible, until the term becomes either a* beta*,* let*, or* triv *redex. In the two first cases, reduce and restart; in the last case, reduce to return the computed abstraction.*
- *CBN. Do* head*-reduction as long as possible, until the term becomes either a* beta *or* triv *redex. In the first case, reduce (with* $\mathit{CBN} - \mathsf{beta}$*) and restart; in the last case, reduce to return the computed abstraction.*

This description is not in terms of priorities, but rather reveals the shared organization of the computation and the roles of the different reduction rules, which are the same in both strategies – see Fig. 8. The shared organization, in turn, shows how "superimposed" CBV and CBN are in the system, in other words, how agnostic the system is.

4.5 Epilogue

An easy consequence of the isomorphism Θ is that the space of calculi in the sequent calculus format has a mirror image in natural deduction. See Fig. 9 for a roadmap. The λ-calculus is displayed in the "imperfect world" – the terminology is inspired in Remark 2.1. of [4].

		CBV	CBN
iteration	pre-processing	$\mathsf{head}_1 + \mathsf{head}_2$	head_1
of the computation cycle	real computation	beta + let	beta; let
return		triv	triv

Figure 8 Agnosticism: the shared organization of CBN and CBV strategies.

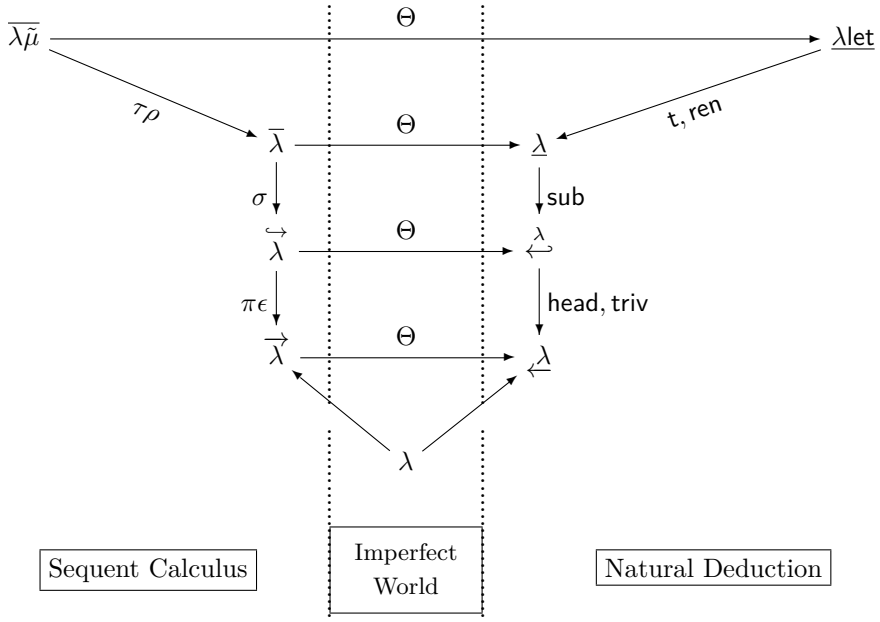


Figure 9 The sequent calculus/natural deduction mirror.

The two computational interpretations of sequent calculus are collected in Fig. 10. The external interpretation works like this: in Fig. 10 the shown interpretation is that of $\underline{\lambda\mathsf{let}}$; $\overline{\lambda\tilde{\mu}}$ is a language of instructions for $\underline{\lambda\mathsf{let}}$ (recall Section 4.2); the behavior of $t \in \overline{\lambda\tilde{\mu}}$ is the isomorphic behavior of $\Theta t \in \underline{\lambda\mathsf{let}}$ written in the language of instructions.

Why is co-control almost invisible in natural deduction? Why does it boil down to the low-profile rule let, which is just a substitution triggering rule? The explanation is in the good old associativity of "applicative terms" [10]. In natural deduction, the control operator shows up in the hidden part of an applicative terms, like $(\mu a.M)N_1 \cdots N_m$. Since we want to get to this M, we collect the outer stuff in the context-like structure $\mathcal{K} = [\cdot]N_1 \cdots N_m$ and trigger a context-substitution $[\mathcal{K}/a]M$. But, in natural deduction, the co-control operator is disguised in let-expressions $\mathsf{let}\, x := HN_1 \cdots N_m \,\mathsf{in}\, P = \Theta(HN_1 \cdots N_m, \tilde{\mu}x.P)$, and so is already at the surface. So we may proceed by ordinary substitution $[HN_1 \cdots N_m/x]P$.

5 Duality

Let us check that co-control, as formulated in $\overline{\lambda\tilde{\mu}}$, is dual to control. We will expand $\overline{\lambda\tilde{\mu}}$ to a self-dual system where control and co-control are each other's dual. This is achieved with three steps.

First step: to unify $\overline{\lambda\tilde{\mu}}$ and $\underline{\lambda\mathsf{let}}$. Some hints are at the end of Section 4.2, the idea comes from [7]. Every non-abstraction term of $\overline{\lambda\tilde{\mu}}$ has the form $\Psi(H, k)$. So we unify $x\hat{}k$ and tk as

sequent calculus	$\overline{\lambda\tilde{\mu}}$	internal interpretation	external interpretation
	t	λ-term in formal vector notation with first-class co-control	bi-directional, agnostic, computational λ-term
right intro.	$\lambda x.t$	λ-abstraction	λ-abstraction
contraction	$x\hat{}k$	2nd form of vector notation fill k in the co-continuation x	$\mathcal{K}[x]$
cut	tk	3rd form of vector notation	$\mathcal{K}[\mathsf{hd}(M)]$
	k	generalized vector	continuation $\mathcal{K}$
axiom	$[]$	empty vector	$\mathsf{app}([\cdot])$
left selection	$\tilde{\mu}x.t$	co-control operator	$\mathsf{let}\, x := [\cdot]\, \mathsf{in}\, P$
left intro.	$u :: k$	vector constructor	$\mathcal{K}[[\cdot]N]$
cut elim. + + focalization	red. rules		
key-step	β	function call	function call
left-sel. elim.	$\tilde{\mu}$	co-control operation	subst. triggering (let)
focalization	τ	co-control operation proper	subst. triggering (t)
focalization	ρ	subst. triggering (renaming)	subst. triggering (ren)
right-perm.	σ	subst. triggering	subst. triggering (sub)
right-perm.	ϵ	erasure of empty vector	return (triv)
left-perm.	π	append of iterated vectors	re-associate (head)

Figure 10 Curry-Howard for sequent calculus.

$\overline{\Psi}(H,k)$ and allow in $\overline{\lambda\tilde{\mu}}$ a new syntactic class $H ::= x \mid \mathsf{hd}(t)$. Every non-abstraction term of $\underline{\lambda\mathsf{let}}$ has the form $\Theta(H,k)$. Se we unify $\mathsf{app}(H)$ and $\mathsf{let}\, x := H\, \mathsf{in}\, P$ as $\underline{\Theta}(H,k)$ and allow in $\underline{\lambda\mathsf{let}}$ a new syntactic class $k ::= [] \mid \tilde{\mu}x.P$. Next let us unify $\overline{\Psi}(H,K)$ and $\underline{\Theta}(H,k)$ as $\chi(H|k)$. After this we realize that $\overline{\lambda\tilde{\mu}}$ and $\underline{\lambda\mathsf{let}}$ are partial views of the same system (the former lacks HN, the latter lacks $u :: k$). So let t and M range over the same set of proof terms. Continuations are internalized and coincide with vectors, co-continuations are internalized and coincide with heads, context-substitution is internalized as ordinary substitution. We work modulo $\chi(HN|k) = \chi(H|N :: k)$, which abstracts the single difference between $\overline{\lambda\tilde{\mu}}$ and $\underline{\lambda\mathsf{let}}$.

Second step: to add control. We introduce the class of "commands" $c ::= \chi(H|k)$, and a non-abstraction term is now $\mu a.c$. Continuations are now given by $k ::= a \mid \tilde{\mu}x.c \mid u :: k$. Sequents have full r.h.s's: for instance, $\Gamma|k : A \vdash \Delta$. Logically, we moved to classical logic.

Third step: to complete the duality. We add the dual implication $A - B$, and the class of co-terms $r ::= \tilde{\lambda}a.r \mid \tilde{\mu}x.c$. The place left vacant in the grammar of continuations by the move of $\tilde{\mu}x.c$ is occupied by the new construction $\tilde{\mathsf{hd}}(r)$. The full suite of sequents is:

$$\Gamma \vdash t : A|\Delta \qquad \Gamma|r : A \vdash \Delta \qquad \Gamma \rhd H : A|\Delta \qquad \Gamma|k : A \rhd \Delta \qquad c : (\Gamma \vdash \Delta)$$

We now easily write the constructors for the inference rules relative to $A - B$, just by dualizing those of implication: the already seen $\tilde{\lambda}a.r$ (left introduction), the co-continuation $H \tilde{::} r$ (right introduction), and the continuation $r\tilde{}k$ (elimination, on the left!). The full system is given in Fig. 11. The typing rules are omitted due to space limitations, but writing them down is now just routine.

The classical, de Morgan/Gentzen duality is the duality between hypotheses and conclusions, l.h.s. and r.h.s. of sequents, conjunction and disjunction (if these were present), $A \supset B$ and $B - A$. Gentzen praised LK for its exhibiting of this duality [8]. Let us denote it by $\tilde{(\cdot)}$, *justement*. At the level of types $A \tilde{\supset} B = \tilde{B} - \tilde{A}$ and vice-versa. Co-terms are dual

$$
\begin{array}{rrcl}
\text{(Terms)} & t,u,M,N & ::= & \lambda x.t \mid \mu a.c \\
\text{(Co-terms)} & r,s & ::= & \tilde{\lambda} a.r \mid \tilde{\mu} x.c \\
\text{(Co-continuations)} & H & ::= & x \mid \mathsf{hd}(M) \mid HN \mid H \,\tilde{::}\, r \\
\text{(Continuations)} & k & ::= & a \mid \tilde{\mathsf{hd}}(r) \mid r\tilde{\ }k \mid u :: k \\
\text{(Commands)} & c & ::= & \chi(H|k)
\end{array}
$$

$$
\begin{array}{lrcl}
(\beta) & \chi(\mathsf{hd}(\lambda x.t)|u :: k) & \to & \chi(\mathsf{hd}(u)|\tilde{\mathsf{hd}}(\tilde{\mu} x.\chi(\mathsf{hd}(t)|k))) \\
(\tilde{\beta}) & \chi(H \,\tilde{::}\, s|\tilde{\mathsf{hd}}(\tilde{\lambda} a.r)) & \to & \chi(\mathsf{hd}(\mu a.\chi(H|\tilde{\mathsf{hd}}(r)))|\tilde{\mathsf{hd}}(s)) \\
(\mu) & \chi(\mathsf{hd}(\mu a.c)|k) & \to & [k/a]c \\
(\tilde{\mu}) & \chi(H|\tilde{\mathsf{hd}}(\tilde{\mu} x.c)) & \to & [H/x]c
\end{array}
$$

$$
\begin{array}{lrcl}
(\cong) & \chi(HN|k) & = & \chi(H|N :: k) \\
(\cong) & \chi(H \,\tilde{::}\, r|k) & = & \chi(H|r\tilde{\ }k)
\end{array}
$$

Figure 11 The unified calculus.

of terms, and vice-versa. The same for co-continuations and continuations. The notation of constructions and the naming of reduction rules self-explains how $\tilde{(\cdot)}$ operates. Commands are self-dual: $\chi(\tilde{H}|k) = \chi(\tilde{k}|\tilde{H})$. The unified system is self-dual, at the level of typing and reduction. For instance, $\Gamma \vdash t : A|\Delta$ iff $\tilde{\Delta}|\tilde{t} : \tilde{A} \vdash \tilde{\Gamma}$, etc.

Notice that the duality between SC and ND links HN with $r\tilde{\ }k$ (and $H \,\tilde{::}\, r$ with $u :: k$), whereas the isomorphism between the two systems (internalized as equations in the unified system) links HN with $N :: k$ (and $H \,\tilde{::}\, r$ with $r\tilde{\ }k$).

Let us take the unified system and forbid four constructions: $\tilde{\lambda}.a.r$, HN, $H \,\tilde{::}\, r$, and $r\tilde{\ }k$. The result is a sequent calculus, a kind of classical $\overline{\lambda\tilde{\mu}}$. This system, of course, is not a self-dual system; but even if we enlarge it to a self-dual sequent calculus, SC say, by putting back the constructions $\tilde{\lambda}a.r$ and $H \,\tilde{::}\, r$ for $A - B$, the formal duality between control and co-control is not achieved yet. The formal treatment of continuations requires the presence of $u :: k$ as much as the formal treatment of co-continuations requires the presence of HN; and, once HN is in, the last forbidden construction $r\tilde{\ }k$ is put back by de Morgan/Gentzen duality.

So classical $\overline{\lambda\tilde{\mu}}$ is not self-dual, but $\overline{\lambda}\mu\tilde{\mu}$ seemingly is [4]. Actually, a critique similar to that of classical $\overline{\lambda\tilde{\mu}}$ applies to $\overline{\lambda}\mu\tilde{\mu}$. Look again at Fig. 1. Despite its compelling symmetry, $\overline{\lambda}\mu\tilde{\mu}$ does not enjoy a duality between control and co-control. Why? Commands in $\overline{\lambda}\mu\tilde{\mu}$, we may say, have the form $\langle H|e\rangle$ with $H ::= \mathsf{hd}(t)$. Hence, the constructors for e's have no dual in the class of H's: the class of e's is fully there, the class of H's is residually there. We seem to see a duality between terms t and "co-terms" e, but here the word "co-terms" is a misnomer. "Co-terms" e's are continuations, rightly captured by the μ-operator; then, either we see terms as the "dual" of continuations, and let them be captured by the $\tilde{\mu}$-operator, but then the latter, although "dual" to the μ-operator, is not a co-control operator; or the $\tilde{\mu}$-operator, if it is to be a co-control operator, should capture, not terms, but co-continuations, a missing kind of expression, which is also typed "on the right"; and the true co-terms are another missing kind of expressions typed "on the left". Notice that the distortion in $\overline{\lambda}\mu\tilde{\mu}$ has nothing to do with the fact that the dual of implication is not included in Fig. 1; if it were, one would add one constructor to the grammar of terms and its "dual" to the grammar of "co-terms", preserving the original "duality" [4], but still failing to achieve true duality, for the same reasons.

What did we learn? There is nothing wrong with classical $\overline{\lambda\tilde{\mu}}$, SC or $\overline{\lambda}\mu\tilde{\mu}$. What happens is that the classical sequent calculus, despite its symmetry, is unable to capture the formal

duality between control and co-control, because the latter requires the full extent of the de Morgan/Gentzen duality, which also involves natural deduction, and is captured only in the unified system.

6 Conclusions

Contributions. On a higher-level, this paper has two main contributions. The first is the intended one, about the Curry-Howard isomorphism for sequent calculus. If systems of combinators correspond to Hilbert systems, and the ordinary λ-calculus corresponds to natural deduction, what variant of the λ-calculus does correspond to the sequent calculus? We propose a clear-cut answer, which turns out to be a coin with two faces: sequent calculus corresponds to a formal vector notation with first-class co-control; and to a bi-directional, agnostic, computational λ-calculus.

The second contribution concerns structural proof theory. We knew from our past experience [7, 17] that sequent calculus has to be developed hand-in-hand with natural deduction. And this was confirmed here in many ways. For instance, co-continuations correspond to a primitive syntactic class in natural deduction, from where one can import, say, the typing rules. Also, things look very different in the other side of the sequent-calculus-vs-natural-deduction mirror, different to the point of un-recognizability. For instance, co-control is almost invisible in natural deduction. The surprise came when we moved to classical logic in order to prove the duality between control and co-control. There we learned that the de Morgan/Gentzen duality comprehends the duality between control and co-control, as long as we unify sequent calculus and natural deduction.

On the technical level, the main contribution is the formulation of co-control. The formulation is entirely based on the identification of the concept of co-continuation. This is the entity variables in sequent calculus proof terms stand for, and with which one may formulate the $\tilde{\mu}$-operator as a co-control operator, dualizing the behavior of the μ-operator. We showed the meaning of co-control in natural deduction, and how co-control subsumes a form of focalization. Finally, it is also noteworthy: (i) The analysis contained in the agnosticism theorem of the superimposition of CBN and CBV present in $\underline{\lambda\mathsf{let}}$ (and $\overline{\lambda\tilde{\mu}}$); (ii) The treatment of the logical operation $A - B$ contained in the unified system.

Related and future work. The author apologizes for the title of this paper, if the reader finds it exaggerated. True, the author believes something simultaneously new and very basic was said here about the computational interpretation of the sequent calculus. On the other hand, this contribution corresponds a small step from the wisdom accumulated before. Specifically, our proposal starts from the following ingredients: the $\overline{\lambda}$-calculus [10], the $\tilde{\mu}$-operator [4], the vector notation [11], the technical result about formal vector notation [6, 3, 5], together with the previous work by the author [7, 17].

It is clear why co-control has been unnoticed in the theory and practice of programming: in a syntax with the natural deduction format, co-control control corresponds to a low-profile substitution triggering rule. Co-control, as such, is only visible in a syntax with the sequent calculus format. Now, such kind of syntax is usually regarded as a machine representation. Therefore, it is natural to ask whether co-control is relevant in the theory and practice of functional languages implementation. This question deserves further investigation.

Acknowledgements. The author was supported by Fundação para a Ciência e Tecnologia through project PEst-OE/MAT/UI0013/2014.

References

1. H. Barendregt and S. Ghilezan. Lambda terms for natural deduction, sequent calculus and cut elimination. *Journal of Functional Programming*, 10(1):121–134, 2000.
2. S. Cerrito and D. Kesner. Pattern matching as cut elimination. In *Proceedings of 14th annual IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 98–108, 1999.
3. I. Cervesato and F. Pfenning. A linear spine calculus. *J. Log. Compt.*, 13(5):639–688, 2003.
4. P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada, September 18-21, 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000.
5. R. Dyckhoff and C. Urban. Strong normalisation of Herbelin's explicit substitution calculus with substitution propagation. *Journal of Logic and Computation*, 13(5):689–706, 2003.
6. J. Espírito Santo. Revisiting the correspondence between cut-elimination and normalisation. In *Proceedings of ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 600–611. Springer-Verlag, 2000.
7. Jo. Espírito Santo. The λ-calculus and the unity of structural proof theory. *Theory of Computing Systems*, 45:963–994, 2009.
8. G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North Holland, 1969.
9. J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. C.U.P., 1989.
10. H. Herbelin. A λ-calculus structure isomorphic to a Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1995.
11. F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. *Arch. for Math. Logic*, 42:59–87, 2003.
12. E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.
13. M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classic natural deduction. In *Int. Conf. Logic Prog. Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Verlag, 1992.
14. B. Pierce and D. Turner. Local type inference. In *Proc. of POPL'98*, pages 252–265. ACM, 1998.
15. D. Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.
16. A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997.
17. José Espírito Santo. Towards a canonical classical natural deduction system. *Ann. Pure Appl. Logic*, 164(6):618–650, 2013.
18. M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.
19. Noam Zeilberger. On the unity of duality. *Ann. Pure Appl. Logic*, 153(1-3):66–96, 2008.

Dependent Types for Nominal Terms with Atom Substitutions

Elliot Fairweather[1], Maribel Fernández[1], Nora Szasz[2], and Alvaro Tasistro[2]

1 **King's College London, UK**
2 **Universidad ORT Uruguay, Uruguay**

Abstract

Nominal terms are an extended first-order language for specifying and verifying properties of syntax with binding. Founded upon the semantics of nominal sets, the success of nominal terms with regard to systems of equational reasoning is already well established. This work first extends the untyped language of nominal terms with a notion of non-capturing atom substitution for object-level names and then proposes a dependent type system for this extended language. Both these contributions are intended to serve as a prelude to a future nominal logical framework based upon nominal equational reasoning and thus an extended example is given to demonstrate that this system is capable of encoding various other formal systems of interest.

1998 ACM Subject Classification F.4.1 Mathematical Logic, lambda calculus and related systems, D.3.3 Language Constructs and Features, data types and structures, frameworks

Keywords and phrases α-equivalence, nominal term, substitution, dependent type

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.180

1 Introduction

There exist many formal systems described by a syntax that makes use of name binding constructs. Nominal terms [35, 15, 14], are, by now, a well-established approach to the specification and verification of properties of such languages and systems. Based upon the name abstraction semantics of nominal sets [23], nominal terms use the properties of permutations of object-level names or 'atoms' to provide an explicit formalisation for both the use of side conditions on names and for the axiomatisation of alpha-equivalence between object-level terms. Following the development of efficient algorithms for matching [4] and unification [5, 25], nominal terms have been applied in rewriting [15, 16, 12] and unoriented equational reasoning [22, 10].

In these works, the capture-avoiding substitution used in many systems of interest has, thus far, needed to be encoded by explicit rewrite rules or axioms for the syntax in question. The first contribution of the present work addresses this issue by extending the language of nominal terms with a notion of capture-avoiding atom substitution at the object-level. Definitions of freshness, alpha-equivalence and matching together with informal proofs of decidability are provided for the new syntax.

The second contribution of this paper is the definition of a proposed dependent type system for the extended language. The language of the type system presented in this paper is user-defined; users define an interdependent signature of type- and term-constructors of interest and give their type declarations. It is the responsibility of the user to maintain the adequacy of their encoding and thus to declare types in accordance with the system to be

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 180–195

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

formalised. In keeping with earlier work on nominal equational reasoning and in contrast with previous work on nominal type theories [8, 9, 32], the system given here remains 'lambda-free'.

Both these contributions are presented as foundations for the future development of a logical framework based upon nominal equational reasoning. A logical framework is a formal system that provides the facility to define a number of other formal systems, such as programming languages, mathematical structures and logics, by abstraction of their common features [24, 2, 27, 30, 29]. Many logical frameworks are developed as a type system; the minimum expressive power desirable of such a type system is that types be allowed to depend upon terms.

The type system presented here is one of such expressive power and to demonstrate this an extended example is given in the form of a specification for first-order logic for which adequacy is proven. Equality axioms are not yet considered in this prototype 'framework' but in the future it is expected that the system be expanded to include a user-defined nominal equational theory, specified as a set of equality axioms or rewriting rules in the style of [20, 15, 16].

2 Syntax

Consider countably infinite, pairwise disjoint sets of **atoms**, $a, b, c, \ldots \in \mathbb{A}$, **variables**, $X, Y, Z, \ldots \in \mathbb{X}$, **term-formers**, $\mathsf{f}, \mathsf{g}, \ldots \in \mathbb{F}$, and **type-formers**, $\mathsf{C}, \mathsf{C}', \ldots \in \mathbb{C}$. The syntax of **permutations**, π, **atom substitutions**, ϑ, **pseudo-terms**, s, t, and **pseudo-types**, σ, τ, is defined by mutual induction and generated by the grammar in Definition 1.

▶ **Definition 1** (Syntax).

$$\begin{array}{llllll} \pi & ::= & \mathsf{id} \mid \pi * (a\ b) & s,\ t & ::= & a \mid [a\colon \sigma]\, t \mid \mathsf{f}\, t \mid (t_1,\ \ldots,\ t_n) \mid\ \vartheta \vert \pi \cdot X \\ \vartheta & ::= & \mathsf{id} \mid [a \mapsto t] * \vartheta & \sigma,\ \tau & ::= & [a\colon \sigma]\, \tau \mid \mathsf{C}\, t \mid (\tau_1 \times\ \ldots \times \tau_n) \end{array}$$

A **permutation** is a bijection on the set of atoms, $\mathbb{A}$, represented as a list of swappings, such that $\pi(a) \neq a$ for finitely many atoms, $a \in \mathbb{A}$. $\pi(a)$ is easily computed using swappings (we omit the inductive definition). An **atom substitution** is a mapping from atoms to pseudo-terms, equal to the identity mapping but for finitely many arguments. This mapping is represented as a list of pairs, ϑ, of the form, $[a_i \mapsto t_i]$ such that the atoms, a_i, are *pairwise distinct*. This list is interpreted as a set of *simultaneous* bindings and not as a sequence, and thus the value of an atom substitution is determined directly from the syntactic representation. The final id and 'list cons' operators in the syntax for both permutations and atom substitutions are commonly omitted. Atom substitutions act upon pseudo-terms and pseudo-types by instantiating atoms and are 'capture-avoiding'. Atom substitutions suspend upon variables and are applied *after* a suspended permutation.

The constructions for pseudo-terms are called respectively **atom terms**, **abstractions**, **function applications**, **tuples** (where $n \geq 0$) and **moderated variables** and those for pseudo-types, **abstraction types**, **constructed types** and **product types** ($n \geq 0$). Abbreviate $\mathsf{f}\,()$ as f, and $\mathsf{C}\,()$ as C. Let $M, N, \ldots$ range over elements of the union of the set of pseudo-terms and set of pseudo-types. There is only one kind of well-formed types: type.

▶ **Definition 2** (Permutation Action).

$$\pi \cdot \mathsf{id} \triangleq \mathsf{id} \qquad \pi \cdot ([a \mapsto t] * \vartheta) \triangleq [\pi(a) \mapsto \pi \cdot t] * (\pi \cdot \vartheta)$$

$$\pi \cdot a \triangleq \pi(a) \qquad \pi \cdot (\vartheta\big|\pi' \cdot X) \triangleq \pi \cdot \vartheta\big|(\pi \mathbin{@} \pi') \cdot X$$
$$\pi \cdot [a\colon \sigma]\, t \triangleq [\pi(a)\colon \pi \cdot \sigma]\, (\pi \cdot t) \qquad \pi \cdot [a\colon \sigma]\, \tau \triangleq [\pi \cdot a\colon \pi \cdot \sigma]\, (\pi \cdot \tau)$$
$$\pi \cdot \mathsf{f}\, t \triangleq \mathsf{f}\, (\pi \cdot t) \qquad \pi \cdot \mathtt{C}\, t \triangleq \mathtt{C}\, (\pi \cdot t)$$
$$\pi \cdot (t_1,\ \ldots,\ t_n) \triangleq (\pi \cdot t_1,\ \ldots,\ \pi \cdot t_n) \qquad \pi \cdot (\tau_1 \times \ldots \times \tau_n) \triangleq (\pi \cdot \tau_1 \times \ldots \times \pi \cdot \tau_n)$$

Call $a \# M$ a **freshness constraint**. Let $\Delta,\ \nabla$, range over sets of freshness constraints of the form $a \# X$; call such sets **freshness contexts**. Write $\Delta \vdash a \# M$ when a derivation exists using the rules given in Definition 3 below; in rule $(atm)^{\#}$ we assume $a \neq b$.

▶ **Definition 3** (Freshness Relation).

$$\frac{}{\Delta \vdash a \# b}\,(atm)^{\#} \qquad \frac{\mathsf{img}^{\#}(\Delta,\ a,\ \vartheta\big|\pi \cdot X) \quad a \in \mathsf{dom}(\vartheta)}{\Delta \vdash a \# \vartheta\big|\pi \cdot X}\,(var : aa)^{\#}$$

$$\frac{\mathsf{img}^{\#}(\Delta,\ a,\ \vartheta\big|\pi \cdot X) \quad a \notin \mathsf{dom}(\vartheta) \quad \pi^{-1}(a) \# X \in \Delta}{\Delta \vdash a \# \vartheta\big|\pi \cdot X}\,(var : ab)^{\#}$$

$$\frac{\Delta \vdash a \# \tau}{\Delta \vdash a \# [a\colon \tau]\, s}\,(abs : aa)^{\#} \qquad \frac{\Delta \vdash a \# s \quad \Delta \vdash a \# \tau}{\Delta \vdash a \# [b\colon \tau]\, s}\,(abs : ab)^{\#}$$

$$\frac{\Delta \vdash a \# s_1 \ \ldots\ \Delta \vdash a \# s_n}{\Delta \vdash a \# (s_1,\ \ldots,\ s_n)}\,(tpl)^{\#} \qquad \frac{\Delta \vdash a \# s}{\Delta \vdash a \# \mathsf{f}\, s}\,(app)^{\#}$$

$$\frac{\Delta \vdash a \# \sigma}{\Delta \vdash a \# [a\colon \sigma]\, \tau}\,(abt : aa)^{\#} \qquad \frac{\Delta \vdash a \# \tau \quad \Delta \vdash a \# \sigma}{\Delta \vdash a \# [b\colon \sigma]\, \tau}\,(abt : ab)^{\#}$$

$$\frac{\Delta \vdash a \# \tau_1 \ \ldots\ \Delta \vdash a \# \tau_n}{\Delta \vdash a \# (\tau_1 \times \ldots \times \tau_n)}\,(prd)^{\#} \qquad \frac{\Delta \vdash a \# t}{\Delta \vdash a \# \mathtt{C}\, t}\,(cns)^{\#}$$

The main differences with respect to the freshness relation for nominal terms are the introduction of new rules for types, $(abt : aa)^{\#}$, $(abt : ab)^{\#}$, $(prd)^{\#}$ and $(cns)^{\#}$, and the rules for moderated variables, $(var : aa)^{\#}$ and $(var : ab)^{\#}$, which take into account suspended atom substitutions as well as suspended permutations. The notation, $\mathsf{img}^{\#}(\Delta,\ a,\ \vartheta\big|\pi \cdot X)$, in these rules, defines the conditions necessary for freshness with regard to the suspended atom substitution and is an abbreviation of the following finite set of hypotheses.

$$\left\{ \left(\Delta \vdash a \# \vartheta(\pi(b))\right) \vee \left(b \# X \in \Delta\right) \mid b \in \mathbb{A},\ \pi(b) \in \mathsf{dom}(\vartheta) \right\}$$

This ensures that the substitution will not introduce the atom a when it is applied to an instance of X. However this disjunction of conditions results in the possibility of multiple derivations $\Delta_i \vdash a \# t$ for a given freshness constraint $a \# t$. If one considers the suspended atom substitution, ϑ, to be id, the conditions upon ϑ are satisfied vacuously and the two rules clearly reduce to that for nominal terms.

Call $M \approx_{\alpha} N$ an **alpha-equality constraint** and write $\Delta \vdash M \approx_{\alpha} N$ when a derivation exists using the rules given in Definition 4 below. Note that because alpha-equivalence is defined using freshness, again multiple possible derivations may exist for a given constraint.

▶ **Definition 4** (Alpha-equivalence Relation).

$$\frac{}{\Delta \vdash a \approx_\alpha a}\,(atm)^\alpha \qquad \frac{\forall a \in \mathsf{ds}(\vartheta_1|\pi_1,\ \vartheta_2|\pi_2).\ a \# X \in \Delta}{\Delta \vdash \vartheta_1|\pi_1 \cdot X \approx_\alpha \vartheta_2|\pi_2 \cdot X}\,(var)^\alpha$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1\ \ldots\ \Delta \vdash s_n \approx_\alpha t_n}{\Delta \vdash (s_1,\ \ldots,\ s_n) \approx_\alpha (t_1,\ \ldots,\ t_n)}\,(tpl)^\alpha \qquad \frac{\Delta \vdash \sigma \approx_\alpha \tau\ \ \Delta \vdash s \approx_\alpha t}{\Delta \vdash [a\colon \sigma]\, s \approx_\alpha [a\colon \tau]\, t}\,(abs : aa)^\alpha$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{f}\, s \approx_\alpha \mathsf{f}\, t}\,(app)^\alpha \qquad \frac{\Delta \vdash \sigma \approx_\alpha \tau\ \ \Delta \vdash a \# t\ \ \Delta \vdash s \approx_\alpha (a\ b) \cdot t}{\Delta \vdash [a\colon \sigma]\, s \approx_\alpha [b\colon \tau]\, t}\,(abs : ab)^\alpha$$

$$\frac{\Delta \vdash \sigma_1 \approx_\alpha \tau_1\ \ldots\ \Delta \vdash \sigma_n \approx_\alpha \tau_n}{\Delta \vdash (\sigma_1 \times \ldots \times \sigma_n) \approx_\alpha (\tau_1 \times \ldots \times \tau_n)}\,(prd)^\alpha \qquad \frac{\Delta \vdash \sigma_1 \approx_\alpha \tau_1\ \ \Delta \vdash \sigma_2 \approx_\alpha \tau_2}{\Delta \vdash [a\colon \sigma_1]\, \sigma_2 \approx_\alpha [a\colon \tau_1]\, \tau_2}\,(abt : aa)^\alpha$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{C}\, s \approx_\alpha \mathsf{C}\, t}\,(cns)^\alpha \qquad \frac{\Delta \vdash \sigma_1 \approx_\alpha \tau_1\ \ \Delta \vdash a \# \tau_2\ \ \Delta \vdash \sigma_2 \approx_\alpha (a\ b) \cdot \tau_2}{\Delta \vdash [a\colon \sigma_1]\, \sigma_2 \approx_\alpha [b\colon \tau_1]\, \tau_2}\,(abt : ab)^\alpha$$

This presentation of alpha-equivalence is defined by induction on the size of the pair, (M, N), and is both syntax-directed and decidable when considered as a recursive predicate. It is a generalisation of the notion of alpha-equivalence on nominal terms.

The only case that is not straightforward is again that of a moderated variable, $(var)^\alpha$. Here it is important to remember that both permutations and atom substitutions are finite mappings and that the image of a suspended atom substitution is given as a sub-term of the syntax of the moderated variable. Thus, the disagreement set of two suspensions, $\vartheta_1|\pi_1$ and $\vartheta_2|\pi_2$, written $\mathsf{ds}(\vartheta_1|\pi_1,\ \vartheta_2|\pi_2)$, is also finite and may be defined as $\{a \mid \vartheta_1(\pi_1(a)) \not\approx_\alpha \vartheta_2(\pi_2(a)),\ a \in \mathbb{A}\}$, which although recursive, is of decreasing size. The reflexivity, symmetry and transitivity of the alpha-equivalence relation have been proved by adapting the proofs given in [15], themselves simplified from those in [35].

The action of an atom substitution, ϑ, upon a pseudo-term or pseudo-type, M, written, $M\,\vartheta$, is defined by induction in the presence of a freshness context, Δ, in Definition 5. For the sake of clarity of presentation this freshness context is not explicitly written throughout the definition. Let ϑ^{-a} denote the atom substitution ϑ restricted to the domain, $\mathsf{dom}(\vartheta) \setminus \{a\}$. The composition of two atom substitutions, written $\vartheta_1 \circ \vartheta_2$, is defined as the atom substitution equivalent to applying ϑ_1 followed by ϑ_2. The syntactic construction of such a composition built from two substitutions represented as sets of bindings can be defined by adapting the algorithm described in [3, 2.1]. Note that this operation itself uses the action of an atom substitution upon pseudo-term and so must be defined simultaneously with Definition 5 and is also parameterised by the freshness context, Δ.

▶ **Definition 5** (Action of Atom Substitution).

$$(\vartheta'|\pi \cdot X)\,\vartheta \triangleq (\vartheta' \circ \vartheta)|\pi \cdot X \qquad a\,\vartheta \triangleq \vartheta(a);\ a \in \mathsf{dom}(\vartheta) \qquad a\,\vartheta \triangleq a;\ a \notin \mathsf{dom}(\vartheta)$$

$$([a\colon \sigma]\, s)\,\vartheta \triangleq [c\colon \sigma\,\vartheta]\,((a\ c) \cdot s)\,\vartheta^{-c};\ \Delta \vdash c \# s,\ c \# \mathsf{img}(\vartheta)$$

$$(\mathsf{f}\, s)\,\vartheta \triangleq \mathsf{f}\,(s\,\vartheta) \qquad (t_1,\ \ldots,\ t_n)\,\vartheta \triangleq (t_1\,\vartheta,\ \ldots,\ t_n\,\vartheta)$$

$$([a\colon \sigma]\, \tau)\,\vartheta \triangleq [c\colon \sigma\,\vartheta]\,((a\ c) \cdot \tau)\,\vartheta^{-c};\ \Delta \vdash c \# \tau,\ c \# \mathsf{img}(\vartheta)$$

$$(\mathsf{C}\, s)\,\vartheta \triangleq \mathsf{C}\,(s\,\vartheta) \qquad (\tau_1 \times \ldots \times \tau_n)\,\vartheta \triangleq (\tau_1\,\vartheta \times \ldots \times \tau_n\,\vartheta)$$

The capture-avoidance of unabstracted atoms is ensured by the fact that when an atom substitution acts upon an abstraction or abstraction type, a suitable alpha-equivalent representative is first chosen with respect to the freshness context, Δ. In practice, this

presentation will result in the creation of freshness constraints for atoms, newly-generated with respect to the system as a whole, and is similar to the approach taken in [16]. A suitable 'even fresher' atom always exists, and it is one's right to add constraints for that atom to the freshness context, a fact which is taken advantage of below in Definition 9. Any implementation of this definition as a recursive function must accommodate a suitable mechanism for the generation of such names; this is most easily achieved by the threading of global state throughout the function or by the use of a global choice function that returns the next available name. Atom substitutions work uniformly on alpha-equivalence classes of pseudo-terms and pseudo-types.

▶ **Definition 6** (Variable Substitutions)**.** A **variable substitution** is a mapping from variables to pseudo-terms, equal to the identity mapping but for finitely many arguments, and written as a set of bindings $[X_1 \mapsto s_1] \ldots [X_n \mapsto s_n]$, such that the variables, $X_1, \ldots, X_n$, are pairwise distinct.

The action of variable substitutions upon atom substitutions, pseudo-terms and pseudo-types is given in Definition 7. A variable substitution, θ, acts upon an atom substitution, ϑ, by instantiating the variables occurring in the pseudo-terms of the image of ϑ, and is written $\vartheta\,\theta$. Note that the instantiation of a variable requires the application of an atom substitution to a pseudo-term and thus the action of variable substitutions is also parameterised by a freshness context, which again is left implicit in the definition below.

▶ **Definition 7** (Variable Substitution Action)**.**

$$\mathsf{id}\,\theta \triangleq \mathsf{id} \qquad ([a \mapsto t] * \vartheta)\,\theta \triangleq [a \mapsto t\,\theta] * (\vartheta\,\theta)$$

$$a\,\theta \triangleq a$$

$$(\vartheta \big| \pi \cdot X)\,\theta \triangleq (\pi \cdot \theta(X))\,(\vartheta\,\theta);\ X \in \mathsf{dom}(\theta) \qquad (\vartheta \big| \pi \cdot X)\,\theta \triangleq (\vartheta\,\theta) \big| \pi \cdot X;\ X \notin \mathsf{dom}(\theta)$$

$$([a\colon \sigma]\,t)\,\theta \triangleq [a\colon \sigma\,\theta]\,(t\,\theta) \qquad (\mathsf{f}\,t)\,\theta \triangleq \mathsf{f}\,(t\,\theta) \qquad (t_1, \ldots, t_n)\,\theta \triangleq (t_1\,\theta, \ldots, t_n\,\theta)$$

$$([a\colon \sigma]\,\tau)\,\theta \triangleq [a\colon \sigma\,\theta]\,(\tau\,\theta) \qquad (\mathtt{C}\,t)\,\theta \triangleq \mathtt{C}\,(t\,\theta) \qquad (\tau_1 \times \ldots \times \tau_n)\,\theta \triangleq (\tau_1\,\theta \times \ldots \times \tau_n\,\theta)$$

The type system introduced in Section 3 requires a formalisation of matching to check that term-formers and type-formers are used in a way that is consistent with their respective type declarations. The concepts of constraint problems and matching are now therefore extended to nominal terms with atom substitutions.

Let C range over freshness and alpha-equality constraints; a **constraint problem**, $\mathcal{C}$, is an arbitrary set of such constraints. Extend the above notations for the derivability of constraints element-wise to constraint problems; thus, write $\Delta \vdash \{C_1, \ldots, C_n\}$ for $\Delta \vdash C_1$, $\ldots$, $\Delta \vdash C_n$. Substitution action extends naturally to constraints and constraint problems.

▶ **Definition 8** (Matching Problem)**.** Given a constraint problem, $\mathcal{C}$, $\{\ldots, a_j \# Q_j, \ldots, M_i \approx_\alpha N_i, \ldots\}$, a corresponding **matching problem** is defined if $(\bigcup_i \mathsf{vars}(M_i)) \cap (\bigcup_i \mathsf{vars}(N_i)) = \varnothing$ and is written $\{\ldots, a_j \# Q_j, \ldots, M_i \stackrel{?}{\approx}_\alpha N_i, \ldots\}$.

A solution to such a problem, if one exists, is a pair, (Δ, θ), of a freshness context, Δ, and a variable substitution, θ, such that $\mathsf{dom}(\theta) \subseteq \bigcup \mathsf{vars}(M_i)$ and $\Delta \vdash \mathcal{C}\,\theta$.

Informally, this says that a matching problem is a constraint problem in which one adds the restriction that the variables in the left-hand sides of alpha-equality constraints are disjoint from the variables in the right-hand sides and that only variables in the left-hand

sides of equality constraints may be instantiated. There may be several Δ_i such that $\Delta_i \vdash \mathcal{C}\,\theta$. As in the case of nominal terms, one can define an ordering between solutions (Δ_i, θ) and define a most general solution as a least element in the ordering. However, unlike nominal matching, here there is no unique most general solution.

The situation is similar for logical frameworks based on the lambda calculus. The solution there is to restrict the form of matching problems. Inspired by LF, [30], the type system in Section 3 is designed so that one only needs to match against a pattern term, t, such that for any variable, $X \in \mathsf{vars}(t)$, then $\mathsf{id}\big|\pi \cdot X$ is a sub-term of t. Thus, the value of X is uniquely determined. Note that a solution may only instantiate variables in the pattern and so if an atom substitution occurs in the matched term then it can be treated as a constant sub-term. Using this assumption, if a matching problem has a solution, there is a unique most general one. The algorithm to compute it is similar to that used to check alpha-equivalence, except that when a variable sub-term, $\vartheta\big|\pi \cdot X$, of the pattern is being matched against, if ϑ is not id then that constraint is postponed until after the constraint for the occurrence of the $\mathsf{id}\big|\pi \cdot X$ sub-term in the pattern has been solved and a unique variable substitution generated.

In addition, it is also important to note that due to the action of atom substitutions suspended upon variables, the freshness context of a solution may contain constraints for atoms, $a \notin \mathsf{atms}(\mathcal{C})$.

▶ **Definition 9** (Pattern Matching Problem)**.** A **pattern matching problem**, consists of two pseudo-terms-in-context or two pseudo-types-in-context, $\nabla \vdash M$ and $\Delta \vdash N$, to be matched, where $\mathsf{vars}(\nabla \vdash M) \cap \mathsf{vars}(\Delta \vdash N) = \varnothing$ and is written $(\nabla \vdash M) \stackrel{?}{\approx}_\alpha (\Delta \vdash N)$.

A solution to such a problem, if one exists, is a variable substitution, θ, such that (∇', θ) is a solution to the matching problem $\nabla \cup \{M \stackrel{?}{\approx}_\alpha N\}$ and there exists a freshness context $\Delta^\#$ (of which each constraint, $a \# X$, is such that $a \notin \mathsf{atms}(M) \cup \mathsf{atms}(N) \cup \mathsf{atms}(\nabla)$), such that $\Delta \cup \Delta^\# \vdash \nabla'$.

A **newly-freshened variant** of a term, t, is a term, written t^ν, in which all the atoms and variables have been replaced by newly generated atoms and variables with respect to those occurring in t (and maybe other elements of syntax, always specified.)

Closed terms were introduced in [15] and shown there to be decidable by an algorithm using newly-freshened variants and nominal matching. Intuitively, a closed term has no unabstracted atoms and all occurrences of a variable must appear under the *same* abstracted atoms. Here, closedness can be checked in a similar way using the matching algorithm mentioned above.

3 Type System

This section starts by introducing the syntax of environments, declarations and judgements used in this type system. The validity of environments (Definition 11), validity of sets of declarations (Definition 10), and derivability of typing judgements (Definition 13), are then defined by mutual induction.

A **type association** is a pair of a variable, X, and a type, σ, written $(X\colon \sigma)$ or an atom and a type, written $(a\colon \sigma)$. A **pseudo-environment**, Γ, is an ordered list of type associations. A pseudo-environment may contain at most one type association for each variable and atom. Here, let $\Gamma \bowtie (a\colon \tau)$, denote the result of **appending** $(a\colon \tau)$ to the end of the list that represents the pseudo-environment, Γ (similarly for a variable association) and let this notation be extended element-wise to lists of associations. The association available for a given atom, a, in Γ is denoted by Γ_a. If there is no type association for a in Γ then Γ_a is

undefined, written $\bot$ (similarly for a variable.) The domain of Γ, denoted $\mathsf{dom}(\Gamma)$, and image of Γ, denoted $\mathsf{img}(\Gamma)$, are defined as usual: $\mathsf{dom}(\Gamma) = \{a \in \mathbb{A} \mid \Gamma_a \neq \bot\} \cup \{X \in \mathbb{X} \mid \Gamma_X \neq \bot\}$ and $\mathsf{img}(\Gamma) = \{\tau \mid \exists a, \Gamma_a = \tau\} \cup \{\tau \mid \exists X, \Gamma_X = \tau\}$. It is important to note that type associations for variables are *never* appended by the typing rules, however the rules for abstractions and abstraction types do append type associations for atoms.

A **pseudo-declaration**, $\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \langle \sigma \hookrightarrow \tau \rangle$ or $\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \langle \sigma \hookrightarrow \mathsf{type} \rangle$, states the type associations and freshness constraints that a term must satisfy in order that an application or constructed type built from that term be well-formed. Thus, informally, this says that if under the type associations in Γ, and the freshness constraints in Δ, t has type σ then $\mathsf{f}\, t$ has type τ, or similarly that $\mathtt{C}\, t$ is of the kind type. In practice, users need not give complete declarations; it is sufficient to write $\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \tau$ or $\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \mathsf{type}$ and the system will infer the complete declaration by computing the type of t.

Pseudo-declarations are given for a term-former or type-former *together* with an argument term in order to allow the use of atoms of that argument in the type of the application or constructed type; for example, see the declarations for $\mathsf{all_i}$ and $\mathsf{all_e}$ in Section 5.

A **pseudo-judgement**, $\Gamma \Vdash_\Sigma \Delta \vdash t \colon \tau$ or $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$, specifies that under a given environment, set of declarations and freshness context, either a term has a particular type or a type is well-formed.

A valid set of declarations, Σ, written $\mathsf{validD}(\Sigma)$, is defined inductively as follows.

▶ **Definition 10** (Valid Set of Declarations)**.**

- The empty set of declarations, $\varnothing$, is valid; $\mathsf{validD}(\varnothing)$.
- If $\mathsf{validD}(\Sigma)$, $\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \langle \sigma \hookrightarrow \tau \rangle$ is a valid declaration under the following conditions.
  - $\Gamma \Vdash_\Sigma \Delta \vdash t \colon \sigma$, where $\mathsf{vars}(\sigma) \subseteq \mathsf{vars}(t) \cup \mathsf{vars}(\tau)$
  - $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$, where τ is not an abstraction type
  - $\Delta \vdash \{t,\ \sigma,\ \tau\}$ is closed
  - for any variable, $X \in \mathsf{vars}(t) \cup \mathsf{vars}(\tau)$, then the sub-term, $\mathsf{id}\big|\pi \cdot X$, occurs in either t or τ.

  Then, provided that there is no declaration in Σ for the term-former, f, it holds that $\mathsf{validD}(\Sigma \cup \{\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \langle \sigma \hookrightarrow \tau \rangle\})$.
- If $\mathsf{validD}(\Sigma)$, then $\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \langle \sigma \hookrightarrow \mathsf{type} \rangle$ is a valid declaration under the following conditions.
  - $\Gamma \Vdash_\Sigma \Delta \vdash t \colon \sigma$, where $\mathsf{vars}(\sigma) \subseteq \mathsf{vars}(t)$
  - $\Delta \vdash \{t,\ \sigma\}$ is closed
  - for any variable, $X \in \mathsf{vars}(t)$ then the sub-term, $\mathsf{id}\big|\pi \cdot X$, occurs in t

  Then, provided that there is no declaration in Σ for the type-former, $\mathtt{C}$, $\mathsf{validD}(\Sigma \cup \{\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \langle \sigma \hookrightarrow \mathsf{type} \rangle\})$.

Assuming a freshness context, Δ, and a valid set of declarations, Σ, a valid environment, written $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$, is defined as follows.

▶ **Definition 11** (Valid Environments)**.**

- The empty list of type associations, $-$, is a valid environment; $\mathsf{validE}(-,\ \Sigma,\ \Delta)$.
- If $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$, then $\mathsf{validE}(\Gamma \bowtie (a \colon \tau),\ \Sigma,\ \Delta \cup \Delta')$, for any Δ' that does not mention atoms in $\mathsf{dom}(\Gamma)$, provided that $a \notin \mathsf{dom}(\Gamma)$, $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$.
- If $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$, then $\mathsf{validE}(\Gamma \bowtie (X \colon \tau),\ \Sigma,\ \Delta \cup \Delta')$, for any Δ' that does not mention atoms in $\mathsf{dom}(\Gamma)$, provided that $X \notin \mathsf{dom}(\Gamma)$, for any atom, a, such that $\Delta \vdash a \# X$ then $\Delta \vdash a \# \tau, a \# \mathsf{img}(\Gamma)$, and $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$.

Informally, the final condition upon variable associations says that in a valid environment, if an atom cannot occur unabstracted in an instance of a variable, X, then it cannot occur unabstracted in the typing of that variable either.

Typing judgements are derived inductively using the rules given in Definition 13. The declarations provided by the user are required in the rules for term-formers, $(app)^\tau$, and type-formers, $(cns)^\tau$. Declarations are matched to pseudo-terms and pseudo-types using pattern matching and therefore each time an application or constructed type is typed a newly-freshened variant of the declaration for that term-former or type-former is generated; we write $\Theta \Vdash \nabla \vdash \mathsf{f}\, s : \langle \sigma \hookrightarrow \rho \rangle \in \Sigma^{\prime\prime}$ (similarly for a constructed type) to emphasise the fact that such a newly-freshened variant of a declaration is being used. In the rule for a variable, $(var)^\tau$, a predicate, typV is used.

▶ **Definition 12.** Let $\Gamma\,|_X$, for $X \in \mathsf{dom}(\Gamma)$, be the list of associations in Γ up to but not including the pair $(X : \tau)$. Write $\mathsf{typV}(\Gamma,\ \Sigma,\ \Delta,\ \vartheta\big|\pi,\ X)$ if the following conditions hold.

- for any atom $a \in \mathsf{dom}(\Gamma\,|_X)$, either $\Delta \vdash a \# X$ or $\Gamma \Vdash_\Sigma \Delta \vdash \vartheta(\pi(a)) : (\pi \cdot \Gamma_a)\,\vartheta$
- for any variable $Y \in \mathsf{dom}(\Gamma\,|_X)$, $\Gamma \Vdash_\Sigma \Delta \vdash \vartheta\big|\pi \cdot Y : (\pi \cdot \Gamma_Y)\,\vartheta$
- $\Gamma \Vdash_\Sigma \Delta \vdash (\pi \cdot \Gamma_X)\,\vartheta : \mathsf{type}$

This definition indicates that for any atom, a, that can occur unabstracted in an instance of the variable, X, then $\vartheta(\pi(a))$ must be typeable with a type compatible with the type of a. Similarly, variables that may be used in the typing of X (i.e., that occur in Γ before X) also have a type compatible with $\vartheta\big|\pi$.

In the rule $(abs)^\tau$, the atom b is not in the set of atoms occurring in the judgement, $\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, t : [a : \sigma]\, \tau$ and the freshness context, $\Delta^\#$, is such that $\Delta^\# \vdash b \# t,\ b \# \sigma,\ b \# \tau$. Similarly in $(abt)^\tau$, b is not in the set of atoms occurring in $\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, \tau : \mathsf{type}$ and $\Delta^\#$, is such that $\Delta^\# \vdash b \# \sigma,\ b \# \tau$.

In the rule $(app)^\tau$, $\mathsf{sol}(\theta)$ means that the variable substitution, θ, is a most general solution to the pattern matching problem $(\nabla \vdash (\mathtt{C}\, s \times \rho)) \stackrel{?}{\approx}_\alpha (\Delta \cup \Delta^\# \vdash (\mathtt{C}\, t \times \tau))$, where $\Delta^\#$ is a freshness context such that $\mathsf{atms}(s) \cup \mathsf{atms}(\rho) \# \mathsf{vars}(t) \cup \mathsf{vars}(\tau)$ and $\mathtt{C}$, is an arbitrary type-constructor used so that the argument term and application type may both be included within the same pattern matching problem. Similarly in $(cns)^\tau$, θ is a most general solution to $(\nabla \vdash s) \stackrel{?}{\approx}_\alpha (\Delta \cup \Delta^\# \vdash t)$ where $\Delta^\#$ is $\mathsf{atms}(s) \# \mathsf{vars}(t)$.

▶ **Definition 13** (Typing rules)**.**

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta) \quad (\pi \cdot \Gamma_X)\,\vartheta = \tau \quad \mathsf{typV}(\Gamma,\ \Sigma,\ \Delta,\ \vartheta\big|\pi,\ X)}{\Gamma \Vdash_\Sigma \Delta \vdash \vartheta\big|\pi \cdot X : \tau}\,(var)^\tau$$

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta) \quad \Gamma_a = \tau}{\Gamma \Vdash_\Sigma \Delta \vdash a : \tau}\,(atm)^\tau \qquad \frac{\Gamma \bowtie (b : \sigma) \Vdash_\Sigma \Delta \cup \Delta^\# \vdash (a\ b) \cdot t : (a\ b) \cdot \tau}{\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, t : [a : \sigma]\, \tau}\,(abs)^\tau$$

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)}{\Gamma \Vdash_\Sigma \Delta \vdash () : ()}\,(tpl : 0)^\tau \qquad \frac{\Gamma \Vdash_\Sigma \Delta \vdash t_1 : \tau_1 \ \ldots\ \Gamma \Vdash_\Sigma \Delta \vdash t_n : \tau_n}{\Gamma \Vdash_\Sigma \Delta \vdash (t_1,\ \ldots,\ t_n) : (\tau_1 \times \ldots \times \tau_n)}\,(tpl : n)^\tau$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t : \sigma\,\theta \quad \Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}}{\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t : \tau}\,(app)^\tau\ \big(\Theta \Vdash \nabla \vdash \mathsf{f}\, s : \langle \sigma \hookrightarrow \rho \rangle \in \Sigma^{\prime\prime},\ \mathsf{sol}(\theta)\big)$$

$$\frac{\Gamma \bowtie (b : \sigma) \Vdash_\Sigma \Delta \cup \Delta^\# \vdash (a\ b) \cdot \tau : \mathsf{type}}{\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, \tau : \mathsf{type}}\,(abt)^\tau$$

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)}{\Gamma \Vdash_\Sigma \Delta \vdash () : \mathsf{type}}\,(prd : 0)^\tau \qquad \frac{\Gamma \Vdash_\Sigma \Delta \vdash \tau_1 : \mathsf{type}\ \ldots\ \Gamma \Vdash_\Sigma \Delta \vdash \tau_n : \mathsf{type}}{\Gamma \Vdash_\Sigma \Delta \vdash (\tau_1 \times \ldots \times \tau_n) : \mathsf{type}}\,(prd : n)^\tau$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t\colon \sigma\,\theta}{\Gamma \Vdash_\Sigma \Delta \vdash \mathtt{C}\,t\colon \mathsf{type}}\,(cns)^\tau\ \big(\Theta \Vdash \nabla \vdash \mathtt{C}\,s\colon \langle \sigma \hookrightarrow \mathsf{type}\rangle \in \Sigma^{\prime\prime},\ \mathsf{sol}(\theta)\big)$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau \quad \Delta \vdash \sigma \approx_\alpha \tau}{\Gamma \Vdash_\Sigma \Delta \vdash t\colon \sigma}\,(\alpha)^\tau$$

Note that in the rules, $(atm)^\tau$, $(var)^\tau$, $(tpl : 0)^\tau$ and $(prd : 0)^\tau$, the validity of the environment is needed as a premise because the environment, Γ, is not assumed to be valid and also, that in the rule, $(var)^\tau$, the suspension, $\vartheta\big|\pi$, must be applied to the type, Γ_X.

4 Meta-theory

The type system presented works uniformly on α-equivalence classes of terms and types (see Theorem 18 below.) In order to prove this property, some standard properties (weakening, strengthening and validity of typing environments) are first stated. The section is concluded with the substitution theorems.

▶ **Theorem 14** (Type Strengthening). *1. If* $\Gamma \Vdash_\Sigma \Delta\cup\Delta' \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta\cup\Delta' \vdash \tau\colon \mathsf{type}$*) and* Δ' *mentions atoms that are not in* $\mathsf{dom}(\Gamma)$ *then* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*).*

2. If $\Gamma \bowtie (a_1\colon \sigma_1)\dots(a_n\colon \sigma_n) \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \bowtie (a_1\colon \sigma_1)\dots(a_n\colon \sigma_n) \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*) and* $\Delta \vdash a_i \# t,\ \tau$ $(1 \le i \le n)$ *then* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*).*

Proof. Both parts are proved by induction on the type derivation. ◀

▶ **Theorem 15** (Type Weakening).

1. *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*), then* $\Gamma' \Vdash_\Sigma \Delta' \vdash t\colon \tau$ *(resp.* $\Gamma' \Vdash_\Sigma \Delta' \vdash \tau\colon \mathsf{type}$*), for any* Γ', Δ' *such that* $\Gamma' = \Gamma \bowtie \Gamma_1$, $\Delta' \supseteq \Delta$ *and* $\mathsf{validE}(\Gamma',\ \Sigma,\ \Delta')$.
2. *If* $\Gamma \bowtie (a_1\colon \sigma_1) \bowtie (a_2\colon \sigma_2) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \bowtie (a_1\colon \sigma_1) \bowtie (a_2\colon \sigma_2) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*), then* $\Gamma \bowtie (a_2\colon \sigma_2) \bowtie (a_1\colon \sigma_1) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \bowtie (a_2\colon \sigma_2) \bowtie (a_1\colon \sigma_1) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$ *provided* $\Delta \vdash a_1 \# \sigma_2$.

Proof. By simultaneous induction on the type derivation. ◀

▶ **Theorem 16** (Validity of Typing Environments). *For any given* $\mathsf{validD}(\Sigma)$*:*
If $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *then* $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$ *and* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$.
If $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$ *then* $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$.

Proof. Both parts are proved by induction on the type derivation, using weakening (in the case where the last rule used was $(atm)^\tau$) and strengthening in the cases where the last rule used is an abstraction rule (for types or terms). ◀

▶ **Lemma 17.** *If* $\Delta \vdash \mathsf{ds}(\vartheta_1\big|\pi_1,\ \vartheta_2\big|\pi_2) \# X$ *then* $\Delta \vdash \mathsf{ds}(\vartheta_1\big|\pi_1,\ \vartheta_2\big|\pi_2) \# \Gamma_X$.
For any M *(pseudo-term or pseudo-type), if* $\Delta \vdash \mathsf{ds}(\vartheta_1\big|\pi_1,\ \vartheta_2\big|\pi_2) \# M$, *then* $\Delta \vdash (\pi_1 \cdot M)\,\vartheta_1 \approx_\alpha (\pi_2 \cdot M)\,\vartheta_2$.

Proof. The first part is a consequence of the definition of valid environment. The second part is proved by induction on the definition of $\approx_\alpha$. ◀

▶ **Theorem 18** (Unicity of Types). *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau_1$ *and* $\Gamma \Vdash_\Sigma \Delta \vdash t'\colon \tau_2$, *where* $\Delta \vdash t \approx_\alpha t'$, *then* $\Delta \vdash \tau_1 \approx_\alpha \tau_2$.

Proof. 1. If $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau_1$ and $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau_2$, then $\Delta \vdash \tau_1 \approx_\alpha \tau_2$.

2. If $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$, $\Delta \vdash t \approx_\alpha t'$, $\Delta \vdash \Gamma \approx_\alpha \Gamma'$ then $\Gamma' \Vdash_\Sigma \Delta \vdash t' : \tau$, where $\Delta \vdash \Gamma \approx_\alpha \Gamma'$ indicates that for each mapping $(a : \sigma)$ (resp. $(X : \sigma)$) in Γ, Γ' contains a mapping $(a : \sigma')$ (resp. $(X : \sigma')$) such that $\Delta \vdash \sigma \approx_\alpha \sigma'$. Similarly for types: if $\Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}$, $\Delta \vdash \tau \approx_\alpha \tau'$, $\Delta \vdash \Gamma \approx_\alpha \Gamma'$ then $\Gamma' \Vdash_\Sigma \Delta \vdash \tau' : \mathsf{type}$.

The first part is proved by induction on the type derivation for $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau_1$. The only case that is not straightforward is the case in which the derivation concludes using the rule $(app)^\tau$. In this case, the unicity of the type is a consequence of the fact that declarations do not overlap (that is, there is a unique declaration that matches the term considered) and matching problems have unique most general solutions under the assumptions for declarations.

The second part is proved by induction on the type derivation. In the case of a variable, Lemma 17 is needed to derive $\mathsf{typV}(\Gamma',\ \Sigma,\ \Delta,\ \vartheta' \big| \pi',\ X)$. ◀

▶ **Theorem 19** (Preservation of Types by Atom Substitution). *If* $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}$*) and* π, ϑ, Γ', Δ' *are such that:*

- $\forall X \in \mathsf{dom}(\Gamma)$, $\Gamma' \Vdash_\Sigma \Delta' \vdash \vartheta \big| \pi \cdot X : (\pi \cdot \Gamma_X)\,\vartheta$
- $\forall a \in \mathsf{dom}(\Gamma)$, *either* $\Delta \vdash a \# t$ *or* $\Gamma' \Vdash_\Sigma \Delta' \vdash \vartheta(\pi(a)) : (\pi \cdot \Gamma_a)\,\vartheta$

then $\Gamma' \Vdash_\Sigma \Delta' \vdash (\pi \cdot t)\,\vartheta : (\pi \cdot \tau)\,\vartheta$ *(resp.* $\Gamma' \Vdash_\Sigma \Delta' \vdash (\pi \cdot \tau)\,\vartheta : \mathsf{type}$*)*

Proof. By induction on the type derivation. In the case where the last rule applied is $(app)^\tau$ or $(cns)^\tau$, one relies on the fact that declarations are closed (that is, there are no unabstracted atoms.) The cases of abstraction rules (for terms or types) follow by induction, since atom substitutions are capture-avoiding. ◀

▶ **Theorem 20** (Preservation of Types by Variable Substitution). *If* $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}$*) and* θ *is a variable substitution such that:*

- $\forall X \in \mathsf{dom}(\Gamma)$, $\Gamma' \Vdash_\Sigma \Delta' \vdash \theta(X) : \Gamma_X\,\theta$ *and* $\Delta' \vdash \Delta\,\theta$
- $\forall a \in \mathsf{dom}(\Gamma)$, *either* $\Delta \vdash a \# t$ *or* $\Gamma' \Vdash_\Sigma \Delta' \vdash a : \Gamma_a\,\theta$

then $\Gamma' \Vdash_\Sigma \Delta' \vdash t\,\theta : \tau\,\theta$ *(resp.* $\Gamma' \Vdash_\Sigma \Delta' \vdash \tau\,\theta : \mathsf{type}$*)*

Proof. By induction on the type derivation. In the case where the term is of the form $\theta \big| \pi \cdot X$ and the last rule applied is $(var)^\tau$ Theorem 19 for atom substitutions is used. ◀

5 Extended Example

First-order logic is a proto-typical system with binding. We consider the language of Arithmetic, and start the specification by defining type-formers for natural numbers, propositions and proofs, and term-formers to build numbers, $\mathsf{0}$ (zero) and s (successor) and propositions, bot ($\bot$), imp ($\Rightarrow$) and all ($\forall$).

$$\frac{}{- \Vdash \varnothing \vdash \mathtt{N} : \mathsf{type}}$$

$$\frac{}{- \Vdash \varnothing \vdash \mathtt{Prop} : \mathsf{type}}$$

$$\frac{P : \mathtt{Prop} \Vdash \varnothing \vdash \mathtt{Proof}\ P : \mathsf{type}}{}$$

$$\frac{}{- \Vdash \varnothing \vdash \mathsf{0} : \mathtt{N}}$$

$$\frac{X : \mathtt{N} \Vdash \varnothing \vdash \mathsf{s}\,X : \mathtt{N}}{}$$

$$\frac{}{- \Vdash \varnothing \vdash \mathsf{bot} : \mathtt{Prop}}$$

$$\frac{P_1 : \mathtt{Prop},\ P_2 : \mathtt{Prop} \Vdash \varnothing \vdash \mathsf{imp}\,(P_1,\ P_2) : \mathtt{Prop}}{}$$

$$\frac{P : \mathtt{Prop} \Vdash \varnothing \vdash \mathsf{all}\,[x : \mathtt{N}]\,P : \mathtt{Prop}}{}$$

Now, define declarations for the predicates used to build proofs; the introduction and elimination of imp and all, $\mathsf{imp_i}$, $\mathsf{imp_e}$, $\mathsf{all_i}$ and $\mathsf{all_e}$ and the elimination of bot, $\mathsf{bot_e}$.

$$\begin{array}{r} P_1 \colon \mathtt{Prop},\ P_2 \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ P_2 \\ \Vdash x \# P_1,\ x \# P_2 \vdash \mathsf{imp_i}\,[x \colon \mathtt{Proof}\ P_1]\,Q \colon \mathtt{Proof}\ \mathsf{imp}\,(P_1,\ P_2) \end{array}$$

$$\begin{array}{r} P_1 \colon \mathtt{Prop},\ P_2 \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ \mathsf{imp}\,(P_1,\ P_2),\ Q_1 \colon \mathtt{Proof}\ P_1 \\ \Vdash \varnothing \vdash \mathsf{imp_e}\,(Q,\ Q_1) \colon \mathtt{Proof}\ P_2 \end{array}$$

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ \mathsf{bot} \\ \Vdash \varnothing \vdash \mathsf{bot_e}\,(P,\ Q) \colon \mathtt{Proof}\ P \end{array}$$

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q \colon [x \colon \mathtt{N}]\,\mathtt{Proof}\ P \\ \Vdash \varnothing \vdash \mathsf{all_i}\,Q \colon \mathtt{Proof}\ \mathsf{all}\,[x \colon \mathtt{N}]\,P \end{array}$$

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ \mathsf{all}\,[x \colon \mathtt{N}]\,P,\ N \colon \mathtt{N} \\ \Vdash x \# N \vdash \mathsf{all_e}\,(Q,\ N) \colon \mathtt{Proof}\ [x \mapsto N] \cdot P \end{array}$$

Note that in the declaration for $\mathsf{bot_e}$ one must use variables, P and Q, as arguments because of the restriction that all variables in the type of $\mathsf{bot_e}$ should occur in its arguments or in its types. Notice also that in the declaration for $\mathsf{all_i}$ above, the variable, Q, of type $[x \colon \mathtt{N}]\,\mathtt{Proof}\ P$, is used, in other words, n is not unabstracted, as expected. The full declaration, which can be inferred is $P \colon \mathtt{Prop},\ Q \colon [x \colon \mathtt{N}]\,\mathtt{Proof}\ P \Vdash \varnothing \vdash \mathsf{all_i}\,Q \colon \langle [x \colon \mathtt{N}]\,\mathtt{Proof}\ P \hookrightarrow \mathtt{Proof}\ \mathsf{all}\,([x \colon \mathtt{N}]\,P)\rangle$. When this declaration is used to type terms built with $\mathsf{all_i}$, pattern matching is used to obtain the values of P and Q.

An induction principle over the natural numbers could be defined as follows.

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q_0 \colon \mathtt{Proof}\ [x \mapsto 0] \cdot P,\ Q_1 \colon [n \colon \mathtt{N}]\,[p \colon \mathtt{Proof}\ [x \mapsto n] \cdot P]\,\mathtt{Proof}\ [x \mapsto \mathtt{s}\,n] \cdot P \\ \Vdash n \# P \vdash \mathsf{ind}\,(Q_0,\ Q_1) \colon \mathtt{Proof}\ \mathsf{all}\,[x \colon \mathtt{N}]\,P \end{array}$$

An encoding of a system in a logical framework is adequate if it faithfully reflects the properties of the encoded system. For instance, in the case of an encoding of first-order logic, one needs to show that the terms used in the dependent type system represent first-order terms, that formulae and proofs correspond to their standardly acknowledged notions, and that only provable propositions have a proof in the system. The goal is to prove that there is a bijection between proofs in first-order logic and the corresponding terms in this system. A formal specification of first-order logic terms, formulae and proofs is given and then it is shown that these are encoded by terms of the correct type and that encoded terms represent only well-formed terms, formulae and proofs. The theorems and proofs presented here follow closely those given in [24] to prove the adequacy of LF but are much simpler due to the fact that here lambda calculus β-reduction is not involved and therefore all terms are of canonical form.

The following grammars define the syntax of the sets of terms (Trm) and formulae (Frm) of first-order logic.

▶ **Definition 21** (First-order Logic Terms and Formulae)**.**

$$\mathsf{T},\ \mathsf{T}' ::= 0 \mid s(\mathsf{T}) \mid x \qquad \mathsf{F},\ \mathsf{F}' ::= \bot \mid \mathsf{F} \to \mathsf{F}' \mid \forall x.\,\mathsf{F}$$

Let $\mathsf{fvars}(\mathsf{T})$ and $\mathsf{fvars}(\mathsf{F})$ denote respectively the set of free variables in the term, T, and the formula, F. Extend this notation element-wise to sets of formulae.

A translation function $[\![\mathsf{T}]\!]$ is defined by induction on T, from Trm to terms in the system using the term-formers, $\mathsf{0}$ and s, for which, see Section 5. Note that the free variables of T are encoded as unabstracted atoms in $[\![\mathsf{T}]\!]$. A corresponding translation function is defined over the elements of Frm.

$$\begin{array}{llll} [\![0]\!] & = \mathsf{0} & [\![\bot]\!] & = \mathsf{bot} \\ [\![s(\mathsf{T})]\!] & = \mathsf{s}\,[\![\mathsf{T}]\!] & [\![\mathsf{F}_1 \to \mathsf{F}_2]\!] & = \mathsf{imp}\,([\![\mathsf{F}_1]\!],\ [\![\mathsf{F}_2]\!]) \\ [\![x]\!] & = x & [\![\forall x.\,\mathsf{F}]\!] & = \mathsf{all}\,[x\colon \mathtt{N}]\,[\![\mathsf{F}]\!] \end{array}$$

Using the declarations given above, it can now be proved both that translated terms and formulae are typeable terms of the system and that typeable encoded terms correspond exactly to well-formed first-order logic terms and formulae.

▶ **Theorem 22.** *For any term,* $\mathsf{T} \in \mathsf{Trm}$*, such that* $\mathsf{fvars}(\mathsf{T}) = x_1,\ \ldots,\ x_n$ *and freshness context,* Δ*, if* $x_1\colon \mathtt{N},\ \ldots,\ x_n\colon \mathtt{N} \Vdash_\Sigma \Delta \vdash [\![\mathsf{T}]\!]\colon \mathtt{N}$*.*

Similarly for any $\mathsf{F} \in \mathsf{Frm}$*, such that* $\mathsf{fvars}(\mathsf{F}) = x_1,\ \ldots,\ x_n$*, and freshness context,* Δ*,* $x_1\colon \mathtt{N},\ \ldots,\ x_n\colon \mathtt{N} \Vdash_\Sigma \Delta \vdash [\![\mathsf{F}]\!]\colon \mathtt{Prop}$*,*

Proof. By structural induction on the syntax of elements of Trm and Frm. ◀

▶ **Theorem 23.** *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \mathtt{N}$ *is a derivable typing judgement and the environment,* Γ*, contains only type associations for unabstracted atoms of the form* $(a\colon \mathtt{N})$*, then* $t \equiv [\![\mathsf{T}]\!]$ *for some term* $\mathsf{T} \in \mathsf{Trm}$*.*

Similarly, if $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \mathtt{Prop}$ *is a derivable typing judgement and the environment,* Γ*, contains only type associations for unabstracted atoms of the form* $(a\colon \mathtt{N})$*, then* $t \equiv [\![\mathsf{F}]\!]$ *for some formula* $\mathsf{F} \in \mathsf{Frm}$*.*

Proof. By induction on typing judgement derivations. The only applicable cases are when the first step of the derivation is by one of the rules, (atm), (app) or (α). ◀

In order to show the adequacy of the encoding of proofs of first-order formulae, first, a natural deduction presentation is given for first-order logic, inspired by the one used in [24] to prove the adequacy of the encoding in LF.

Let judgements have the form $\mathcal{E} \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}$, indicating that there is a proof P of the formula F, using the list of hypotheses, $\mathcal{E}$, and the set of free variables, $\mathcal{V}$, where $(\mathsf{fvars}(\mathcal{E}) \subseteq \mathcal{V})$. The introduction rules for implication and universal quantification are shown below.

$$\frac{\mathcal{E},\ (v_{\mathsf{F}_1}\colon \mathsf{F}_1) \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}_2}{\mathcal{E} \vdash_\mathcal{V} \to_i (\mathsf{P} \setminus v_{\mathsf{F}_1})\colon \mathsf{F}_1 \to \mathsf{F}_2} \qquad \frac{\mathcal{E} \vdash_{\mathcal{V} \cup \{x\}} \mathsf{P}\colon \mathsf{F}}{\mathcal{E} \vdash_\mathcal{V} \forall_i(x.\,\mathsf{P})\colon \forall x.\,\mathsf{F}}$$

Here, v_{F_1} is the variable name of a proof of F_1, the notation $(\mathsf{P} \setminus v_{\mathsf{F}_1})$ denotes the proof P where v_{F_1} is discharged, and in the rule for $\forall_i$, the condition $\mathsf{fvars}(\mathcal{E}) \subseteq \mathcal{V}$ on judgements implies that x is not used in $\mathcal{E}$.

A natural deduction judgement, J, of the form $\mathcal{E} \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}$ is translated to a typing judgement of the system, $[\![\mathsf{J}]\!]$, as follows.

$$[\![\mathcal{E} \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}]\!] = [\![\mathcal{V}]\!] \bowtie [\![\mathcal{E}]\!] \Vdash_\Sigma [\![\mathsf{P}]\!]^{\Delta}_{\mathsf{vars}([\![\mathsf{J}]\!])} \vdash [\![\mathsf{P}]\!]\colon \mathtt{Proof}\ [\![\mathsf{F}]\!]$$

Here, if $\mathcal{V}$ is $\{x_1,\ \ldots,\ x_n\}$ then $[\![\mathcal{V}]\!] = x_1\colon \mathtt{N},\ \ldots,\ x_n\colon \mathtt{N}$ and $[\![\mathcal{E}]\!]$ contains $v_{\mathsf{F}_i}\colon \mathtt{Proof}\ [\![\mathsf{F}_i]\!]$ for each $(v_{\mathsf{F}_i}\colon \mathsf{F}_i)$ in Γ. The translation function from proofs to terms, $[\![\mathsf{P}]\!]$, is defined inductively; two cases are given.

$$[\![\forall_i(x.\,\mathsf{P})]\!] = \mathsf{all}_\mathsf{i}\,[x\colon \mathtt{N}]\,[\![\mathsf{P}]\!] \quad [\![\to_i (\mathsf{P} \setminus v_{\mathsf{F}_1})]\!] = \mathsf{imp}_\mathsf{i}\,[v_{\mathsf{F}_1}\colon \mathtt{Proof}\ [\![\mathsf{F}_1]\!]]\,[\![\mathsf{P}]\!]$$

A second translation function from a proof, P, to a freshness context, parameterised by a set of variables, $\mathcal{X}$, and written $[\![\mathsf{P}]\!]^{\Delta}_{\mathcal{X}}$ is also required; again two cases are given.

$$[\![\forall_i(x.\,\mathsf{P})]\!]^{\Delta}_{\mathcal{X}} = [\![P]\!]^{\Delta}_{\mathcal{X}} \quad [\![\rightarrow_i (\mathsf{P} \setminus v_{\mathsf{F}_1})]\!]^{\Delta}_{\mathcal{X}} = \{v_{\mathsf{F}_1} \# X \mid X \in \mathcal{X}\} \cup [\![P]\!]^{\Delta}_{\mathcal{X}}$$

One can now prove the following property relating natural deduction proofs in first-order logic and their encoding in the system presented here.

▶ **Theorem 24.** *If* $\mathcal{E} \vdash_{\mathcal{V}} \mathsf{P} : \mathsf{F}$ *is a derivable judgement,* J*, of natural deduction then* $[\![\mathcal{V}]\!] \bowtie [\![\mathcal{E}]\!] \Vdash_{\Sigma} [\![\mathsf{P}]\!]^{\Delta}_{\mathsf{vars}([\![\mathsf{J}]\!])} \vdash [\![\mathsf{P}]\!] : \mathtt{Proof}\,[\![\mathsf{F}]\!]$ *is a derivable typing judgement.*

Proof. By induction on the syntax of the proof, P. In the case where P is of the form $\rightarrow_i (\mathsf{P}' \setminus v_{\mathsf{F}_1})$ of these, the use of the rule (α) follows from the fact that $[\![\mathsf{P}]\!]^{\Delta}_{\mathsf{vars}([\![\mathsf{J}]\!])} \vdash v_{\mathsf{F}_1} \# [\![\mathsf{F}_1]\!],\ v_{\mathsf{F}_1} \# [\![\mathsf{F}_2]\!]$. ◀

Finally, in order to complete the adequacy proof, it is shown that only provable first-order formulae are encoded by terms of type $\mathtt{Proof}\,t$.

▶ **Theorem 25.** *If* $\Gamma \Vdash_{\Sigma} \Delta \vdash s : \mathtt{Proof}\,t$ *where the environment,* Γ*, contains only type associations for atoms either of the form* $(a : \mathtt{N})$ *or* $(v_{\mathsf{F}_1} : \mathtt{Proof}\,t_i)$ *where* $t_i \equiv [\![\mathsf{F}_i]\!]$ *is an encoding of some formula* F_i*, then* $s \equiv [\![\mathsf{P}]\!]$ *where* P *is a proof by natural deduction of some formula* F *such that* $t \equiv [\![\mathsf{F}]\!]$.

Proof. By induction on typing judgement derivations. In the derivation for $\mathsf{imp_i}$, $\Delta \vdash v_{\mathsf{F}_1} \# t_1,\ v_{\mathsf{F}_1} \# t_2,\ v_{\mathsf{F}_1} \# \mathsf{imp_i}\,[v_{\mathsf{F}_1} : \mathtt{Proof}\,t_1]\,s'$ and so v_{F_1} cannot be a free variable of P or F. ◀

6 Related Work

Nominal sets have been used to give semantics to systems based on nominal abstract syntax (see, for instance, [31, 18, 7, 11]) and proof theories for nominal logic have also been considered [19, 6]). Atom substitutions and their properties have been defined as systems of equational rules in [17, 21, 15]. Nominal equational theories have been investigated in [22, 10, 15] amongst others, and type systems for nominal terms and equational theories, using rank-1 polymorphic types, are defined in [14, 13, 12]. However, although proofs play an important role in all of these works, none of these systems deal explicitly with proof terms, and do not yield directly a nominal type theory. Nominal type theory has been investigated by Schöpp and Stark [34], using categorical models of nominal logic. The nominal dependent type theories developed following this approach are very expressive, but it is not clear whether their computational properties make them suitable for use in a logical framework. Nominal type theory as a basis for logical frameworks has been investigated by Cheney [8, 9], as extensions of a typed λ-calculus with names, name-abstraction and concretion operators, and name-abstraction types. A system combining λ-calculus and nominal features is also investigated by Pitts [32] to define a nominal version of Gödel's System T. A key difficulty encountered when following the approach of combining λ-calculus and nominal syntax is the interaction between name abstraction and functional abstraction (see [8] for a detailed discussion.) Westbrook [36] extends the Calculus of Inductive Constructions with a name-abstraction construct in the style of [8].

One of the best known examples of logical frameworks is LF [24], based on a typed λ-calculus with dependent types. The system presented here has similar expressive power, however there is no primitive notion of functional abstraction, instead there are term- and type- constructors in the user-defined signature. Other differences with LF include the

distinction between atoms (which can be abstracted or unabstracted), variables (which cannot be abstracted but can be instantiated, with non-capture-avoiding substitutions), and the use of name swappings (or more generally, permutations), to axiomatise α-equivalence.

Compared with previous approaches to the definition of λ-free logical frameworks [26, 1, 33], abstraction is a first-class ingredient in the syntax presented here and can be used in arguments for term- or type-constructors (as in [26, 1, 33]), or on its own (unlike [26, 1]) although user-defined constructors of abstraction type are not allowed. Also, as in [26, 1, 33], instantiation is a primitive notion in the system; it is used instead of the application operation used in λ-calculus based logical frameworks. However the approach here does not rely on explicit lists of arities and η-long normal forms as in [26, 1]. The triggering of suspended atom substitutions by instantiating variables is similar to the hereditary substitution mechanism of DMBEL [33].

There may exist similarities between the work here and that of 'contextual modal type theory' [28], and the dependent type system of this paper may benefit from a study of the handling of type environments and substitutions therein.

7 Conclusions and Future Work

This paper has presented a dependent type system for nominal terms with atom substitutions. A definition of matching over this syntax have been given together with an algorithm for solving such problems. This algorithm has been implemented but the complexity of problems has not been analysed; this is left for future work. A set of axioms and rules was then defined for determining the typeability of pseudo-terms and pseudo-types in this system in the presence of user-defined declarations for term-formers and type-formers. An extended example for first-order logic was presented and its adequacy proven. The type system itself has not been implemented. In its present form, the inclusion of the rule $(\alpha)^\tau$ means that the inference of derivations is not completely syntax-directed. This property, that derivable typing judgements hold for alpha-equivalent classes of types, may be derivable and if so should help in the development of a type inference algorithm for the system. Further benefits may also be gained from the inclusion of some of the more sophisticated ideas used for other type systems considered in [12]. Although we have not included computation rules in our language, dynamic features, such as reduction in the λ-calculus, or proof normalisation for a logic, may be represented using relations between terms. However, a more direct definition using equality axioms, such as in [26], would be easier to use. An extension of the logical framework to include a user-defined nominal equational theory, specified as a set of equality axioms or rewriting rules, also remains as future work.

References

1 Robin Adams. Lambda-free logical frameworks. *CoRR*, abs/0804.1879, 2008.

2 Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In *Automated Deduction – CADE-17*, 2000.

3 Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*. Elsevier, 2001.

4 Christophe Calvès and Maribel Fernández. Matching and alpha-equivalence check for nominal terms. *Journal of Computer System Sciences*, 76, 2010.

5 Christophe Calvès and Maribel Fernández. The first-order nominal link. In *Logic-Based Program Synthesis and Transformation – 20th International Symposium, LOPSTR 2010,*

Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers, volume 6564 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2011.

6 James Cheney. A simpler proof theory for nominal logic. In *FoSSaCS*, 2005.

7 James Cheney. Completeness and Herbrand theorems for nominal logic. *Journal of Symbolic Logic*, 71, 2006.

8 James Cheney. A simple nominal type theory. *Electronic Notes in Theoretical Computer Science*, 228, 2009.

9 James Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8, 2012.

10 Ranald A. Clouston. *Equational Logic for Names and Binding*. PhD thesis, University of Cambridge, 2010.

11 Roy L. Crole and Frank Nebel. Nominal lambda calculus: An internal language for fm-cartesian closed categories. *Electr. Notes Theor. Comput. Sci.*, 298:93–117, 2013.

12 Elliot Fairweather. *Type Systems for Nominal Terms*. PhD thesis, King's College London, 2014.

13 Elliot Fairweather, Maribel Fernández, and Murdoch J. Gabbay. Principal types for nominal theories. In *Proceedings of the 18th International Symposium on Fundamentals of Computation Theory (FCT 2011)*, 2011.

14 Maribel Fernández and Murdoch J. Gabbay. Curry-style types for nominal terms. In *Types for Proofs and Programs (TYPES'06)*. Springer, 2007.

15 Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205, 2007.

16 Maribel Fernández and Murdoch J. Gabbay. Closed nominal rewriting and efficiently computable nominal algebra equality. In *Proceedings of the 5th International Workshop on Logical Frameworks and Meta-Languages (LFMTP 2010)*, 2010.

17 Murdoch J. Gabbay. A study of substitution, using nominal techniques and Fraenkel-Mostowski sets. *Theoretical Computer Science*, 410, 2009.

18 Murdoch J. Gabbay. Two-level nominal sets and semantic nominal terms: an extension of nominal set theory for handling meta-variables. *Mathematical Structures in Computer Science*, 21, 2011.

19 Murdoch J. Gabbay and James Cheney. A sequent calculus for nominal logic. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, 2004.

20 Murdoch J. Gabbay and Aad Mathijssen. Nominal algebra. In *18th Nordic Workshop on Programming Theory*, 2006.

21 Murdoch J. Gabbay and Aad Mathijssen. Capture-avoiding substitution as a nominal algebra. *Formal Aspects of Computing*, 20, 2008.

22 Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: Equational logic with names and binding. *Journal of Logic and Computation*, 19, 2009.

23 Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS 1999)*, pages 214–224. IEEE Computer Society Press, July 1999.

24 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science (LICS 1987)*. IEEE Computer Society Press, 1987.

25 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, 2010.

26 Zhaohui Luo. PAL$^+$: a lambda-free logical framework. *Journal of Functional Programming*, 13, 2003.

27 Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs, (TYPES'93)*, 1994.

28 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9, 2008.

29 Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

30 Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16*, 1999.

31 Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software (STACS 2001)*, 2001.

32 Andrew M. Pitts. Nominal system T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*, 2010.

33 Gordon Plotkin. An algebraic framework for logics and type theories, 2006. Talk given at LFMTP'06.

34 Ulrich Schöpp and Ian Stark. A Dependent Type Theory with Names and Binding. In *CSL*, 2004.

35 Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323, 2004.

36 Edwin Westbrook. *Higher-order Encodings with Constructors*. PhD thesis, Washington University in St. Louis, 2008.

Realizability Toposes from Specifications*

Jonas Frey

Department of Computer Science
University of Copenhagen, Denmark
jofr@di.ku.dk

Abstract

We investigate a framework of Krivine realizability with I/O effects, and present a method of associating realizability models to *specifications* on the I/O behavior of processes, by using adequate interpretations of the central concepts of *pole* and *proof-like term*. This method does in particular allow to associate realizability models to computable functions.

Following recent work of Streicher and others we show how these models give rise to *triposes* and *toposes*.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Krivine machine, classical realizability, realizability topos, bisimulation

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.196

1 Introduction

Krivine realizability with side effects has been introduced by Miquel in [14]. In this article we demonstrate how an instance of Miquel's framework including I/O instructions allows to associate *realizability toposes* to *specifications*, i.e. sets of requirements imposed on the I/O behavior of programs. Since the requirement to compute a specific function f can be viewed as a specification, we do in particular obtain a way to *associate toposes to computable functions*.

These toposes are different from traditional 'Kleene' realizability toposes such as the *effective topos* [6] in that we associate toposes to *individual* computable functions, whereas the effective topos incorporates *all* recursive functions on equal footing. Another difference to the toposes based on Kleene realizability is that the internal logic of the latter is *constructive*, whereas the present approach is based on Krivine's realizability interpretation [10], which validates classical logic.

To represent specifications we make use of the fact that Krivine's realizability interpretation is parametric over a set of processes called the *pole*. The central observation (Lemma 27 and Theorem 28) is that non-trivial specifications on program behavior give rise to poles leading to consistent (i.e. non-degenerate) interpretations.

To give a categorical account of Krivine realizability we follow recent work of Streicher [18] and others [17, 20, 2], which demonstrates how Krivine realizability models give rise to *triposes*. Toposes are then obtained via the tripos-to-topos construction [7].

Our basic formalism is an extension of the Krivine machine (2) that gives an operational semantics to I/O instructions for single bits. We give two formulations of the operational semantics – one (3) in terms of a transition relation on processes including a *state* (which is adequate for reasoning about function computation), and one (4) in terms of a labeled

* This work is supported by the Danish Council for Independent Research Sapere Aude grant "Complexity through Logic and Algebra" (COLA).

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 196–210

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

transition system admitting to reason about program equivalence in terms of bisimulation. The two operational semantics are related by Corollary 7, which we use to prove a Turing completeness result in Theorem 12.

1.1 Related work

The idea of adding instructions with new evaluation rules to the machine plays a central role in Krivine's writings, as a means to realize non-logical axioms. Citing from [11]:

> *"Indeed, when we realize usual axioms of mathematics, we need to introduce, one after the other, the very standard tools in system programming: for the law of Peirce, these are continuations (particularly useful for exceptions); for the axiom of dependent choice, these are the clock and the process numbering; for the ultrafilter axiom and the well ordering of* $\mathbb{R}$*, these are no less than I/O instructions on a global memory, in other words assignment."*

Although features like exceptions and memory are often called *effects*, it is arguable whether they should be called *side* effects, since they do not interact with the outside world.

The idea to add instructions for *side* effects which are influenced by – and influence – the outside world,p has already been investigated by Miquel [14, Section 2.2], and our execution relation (3) can be viewed as an instance of his framework.

What sets the present approach apart is that Miquel views the state of the world (represented by a forcing condition) as being *part* of a process and requires poles to be saturated w.r.t. all (including effectful) reductions, whereas for us poles are sets of 'bare' processes without state, which are saturated only w.r.t. reduction free of side-effects.

This difference is crucial in that it enables the construction of poles from specifications.

2 Syntax and machine

In this section we recall Krivine's abstract machine with continuations as described in [10]. We then go on to describe an extension of the syntax by I/O instructions, and describe an operational semantics as a transition relation on triples (p, ι, o) of process, input, and output.

2.1 Krivine's machine

We recall the underlying syntax and machine of Krivine's classical realizability from [10]. The syntax consists of three syntactic classes called *terms, stacks*, and *processes*.

$$\begin{array}{lll} \text{Terms:} & t ::= x \mid \lambda x.t \mid t\,t \mid \mathsf{cc} \mid \mathsf{k}_\pi & \\ \text{Stacks:} & \pi ::= \pi_0 \mid t{\cdot}\pi & t \text{ closed}, \pi_0 \in \Pi_0 \\ \text{Processes:} & p ::= t \star \pi & t \text{ closed} \end{array} \tag{1}$$

Thus, the terms are the terms of the λ-calculus, augmented by a constant cc for call/cc, and continuation terms k_π for any stack π. A stack, in turn, is a list of closed terms terminated by an element π_0 of a designated set Π_0 of *stack constants*. A process is a pair $t \star \pi$ of a closed term and a stack. The set of closed terms is denoted by Λ, the set of stacks is Π, and the set of processes is $\Lambda{\star}\Pi$.

Krivine's machine is now defined by a transition relation $\succ$ on processes called *evaluation*.

$$\begin{array}{lrcl} \text{(push)} & tu \star \pi & \succ & t \star u{\cdot}\pi \\ \text{(pop)} & (\lambda x\,.\,t[x]) \star u{\cdot}\pi & \succ & t[u] \star \pi \\ \text{(save)} & \mathsf{cc} \star t{\cdot}\pi & \succ & t \star \mathsf{k}_\pi{\cdot}\pi \\ \text{(restore)} & \mathsf{k}_\pi \star t{\cdot}\rho & \succ & t \star \pi \end{array} \tag{2}$$

The first two rules implement *weak head reduction* of λ-terms, and the third and fourth rule capture and restore continuations.

2.2 The machine with I/O

To incorporate I/O we modify the syntax as follows:

Terms:	$t ::= x \mid \lambda x.t \mid t\,t \mid \mathsf{cc} \mid \mathsf{k}_\pi \mid \mathsf{r} \mid \mathsf{w1} \mid \mathsf{w0} \mid \mathsf{end}$	
Stacks:	$\pi ::= \varepsilon \mid t{\cdot}\pi$	t closed
Processes:	$p ::= t \star \pi \mid \top$	t closed

The grammar for terms is extended by constants $\mathsf{r}, \mathsf{w0}, \mathsf{w1}, \mathsf{end}$ for reading, writing and termination, and in exchange the stack constants are omitted – ε is the empty stack. Finally there is a *process constant* $\top$ also representing termination – the presence of both end and $\top$ will be important in Section 3.

We write Λ_e and Π_e for the sets of terms and stacks of the syntax with I/O, and P for the set of processes. Furthermore, we denote by Λ_p the set of *pure* terms, i.e. terms not containing any of $\mathsf{r}, \mathsf{w0}, \mathsf{w1}, \mathsf{end}$.

The operational semantics of the extended syntax is given in terms of *execution contexts*, which are triples (p, ι, o) of a process p, and a pair $\iota, o \in \{0,1\}^*$ of binary strings representing input and output. On these execution contexts, we define the *execution relation* $\rightsquigarrow$ as follows:

$$
\begin{array}{lrcll}
(\tau) & (\ t \star & \pi,\ \iota, o) & \rightsquigarrow & (u \star \rho,\ \iota,\ o) \quad \text{whenever } t \star \pi \succ u \star \rho \\
(\mathsf{r0}) & (\ \mathsf{r} \star & t{\cdot}u{\cdot}v{\cdot}\pi,\ 0\iota, o) & \rightsquigarrow & (\ t \star \pi,\ \iota,\ o) \\
(\mathsf{r1}) & (\ \mathsf{r} \star & t{\cdot}u{\cdot}v{\cdot}\pi,\ 1\iota, o) & \rightsquigarrow & (u \star \pi,\ \iota,\ o) \\
(\mathsf{r}\varepsilon) & (\ \mathsf{r} \star & t{\cdot}u{\cdot}v{\cdot}\pi,\ \varepsilon, o) & \rightsquigarrow & (v \star \pi, \varepsilon,\ o) \\
(\mathsf{w0}) & (\ \mathsf{w0} \star & t{\cdot}\pi,\ \iota, o) & \rightsquigarrow & (\ t \star \pi,\ \iota, 0o) \\
(\mathsf{w1}) & (\ \mathsf{w1} \star & t{\cdot}\pi,\ \iota, o) & \rightsquigarrow & (\ t \star \pi,\ \iota, 1o) \\
(\mathsf{e}) & (\mathsf{end} \star & \pi,\ \iota, o) & \rightsquigarrow & (\ \top\ ,\ \iota,\ o)
\end{array}
\tag{3}
$$

Thus, if there is neither of $\mathsf{r}, \mathsf{w0}, \mathsf{w1}, \mathsf{end}$ in head position, the process is reduced as in (2) without changing ι and o. If r is in head position, the computation selects one of the first three arguments depending on whether the input starts with a 0, a 1, or is empty. $\mathsf{w0}$ and $\mathsf{w1}$ write out 0 and 1, and end discards the stack and yields $\top$, which represents successful termination.

We observe that the execution relation is *deterministic*, i.e. for every execution context there is at most one transition possible, which is determined by the term in head position, and in case of r also by the input.

2.3 Representing functions

We view the above formalism as a model of computation that explicitly includes reading of input, and writing of output.

Consequently, when thinking about expressivity we are not so much interested in the ability of the machine to transform abstract representations of data like 'Church numerals', but rather in the functions on binary strings that processes can compute by reading their argument from the input, and writing the result to the output.

▶ **Definition 1.** For $n \in \mathbb{N}$, $\mathrm{bin}(n) \in \{0,1\}^*$ is the base 2 representation of n. 0 is represented by the empty string, thus we have e.g. $\mathrm{bin}(0) = \varepsilon$, $\mathrm{bin}(1) = 1$, $\mathrm{bin}(2) = 10$, $\mathrm{bin}(3) = 11$, ...

A process p is said to *implement* a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$, if $(p, \mathrm{bin}(n), \varepsilon) \rightsquigarrow^* (\top, \varepsilon, \mathrm{bin}(f(n)))$ for all $n \in \mathrm{dom}(f)$.

▶ Remark. There is a stronger version of the previous definition which requires $(p, \mathrm{bin}(n), \varepsilon)$ to diverge or block for $n \notin \mathrm{dom}(f)$, and a completeness result like Thm. 12 can be shown for the strengthened definition as well.

We use the weaker version, since we expect the poles $\perp\!\!\!\perp_f$ defined in Section 5.2.1 to be better behaved this way.

2.4 β-reduction

To talk about contraction of single β-redexes which are not necessarily in head position in a process p, we define *contexts* – which are terms/stacks/processes with a single designated hole $[\cdot]$ in term position – by the following grammar:

$$\begin{array}{lrcll} \text{Term contexts:} & t[\cdot] & ::= & [\cdot] \mid \lambda x.t[\cdot] \mid t[\cdot]\,t \mid t\,t[\cdot] \mid \mathsf{k}_{\pi[\cdot]} & \\ \text{Stack contexts:} & \pi[\cdot] & ::= & t{\cdot}\pi[\cdot] \mid t[\cdot]{\cdot}\pi & t,t[\cdot] \text{ closed} \\ \text{Process contexts:} & p[\cdot] & ::= & t[\cdot] \star \pi \mid t \star \pi[\cdot] & \end{array}$$

Contexts are used to talk about substitution that allows capturing of variables – as described in [1, 2.1.18], given a context $t[\cdot]/\pi[\cdot]/p[\cdot]$ and a term u, $t[u]/\pi[u]/p[u]$ is the the result of replacing the hole $[\cdot]$ in $t[\cdot]/\pi[\cdot]/p[\cdot]$ by u, allowing potential free variables in u to be captured. We say that u is *admissible* for $t[\cdot]/\pi[\cdot]/p[\cdot]$, if $t[u]/\pi[u]/p[u]$ is a valid term/stack/process conforming to the closedness condition for terms making up stacks.

Now we can express β-reduction as the action of contracting a single redex: given a redex $(\lambda x\,.\,u)v$ which is admissible for a context $t[\cdot]/\pi[\cdot]/p[\cdot]$, we have

$$t[(\lambda x\,.\,u)v] \to_\beta t[u[v/x]] \qquad \pi[(\lambda x\,.\,u)v] \to_\beta \pi[u[v/x]] \qquad p[(\lambda x\,.\,u)v] \to_\beta p[u[v/x]],$$

and any single β-reduction can uniquely be written this way. β-equivalence $\simeq_\beta$ is the equivalence relation generated by β-reduction.

3 Bisimulation and $\top$-equivalence

To reason efficiently about execution of processes with side effects – in particular to show Turing completeness in Section 4 – we want to show that although the computation model imposes a deterministic reduction strategy, we can perform β-reduction anywhere in a process without changing its I/O behavior.

The natural choice of concept to capture 'equivalence of I/O behavior' is *weak bisimilarity* (see [13, Section 4.2]), and in order to make this applicable to processes we have to reformulate the operational semantics as a *labeled transition system* (LTS).

We use the set $\mathcal{L} = \{\mathsf{r0}, \mathsf{r1}, \mathsf{r}\varepsilon, \mathsf{w0}, \mathsf{w1}, \mathsf{e}\}$ of labels, where $\mathsf{r0}$, $\mathsf{r1}$ represent reading of a 0 or 1, respectively, and $\mathsf{w0}$, $\mathsf{w1}$ represent writing of bits. $\mathsf{r}\varepsilon$ represents the unsuccessful attempt of reading on empty input, and e represents successful termination. The set $Act = \mathcal{L} \cup \{\tau\}$ of *actions* contains the labels as well as the symbol τ representing a 'silent' transition, that is used to represent effect-free evaluation.

The transition system on processes is now given as follows.

$$\begin{array}{lll} (\lambda x\,.\,t[x]) \star t{\cdot}\pi \xrightarrow{\tau} t[u] \star \pi & \mathsf{r} \star t{\cdot}u{\cdot}v{\cdot}\pi \xrightarrow{\mathsf{r0}} t \star \pi & \mathsf{w0} \star t{\cdot}\pi \xrightarrow{\mathsf{w0}} t \star \pi \\ tu \star \pi \xrightarrow{\tau} t \star u{\cdot}\pi & \mathsf{r} \star t{\cdot}u{\cdot}v{\cdot}\pi \xrightarrow{\mathsf{r1}} u \star \pi & \mathsf{w1} \star t{\cdot}\pi \xrightarrow{\mathsf{w1}} t \star \pi \\ \mathsf{cc} \star t{\cdot}\pi \xrightarrow{\tau} t \star \mathsf{k}_\pi{\cdot}\pi & \mathsf{r} \star t{\cdot}u{\cdot}v{\cdot}\pi \xrightarrow{\mathsf{r}\varepsilon} v \star \pi & \mathsf{end} \star \pi \xrightarrow{\mathsf{e}} \top \\ \mathsf{k}_\pi \star t{\cdot}\rho \xrightarrow{\tau} t \star \pi & & \end{array} \tag{4}$$

Observe that the τ-transitions are in correspondence with the transitions of the evaluation relation (2), and the labeled transitions correspond to the remaining transitions of the execution relation (3).

We now recall the definition of *weak bisimulation relation* from [13, Section 4.2].

▶ **Definition 2.**

- For processes p, q we write $p \overset{\tau}{\Rightarrow} q$ for $p \overset{\tau}{\rightarrow}{}^{*} q$, and for $\alpha \neq \tau$ we write $p \overset{\alpha}{\Rightarrow} q$ for $\exists p', q' \,.\, p \overset{\tau}{\Rightarrow} p' \overset{\alpha}{\rightarrow} q' \overset{\tau}{\Rightarrow} q$.
- A *weak bisimulation* on P is a binary relation $R \subseteq \mathsf{P}^2$ such that for all $\alpha \in Act$ and $(p, q) \in R$ we have

$$\begin{aligned} p \overset{\alpha}{\rightarrow} p' \quad &\Rightarrow \quad \exists q' \,.\, q \overset{\alpha}{\Rightarrow} q' \wedge (p', q') \in R \quad \text{and} \\ q \overset{\alpha}{\rightarrow} q' \quad &\Rightarrow \quad \exists p' \,.\, p \overset{\alpha}{\Rightarrow} p' \wedge (p', q') \in R. \end{aligned} \tag{5}$$

- Two processes p, q are called *weakly bisimilar* (written $p \approx q$), if there exists a weak bisimulation relation R with $(p, q) \in R$.

We recall the following important properties of the weak bisimilarity relation $\approx$.

▶ **Lemma 3.** *Weak bisimilarity is itself a weak bisimulation, and furthermore it is an equivalence relation.*

Proof. [13, Proposition 4.2.7] ◀

To show that β-equivalent processes are bisimilar, we have to find a bisimulation relation containing β-equivalence. The following relation does the job.

▶ **Definition 4** (γ-equivalence)**.** γ-equivalence (written $p \simeq_\gamma q$) is the equivalence relation on processes that is generated by β-reduction and τ-transitions.

▶ **Lemma 5.** *γ-equivalence of processes is a weak bisimulation.*

Proof. It is sufficient to verify conditions (5) on the generators of γ-equivalence, i.e. one-step β-reductions and τ-transitions. Therefore we show the following:

1. if $p \overset{\tau}{\rightarrow} q$ and $p \overset{\alpha}{\rightarrow} p'$ then there exists q' with $q \overset{\alpha}{\Rightarrow} q'$ and $p' \simeq_\gamma q'$
2. if $p \overset{\tau}{\rightarrow} q$ and $q \overset{\alpha}{\rightarrow} q'$ then there exists p' with $p \overset{\alpha}{\Rightarrow} p'$ and $p' \simeq_\gamma q'$
3. if $p \rightarrow_\beta q$ and $p \overset{\alpha}{\rightarrow} p'$ then there exists q' with $q \overset{\alpha}{\Rightarrow} q'$ and $p' \simeq_\gamma q'$
4. if $p \rightarrow_\beta q$ and $q \overset{\alpha}{\rightarrow} q'$ then there exists p' with $p \overset{\alpha}{\Rightarrow} p'$ and $p' \simeq_\gamma q'$

In the first case, the fact that the LTS can only branch if r is in head position, and this does not involve τ-transitions, implies that $\alpha = \tau$ and $p' = q$, and we can choose $q' = q$ as well. In the second case we have $p \overset{\alpha}{\Rightarrow} q'$ and thus can choose $p' = q'$.

For cases 3 and 4, which we treat simultaneously, we have to analyze the structure of p and q, which are of the form $r[(\lambda x \,.\, s)t]$ and $r[s[t/x]]$ for some context $r[\cdot]$ (see Section 2.4). The proof proceeds by cases on the structure of $r[\cdot]$.

If $r[\cdot]$ is of either of the forms $(st \star \pi)[\cdot]$[1], $\mathsf{w0} \star \pi[\cdot]$, $\mathsf{w1} \star \pi[\cdot]$, $\mathsf{cc} \star \pi[\cdot]$, $(\mathsf{k}_\pi \star \rho)[\cdot]$, or $\mathsf{end} \star \pi[\cdot]$, then it is immediately evident that p and q can perform the same unique transition (if any), and the results will again be β-equivalent (possibly trivially, since the redex can get deleted in the transition).

If $r[\cdot]$ is of the form $((\lambda y \,.\, u) \star \pi)[\cdot]$ then this is true as well, regardless of whether the hole is in u or in π (here the redex can be duplicated, if the hole is in the first term in π).

[1] The notation is meant to convey that we don't care if the hole is in s, t, or π.

If $r[\cdot]$ is of the form $\mathsf{r} \star \pi[\cdot]$ then several transitions may be possible, but any transition taken by either of p or q can also be taken by the other, and the results will again be β-equivalent.

It remains to consider $r[\cdot]$ of the form $[\cdot]\star\pi$. In this case, $p = (\lambda x\,.\,s)t\star\pi$ and $q = s[t/x]\star\pi$. Here p can perform the transition $p \xrightarrow{\tau} (\lambda x\,.\,s)\star t\cdot\pi$ which can be matched by $q \overset{\tau}{\Rightarrow} q$ where we have $(\lambda x\,.\,s)\star t\cdot\pi \simeq_\gamma p \simeq_\gamma q$. In the other direction we have $p \overset{\alpha}{\Rightarrow} q'$ for every $q \xrightarrow{\alpha} q'$ since $p \overset{\tau}{\Rightarrow} q$. ◀

The following definition and corollary makes the link between the execution relation (3) and the LTS (4).

▶ **Definition 6.** Two execution contexts (p, ι, o), (q, ι', o') are called $\top$-*equivalent* (written $(p, \iota, o) \sim_\top (q, \iota', o')$), if for all $\iota'', o'' \in \{0,1\}^*$ we have

$$(p, \iota, o) \rightsquigarrow^* (\top, \iota'', o'') \quad \text{iff} \quad (q, \iota', o') \rightsquigarrow^* (\top, \iota'', o'').$$

▶ **Corollary 7.**

1. $p \approx q$ *implies* $(p, \iota, o) \sim_\top (q, \iota, o)$ *for all* $\iota, o \in \{0,1\}^*$.
2. $(p, \iota, o) \rightsquigarrow^* (q, \iota', o')$ *implies* $(p, \iota, o) \sim_\top (q, \iota', o')$.
3. $(p, \iota, o) \sim_\top (\top, \iota', o')$ *implies* $(p, \iota, o) \rightsquigarrow^* (\top, \iota', o')$.

Proof. For the first claim we show that

$$p \approx q, \quad (p, \iota, o) \rightsquigarrow^* (\top, \iota', o') \quad \text{implies} \quad (q, \iota, o) \rightsquigarrow^* (\top, \iota', o')$$

by induction on the length of $(p, \iota, o) \rightsquigarrow^* (\top, \iota', o')$. The base case is clear. For the induction step assume that $(p, \iota, o) \rightsquigarrow (p^*, \iota^*, o^*) \rightsquigarrow^* (\top, \iota', o')$. If the initial transition is a (τ) in the execution relation (3), then have $p^* \cong q$, $\iota^* = \iota$ and $o^* = o$, and we can apply the induction hypothesis. If the initial transition corresponds to another clause in (3), then there is a corresponding transition $p \xrightarrow{\alpha} q$ with $\alpha \in \mathcal{L}$ in the LTS (4), and by bisimilarity there exists a q^* with $q \overset{\alpha}{\Rightarrow} q^*$ and $p^* \approx q^*$. Now the induction hypothesis implies $(q^*, \iota^*, o^*) \rightsquigarrow^* (\top, \iota', o')$, and from $q \overset{\alpha}{\Rightarrow} q^*$ we can deduce $(q, \iota, o) \rightsquigarrow^* (q^*, \iota^*, o^*)$ by cases on α.

The second claim follows since $\rightsquigarrow$ is deterministic.

The third claim follows since $(\top, \iota', o')$ can not perform any more transitions. ◀

4 Expressivity

In this section we show that the machine with I/O is Turing complete, i.e. that every computable $f : \mathbb{N} \rightharpoonup \mathbb{N}$ can be implemented in the sense of Def. 1 by a process p.

Roughly speaking, given f, we define a process p that reads the input, transforms it into a Church numeral, applies a term t that computes f on the level of Church numerals, and then writes the result out.

To decompose the task we define terms R and W for reading and writing, with the properties that $(R * \pi, \mathrm{bin}(n), o) \sim_\top (\overline{n} \star \pi, \varepsilon, o)$ ($\overline{n}$ is the n-th Church numeral), and $(W\overline{n} * \pi, \iota, \varepsilon) \sim_\top (\top, \iota, \mathrm{bin}(n))$ for all $n \in \mathbb{N}$.

Now the naive first attempt to combine R and W with the term t computing the function would be something like $W(tR)$, but this would only work if the operational semantics was call by value. The solution is to use Krivine's *storage operators* [9] which where devised precisely to simulate call by value in call by name, and we use a variation of them.

The following definition introduces the terms R and W, after giving some auxiliary definitions.

▶ **Definition 8.**
E, Z, B, C, H, Y, S are λ-terms satisfying

$$\begin{array}{llll} B\,\overline{n} \simeq_\beta \overline{2n} & E\,\overline{(2n)}\,s\,t & \simeq_\beta s & Y\,t \simeq_\beta t\,(Y\,t) \\ C\,\overline{n} \simeq_\beta \overline{2n+1} & E\,\overline{(2n+1)}\,s\,t & \simeq_\beta t & \\ H\,\overline{n} \simeq_\beta \overline{\mathrm{floor}(n/2)} & Z\,\overline{(0)}\,s\,t & \simeq_\beta s & \\ S\,\overline{n} \simeq_\beta \overline{n+1} & Z\,\overline{(n+1)}\,s\,t & \simeq_\beta t & \end{array}$$

for all terms s, t and $n \in \mathbb{N}$, where $\overline{n}$ is the Church numeral $\lambda f x\,.\,f^n x$.[2]
The terms F, R, W are defined as follows:

$$\begin{array}{l} F \;= \lambda h y\,.\,h(S\,y)^3 \\ R \;= YQ\,\overline{0} \quad \text{where} \quad Q = \lambda x n\,.\,\mathsf{r}(x(B\,n))(x(C\,n))n \\ W = YV \quad \text{where} \quad V = \lambda x n\,.\,Z\,n\,\mathsf{end}(E\,n(\mathsf{w0}\,x(H\,n))(\mathsf{w1}\,x(H\,n))) \end{array}$$

The next three lemmas explain the roles of the terms R, F, and W.

▶ **Lemma 9.** *For all $n \in \mathbb{N}$, $\pi \in \Pi$ and $o \in \{0,1\}^*$ we have $(R \star \pi, \mathrm{bin}(n), o) \sim_\top (\overline{n} \star \pi, \varepsilon, o)$.*

Proof. For all $n \in \mathbb{N}$ we have $YQ\,\overline{n} \simeq_\beta Q(YQ)\overline{n} \simeq_\beta \mathsf{r}(YQ\overline{(2n)})(YQ\overline{(2n+1)})\overline{n}$, and thus

$$\begin{array}{l} (YQ\,\overline{n} \star \pi, \varepsilon, o) \;\sim_\top (\overline{n} \star \pi, \varepsilon, o) \\ (YQ\,\overline{n} \star \pi, 0\iota, o) \sim_\top (YQ\,\overline{(2n)} \star \pi, \iota, o) \\ (YQ\,\overline{n} \star \pi, 1\iota, o) \sim_\top (YQ\,\overline{(2n+1)} \star \pi, \iota, o) \end{array}$$

The claim follows by induction on the length of $\mathrm{bin}(n)$, since $\mathrm{bin}(2n) = \mathrm{bin}(n)0$ for $n > 0$, and $\mathrm{bin}(2n+1) = \mathrm{bin}(n)1$. ◀

▶ **Lemma 10.** *For $n \in \mathbb{N}$ and t any closed term, we have $\overline{n}\,F\,t\,\overline{0} \simeq_\beta t\,\overline{n}$.*

Proof. This is because $\overline{n}\,F\,t\overline{0} \simeq_\beta F^n\,t\,\overline{0} \simeq_\beta t\,(S^n\overline{0}) \simeq_\beta t\,\overline{n}$, where the second step can be shown by induction on n. ◀

▶ **Lemma 11.** *For all $n \in \mathbb{N}$, $\pi \in \Pi$ and $\iota \in \{0,1\}^*$ we have $(W\overline{n} \star \pi, \iota, \varepsilon) \sim_\top (\top, \iota, \mathrm{bin}(n))$.*

Proof. We have $W\,\overline{n} \simeq_\beta V\,W\,\overline{n} \simeq_\beta Z\,\overline{n}\,\mathsf{end}(E\,\overline{n}(\mathsf{w0}\,W(H\,\overline{n}))(\mathsf{w1}\,W(H\,\overline{n})))$, and therefore

$$\begin{array}{llll} (W\,\overline{0} \star \pi, \iota, o) & \sim_\top (\mathsf{end} \star \pi, \iota, o) & \sim_\top (\top, \iota, o) & \\ (W\,\overline{(2n)} \star \pi, \iota, o) & \sim_\top (\mathsf{w0}\,W(H\,\overline{(2n)}) \star \pi, \iota, o) & \sim_\top (W(\overline{n}) \star \pi, \iota, 0o) & \text{for } (n > 0) \\ (W\,\overline{(2n+1)} \star \pi, \iota, o) & \sim_\top (\mathsf{w1}\,W(H\,\overline{(2n+1)}) \star \pi, \iota, o) & \sim_\top (W(\overline{n}) \star \pi, \iota, 1o). & \end{array}$$

The claim follows again by induction on the length of $\mathrm{bin}(n)$. ◀

▶ **Theorem 12.** *Every computable function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ can be implemented by a process p.*

Proof. From [5, Thm. 4.23] we know that there exists a term t with $t\,\overline{n} \simeq_\beta \overline{f(n)}$ for $n \in \mathrm{dom}(f)$. The process p is given by $R \star F{\cdot}t{\cdot}\overline{0}{\cdot}F{\cdot}W{\cdot}\overline{0}$. Indeed, for $n \in \mathrm{dom}(f)$ we have

$$\begin{array}{lll} (R \star F{\cdot}t{\cdot}\overline{0}{\cdot}F{\cdot}W{\cdot}\overline{0}, \mathrm{bin}(n), \varepsilon) \sim_\top (\overline{n} \star F{\cdot}t{\cdot}\overline{0}{\cdot}F{\cdot}W{\cdot}\overline{0}, \varepsilon, \varepsilon) & \sim_\top (\overline{n}\,Ft\overline{0} \star F{\cdot}W{\cdot}\overline{0}, \varepsilon, \varepsilon) \\ \sim_\top (t\,\overline{n} \star F{\cdot}W{\cdot}\overline{0}, \varepsilon, \varepsilon) & \sim_\top (\overline{f(n)} \star F{\cdot}W{\cdot}\overline{0}, \varepsilon, \varepsilon) \\ \sim_\top (W\,\overline{f(n)} \star \varepsilon, \varepsilon, \varepsilon) & \sim_\top (\top, \varepsilon, \mathrm{bin}(f(n))) \end{array}$$

and we deduce $(R \star F{\cdot}t{\cdot}\overline{0}{\cdot}F{\cdot}W{\cdot}\overline{0}, \mathrm{bin}(n), \varepsilon) \rightsquigarrow^* (\top, \varepsilon, \mathrm{bin}(f(n)))$ by Corollary 7-3. ◀

---

[2] Such terms exist by elementary λ-calculus, see e.g. [5, Chapters 3,4]. In particular, Y is known as *fixed point operator*.
[3] This is (part of) a *storage operator* for Church numerals [9].

5 Realizability and Triposes

The aim of this section is to describe how the presence of I/O instructions allows to define new realizability models, which we do in the categorical language of triposes and toposes[21].

In Subsection 5.1 we give a categorical reading of Krivine's realizability interpretation as described in [10] and show how it gives rise to triposes. In Subsection 5.2 we show how the definitions can be adapted to the syntax and machine with I/O, and how this allows us to define new realizability models from specifications.

The interpretation of Krivine realizability in terms of triposes is due to Streicher [18], and has further been explored in [2]. However, the presentation here is more straightforward since the constructions and proofs do not rely on *ordered combinatory algebras*, but directly rephrase Krivine's constructions categorically.

5.1 Krivine's classical realizability

Throughout this subsection we work with the syntax (1) *without I/O instructions but with stack constants.*

Krivine's realizability interpretation is always given relative to a set of processes called a 'pole' – the choice of pole determines the interpretation.

▶ **Definition 13.** A *pole* is a set $\perp\!\!\!\perp \subseteq \Lambda\star\Pi$ of processes which is *saturated*, in the sense that $p \in \perp\!\!\!\perp$ and $p' \succ p$ implies $p' \in \perp\!\!\!\perp$.

As Miquel [15] demonstrated, the pole can be seen as playing the role of the parameter R in Friedman's negative translation [3]. In the following we assume that a pole $\perp\!\!\!\perp$ is fixed.

A *truth value* is by definition a set $S \subseteq \Pi$ of stacks. Given a truth value S and a term t, we write $t \Vdash S$ – and say 't realizes S' – if $\forall \pi \in S\,.\, t\star\pi \in \perp\!\!\!\perp$. We write $S^{\perp\!\!\!\perp} = \{t \in \Lambda \mid t \Vdash S\}$ for the set of realizers of Π. So unlike in Kleene realizability the elements of a truth value are not its realizers – they should rather be seen as 'refutations', and indeed larger subsets of Π represent 'falser' truth values[4]; in particular falsity is defined as

$$\bot \quad = \quad \Pi.$$

Given truth values $S, T \subseteq \Pi$, we define the implication $S \Rightarrow T$ as follows.

$$S \Rightarrow T \quad = \quad S^{\perp\!\!\!\perp}\cdot T = \{s\cdot\pi \mid s \Vdash S, \pi \in T\}$$

With these definitions we can formulate the following lemma, which relates refutations of a truth value S with realizers of its negation.

▶ **Lemma 14.** *Given $\pi \in S \subseteq \Pi$, we have $\mathsf{k}_\pi \Vdash S \Rightarrow \bot$.*

Proof. We have to show that $\mathsf{k}_\pi \star t\cdot\rho \in \perp\!\!\!\perp$ for all $t \Vdash S$ and $\rho \in \Pi$. This is because $\mathsf{k}_\pi \star t\cdot\rho \succ t \star \pi$, where $\pi \in S$ and $t \Vdash S$. ◀

A *(semantic) predicate* on a set I is a function $\varphi : I \to P(\Pi)$ from I to truth values. On semantic predicates we define the basic logical operations of falsity, implication, universal

[4] For this reason, Miquel [15, 16] calls the elements of $P(\Pi)$ *falsity values*.

quantification, and reindexing by

$$\begin{array}{rcll} \bot(i) & = & \Pi & \textit{(falsity)} \\ (\varphi \Rightarrow \psi)(i) & = & \varphi(i) \Rightarrow \psi(i) = \varphi(i)^{\perp\!\!\!\perp} \cdot \psi(i) & \textit{(implication)} \\ \forall_f(\theta)(i) & = & \bigcup_{f(j)=i} \theta(j) & \textit{(universal quantification)} \\ f^*\varphi & = & \varphi \circ f & \textit{(reindexing)} \end{array} \tag{6}$$

for $\varphi, \psi : I \to P(\Pi)$, $\theta : J \to P(\Pi)$ and $f : J \to I$. Thus, for any function $f : J \to I$, the function $\forall_f$ (called 'universal quantification along f') maps predicates on J to predicates on I[5], and the function f^* (called 'reindexing along f') maps predicates on I to predicate on J. We write $\forall_I$ for universal quantification along the terminal projection $I \to 1$.

Next, we come to the concept of 'truth/validity' of the interpretation. We can not simply call a truth value 'true' if it has a realizer – this would lead to inconsistency as soon as the pole $\perp\!\!\!\perp$ is nonempty, since $\mathsf{k}_\pi t \Vdash \bot$ for any process $t \star \pi \in \perp\!\!\!\perp$. The solution is to single out a set PL of 'well-behaved' realizers called 'proof-like terms'. We recall the definition from [10].

▶ **Definition 15.** The set $\mathsf{PL} \subseteq \Lambda$ of *proof-like terms* is the set of terms t that do not contain any continuations k_π.

As Krivine [10, pg. 2] points out, t is a proof-like term if and only if it does not contain any stack constant $\pi_0 \in \Pi_0$ (since continuation terms k_π necessarily contain a stack constant at the end of π, and conversely stacks can only occur as continuations in a term).

Proof-like terms give us a concept of logical validity – a truth value S is called *valid*, if there exists a proof-like term t with $t \Vdash S$.

With this notion, we are ready to define the centerpiece of the realizability model, which is the *entailment relation* on predicates.

▶ **Definition 16.** For any set I and integer n, the $(n+1)$-ary entailment relation $(\vdash_I)$ on predicates on I is defined by

$$\varphi_1 \dots \varphi_n \vdash_I \psi \quad \text{if and only if} \quad \exists t \in \mathsf{PL} \,.\, t \Vdash \forall_I(\varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \psi).$$

If the right hand side proposition holds, we call t a *realizer* of $\varphi_1 \dots \varphi_n \vdash_I \psi$.

Thus, $\varphi_1 \dots \varphi_n \vdash_I \psi$ means that the truth value $\forall_I(\varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \psi)$ is valid. More explicitly this can be written out as

$$\exists t \in \mathsf{PL}\ \forall i \in I, u_1 \in \varphi_1(i)^{\perp\!\!\!\perp}, \dots, u_n \in \varphi_n(i)^{\perp\!\!\!\perp}, \pi \in \psi(i) \,.\, t \star u_1 \cdot \dots \cdot u_n \cdot \pi \in \perp\!\!\!\perp.$$

With the aim to show that the semantic predicates form a tripos in Theorem 22, we now prove that the entailment ordering models the logical rules in Table (1). The first eight rules form a standard natural deduction system for (the $\bot, \Rightarrow$ fragment of) classical propositional logic, but for universal quantification we give categorically inspired rules that bring us quicker to where we want, and in particular avoid having to deal with variables.

▶ **Lemma 17.** *The rules displayed in Table 1 are admissible for the entailment relation, in the sense that if the hypotheses hold then so does the conclusion.*

[5] The usual $\forall x : A$ from predicate logic corresponds to taking f to be a projection map $\pi_1 : \Gamma \times A \to \Gamma$, see e.g. [8, Chapter 4].

Table 1 Admissible rules for the entailment relation.

$$\frac{}{\varphi \vdash_I \varphi}\ (\mathrm{Ax}) \qquad \frac{\Gamma \vdash_I \bot}{\Gamma \vdash_I \psi}\ (\bot\mathrm{E})$$

$$\frac{\Gamma, \varphi \vdash_I \psi}{\Gamma \vdash_I \varphi \Rightarrow \psi}\ (\Rightarrow\mathrm{I}) \qquad \frac{\Gamma \vdash_I \psi \qquad \Delta \vdash_I \psi \Rightarrow \theta}{\Gamma, \Delta \vdash_I \theta}\ (\Rightarrow\mathrm{E})$$

$$\frac{\Gamma \vdash_I \psi}{\sigma(\Gamma) \vdash_I \psi}\ (\mathrm{S}) \qquad \frac{}{\Delta \mid \Gamma \vdash_I ((\psi \Rightarrow \bot) \Rightarrow \psi) \Rightarrow \psi}\ (\mathrm{PeL})$$

$$\frac{\Gamma \vdash_I \psi}{A, \Gamma \vdash_I \psi}\ (\mathrm{W}) \qquad \frac{A, A, \Gamma \vdash_I \psi}{A, \Gamma \vdash_I \psi}\ (\mathrm{C})$$

$$\frac{f^*\Gamma \vdash_J \xi}{\Gamma \vdash_I \forall_f \xi}\ (\forall\mathrm{I}) \qquad \frac{\Gamma \vdash_I \forall_f \xi}{f^*\Gamma \vdash_J \xi}\ (\forall\mathrm{E})$$

φ, ψ, θ are predicates on I, i.e. functions $I \to P(\Pi)$, and $\Gamma \equiv \varphi_1 \dots \varphi_n$ and $\Delta \equiv \psi_1 \dots \psi_m$ are lists of such predicates. ξ is a predicate on J, and $f : J \to I$ is a function. σ is a permutation of $\{1, \dots, n\}$. $f^*\Gamma$ is an abbreviation for $f^*\varphi_1 \dots f^*\varphi_n$, and $\sigma(\Gamma)$ is an abbreviation for $\varphi_{\sigma(1)} \dots \varphi_{\sigma(n)}$.

Proof. (Ax) rule: The conclusion is realized by $\lambda x\,.\,x$.

($\bot$E) rule: every realizer of the hypothesis is also a realizer of the conclusion, since $\psi(i) \subseteq \bot(i) = \Pi$ for all $i \in I$.

($\Rightarrow$I) rule: the hypothesis and the conclusion have precisely the same realizers.

($\Rightarrow$E) rule: if t realizes $\Delta \vdash_I \psi \Rightarrow \theta$ and u realizes $\Gamma \vdash_i \psi$ then $\Gamma, \Delta \vdash_i \theta$ is realized by $\lambda x_1 \dots x_n y_1 \dots y_m\,.\,t y_1 \dots y_m (u x_1 \dots x_n)$.

(PeL) rule ('Peirce's law'): the conclusion is realized by $\mathfrak{cc}$. To see this, let $i \in I$, $t \Vdash (\psi(i) \Rightarrow \bot) \Rightarrow \psi(i)$, and $\pi \in \psi(i)$. Then we have $\mathfrak{cc} \star t \cdot \pi \succ t \star \mathsf{k}_\pi \cdot \pi$, which is in $\perp\!\!\!\perp$ since $\mathsf{k}_\pi \cdot \pi \in (\psi(i) \Rightarrow \bot) \Rightarrow \psi(i)$ by Lemma 14 and the definition (6) of implication.

(W) rule: if t realizes $\Gamma \vdash_I \psi$, then $\lambda x\,.\,t$ realizes $A, \Gamma \vdash_I \psi$.

(C) rule: if t realizes $A, A, \Gamma \vdash_I \psi$, then $\lambda x\,.\,txx$ realizes $A, \Gamma \vdash_I \psi$.

(S) rule: if t realizes $\Gamma \vdash_I \psi$, then $\lambda x_{\sigma(1)} \dots x_{\sigma(n)}\,.\,t x_1 \dots x_n$ realizes $\sigma(\Gamma) \vdash_I \psi$.

($\forall$I) and ($\forall$E) rules: $\Gamma \vdash_I \forall_f \xi$ and $f^*\Gamma \vdash_J \xi$ have exactly the same realizers. Indeed, a realizer of $f^*\Gamma \vdash_J \xi$ is a term t satisfying

$$\forall j \in J, u_1 \in \varphi_1(f(j))^{\perp\!\!\!\perp}, \dots, u_n \in \varphi_n(f(j))^{\perp\!\!\!\perp}, \pi \in \xi(j)\,.\,t \star u_1 \cdot \ldots \cdot u_n \cdot \pi \in \perp\!\!\!\perp,$$

and a realizer of $\Gamma \vdash_J \forall_f \xi$ is a term t satisfying

$$\forall i \in I, u_1 \in \varphi_1(i)^{\perp\!\!\!\perp}, \dots, u_n \in \varphi_n(i)^{\perp\!\!\!\perp}, \pi \in \textstyle\bigcup_{f(j)=i} \xi(j)\,.\,t \star u_1 \cdot \ldots \cdot u_n \cdot \pi \in \perp\!\!\!\perp,$$

and both statements can be rephrased as a quantification over pairs (i, j) with $f(j) = i$. ◀

We only defined the propositional connectives $\bot, \Rightarrow$, since $\top, \wedge, \vee, \neg$ can be encoded as follows:

$$\begin{array}{rcl@{\qquad}rcl} \top & \equiv & \bot \Rightarrow \bot & \neg\varphi & \equiv & \varphi \Rightarrow \bot \\ \varphi \wedge \psi & \equiv & (\varphi \Rightarrow (\psi \Rightarrow \bot)) \Rightarrow \bot & \varphi \vee \psi & \equiv & (\varphi \Rightarrow \bot) \Rightarrow \psi \end{array} \tag{7}$$

With these encodings it is routine to show the following.

▶ **Lemma 18.** *With the connectives $\top, \wedge, \vee, \neg$ encoded as in (7), the rules of propositional classical natural deduction (e.g. system Nc in [19, Section 2.1.8]) are derivable from the rules in Table 1.*

With this we can show that for any set I, the binary part of the entailment relation makes $P(\mathsf{\Pi})^I$ into a *Boolean prealgebra*.

▶ **Definition 19.** A *Boolean prealgebra* is a preorder $(B, \leq)$ which

1. has binary *joins* and *meets* – denoted by $x \vee y$ and $x \wedge y$ for $x, y \in D$,
2. has a *least element* $\bot$ and a *greatest element* $\top$,
3. is *distributive* in the sense that $x \wedge (y \vee z) \cong (x \wedge y) \vee (x \wedge z)$ for all $x, y, z \in B$, and
4. is *complemented*, i.e. for every $x \in D$ there exists a $\neg x$ with $x \wedge \neg x \cong \bot$ and $x \vee \neg x \cong \top$.

▶ **Lemma 20.** *Writing $\varphi \leq \psi$ for $\varphi \vdash_I \psi$, $(P(\mathsf{\Pi})^I, \leq)$ is a Boolean prealgebra.*

Proof. The (Ax) rule implies that $\leq$ is reflexive, and transitivity follows from the derivation

$$\dfrac{\varphi \vdash_I \psi \qquad \dfrac{\psi \vdash_I \theta}{\vdash_I \psi \Rightarrow \theta}}{\varphi \vdash_I \theta}.$$

Thus, $\leq$ is a preorder on $P(\mathsf{\Pi})^I$.

The joins, meets, complements, and least and greatest element are given by the corresponding logical operations as defined in (6) and (7).

The required properties all follow from derivability of corresponding entailments and rules in classical natural deduction – for example, $\varphi \wedge \psi$ is a binary meet of φ and ψ since

$(*)$ the entailments $\varphi \wedge \psi \vdash_I \varphi$ and $\varphi \wedge \psi \vdash_I \psi$ and the rule $\dfrac{\theta \vdash_I \varphi \qquad \theta \vdash_I \psi}{\theta \vdash_I \varphi \wedge \psi}$

are derivable.

Distributivity follows from derivability of the entailments $\varphi \wedge (\psi \vee \theta) \vdash_I (\varphi \wedge \psi) \vee (\varphi \wedge \theta)$ and $(\varphi \wedge \psi) \vee (\varphi \wedge \theta) \vdash_I \varphi \wedge (\psi \vee \theta)$. ◀

We now come to *triposes*, which are a kind categorical model for higher order logic. We use a 'strictified' version of the original definition [7, Def. 1.2] since this bypasses some subtleties and is sufficient for our purposes. Furthermore, we are only interested modeling classical logic here, and thus can restrict attention to triposes whose fibers are Boolean (instead of Heyting) prealgebras.

▶ **Definition 21.** A *strict*[6] *Boolean tripos* is a contravariant functor $\mathcal{P} : \mathbf{Set}^{\mathsf{op}} \to \mathbf{Ord}$ from the category of sets to the category of preorders such that

- for every set I, the preorder $\mathcal{P}(I)$ is a Boolean prealgebra, and for any function $f : J \to I$, the induced monotone map $\mathcal{P}(f) : \mathcal{P}(I) \to \mathcal{P}(J)$ preserves all Boolean prealgebra structure.

[6] 'Strict' refers to the facts that (i) $\mathcal{P}$ is a *functor*, not merely a pseudofunctor (ii) the Boolean prealgebra structure is preserved 'on the nose' by the monotone maps $\mathcal{P}(f)$ (iii) the Beck-Chevalley condition is required up to equality, not merely isomorphism, (iv) we require equality and uniqueness in the last condition. Every strict tripos is a tripos in the usual sense, and conversely it can be shown that any tripos is equivalent to a strict one.

- for any $f : J \to I$, $\mathcal{P}(f)$ has left and right adjoints[7] $\exists_f \dashv \mathcal{P}(f) \dashv \forall_f$ such that

$$\text{for any pullback square}^{8} \quad \begin{array}{ccc} L & \xrightarrow{q} & K \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle g} \\ J & \xrightarrow{f} & I \end{array} \tag{8}$$

  we have $\mathcal{P}(g) \circ \forall_f = \forall_q \circ \mathcal{P}(p)$ (this is the *Beck-Chevalley condition*), and
- there exists a *generic predicate*, i.e. a set Prop and an element $\mathsf{tr} \in \mathcal{P}(\mathsf{Prop})$ such that for every set I and $\varphi \in \mathcal{P}(I)$ there exists a unique function $f : I \to \mathsf{Prop}$ with $\mathcal{P}(f)(\mathsf{tr}) = \varphi$.

The assignment $I \mapsto (P(\Pi)^I, \leq)$ extends to a functor $\mathcal{P}_{\perp\!\!\!\perp} : \mathbf{Set}^{\mathsf{op}} \to \mathbf{Ord}$ by letting $\mathcal{P}_{\perp\!\!\!\perp}(f) = f^*$, i.e. mapping every function $f : J \to I$ to the reindexing function along f, which is monotone since every realizer of $\varphi \vdash_I \psi$ is also a realizer of $\varphi \circ f \vdash_J \psi \circ f$.

▶ **Theorem 22.** *$\mathcal{P}_{\perp\!\!\!\perp}$ is a strict Boolean tripos.*

Proof. We have shown in Lemma 20 that the preorders $(P(\Pi)^I, \leq)$ are Boolean prealgebras. It is immediate from (6) that the reindexing functions f^* preserve $\bot$ and $\Rightarrow$, and the other Boolean operations are preserved since they are given by encodings.

The identity function $\mathrm{id} : P(\Pi) \to P(\Pi)$ is a generic predicate for $\mathcal{P}_{\perp\!\!\!\perp}$.

The ($\forall$I) and ($\forall$E) rules together imply that the operation $\forall_f : P(\Pi)^I \to P(\Pi)^J$ is right adjoint to f^* for any $f : J \to I$. Existential quantification along $f : J \to I$ is given by $\exists_f = \neg \circ \forall_f \circ \neg$, which is left adjoint to f^* since

$$\neg\forall_f \neg\varphi \vdash_I \psi \quad \text{iff} \quad \neg\psi \vdash_I \forall_f \neg\varphi \quad \text{iff} \quad f^*\neg\psi \vdash_J \neg\varphi \quad \text{iff} \quad \neg f^*\psi \vdash_J \neg\varphi \quad \text{iff} \quad \varphi \vdash_J f^*\psi$$

for all $\varphi : J \to P(\Pi)$ and $\psi : I \to P(\Pi)$.

It remains to verify the Beck-Chevalley condition. Given a square as in (8) we have

$$g^*\forall_f(\varphi(k)) = \bigcup\{\varphi(j) \mid f(j) = g(k)\} \quad \text{and} \quad \forall_q(p^*(k)) = \bigcup\{\varphi(j) \mid \exists l \,.\, pl = j \wedge ql = k\},$$

and the two terms are equal since the square is a pullback. ◀

Thus we obtain a tripos $\mathcal{P}_{\perp\!\!\!\perp}$ *for each pole* $\perp\!\!\!\perp$. As Hyland, Johnstone, and Pitts showed in [7], every tripos $\mathcal{P}$ gives rise to a topos $\mathbf{Set}[\mathcal{P}]$ via the *tripos-to-topos construction*. Since the fibers of the triposes $\mathcal{P}_{\perp\!\!\!\perp}$ are Boolean prealgebras, the toposes $\mathbf{Set}[\mathcal{P}_{\perp\!\!\!\perp}]$ are Boolean as well, which means that their internal logic is classical.

5.1.1 Consistency

Triposes of the form $\mathcal{P}_{\perp\!\!\!\perp}$ can be degenerate in two ways: if $\perp\!\!\!\perp$ is empty then $\mathcal{P}_{\perp\!\!\!\perp}(I) \simeq (P(I), \subseteq)$ for every set I, and the topos $\mathbf{Set}[\mathcal{P}_{\perp\!\!\!\perp}]$ is equivalent to the category $\mathbf{Set}$.

If, in the other extreme, the pole is so big that there exists a proof-like t realizing $\bot$, i.e. falsity is valid in the model, then we have $\mathcal{P}_{\perp\!\!\!\perp}(I) \simeq 1$ for all I (since t realizes every entailment $\varphi \vdash_I \psi$), and the topos $\mathbf{Set}[\mathcal{P}_{\perp\!\!\!\perp}]$ is equivalent to the terminal category.

By *consistency* we mean that falsity is *not* valid, or equivalently that

$$\forall t \in \mathsf{PL}\ \exists \pi \in \Pi \,.\, t \star \pi \notin \perp\!\!\!\perp. \tag{9}$$

[7] 'Adjoint' in the sense of 'adjoint functor', where monotone maps are viewed as functors between degenerate categories.

[8] The square being a pullback means that $f \circ p = g \circ q$ and $\forall jk \,.\, f(j) = g(k) \Rightarrow \exists! l \,.\, p(l) = j \wedge q(l) = k$.

The 'canonical' (according to Krivine [12]) non-trivial consistent pole is the *thread model*, which is given by postulating a stack constant π_t for each proof-like term t, and defining $\perp\!\!\!\perp = \{p \in \Lambda\star\Pi \mid \neg\exists t \in \mathsf{PL}\,.\, t \star \pi_t \rightsquigarrow^* p\}$. Then the processes $t \star \pi_t$ are not in $\perp\!\!\!\perp$ for any proof-like t, which ensures the validity of condition (9).

In the next section we show how the presence of side effects allows to define a variety of new, 'meaningful' consistent poles.

5.2 Krivine realizability with I/O

The developments of the previous section generalize pretty much directly to the syntax with I/O. Concretely, we carry over the definitions of *pole*, *truth value*, *realizer*, *predicate*, and of the basic logical operations $\bot, \Rightarrow, \forall$, by replacing Λ with Λ_e, Π with Π_e, and $\Lambda\star\Pi$ with P.

We point out that in presence of effects, Definition 13 only means that $\perp\!\!\!\perp$ is saturated w.r.t. *effect-free* evaluation, in contrast to Miquel's approach [14] where a pole is a set of (what we call) execution contexts, closed under the entire execution relation.

The concept of proof-like term deserves some reexamination. It turns out that the appropriate concept of *proof-like term* is 'term not containing any side effects'. This is consistent with Definition 15 if we read 'free of side effects' as 'free of non-logical constructs', which are the stack constants in Krivine's case. Continuation terms, on the other hand, can be considered proof-like. We redefine therefore:

▶ **Definition 23.** The set $\mathsf{PL} \subseteq \Lambda_e$ of *proof-like terms* is the set of terms not containing any of the constants $\mathsf{r}, \mathsf{w0}, \mathsf{w1}, \mathsf{end}$.

With this rephrased definition of proof-like term, we can define the entailment relation on the extended predicates in the same way:

▶ **Definition 24.** For any set I and integer n, the $(n+1)$-ary entailment relation $(\vdash_I)$ on the set $P(\Pi_e)^I$ of extended predicates on I is defined by

$$\varphi_1 \dots \varphi_n \vdash_I \psi \quad \text{if and only if} \quad \exists t \in \mathsf{PL}\,.\, t \Vdash \forall_I(\varphi_1 \Rightarrow \dots \Rightarrow \varphi_n \Rightarrow \psi).$$

As a special case, the ordering on extended predicates is defined by

$$\varphi \leq \psi \quad \text{if and only if} \quad \exists t \in \mathsf{PL}\,.\, t \Vdash \forall_I(\varphi \Rightarrow \psi).$$

With these definitions, we can state analogues of Lemma 20 and Theorem 22:

▶ **Theorem 25.**

- *For each set I, the order $(P(\Pi_e)^I, \leq)$ of extended predicates is a Boolean prealgebra.*
- *The assignment $I \mapsto (P(\Pi_e)^I, \leq)$ gives rise to a strict Boolean tripos $\mathcal{P}_{\perp\!\!\!\perp} : \mathbf{Set}^{\mathsf{op}} \to \mathbf{Ord}$.*

Proof. This follows from the arguments in Section 5.1, since the proofs of Lemmas 14,17, 18,20, and of Theorem 22 are not obstructed in any way by the new constants, nor do they rely on stack constants. The redefinition of 'proof-like term' does not cause any problems either, since we never relied on proof-like terms not containing continuation terms. ◀

The above rephrasing of the definition of proof-like term admits an intuitive reformulation of the consistency criterion (9):

▶ **Lemma 26.** *A pole $\perp\!\!\!\perp$ is consistent if and only if every $p \in \perp\!\!\!\perp \setminus \{\top\}$ contains a non-logical constant, i.e. one of* $\mathsf{r}, \mathsf{w0}, \mathsf{w1}, \mathsf{end}$.

Proof. If every element of $p \in \perp\!\!\!\perp \setminus \{\top\}$ contains a non-logical constant, then $t \star \varepsilon$ is not in $\perp\!\!\!\perp$ for any proof-like t, which implies (9).

On the other hand, if $t * \pi \in \perp\!\!\!\perp$ does not contain any non-logical constant then $\mathsf{k}_\pi t$ is a proof-like term which realizes $\bot$, since for any $\rho \in \Pi_e$ we have $\mathsf{k}_\pi t \star \rho \succ \mathsf{k}_\pi \star t \cdot \rho \succ t * \pi \in \perp\!\!\!\perp$. ◀

5.2.1 Poles from specifications

The connection between poles and specifications is established by the following lemma.

▶ **Lemma 27.** *Every set $\perp\!\!\!\perp$ of processes that is closed under weak bisimilarity is a pole.*

Proof. This is because $p \approx q$ whenever $p \succ q$, which follows from Lemma 5. ◀

Since we can assume that for any reasonable specification the processes implementing it are closed under weak bisimilarity, we can thus conclude that for any specification, the set of processes implementing it is a pole. For example:

- $\perp\!\!\!\perp_{\mathrm{cp}}$ is the set of processes that read the input, copy every bit immediately to the output, and terminate when the input is empty. We have $Y \star (\lambda x\,.\,\mathsf{r}(\mathsf{w0}\,x)(\mathsf{w1}\,x)\mathsf{end}) \in \perp\!\!\!\perp_{\mathrm{cp}}$.
- $\perp\!\!\!\perp_{\mathrm{cp}'}$ contains the processes that first read the entire input, and then write out the same string and terminate. We have $R \star F{\cdot}W{\cdot}\overline{0} \in \perp\!\!\!\perp_{\mathrm{cp}'}$ with the notations of Section 4.
- For any partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$, the pole $\perp\!\!\!\perp_f$ consists of those processes that implement f in the sense of Definition 1.
- Since poles are closed under unions, we can define the pole $\perp\!\!\!\perp_F = \bigcup_{f\in F} \perp\!\!\!\perp_f$ for any set $F \subseteq (\mathbb{N} \rightharpoonup \mathbb{N})$ of partial functions.

5.2.2 Toposes from computable functions

We are particularly interested in the poles $\perp\!\!\!\perp_f$ associated to computable functions f, and we want to use the associated triposes $\mathcal{P}_f = \mathcal{P}_{\perp\!\!\!\perp_f}$ and toposes $\mathbf{Set}[\mathcal{P}_f]$ to study these functions.

The following theorem provides a first 'sanity check', in showing that the associated models are non-degenerate.

▶ **Theorem 28.** *Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$.*

- *$\perp\!\!\!\perp_f$ is consistent if and only if f is not totally undefined.*
- *$\perp\!\!\!\perp_f$ is non-empty if and only if f is computable.*

Proof. The first claim follows from Lemma 26. If $n \in \mathrm{dom}(f)$ and $t \star \pi$ implements f, then $(t \star \pi, \mathrm{bin}(n), \varepsilon)$ must terminate and thus $t \star \pi$ must contain an end instruction. The totally undefined function, on the other hand, is by definition implemented by every process.

For the second claim, we have shown in Theorem 12 that every computable f is implemented by some process. Conversely, every implementable function is computable since the Krivine machine with I/O is an effective model of computation. ◀

5.3 Discussion and future work

The structure and properties of the toposes $\mathbf{Set}[\mathcal{P}_f]$ remain mysterious for the moment, and in future work we want to explore which kind of properties of f are reflected in $\mathbf{Set}[\mathcal{P}_f]$. In the spirit of Grothendieck [4] we want to view the toposes $\mathbf{Set}[\mathcal{P}_f]$ as *geometric* rather than logical objects, the guiding intuition being that $\mathbf{Set}[\mathcal{P}_f]$ can be seen as representation of *'the space of solutions to the algorithmic problem of computing f'*, encoding e.g. information on how algorithms computing f can be decomposed into simpler parts.

Evident problems to investigate are to understand the lattice of truth values in $\mathbf{Set}[\mathcal{P}_f]$, and to determine for which pairs f, g of functions the associated toposes are equivalent, and which functions can be separated.

A more audacious goal is to explore whether $\mathbf{Set}[\mathcal{P}_f]$ can teach us something about the complexity of a computable function f. The Krivine machine with I/O seems to be a model of computation that is fine grained enough to recognize and differentiate time complexity

of different implementations of f, but it remains to be seen in how far this information is reflected in the 'geometry' of $\mathbf{Set}[\mathcal{P}_f]$.

Acknowledgements. Thanks to Jakob Grue Simonsen and Thomas Streicher for many discussions.

References

1 H.P. Barendregt. *The lambda calculus, Its syntax and semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, revised edition, 1984.

2 W. Ferrer, J. Frey, M. Guillermo, O. Malherbe, and A. Miquel. Ordered combinatory algebras and realizability. *arXiv preprint arXiv:1410.5034*, 2014.

3 H. Friedman. Classically and intuitionistically provably recursive functions. In *Higher set theory (Proc. Conf., Math. Forschungsinst., Oberwolfach, 1977)*, volume 669 of *Lecture Notes in Math.*, pages 21–27. Springer, Berlin, 1978.

4 A. Grothendieck, M. Artin, and J.L. Verdier. Théorie des topos et cohomologie étale des schémas. *Lecture Notes in Mathematics*, 269, 1972.

5 J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and combinators, an introduction*. Cambridge University Press, Cambridge, 2008.

6 J.M.E. Hyland. The effective topos. In *The L.E.J. Brouwer Centenary Symposium (Noordwijkerhout, 1981)*, volume 110 of *Stud. Logic Foundations Math.*, pages 165–216. North-Holland, Amsterdam, 1982.

7 J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. Tripos theory. *Math. Proc. Cambridge Philos. Soc.*, 88(2):205–231, 1980.

8 B. Jacobs. *Categorical logic and type theory*. Elsevier Science Ltd, 2001.

9 J.L. Krivine. Lambda-calcul, évaluation paresseuse et mise en mémoire. *RAIRO Inform. Théor. Appl.*, 25(1):67–84, 1991.

10 J.L. Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.

11 J.L. Krivine. Realizability algebras: a program to well order $\mathbb{R}$. *Log. Methods Comput. Sci.*, 7(3):3:02, 47, 2011.

12 J.L. Krivine. Realizability algebras II: New models of ZF + DC. *Log. Methods Comput. Sci.*, 8(1):1:10, 28, 2012.

13 R. Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of theoretical computer science, Vol. B*, pages 1201–1242. Elsevier, Amsterdam, 1990.

14 A. Miquel. Classical modal realizability and side effects. *preprint*, 2009.

15 A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Log. Methods Comput. Sci.*, 7(2):2:2, 47, 2011.

16 A. Miquel. Forcing as a program transformation. In *26th Annual IEEE Symposium on Logic in Computer Science—LICS 2011*, pages 197–206. IEEE Computer Soc., Los Alamitos, CA, 2011.

17 W.P. Stekelenburg. *Realizability Categories*. PhD thesis, Utrecht University, 2013.

18 T. Streicher. Krivine's classical realisability from a categorical perspective. *Mathematical Structures in Computer Science*, 23(06):1234–1256, 2013.

19 A.S. Troelstra and H. Schwichtenberg. *Basic proof theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1996.

20 J. van Oosten. *Classical Realizability*. Invited talk at "Cambridge Category Theory Seminar", slides at `http://www.staff.science.uu.nl/~ooste110/talks/cambr060312.pdf`.

21 J. van Oosten. *Realizability: An Introduction to its Categorical Side*. Elsevier Science Ltd, 2008.

Standardization of a Call-By-Value Lambda-Calculus*

Giulio Guerrieri[1], Luca Paolini[2], and Simona Ronchi Della Rocca[2]

1 **Laboratoire PPS, UMR 7126, Université Paris Diderot, Sorbonne Paris Cité**
75205 Paris, France
giulio.guerrieri@pps.univ-paris-diderot.fr

2 **Dipartimento di Informatica, Università degli Studi di Torino**
Corso Svizzera 185, Torino, Italy
{paolini,ronchi}@di.unito.it

Abstract

We study an extension of Plotkin's call-by-value lambda-calculus by means of two commutation rules (sigma-reductions). Recently, it has been proved that this extended calculus provides elegant characterizations of many semantic properties, as for example solvability. We prove a standardization theorem for this calculus by generalizing Takahashi's approach of parallel reductions. The standardization property allows us to prove that our calculus is conservative with respect to the Plotkin's one. In particular, we show that the notion of solvability for this calculus coincides with that for Plotkin's call-by-value lambda-calculus.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.3.2 Semantics of Programming Language, F.4.1 Mathematical Logic

Keywords and phrases standardization, sequentialization, lambda-calculus, sigma-reduction, parallel reduction, call-by-value, head reduction, internal reduction, solvability, potential valuability, observational equivalence

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.211

1 Introduction

The λ_v-calculus (λ_v for short) has been introduced by Plotkin in [15], in order to give a formal account of the call-by-value evaluation, which is the most commonly used parameter passing policy for programming languages. λ_v shares the syntax with the classical, call-by-name, λ-calculus (λ for short), but its reduction rule, β_v, is a restriction of β, firing only in case the argument is a value (i.e., a variable or an abstraction). While β_v is enough for evaluation, it turned out to be too weak to study operational properties of terms. For example, in λ, the β-reduction is sufficient to characterize solvability and (using extensionality) separability, but, in order to characterize similar properties for λ_v, it has been necessary to introduce different notions of reduction unsuitable for a correct call-by-value evaluation (see [13, 14]): this is disappointing and requires complex reasoning. In this paper we study λ_v^σ, the extension of λ_v proposed in [3]. It keeps the λ_v (and λ) syntax and it adds to the β_v-reduction two commutation rules, called σ_1 and σ_3, which unblock β_v-redexes that are hidden by the "hyper-sequential structure" of terms. It is well-known (see [14, 1]) that in λ_v there are normal forms that are unsolvable, e.g. $(\lambda yx.xx)(zz)(\lambda x.xx)$. The more evident benefit of λ_v^σ

* This work was partially supported by LINTEL TO_Call1_2012_0085, i.e. a Research Project funded by the "Compagnia di San Paolo".

is that the commutation rules make all normal forms solvable (indeed $(\lambda yx.xx)(zz)(\lambda x.xx)$ is not a λ_v^σ normal form). More generally, the so obtained language, allows us to characterize operational properties, like solvability and potential valuability, in an internal and elegant way (see [3]). In this paper we prove a standardization property in λ_v^σ, and some of its consequences, namely its soundness with respect to the semantics of λ_v.

Let us recall the notion of standardization, which has been first studied in the ordinary λ-calculus (see, for example [5, 8, 2]). A reduction sequence is standard if its redexes are ordered in a given way, and the corresponding standardization theorem establishes that every reduction sequence can constructively be transformed into a standard one. Standardization is a key tool to grasp the way in which reductions work, that sheds some light on redexes relationships and their dependencies. It is useful for characterization of semantic properties through reduction strategies (the proof of operational semantics adequacy is a typical use).

In the λ_v setting standardization theorems have been proved by Plotkin [15], Paolini and Ronchi Della Rocca [14, 12] and Crary [4]. The definition of standard sequence of reductions considered by Plotkin and Crary coincides, and it imposes a partial order on redexes, while Paolini and Ronchi Della Rocca define a total order on them. All these proofs are developed by using the notion of parallel reduction introduced by Tait and Martin-Löf (see Takahashi [17] for details and interesting technical improvements). We emphasize that this method does not involve the notion of residual of a redex, on which many classical proofs for the λ-calculus are based (see for example [8, 2]). As in [15, 17, 14, 4], we use a suitable notion of parallel reduction for developing our standardization theorem for λ_v^σ. In particular we consider two groups of redexes, head β_v-redexes and head σ-redexes (putting together σ_1 and σ_3), and we induce a total order on head redexes of the two groups, without imposing any order on head σ-redexes themselves. More precisely, when σ-redexes are missing, this notion of standardization coincides with that presented in [14]. Moreover, we show that it is not possible to strengthen our standardization by (locally) ordering σ_1-reduction to σ_3-reduction (or viceversa).

As usual, our standardization proof is based on a sequentialization result: inner reductions can always be postponed after the head ones, for a non-standard definition of head reduction. Sequentialization has interesting consequences: it allows us to prove that fundamental operational properties in λ_v^σ, like observational equivalence, potential valuability and solvability, are conservative with respect to the corresponding notions of λ_v. This fully justifies the project in [3] where λ_v^σ has been introduced as a tool for studying the operational behaviour of λ_v.

Other variants of λ_v have been introduced in the literature for modeling the call-by-value computation. We would like to cite here at least the contributions of Moggi [10], Felleisen and Sabry [16], Maraist et al. [9], Herbelin and Zimmerman [7], Accattoli and Paolini [1]. All these proposals are based on the introduction of new constructs to the syntax of λ_v, so the comparison between them is not easy with respect to syntactical properties (some detailed comparison is given in [1]). We point out that the calculi introduced in [10, 16, 9, 7] present some variants of our σ_1 and/or σ_3 rules, often in a setting with explicit substitutions.

Outline. In Section 2 we introduce the language λ_v^σ and its operational behaviour; in Section 3 the sequentialization property is proved; Section 4 contains the main result, i.e., standardization ; in Section 5 some conservativity results with respect to Plotkin's λ_v-calculus are proved. Section 6 concludes the paper, with some hints for future work.

2 The call-by-value lambda calculus with sigma-rules

In this section we present λ_v^σ, a call-by-value λ-calculus introduced in [3] that adds two σ-reduction rules to pure (i.e. without constants) call-by-value λ-calculus defined by Plotkin in [15].

The syntax of terms of λ_v^σ [3] is the same as the one of ordinary λ-calculus and Plotkin's call-by-value λ-calculus λ_v [15] (without constants). Given a countable set $\mathcal{V}$ of *variables* (denoted by $x, y, z, \dots$), the sets Λ of *terms* and Λ_v of *values* are defined by mutual induction:

$$\begin{array}{lrll} (\Lambda_v) & V, U & ::= & x \mid \lambda x.M \qquad \textit{values} \\ (\Lambda) & M, N, L & ::= & V \mid MN \qquad \textit{terms} \end{array}$$

Clearly, $\Lambda_v \subseteq \Lambda$. All terms are considered up to α-conversion. The set of free variables of a term M is denoted by $\mathsf{fv}(M)$. Given $V_1, \dots, V_n \in \Lambda_v$ and pairwise distinct variables $x_1, \dots, x_n$, $M\{V_1/x_1, \dots, V_n/x_n\}$ denotes the term obtained by the *capture-avoiding simultaneous substitution* of V_i for each free occurrence of x_i in the term M (for all $1 \leq i \leq n$). Note that, for all $V, V_1, \dots, V_n \in \Lambda_v$ and pairwise distinct variables $x_1, \dots, x_n$, $V\{V_1/x_1, \dots, V_n/x_n\} \in \Lambda_v$.

Contexts (with exactly one hole $(\!|\cdot|\!)$), denoted by $\mathtt{C}$, are defined as usual via the grammar:

$$\mathtt{C} ::= (\!|\cdot|\!) \mid \lambda x.\mathtt{C} \mid \mathtt{C}M \mid M\mathtt{C}\,.$$

We use $\mathtt{C}(\!|M|\!)$ for the term obtained by the capture-allowing substitution of the term M for the hole $(\!|\cdot|\!)$ in the context $\mathtt{C}$.

▶ **Notation.** From now on, we set $I = \lambda x.x$ and $\Delta = \lambda x.xx$.

The reduction rules of λ_v^σ consist of Plotkin's β_v-reduction rule, introduced in [15], and two simple commutation rules called σ_1 and σ_3, studied in [3].

▶ **Definition 1** (Reduction rules)**.** We define the following binary relations on Λ (for any $M, N, L \in \Lambda$ and any $V \in \Lambda_v$):

$$\begin{array}{rcll} (\lambda x.M)V & \mapsto_{\beta_v} & M\{V/x\} & \\ (\lambda x.M)NL & \mapsto_{\sigma_1} & (\lambda x.ML)N & \text{with } x \notin \mathsf{fv}(L) \\ V((\lambda x.L)N) & \mapsto_{\sigma_3} & (\lambda x.VL)N & \text{with } x \notin \mathsf{fv}(V). \end{array}$$

For any $\mathsf{r} \in \{\beta_v, \sigma_1, \sigma_3\}$, if $M \mapsto_\mathsf{r} M'$ then M is a r-*redex* and M' is its r-*contractum*. In this sense, a term of the shape $(\lambda x.M)N$ (for any $M, N \in \Lambda$) is a β-*redex*.

We set $\mapsto_\sigma = \mapsto_{\sigma_1} \cup \mapsto_{\sigma_3}$ and $\mapsto_\mathsf{v} = \mapsto_{\beta_v} \cup \mapsto_\sigma$.

The side conditions on $\mapsto_{\sigma_1}$ and $\mapsto_{\sigma_3}$ in Definition 1 can be always fulfilled by α-renaming.

Obviously, any β_v-redex is a β-redex but the converse does not hold: $(\lambda x.z)(yI)$ is a β-redex but not a β_v-redex. Redexes of different kind may overlap: for example, the term $\Delta I\Delta$ is a σ_1-redex and it contains the β_v-redex ΔI; the term $\Delta(I\Delta)(xI)$ is a σ_1-redex and it contains the σ_3-redex $\Delta(I\Delta)$, which contains in turn the β_v-redex $I\Delta$.

According to the Girard's call-by-value "boring" translation $(\cdot)^v$ of terms into Intuitionistic Multiplicative Exponential Linear Logic proof-nets, defined by $(A \Rightarrow B)^v = !A^v \multimap !B^v$ (see [6]), the images under $(\cdot)^v$ of a σ_1-redex (resp. σ_3-redex) and its contractum are equal modulo some "bureaucratic" steps of cut-elimination.

▶ **Notation.** Let R be a binary relation on Λ. We denote by R^* (resp. R^+; $\mathsf{R}^=$) the reflexive-transitive (resp. transitive; reflexive) closure of R.

▶ **Definition 2** (Reductions). Let $\mathsf{r} \in \{\beta_v, \sigma_1, \sigma_3, \sigma, \mathsf{v}\}$.

The r-*reduction* $\to_{\mathsf{r}}$ is the contextual closure of $\mapsto_{\mathsf{r}}$, i.e. $M \to_{\mathsf{r}} M'$ iff there is a context $\mathtt{C}$ and $N, N' \in \Lambda$ such that $M = \mathtt{C}\langle\!\langle N \rangle\!\rangle$, $M' = \mathtt{C}\langle\!\langle N' \rangle\!\rangle$ and $N \mapsto_{\mathsf{r}} N'$.

The r-*equivalence* $=_{\mathsf{r}}$ is the reflexive-transitive and symmetric closure of $\to_{\mathsf{r}}$.

Let M be a term: M is r-*normal* if there is no term N such that $M \to_{\mathsf{r}} N$; M is r-*normalizable* if there is a r-normal term N such that $M \to_{\mathsf{r}}^* N$; M is *strongly* r-*normalizing* if there is no sequence $(N_i)_{i\in\mathbb{N}}$ such that $M = N_0$ and $N_i \to_{\mathsf{r}} N_{i+1}$ for any $i \in \mathbb{N}$. Finally, $\to_{\mathsf{r}}$ is *strongly normalizing* if every $N \in \Lambda$ is strongly r-normalizing.

Patently, $\to_\sigma \subsetneq \to_{\mathsf{v}}$ and $\to_{\beta_v} \subsetneq \to_{\mathsf{v}}$.

▶ **Remark 3.** For any $\mathsf{r} \in \{\beta_v, \sigma_1, \sigma_3, \sigma, \mathsf{v}\}$ (resp. $\mathsf{r} \in \{\sigma_1, \sigma_3, \sigma\}$), values are closed under r-reduction (resp. r-expansion): for any $V \in \Lambda_v$, if $V \to_{\mathsf{r}} M$ (resp. $M \to_{\mathsf{r}} V$) then $M \in \Lambda_v$; more precisely, $V = \lambda x.N$ and $M = \lambda x.N'$ for some $N, N' \in \Lambda$ with $N \to_{\mathsf{r}} N'$ (resp. $N' \to_{\mathsf{r}} N$).

▶ **Proposition 4** (See [3]). *The σ-reduction is confluent and strongly normalizing. The v-reduction is confluent.*

The λ_v^σ-*calculus*, λ_v^σ for short, is the set Λ of terms endowed with the v-reduction $\to_{\mathsf{v}}$. The set Λ endowed with the β_v-reduction $\to_{\beta_v}$ is the λ_v-calculus (λ_v for short), i.e. the Plotkin's call-by-value λ-calculus [15] (without constants), which is thus a sub-calculus of λ_v^σ.

▶ **Example 5.** $M = (\lambda y.\Delta)(xI)\Delta \to_{\sigma_1} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} \dots$ and $N = \Delta((\lambda y.\Delta)(xI)) \to_{\sigma_3} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} \dots$ are the only possible v-reduction paths from M and N respectively: M and N are not v-normalizable, and $M =_{\mathsf{v}} N$. Meanwhile, M and N are β_v-normal and different, hence $M \neq_{\beta_v} N$ (by confluence of $\to_{\beta_v}$, see [15]).

Informally, σ-rules unblock β_v-redexes which are hidden by the "hyper-sequential structure" of terms. This approach is alternative to the one in [1] where hidden β_v-redexes are reduced using rules acting at a distance (through explicit substitutions). It can be shown that the call-by-value λ-calculus with explicit substitution introduced in [1] can be embedded in λ_v^σ.

3 Sequentialization

In this section we aim to prove a sequentialization theorem (Theorem 22) for the λ_v^σ-calculus by adapting Takahashi's method [17, 4] based on parallel reductions.

▶ **Notation.** From now on, we always assume that $V, V' \in \Lambda_v$.

Note that the generic form of a term is $VM_1 \dots M_m$ for some $m \in \mathbb{N}$ (in particular, values are obtained when $m = 0$). The sequentialization result is based on a partitioning of v-reduction between head and internal reduction.

▶ **Definition 6** (Head β_v-reduction). We define inductively the *head* β_v-*reduction* $\overset{h}{\to}_{\beta_v}$ by the following rules ($m \in \mathbb{N}$ in both rules):

$$\frac{}{(\lambda x.M)VM_1 \dots M_m \overset{h}{\to}_{\beta_v} M\{V/x\}M_1 \dots M_m}\ \beta_v \qquad \frac{N \overset{h}{\to}_{\beta_v} N'}{VNM_1 \dots M_m \overset{h}{\to}_{\beta_v} VN'M_1 \dots M_m}\ \mathit{right}$$

The head β_v-reduction $\overset{h}{\to}_{\beta_v}$ reduces exactly the same redexes (see also [13]) as the "left reduction" defined in [15, p. 136] for λ_v and called "evaluation" in [16, 4]. If $N \overset{h}{\to}_{\beta_v} N'$ then N' is obtained from N by reducing the leftmost-outermost β_v-redex, not in the scope of a λ: thus, the head β_v-reduction is deterministic (i.e., it is a partial function from Λ to Λ) and does not reduce values.

▶ **Definition 7** (Head σ-reduction)**.** We define inductively the *head σ-reduction* $\overset{h}{\to}_\sigma$ by the following rules ($m \in \mathbb{N}$ in all the rules, $x \notin \mathsf{fv}(L)$ in the rule σ_1, $x \notin \mathsf{fv}(V)$ in the rule σ_3):

$$\frac{}{(\lambda x.M)NLM_1 \dots M_m \overset{h}{\to}_\sigma (\lambda x.ML)NM_1 \dots M_m}\,\sigma_1 \qquad \frac{N \overset{h}{\to}_\sigma N'}{VNM_1 \dots M_m \overset{h}{\to}_\sigma VN'M_1 \dots M_m}\,right$$

$$\frac{}{V((\lambda x.L)N)M_1 \dots M_m \overset{h}{\to}_\sigma (\lambda x.VL)NM_1 \dots M_m}\,\sigma_3$$

The *head* (v-)*reduction* is $\overset{h}{\to}_\mathsf{v} = \overset{h}{\to}_{\beta_v} \cup \overset{h}{\to}_\sigma$. The *internal* ($\mathsf{v}$-)*reduction* is $\overset{int}{\to}_\mathsf{v} = \to_\mathsf{v} \setminus \overset{h}{\to}_\mathsf{v}$.

Notice that $\mapsto_{\beta_v} \subsetneq \overset{h}{\to}_{\beta_v} \subsetneq \to_{\beta_v}$ and $\mapsto_\sigma \subsetneq \overset{h}{\to}_\sigma \subsetneq \to_\sigma$ and $\mapsto_\mathsf{v} \subsetneq \overset{h}{\to}_\mathsf{v} \subsetneq \to_\mathsf{v}$. Values are normal forms for the head reduction, but the converse does not hold: $xI \notin \Lambda_v$ is head-normal.

Informally, if $N \overset{h}{\to}_\sigma N'$ then N' is obtained from N by reducing "one of the leftmost" σ_1- or σ_3-redexes, not in the scope of a λ: in general, a term may contain several head σ_1- and σ_3-redexes. Indeed, differently from $\overset{h}{\to}_{\beta_v}$, the head σ-reduction $\overset{h}{\to}_\sigma$ is not deterministic, for example the leftmost-outermost σ_1- and σ_3-redexes may overlap: if $M = (\lambda y.y')(\Delta(xI))I$ then $M \overset{h}{\to}_\sigma (\lambda y.y'I)(\Delta(xI)) = N_1$ by applying the rule σ_1 and $M \overset{h}{\to}_\sigma (\lambda z.(\lambda y.y')(zz))(xI)I = N_2$ by applying the rule σ_3. Note that N_1 contains only a head σ_3-redex and $N_1 \overset{h}{\to}_\sigma (\lambda z.(\lambda y.y'I)(zz))(xI) = N$ which is normal for $\overset{h}{\to}_\mathsf{v}$; meanwhile N_2 contains only a head σ_1-redex and $N_2 \overset{h}{\to}_\sigma (\lambda z.(\lambda y.y')(zz)I)(xI) = N'$ which is normal for $\overset{h}{\to}_\mathsf{v}$: $N \neq N'$, hence the head reduction $\overset{h}{\to}_\mathsf{v}$ is not confluent and a term may have several head-normal forms (this example does not contradict the confluence of σ-reduction because $N' \to_\sigma N$ but by performing an internal reduction step). Later, in Corollary 26.2 we show that if a term M has a head normal form $N \in \Lambda_v$ then N is the unique head normal form of M.

▶ **Definition 8** (Parallel reduction)**.** We define inductively the *parallel reduction* $\Rightarrow$ by the following rules ($x \notin \mathsf{fv}(L)$ in the rule σ_1, $x \notin \mathsf{fv}(V)$ in the rule σ_3):

$$\frac{V \Rightarrow V' \quad M_i \Rightarrow M_i' \quad (m \in \mathbb{N},\ 0 \le i \le m)}{(\lambda x.M_0)VM_1 \dots M_m \Rightarrow M_0'\{V'/x\}M_1' \dots M_m'}\,\beta_v \qquad \frac{N \Rightarrow N' \quad L \Rightarrow L' \quad M_i \Rightarrow M_i' \quad (m \in \mathbb{N},\ 0 \le i \le m)}{(\lambda x.M_0)NLM_1 \dots M_m \Rightarrow (\lambda x.M_0'L')N'M_1' \dots M_m'}\,\sigma_1$$

$$\frac{V \Rightarrow V' \qquad N \Rightarrow N' \qquad L \Rightarrow L' \qquad M_i \Rightarrow M_i' \quad (m \in \mathbb{N},\ 1 \le i \le m)}{V((\lambda x.L)N)M_1 \dots M_m \Rightarrow (\lambda x.V'L')N'M_1' \dots M_m'}\,\sigma_3$$

$$\frac{M_i \Rightarrow M_i' \quad (m \in \mathbb{N},\ 0 \le i \le m)}{(\lambda x.M_0)M_1 \dots M_m \Rightarrow (\lambda x.M_0')M_1' \dots M_m'}\,\lambda \qquad \frac{M_i \Rightarrow M_i' \quad (m \in \mathbb{N},\ 1 \le i \le m)}{x\,M_1 \dots M_m \Rightarrow x\,M_1' \dots M_m'}\,var$$

In Definition 8 the rule *var* has no premises when $m = 0$: this is the base case of the inductive definition of $\Rightarrow$. The rules σ_1 and σ_3 have exactly three premises when $m = 0$.

Intuitively, $M \Rightarrow M'$ means that M' is obtained from M by reducing a number of β_v-, σ_1- and σ_3-redexes (existing in M) simultaneously.

▶ **Definition 9** (Internal and strong parallel reduction)**.** We define inductively the *internal parallel reduction* $\overset{int}{\Rightarrow}$ by the following rules:

$$\frac{N \Rightarrow N'}{\lambda x.N \overset{int}{\Rightarrow} \lambda x.N'}\,\lambda \qquad \frac{}{x \overset{int}{\Rightarrow} x}\,var \qquad \frac{V \Rightarrow V' \quad N \overset{int}{\Rightarrow} N' \quad M_i \Rightarrow M_i' \quad (m \in \mathbb{N},\ 1 \le i \le m)}{VNM_1 \dots M_m \overset{int}{\Rightarrow} V'N'M_1' \dots M_m'}\,right$$

The *strong parallel reduction* $\Rrightarrow$ is defined by: $M \Rrightarrow N$ iff $M \Rightarrow N$ and there exist $M', M'' \in \Lambda$ such that $M \overset{h}{\to}_{\beta_v}^* M' \overset{h}{\to}_\sigma^* M'' \overset{int}{\Rightarrow} N$.

Notice that the rule *right* for $\overset{int}{\Rightarrow}$ has exactly two premises when $m = 0$.

▶ **Remark 10.** The relations $\Rightarrow$, $\Rrightarrow$ and $\overset{int}{\Rightarrow}$ are reflexive. The reflexivity of $\Rrightarrow$ follows immediately from the reflexivity of $\Rightarrow$ and $\overset{int}{\Rightarrow}$. The proofs of reflexivity of $\Rightarrow$ and $\overset{int}{\Rightarrow}$ are both by structural induction on a term: in the case of $\Rightarrow$, recall that every term is of the form $(\lambda x.N)M_1 \dots M_m$ or $x\,M_1 \dots M_m$ for some $m \in \mathbb{N}$ and then apply the rule λ or *var* respectively, together with the inductive hypothesis; in the case of $\overset{int}{\Rightarrow}$, recall that every term is of the form $\lambda x.M$ or x or $VNM_1 \dots M_m$ for some $m \in \mathbb{N}$ and then apply the rule λ (together with the reflexivity of $\Rightarrow$) or *var* or *right* (together with the reflexivity of $\Rightarrow$ and the inductive hypothesis) respectively.

One has $\overset{int}{\Rightarrow} \subsetneq \Rrightarrow \subseteq \Rightarrow$ (first, prove that $\overset{int}{\Rightarrow} \subseteq \Rightarrow$ by induction on the derivation of $M \overset{int}{\Rightarrow} M'$, the other inclusions follow from the definition of $\Rrightarrow$) and, since $\Rightarrow$ is reflexive (Remark 10), $\overset{h}{\to}_{\beta_v} \subsetneq \Rightarrow$ and $\overset{h}{\to}_{\sigma} \subsetneq \Rightarrow$. Observe that $\Delta\Delta\ \mathsf{R}\ \Delta\Delta$ for any $\mathsf{R} \in \{\mapsto_{\beta_v}, \overset{h}{\to}_{\beta_v}, \Rightarrow, \overset{int}{\Rightarrow}, \Rrightarrow\}$, even if for different reasons: for example, $\Delta\Delta \overset{int}{\Rightarrow} \Delta\Delta$ by reflexivity of $\overset{int}{\Rightarrow}$ (Remark 10), whereas $\Delta\Delta \overset{h}{\to}_{\beta_v} \Delta\Delta$ by reducing the (leftmost-outermost) β_v-redex.

Next two further remarks collect many minor properties that can be easily proved.

▶ **Remark 11.**
1. The head β_v-reduction $\overset{h}{\to}_{\beta_v}$ does not reduce a value (in particular, does not reduce under λ's), i.e., for any $M \in \Lambda$ and any $V \in \Lambda_v$, one has $V \not\overset{h}{\to}_{\beta_v} M$.
2. The head σ-reduction $\overset{h}{\to}_{\sigma}$ does neither reduce a value nor reduce to a value, i.e., for any $M \in \Lambda$ and any $V \in \Lambda_v$, one has $V \not\overset{h}{\to}_{\sigma} M$ and $M \not\overset{h}{\to}_{\sigma} V$.
3. Variables and abstractions are preserved under $\overset{int}{\Leftarrow}$ ($\overset{int}{\Rightarrow}$-expansion), i.e., if $M \overset{int}{\Rightarrow} x$ (resp. $M \overset{int}{\Rightarrow} \lambda x.N'$) then $M = x$ (resp. $M = \lambda x.N$ for some $N \in \Lambda$ such that $N \Rightarrow N'$).
4. If $M \Rightarrow M'$ then $\lambda x.M\ \mathsf{R}\ \lambda x.M'$ for any $\mathsf{R} \in \{\Rightarrow, \overset{int}{\Rightarrow}, \Rrightarrow\}$. Indeed, for $\mathsf{R} \in \{\Rightarrow, \overset{int}{\Rightarrow}\}$ apply the rule λ to conclude, then $\lambda x.M \Rrightarrow \lambda x.M'$ according to the definition of $\Rrightarrow$.
5. For any $V, V' \in \Lambda_v$, $V \overset{int}{\Rightarrow} V'$ iff $V \Rightarrow V'$. The left-to-right direction holds because $\overset{int}{\Rightarrow} \subseteq \Rightarrow$; conversely, assume $V \Rightarrow V'$: if V is a variable then necessarily $V = V'$ and hence $V \overset{int}{\Rightarrow} V'$ by applying the rule *var* for $\overset{int}{\Rightarrow}$; otherwise $V = \lambda x.N$ for some $N \in \Lambda$, and then necessarily $V' = \lambda x.N'$ with $N \Rightarrow N'$, so $V \overset{int}{\Rightarrow} V'$ by applying the rule λ for $\overset{int}{\Rightarrow}$.

▶ **Remark 12.**
1. If $M \Rightarrow M'$ and $N \Rightarrow N'$ then $MN \Rightarrow M'N'$. For the proof, it is sufficient to consider the last rule of the derivation of $M \Rightarrow M'$.
2. If $\mathsf{R} \in \{\overset{h}{\to}_{\beta_v}, \overset{h}{\to}_{\sigma}\}$ and $M\ \mathsf{R}\ M'$, then $MN\ \mathsf{R}\ M'N$ for any $N \in \Lambda$. For the proof, it is sufficient to consider the last rule of the derivation of $M\ \mathsf{R}\ M'$, for any $\mathsf{R} \in \{\overset{h}{\to}_{\beta_v}, \overset{h}{\to}_{\sigma}\}$.
3. If $M \overset{int}{\Rightarrow} M'$ and $N \Rightarrow N'$ where $M' \notin \Lambda_v$, then $MN \overset{int}{\Rightarrow} M'N'$: indeed, the last rule in the derivation of $M \overset{int}{\Rightarrow} M'$ can be neither λ nor *var* because $M' \notin \Lambda_v$. The hypothesis $M' \notin \Lambda_v$ is crucial: for example, $x \overset{int}{\Rightarrow} x$ and $I\Delta \Rightarrow \Delta$ but $I\Delta \not\overset{int}{\Rightarrow} \Delta$ and thus $x(I\Delta) \not\overset{int}{\Rightarrow} x\Delta$.
4. $\to_{\mathsf{v}} \subseteq \Rightarrow \subseteq \to_{\mathsf{v}}^*$. As a consequence, $\Rightarrow^* = \to_{\mathsf{v}}^*$ and (by Proposition 4) $\Rightarrow$ is confluent.
5. $\overset{int}{\to}_{\mathsf{v}} \subseteq \overset{int}{\Rightarrow} \subseteq \overset{int}{\to}_{\mathsf{v}}^*$, so $\overset{int}{\Rightarrow}^* = \overset{int}{\to}_{\mathsf{v}}^*$. Thus, by Remark 11.3, variables and abstractions are preserved under $\overset{int}{\to}_{\mathsf{v}}^*$-expansion, i.e., if $M \overset{int}{\to}_{\mathsf{v}}^* x$ (resp. $M \overset{int}{\to}_{\mathsf{v}}^* \lambda x.N'$) then $M = x$ (resp. $M = \lambda x.N$ with $N \to_{\mathsf{v}}^* N'$).
6. For any $\mathsf{R} \in \{\overset{h}{\to}_{\beta_v}, \overset{h}{\to}_{\sigma}\}$, if $M\ \mathsf{R}\ M'$ then $M\{V/x\}\ \mathsf{R}\ M'\{V/x\}$ for any $V \in \Lambda_v$. The proof is by straightforward induction on the derivation of $M\ \mathsf{R}\ M'$ for any $\mathsf{R} \in \{\overset{h}{\to}_{\beta_v}, \overset{h}{\to}_{\sigma}\}$.

As expected, a basic property of parallel reduction $\Rightarrow$ is the following:

▶ **Lemma 13** (Substitution lemma for $\Rightarrow$). *If $M \Rightarrow M'$ and $V \Rightarrow V'$ then $M\{V/x\} \Rightarrow M'\{V'/x\}$.*

Proof. By straightforward induction on the derivation of $M \Rightarrow M'$. ◀

The following lemma will play a crucial role in the proof of Lemmas 18-19 and shows that the head σ-reduction $\overset{h}{\to}_{\sigma}$ can be postponed after the head β_v-reduction $\overset{h}{\to}_{\beta_v}$.

▶ **Lemma 14** (Commutation of head reductions).
1. *If* $M \overset{h}{\to}_\sigma L \overset{h}{\to}_{\beta_v} N$ *then there exists* $L' \in \Lambda$ *such that* $M \overset{h}{\to}_{\beta_v} L' \overset{h}{\to}^{=}_\sigma N$.
2. *If* $M \overset{h}{\to}^*_\sigma L \overset{h}{\to}^*_{\beta_v} N$ *then there exists* $L' \in \Lambda$ *such that* $M \overset{h}{\to}^*_{\beta_v} L' \overset{h}{\to}^*_\sigma N$.
3. *If* $M \overset{h}{\to}^*_{\mathsf{v}} M'$ *then there exists* $N \in \Lambda$ *such that* $M \overset{h}{\to}^*_{\beta_v} N \overset{h}{\to}^*_\sigma M'$.

Proof. 1. By induction on the derivation of $M \overset{h}{\to}_\sigma L$. Let us consider its last rule r.
If $\mathsf{r} = \sigma_1$ then $M = (\lambda x.M_0)N_0L_0M_1 \dots M_m$ and $L = (\lambda x.M_0L_0)N_0M_1 \dots M_m$ where $m \in \mathbb{N}$ and $x \notin \mathsf{fv}(L_0)$. Since $L \overset{h}{\to}_{\beta_v} N$, there are only two cases:
- either $N_0 \overset{h}{\to}_{\beta_v} N_0'$ and $N = (\lambda x.M_0L_0)N_0'M_1 \dots M_m$ (according to the rule *right* for $\overset{h}{\to}_{\beta_v}$), then $M \overset{h}{\to}_{\beta_v} (\lambda x.M_0)N_0'L_0M_1 \dots M_m \overset{h}{\to}_\sigma N$;
- or $N_0 \in \Lambda_v$ and $N = M_0\{N_0/x\}L_0M_1 \dots M_m$ (by the rule β_v, as $x \notin \mathsf{fv}(L_0)$), so $M \overset{h}{\to}_{\beta_v} N$.

If $\mathsf{r} = \sigma_3$ then $M = V((\lambda x.L_0)N_0)M_1 \dots M_m$ and $L = (\lambda x.VL_0)N_0M_1 \dots M_m$ with $m \in \mathbb{N}$ and $x \notin \mathsf{fv}(V)$. Since $L \overset{h}{\to}_{\beta_v} N$, there are only two cases:
- either $N_0 \overset{h}{\to}_{\beta_v} N_0'$ and $N = (\lambda x.VL_0)N_0'M_1 \dots M_m$ (according to the rule *right* for $\overset{h}{\to}_{\beta_v}$), then $M \overset{h}{\to}_{\beta_v} V((\lambda x.L_0)N_0')M_1 \dots M_m \overset{h}{\to}_\sigma N$;
- or $N_0 \in \Lambda_v$ and $N = VL_0\{N_0/x\}M_1 \dots M_m$ (by the rule β_v, as $x \notin \mathsf{fv}(V)$), so $M \overset{h}{\to}_{\beta_v} N$.

Finally, if $\mathsf{r} = \mathit{right}$ then $M = VN_0M_1 \dots M_m$ and $L = VN_0'M_1 \dots M_m$ with $m \in \mathbb{N}$ and $N_0 \overset{h}{\to}_\sigma N_0'$. By Remark 11.2, $N_0' \notin \Lambda_v$ and thus, since $L \overset{h}{\to}_{\beta_v} N$, the only possibility is that $N_0' \overset{h}{\to}_{\beta_v} N_0''$ and $N = VN_0''M_1 \dots M_m$ (according to the rule *right* for $\overset{h}{\to}_{\beta_v}$). By induction hypothesis, there exists $N_0''' \in \Lambda$ such that $N_0 \overset{h}{\to}_{\beta_v} N_0''' \overset{h}{\to}^{=}_\sigma N_0''$. Therefore, $M \overset{h}{\to}_{\beta_v} VN_0'''M_1 \dots M_m \overset{h}{\to}^{=}_\sigma N$.

2. Immediate consequence of Lemma 14.1, using standard techniques of rewriting theory.

3. Immediate consequence of Lemma 14.2, using standard techniques of rewriting theory. ◀

We are now able to travel over again Takahashi's method [17, 4] in our setting with β_v- and σ-reduction. The next four lemmas govern the strong parallel reduction and will be used to prove Lemma 19.

▶ **Lemma 15.** *If* $M \Rrightarrow M'$ *and* $N \Rightarrow N'$ *and* $M' \notin \Lambda_v$ *, then* $MN \Rrightarrow M'N'$.

Proof. One has $MN \Rightarrow M'N'$ by Remark 12.1 and since $M \Rightarrow M'$. By hypothesis, there exist $m, n \in \mathbb{N}$ and $M_0, \dots, M_m, N_0, \dots, N_n$ such that $M = M_0$, $M_m = N_0$, $N_n \overset{int}{\Rightarrow} M'$, $M_i \overset{h}{\to}_{\beta_v} M_{i+1}$ for any $0 \le i < m$ and $N_j \overset{h}{\to}_\sigma N_{j+1}$ for any $0 \le j < n$; by Remark 12.2, $M_iN \overset{h}{\to}_{\beta_v} M_{i+1}N$ for any $0 \le i < m$ and $N_jN \overset{h}{\to}_\sigma N_{j+1}N$ for any $0 \le j < n$. As $M' \notin \Lambda_v$, one has $N_nN \overset{int}{\Rightarrow} M'N'$ by Remark 12.3. Therefore, $MN \Rrightarrow M'N'$. ◀

▶ **Lemma 16.** *If* $M \Rrightarrow M'$ *and* $N \Rrightarrow N'$ *then* $MN \Rrightarrow M'N'$.

Proof. If $M' \notin \Lambda_v$ then $MN \Rrightarrow M'N'$ by Lemma 15 and since $N \Rightarrow N'$.

Assume $M' \in \Lambda_v$: $MN \Rightarrow M'N'$ by Remark 12.1, as $M \Rightarrow M'$ and $N \Rightarrow N'$. By hypothesis, there are $m, m', n, n' \in \mathbb{N}$ and $M_0, \dots, M_m, M_0', \dots, M_{m'}', N_0, \dots, N_n, N_0', \dots, N_{n'}'$ such that:
- $M = M_0$, $M_m = M_0'$, $M_{m'}' \overset{int}{\Rightarrow} M'$, $M_i \overset{h}{\to}_{\beta_v} M_{i+1}$ for any $0 \le i < m$, and $M_{i'}' \overset{h}{\to}_\sigma M_{i'+1}'$ for any $0 \le i' < m'$,
- $N = N_0$, $N_n = N_0'$, $N_{n'}' \overset{int}{\Rightarrow} N'$, $N_j \overset{h}{\to}_{\beta_v} N_{j+1}$ for any $0 \le j < n$ and $N_{j'}' \overset{h}{\to}_\sigma N_{j'+1}'$ for any $0 \le j' < n'$.

By Remark 11.3, $M_{m'}' \in \Lambda_v$ since $M' \in \Lambda_v$, therefore $m' = 0$ by Remark 11.2, and thus $M_m = M_0' \overset{int}{\Rightarrow} M'$ (and $M_m \Rightarrow M'$ since $\overset{int}{\Rightarrow} \subseteq \Rightarrow$) and $M_m \in \Lambda_v$. Using the rules *right* for $\overset{h}{\to}_{\beta_v}$ and $\overset{h}{\to}_\sigma$, one has $M_mN_j \overset{h}{\to}_{\beta_v} M_mN_{j+1}$ for any $0 \le j < n$, and $M_mN_{j'}' \overset{h}{\to}_\sigma M_mN_{j'+1}'$ for any $0 \le j' < n'$. By Remark 12.2, $M_iN_0 \overset{h}{\to}_{\beta_v} M_{i+1}N_0$ for any $0 \le i < m$. By applying

the rule *right* for $\overset{int}{\Rightarrow}$, one has $M_m N'_{n'} \overset{int}{\Rightarrow} M'N'$. Therefore, $MN = M_0N_0 \overset{h}{\to}_{\beta_v}^{*} M_mN_0 \overset{h}{\to}_{\beta_v}^{*} M_mN_n = M_mN'_0 \overset{h}{\to}_{\sigma}^{*} M_mN'_{n'} \overset{int}{\Rightarrow} M'N'$ and hence $MN \Rrightarrow M'N'$. ◀

▶ **Lemma 17.** *If $M \overset{int}{\Rightarrow} M'$ and $V \Rrightarrow V'$, then $M\{V/x\} \Rrightarrow M'\{V'/x\}$.*

Proof. By Lemma 13, one has $M\{V/x\} \Rightarrow M'\{V'/x\}$ since $M \Rightarrow M'$ and $V \Rightarrow V'$. We proceed by induction on $M \in \Lambda$. Let us consider the last rule r of the derivation of $M \overset{int}{\Rightarrow} M'$.

If $\mathsf{r} = \mathit{var}$ then there are two cases: either $M = x$ and then $M\{V/x\} = V \Rrightarrow V' = M'\{V'/x\}$; or $M = y \neq x$ and then $M\{V/x\} = y = M'\{V'/x\}$, so $M\{V/x\} \Rrightarrow M'\{V'/x\}$ by Remark 10.

If $\mathsf{r} = \lambda$ then $M = \lambda y.N$ and $M' = \lambda y.N'$ with $N \Rightarrow N'$; we can suppose without loss of generality that $y \notin \mathsf{fv}(V) \cup \{x\}$. One has $N\{V/x\} \Rightarrow N'\{V'/x\}$ according to Lemma 13. By applying the rule λ for $\overset{int}{\Rightarrow}$, one has $M\{V/x\} = \lambda y.N\{V/x\} \overset{int}{\Rightarrow} \lambda y.N'\{V'/x\} = M'\{V'/x\}$ and thus $M\{V/x\} \Rrightarrow M'\{V'/x\}$.

Finally, if $\mathsf{r} = \mathit{right}$ then $M = UNM_1 \dots M_m$ and $M' = U'N'M'_1 \dots M'_m$ for some $m \in \mathbb{N}$ with $U, U' \in \Lambda_v$, $U \Rightarrow U'$, $N \overset{int}{\Rightarrow} N'$ and $M_i \Rightarrow M'_i$ for any $1 \leq i \leq m$. By induction hypothesis, $U\{V/x\} \Rrightarrow U'\{V'/x\}$ (indeed $U \overset{int}{\Rightarrow} U'$ according to Remark 11.5) and $N\{V/x\} \Rrightarrow N'\{V'/x\}$. By Lemma 13, $M_i\{V/x\} \Rightarrow M'_i\{V'/x\}$ for any $1 \leq i \leq m$. By Lemma 16, $U\{V/x\}N\{V/x\} \Rrightarrow U'\{V'/x\}N'\{V/x\}$ and hence, by applying Lemma 15 m times since $U'\{V'/x\}N'\{V/x\} \notin \Lambda_v$, one has $M\{V/x\} = U\{V/x\}N\{V/x\}M_1\{V/x\} \dots M_m\{V/x\} \Rrightarrow U'\{V'/x\}N'\{V'/x\}M'_1\{V'/x\} \dots M'_m\{V'/x\} = M'\{V'/x\}$. ◀

In the proof of the two next lemmas, as well as in the proof of Corollary 21 and Theorem 22, our Lemma 14 plays a crucial role: indeed, since the head σ-reduction well interact with the head β_v-reduction, Takahashi's method [17, 4] is still working when adding the reduction rules σ_1 and σ_3 to Plotkin's β_v-reduction.

▶ **Lemma 18** (Substitution lemma for $\Rrightarrow$). *If $M \Rrightarrow M'$ and $V \Rrightarrow V'$ then $M\{V/x\} \Rrightarrow M'\{V'/x\}$.*

Proof. By Lemma 13, one has $M\{V/x\} \Rightarrow M'\{V'/x\}$ since $M \Rightarrow M'$ and $V \Rightarrow V'$. By hypothesis, there exist $m, n \in \mathbb{N}$ and $M_0, \dots, M_m, N_0, \dots, N_n$ such that $M = M_0$, $M_m = N_0$, $N_n \overset{int}{\Rightarrow} M'$, $M_i \overset{h}{\to}_{\beta_v} M_{i+1}$ for any $0 \leq i < m$ and $N_j \overset{h}{\to}_{\sigma} N_{j+1}$ for any $0 \leq j < n$; by Remark 12.6, $M_i\{V/x\} \overset{h}{\to}_{\beta_v} M_{i+1}\{V/x\}$ for any $0 \leq i < m$, and $N_j\{V/x\} \overset{h}{\to}_{\sigma} N_{j+1}\{V/x\}$ for any $0 \leq j < n$. By Lemma 17, one has $N_n\{V/x\} \Rrightarrow M'\{V'/x\}$, thus there exist $L, N \in \Lambda$ such that $M\{V/x\} \overset{h}{\to}_{\beta_v}^{*} N_0\{V/x\} \overset{h}{\to}_{\sigma}^{*} N_n\{V/x\} \overset{h}{\to}_{\beta_v}^{*} N \overset{h}{\to}_{\sigma}^{*} L \overset{int}{\Rightarrow} M'\{V'/x\}$. By Lemma 14.2, there exists $N' \in \Lambda$ such that $M\{V/x\} \overset{h}{\to}_{\beta_v}^{*} N_0\{V/x\} \overset{h}{\to}_{\beta_v}^{*} N' \overset{h}{\to}_{\sigma}^{*} N \overset{h}{\to}_{\sigma}^{*} L \overset{int}{\Rightarrow} M'\{V'/x\}$, therefore $M\{V/x\} \Rrightarrow M'\{V'/x\}$. ◀

Now we are ready to prove a key lemma, which states that parallel reduction $\Rightarrow$ coincides with strong parallel reduction $\Rrightarrow$ (the inclusion $\Rrightarrow \subseteq \Rightarrow$ is trivial).

▶ **Lemma 19** (Key Lemma). *If $M \Rightarrow M'$ then $M \Rrightarrow M'$.*

Proof. By induction on the derivation of $M \Rightarrow M'$. Let us consider its last rule r.

If $\mathsf{r} = \mathit{var}$ then $M = x\,M_1 \dots M_m$ and $M' = x\,M'_1 \dots M'_m$ where $m \in \mathbb{N}$ and $M_i \Rightarrow M'_i$ for any $1 \leq i \leq m$. By reflexivity of $\Rrightarrow$ (Remark 10), $x \Rrightarrow x$. By induction hypothesis, $M_i \Rrightarrow M'_i$ for any $1 \leq i \leq m$. Therefore, $M \Rrightarrow M'$ by applying Lemma 16 m times.

If $\mathsf{r} = \lambda$ then $M = (\lambda x.M_0)M_1 \dots M_m$ and $M' = (\lambda x.M'_0)M'_1 \dots M'_m$ where $m \in \mathbb{N}$ and $M_i \Rightarrow M'_i$ for any $0 \leq i \leq m$. By induction hypothesis, $M_i \Rrightarrow M'_i$ for any $1 \leq i \leq m$. According to Remark 11.4, $\lambda x.M_0 \Rrightarrow \lambda x.M'_0$. So $M \Rrightarrow M'$ by applying Lemma 16 m times.

If $\mathsf{r} = \beta_v$ then $M = (\lambda x.M_0)VM_1 \dots M_m$ and $M' = M'_0\{V'/x\}M'_1 \dots M'_m$ where $m \in \mathbb{N}$, $V \Rightarrow V'$ and $M_i \Rightarrow M'_i$ for any $0 \leq i \leq m$. By induction hypothesis, $V \Rrightarrow V'$ and $M_i \Rrightarrow M'_i$

for any $0 \leq i \leq m$. By applying the rule β_v for $\overset{h}{\to}_{\beta_v}$, one has $M \overset{h}{\to}_{\beta_v} M_0\{V/x\}M_1 \dots M_m$; moreover $M_0\{V/x\}M_1 \dots M_m \Rrightarrow M'$ by Lemma 18 and by applying Lemma 16 m times, thus there are $L, N \in \Lambda$ such that $M \overset{h}{\to}_{\beta_v} M_0\{V/x\}M_1 \dots M_m \overset{h}{\to}^*_{\beta_v} L \overset{h}{\to}^*_{\sigma} N \overset{int}{\Rightarrow} M'$. So $M \Rrightarrow M'$.

If $\mathsf{r} = \sigma_1$ then $M = (\lambda x.M_0)N_0L_0M_1 \dots M_m$ and $M' = (\lambda x.M'_0L'_0)N'_0M'_1 \dots M'_m$ where $m \in \mathbb{N}$, $L_0 \Rightarrow L'_0$, $N_0 \Rightarrow N'_0$ and $M_i \Rightarrow M'_i$ for any $0 \leq i \leq m$. By induction hypothesis, $N_0 \Rrightarrow N'_0$ and $M_i \Rrightarrow M'_i$ for any $1 \leq i \leq m$. By applying the rule σ_1 for $\overset{h}{\to}_{\sigma}$, one has $M \overset{h}{\to}_{\sigma} (\lambda x.M_0L_0)N_0M_1 \dots M_m$. By Remark 12.1, $M_0L_0 \Rightarrow M'_0L'_0$ and thus $\lambda x.M_0L_0 \Rrightarrow \lambda x.M'_0L'_0$ according to Remark 11.4. So $(\lambda x.M_0L_0)N_0M_1 \dots M_m \Rrightarrow M'$ by applying Lemma 16 $m+1$ times, hence there are $L, N \in \Lambda$ such that $M \overset{h}{\to}_{\sigma} (\lambda x.M_0L_0)N_0M_1 \dots M_m \overset{h}{\to}^*_{\beta_v} L \overset{h}{\to}^*_{\sigma} N \overset{int}{\Rightarrow} M'$. By Lemma 14.2, there is $L' \in \Lambda$ such that $M \overset{h}{\to}^*_{\beta_v} L' \overset{h}{\to}^*_{\sigma} L \overset{h}{\to}^*_{\sigma} N \overset{int}{\Rightarrow} M'$ and thus $M \Rrightarrow M'$.

Finally, if $\mathsf{r} = \sigma_3$ then $M = V((\lambda x.L_0)N_0)M_1 \dots M_m$ and $M' = (\lambda x.V'L'_0)N'_0M'_1 \dots M'_m$ with $m \in \mathbb{N}$, $V \Rightarrow V'$, $L_0 \Rightarrow L'_0$, $N_0 \Rightarrow N'_0$ and $M_i \Rightarrow M'_i$ for any $1 \leq i \leq m$. By induction hypothesis, $N_0 \Rrightarrow N'_0$ and $M_i \Rrightarrow M'_i$ for any $1 \leq i \leq m$. By the rule σ_3 for $\overset{h}{\to}_{\sigma}$, one has $M \overset{h}{\to}_{\sigma} (\lambda x.VL_0)N_0M_1 \dots M_m$. By Remark 12.1, $VL_0 \Rightarrow V'L'_0$ and thus $\lambda x.VL_0 \Rrightarrow \lambda x.V'L'_0$ according to Remark 11.4. So $(\lambda x.VL_0)N_0M_1 \dots M_m \Rrightarrow M'$ by applying Lemma 16 $m+1$ times, hence there are $L, N \in \Lambda$ such that $M \overset{h}{\to}_{\sigma} (\lambda x.VL_0)N_0M_1 \dots M_m \overset{h}{\to}^*_{\beta_v} L \overset{h}{\to}^*_{\sigma} N \overset{int}{\Rightarrow} M'$. By Lemma 14.2, there is $L' \in \Lambda$ such that $M \overset{h}{\to}^*_{\beta_v} L' \overset{h}{\to}^*_{\sigma} L \overset{h}{\to}^*_{\sigma} N \overset{int}{\Rightarrow} M'$, so $M \Rrightarrow M'$. ◀

Next Lemma 20 and Corollary 21 show that internal parallel reduction can be shifted after head reductions.

▶ **Lemma 20** (Postponement). *If $M \overset{int}{\Rightarrow} L$ and $L \overset{h}{\to}_{\beta_v} N$ (resp. $L \overset{h}{\to}_{\sigma} N$) then there exists $L' \in \Lambda$ such that $M \overset{h}{\to}_{\beta_v} L'$ (resp. $M \overset{h}{\to}_{\sigma} L'$) and $L' \Rightarrow N$.*

Proof. By induction on the derivation of $M \overset{int}{\Rightarrow} L$. Let us consider its last rule r.

If $\mathsf{r} = \mathit{var}$, then $M = x = L$ which contradicts $L \overset{h}{\to}_{\beta_v} N$ and $L \overset{h}{\to}_{\sigma} N$ by Remarks 11.1-2.

If $\mathsf{r} = \lambda$ then $L = \lambda x.L'$ for some $L' \in \Lambda$, which contradicts $L \overset{h}{\to}_{\beta_v} N$ and $L \overset{h}{\to}_{\sigma} N$ by Remarks 11.1-2.

Finally, if $\mathsf{r} = \mathit{right}$ then $M = VM_0M_1 \dots M_m$ and $L = V'L_0L_1 \dots L_m$ where $m \in \mathbb{N}$, $V \Rightarrow V'$ (so $V \overset{int}{\Rightarrow} V'$ by Remark 11.5), $M_0 \overset{int}{\Rightarrow} L_0$ (thus $M_0 \Rightarrow L_0$ since $\overset{int}{\Rightarrow} \subseteq \Rightarrow$) and $M_i \Rightarrow L_i$ for any $1 \leq i \leq m$.

- If $L \overset{h}{\to}_{\beta_v} N$ then there are two cases, depending on the last rule r' of the derivation of $L \overset{h}{\to}_{\beta_v} N$.
  - If $\mathsf{r}' = \beta_v$ then $V' = \lambda x.N'_0$, $L_0 \in \Lambda_v$ and $N = N'_0\{L_0/x\}L_1 \dots L_m$, thus $M_0 \in \Lambda_v$ and $V = \lambda x.N_0$ with $N_0 \Rightarrow N'_0$ by Remark 11.3. By Lemma 13, one has $N_0\{M_0/x\} \Rightarrow N'_0\{L_0/x\}$. Let $L' = N_0\{M_0/x\}M_1 \dots M_m$: so $M = (\lambda x.N_0)M_0M_1 \dots M_m \overset{h}{\to}_{\beta_v} L'$ (apply the rule β_v for $\overset{h}{\to}_{\beta_v}$) and $L' \Rightarrow N$ by applying Remark 12.1 m times.
  - If $\mathsf{r}' = \mathit{right}$ then $N = V'N_0L_1 \dots L_m$ with $L_0 \overset{h}{\to}_{\beta_v} N_0$. By induction hypothesis, there exists $L'_0 \in \Lambda$ such that $M_0 \overset{h}{\to}_{\beta_v} L'_0 \Rightarrow N_0$. Let $L' = VL'_0M_1 \dots M_m$: so $M \overset{h}{\to}_{\beta_v} L'$ (apply the rule *right* for $\overset{h}{\to}_{\beta_v}$) and $L' \Rightarrow N$ by applying Remark 12.1 $m+1$ times.
- If $L \overset{h}{\to}_{\sigma} N$ then there are three cases, depending on the last rule r' of the derivation of $L \overset{h}{\to}_{\sigma} N$.
  - If $\mathsf{r}' = \sigma_1$ then $m > 0$, $V' = \lambda x.N'_0$ and $N = (\lambda x.N'_0L_1)L_0L_2 \dots L_m$, thus $V = \lambda x.N_0$ with $N_0 \Rightarrow N'_0$ by Remark 11.3. Using Remarks 12.1 and 11.4, one has $\lambda x.N_0M_1 \Rightarrow \lambda x.N'_0L_1$. Let $L' = (\lambda x.N_0M_1)M_0M_2 \dots M_m$: so $M = (\lambda x.N_0)M_0M_1 \dots M_m \overset{h}{\to}_{\sigma} L'$ (apply the rule σ_1 for $\overset{h}{\to}_{\sigma}$) and $L' \Rightarrow N$ by applying Remark 12.1 m times.
  - If $\mathsf{r}' = \sigma_3$ then $L_0 = (\lambda x.L_{01})L_{02}$ and $N = (\lambda x.V'L_{01})L_{02}L_1 \dots L_m$. Since $M_0 \overset{int}{\Rightarrow} (\lambda x.L_{01})L_{02}$, necessarily $M_0 = (\lambda x.M_{01})M_{02}$ with $M_{01} \Rightarrow L_{01}$ and $M_{02} \overset{int}{\Rightarrow} L_{02}$ (so $M_{02} \Rightarrow L_{02}$). Using Remarks 12.1 and 11.4, one has $\lambda x.VM_{01} \Rightarrow \lambda x.V'L_{01}$. Let

$L' = (\lambda x.VM_{01})M_{02}M_1 \dots M_m$: therefore $M = V((\lambda x.M_{01})M_{02})M_1 \dots M_m \overset{h}{\to}_\sigma L'$ (apply the rule σ_3 for $\overset{h}{\to}_\sigma$) and $L' \Rightarrow N$ by applying Remark 12.1 $m+1$ times.

- If $\mathsf{r}' = \mathit{right}$ then $N = V'N_0L_1 \dots L_m$ with $L_0 \overset{h}{\to}_\sigma N_0$. By induction hypothesis, there exists $L_0' \in \Lambda$ such that $M_0 \overset{h}{\to}_\sigma L_0' \Rightarrow N_0$. Let $L' = VL_0'M_1 \dots M_m$: so $M \overset{h}{\to}_\sigma L'$ (apply the rule *right* for $\overset{h}{\to}_\sigma$) and $L' \Rightarrow N$ by applying Remark 12.1 $m+1$ times.

◀

▶ **Corollary 21.** *If $M \overset{int}{\Rightarrow} L$ and $L \overset{h}{\to}_{\beta_v} N$ (resp. $L \overset{h}{\to}_\sigma N$), then there exist $L', L'' \in \Lambda$ such that $M \overset{h}{\to}^+_{\beta_v} L' \overset{h}{\to}^*_\sigma L'' \overset{int}{\Rightarrow} N$ (resp. $M \overset{h}{\to}^*_{\beta_v} L' \overset{h}{\to}^*_\sigma L'' \overset{int}{\Rightarrow} N$).*

Proof. Immediate by Lemma 20 and Lemma 19, applying Lemma 14.2 if $L \overset{h}{\to}_\sigma N$. ◀

Now we obtain our first main result (Theorem 22): any v-reduction sequence can be sequentialized into a head β_v-reduction sequence followed by a head σ-reduction sequence, followed by an internal reduction sequence. In ordinary λ-calculus, the well-known result corresponding to our Theorem 22 says that a β-reduction sequence can be factorized in a head reduction sequence followed by an internal reduction sequence (see for example [17, Corollary 2.6]).

▶ **Theorem 22** (Sequentialization)**.** *If $M \to_{\mathsf{v}}^* M'$ then there exist $L, N \in \Lambda$ such that $M \overset{h}{\to}^*_{\beta_v} L \overset{h}{\to}^*_\sigma N \overset{int}{\to}^*_{\mathsf{v}} M'$.*

Proof. By Remark 12.4, $M \Rightarrow^* M'$ and thus there are $m \in \mathbb{N}$ and $M_0, \dots, M_m \in \Lambda$ such that $M = M_0$, $M_m = M'$ and $M_i \Rightarrow M_{i+1}$ for any $0 \leq i < m$. We prove by induction on $m \in \mathbb{N}$ that there are $L, N \in \Lambda$ such that $M \overset{h}{\to}^*_{\beta_v} L \overset{h}{\to}^*_\sigma N \overset{int}{\Rightarrow}^* M'$, so $N \overset{int}{\to}^*_{\mathsf{v}} M'$ by Remark 12.5.

If $m = 0$ then $M = M_0 = M'$ and hence we conclude by taking $L = M' = N$.

Finally, suppose $m > 0$. By induction hypothesis applied to $M_1 \Rightarrow^* M'$, there exist $L', N' \in \Lambda$ such that $M_1 \overset{h}{\to}^*_{\beta_v} L' \overset{h}{\to}^*_\sigma N' \overset{int}{\Rightarrow}^* M'$. By applying Lemma 19 to M, there exist $L_0, N_0 \in \Lambda$ such that $M \overset{h}{\to}^*_{\beta_v} L_0 \overset{h}{\to}^*_\sigma N_0 \overset{int}{\Rightarrow} M_1$. By applying Corollary 21 repeatedly, there exists $N \in \Lambda$ such that $N_0 \overset{h}{\to}^*_{\mathsf{v}} N \overset{int}{\Rightarrow} N'$ and hence $M \overset{h}{\to}^*_{\mathsf{v}} N \overset{int}{\Rightarrow}^* M'$. According to Lemma 14.3, there exists $L \in \Lambda$ such that $M \overset{h}{\to}^*_{\beta_v} L \overset{h}{\to}^*_\sigma N \overset{int}{\Rightarrow}^* M'$. ◀

It is worth noticing that in Definition 7 there is no distinction between head σ_1- and head σ_3-reduction steps, and, according to it, the sequentialization of Theorem 22 imposes no order between head σ-reductions. We denote by $\overset{h}{\to}_{\sigma_1}$ and $\overset{h}{\to}_{\sigma_3}$ respectively the reduction relations $\to_{\sigma_1} \cap \overset{h}{\to}_\sigma$ and $\to_{\sigma_3} \cap \overset{h}{\to}_\sigma$. So, a natural question arises: is it possible to sequentialize them? The answer is negative, as proved by the next two counterexamples.

- Let $M = x((\lambda y.z')(zI))\Delta$ and $N = (\lambda y.xz'\Delta)(zI)$: $M \overset{h}{\to}_{\sigma_3} (\lambda y.xz')(zI)\Delta \overset{h}{\to}_{\sigma_1} N$, but there exists no L such that $M \overset{h}{\to}^*_{\sigma_1} L \overset{h}{\to}^*_{\sigma_3} N$. In fact M contains only a head σ_3-redex and $(\lambda y.xz')(zI)\Delta$ contains only a head σ_1-redex
- Let $M = x((\lambda y.z')(zI)\Delta)$ and $N = (\lambda y.x(z'\Delta))(zI)$: $M \overset{h}{\to}_{\sigma_1} x((\lambda y.z'\Delta)(zI)) \overset{h}{\to}_{\sigma_3} N$ but there is no L such that $M \overset{h}{\to}^*_{\sigma_3} L \overset{h}{\to}^*_{\sigma_1} N$. In fact M contains only a head σ_1-redex and $x((\lambda y.z'\Delta)(zI))$ contains only a head σ_3-redex.

So, the impossibility of sequentializing a head σ-reduction sequence is due to the fact that a head σ_1-reduction step can create a head σ_3-redex, and viceversa. This is not a problem, since head σ-reduction is strongly normalizing (by Proposition 4 and since $\overset{h}{\to}_\sigma \subseteq \to_\sigma$). Our approach does not force a strict order of head σ-reductions.

4 Standardization

Now we are able to prove the main result of this paper, i.e., a standardization theorem for λ_v^σ (Theorem 25). In particular we provide a notion of standard reduction sequence that avoids any auxiliary notion of residual redexes, by closely following the definition given in [15].

▶ **Notation.** For any $k, m \in \mathbb{N}$ with $k \leq m$, we denote by $\lceil M_0, \cdots, M_k, \cdots M_m \rceil^{head}$ a sequence of terms such that $M_i \overset{h}{\to}_{\beta_v} M_{i+1}$ when $0 \leq i < k$, and $M_i \overset{h}{\to}_{\sigma} M_{i+1}$ when $k \leq i < m$.

It is easy to check that $\lceil M \rceil^{head}$ for any $M \in \Lambda$. The notion of standard sequence of terms is defined by using the previous notion of *head*-sequence. Our notion of standard reduction sequence is mutually defined together with the notion of inner-sequence of terms (Definition 23). This definition allows us to avoid non-deterministic cases remarked in [7] (we provide more details at the end of this section). We denote by $\lceil M_0, \cdots, M_m \rceil^{std}$ (resp. $\lceil M_0, \cdots, M_m \rceil^{in}$) a standard (resp. inner) sequence of terms.

▶ **Definition 23** (Standard and inner sequences). *Standard* and *inner sequences* of terms are defined by mutual induction as follows:

1. if $\lceil M_0, \cdots, M_m \rceil^{head}$ and $\lceil M_m, \cdots, M_{m+n} \rceil^{in}$ then $\lceil M_0, \cdots, M_{m+n} \rceil^{std}$, where $m, n \in \mathbb{N}$;
2. $\lceil M \rceil^{in}$, for any $M \in \Lambda$;
3. if $\lceil M_0, \cdots, M_m \rceil^{std}$ then $\lceil \lambda z.M_0, \cdots, \lambda z.M_m \rceil^{in}$, where $m \in \mathbb{N}$;
4. if $\lceil V_0, \cdots, V_h \rceil^{std}$ and $\lceil N_0, \cdots, N_n \rceil^{in}$ then $\lceil V_0 N_0, \cdots, V_0 N_n, \cdots, V_h N_n \rceil^{in}$, where $h, n \in \mathbb{N}$;
5. if $\lceil N_0, \cdots, N_n \rceil^{in}$, $\lceil M_0, \cdots, M_m \rceil^{std}$ and $N_0 \notin \Lambda_v$, then $\lceil N_0 M_0, \cdots, N_n M_0, \cdots, N_n M_m \rceil^{in}$, where $m, n \in \mathbb{N}$.

For instance, let $M = (\lambda y.Ix)(z(\Delta I))(II)$: $M \to_{\mathsf{v}} (\lambda y.Ix)(z(\Delta I))I \to_{\mathsf{v}} (\lambda y.x)(z(\Delta I))I$ and $M \to_{\mathsf{v}} (\lambda y.Ix(II))(z(\Delta I)) \to_{\mathsf{v}} (\lambda y.Ix(II))(z(II))$ are not standard sequences; $M \to_{\mathsf{v}} (\lambda y.Ix)(z(\Delta I))I$ and $M \to_{\mathsf{v}} (\lambda y.Ix)(z(II))(II) \to_{\mathsf{v}} (\lambda y.Ix)(zI)(II) \to_{\mathsf{v}} (\lambda y.Ix(II))(zI) \to_{\mathsf{v}} (\lambda y.x(II))(zI) \to_{\mathsf{v}} (\lambda y.xI)(zI)$ are standard sequences.

▶ **Remark 24.** For any $n \in \mathbb{N}$, if $\lceil N_0, \cdots, N_n \rceil^{in}$ (resp. $\lceil N_0, \cdots, N_n \rceil^{head}$) then $\lceil N_0, \cdots, N_n \rceil^{std}$. Indeed, $\lceil N_0 \rceil^{head}$ (resp. $\lceil N_n \rceil^{in}$ by Definition 23.2), so $\lceil N_0, \cdots, N_n \rceil^{std}$ by Definition 23.1.

In particular, $\lceil N \rceil^{std}$ for any $N \in \Lambda$: apply Definition 23.2 and Remark 24 for $n = 0$.

▶ **Theorem 25** (Standardization).

1. *If* $M \to_{\mathsf{v}}^* M'$ *then there is a sequence* $\lceil M, \cdots, M' \rceil^{std}$.
2. *If* $M \overset{int}{\to}_{\mathsf{v}}^* M'$ *then there is a sequence* $\lceil M, \cdots, M' \rceil^{in}$.

Proof. Both statements are proved simultaneously by induction on $M' \in \Lambda$.

1. - If $M' = z$ then, by Theorem 22, $M \overset{h}{\to}_{\beta_v}^* L \overset{h}{\to}_{\sigma}^* N \overset{int}{\to}_{\mathsf{v}}^* z$ for some $L, N \in \Lambda$. By Remarks 12.5 and 11.2, $L = N = z$; therefore $M \overset{h}{\to}_{\beta_v}^* z$ and hence there is a sequence $\lceil M, \cdots, z \rceil^{head}$. Thus, $\lceil M, \cdots, z \rceil^{std}$ by Remark 24.
   - If $M' = \lambda z.N'$ then, by Theorem 22, $M \overset{h}{\to}_{\beta_v}^* L \overset{h}{\to}_{\sigma}^* L' \overset{int}{\to}_{\mathsf{v}}^* \lambda z.N'$ for some $L, L' \in \Lambda$. By Remarks 12.5 and 11.2, $L = L' = \lambda z.N$ with $N \to_{\mathsf{v}}^* N'$. So $M \overset{h}{\to}_{\beta_v}^* \lambda z.N$ and hence there is a sequence $\lceil M, \cdots, \lambda z.N \rceil^{head}$. By induction on (1), there is a sequence $\lceil N, \cdots, N' \rceil^{std}$, thus $\lceil \lambda z.N, \cdots, \lambda z.N' \rceil^{in}$ by Definition 23.3. Therefore $\lceil M, \cdots, \lambda z.N, \cdots, \lambda z.N' \rceil^{std}$ by Definition 23.1.
   - If $M' = N'L'$ then, by Theorem 22, $M \overset{h}{\to}_{\beta_v}^* M'' \overset{h}{\to}_{\sigma}^* M_0 \overset{int}{\to}_{\mathsf{v}}^* N'L'$ for some $M'', M_0 \in \Lambda$. By Remark 3, $M_0 = NL$ for some $N, L \in \Lambda$, since $\overset{int}{\to}_{\mathsf{v}}^* \subseteq \to_{\mathsf{v}}^*$ and $M' \notin \Lambda_v$. Thus there is a sequence $\lceil M, \cdots, M'', \cdots, NL \rceil^{head}$. By Remark 12.5, $NL \overset{int}{\Rightarrow}^* N'L'$; clearly, each step of $\overset{int}{\Rightarrow}$ is an instance of the rule *right* of Definition 9. There are two sub-cases.

- If $N \in \Lambda_v$ then $N \Rightarrow^* N'$ and $L \overset{int}{\Rightarrow}^* L'$, so $N \to_{\mathsf{v}}^* N'$ and $L \overset{int}{\to}_{\mathsf{v}}^* L'$ by Remarks 12.4-5. By induction respectively on (1) and (2), there are sequences $\lceil N, \cdots, N' \rceil^{std}$ and $\lceil L, \cdots, L' \rceil^{in}$, thus $\lceil NL, \cdots, NL', \cdots, N'L' \rceil^{in}$ by Definition 23.4. Therefore $\lceil M, \cdots, M'', \cdots, NL, \cdots, NL', \cdots, N'L' \rceil^{std}$ by Definition 23.1.
- If $N \notin \Lambda_v$ (i.e., $N = VM_1 \dots M_m$ with $m > 0$) then $N \overset{int}{\Rightarrow}^* N'$ and $L \Rightarrow^* L'$, so $N \overset{int}{\to}_{\mathsf{v}}^* N'$ and $L \to_{\mathsf{v}}^* L'$ by Remarks 12.4-5. By induction respectively on (2) and (1), there are sequences $\lceil N, \cdots, N' \rceil^{in}$ and $\lceil L, \cdots, L' \rceil^{std}$. Hence $\lceil NL, \cdots, N'L, \cdots, N'L' \rceil^{in}$ by Definition 23.5 . Thus $\lceil M, \cdots, M'', \cdots, NL, \cdots, N'L, \cdots, N'L' \rceil^{std}$ by Definition 23.1.

2. - If $M' = z$ then $M = z$ by Remark 12.5, hence $\lceil z \rceil^{in}$ by Definition 23.2.
   - If $M' = \lambda z.L'$ then $M = \lambda z.L$ and $L \to_{\mathsf{v}}^* L'$ by Remark 12.5. Hence there is a sequence $\lceil L, \cdots, L' \rceil^{std}$ by induction on (1). By Definition 23.3, $\lceil \lambda z.L, \cdots, \lambda z.L' \rceil^{in}$.
   - If $M' = N'L'$ then $M = NL$ for some $N, L \in \Lambda$ by Remark 3, since $\overset{int}{\to}_{\mathsf{v}}^* \subseteq \to_{\mathsf{v}}^*$ and $M' \notin \Lambda_v$. By Remark 12.5, $NL \overset{int}{\Rightarrow}^* N'L'$; clearly, each step of $\overset{int}{\Rightarrow}$ is an instance of the rule *right* of Definition 9. There are two sub-cases.
     - If $N \in \Lambda_v$ then $N \Rightarrow^* N'$ and $L \overset{int}{\Rightarrow}^* L'$, so $N \to_{\mathsf{v}}^* N'$ and $L \overset{int}{\to}_{\mathsf{v}}^* L'$ by Remarks 12.4-5. Thus there are sequences $\lceil N, \cdots, N' \rceil^{std}$ and $\lceil L, \cdots, L' \rceil^{in}$ by induction respectively on (1) and (2). Therefore, by Definition 23.4, $\lceil NL, \cdots, NL', \cdots, N'L' \rceil^{in}$.
     - If $N \notin \Lambda_v$ (i.e. $N = VM_1 \dots M_m$ with $m > 0$) then $N \overset{int}{\Rightarrow}^* N'$ and $L \Rightarrow^* L'$, thus $N \overset{int}{\to}_{\mathsf{v}}^* N'$ and $L \to_{\mathsf{v}}^* L'$ by Remarks 12.4-5. By induction respectively on (2) and (1), there are sequences $\lceil N, \cdots, N' \rceil^{in}$ and $\lceil L, \cdots, L' \rceil^{std}$. So $\lceil NL, \cdots, N'L, \cdots, N'L' \rceil^{in}$ by Definition 23.5. ◀

Due to non-sequentialization of head σ_1- and head σ_3-reductions, several standard sequences may have the same starting term and ending term: for instance, if $M = I(\Delta I)I$ and $N = (\lambda z.(\lambda x.xI)(zz))I$ then $M \to_{\mathsf{v}} (\lambda x.xI)(\Delta I) \to_{\mathsf{v}} N$ and $M \to_{\mathsf{v}} (\lambda z.I(zz))II \to_{\mathsf{v}} (\lambda z.I(zz)I)I \to_{\mathsf{v}} N$ are both standard sequences from M to N.

Finally, we can compare our notion of standardization with that given in [15]. To make the comparison possible we avoid σ-reductions and we recall that $\overset{h}{\to}_{\beta_v}$ is exactly the Plotkin's left-reduction [15, p. 136]. As remarked in [7, §1.5 p. 149], both sequences $(\lambda z.II)(II) \to_{\mathsf{v}} (\lambda z.I)(II) \to_{\mathsf{v}} (\lambda z.I)I$ and $(\lambda z.II)(II) \to_{\mathsf{v}} (\lambda z.II)I \to_{\mathsf{v}} (\lambda z.I)I$ are standard according to [15]. On the other hand, only the second sequence is standard in our sense. It is easy to check that collapsing the two notions of inner and standard sequence given in Definition 23, we get a notion of standard sequence that accept both the above sequences.

5 Some conservativity results

The sequentialization result (Theorem 22) has some interesting semantic consequences. It allows us to prove that (Corollary 29) the λ_v^σ-calculus is sound with respect to the call-by-value observational equivalence introduced by Plotkin in [15] for λ_v. Moreover we can prove that some notions, like that of potential valuability and solvability, introduced in [13] for λ_v, coincide with the respective notions for λ_v^σ (Theorem 31). This justifies the idea that λ_v^σ is a useful tool for studying the properties of λ_v. Our starting point is the following corollary.

▶ **Corollary 26.**

1. *If $M \to_{\mathsf{v}}^* V \in \Lambda_v$ then there exists $V' \in \Lambda_v$ such that $M \overset{h}{\to}_{\beta_v}^* V' \overset{int}{\to}_{\mathsf{v}}^* V$.*
2. *For every $V \in \Lambda_v$, $M \overset{h}{\to}_{\beta_v}^* V$ if and only if $M \overset{h}{\to}_{\mathsf{v}}^* V$.*

Proof. The first point is proved by observing that, by Theorem 22, there are $N, L \in \Lambda$ such that $M \overset{h}{\to}_{\beta_v}^* L \overset{h}{\to}_{\sigma}^* N \overset{int}{\to}_{\mathsf{v}}^* V$. By Remark 12.5 , $N \in \Lambda_v$ and thus $L = N$ according to

Remark 11.2. Concerning the second point, the right-to-left direction is a consequence of Lemma 14.3 and Remark 11.2; the left-to-right direction follows from $\overset{h}{\to}_{\beta_v} \subseteq \overset{h}{\to}_{\mathsf{v}}$. ◀

Let us recall the notion of observational equivalence defined by Plotkin [15] for λ_v.

▶ **Definition 27** (Halting, observational equivalence). Let $M \in \Lambda$. We say that (*the evaluation of*) M *halts* if there exists $V \in \Lambda_v$ such that $M \overset{h}{\to}^*_{\beta_v} V$.

The (*call-by-value*) *observational equivalence* is an equivalence relation $\cong$ on Λ defined by: $M \cong N$ if, for every context $\mathtt{C}$, one has that $\mathtt{C}\langle\!\langle M \rangle\!\rangle$ halts iff $\mathtt{C}\langle\!\langle N \rangle\!\rangle$ halts.[1]

Clearly, similar notions can be defined for λ^σ_v using $\overset{h}{\to}_{\mathsf{v}}$ instead of $\overset{h}{\to}_{\beta_v}$. Head σ-reduction plays no role neither in deciding the halting problem for evaluation (Corollary 26.1), nor in reaching a particular value (Corollary 26.2). So, we can conclude that the notions of halting and observational equivalence in λ^σ_v coincide with the ones in λ_v, respectively.

Now we compare the equational theory of λ^σ_v with Plotkin's observational equivalence.

▶ **Theorem 28** (Adequacy of v-reduction). *If $M \to^*_{\mathsf{v}} M'$ then: M halts iff M' halts.*

Proof. If $M' \overset{h}{\to}^*_{\beta_v} V \in \Lambda_v$ then $M \to^*_{\mathsf{v}} M' \to^*_{\mathsf{v}} V$ since $\overset{h}{\to}_{\beta_v} \subseteq \to_{\mathsf{v}}$. By Corollary 26.1, there exists $V' \in \Lambda_v$ such that $M \overset{h}{\to}^*_{\beta_v} V'$. Thus M halts.

Conversely, if $M \overset{h}{\to}^*_{\beta_v} V \in \Lambda_v$ then $M \to^*_{\mathsf{v}} V$ since $\overset{h}{\to}_{\beta_v} \subseteq \to_{\mathsf{v}}$. By confluence of $\to_{\mathsf{v}}$ (Proposition 4, since $M \to^*_{\mathsf{v}} M'$) and Remark 3 (since $V \in \Lambda_v$), there is $V' \in \Lambda_v$ such that $V \to^*_{\mathsf{v}} V'$ and $M' \to^*_{\mathsf{v}} V'$. By Corollary 26.1, there is $V'' \in \Lambda_v$ such that $M' \overset{h}{\to}^*_{\beta_v} V''$. So M' halts. ◀

▶ **Corollary 29** (Soundness). *If $M =_{\mathsf{v}} N$ then $M \cong N$.*

Proof. Let $\mathtt{C}$ be a context. By confluence of $\to_{\mathsf{v}}$ (Proposition 4), $M =_{\mathsf{v}} N$ implies that there exists $L \in \Lambda$ such that $M \to^*_{\mathsf{v}} L$ and $N \to^*_{\mathsf{v}} L$, hence $\mathtt{C}\langle\!\langle M \rangle\!\rangle \to^*_{\mathsf{v}} \mathtt{C}\langle\!\langle L \rangle\!\rangle$ and $\mathtt{C}\langle\!\langle N \rangle\!\rangle \to^*_{\mathsf{v}} \mathtt{C}\langle\!\langle L \rangle\!\rangle$. By Theorem 28, $\mathtt{C}\langle\!\langle M \rangle\!\rangle$ halts iff $\mathtt{C}\langle\!\langle L \rangle\!\rangle$ halts iff $\mathtt{C}\langle\!\langle N \rangle\!\rangle$ halts. Therefore, $M \cong N$. ◀

Plotkin [15, Theorem 5] has already proved that $M =_{\beta_v} N$ implies $M \cong N$, but our Corollary 29 is not obvious since our λ^σ_v-calculus equates more than Plotkin's λ_v-calculus ($=_{\beta_v} \subseteq =_{\mathsf{v}}$ since $\to_{\beta_v} \subseteq \to_{\mathsf{v}}$, and Example 5 shows that this inclusion is strict).

The converse of Corollary 29 does not hold since $\lambda x.x(\lambda y.xy) \cong \Delta$ but $\lambda x.x(\lambda y.xy)$ and Δ are different v-normal forms and hence $\lambda x.x(\lambda y.xy) \neq_{\mathsf{v}} \Delta$ by confluence of $\to_{\mathsf{v}}$ (Proposition 4).

A further remarkable consequence of Corollary 26.1 is that the notions of potential valuability and solvability for λ^σ_v-calculus (studied in [3]) can be shown to coincide with the ones for Plotkin's λ_v-calculus (studied in [13, 14]), respectively. Let us recall their definition.

▶ **Definition 30** (Potential valuability, solvability). Let M be a term:

- M is v-*potentially valuable* (resp. β_v-*potentially valuable*) if there are $m \in \mathbb{N}$, pairwise distinct variables $x_1, \dots, x_m$ and $V, V_1, \dots, V_m \in \Lambda_v$ such that $M\{V_1/x_1, \dots, V_m/x_m\} \to^*_{\mathsf{v}} V$ (resp. $M\{V_1/x_1, \dots, V_m/x_m\} \to^*_{\beta_v} V$);
- M is v-*solvable* (resp. β_v-*solvable*) if there are $n, m \in \mathbb{N}$, variables $x_1, \dots, x_m$ and $N_1, \dots, N_n \in \Lambda$ such that $(\lambda x_1 \dots x_m.M)N_1 \cdots N_n \to^*_{\mathsf{v}} I$ (resp. $(\lambda x_1 \dots x_m.M)N_1 \cdots N_n \to^*_{\beta_v} I$).

▶ **Theorem 31.** *Let M be a term:*

1. *M is v-potentially valuable if and only if M is β_v-potentially valuable;*
2. *M is v-solvable if and only if M is β_v-solvable.*

[1] Original Plotkin's definition of call-by-value observational equivalence (see [15]) also requires that $\mathtt{C}\langle\!\langle M \rangle\!\rangle$ and $\mathtt{C}\langle\!\langle N \rangle\!\rangle$ are closed terms, according to the tradition identifying programs with closed terms.

Proof. In both points, the implication from right to left is trivial since $\to_{\beta_v} \subseteq \to_{\mathsf{v}}$. Let us prove the other direction.

1. Since M is v-potentially valuable, there are variables $x_1, \ldots, x_m$ and $V, V_1, \ldots, V_m \in \Lambda_v$ (with $m \geq 0$) such that $M\{V_1/x_1, \ldots, V_m/x_m\} \to_{\mathsf{v}}^* V$; then, there exists $V' \in \Lambda_v$ such that $M\{V_1/x_1, \ldots, V_m/x_m\} \to_{\beta_v}^* V'$ by Corollary 26.1 and because $\overset{h}{\to}_{\beta_v} \subseteq \to_{\beta_v}$, therefore M is β_v-potentially valuable.
2. Since M is v-solvable, there exist variables $x_1, \ldots, x_m$ and terms $N_1, \ldots, N_n$ (for some $n, m \geq 0$) such that $(\lambda x_1 \ldots x_m.M)N_1 \cdots N_n \to_{\mathsf{v}}^* I$; then, there exists $V \in \Lambda_v$ such that $(\lambda x_1 \ldots x_m.M)N_1 \cdots N_n \to_{\beta_v}^* V \overset{int}{\to}_{\mathsf{v}}^* I$ by Corollary 26.1 and because $\overset{h}{\to}_{\beta_v} \subseteq \to_{\beta_v}$. According to Remark 12.5 , $V = \lambda x.N$ for some $N \in \Lambda$ such that $N \to_{\mathsf{v}}^* x$. By Corollary 26.1, there is $V' \in \Lambda_v$ such that $N \overset{h}{\to}_{\beta_v}^* V' \overset{int}{\to}_{\mathsf{v}}^* x$, hence $V' = x$ by Remark 12.5 again. Since $\overset{h}{\to}_{\beta_v} \subseteq \to_{\beta_v}$, $N \to_{\beta_v}^* x$ and thus $V = \lambda x.N \to_{\beta_v}^* I$, therefore M is β_v-solvable. ◀

So, due to Theorem 31, the semantic (via a relational model) and operational (via two sub-reductions of $\to_{\mathsf{v}}$) characterization of v-potential valuability and v-solvability given in [3, Theorems 24-25] is also a semantic and operational characterization of β_v-potential valuability and β_v-solvability. The difference is that in λ_v^σ these notions can be studied operationally inside the calculus, while it has been proved in [13, 14] that the β_v-reduction is too weak to characterize them: an operational characterization of β_v-potential valuability and β_v-solvability cannot be given inside λ_v. Hence, λ_v^σ is a useful, conservative and "complete" tool for studying semantic properties of λ_v.

6 Conclusions

In this paper we have proved a standardization theorem for the λ_v^σ-calculus introduced in [3]. The used technique is a notion of parallel reduction. Let us recall that parallel reduction in λ-calculus has been defined by Tait and Martin-Löf in order to prove confluence of the β-reduction, without referring to the difficult notion of residuals. Takahashi in [17] has simplified this technique and showed that it can be successfully applied to standardization. We would like to remark that our parallel reduction cannot be used to prove confluence of $\to_{\mathsf{v}}$. Indeed, take $M = (\lambda x.L)\big((\lambda y.N)((\lambda z.N')N'')\big)L'$, $M_1 = (\lambda x.LL')\big((\lambda y.N)((\lambda z.N')N'')\big)$ and $M_2 = \big((\lambda y.(\lambda x.L)N)((\lambda z.N')N'')\big)L'$: then $M \Rightarrow M_1$ and $M \Rightarrow M_2$ but there is no term M' such that $M_1 \Rightarrow M'$ and $M_2 \Rightarrow M'$. To sum up, $\Rightarrow$ does not enjoy the Diamond Property.

The standardization result allows us to formally verify the correctness of λ_v^σ with respect to the semantics of λ_v, so we can use λ_v^σ as a tool for studying properties of λ_v. This is a remarkable result: in fact some properties, like potential valuability and solvability, cannot be characterized in λ_v by means of β_v-reduction (as proved in [13, 14]), but they have a natural operational characterization in λ_v^σ (via two sub-reductions of $\to_{\mathsf{v}}$).

We plan to continue to explore the call-by-value computation, using λ_v^σ. As a first step, we would like to revisit and improve the Separability Theorem given in [11] for λ_v. Still the issue is more complex than in the call-by-name, indeed in ordinary λ-calculus different $\beta\eta$-normal forms can be separated (by the Böhm Theorem), while in λ_v there are different normal forms that cannot be separated, but which are only semi-separable (e.g. I and $\lambda z.(\lambda u.z)(zz)$). We hope to completely characterize separable and semi-separable normal forms in λ_v^σ. This should be a first step aimed to define a semantically meaningful notion of approximants. Then, we should be able to provide a new insight on the denotational analysis of the call-by-value, maybe overcoming limitations as that of the absence of fully abstract filter models [14, Theorem 12.1.25]. Last but not least, an unexplored but challenging

research direction is the use of commutation rules to improve the call-by-value evaluation. We do not have concrete evidence supporting such possibility, but since λ_v^σ is strongly related to the calculi presented in [7, 1], which are endowed with explicit substitutions, we are confident that a sharp use of commutations can have a relevant impact in the evaluation.

References

1 Beniamino Accattoli and Luca Paolini. Call-by-Value Solvability, Revisited. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, volume 7294 of *Lecture Notes in Computer Science*, pages 4–16. Springer-Verlag, 2012.

2 Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics*. North Holland, 1984.

3 Alberto Carraro and Giulio Guerrieri. A Semantical and Operational Account of Call-by-Value Solvability. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures*, volume 8412 of *Lecture Notes in Computer Science*, pages 103–118. Springer-Verlag, 2014.

4 Karl Crary. A Simple Proof of Call-by-Value Standardization. Technical Report CMU-CS-09-137, Carnegie Mellon University, 2009.

5 Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

6 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

7 Hugo Herbelin and Stéphane Zimmermann. An Operational Account of Call-by-Value Minimal and Classical lambda-Calculus in "Natural Deduction" Form. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 2009.

8 Roger Hindley. Standard and normal reductions. *Transactions of the American Mathematical Society*, pages 253–271, 1978.

9 John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 228(1–2):175–210, 1999.

10 Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Logic in Computer Science*, pages 14–23. IEEE Computer Society, 1989.

11 Luca Paolini. Call-by-Value Separability and Computability. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Italian Conference in Theoretical Computer Science*, volume 2202 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag, 2002.

12 Luca Paolini and Simona Ronchi Della Rocca. Parametric parameter passing lambda-calculus. *Information and Computation*, 189(1):87–106, 2004.

13 Luca Paolini and Simona Ronchi Della Rocca. Call-by-value Solvability. *Theoretical Informatics and Applications*, 33(6):507–534, 1999. RAIRO Series, EDP-Sciences.

14 Luca Paolini and Simona Ronchi Della Rocca. *The Parametric λ-Calculus: a Metamodel for Computation*. Texts in Theoretical Computer Science: An EATCS Series. Springer-Verlag, 2004.

15 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

16 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6(3-4):289–360, 1993.

17 Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995.

Wild ω-Categories for the Homotopy Hypothesis in Type Theory

André Hirschowitz[1], Tom Hirschowitz[2], and Nicolas Tabareau[3]

1 **CNRS, Université de Nice Sophia-Antipolis, France**
2 **CNRS, Université Savoie Mont-Blanc, France**
3 **Inria, Nantes, France**

Abstract

In classical homotopy theory, the *homotopy hypothesis* asserts that the fundamental ω-groupoid construction induces an equivalence between topological spaces and weak ω-groupoids. In the light of Voevodsky's *univalent foundations* program, which puts forward an interpretation of types as topological spaces, we consider the question of transposing the homotopy hypothesis to type theory. Indeed such a transposition could stand as a new approach to specifying higher inductive types. Since the formalisation of general weak ω-groupoids in type theory is a difficult task, we only take a first step towards this goal, which consists in exploring a shortcut through *strict* ω-categories.

The first outcome is a satisfactory type-theoretic notion of strict ω-category, which has hsets of cells in all dimensions. For this notion, defining the 'fundamental strict ω-category' of a type seems out of reach. The second outcome is an 'incoherently strict' notion of type-theoretic ω-category, which admits arbitrary types of cells in all dimensions. These are the 'wild' ω-categories of the title. They allow the definition of a 'fundamental wild ω-category' map, which leads to our (partial) homotopy hypothesis for type theory (stating an adjunction, not an equivalence).

All of our results have been formalised in the Coq proof assistant. Our formalisation makes systematic use of the machinery of coinductive types.

1998 ACM Subject Classification F.4.1 Mathematical logic

Keywords and phrases Homotopy Type theory, Omega-categories, Coinduction, Homotopy hypothesis

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.226

1 Introduction

Martin-Löf type theory offers an alternative to the classical set-theoretic approach to mathematics. The univalent foundations program [23] advocates understanding sets as a very special kind of types (so-called 0-*types*, or *hsets*). Conversely, types should be understood in set-theoretic terms as some kind of topological spaces, hsets corresponding to those whose connected components are contractible. And, indeed, types have been interpreted as ω-groupoids of various flavours, most notably simplicial [13] and globular [3, 16, 24]. Now, on the set-theoretic side, the *homotopy hypothesis* states that topological spaces and ω-groupoids are equivalent [22]. This work addresses the question of coining a type-theoretic counterpart of this homotopy hypothesis.

We here propose a preliminary and partial version which barely expresses that the 'fundamental ω-groupoid' functor has a kind of left adjoint (instead of being an equivalence). For our proposal, it is sufficient to choose an appropriate target category T of 'ω-groupoids', together with an appropriate 'fundamental ω-groupoid' functor $\pi\colon \mathsf{Type} \to \mathsf{T}$. Indeed, we may then express our partial homotopy hypothesis as follows:

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 226–240

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Hypothesis 1.** *There exist* $L\colon \mathsf{T} \to \mathsf{Type}$ *and* $\eta\colon \forall C, \mathsf{T}(C, \pi(L(C)))$ *such that for each* $C\colon \mathsf{T}$ *and* $T\colon \mathsf{Type}$*, the map*

$$\eta(C)^\star\colon \mathsf{T}(\pi(L(C)), \pi(T)) \to \mathsf{T}(C, \pi(T))$$

given by precomposition with $\eta(C)$ *is an equivalence.*

The idea of a 'fundamental ω-category' map goes back to van den Berg/Garner and Lumsdaine [3, 16]. There, the 'fundamental' ω-categories extracted from types are definitely weak, because expressions like $f \circ (g \circ h)$ and $(f \circ g) \circ h$ (for suitable cells f, g, h and higher categorical composition $\circ$) may differ definitionally. However, they are always equal propositionally, so when reasoning inside type theory as we do, the distinction becomes invisible. Based on this observation, we here explore the shortcut of a 'fundamental *strict* ω-category' map.

Thus, our first task is to transpose to type theory the classical notion of strict ω-category. This is the subject of Section 3 (after a brief recap on set-theoretic strict ω-categories in Section 2). There we face a crucial choice concerning the type of cells in all dimensions. If we take these types to be hsets, we get an honest notion of *strict* ω-category. While if we allow these types to be arbitrary, we get an 'incoherently strict' notion, which we call *wild*. In wild ω-categories, some coherence diagrams usually showing up in definitions of *weak* ω-categories make sense, but are not required to commute – even weakly, hence the name 'wild'.

Now the crucial point is that we are able to define a 'fundamental wild ω-category' map, while a strict one seems out of reach – a difficulty previously observed by Altenkirch et al. [2] in a similar context.

Let us mention another crucial choice faced when defining both notions. Indeed, we have to express equations as commuting diagrams of morphisms between globular types. Because such morphisms form a coinductive type, there are two standard choices [8] for their equality (identity types and bisimilarity). However, we weren't able to define our 'fundamental ω-category' map using either of them, so we work with a third, coarser one (Definition 8). (Of course, using identity types or bisimilarity yields other, perfectly sensible definitions.)

Altogether this yields in Section 5 a first version of our hypothesis with T the type for wild ω-categories. Our hypothesis essentially asserts that new types may be constructed from the homotopical information carried by any wild ω-category. This is clearly akin to the assumption of existence for *higher inductive types* [23], as well as to the Rezk completion for precategories in [1] and we briefly discuss the relationship.

In Section 6, we get back to ω-groupoids, as opposed to ω-categories. We briefly discuss problems and solutions concerning the definition of wild ω-groupoids and a perhaps more primitive formulation of our homotopy hypothesis involving them. We finally conclude and sketch further directions in Section 7.

A note on the formalisation

This paper presents informally a mathematical development based not on set theory but on Martin-Löf type theory enriched with coinduction. All our definitions and statements have been formalised in the Coq proof assistant [21]. The formalisation is available as [9]. The code is composed of 2750 lines of definitions/theorems and less than 800 lines of proofs (as given by `coqwc`). Definitions and theorems constitute 75% of the formalisation, which is a lot. This is because `coqwc` only counts as proof what is coded using the tactic language, whereas most of our proofs are coded directly in Gallina. The reason for this is that the

computational content of many of our proofs turned out to be crucial for their usability at later stages. This phenomenon has also been observed, e.g., of the Coq/HoTT library (`https://github.com/HoTT/HoTT`).

2 Set-theoretic strict ω-categories

Before presenting our definition of wild and strict ω-categories in type theory, we briefly sketch the traditional set-theoretic definition of strict ω-categories. We refer the reader to Lafont et al. [15] for further detail.

The base for strict ω-categories is the notion of globular sets, which are generally defined as presheaves over the so-called *globular category* Glob:

$$[0] \underset{t_0}{\overset{s_0}{\rightrightarrows}} [1] \underset{t_1}{\overset{s_1}{\rightrightarrows}} [2] \qquad \dots \qquad [n] \underset{t_n}{\overset{s_n}{\rightrightarrows}} [n+1] \qquad \dots$$

where for all n, $s_{n+1} \circ s_n = t_{n+1} \circ s_n$ and $s_{n+1} \circ t_n = t_{n+1} \circ t_n$.

▶ **Notation 1.** *We denote by s_n^p the composite $s_{p-1} \circ \dots \circ s_n$, which returns the n-dimensional source of a p-cell. Similarly, we use t_n^p.*

The idea is just that a globular set X has *objects*, the elements of $X[0]$, *1-cells* between them, and so on, each $(n+1)$-cell having *parallel* n-cells as source and target. E.g., a typical 2-cell looks like this:

$$x \underset{\alpha \cdot t_1}{\overset{\alpha \cdot s_1}{\Downarrow \alpha}} y,$$

where we use the standard shorthand notation, e.g., $\alpha \cdot s_1$ for $X(s_1)(\alpha)$. In the picture, $x = \alpha \cdot t_1 \cdot s_0 = \alpha \cdot s_1 \cdot s_0$ and $y = \alpha \cdot t_1 \cdot t_0 = \alpha \cdot s_1 \cdot t_0$.

We first concentrate on the data for composition and the so-called *interchange law*. A strict ω-category is a globular set X equipped (among other data), for all $n < p$, with a partial, binary *composition* operation on p-cells, defined on a pair (β, α) when the iterated source $\beta \cdot s_n^p$ of β matches the iterated target $\alpha \cdot t_n^p$ (recall Notation 1). The result is denoted by $\beta \circ_n \alpha$. Each such composition operation is required to be associative on the nose.

The source and target of such a composition are given by obvious globular intuition, generally not even spelled out. When $p = n+1$, composition is like categorical composition, i.e., we have $(\beta \circ_n \alpha) \cdot s_n = \alpha \cdot s_n$ and similarly $(\beta \circ_n \alpha) \cdot t_n = \beta \cdot t_n$. When $p > n+1$, composition is more like horizontal composition of 2-cells in a 2-category, as in

$$x \underset{\alpha \cdot t_1}{\overset{\alpha \cdot s_1}{\Downarrow \alpha}} y \underset{\beta \cdot t_1}{\overset{\beta \cdot s_1}{\Downarrow \beta}} z,$$

so we have

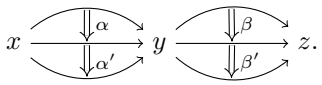
$$(\beta \circ_n \alpha) \cdot s_{p-1} = (\beta \cdot s_{p-1}) \circ_n (\alpha \cdot s_{p-1})$$

and similarly for t_{p-1}.

The crux of the definition of strict ω-category is the *interchange* law, which generalises the perhaps more well-known 2-categorical interchange law. It says that whenever $n < p < q$ and $\alpha, \alpha', \beta, \beta'$ are adequately composable q-cells, we have

$$(\beta' \circ_p \beta) \circ_n (\alpha' \circ_p \alpha) = (\beta' \circ_n \alpha') \circ_p (\beta \circ_n \alpha).$$

Graphically, for $n = 0$, $p = 1$, $q = 2$, both ways of composing the diagram below coincide:



Finally, we require identities $id_x \colon x \to x$ for all n-cells $x \in X[n]$. Let id_x^p denote the iteration $id_{id_{\cdots id_x}}$, p times (with $id_x^0 = x$). Identities should satisfy

$$id_x^p \circ_n \alpha = \alpha \qquad \beta \circ_n id_x^p = \beta \qquad id_{\beta \circ_n \alpha} = id_\beta \circ_n id_\alpha, \tag{1}$$

for all $x \in X[n]$, $p \geq 1$, α, and β such that $\alpha \cdot t_n^{n+p} = x$ and $\beta \cdot s_n^{n+p} = x$.

While this presentation is perfectly sensible when working in set theory, it is less convenient in type theory because many properties of the structure are imposed *by equations* rather than obtained *by construction*. For instance, compatibility between source and target maps in globular sets relies on equations. Using equations introduces a lot of non-definitional equalities that are very hard to deal with. On the contrary, we will see in the next section that our use of coinductive definitions allows us to define structures more computationally, thus avoiding the common pitfall of using equations.

3 Wild and strict ω-categories

In this section, we present our definition of wild and strict ω-categories. Our heavy use of coinduction is inspired by Cheng and Leinster's account [7] of Trimble's ω-categories, as well as by Lafont et al.'s presentation [15] of, e.g., weak equivalences. We start with globular types, a type-theoretic counterpart of globular sets. Essentially, a wild ω-category is a globular type equipped with identities and compositions in all dimensions, satisfying unitality and associativity axioms, and such that, at each dimension, composition preserves higher-dimensional identities and compositions. A strict ω-category is a wild ω-category whose underlying globular type is a globular hset, i.e., consists of hsets in all dimensions. The main goal of the present section is to give a precise meaning to the previous sketch of definition. Indeed, this is not completely straightforward, and we will explicitly discuss our design choices.

3.1 Globular types

The coinductive presentation of globular sets, which we'll here call globular types, is well-known and extremely simple:

▶ **Definition 1.** A *globular type* X consists of a type $|X|$, plus for all $x, y \in |X|$, a globular type $X[x, y]$.

Many of our definitions will, like the previous one, be coinductive or corecursive, and we will systematically omit to mention this feature. For instance, a morphism $X \twoheadrightarrow Y$ of globular types consists of

- a function $|f| \colon |X| \to |Y|$ between object types, and
- for all $x, x' \in |X|$, a morphism $f_{x,x'} \colon X[x, x'] \twoheadrightarrow Y[|f|(x), |f|(x')]$ of globular types.

Of course, morphisms of globular types compose.

We also need a definition of the cartesian product of globular types:

▶ **Definition 2.** The *product* $X \times Y$ of two globular types X and Y is defined by

- $|X \times Y| = |X| \times |Y|$ and
- for all $x, x' \in |X|$ and $y, y' \in |Y|$, $(X \times Y)[(x, y), (x', y')] = X[x, x'] \times Y[y, y']$.

3.2 Data for composition and identities

Let us now start our definition of wild ω-categories.

In the following definition of composition for globular types, we choose to tie together infinitely many elementary compositions into a morphism of globular types:

▶ **Definition 3.** To equip a globular type X with *ω-categorical composition* is to give:
- for all $x, y, z \in |X|$, a morphism $\mathsf{comp}_X(x,y,z)\colon X[x,y] \times X[y,z] \nrightarrow X[x,z]$ of globular types,
- and, for all $x, y \in |X|$, ω-categorical composition on $X[x,y]$.

In the terminology of Section 2, the function $|\mathsf{comp}_X|$ between object types represents composition for $p = n+1$, whereas compositions for $p > n+1$ are packed in higher-dimensional components $(\mathsf{comp}_X)_{(f,g),(f',g')}$. Similarly:

▶ **Definition 4.** To equip a globular type X with *ω-categorical identities* is to give:
- for all $x \in |X|$, an element $id^X_x \in |X[x,x]|$, and
- for all $x, y \in |X|$, ω-categorical identities on $X[x,y]$.

▶ **Definition 5.** An *ω-precategory* is a globular type equipped with ω-categorical composition and identities.

▶ Remark. Our use of 'precategory' conflicts with [1], where it is used for 'not-necessarily-univalent' categories. However, it coincides with Cheng's [6].

Most notions defined so far on globular types lift to ω-precategories, namely the object type $|X|$, the hom-ω-precategory $X[x,y]$, and cartesian product $X \times Y$. We lift notations accordingly. However, lifting the notion of morphism of globular types requires some work, which we now do, yielding the notion of ω-functor.

3.3 Omega-functors

It is not straightforward to define what it means for a morphism $F\colon X \nrightarrow Y$ of globular types between ω-precategories to preserve composition.

Let us treat the object level first, and consider any $x, x', x'' \in |X|$. The idea is that F preserves composition at x, x', x'' iff the diagram

$$\begin{array}{ccc} X[x,x'] \times X[x',x''] & \xrightarrow{\quad\mathsf{comp}_X(x,x',x'')\quad} & X[x,x''] \\ {\scriptstyle F_{x,x'} \times F_{x',x''}}\downarrow & & \downarrow{\scriptstyle F_{x,x''}} \\ Y[|F|x,|F|x'] \times Y[|F|x',|F|x''] & \xrightarrow[\mathsf{comp}_Y(|F|x,|F|x',|F|x'')]{} & Y[|F|x,|F|x''], \end{array} \tag{2}$$

commutes.

To give a meaning to this commutation, we need to choose a notion of equality of globular morphisms. Because globular morphisms form a coinductive type, identity types are expected to be too fine as a notion of equality between them [8]. The standard choice for this is bisimilarity:

▶ **Definition 6.** Two globular morphisms $F, G\colon X \nrightarrow Y$ are *bisimilar* iff
- for all $x \in |X|$, $|f|(x) = |g|(x)$, and
- for all $x, x' \in |X|$, $f_{x,x'}$ and $(p_x)_*((p_{x'})_*(g_{x,x'}))$ are bisimilar,

where $p_*(-)$ denotes transport [23] along p and p_x is the given equality $|f|(x) = |g|(x)$.

Transport is necessary here in order to compare $f_{x,x'} : X[x,x'] \nrightarrow Y[|f|(x),|f|(x')]$ and $g_{x,x'} : X[x,x'] \nrightarrow Y[|g|(x),|g|(x')]$.

Bisimilarity would be a very reasonable notion of equality between globular morphisms. However, we were not able to construct the 'fundamental wild ω-category' of a type for the resulting notion of wild ω-category. So we here decide to use the following extensional and inductive definition (based on globular cells).

▶ **Definition 7.** We define the type of globular n-cells of an ω-precategory A, noted $cell_n\, A$, by induction on the natural number n as

- $cell_0\, A := |A|$, and
- $cell_{n+1}\, A := \sum_{a,a':|A|} cell_n(A[a,a'])$.

Given a globular morphism $F : A \nrightarrow B$ and $c : cell_n\, A$, the globular cell Fc of B obtained by applying F to c can by defined inductively by

- $|F|c$ when $n = 0$, and
- $(|F|a, |F|a', F_{a,a'}c')$ when $c = (a, a', c')$.

▶ **Definition 8.** Two globular morphisms $F, G : A \nrightarrow B$ are *extensionally equal* iff

$$\text{for all } n \in \mathbb{N} \text{ and } c \in cell_n\, A,\ Fc = Gc.$$

We may now settle the following (obviously proof-relevant) definition:

▶ **Definition 9.** A morphism $F : X \nrightarrow Y$ of globular types between ω-precategories *preserves composition* iff

- for all $x, x', x'' \in |X|$, the square (2) commutes extensionally, and
- for all $x, x' \in |X|$, $F_{x,x'}$ preserves composition.

Preservation of composition will be used below in Definition 12 to express the interchange law. E.g., consider any ω-precategory X, and $x, y, z \in |X|$. Viewing $\mathsf{comp}_X(x,y,z)$ as a morphism of globular types between ω-precategories, saying that it preserves composition amounts to stating the interchange law of Section 2, specialised to $n = 0$. For instance, on objects, it means that, taking X in (2) to be $X[x,y] \times X[y,z]$, for all $f, f', f'' \in X[x,y]$, $g, g', g'' \in X[y,z]$, $a \in X[x,y][f,f']$, $a' \in X[x,y][f',f'']$, $b \in X[y,z][g,g']$, and $b' \in X[y,z][g',g'']$, we have $(b' \bullet b) \circ (a' \bullet a) = (b' \circ a') \bullet (b \circ a)$ (using some hopefully clear notation).

We may treat preservation of identities in a similar way:

▶ **Definition 10.** A morphism $F : X \nrightarrow Y$ of globular types between ω-precategories *preserves identities* iff

- for all $x \in |X|$, $|F_{x,x}|(id^X_x) = id^Y_{|F|(x)}$, and
- for all $x, y \in |X|$, $F_{x,y}$ preserves identities.

This will again be used in Definition 12 to enforce the third law of (1). Indeed, for any ω-precategory X, and $x, y, z \in |X|$, saying that $\mathsf{comp}_X(x,y,z)$ preserves identities entails, e.g., that the identity on any pair $(f,g) \in |X[x,y] \times X[y,z]|$ should be mapped by

$$X[x,y][f,f] \times X[y,z][g,g] \xrightarrow{(\mathsf{comp}_X(x,y,z))_{(f,g),(f,g)}} X[x,z][g \circ f, g \circ f]$$

to the identity on $g \circ f$ (abbreviating $|\mathsf{comp}_X(x,y,z)|(f,g)$ to $g \circ f$).

▶ **Definition 11.** A morphism of globular types between ω-precategories is an *ω-functor* iff it preserves composition and identities.

3.4 Wild ω-categories

It is now routine to extend the previous techniques to define *associativity* as extensional commutation in all dimensions of all diagrams

$$\begin{array}{ccc} X[x,y] \times X[y,z] \times X[z,t] & \xrightarrow{\mathsf{comp}_X(x,y,z) \times X[z,t]} & X[x,z] \times X[z,t] \\ \downarrow{\scriptstyle X[x,y] \times \mathsf{comp}_X(y,z,t)} & & \downarrow{\scriptstyle \mathsf{comp}_X(x,z,t)} \\ X[x,y] \times X[y,t] & \xrightarrow[\mathsf{comp}_X(x,y,t)]{} & X[x,t] \end{array}$$

and *unitality* as extensional commutation in all dimensions of all diagrams

$$\begin{array}{ccc} X[x,y] & \xrightarrow{\langle \ulcorner id_x \urcorner \circ !, X[x,y] \rangle} & X[x,x] \times X[x,y] \\ \downarrow{\scriptstyle \langle X[x,y], \ulcorner id_y \urcorner \circ ! \rangle} & \searrow & \downarrow{\scriptstyle \mathsf{comp}_X(x,x,y)} \\ X[x,y] \times X[y,y] & \xrightarrow[\mathsf{comp}_X(x,y,y)]{} & X[x,y], \end{array}$$

where $!\colon A \twoheadrightarrow 1$ denotes the unique morphism to the terminal globular type with 1 at all stages (for all A), and $\ulcorner id_x \urcorner\colon 1 \twoheadrightarrow X[x,x]$ maps the unique element of $|1|$ to id_x, the unique endo 1-cell over it to id_{id_x}, and so on.

We may at last define the type ω-wCat of wild ω-categories:

▶ **Definition 12.** A *wild ω-category* is an ω-precategory, satisfying associativity and unitality, whose compositions are ω-functors in all dimensions. Morphisms ω-$\mathsf{wCat}(C, D)$ between wild ω-categories C and D are simply ω-functors between the underlying ω-precategories.

The complete formal definition of wild ω-categories is given in the file `omega_categories.v` [9].

3.5 Strict ω-categories

The definition of wild ω-categories is not satisfactory as a type-theoretic account of strict ω-categories. As a matter of fact, wild ω-categories are not even weak ω-categories. Indeed, they appear to lack some coherence conditions. For instance, for any wild ω-category X, $f \in |X[x,y]|$ and $g \in |X[y,z]|$, there are two proofs of $g \circ (id_y \circ f) = g \circ f$ (the less trivial one goes to $(g \circ id_y) \circ f$ and then simplifies). These proofs induce by transport two 2-cells, say l and r. In weak higher categories, one imposes that l and r are related by a 'coherence' 3-cell. This is not the case in our wild ω-categories, which may thus be viewed as 'incoherently' weak ω-categories.

One perhaps reassuring perspective is that wild ω-categories can be restricted to ω-categories where all higher coherences are trivially satisfied. This is the case when the involved types are all hsets, which leads to:

▶ **Definition 13.** A *strict* ω-category is a wild ω-category X such that $|X|$ is an hset and for all $x, y \in |X|$, $X[x,y]$ is a strict ω-category. We call ω-sCat the type of strict ω-categories.

Strict ω-categories are intuitively close to set-theoretic strict ω-categories, and, as suggested by a referee, we expect their interpretation in standard models such as the simplicial model [13] to coincide with set-theoretic ω-categories. However, as we have seen, our definition relies on extensional equality (Definition 8), where identity types or bisimilarity could have been used. We thus in fact have three notions of strict ω-categories, and it is not entirely clear that all three are interpreted in the simplicial model as set-theoretic strict ω-categories.

As previously observed in a similar context [2], it seems impossible to define any satisfactory 'fundamental strict ω-category'. That is why we devote the rest of the paper to the definition of the 'fundamental wild ω-category' and use it to propose (partial) homotopy hypotheses.

4 Fundamental wild ω-category

We have defined wild ω-categories, and now turn to the definition of the 'fundamental wild ω-category' map $\pi\colon \mathsf{Type} \to \omega\text{-}\mathsf{wCat}$. This is essentially an internal variant of van den Berg and Garner's [3] and Lumsdaine's [16] constructions, with wild ω-categories instead of weak ω-categories. For any type T, we first easily define the globular type underlying $\pi(T)$. We then explain the definition of composition, which is a good example of how we had to generalise some of our statements in order to be able to tie the coinduction loop. We refer the reader to the formalisation for complete definitions and proofs. The part on composition is technical and may safely be skipped: the sequel only makes use of Theorem 16.

We start by defining the globular type underlying the fundamental wild ω-category.

▶ **Definition 14.** For any type T, the underlying globular type of $\pi(T)$ (still denoted by $\pi(T)$ by abuse of notation) has T itself as its type of objects, and $(\pi(T))[x, y] = \pi(x = y)$ for all $x, y \in T$.

We now turn to defining composition on $\pi(T)$. The definition of composition seems to introduce a lot of choices. Indeed, let us start by fixing $x, y, z \in T$ and look at what $\pi(x = y) \times \pi(y = z) \twoheadrightarrow \pi(x = z)$ is on objects (i.e., on 1-cells in $\pi(T)$). The obvious choice is concatenation of equality proofs, which we denote by $(a\colon x = y), (b\colon y = z) \mapsto (a \centerdot b)$ [23]. But actually, here we need to choose between two different definitions of concatenation, depending on whether a or b is eliminated first. Fortunately, it is well known that both definitions are equal, so the choice does not really matter.

Actually, this situation occurs at every dimension: the definition of composition is not unique, but all potential candidates are equal. This claim is justified by the work of Lumsdaine [16], where he constructs the operad $P_{ML^{Id}}$ of all definable composition laws over a (generic) type and shows that this operad is contractible. Contractibility means that all possible choices of composition are equal. The difficulty is to show that our particular choice of composition gives rise to a wild ω-category. For lack of space, we will only sketch the definition of our compositions, referring to the formalisation [9] for details.

Let us start with the definition of $\pi(x = y) \times \pi(y = z) \twoheadrightarrow \pi(x = z)$ in low dimensions, for $x, y, z \in T$. On objects, we have seen that the obvious choice is concatenation. On 1-cells, consider $a, a'\colon x = y$, $b, b'\colon y = z$, together with $e\colon a = a'$ and $f\colon b = b'$. How to compose e and f into a proof of $a \centerdot b = a' \centerdot b'$? We consider, for all types A, B, C and map $\varphi\colon A \to B \to C$, the obvious function $\mathsf{ap2}_{A,B,C,\varphi}$ of type

$$\forall a, a' \in A, b, b' \in B, e \in (a = a'), f \in (b = b'), \varphi\ a\ b = \varphi\ a'\ b'.$$

Applying this with $A = (x = y)$, $B = (y = z)$, $C = (x = z)$, and $\varphi = \centerdot$, we indeed get $\mathsf{ap2}\ a\ a'\ b\ b'\ e\ f$ of type $a \centerdot b = a' \centerdot b'$ (omitting the subscript of $\mathsf{ap2}$ for readability). To now deal with 2-cells, considering $e'\colon a = a'$, $f'\colon b = b'$, $u\colon e = e'$, and $v\colon f = f'$, we again apply $\mathsf{ap2}$, with $A = (e = e')$, $B = (f = f')$, $C = (\varphi'\ e\ f = \varphi'\ e'\ f')$, with $\varphi'\ e\ f = \mathsf{ap2}\ a\ a'\ b\ b'\ e\ f$ for all e, f. In the next dimension, we'll need a different φ'' with one more layer of $\mathsf{ap2}$.

It would be obvious how to formalise this process coinductively if it weren't for the first level, where $\centerdot$ is used. The trick is thus to abstract over this. Here, things become slightly

more verbose, and we apologise to the reader: for all types A, B, C, functions $\varphi\colon A \to B \to C$, and elements $a, a' \in A$ and $b, b' \in B$, we coinductively define a morphism of globular types

$$\mathsf{comp2}_{A,B,C,\varphi}(a, a', b, b')\colon \pi(a = a') \times \pi(b = b') \twoheadrightarrow \pi(\varphi\ a\ b = \varphi\ a'\ b')$$

(the '2' refers to ap2) by

- mapping $e\colon a = a'$ and $f\colon b = b'$ to $\mathsf{ap2}\ \varphi\ e\ f\colon \varphi\ a\ b = \varphi\ a'\ b'$,
- and then defining $(\mathsf{comp2}_{A,B,C,\varphi}(a, a', b, b'))_{(e,f),(e',f')}$ to be

$$\mathsf{comp2}_{A[a,a'],B[b,b'],C[\varphi\ a\ b,\varphi\ a'\ b'],\mathsf{ap2}\ \varphi}(e, e', f, f'). \tag{3}$$

This is well-defined, since $(\mathsf{comp2}_{A,B,C,\varphi}(a, a', b, b'))_{(e,f),(e',f')}$ should have type

$$(\pi(a = a') \times \pi(b = b'))[(e, f), (e', f')] \twoheadrightarrow \pi(\varphi\ a\ b = \varphi\ a'\ b')[\mathsf{ap2}\ \varphi\ e\ f, \mathsf{ap2}\ \varphi\ e'\ f'],$$

i.e.,

$$\pi(a = a')[e, e'] \times \pi(b = b')[f, f'] \twoheadrightarrow \pi(\varphi\ a\ b = \varphi\ a'\ b')[\mathsf{ap2}\ \varphi\ e\ f, \mathsf{ap2}\ \varphi\ e'\ f'],$$

or equivalently

$$\pi(e = e') \times \pi(f = f') \twoheadrightarrow \pi(\mathsf{ap2}\ \varphi\ e\ f = \mathsf{ap2}\ \varphi\ e'\ f'),$$

which is indeed the type of (3). It is now routine to define, for all types A and elements $a, a', a'' \in A$,

$$\mathsf{hcomp}_A(a, a', a'')\colon \pi(a = a') \times \pi(a' = a'') \twoheadrightarrow \pi(a = a'')$$

by

- mapping $e\colon a = a'$ and $f\colon a' = a''$ to $e \centerdot f$, with
- $\mathsf{hcomp}_A(a, a', a'')_{(e,f),(e',f')} = \mathsf{comp2}_{(a=a'),(a'=a''),(a=a''),(\lambda e.\lambda f.e \centerdot f)}(e, e', f, f')$.

This is again well-defined, because the latter has type

$$\pi(e = e') \times \pi(f = f') \twoheadrightarrow \pi(e \centerdot f = e' \centerdot f'),$$

i.e.,

$$\pi(a = a')[e, e'] \times \pi(a' = a'')[f, f'] \twoheadrightarrow \pi(a = a'')[e \centerdot f, e' \centerdot f'],$$

as expected.

▶ **Definition 15.** This yields composition structure on $\pi(T)$, for any T:

- for all $x, y, z \in T$, we let $\mathsf{comp}_{\pi(T)}(x, y, z) = \mathsf{hcomp}_T(x, y, z)$;
- for all $x, y \in T$, we get composition structure on $\pi(x = y)$ by coinduction hypothesis.

One may similarly define identity structure and show:

▶ **Theorem 16.** *This ω-precategory structure makes $\pi(T)$ into a wild ω-category.*

Proof. Let us say a few words about the proof. It resorts to a high level of generalisation to tie the coinduction loop for each law of ω-categories. In particular, a more explicit coinductive definition of the interchange law is developed and proved equivalent to the compact version that composition preserves composition. This explicit interchange law, plus a proof that composition in all dimensions sends pairs of proofs by reflexivity to some proof by reflexivity, enables us to prove the interchange law by coinduction, using elimination of identity types, a.k.a. path induction. The other laws are dealt with similarly. The complete proof is given in the file `type_to_omega_cat.v` [9]. ◀

5 Partial Homotopy Hypothesis

The classical homotopy hypothesis states an equivalence between spaces and ω-groupoids. It can be formulated either at the level of so-called homotopy categories [12], or at the level of model categories [11] or even at the level of $(\infty, 1)$-categories [22]. On the type-theoretic side, we propose our first partial homotopy hypothesis:

▶ **Hypothesis 1.** *There exist* $L\colon \omega\text{-}\mathsf{wCat} \to \mathsf{Type}$ *and* $\eta\colon \forall C, \omega\text{-}\mathsf{wCat}(C, \pi(L(C)))$ *such that for each wild* ω*-category* C *and type* T*, the map*

$$\eta(C)^\star\colon \omega\text{-}\mathsf{wCat}(\pi(L(C)), \pi(T)) \to \omega\text{-}\mathsf{wCat}(C, \pi(T))$$

given by precomposition with $\eta(C)$ *is an equivalence.*

We conjecture that Hypothesis 1 is consistent, and more precisely that it holds in the groupoid model [10]. Indeed, in this model, small types are small discrete groupoids and (small) wild ω-categories are the set-theoretic, strict ω-categories of Section 2. The fundamental wild ω-category $\pi(T)$ of any small discrete groupoid T is a discrete globular set, equipped with the only possible additionnal structure. Thus, morphisms from any strict ω-category C into $\pi(T)$ are just maps from $C[0]$ to T compatible with (the equivalence relation induced by) $C[1]$. Hence we can define $L(C)$ to be the quotient $C[0]/C[1]$, $\eta(C)$ being induced by the quotient map.

This indicates in particular that Hypothesis 1 is consistent with the Univalence Axiom. But it also shows that the asserted adjunction may be far from an equivalence. An easy way of strengthening our hypothesis is to require $\eta(C)$ to be a weak equivalence, in the sense of [15]:

▶ **Definition 17.** An ω-functor $F\colon C \twoheadrightarrow D$ is a *weak equivalence* iff

- for all $d \in |D|$, there exists $c \in |C|$ such that $|D[|F|(c), d]|$ is inhabited, and
- for all $c, c' \in |C|$, $F_{c,c'}$ is a weak equivalence.

Another possibility is to state a proper adjunction between L and π and ask both its unit and counit to be weak equivalences in the appropriate sense. We haven't yet investigated such strengthened hypotheses.

For now, let us modestly check how our hypothesis implies the existence of a type corresponding to the standard type-theoretic circle. For this purpose we introduce the wild ω-category S^1 as follows: it has a single object $\star$, and $S^1[\star, \star]$ is the discrete wild ω-category on $\mathbb{N}$-many objects with composition given by addition. Of course, we could have worked with $\mathbb{Z}$ instead. Assuming our hypothesis, we prove the expected, non-dependent induction principle for $L(S^1)$, together with (propositional versions of) computational rules.

▶ **Theorem 18.** *There exists a term inhabiting the non-dependent recursion principle of the circle*

$$L(S^1)_{\mathsf{rec}}\colon \forall T \in \mathsf{Type}, b \in T, l \in (b = b), L(S^1) \to T,$$

satisfying (propositionally) the expected computational laws:

$$L(S^1)_{\beta,\star}\colon \forall T, b, l, L(S^1)_{\mathsf{rec}}\ T\ b\ l\ (\eta\ \star) = b,$$
$$L(S^1)_{\beta,1}\colon \forall T, b, l, p_*((\mathsf{ap}\ (L(S^1)_{\mathsf{rec}}\ T\ b\ l)\ (\eta_{\star,\star}\ 1))) = l,$$

where $p_*(-)$ *denotes transport [23] along* $p = L(S^1)_{\beta,\star}\ T\ b\ l$*.*

Proof. Any ω-functor from S^1 is determined by the images of $\star$ and $1 \in S^1[\star, \star]$. Using the inverse of the equivalence given by Hypothesis 1, this induces an essentially unique ω-functor $\pi(L(S^1)) \twoheadrightarrow \pi(T)$, from which we extract the underlying map $L(S^1) \to T$. Using the retraction part of the equivalence given by Hypothesis 1, we deduce the computational laws. The complete proof[1] is given in the file `homotopy_hypothesis.v` [9]. ◀

This suggests that Hypothesis 1 may imply existence of certain *higher inductive types* [23]. The basic idea of higher inductive types is to generate not only the elements of an inductive type, but also equality proofs between them, and so on. Lumsdaine and Shulman [17] propose a semantics for them (in particular) in $(\infty, 1)$-toposes, using so-called strictly Reedy-functorial path objects. Closer to implementation, Sojakova [19] proposes an operational definition of higher inductive types as so-called homotopy-initial algebras. Both accounts fix a particular syntax for higher inductive types.

If, along the lines of Theorem 18, we could show that the partial homotopy hypothesis entails adequate induction principles, it could be understood as *specifying* higher inductive types, in a syntax-independent way. Of course, this would only be a definition from the internal point of view, i.e., the corresponding computational behaviour would not be accounted for.

So, given any candidate syntax for (or combinatorial description of) higher inductive types, this opens the option of describing 'the corresponding' wild ω-category, from which Hypothesis 1 would yield the desired type and reasoning principles.

▶ Remark. It is well-known that set-theoretic strict ω-categories cannot represent all homotopy types. E.g., they do not model the homotopy type of the 2-sphere [18]. Depending on the ambient type theory, wild ω-categories may be much more expressive than strict ω-categories. Nevertheless, we suspect that even in such cases, they may not adequately represent all types.

Let us consider a few example syntaxes.

To start with, Ahrens et al.'s categories [1] straightforwardly embed into wild ω-categories, and are enough to specify (a groupoidal version of) S^1 as above (but not S^2). Their categories may express 'non-freeness' properties of composition. E.g., we may consider the category obtained by quotienting S^1 under $1 + 1 = 1$, i.e., replace $\mathbb{N}$ by booleans and addition with sup. Or similarly quotient under $1 + 1 = 0$, i.e., work with $\mathbb{Z}/2\mathbb{Z}$. Also, it seems plausible to extend their Yoneda-based *Rezk completion* procedure—which (in their terms) constructs a (univalent) category from a precategory—to a proof of our hypothesis for categories. Please note, however, that they work in homotopy type theory, and their construction uses univalence (because the category of sets needs to be univalent) and higher inductive types (through propositional truncation).

A different 'syntax' is offered by globular types themselves, and we may hope for a type-theoretic analogue of the standard adjunction computing the free strict ω-category associated to any globular set. In contrast this syntax does not allow to express non-freeness properties. E.g., we may express S^1, but none of the quotients of S^1 evoked above. Also, we cannot express the higher inductive type for S^2 with one base point b and an equality proof on refl_b, because we cannot talk about identities. However, we can perfectly consider the globular set with two base points 0 and 1, two 1-cells $s, t \colon 0 \to 1$, and two 2-cells $s \to t$.

[1] The proof that the constructed ω-functor from S^1 actually preserves composition remains incomplete at the time of writing.

Finally, the most expressive such syntax would probably be offered by *computads* [20], a.k.a. *polygraphs* [5, 15]. A computad is essentially a graph, together with a set of 2-cells between parallel paths, together with a set of 3-cells between parallel paths of 2-cells, and so on. In particular, one may talk about identities and composition in all dimensions. And indeed, any strict ω-category is weakly equivalent to some strict ω-category which is free on a computad [15].

6 Towards ω-groupoids

In the fundamental wild ω-category $\pi(T)$ associated to any type T, all cells are invertible. We will see below that defining general ω-groupoids is not straightforward. Nevertheless, following Ahrens et al. [1], we have an economical, yet a bit restrictive definition:

▶ **Definition 19.** A wild ω-category X is a *univalent ω-groupoid* when for all $x, y \in |X|$,
- the map $(x = y) \to |X[x, y]|$ (induced by transporting id_x) is an equivalence (of types) [23],
- and $X[x, y]$ is a univalent ω-groupoid.

This univalence automatically entails existence of inverses in a very strong sense. In particular, for any wild ω-category X and $x, y, z \in |X|$, the diagram

$$\begin{array}{ccc} (x = y) \times (y = z) & \longrightarrow & (x = z) \\ \downarrow & & \downarrow \\ |X[x, y]| \times |X[y, z]| & \longrightarrow & |X[x, z]| \end{array}$$

(extensionally) commutes in Type, so that, if X is univalent, inverses in the ω-groupoidal sense have to be inverses in the identity type sense. The above definition has to be about *ω-groupoids*, as it implies that all cells are invertible. We prove:

▶ **Proposition 1.** *The ω-category $\pi(T)$ is a univalent ω-groupoid, for all types T.*

▶ **Proposition 2.** *For all univalent ω-groupoids G, there is a type T and an extensional equivalence $\pi(T) \simeq G$.*

By extensional equivalence we here mean an equivalence $e\colon |\pi(T)| \simeq |G|$ of types, such that for all $x, y \in T$ the map $\pi(x = y) \to |G[e\ x, e\ y]|$ induced by e is an extensional equivalence.

Proof. Take $T = |G|$. ◀

We conclude this section by considering perhaps more primitive versions of our hypothesis using some notion of *ω-groupoid* rather than ω-categories. Of course, univalent ω-groupoids are not interesting for this purpose, so we seek a definition of what it means for some (possibly higher) cell in a wild ω-category to be invertible. We first review possible notions of ω-groupoids and transpose a result of Cheng [6] showing that two of them are equivalent. We then get back to ω-groupoidal statements of our hypothesis.

Brown et al. [4] use *strict inverses*, i.e., for them an ω-groupoid is an ω-category in which for every n-cell $f\colon x \to y$ there is an n-cell $g\colon y \to x$ such that $g \circ_{n-1} f = id_x$ and $f \circ_{n-1} g = id_y$ (for $n > 0$). Others [20, 12, 18] consider *weak inverses*, in several apparently different ways. A recent preprint [14] shows that two such definitions coincide, namely those of Street [20], and Kapranov and Voevodsky [12]. The latter had previously been shown by Simpson [18] to be equivalent to an apparently stronger definition. Finally, Cheng [6] shows that these definitions are further equivalent to a seemingly weaker definition *when required of the whole ω-category*. A bit more precisely, Cheng defines the notion of a *dual*

to an n-cell in an ω-category, which is in general weaker than that of a *weak inverse* in the sense of [20, 12, 14]. But she shows that for a given ω-category, having all duals is equivalent to having all weak inverses. We now show how to recover this equivalence in our setting.

It is easy to define what it means for a wild ω-category to have all duals.

▶ **Definition 20.** In a wild ω-category X, for all $x, y \in |X|$ and $f \in |X[x, y]|$, a *dual* for f is a 1-cell $g \in |X[y, x]|$ such that there exist 2-cells inhabiting

$$|X[y, y][f \circ g, id_y]| \qquad \text{and} \qquad |X[x, x][g \circ f, id_x]|.$$

We then say that X *has all duals* iff for all $x, y \in |X|$ and $f \in |X[x, y]|$, f has a dual, and for all $x, y \in |X|$, $X[x, y]$ has all duals.

The notion of weak inverse is a bit harder. We follow the coinductive presentation of [15] (a definition considered 'unsound' by Cheng, but which Coq readily accepts!).

▶ **Definition 21.** We pose the following mutually coinductive definitions:

- two objects x and y of a wild ω-category X are *equivalent*, notation $x \sim y$, iff there exists a reversible 1-cell $f \in |X[x, y]|$;
- a 1-cell $f \in |X[x, y]|$ is *reversible* when it has a weak inverse;
- a *weak inverse* for a 1-cell $f \in |X[x, y]|$ is a 1-cell $g \in |X[y, x]|$ such that $g \circ f \sim id_x$ and $f \circ g \sim id_y$.

▶ **Definition 22.** A wild ω-category *has all weak inverses* when for all $x, y \in |X|$, any $f \in |X[x, y]|$ is reversible, and $X[x, y]$ has all weak inverses.

▶ **Proposition 3.** *Any wild ω-category has all duals iff it has all weak inverses.*

This allows us to comfortably state a first definition of ω-groupoid:

▶ **Definition 23.** A *wild ω-groupoid* is a wild ω-category with all duals.

One possible problem with this definition is that, ideally, being an ω-groupoid should be a mere property of ω-categories, i.e., for all X, the type 'X has all weak inverses' should be a mere proposition [23]. The very definition of univalence [23] suggests that having all duals might not suffice and that one may have to resort to some sensible notion of ω-adjunction in this context, but there does not seem to be any commonly accepted such notion in the literature, and we leave the question open.

There is one possible solution if we wish to delve into HoTT — except for this paragraph, our whole development remains within Martin-Löf type theory with coinduction. Namely, we could *truncate* the naive definition above. Indeed, according to the naive definition above, a wild ω-groupoid is a wild ω-category, equipped with a choice of duals for all cells in all dimensions. So we may define *brutal ω-groupoids* to denote wild ω-categories for which there exists, in the mere propositional sense, duals for all cells in all dimensions. It would however be preferable to have a mere property without resorting to truncation, as is done in the standard treatment of the univalence axiom.

We conclude this section by stating a groupoidal variant of our hypothesis. Since we have several candidate definitions for general ω-groupoids, we state our hypothesis taking this as a parameter. So let ω-Gpd denote some type of ω-groupoids.

▶ **Hypothesis 2.** *There exist $L \colon \omega\text{-}\mathsf{Gpd} \to \mathsf{Type}$ and $\eta : \forall G \in \omega\text{-}\mathsf{Gpd}, \omega\text{-}\mathsf{wCat}(G, \pi(L(G)))$ such that for each ω-groupoid G and type T, the following map is an equivalence:*

$$\eta(G)^\star \colon \omega\text{-}\mathsf{wCat}(\pi(L(G)), \pi(T)) \to \omega\text{-}\mathsf{wCat}(G, \pi(T)).$$

▶ Remark. We here use ω-functors as morphisms of ω-groupoids. An alternative would be to require morphisms of ω-groupoids to preserve weak inverses. If we could refine the type of inverses to a given morphism into a mere proposition, then both possibilities would coincide.

7 Conclusion and Future Work

We have defined wild and strict ω-categories, as well as wild and univalent ω-groupoids. We have constructed a 'fundamental ω-groupoid' map from types into all of these notions but strict ω-categories. We have stated a few variants of our partial homotopy hypothesis which postulates some correspondence between types and the given ω-dimensional structure. We have loosely related such hypotheses to the existence of higher inductive types and to the Rezk completion for one-dimensional categories.

The main remaining issue in our development is that we have used a rather coarse notion of equality between globular morphisms (Definition 8) in order to be able to define our 'fundamental wild ω-category' map. We wonder whether this could be done using a more standard notion like bisimilarity.

As explained in the introduction, this paper grew out of an attempt to use strict, as opposed to weak, ω-groupoids in the statement of a type-theoretic homotopy hypothesis. Beyond the issues raised by the definition of type-theoretic ω-groupoids (Section 6), it now seems likely that the lack of coherence of wild ω-categories disqualifies them for the full homotopy hypothesis.

Our main future challenge is thus clearly to propose a type-theoretic notion of weak ω-category allowing the definition of a 'fundamental weak ω-category' map. Assuming that we succeed in defining weak ω-categories, stating a full version of the homotopy hypothesis would first require us to define weak ω-groupoids properly. Thus, we expect the discussion of Section 6 about weak inverses to also be relevant in the weak case. Namely, we will need to investigate whether 'having all duals' is a mere property of weak ω-categories, and, if not, how to refine it into one. Finally, a full version of the homotopy hypothesis would essentially assert the existence of an infinite-dimensional generalisation of the Rezk completion. We wonder whether the construction of [1] could be adapted to this setting.

Acknowledgements. We thank the anonymous referees for insightful comments and suggestions. T. Hirschowitz and N. Tabareau acknowledge support from Récré ANR-11-BS02-0010.

References

1 B. Ahrens, K. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, 2015.

2 T. Altenkirch, N. Li, and O. Rypáček. Some constructions on ω-groupoids. In *LFMTP '14*, pages 4:1–4:8. ACM, 2014.

3 B. van den Berg and R. Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

4 R. Brown and P. Higgins. The equivalence of ∞-groupoids and crossed complexes. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 22, 1981.

5 A. Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical Computer Science*, 115(1):43–62, 1993.

6 E. Cheng. An ω-category with all duals is an ω-groupoid. *Applied Categorical Structures*, 15:439–453, 2007.

7 E. Cheng and T. Leinster. Weak ω-categories via terminal coalgebras. *ArXiv e-prints*, December 2012.
8 A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2014.
9 A. Hirschowitz, T. Hirschowitz, and N. Tabareau. Formalisation of strict omega-categories and the homotopy hypothesis in type theory, using coinduction. Coq code, `https://github.com/tabareau/omega_categories`, 2015.
10 M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS*, pages 208–212. IEEE Computer Society, 1994.
11 M. Hovey. *Model Categories*, volume 63 of *Mathematical Surveys and Monographs*. American Mathematical Society, 1999.
12 M. M. Kapranov and V. A. Voevodsky. ∞-groupoids and homotopy types. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 32(1):29–46, 1991.
13 C. Kapulkin, P. LeFanu Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. *ArXiv e-prints*, November 2012.
14 S. Kasangian, G. Metere, and E. Vitale. Weak inverses for strict n-categories. Preprint, 2009.
15 Y. Lafont, F. Métayer, and K. Worytkiewicz. A folk model structure on omega-cat. *Advances in Mathematics*, 224(3):1183 – 1231, 2010.
16 P. LeFanu Lumsdaine. Weak omega-categories from intensional type theory. *Logical Methods in Computer Science*, 6(3), 2010.
17 P. LeFanu Lumsdaine and M. Shulman. Semantics of higher inductive types. Preprint, 2012.
18 C. Simpson. Homotopy types of strict 3-groupoids. *ArXiv e-prints*, 1998.
19 K. Sojakova. Higher inductive types as homotopy-initial algebras. In *POPL*, pages 31–42. ACM, 2015.
20 R. Street. The algebra of oriented simplexes. *Journal of Pure and Applied Algebra*, 49(3):283–335, 1987.
21 The Coq development team. *Coq 8.4 Reference Manual*. Inria, 2012.
22 The NLab. Homotopy hypothesis. Accessed April 2015.
23 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.
24 M. A. Warren. The strict ω-groupoid interpretation of type theory. *CRM Proceedings and Lecture Notes*, 53:291–340, 2011.

Multivariate Amortised Resource Analysis for Term Rewrite Systems*

Martin Hofmann[1] and Georg Moser[2]

1 **Institute of Computer Science**
LMU Munich, Germany
hofmann@ifi.lmu.de

2 **Institute of Computer Science**
University of Innsbruck, Austria
georg.moser@uibk.ac.at

Abstract

We study amortised resource analysis in the context of term rewrite systems. We introduce a novel amortised analysis based on the potential method. The method is represented in an inference system akin to a type system and gives rise to polynomial bounds on the innermost runtime complexity of the analysed rewrite system. The crucial feature of the inference system is the admittance of multivariate bounds in the context of arbitrary data structures in a completely uniform way. This extends our earlier univariate resource analysis of typed term rewrite systems and continues our program of applying automated amortised resource analysis to rewriting.

1998 ACM Subject Classification F.3.2 Program Analysis

Keywords and phrases program analysis, amortised analysis, term rewriting, multivariate bounds

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.241

1 Introduction

Amortised resource analysis was pioneered by Sleator and Tarjan who used it to analyse the performance of new data structures that sometimes need to perform costly operations that pay off later on, e.g. rebalancing operations on a search tree.

Briefly, one assigns to the participating datastructures a nonnegative real-value, the *potential* in an a priori arbitrary fashion. One then defines the amortised cost of an operation as its actual cost, e.g. runtime plus the difference in potential of all datastructures before and after the operations. In this way, the amortised cost of a costly operation may be small if it results in a big decrease of potential. On the other hand some cheap operations that increase the potential will be overcharged. In this way, one can "save money" now to pay for costly operations later. By a simple telescoping argument the sum of all amortised costs in a sequence of operations plus the potential of the initial input data structure is also an upper bound on the *actual* cost of that sequence. In this way, amortised analysis yields rigorous bounds on actual resource usage and not just approximate or average bounds. If the potential functions are chosen well then the amortised costs of operations are either constant or exhibit merely a very simple dependency on the maximum size of all intermediate results which considerably facilitates a compositional analysis: the costs of running composite expressions

* This research is partially supported by FWF (Austrian Science Fund) project P25781.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 241–256

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

can be calculated as the sum of the individual costs. If costs are highly input-dependent, on the other hand, one must get bounds on the sizes or shapes of intermediate results which can be very difficult. This compositional aspect of amortised analysis makes it attractive for syntax-directed automation.

Of course, the crux of the matter is the choice of the correct potential functions. A simple concrete example is the implementation of a queue by two stacks, an in-tray and an out-tray. Incoming elements are added to the in-tray, outgoing elements are taken from the top of the out-tray. Only if the out-tray becomes empty the entire in-tray is reverse-copied into the out-tray. In this case, the length of the in-tray is clearly a suitable potential function. The costly operation of copying can entirely be paid from the big decrease in potential it causes.

In a nutshell automated amortised analysis works as follows. One selects a collection of basic potential functions, called basic resource functions, and assumes that all potential functions are linear combinations of these basic resource functions. One then performs a symbolic amortised cost analysis where the coefficients of these linear combinations as well as the amortised costs of operations (assumed constant) are unknowns. This yields a system of linear constraints for these unknowns whose solution provides the desired amortised analysis from which actual cost bounds in the form of functions of the size of the initial input can be easily read off.

Most automated amortised analyses are univariate in the sense that the joint potential of several arguments to an operation (a "context") is calculated as the sum of the individual potentials. This, however, proved unsatisfactory in the analysis of nested data structures such as lists of lists or trees and led to the development of multivariate analysis where also products and sums of products of the individual potentials may be used [9].

While automated amortised analysis has hitherto mostly been applied to functional and to a lesser extent to imperative programs; we are here interested in its application to term rewriting understood as a generalisation of functional programming to arbitrary constructor-defined datatypes. After a first step in this direction [12] which was based on univariate analysis, we now generalise to multivariate analysis and indeed subsume and further extend the entire system from [9].

On the one hand this gives a more general treatment of algebraic datatypes which are now untyped and merely defined by their constructor symbols. On the other hand, this necessitates a more general approach to basic resource functions which also streamlines the existing format in [9] or for that matter [11]. In [9] a basic resource function for a list type $L(A)$ is given by a finite list $[p_1, \ldots, p_k]$ of basic resource functions for the underlying type of entries A. The interpretation of such a list as a nonegative $\mathbb{R}$-valued potential function was then given by the formula

$$[p_1, \ldots, p_k]([v_1, \ldots, v_n]) := \sum_{1 \leqslant i_1 < \cdots < i_k \leqslant n} p_1(v_{i_1}) \cdot \cdots \cdot p_k(v_{i_k}) \; .$$

By treating tree types as a list of entries in depth-first order this same format could then be applied to trees as well. While these formats provided a smooth interaction with the typing rules and allow a very precise analysis of many examples they still look somewhat arbitrary and unjustified.

In the present paper, these formats are subsumed under a very general pattern that is on the one hand simpler and on the other hand permits an even smoother interaction with the typing rules for constructors and matching.

Namely, for us, a basic resource function is defined simply by a bottom-up tree automaton

$\mathcal{A}$ which acts on values by

$$p_{\mathcal{A}}(v) := \text{number of accepting runs of } \mathcal{A} \text{ on } v\ .$$

If c is a binary constructor, we have

$$p_{\mathcal{A}}(c(v_1, v_2)) = \sum_{c(\beta_1,\beta_2)\to\alpha\in\mathcal{A}} p_{(\mathcal{A},\beta_1)}(v_1) \cdot p_{(\mathcal{A},\beta_2)}(v_2)\ .$$

where α is the final state of $\mathcal{A}$ and $c(\beta_1, \beta_2) \to \alpha$ represents a transition in $\mathcal{A}$ and $(\mathcal{A}, \beta_i)$ denotes $\mathcal{A}$ with the final state set to β_i. Using this formula, the above formats for potentials on lists and trees are readily derived and obviously much more general potential functions can be defined which for example perform a rudimentary kind of type checking by simply ignoring certain constructors or, more interestingly, insisting on certain local patterns.

We note that the expression $p_{\mathcal{A}}(v)$ is known as the *ambiguity* of $\mathcal{A}$ [13] and has been extensively studied. In particular, the above recursive expansion of $p_{\mathcal{A}}(v)$ is known and attributed to Kuich, cf. [13]. However, these previous studies focused mainly on bounding $\max_v p_{\mathcal{A}}(v)$ as a function of the number of states (in the case where this quantity is at all finite) and has to our knowledge never been applied to complexity analysis of tree-like data structures.

Let us now look at a concrete example. Consider the following TRS $\mathcal{R}_{\mathsf{dyade}}$, encoding vector multiplication. The example forms a direct translation of the `dyade.raml` program discussed in [9].

$$\begin{array}{rr}
0 + y \to y & \mathsf{s}(x) + y \to \mathsf{s}(x + y) \\
0 \times y \to 0 & \mathsf{s}(x) \times y \to y + (x \times y) \\
\mathsf{mult}(n, []) \to [] & \mathsf{mult}(n, x :: xs) \to (n \times x) :: \mathsf{mult}(n, xs) \\
\mathsf{dyade}([], ls) \to [] & \mathsf{dyade}(x :: xs, ls) \to \mathsf{mult}(x, ls) :: \mathsf{dyade}(xs, ls)\ .
\end{array}$$

Consider a call to $\mathsf{dyade}(ls_1, ls_2)$, where ls_1 and ls_2 are lists. It is easy to see that the runtime complexity of $\mathcal{R}_{\mathsf{dyade}}$ crucially depends on the sum of the entries of ls_1 times the sum of the entries of ls_2, that is an optimal (automated) analysis should provide us with the certificate $\mathsf{O}(|ls_1| \cdot |ls_2|)$. However, state-of-the-art complexity tools, like AProVE [7], or T$_\mathsf{C}$T [2] overestimate the actual resource usage. For example T$_\mathsf{C}$T will provide a polynomial interpretation of degree 2, which is quadratic in $|ls_1|$, even if the monotonicity conditions are weakened suitably, cf. [8]. Also our earlier amortised resource analysis of typed TRS [12] can only provide the non-optimal bound $\mathsf{O}(|ls_1|^2 + |ls_2|^2)$. On the other hand the automated analysis of the RaML prototype is more to the point; the analysis with respect to `dyade.raml`, just overestimates the optimal bound by a linear factor and provides unnecessary big constants. The multivariate amortised analysis provided in this paper allows to lift this analysis to the above example and provides essentially optimal bounds (see the example on page 254).

This paper is structured as follows. In the next section we cover basics. In Section 3 we introduce resource functions as generalisations of resource polynomials to arbitrary constructor-defined datatypes. In Section 4 we present our type system and establish its soundness. Finally, we conclude in Section 5, where we also present related work.

2 Term Rewrite Systems and Tree Automata

We assume familiarity with term rewriting [4, 16] and tree automata [6] but briefly review basic concepts and notations.

$$\frac{x\sigma = v}{\sigma \overset{|0}{\vdash} x \Rightarrow v} \qquad \frac{c \in \mathcal{C} \quad x_1\sigma = v_1 \quad \cdots \quad x_n\sigma = v_n}{\sigma \overset{|0}{\vdash} c(x_1, \ldots, x_n) \Rightarrow c(v_1, \ldots, v_n)}$$

$$\frac{f(l_1, \ldots, l_n) \to r \in \mathcal{R} \quad \exists \tau\ \forall i\colon x_i\sigma = l_i\tau \quad \sigma \uplus \tau \overset{|m}{\vdash} r \Rightarrow v}{\sigma \overset{|m+1}{\vdash} f(x_1, \ldots, x_n) \Rightarrow v}$$

all x_i are fresh

$$\frac{\sigma \uplus \rho \overset{|m_0}{\vdash} f(x_1, \ldots, x_n) \Rightarrow v \quad \sigma \overset{|m_1}{\vdash} t_1 \Rightarrow v_1 \quad \cdots \quad \sigma \overset{|m_n}{\vdash} t_n \Rightarrow v_n \quad m = \sum_{i=0}^{n} m_i}{\sigma \overset{|m}{\vdash} f(t_1, \ldots, t_n) \Rightarrow v}$$

Here $\rho := \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

Figure 1 Operational Big-Step Semantics.

Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a signature, such that $\mathcal{F}$ contains at least one constant. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $\mathcal{V}\mathsf{ar}(t)$ to denote the set of variables occurring in term t. The *size* $|t|$ of a term is defined as the number of symbols in t. We suppose $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{C}$ denotes a finite, non-empty set of *constructor symbols*, $\mathcal{D}$ is a finite set of *defined function symbols*, and $\uplus$ denotes disjoint union. The set of ground constructor terms is denoted as $\mathcal{T}(\mathcal{C})$, ground constructor terms are also called *values*. A *substitution* σ is a mapping from variables to terms. Substitutions are conceived as sets of assignments: $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. We write $\mathsf{dom}(\sigma)$ ($\mathsf{rg}(\sigma)$) to denote the domain (range) of σ. Let σ, τ be substitutions such that $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\tau) = \varnothing$. Then we denote the (disjoint) union of σ and τ as $\sigma \uplus \tau$. We call a substitution σ *normalised* if all terms in the range of σ are values.

A *rewrite rule* is a pair $l \to r$ of terms, such that (i) the root symbol of l is defined, and (ii) $\mathcal{V}\mathsf{ar}(l) \supseteq \mathcal{V}\mathsf{ar}(r)$. A rule $l \to r$ is called *left-linear*, if l is linear. A *term rewrite system* (*TRS* for short) over $\mathcal{F}$ is a finite set of rewrite rules. In the sequel, $\mathcal{R}$ always denotes a TRS. The rewrite relation is denoted as $\to_{\mathcal{R}}$ and we use the standard notations for its transitive and reflexive closure. We simply write $\to$ for $\to_{\mathcal{R}}$ if $\mathcal{R}$ is clear from context. Let s and t be terms. If exactly n steps are performed to rewrite s to t, we write $s \to^n t$. In the sequel we are concerned with *innermost* rewriting, that is, an eager evaluation strategy. The *innermost rewrite relation* $\overset{\mathrm{i}}{\to}_{\mathcal{R}}$ of a TRS $\mathcal{R}$ is defined on terms as follows: $s \overset{\mathrm{i}}{\to}_{\mathcal{R}} t$ if there exists a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a substitution σ such that $s = C[l\sigma]$, $t = C[r\sigma]$, and all proper subterms of $l\sigma$ are normal forms of $\mathcal{R}$.

A TRS is *left-linear* if all rules are left-linear, it is non-overlapping if there a no critical pairs, that is, no ambiguity exists in applying rules. A TRS is *orthogonal* if it is left-linear and non-overlapping. A TRS is *completely defined* if all ground normal-forms are values. Note that an orthogonal TRS is confluent. Let s and t be terms, such that t is in normal-form. Then a *derivation* $D\colon s \to^*_{\mathcal{R}} t$ with respect to a TRS $\mathcal{R}$ is a finite sequence of rewrite steps. The *derivation height* of a term s with respect to a well-founded, finitely branching relation $\to$ is defined as: $\mathsf{dh}(s, \to) = \max\{n \mid \exists t\ s \to^n t\}$. A term $t = f(t_1, \ldots, t_k)$ is called *basic* if f is defined, and all $t_i \in \mathcal{T}(\mathcal{C})$. We define the *(innermost) runtime complexity* (with respect to $\mathcal{R}$): $\mathsf{rc}_{\mathcal{R}}(n) := \max\{\mathsf{dh}(t, \overset{\mathrm{i}}{\to}_{\mathcal{R}}) \mid t \text{ is basic and } |t| \leqslant n\}$.

We study *constructor* TRSs $\mathcal{R}$, that is, for each rule $f(l_1, \ldots, l_n) \to r$ we have that the arguments l_i are constructor terms. Furthermore, we restrict to *completely defined* and *orthogonal* systems. These restrictions are natural in the context of functional programming as orthogonal TRSs correspond to first-order function programs with pattern matching. Let $\mathcal{F}$ denote the signature underlying $\mathcal{R}$.

As $\mathcal{R}$ is completely defined, any derivation ends in a value. In connection with innermost rewriting this yields a *call-by-value* strategy. Furthermore, as $\mathcal{R}$ is non-overlapping any innermost derivation is determined modulo the order in which parallel redexes are contracted. This allows us to recast innermost rewriting into an operational big-step semantics instrumented with resource counters, cf. Figure 1. The semantics resembles similar definitions given in the literature on amortised resource analysis.

▶ **Proposition 1.** *Let f be a defined function symbol of arity n and σ a normalised substitution. Then $\sigma \mid\!\!\!\overset{m}{\vdash} f(x_1, \ldots, x_n) \Rightarrow v$ holds iff $\mathsf{dh}(f(x_1\sigma, \ldots, x_n\sigma), \to_{\mathcal{R}}) = m$.*

We suit the standard definition of bottom-up tree automata to our context. A *tree automaton* is a quadruple $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \alpha, \Delta)$, consisting of a finite signature $\mathcal{F}$, a finite non-empty set of states Q (disjoint from $\mathcal{F}$), a unique final state α, and a set of non-empty transitions Δ. Every rule in Δ has the following form $f(\alpha_1, \ldots, \alpha_n) \to \beta$ with $f \in \mathcal{F}$, $\alpha_1, \ldots, \alpha_n, \beta \in Q$.

Note that we only consider tree automata that consist of at least one state and feature a non-empty transition relation. As we will only be concerned with tree automata, we drop the qualifier "tree" and simply speak of an automaton. Observe that an automaton $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \alpha, \Delta)$ is conceivable as a finite ground TRS Δ over the signature $\mathcal{F} \cup \mathcal{Q}$, where the shape of the rewrite rules is restricted. The induced rewrite relation on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is denoted as $\to_{\mathcal{A}}$. A ground term t is *accepted* by $\mathcal{A}$ if $t \to_{\mathcal{A}}^* \alpha$; we set $L(\mathcal{A}) := \{t \mid t \to_{\mathcal{A}}^* \alpha\}$. Two automata $\mathcal{A}$ and $\mathcal{B}$ are *equivalent*, if $L(\mathcal{A}) = L(\mathcal{B})$. We use the notation $(\mathcal{A}, \beta)$ to refer to the automaton $(\mathcal{F}, \mathcal{Q}, \beta, \Delta)$ where $\beta \in \mathcal{Q}$. Note that $(\mathcal{A}, \alpha) = \mathcal{A}$ and we sometimes use the succinct notation instead of the expanded one.

In the sequel, $\mathcal{A}$ will always denote an automaton. Henceforth, $\mathcal{R}$ and $\mathcal{F}$, as well as the defined symbols $\mathcal{D}$ and constructors $\mathcal{C}$ are kept fixed. Furthermore, all considered substitutions are normalised.

3 Resource Functions

We define a set $\mathcal{BF}$ of *basic functions*, that map terms to natural numbers. Basic functions are indexed by a pair consisting of an automaton $\mathcal{A}$ and a state α. Resource functions will then be defined as nonnegative rational linear combinations of basic functions.

▶ **Definition 2.** Let $\mathcal{A} = (\mathcal{C}, \mathcal{Q}, \alpha, \Delta)$. For $v \in \mathcal{T}(\mathcal{C})$ we define the *basic function* $p_{\mathcal{A}}$, whose value $p_{\mathcal{A}}(v)$ is the number of accepting runs of $\mathcal{A}$ on v. The set of basic functions is denoted as $\mathcal{BF}$.

For any set of constructors $\mathcal{C}$, there exists an automaton $\mathcal{A}$ with $p_{\mathcal{A}}(v) = 1$ for all values v. Moreover $\mathcal{A}$ is unique upto renaming of the single state α. We call $\mathcal{A}$ the *canonical* automaton, denoted by $\varnothing$, whose unique state is denoted by $\varnothing$. As mentioned in the introduction $p_{\mathcal{A}}(v)$ is called *ambiguity* in the literature (see for example [13, 14]). In particular it is known that the finiteness of the *degree of ambiguity* $\sup_v p_{\mathcal{A}}(v)$ is polytime decidable [13]. The following is direct from the definition.

▶ **Proposition 3.** *Let $v \in \mathcal{T}(\mathcal{C})$ be a value such that $v = c(v_1, \ldots, v_n)$ and let $\alpha \in \mathcal{Q}$. We then have:*

$$p_{(\mathcal{A},\alpha)}(c(v_1, \ldots, v_n)) := \sum_{c(\alpha_1,\ldots,\alpha_n)\to\alpha\in\Delta} p_{(\mathcal{A},\alpha_1)}(v_1) \cdots p_{(\mathcal{A},\alpha_n)}(v_n)\ .$$

Note that $p_{(\mathcal{A},\alpha)}(c) = \sum_{c\to\alpha\in\Delta} 1$, as the empty product equals 1.

We could alternatively have used the latter as a recursive definition of $p_{(\mathcal{A},\alpha)}(v)$. The advantage of the definition based on runs is that we can immediately read off an exponential upper bound:

▶ **Proposition 4.** *Let q be the number of states of $\mathcal{A}$ and n the size of $v \in \mathcal{T}(\mathcal{C})$. Then $p_{\mathcal{A}}(v) \leqslant q^n$.*

It is also easy to see that this bound is actually taken on so that unlike the basic functions in [9] ours are not in general polynomials. If desired, it is however easy to impose syntactic restrictions that ensure polynomial growth. For example, we can use a ranking function on states and require that for each transition $c(\alpha_1, \ldots, \alpha_n) \to \alpha$ the ranks of the α_i are not bigger than that of α and that they strictly decrease for all but one transition with symbol c and result state α. All the concrete basic functions we use in examples satisfy this restriction and thus exhibit polynomial growth. We believe, however, that in some cases also exponential bounding functions may prove useful.

Let $\mathcal{C} = \{\mathsf{0}, \mathsf{s}, [], ::\}$ and consider the following automaton $\mathcal{A}$ with final state β_k.

$$
\begin{array}{lllll}
\mathsf{0} \to \alpha_0 & \mathsf{s}(\alpha_0) \to \alpha_1 & \cdots & \mathsf{s}(\alpha_{\ell-1}) \to \alpha_\ell & \\
& \mathsf{s}(\alpha_0) \to \alpha_0 & \cdots & \mathsf{s}(\alpha_{\ell-1}) \to \alpha_{\ell-1} & \mathsf{s}(\alpha_\ell) \to \alpha_\ell \\
[] \to \beta_0 & \alpha_{i_k} :: \beta_0 \to \beta_1 & \cdots & \alpha_{i_1} :: \beta_{k-1} \to \beta_k & \\
& \alpha_0 :: \beta_0 \to \beta_0 & \cdots & \alpha_0 :: \beta_{k-1} \to \beta_{k-1} & \alpha_0 :: \beta_k \to \beta_k \,.
\end{array}
$$

First, by a simple inductive argument we see that $p_{(\mathcal{A},\alpha_i)}(\mathsf{s}^n(\mathsf{0})) = \binom{n}{i}$ for $n \geqslant 0$ and $i = 0, \ldots, \ell$. Based on this, we conclude by induction on m:

$$
p_{(\mathcal{A},\beta_k)}([\mathbf{n}_1, \ldots, \mathbf{n}_m]) = \sum_{1 \leqslant j_1 < \cdots < j_k \leqslant m} \binom{n_{j_1}}{i_1} \cdots \binom{n_{j_k}}{i_k} \,,
$$

where we denote the numeral $\mathsf{s}^n(\mathsf{0})$ by $\mathbf{n}$, $[\mathbf{n}_1, \ldots, \mathbf{n}_m]$ abbreviates the corresponding cons-list, and $1 \leqslant i_j \leqslant \ell$ for all $j = 1, \ldots, k$.

Consider the set, denoted as $\mathfrak{A}$, of all non-equivalent (and non-empty) automata over $\mathcal{C}$. In the following we will frequently appeal to an enumeration of $\mathfrak{A}$, referring to a suitable chosen, but inessential encoding of automata. Note that in effect we will only work with a small, in particular finite subset of $\mathfrak{A}$.

▶ **Definition 5.** A *resource function* $r\colon \mathcal{T}(\mathcal{C}) \to \mathbb{Q}^+$ is a non-negative rational linear combination of basic functions, that is,

$$
r(t) := \sum_{\mathcal{A} \in \mathfrak{A}} q_{\mathcal{A}} \cdot p_{\mathcal{A}}(t) \,,
$$

where $p_{\mathcal{A}} \in \mathcal{BF}$. The set of resource functions is denoted as $\mathcal{RF}$.

The example above hints at the fact that the expressivity of our basic functions exceeds the expressivity of the *base polynomials* considered by Hoffmann et al. [10, 9]. The next proposition makes this fact precise.

▶ **Lemma 6.** *All the resource polynomials from [9] are also resource functions in the present automata-based setting.*

Proof. This is proved by induction on the definition of resource polynomials [9]. We do not need to recall their definition here; the inductive cases we establish reveal enough detail.

If p, q are basic resource polynomials for types A, B respectively then $\lambda ab(p(a) \cdot q(b))$ is a base resource polynomial for the product type $A \times B$. In rewriting, we can simulate product types by introducing a binary constructor $\mathsf{pair}(x, y)$. Now, if, inductively $p = p_{(\mathcal{A},\alpha)}$ and $q = p_{(\mathcal{B},\beta)}$, and w.l.o.g. the two automata have disjoint state sets, then we can build an automaton $\mathcal{C}$ whose states are the union of the states of $\mathcal{A}$ and $\mathcal{B}$ together with a new state γ and a transition $\mathsf{pair}(\alpha, \beta) \to \gamma$ in addition to the transitions from $\mathcal{A}$ and $\mathcal{B}$. We then have $p_{(\mathcal{C},\gamma)}(a, b) = p_{(\mathcal{A},\alpha)}(a) \cdot p_{(\mathcal{B},\beta)}(b) = p(a) \cdot q(b)$ as required. If $p_1, \ldots, p_k$ are base polynomials for type A then the base polynomial $[p_1, \ldots, p_k]$ given by

$$[p_1, \ldots, p_k]([a_1, \ldots, a_m]) := \sum_{1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant m} p_1(a_{i_1}) \cdot \cdots \cdot p_k(a_{i_k}) \ ,$$

is the generic base polynomial for lists over type A. Assuming that lists are constructed with the symbols [] and :: and that, inductively, $p_i = p_{(\mathcal{A}_i,\alpha_i)}$, we can build an automaton $\mathcal{B}$ as the disjoint union of the $\mathcal{A}_i$ together with $k + 1$ states $\beta_0, \beta_1, \ldots, \beta_k$, and transitions:

$$\varnothing :: \beta_i \to \beta_i \qquad \alpha_{i+1} :: \beta_{i+1} \to \beta_i \qquad [] \to \varnothing \ ,$$

where $i = 0, \ldots, k - 1$. (Recall that $\varnothing$ denotes the final state of the canonical automaton $\varnothing$, whose inclusion we here tacitly assume.) We obtain $[p_1, \ldots, p_k]([a_1, \ldots, a_n]) = p_{(\mathcal{B},\beta_k)}(a_1 :: a_2 \cdots a_n :: [])$. Finally, the generic resource polynomial for A-labelled trees takes the form

$$[p_1, \ldots, p_k](t) := [p_1, \ldots, p_k](l_t) \ ,$$

where l_t is the list of entries of t (in the leaves) in depth-first order. Note that we have

$$ls(t) := \begin{cases} \sum_{i=0}^{k} ([p_1, \ldots, p_i](t_1) \cdot & \text{if } ls = [p_1, \ldots, p_k],\ k > 1, \text{ and } t = \mathsf{node}(t_1, t_2) \\ \qquad \cdot [p_{i+1}, \ldots, p_k](t_2)) & \\ p(a) & \text{if } ls = [p] \text{ and } t = \mathsf{leaf}(a) \\ 1 & \text{if } ls = [] \text{ and } t = \mathsf{leaf}(a) \\ 0 & \text{otherwise .} \end{cases}$$

Thus, assuming the automata $\mathcal{A}_i$ $(i = 1, \ldots, k)$ as above, we can construct a new automaton $\mathcal{B}$ for $[p_1, \ldots, p_k]$ as the disjoint union of the $\mathcal{A}_i$ together with new states $\beta_{i,j}$ for $1 \leqslant i \leqslant j \leqslant k$ and the following transitions:

$$\mathsf{leaf}(\alpha_i) \to \beta_{i,i+1} \qquad \mathsf{leaf}(\varnothing) \to \beta_{i,i} \qquad \mathsf{node}(\beta_{i,t}, \beta_{t,j}) \to \beta_{i,j} \ ,$$

where $t = i, \ldots, j$. As above, we obtain $[p_1, \ldots, p_k](t) = p_{(\mathcal{B},\beta_{1,k})}(l_t)$. Thus the lemma follows. ◀

▶ **Lemma 7.** *If r and r' are resource functions, then $r + r', r \cdot r' \in \mathcal{RF}$.*

Proof. First, resource functions are obviously closed under addition. With respect to multiplication we employ the linearity of resource functions to see that it suffices to prove the claim for basic functions. Here we argue similar to the proof of Lemma 6 by using a product automata construction. ◀

▶ **Definition 8.** We define the set of *multivariate basic functions*, denoted as $\mathcal{BF}(n)$:

$$\mathcal{BF}(n) := \{v_1, \ldots, v_n \mapsto p_{\mathcal{A}_1}(v_1) \cdots p_{\mathcal{A}_n}(v_n) \mid \text{for all } i\text{: } v_i \in \mathcal{T}(\mathcal{C}) \text{ and } p_{\mathcal{A}_i} \in \mathcal{BF}\} \ .$$

We enumerate the set $\mathcal{BF}(n)$ by sequences of automata, such that $p_{(\mathcal{A}_1,\ldots,\mathcal{A}_n)}(v_1, \ldots, v_n) = p_{\mathcal{A}_1}(v_1) \cdots p_{\mathcal{A}_n}(v_n)$.

In the following sequences of automata (like $(\mathcal{A}_1, \dots, \mathcal{A}_n)$) are sometimes abbreviated as $\vec{\mathcal{A}}$, in particular we set $\vec{\varnothing} = (\varnothing, \dots, \varnothing)$. Note that $\mathcal{BF}(1) = \mathcal{BF}$; in the following we use $\mathcal{BF}$ as unique denotation.

▶ **Lemma 9.** *For all $p \in \mathcal{BF}(2)$, there exists $p' \in \mathcal{BF}$, such that $p(v, v) = p'(v)$ for all values v.*

Proof. By definition there exists automata $\mathcal{A}, \mathcal{B}$ such that $p(v, v) = p_{\mathcal{A}}(v) \cdot p_{\mathcal{B}}(v)$. By the previous lemma there exists an automaton $\mathcal{C}$ such that $p_{\mathcal{C}}(v) = p_{\mathcal{A}}(v) \cdot p_{\mathcal{B}}(v)$. Thus we set $p' := p_{\mathcal{C}}$ to conclude the lemma. ◀

Let $\mathcal{C}$ denote a set of constructor symbols. A *resource annotation over $\mathcal{C}$*, or simply *annotation*, is a family $Q = (q_{\mathcal{A}})_{\mathcal{A} \in \mathfrak{A}}$ with $q_{\mathcal{A}} \in \mathbb{Q}^+$ with all but finitely many of the coefficients $q_{\mathcal{A}}$ equal to 0. It represents a (finite) linear combination of basic resource functions. We generalise annotation for sequences of terms. An annotation for a sequence of length n is a family $Q = (q_{(\mathcal{A}_1,\dots,\mathcal{A}_n)})_{\mathcal{A}_i \in \mathfrak{A}}$ again vanishing almost everywhere. We denote annotations with upper-case letters from the end of the alphabet and use the convention that the corresponding lower-case letter denotes the elements of the family.

▶ **Definition 10.** The *potential* of a value v with respect to an annotation Q (of length 1), that is, $Q = (q_{\mathcal{A}})_{\mathcal{A} \in \mathfrak{A}}$, is defined as:

$$\Phi(v : Q) := \sum_{\mathcal{A} \in \mathfrak{A}} q_{\mathcal{A}} \cdot p_{\mathcal{A}}(v) \,,$$

where $p_{\mathcal{A}} \in \mathcal{BF}$. We generalise this to the potential of a term sequence with respect to an annotation $\overline{Q} = (q_{(\mathcal{A}_1,\dots,\mathcal{A}_n)})_{\mathcal{A}_i \in \mathfrak{A}}$: $\Phi(v_1, \dots, v_n : \overline{Q}) := \sum_{\mathcal{A}_1,\dots,\mathcal{A}_n \in \mathfrak{A}} q_{(\mathcal{A}_1,\dots,\mathcal{A}_n)} \cdot p_{(\mathcal{A}_1,\dots,\mathcal{A}_n)}(v_1, \dots, v_n)$, where $p_{(\mathcal{A}_1,\dots,\mathcal{A}_n)} \in \mathcal{BF}(n)$.

We are ready to generalise the notion of additive shift studied in [10, 9, 12].

▶ **Definition 11.** Suppose $Q = (q_{(\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B})})_{\mathcal{A}_i,\mathcal{B} \in \mathfrak{A}}$ denotes a resource annotation of length $m + 1$. Let $c \in \mathcal{C}$ be a constructor symbol of arity n. The *additive shift for c* of Q is an annotation $\lhd_c(Q) = (q'_{(\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B}_1,\dots,\mathcal{B}_n)})_{\mathcal{A}_i,\mathcal{B}_j \in \mathfrak{A}}$ for a sequence of length $n + m$, where $q'_{(\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B}_1,\dots,\mathcal{B}_n)}$ is defined as follows:

$$q'_{(\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B}_1,\dots,\mathcal{B}_n)} := \sum_{\substack{\mathcal{B}=(\mathcal{C},\mathcal{Q},\beta,\Delta)\in\mathfrak{A} \\ \text{with } c(\beta_1,\dots,\beta_n) \to \beta \in \Delta}} q_{(\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B})} \,, \tag{1}$$

Here $\mathcal{B}_i = (\mathcal{C}, \mathcal{Q}_{\mathcal{B}_i}, \beta_i, \Delta_{\mathcal{B}_i})$ for each $i = 1, \dots, n$.

The correctness of the additive shift operation follows from the next lemma.

▶ **Lemma 12.** *Let $v = c(v_1, \dots, v_n)$ be a value and let $Q = (q_{(\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B})})_{\mathcal{A}_1,\dots,\mathcal{A}_m,\mathcal{B} \in \mathfrak{A}}$ be an annotation of length $m + 1$. Then $\Phi(w_1, \dots, w_m, v : Q) = \Phi(w_1, \dots, w_m, v_1, \dots, v_n : \lhd_c(Q))$.*

Proof. In proof, we restrict to the case $m = 0$. Consider the value $v = c(v_1, \ldots, v_n)$:

$$\begin{aligned}\Phi(v : Q) &= \sum_{\mathcal{B} \in \mathfrak{A}} q_{\mathcal{B}} \cdot p_{\mathcal{B}}(c(v_1, \ldots, v_n)) \\ &= \sum_{\mathcal{B} \in \mathfrak{A}} q_{\mathcal{B}} \cdot \left(\sum_{c(\beta_1, \ldots, \beta_n) \to \beta \in \Delta} p_{(\mathcal{B}, \beta_1)}(v_1) \cdots p_{(\mathcal{B}, \beta_n)}(v_n) \right) \\ &= \sum_{\mathcal{B}_1, \ldots, \mathcal{B}_n \in \mathfrak{A}} q'_{(\mathcal{B}_1, \ldots, \mathcal{B}_n)} \cdot (p_{\mathcal{B}_1}(v_1) \cdots p_{\mathcal{B}_n}(v_n)) \\ &= \sum_{\mathcal{B}_1, \ldots, \mathcal{B}_n \in \mathfrak{A}} q'_{(\mathcal{B}_1, \ldots, \mathcal{B}_n)} \cdot p_{(\mathcal{B}_1, \ldots, \mathcal{B}_n)}(v_1, \ldots, v_n) = \Phi(v_1, \ldots, v_n : \lhd_c(Q)) \ .\end{aligned}$$

Here is is straightforward to check that $q'_{(\mathcal{B}_1, \ldots, \mathcal{B}_n)}$ is as in (1). Thus the last equation (and the lemma) follows. ◀

Let $Q = (q_{(A_1, \ldots, A_n)})_{\mathcal{A}_i \in \mathfrak{A}}$ denote a resource annotation of length n. Let $\vec{B} = (\mathcal{B}_1, \ldots, \mathcal{B}_m)$; we define the *projection* of Q with respect to $\vec{\mathcal{B}}$ to an annotation of length $\ell < n$. The projection is denoted as $\pi_\ell^{\vec{\mathcal{B}}}(Q)$. We set

$$\pi_\ell^{\mathcal{B}_1, \ldots, \mathcal{B}_m}(Q) := (q'_{(\mathcal{A}_1, \ldots, \mathcal{A}_\ell)})_{\mathcal{A}_i \in \mathfrak{A}} \ ,$$

where $q'_{(\mathcal{A}_1, \ldots, \mathcal{A}_\ell)} = q_{(\mathcal{A}_1, \ldots, \mathcal{A}_\ell, \mathcal{B}_1, \ldots, \mathcal{B}_m)}$ and $n = \ell + m$. Suppose $\Gamma, v_1, v_2 : Q$ denotes an annotated sequence of length $m + 2$. Suppose $v_1 = v_2$ and we want to *share* the values. Then we make use of the operator $\curlyvee(Q)$ that adapts the potential suitably. The operator is also called *sharing operator*.

▶ **Lemma 13.** *Let $\Gamma, v_1, v_2 : Q$ denote an annotated sequence of length $m + 2$. Then there exists a resource annotation $\curlyvee(Q)$ such that $\Phi(\Gamma, v_1, v_2 : Q) = \Phi(\Gamma, v : \curlyvee(Q))$, if $v_1 = v_2 = v$.*

Proof. This follows from Lemma 9. ◀

Let Q be an annotation over the set of constructor symbols $\mathcal{C}$ and let $K \in \mathbb{Q}^+$. Then we define $Q' := Q + K$ as follows: $Q' = (q'_{\mathcal{A}})_{\mathcal{A} \in \mathfrak{A}}$, where $q'_{\varnothing} := q_{\varnothing} + K$ and for all $\mathcal{A} \neq \varnothing$, $q'_{\mathcal{A}} := q_{\mathcal{A}}$. We define the comparison $\leqslant$ of two annotations Q, Q' over $\mathcal{C}$ pointwise: $Q \leqslant Q'$ if for all $\mathcal{A} \in \mathfrak{A}$: $q_{\mathcal{A}} \leqslant q'_{\mathcal{A}}$.

4 Amortised Cost Analysis

The rest of the paper is essentially an adaptation of the type systems given in [10, 9, 12]. There are no essential surprises but care must be taken with the rule for the evaluation of non-constructor terms which is essentially a combination of the let rule and the function application rule from [9].

Let Γ denote a sequence of variables $x_1, \ldots, x_n$; Γ is called a *variable context* or simply a *context*. The *length* n of Γ is denoted as $|\Gamma|$. A *resource annotation for* Γ or simply *annotation* is an annotation of length $|\Gamma|$. Now let Q be a resource annotation for Γ, let Q' be a resource annotation, let t be a term and let v denote its normalform. Then the typing judgement $\Gamma : Q \vdash t : Q'$ expresses that for *bounded* TRS (see Definition 15) the potential of Γ with respect to Q is sufficient to pay for the total cost m of the evaluation $\sigma \overset{m}{\vdash} t \Rightarrow v$ (σ a substitution), plus the potential of the value v with respect to Q'.

We comment on the inference rules in Figure 2. For the majority of the typing rules their definition is straightforward. Consider exemplary the typing rule for constructors

$c \in \mathcal{C}$. Due to Lemma 6 the potential of a value $c(x_1\sigma, \ldots, x_n\sigma)$ equals the potential of the variable context modulo an additive shift with respect to c. This is precisely expressed in the corresponding type rule. On the other hand the composition rule is more involved. Consider the judgement $\Gamma_1, \ldots, \Gamma_n : Q \vdash f(t_1, \ldots, t_n) : Q'$. Observe that on the basis of the sharing rule we can assume that $t = f(t_1, \ldots, t_n)$ is linear and thus the variable context splits into its parts Γ_i $(i = 1, \ldots, n)$. The intuition of the composition is that the potential of $\Gamma_1, \ldots, \Gamma_n$ (represented through the annotation Q) should be distributed over the evaluations of all arguments t_i and the evaluation of the function f, in a way such that the interdependency of the bounds is preserved. This is achieved by type checking each of the arguments and the context individually and verifying that all issuing annotations are consistent with each other.

In order to see how this works, we detail some of the constraints. First, consider $\pi^{\vec{j}_1}_{\Gamma_1}(Q) = P_1(\vec{j}_1)$. Here $\vec{j}_1$ denotes an index for $\Gamma_2, \ldots, \Gamma_n$. The projection asserts that the resources annotated in Q are projected to Γ_1, so that the typing $\Gamma_1 : P_1(\vec{j}_1) \vdash^{(\mathsf{cf})} t_1 : R_1(\vec{j}_1)$ is realisable for every index $\vec{j}_1$. The constraint $p^{i+1,\vec{j}_{i+1}}_{\ell} = r^{i,\vec{j}_i}_{k}$, where $1 \leqslant i < n$, ℓ an index for Γ_i and $k \in I$, guarantees that the annotations for the remaining contexts Γ_i, $i > 1$ are consistent with each other. Finally, constraint $\pi^{\vec{j}_n}_{x_n}(S) = R_n(\vec{j}_n)$ links the potential after the evaluation of the last argument t_n consistently with the annotation S for the variable context $x_1, \ldots, x_n$.

The type system given requires the use of *cost-free* judgements. Here the rules of the given TRS are considered as weak rules that are not taken into account in the complexity evaluation (see Definition 15).

▶ **Definition 14.** An *annotated signature* $\overline{\mathcal{F}}$ is a mapping from $\mathcal{D}$ to sets of pairs of resource annotations:

$$\overline{\mathcal{F}}(f) := \left\{ Q \to Q' \mid \begin{array}{l} \text{if the arity of } f \text{ is } n,\ Q \text{ is an annotation of length} \\ n \text{ and } Q' \text{ a resource annotation} \end{array} \right\} .$$

Usually, we confuse the signature and the annotated signature and denote the latter simply as $\mathcal{F}$.

The set of indices of an annotation for a context Γ is defined as follows:

$$I(\Gamma) := \{(\mathcal{A}_1, \ldots, \mathcal{A}_n) \mid \text{if } \mathcal{A} \in \mathfrak{A} \text{ and } |\Gamma| = n\} .$$

We also set $I := \mathfrak{A}$. Let Q be a resource annotation of length n, let $\Gamma = x_1, \ldots, x_\ell$ be a variable context, and let $\vec{i} = \mathcal{B}_1, \ldots, \mathcal{B}_m$ be an index of length m. We define the *projection with respect to* Γ: $\pi^{\vec{i}}_{\Gamma}(Q) := \pi^{\vec{i}}_{|\Gamma|}(Q)$.

Recall that any rewrite rule $l \to r \in \mathcal{R}$ can be written as $f(l_1, \ldots, l_n) \to r$ with $l_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ and that no variable occurs twice in l. Similar to the notion of well-typed TRSs in [12], we introduce *bounded* TRSs.

▶ **Definition 15.** Let $f(l_1, \ldots, l_n) \to r$ be a rewrite rule in $\mathcal{R}$ and let $\mathcal{V}\mathsf{ar}(f(l_1, \ldots, l_n)) = \{x_1, \ldots, x_\ell\}$. Suppose f is defined and let $Q \to Q' \in \mathcal{F}(f)$. Suppose further, for any such annotation $Q \to Q'$ we can derive

$$x_1, \ldots, x_\ell : P \vdash r : Q' , \tag{2}$$

where $P + 1 = R$ and R is obtained by iteratively applying shift operations with respect to the constructors occurring in $\bigcup_{i=1}^{n} l_i$ to Q. Then we say f is *bounded* wrt. $\mathcal{F}$. On the other hand f is *weakly bounded* if the rewrite step is not counted, that is, the judgement $x_1, \ldots, x_\ell : R \vdash r : Q'$ is asserted instead of (2). A TRS $\mathcal{R}$ over $\mathcal{F}$ is *bounded* (*weakly bounded*) if any defined f is bounded (weakly bounded).

$$\frac{}{x : Q \vdash x : Q} \qquad \frac{\Gamma : \pi^{\varnothing}_{\Gamma}(Q) \vdash t : Q'}{\Gamma, x : Q \vdash t : Q'}$$

$$\frac{f \in \mathcal{D} \quad Q \to Q' \in \mathcal{F}(f) \quad f \text{ is } n\text{-ary}}{x_1, \ldots, x_n : Q \vdash f(x_1, \ldots, x_n) : Q'} \qquad \frac{c \in \mathcal{C} \quad c \text{ is } n\text{-ary}}{x_1, \ldots, x_n : \lhd_c(Q) \vdash c(x_1, \ldots, x_n) : Q}$$

$$\frac{\begin{array}{c}\forall \vec{j}_1, \ldots, \vec{j}_n \quad \vec{j}_i \in I^{i-1} \times I(\Gamma_{i+1}) \times \cdots \times I(\Gamma_n) \\ P_i(\vec{j}_i) = (p^{\ell, \vec{j}_i})_{\ell \in I(\Gamma_i)} \quad R_i(\vec{j}_i) = (r^{k, \vec{j}_i})_{k \in I} \\ p_\ell^{i+1, \vec{j}_{i+1}} = r_k^{i, \vec{j}_i} \text{ if } \vec{j}_i = a \mathbin{@} \ell \mathbin{@} b \text{ and } \vec{j}_{i+1} = a \mathbin{@} k \mathbin{@} b \\ \pi^{\vec{j}_1}_{\Gamma_1}(Q) = P_1(\vec{j}_1) \quad \pi^{\vec{j}_n}_{x_n}(S) = R_n(\vec{j}_n) \quad x_1, \ldots, x_n : S \vdash f(x_1, \ldots, x_n) : Q' \\ \Gamma_1 : P_1(\vec{j}_1) \vdash^{(\mathsf{cf})} t_1 : R_1(\vec{j}_1) \quad \ldots \quad \Gamma_n : P_n(\vec{j}_n) \vdash^{(\mathsf{cf})} t_n : R_n(\vec{j}_n)\end{array}}{\Gamma_1, \ldots, \Gamma_n : Q \vdash f(t_1, \ldots, t_n) : Q'}$$

$$\frac{\Gamma, x, y : Q \vdash t[x, y] : Q' \quad x, y \text{ are fresh}}{\Gamma, z : \curlyvee(Q) \vdash t[z, z] : Q'} \qquad \frac{\Gamma, x : P \vdash t : P' \quad P \leqslant Q \quad P' \geqslant Q'}{\Gamma, x : Q \vdash t : Q'}$$

In the composition rule a and b denote suitable chosen indices, where @ denotes concatenation of indices. Further, the judgement $\Gamma_i : P_i(\vec{j}_i) \vdash^{(\mathsf{cf})} t_i : R_i(\vec{j}_i)$ abbreviates that $\Gamma_i : P_i(\vec{\varnothing}) \vdash t_i : R_i(\vec{\varnothing})$ and $\Gamma_i : P_i(\vec{j}_i) \vdash^{\mathsf{cf}} t_i : R_i(\vec{j}_i)$ for all $\vec{j}_i \neq \vec{\varnothing}$ and $i = 1, \ldots, n$.

Figure 2 Multivariate Analysis of Term Rewrite Systems.

$$\frac{\dfrac{y_1 : P_1(j) \vdash^{(\mathsf{cf})} y_1 : R_1(j) \quad \dfrac{\times : P_2(i) \to R_2(i)}{x, y_2 : P_2(i) \vdash^{(\mathsf{cf})} x \times y_2 : R_2(i)} \quad \dfrac{+ : S \to S'}{u, v : S \vdash u + v : S'}}{y_1, x, y_2 : T \vdash y_1 + (x \times y_2) : M'}}{x, y : \overline{M}_2 \vdash y + (x \times y) : M'} \; .$$

Here $i \in I$, and $j \in I(x, y_2)$.

Figure 3 Derivation of $x, y : \overline{M}_2 \vdash y + (x \times y) : M'$.

Let σ denote a substitution, $\Gamma = x_1, \ldots, x_n$ a context and Q a resource annotation. Then we define the *potential* of $\Gamma : Q$ with respect to σ:

$$\Phi(\sigma, \Gamma : Q) := \Phi(x_1\sigma, \ldots, x_n\sigma : Q) \; .$$

Note that the above definition employs the shift operator in a similar way as in [9], where this is part of the type system in the case for pattern matching.

Before we state our main result, we exemplify the use of the type system on a simple (but clarifying) example. Consider the following TRS $\mathcal{R}_\times$, restricting our motivating example (see page 243).

$$\begin{array}{ll} \mathsf{0} + y \to y & \mathsf{s}(x) + y \to \mathsf{s}(x + y) \\ \mathsf{0} \times y \to \mathsf{0} & \mathsf{s}(x) \times y \to y + (x \times y) \; . \end{array}$$

We consider the canonical automaton $\varnothing$ for the constructor symbols $\{\mathsf{0}, \mathsf{s}\}$ together with the automaton $\mathcal{A}$ defined as follows:

$$\mathcal{A}: \qquad \mathsf{0} \to \varnothing \qquad \mathsf{s}(\varnothing) \to \alpha \qquad \mathsf{s}(\varnothing) \to \varnothing \qquad \mathsf{s}(\alpha) \to \alpha$$

We show that $\mathcal{R}_\times$ is bounded with respect to the following annotated signatures:

$$+\colon \left\{(p_{(\varnothing,\varnothing)}, p_{(\mathcal{A},\varnothing)}) \to (p'_\varnothing) \mid p_{(\varnothing,\varnothing)} \geqslant 1, p_{(\mathcal{A},\varnothing)} \geqslant 1, p_{(\varnothing,\varnothing)} - 1 \geqslant p'_\varnothing\right\}$$

$$\times\colon \left\{\begin{pmatrix} m_{(\varnothing,\varnothing)} & m_{(\mathcal{A},\varnothing)} \\ 0 & m_{(\mathcal{A},\mathcal{A})} \end{pmatrix} \to (m'_\varnothing, m'_\mathcal{A}) \mid \begin{matrix} m_{(\varnothing,\varnothing)} \geqslant 1, m_{(\mathcal{A},\varnothing)} \geqslant 2, \\ m_{(\mathcal{A},\mathcal{A})} \geqslant 1, m_{(\varnothing,\varnothing)} - 1 \geqslant m'_\varnothing \end{matrix} \right\} .$$

Each of the four rules induced one of the following demands, cf. (2). (Recall that we denote annotations with upper-case letters and the elements of the families with the corresponding lower-case letters.)

$$y\colon \overline{P}_1 \vdash y\colon P' \qquad \overline{P}_1 + 1 = \lhd_0(P) \tag{3}$$

$$x, y\colon \overline{P}_2 \vdash \mathsf{s}(x+y)\colon P' \qquad \overline{P}_2 + 1 = \lhd_\mathsf{s}(P) \tag{4}$$

$$y\colon \overline{M}_1 \vdash \mathsf{0}\colon M' \qquad \overline{M}_1 + 1 = \lhd_0(M) \tag{5}$$

$$x, y\colon \overline{M}_2 \vdash y + (x \times y)\colon M' \qquad \overline{M}_2 + 1 = \lhd_\mathsf{s}(M) \tag{6}$$

It is not difficult to see that (3) and (4) induce the constraints $p_{(\varnothing,\varnothing)} \geqslant 1$, $p_{(\mathcal{A},\varnothing)} \geqslant 1$, and $p_{(\varnothing,\varnothing)} - 1 \geqslant p'_\varnothing$, and to ease the presentation we set $p'_\mathcal{A} = 0$. Furthermore, the typing judgement (5) yields the constraint $m_{(\varnothing,\varnothing)} - 1 \geqslant m'_\varnothing$.

Of more interest is the precise derivation of (6). In this derivation we make use of the weakly boundedness of $\mathcal{R}_\times$ with respect to the cost-free signature $+\colon (p_{(\varnothing,\varnothing)}) \to p'_\varnothing$ and $\times\colon (m_{(\varnothing,\varnothing)}) \to m'_\varnothing$, where it suffices to demand that $p_{(\varnothing,\varnothing)} \geqslant p'_\varnothing$ and $m_{(\varnothing,\varnothing)} \geqslant m'_\varnothing$. We obtain the following derivation in Figure 3. This derivation induces the constraint $\curlyvee(T) = \overline{M}_2$, as first (reading bottom-up) we employ a sharing rule. The composition rule yields the constraints $\pi^j_{y_1}(T) = P_1(j)$ $(j \in I(x, y_2))$, $\pi^i_v(S) = R_2(i)$ $(i \in I)$, and $r^{1,j}_i = p^{2,i}_j$, where $R_1(j) = (r^{1,j})_{i\in I}$ and $P_2(i) = (p^{2,i})_{j\in I(x,y_2)}$. Finally, the axioms yield the constraints $P_1(j) \geqslant R_i(j)$ for $j \in I(x, y_2)$ as well as the indicated conditions on the signature. It is tedious, but straightforward to check that these constraints can be met for the given annotations. Thus, $\times$ can be typed with the annotation $m_{(\varnothing,\varnothing)} = 1$, $m_{(\mathcal{A},\varnothing)} = 2$, and $m_{(\mathcal{A},\mathcal{A})} = 1$ which yields the bound $\mathsf{dh}(\mathbf{m} \times \mathbf{n}, \to_{\mathcal{R}_\times}) \leqslant m \cdot n + 2m + 1$.

It is instructive to depict the annotation $\overline{M}_2$ in matrix format, which allows a simple expression of the $\lhd_\mathsf{s}$-operator.

$$\overline{M}_2 = \begin{pmatrix} \overline{m}_{\varnothing,\varnothing} & \overline{m}_{\mathcal{A},\varnothing} \\ \overline{m}_{\varnothing,\mathcal{A}} & \overline{m}_{\mathcal{A},\mathcal{A}} \end{pmatrix} = \begin{pmatrix} m_{\varnothing,\varnothing} + m_{\mathcal{A},\varnothing} - 1 & m_{\mathcal{A},\varnothing} \\ m_{\mathcal{A},\mathcal{A}} & m_{\mathcal{A},\mathcal{A}} \end{pmatrix} = \lhd_\mathsf{s}(M) .$$

Then it becomes apparent that the annotation $\overline{M}_2$ is the result of adding the auxiliary annotation

$$\begin{pmatrix} m_{\mathcal{A},\varnothing} - 1 & 0 \\ m_{\mathcal{A},\mathcal{A}} & 0 \end{pmatrix} , \tag{7}$$

to the annotation for multiplication. Intuitively the type system asserts that we can split the annotation $\overline{M}_2$ into an annotation M that pays for the recursive call and the annotation (7) that pays for the call to addition.

▶ **Theorem 16.** *Let $\mathcal{R}$ be bounded. Suppose $\Gamma\colon Q \vdash t\colon Q'$ and $\sigma \overset{m}{\vdash\!\!\!|} t \Rightarrow v$. Then $\Phi(\sigma, \Gamma\colon Q) - \Phi(v\colon Q') \geqslant m$.*

Proof. Let Π be the proof deriving $\sigma \overset{m}{\vdash\!\!\!|} t \Rightarrow v$ and let Ξ be the proof of $\Gamma\colon Q \vdash t\colon Q'$. The proof of the theorem proceeds by main-induction on the length of Π and by side-induction

on the length of Ξ. We consider the case for composition. Suppose the last rule in Π has the form

$$\frac{\sigma \uplus \rho \mid\!\!\overset{m_0}{\vdash} f(x_1,\ldots,x_n) \Rightarrow v \quad \sigma \mid\!\!\overset{m_i}{\vdash} t_i \Rightarrow v_i \quad i=1,\ldots,n \quad m = \sum_{i=0}^{n} m_i}{\sigma \mid\!\!\overset{m}{\vdash} t \Rightarrow v} \ .$$

We can assume that $t = f(x_1,\ldots,x_n)$ is linear, due the presence of the share operator. In proof we restrict to the case where $n = 2$. Hence the last rule in the type inference Ξ is of the following form.

$$\frac{\Gamma_1 : P(j) \vdash^{(\mathsf{cf})} t_1 : \overline{P}(j) \quad \Gamma_2 : R(i) \vdash^{(\mathsf{cf})} t_2 : \overline{R}(i) \quad x, y : S \vdash f(x,y) : Q'}{\Gamma_1, \Gamma_2 : Q \vdash f(t_1,t_2) : Q'}$$

The following conditions hold, where we use the notations $P(j) = (p_i^j)_{i\in I(\Gamma_1)}$, $\overline{P}(j) = (\overline{p}_i^j)_{i\in I(\Gamma_1)}$, $R(i) = (r_j^i)_{j\in I(\Gamma_2)}$, and $\overline{R}(i) = (\overline{r}_j^i)_{j\in I(\Gamma_2)}$

$$\forall j \in I(\Gamma_2) \quad \pi^j_{\Gamma_1}(Q) = P(j) \qquad \forall i \in I \quad \pi^i_y(S) = \overline{R}(i) \qquad \forall i,j\ \overline{p}_i^j = r_j^i \ . \tag{8}$$

By induction hypothesis, we have (i) $\Phi(\sigma \uplus \rho, x, y : R) - \Phi(v : Q') \geqslant m_0$, (ii) for all $j \in I(\Gamma_2)$: $\Phi(\sigma, \Gamma_1 : P(j)) - \Phi(v_1 : \overline{P}(j)) \geqslant m_1$, and (iii) for all $i \in I$: $\Phi(\sigma, \Gamma_2 : R(i)) - \Phi(v_2 : \overline{R}(i)) \geqslant m_2$. Let $\vec{x} := x_1,\ldots,x_n$, where $\mathcal{V}\mathsf{ar}(t_1) = \{x_1,\ldots,x_n\}$ and let $\vec{y} := y_1,\ldots,y_n$ with $\mathcal{V}\mathsf{ar}(t_2) = \{y_1,\ldots,y_n\}$. The theorem follows by a straightforward calculation:

$$\begin{aligned}
\Phi(\sigma, \Gamma_1, \Gamma_2 : Q) &= \sum_{i\in I(\Gamma_1), j\in I(\Gamma_2)} q_{(i,j)} \cdot p_i(\vec{x}\sigma) \cdot p_j(\vec{y}\sigma) \\
&= \sum_{j\in I(\Gamma_2)} p_j(\vec{y}\sigma) \cdot \left(\sum_{i\in I(\Gamma_1)} p_i^j \cdot p_i(\vec{x}\sigma) \right) \\
&\geqslant \sum_{j\in I(\Gamma_2)} p_j(\vec{y}\sigma) \cdot \left(\sum_{i\in I} \overline{p}_i^j \cdot p_i(v_1) \right) + m_1 \\
&= \sum_{i\in I} p_i(v_1) \cdot \left(\sum_{j\in I(\Gamma_2)} r_j^i \cdot p_j(\vec{y}\sigma) \right) + m_1 \\
&\geqslant \sum_{i\in I} p_i(v_1) \cdot \left(\sum_{j\in I} \overline{r}_j^i \cdot p_j(v_2) \right) + m_1 + m_2 \\
&\geqslant \sum_{i\in I} q_i' \cdot p_i(v) + \sum_{i=0}^{2} m_i = \Phi(v : Q') + \sum_{i=0}^{2} m_i \ .
\end{aligned}$$

Here we tacitly employ the conditions (8) together with the induction hypothesis which is employed in line 3, 5, and 6. ◀

The following corollary to the theorem is immediate.

▶ **Corollary 17.** *Assume the conditions of the theorem. If additionally for all values v and annotations Q, $\Phi(v : Q) \in \mathsf{O}(n^k)$, where $n = |v|$, then $\mathsf{rc}_{\mathcal{R}}(n) \in \mathsf{O}(n^k)$.*

▶ Remark. Recall that we restrict to completely defined, orthogonal constructor TRSs. It is not difficult to see that Theorem 16 (and its corollary) generalise to the case where completely definedness is dropped. While completely definedness is essential for the correctness of the

big-step semantics presented in Figure 1, the proof of Theorem 16 extends with relative ease. The induction on the length of $\Pi\colon \sigma \overset{m}{\mid\!\!\frac{}{}} t \Rightarrow v$ is replaced by an induction on the length of an innermost derivation $D\colon t\sigma \to^{+}_{\mathcal{R}} v$.

Finally, we consider the motivating TRS $\mathcal{R}_{\mathsf{dyade}}$. Based on the above example (page 251) it remains to consider the remaining four rules of $\mathcal{R}_{\mathsf{dyade}}$:

$$\mathsf{mult}(n, []) \to [] \qquad \mathsf{mult}(n, x :: xs) \to (n \times x) :: \mathsf{mult}(n, xs)$$
$$\mathsf{dyade}([], ls) \to [] \qquad \mathsf{dyade}(x :: xs, ls) \to \mathsf{mult}(x, ls) :: \mathsf{dyade}(xs, ls)\ .$$

We consider the following automata $\varnothing$, $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, where $\varnothing$ denotes the canonical automata for $\{\mathsf{0}, \mathsf{s}, [], ::\}$ and $\mathcal{A}$ is defined as on page 251. The definition of $\mathcal{B}$ and $\mathcal{C}$ is given below.

$$\begin{array}{lllll} \mathcal{B}\colon & \mathsf{0} \to \varnothing & \mathsf{s}(\varnothing) \to \varnothing & [] \to \varnothing & \varnothing :: \varnothing \to \beta \qquad \varnothing :: \beta \to \beta \\ \mathcal{C}\colon & \mathsf{0} \to \varnothing & \mathsf{s}(\varnothing) \to \varnothing & \mathsf{s}(\varnothing) \to \alpha & \mathsf{s}(\alpha) \to \alpha \qquad [] \to \varnothing \\ & \alpha :: \varnothing \to \gamma & \varnothing :: \gamma \to \gamma\ . & & \end{array}$$

Note that $p_{\varnothing}(v) = 1$, $p_{\mathcal{A}}(\mathbf{n}) = n$, $p_{\mathcal{B}}(l) = |l|$, and $p_{\mathcal{C}}(l) = \sum_{i=1}^{m} n_i$, where $l = [\mathbf{n}_1, \dots, \mathbf{n}_m]$.

We make use of a similar denotation of the annotations as in the example on page 251 and set $\mathsf{mult}\colon M \to (m'_{\varnothing,\varnothing})$, where $M = (m_{\mathcal{A}_1,\mathcal{A}_2})_{\mathcal{A}_1\in\{\varnothing,\mathcal{A}\},\mathcal{A}_2\in\{\varnothing,\mathcal{B},\mathcal{C}\}}$ and similarly for dyade: $\mathsf{dyade}\colon D \to (d'_{\varnothing,\varnothing})$, where $D = (d_{\mathcal{A}_1,\mathcal{A}_2})_{\mathcal{A}_1\in\{\varnothing,\mathcal{B},\mathcal{C}\},\mathcal{A}_2\in\{\varnothing,\mathcal{B},\mathcal{C}\}}$ Thus we can assert the signature of mult and dyade as follows:

$$\mathsf{mult}\colon \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 1 \end{pmatrix} \to (0) \qquad \mathsf{dyade}\colon \begin{pmatrix} 1 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix} \to (0)\ .$$

Considering $\mathsf{dyade}(x :: xs, ls) \to \mathsf{mult}(x, ls) :: \mathsf{dyade}(xs, ls)$, we study the effects of the additive shift on D. Let $\overline{D} := \lhd_{::}(D) - 1$ such that $\overline{D} = (\overline{d}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3})_{\mathcal{A}_1\in\{\varnothing,\mathcal{A}\},\mathcal{A}_2,\mathcal{A}_3\in\{\varnothing,\mathcal{B},\mathcal{C}\}}$. For $\mathcal{A}_1 = \mathcal{A}$, then $\overline{d}_{\mathcal{A},\mathcal{A}_2,\mathcal{A}_3}$ vanishes almost everywhere, but $\overline{d}_{\mathcal{A},\varnothing,\mathcal{B}} = d_{\mathsf{C},\mathcal{B}} = 1$ and $\overline{d}_{\mathcal{A},\varnothing,\mathcal{C}} = d_{\mathsf{C},\mathcal{C}} = 1$. To ease the presentation, we ignore these positive annotations and only consider the restricted annotation $(\overline{d}_{\varnothing,\mathcal{A}_2,\mathcal{A}_3})_{\mathcal{A}_2,\mathcal{A}_3\in\{\varnothing,\mathcal{B},\mathcal{C}\}}$. This annotation is again representable as a matrix and typability of the rule follows, as we can decompose the matrix suitably:

$$\begin{pmatrix} 2 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}\ .$$

In order to estimate $\Phi(\sigma, xs, ys\colon D)$ for arbitrary σ we analyse the base functions of the considered automata. Thus the analysis yields the essentially optimal bound of the execution of $\mathsf{dyade}(ls_1, ls_2)$ as a multiplicative bound in the sum of the values of the first and second list ls_1 $(= xs\sigma)$ and ls_2 $(= ys\sigma)$, respectively, together with a linear factor.

5 Conclusion

We have presented a novel amortised resource analysis in the context of term rewrite systems. The method is represented in an inference system akin to a type system and can give rise to polynomial bounds on the innermost runtime complexity of the analysed rewrite system. The crucial feature of the inference system is the admittance of multivariate bounds in the

context of arbitrary data structures in a completely uniform way. This extends our earlier univariate resource analysis of typed term rewrite systems and continues our program of applying automated amortised resource analysis to rewriting.

We already briefly commented on the differences of the here presented study to our earlier work [12] in the introduction. As far as we can tell this and the present result are currently the only attempts to lift amortised cost analysis to rewriting or provide such a study in the context of arbitrary constructor-defined datastructures. Hoffmann and Shao provide in [11] a multivariate amortised analysis of integers and arrays that extend upon [10]. These language extensions are also provided in RaML. However, the treatment still appears to be ad-hoc and does not provide a similar uniform framework than ours. Further we mention some general work on automated resource analysis. Albert et al. [1] underlies COSTA, an automated tool for the resource analysis of Java programs. Sinn et al. provide in [15] related approaches for the runtime complexity analysis of C programs, incorporated into LOOPUS. Very recently Brockschmidt et al. [5] have provided a runtime complexity analysis of integer programs, taking also size considerations into account. Basic steps for a modular complexity framework for rewrite systems have been established in [3]. Finally, the RaML prototype [10] provides an automated potential-based resource analysis for various resource bounds of functional programs. For term rewriting AProVE [7] and T$_\mathsf{C}$T [2] are the most powerful tools for complexity analysis of rewrite systems as witnessed during last year's termination competition.[1]

In future work we will clarify the automatability of the method. We expect that by restricting the number of states and the format of those tree automata $\mathcal{A}$, whose annotation do not vanish, we can reduce inference of annotations to linear constraint solving in much the same way as in [10]. More challenging would be a combination of linear programming and combinatorial constraint solving to infer the best possible structure of automata.

References

1 E. Albert, P. Arenas, S. Genaim, G. Puebla, and G. Román-Díez. Conditional termination of loops over heap-allocated data. *Sci. Comput. Program.*, 92:2–24, 2014.

2 M. Avanzini and G. Moser. Tyrolean complexity tool: Features and usage. In *Proc. 24th RTA*, volume 21 of *LIPIcs*, pages 71–80, 2013.

3 M. Avanzini and G. Moser. A combination framework for complexity. *IC*, 2015. To appear.

4 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

5 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. 20th TACAS*, volume 8413 of *LNCS*, pages 140–155, 2014.

6 H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

7 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014.

8 N. Hirokawa and G. Moser. Automated complexity analysis based on context-sensitive rewriting. In *Proc. of Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 257–271, 2014.

[1] See `http://nfa.imn.htwk-leipzig.de/termcomp/competition/20`.

9 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3):14, 2012.

10 J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ML. In *Proc. 24th CAV*, volume 7358 of *LNCS*, pages 781–786, 2012.

11 J. Hoffmann and Z. Shao. Type-based amortized resource analysis with integers and arrays. In *Proc. 12th FLOPS*, volume 8475 of *LNCS*, pages 152–168, 2014.

12 M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Proc. of Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 272–286, 2014.

13 H. Seidl. On the finite degree of ambiguity of finite tree automata. *Acta Informatica*, 26(6):527–542, 1989.

14 H. Seidl. Ambiguity, Validy, and Costs. Technical report, TU München, 1992. Habiliation Thesis.

15 M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.

16 TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracks in Theoretical Computer Science*. Cambridge University Press, 2003.

Termination of Dependently Typed Rewrite Rules

Jean-Pierre Jouannaud[1,2] and Jianqi Li[2]

1 **LIX, École Polytechnique, INRIA, and Université Paris-Sud, France**
2 **School of Software, Tsinghua University, NLIST, Beijing, China**

Abstract

Our interest is in automated termination proofs of higher-order rewrite rules in presence of dependent types modulo a theory T on base types. We first describe an original transformation to a type discipline without type dependencies which preserves non-termination. Since the user must reason on expressions of the transformed language, we then introduce an extension of the computability path ordering CPO for comparing dependently typed expressions named DCPO. Using the previous result, we show that DCPO is a well-founded order, behaving well in practice.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.4.2 Other Rewriting Systems

Keywords and phrases rewriting, dependent types, strong normalization, path orderings

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.257

1 Introduction

This paper addresses the problem of (semi-)automating termination proofs for typed higher-order calculi defined by rewrite rules. Since many automated techniques exist for showing termination of simply typed higher-order rewrite rules, our first approach is to reduce the former to the latter.

To this end, we introduce a non-termination preserving transformation from dependently typed algebraic λ-terms to simply typed algebraic λ-terms. Unlike the transformation used for showing strong normalization of LF [13], the present one uses algebraic symbols and type constructors in an essential way. Dependently typed rewrite rules can then be shown terminating via the transformation. The user can therefore benefit from all existing tools allowing to check termination of higher-order rewrite rules. The drawback is that these tools will operate on the transformed rules.

Among all termination proof techniques, we favour the one reducing termination proofs to ordering comparisons between lefthand and righthand sides of rules. These comparisons require well-founded orders on typed algebraic λ-terms which are stable by context application and substitution instance. CPO is such an order on *simply typed* algebraic λ-terms, defined recursively on the structure of the compared terms [9]. CPO is indeed well-founded on *weakly polymorphic* λ-terms, the familiar ML-discipline for which quantifiers on types can only occur in prefix position. A recent extension of core CPO to appear in LMCS handles inductive types, constructors possibly taking functional arguments, and function symbols smaller than application and abstraction.

We formulate here a new extension DCPO of CPO for dependently typed algebraic λ-terms. DCPO is then viewed as an infinite set of dependently typed rewrite rules which are shown terminating by checking the transformed rules with CPO. It follows that DCPO is a well-founded order of the set of dependently typed λ-terms, whose syntax-directed comparisons require little input from the user, in the form of a precedence on the algebraic symbols used in the rules. DCPO is our answer for practice.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 257–272

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Variables:	**Abstraction:**	**Application:**	**Functions:**
$\dfrac{x{:}\sigma \in \Gamma}{\Gamma \ \vdash_\Sigma x{:}\sigma}$	$\dfrac{\Gamma,\, x{:}\sigma \ \vdash_\Sigma t{:}\tau}{\Gamma \ \vdash_\Sigma (\lambda x{:}\sigma.t){:}\sigma \to \tau}$	$\dfrac{\Gamma \ \vdash_\Sigma s{:}\sigma \to \tau \quad \Gamma \ \vdash_\Sigma t{:}\sigma}{\Gamma \ \vdash_\Sigma @(s,t){:}\tau}$	$\dfrac{f^n{:}\overline{\sigma} \to \sigma \in \mathcal{F} \quad \Gamma \ \vdash_\Sigma \overline{t}{:}\overline{\sigma}}{\Gamma \ \vdash_\Sigma f(\overline{t}){:}\sigma}$

Figure 1 Type system for monomorphic higher-order algebras.

Dependent programming has become a major trend in recent years [23, 16]. In practice, many types depend on natural numbers. Typing dependent definitions requires then a convertibility relation T including arithmetic laws [20], see Example 3.4. Our results allow for *dependent types modulo* T.

Sections 4 and 5 describe the non-termination preserving transformation and DCPO.

2 Higher-Order Algebras $\lambda^{\to}_{\Sigma}$

We assume a *signature* $\Sigma = \mathcal{S} \uplus \mathcal{F}$ of *sort symbols* in $\mathcal{S}$ and *function symbols* in $\mathcal{F}$. The set $\mathcal{T}^{\to}_{\mathcal{S}}$ of *simple types* (in short, *types*) is generated by the grammar $\sigma, \tau := a \in \mathcal{S} \mid \sigma \to \tau$. The (arrow) type constructor $\to$ associates to the right. The output sort of a type σ is itself if $\sigma \in \mathcal{S}$ and the output sort of τ if $\sigma = \nu \to \tau$. We use σ, τ, μ, ν for simple types.

Function symbols are meant to be algebraic operators upper-indexed by their fixed *arity* n. *Function declarations* are written $f^n : \sigma_1 \to \ldots \to \sigma_n \to \sigma$ (in short, $f : \overline{\sigma} \to \sigma$), where $\overline{\sigma}$ and σ are the *input* and *output types* of f^n. We use f^n, g^m for function symbols, possibly omitting m, n. $\lambda x : \sigma.s$, $@(s,t)$ and $f^n(t_1, \ldots, t_n)$ (or $f(\overline{t})$) are an *abstraction*, an *application*, and a *pre-algebraic* raw term. $f^0()$ is identified with f. We use x, y, z for variables, s, t, u, v, w, l, r for raw terms, $\mathrm{FV}(s)$ for the set of free variables of s and $|s|$ for the *size* of s.

Raw terms are seen as finite labeled trees by considering $\lambda x : \sigma.s$, for each $x : \sigma$, as a unary abstraction operator taking s as argument to construct the raw term $\lambda x : \sigma.s$. We abbreviate abstraction operators by λ. *Positions* are strings of strictly positive integers. We use i, j for positive integers, p, q for arbitrary positions. The empty string Λ is the *root* or *head* position and $\cdot$ is string concatenation. $\mathcal{P}os(t)$ is the set of positions of t.

Given a raw term s, $s(p)$ and $s|_p$ denote respectively the *symbol* and *subterm* of s at position p. For example, $(\lambda x : \sigma.u)|_1 = u$. The result of replacing the subterm $s|_p$ by the term t is written $s[t]_p$. A context term $s[x]_p$, in short $s[\,]_p$ or even $s[\,]$, is a term s in which x is a fresh variable, called *hole*, occurring at position p. All these notions extend as expected to a set P of disjoint positions, writing $s[t]_P$ for replacement of all terms in $s|_P$ by a single term t, and $s[\overline{x}]_P$ for a context with many holes.

An *environment* Γ is a finite set of pairs $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ where x_i is a variable, σ_i is a type, and $x_i \neq x_j$ for $i \neq j$. $\mathrm{FV}(\Gamma) = \{x_1, \ldots, x_n\}$ is the set of variables of Γ. Our typing judgements are written as $\Gamma \ \vdash_\Sigma s : \sigma$. A raw term s has type σ in the environment Γ if the judgement $\Gamma \ \vdash_\Sigma s : \sigma$ is provable in the inference system given in Figure 1. Typable raw terms are called *algebraic* λ*-terms* (in short, *terms* or *objects*), and their set is denoted by $\lambda_{\mathcal{F}}$ (or $\mathcal{O}^{\to}_{\mathcal{F}}$). Objects have a unique type in a given, possibly omitted, environment.

Substitutions are type-preserving homomorphisms avoiding captures, see for example [2], written here in postfix form, using the notation $\{x_i \mapsto s_i\}_i$. The congruence on terms generated by renaming the free variable x in s by the *fresh* variable $z \notin \mathrm{FV}(\lambda x.s)$ to yield the term $\lambda z.s\{x \mapsto z\}$ is called α*-conversion*, denoted by $=_\alpha$. We use γ, θ for substitutions.

A higher-order rewrite rule is a quadruple $\Delta \ \vdash_\Sigma l \to r : \sigma$ made of lefthand and righthand side terms l, r, and possibly omitted environment Δ and type σ. Rules β and η are particular

rewrite rule schemas. Reductions are defined as usual, and are a particular case of reductions in presence of dependent types, see Definition 3.3. A *higher-order reduction ordering* $\succeq$ is a quasi-order on terms satisfying: (i) its strict part $\succ$ is well-founded ; (ii) its equivalence is a congruence ; (iii) *monotonicity*: $s \succ t$ implies $u[s]_p \succ u[t]_p$ (assuming typability); (iv) *stability*: $s \succ t$ implies $s\gamma \succ t\gamma$ for all substitutions γ; and (v) *functionality*: $\longrightarrow_\beta \cup \longrightarrow_\eta \subseteq \succ$.

Given a set E the notation $\overline{s}$ shall be used for a list, multiset, or set of elements of E. Given a binary relation $\succ$ on E, we use $\overline{s} \succ_{lex} \overline{t}$ and $\overline{s} \succ_{mul} \overline{t}$ for its lexicographic and multiset extensions respectively. We use $s \succ \overline{t}$ for $(\forall t \in \overline{t})\, s \succ t$, and $\overline{s} \succ t$ for $(\exists s \in \overline{s})\, s \succ t$.

A rewrite relation generated by a set of rules $R \cup \{\beta, \eta\}$ can be proved terminating by checking whether $l \succ r$ for all rules in R with some higher-order reduction ordering $\succ$ [14]. CPO is such a higher-order reduction ordering based on three ingredients [9]:

- an order $\geq^{\to}$ on simple types, whose strict part $>^{\to}$ satisfies
  - **(i)** *well-foundedness:* $>^{\to} \cup \{\langle \sigma \to \tau, \sigma \rangle \mid \sigma, \tau \in \mathcal{T}^{\to}_{\mathcal{S}}\}$ is well-founded ;
  - **(ii)** *right arrow subterm:* $\sigma \to \tau >^{\to} \tau$;
  - **(iii)** *preservation:* $\sigma \to \tau =^{\to} \nu$ iff $\nu = \sigma' \to \tau'$, $\sigma =^{\to} \sigma'$, $\tau =^{\to} \tau'$ and
  - **(iv)** *decreasingness:* $\sigma \to \tau >^{\to} \nu$ implies $\tau \geq^{\to} \nu$ or $\nu = \sigma \to \mu$ and $\tau >^{\to} \mu$.
- a quasi-order $\geq_{\mathcal{F}}$ on $\mathcal{F} \cup \{@, \lambda\} \cup \mathcal{X}$ called *precedence* s.t.: (i) its strict part $>_{\mathcal{F}}$ restricts to $\mathcal{F} \cup \{@, \lambda\}$, is well-founded, and satisfies $(\forall f \in \mathcal{F})\ f >_{\mathcal{F}} @ >_{\mathcal{F}} \lambda$; (ii) its equivalence $=_{\mathcal{F}}$ contains pairs in $\mathcal{F} \times \mathcal{F}$ and all pairs $\{(x, x) \mid x \in \mathcal{X}\}$.
- $(\forall f \in \mathcal{F} \cup \{@, \lambda\})$, a status operator $\__f \in \{lex, mul\}$ in postfix index position such that $\__@ = mul$. Equivalent symbols have the same status.

The following auxiliary relations are used to define CPO [9] :

- $s \geq^X t$ iff $s =_\alpha t$ or $s >^X t$, for a set of variables X disjoint from $\mathrm{FV}(s)$, is the *main order*;
- $s >^X \overline{t}$ and $\overline{s} >^X t$ defined respectively as $(\forall v \in \overline{t})\, s >^X v$ and $(\exists u \in \overline{s})\, u >^X t$;
- $s : \sigma >^X t : \tau$ (resp., $s : \sigma \geq^X t : \tau$) for $s >^X t$ (resp., $s \geq^X t$) and $\sigma \geq^{\to} \tau$;
- we are interested in the *typed order* $s : \sigma >^\emptyset t : \tau$, written $s : \sigma > t : \tau$.

▶ **Definition 2.1.** Given $\Gamma \vdash_{\mathcal{F}} s : \sigma$ and $\Gamma \vdash_{\mathcal{F}} t : \tau$, $s >^X t$ iff either

$t \in X$ and $s \notin \mathcal{X}$	VAR
$s = \lambda x{:}\mu.u$ and $u\{x \mapsto z\} : \nu \geq^X t : \tau$	SUBTλ
$s = f(\overline{s})$, $t = \lambda x{:}\mu.v$ and $s >^{X \cup \{z:\mu\}} v\{x \mapsto z\}$	$\mathcal{F}\lambda$
$s = @(u, w)$, $t = \lambda x{:}\mu.v$, $x \notin \mathrm{FV}(v)$ and $s >^X v$	$@\lambda$
$s = \lambda x{:}\mu.u$, $t = \lambda y{:}\mu.v$ and $u\{x \mapsto z\} >^X v\{y \mapsto z\}$	$\lambda\lambda$
$s = @(\lambda x{:}\mu.u,\ w)$ and $u\{x \mapsto w\} \geq^X t$	BETA
$s = \lambda x{:}\mu.@(u, x)$, $x \notin \mathrm{FV}(u)$ and $u \geq^X t$	ETA

otherwise $s = f(\overline{s}), t = g(\overline{t})$ with $f, g \in \mathcal{F} \cup \{@, \lambda\} \cup \mathcal{X}$, and either

SUBT $\overline{s} : \overline{\sigma} \ \geq\ t : \tau$ PREC $f >_{\mathcal{F}} g$ and $s >^X \overline{t}$ STAT $f =_{\mathcal{F}} g,\ s >^X \overline{t}$ and $\overline{s} : \overline{\sigma} \ >_f\ \overline{t} : \overline{\tau}$

This definition of CPO is organized differently from [9] to be more compact. The last three cases originating in Dershowitz' recursive path ordering [12] describe the normal behaviour of head symbols, whether or not in $\mathcal{F}$. Note here that $>_f$ is the status extension (lexicographic or multiset) of the order $>$. The first 7 cases describe other behaviours, either specific (var, beta, eta), or using explicit α-conversion for the others. In its recursive call, case$\mathcal{F}\lambda$ increases the set X of upper variables, while other recursive calls either keep it unchanged or reset it to $\emptyset$. Relaxations of these recursive calls are indeed ill-founded.

▶ **Theorem 2.2** ([9]). *$>^+$ is a higher-order reduction ordering.*

3 Dependent algebras λ^{Π}_{Σ}

We move to a calculus with dependent types inspired by Edinburgh's LF [13], an extension of the simply typed λ-calculus which can be seen as a formal basis for dependently typed programming languages and their formal study as done with Elf [18].

In higher-order algebras, types are typable by a single, usually omitted constant TYPE. In dependent algebras, as in other type theories, types (called *type families*) are typed by *kinds*, TYPE being one of them, which describe their functional structure. Let $\mathcal{S}, \mathcal{F}$ and $\mathcal{V}$ be pairwise disjoint sets of respectively *type symbols*, *algebraic function symbols*, and *variables*. Algebraic function symbols in $\mathcal{F}$ may carry an arity upper-indexing their name. Type symbols in $\mathcal{S}$ are curried constants which kind may be functional. We use respectively f and a for a typical function or type symbol. Raw terms are given by the following grammar:

$$\begin{array}{rcl} \mathit{Kinds}\ K & := & \mathrm{TYPE} \mid \Pi x : A.K \\ \mathit{Types}\ A, B & := & a \mid \Pi x : A.B \mid \lambda x : A.B \mid A\ N \\ \mathit{Objects}\ M, N & := & x \mid \lambda x : A.M \mid M\ N \mid f^n(M_1, ..., M_n) \end{array}$$

λ and Π are LF's binders. Notation $=_\alpha$ stands for similarity. Here, λ and Π are binary operators, whose arguments are a type ($(\lambda x : A.u)|_1 = A$), and the body of the abstraction or product ($(\lambda x : A.u)|_2 = u$). Both may originate computations. We write f for $f^0()$.

3.1 Typing judgements

All LF expressions are typed, objects by types, types by kinds, kinds by a special untyped constant KIND (the *universe*) asserting their well-formedness, we also call them *valid*. Five kinds of judgement are recursively defined by the LF type system of Figure 2.

$\vdash_{sig} \Sigma$	Σ is a valid signature
$\Sigma \vdash_{\mathcal{C}} \Gamma$	Γ is a valid context assuming $\vdash_{sig} \Sigma$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}$	K is a valid kind assuming $\Sigma \vdash_{\mathcal{C}} \Gamma$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K$	type A has kind K assuming $\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : A$	object M has type A assuming $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} C\ :\ D$ and	$\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{K}} C\ :\ D$ are self explanatory

Environments pair up a signature and a context. Signatures assign kinds to type symbols and product types to function symbols. Contexts assign product types to variables.

$$\begin{array}{rcll} \mathit{Env}\ \Theta & := & \Sigma\,;\,\Gamma & \\ \mathit{Sig}\ \Sigma & := & \mathtt{nil} \mid \Sigma, a : K \mid \Sigma, f^n : \Pi\{x_i : A_i\}_n.A & \\ \mathit{Con}\ \Gamma & := & \mathtt{nil} \mid \Gamma, x : A & \end{array}$$

where `nil` is the empty set, $\Pi\{x_i : A_i\}_n.A$ is $\Pi x_1 : A_1.(\ldots(\Pi x_n : A_n.A)\ldots)$

In dependent calculi, the order of constants or variables in the environment is determined by their types. This impacts the expression of the so-called substitution lemma:

▶ **Lemma 3.1.** *Let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : A$ *and* $\Sigma\,;\,\Gamma, x : A, \Gamma' \vdash_{\mathcal{T}\vee\mathcal{O}} s\ :\ \sigma$. *Then,* $\Sigma\,;\,\Gamma, \Gamma'\{x \mapsto M\} \vdash_{\mathcal{T}\vee\mathcal{O}} s\{x \mapsto M\}\ :\ \sigma\{x \mapsto M\}$.

Applying the substitution lemma several times introduces an order on the application of elementary substitutions. We use the notation $M \circ^n \{x_i \mapsto N_i\}_i$ to denote the sequential application to M of the *elementary* substitutions $\{x_1 \mapsto N_1\}$, ..., $\{x_n \mapsto N_n\}$ in this order. We use the word *dependent substitution* to stress this sequential behaviour of substitutions.

Given a valid signature Σ and a context Γ valid in Σ, the set λ_Σ of valid expressions, called *terms* is the (disjoint) union of the sets $\mathcal{K}^{\Pi}_{\Sigma}$ of valid *kinds*, $\mathcal{T}^{\Pi}_{\Sigma}$ of valid *types* and $\mathcal{O}^{\Pi}_{\Sigma}$ of

Signatures

$$[\textsc{Empty}]\ \frac{}{\vdash_{sig} \mathtt{nil}} \qquad [\textsc{TConst}]\ \frac{\Sigma\,;\,\mathtt{nil} \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}}{\vdash_{sig} \Sigma,\ a : K}\ a \notin dom(\Sigma)$$

$$[\textsc{Const}]\ \frac{\vdash_{sig} \Sigma \qquad \Sigma\,;\,\{x_i : A_i\}_k \vdash_{\mathcal{T}} A_{k+1}(k = 0..n)\ :\ \mathrm{TYPE}}{\vdash_{sig} \Sigma,\ f^n : \Pi\{x_i : A_i\}_n.A_{n+1}}\ f \notin dom(\Sigma)$$

Contexts

$$[\textsc{Empty}]\ \frac{\vdash_{sig} \Sigma}{\Sigma \vdash_{\mathcal{C}} \mathtt{nil}} \qquad [\textsc{Var}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ \mathrm{TYPE}}{\Sigma \vdash_{\mathcal{C}} \Gamma,\ x : A}\ x \notin dom(\Gamma)$$

Kinds

$$[\textsc{Univ}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma}{\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} \mathrm{TYPE}\ :\ \mathrm{KIND}} \qquad [\textsc{Prod}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} \Pi x : A.K\ :\ \mathrm{KIND}}$$

Types

$$[\textsc{Ax}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} a\ :\ K}\ a : K \in \Sigma \qquad [\textsc{Abs}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{T}} B\ :\ K}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} \lambda x : A.B\ :\ \Pi x : A.K}$$

$$[\textsc{App}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ \Pi x : B.K \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : B}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\,M\ :\ K\{x \mapsto M\}} \qquad [\textsc{Prod}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{T}} B\ :\ \mathrm{TYPE}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} \Pi x : A.B\ :\ \mathrm{TYPE}}$$

$$[\textsc{Conv}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{K}} K'\ :\ \mathrm{KIND}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K'}\ K \equiv K'$$

Objects

$$[\textsc{Ax}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma \qquad x : A \in \Gamma}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} x : A} \qquad [\textsc{Fun}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M_i : A_i\{x_1 \mapsto M_1, ..., x_{i-1} \mapsto M_{i-1}\}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} f^n(M_1, ..., M_n) : A \circ^n \{x_i \mapsto M_i\}_i}\ \text{where } f^n : \Pi\{x_i : A_i\}_n.A \in \Sigma$$

$$[\textsc{Abs}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{O}} M : B}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} \lambda x : A.M : \Pi x : A.B} \qquad [\textsc{App}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : \Pi x : A.B \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{O}} N : A}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M\,N : B\{x \mapsto N\}}$$

$$[\textsc{Conv}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s\ :\ \sigma \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{K}} \sigma'\ :\ \mathrm{TYPE/KIND}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s\ :\ \sigma'}\ \sigma \equiv \sigma'$$

Figure 2 LF Typing rules.

valid *objects* (possibly abbreviated as $\mathcal{K}, \mathcal{T}, \mathcal{O}$). Our presentation of LF performs classically the necessary sanitary checks when forming the signatures and contexts and only those.

$\equiv$ denoting the *convertibility relation*, a congruence discussed next in more details, that is generated by $\beta\eta$-conversion on similar terms on the one hand and on the other hand by an additional arbitrary congruence between object terms stable by substitution (possibly identifying them all [4]). By its definition as a congruence, convertibility respects our syntactic categories, objects, types and kinds.

Our dependently typed calculus is referred to as λ^{Π}_{Σ} to stress the signature Σ, or simply LF.

Lexicography: we use K for kinds, A,B,C,D for types, M,N for objects, s,t,u,v,w,l,r for (objects or types), σ, τ, μ, ν for (type or kinds), and γ, θ for substitutions.

3.2 Dependent rewriting and convertibility

In LF, the usual rules of the λ-calculus apply at the object and type levels, making four different rules generating a congruence called *$\beta\eta$-convertibility*:

$$\begin{array}{lll|lll} \text{beta:} & (\lambda x\!:\!A.B)\ M & \to_{\beta_{\mathcal{T}}} \ B\{x \mapsto M\} & (\lambda x\!:\!A.M)\ N & \to_{\beta_{\mathcal{O}}} \ M\{x \mapsto N\} \\ \text{eta:} & \lambda x\!:\!A.(B\,x) & \to_{\eta_{\mathcal{T}}} \ B \ \text{ if } x \notin \mathrm{FV}(B) & \lambda x\!:\!A.(M\ x) & \to_{\eta_{\mathcal{O}}} \ M \ \text{ if } x \notin \mathrm{FV}(M) \end{array}$$

Convertibility plays a key role for typing via Conv. In LF, convertibility is defined as the congruence generated by $\beta\eta$-reductions. In reality, convertibility must be strengthened on object level terms in order to type most examples. This problem has been considered in the framework of the calculus of constructions with the Calculus on Inductive Constructions [17], the Calculus of Algebraic Constructions [7] and the Calculus of Constructions Modulo Theory [20, 3], for which convertibility includes respectively: primitive recursion at higher type generated by the user's inductive types; the user's higher-order rules; and a decidable object-level first-order theory like Presburger arithmetic. These frameworks can be restricted to the LF type system seen as a particular case of the calculus of constructions. In the context of LF, it *relates* to the liquid types discipline [19], which shares similar objectives.

We now introduce dependent rewrite rules and rewriting.

▶ **Definition 3.2.** Given a valid signature Σ, a *plain dependent rewriting system* is a set $\{\Delta_i \vdash l_i : \sigma_i \to r_i : \tau_i\}_i$ of quintuples made, for every index i, of a context Δ_i, lefthand and righthand side terms l_i, r_i, and terms σ_i, τ_i, s.t. $\mathrm{FV}(r_i) \subseteq \mathrm{FV}(l_i)$, $\Sigma\,;\, \Delta_i \vdash_{\mathcal{T}\vee\mathcal{O}} l_i \,:\, \sigma_i$ and $\Sigma\,;\, \Delta_i \vdash_{\mathcal{T}\vee\mathcal{O}} r_i \,:\, \tau_i$ with $\sigma_i \equiv \tau_i$. Δ_i, σ_i and τ_i may be omitted.

▶ **Definition 3.3** (Dependent rewriting)**.** Given a rewriting system R, one step rewriting is a relation over terms, written $\Sigma\,;\, \Gamma \vdash_R s \longrightarrow^p t$ (in short, $s \longrightarrow_R t$) defined as:

- s and t are both types or objects which are typable under $\Sigma;\ \Gamma$,
- $\Delta \vdash l \to r : A \in R$, where $\mathrm{FV}(\Delta) \cap \mathrm{FV}(\Gamma) = \emptyset$.
- $s = s[l \circ \gamma]_p$ and $t = s[r \circ \gamma]_p$, where γ is a dependent substitution wrt to Δ.

A major semantic property expected from rewrite rules is that rewriting preserves types. In presence of dependencies, types are usually preserved up to some congruence defined by the rules themselves, as in the Calculus of Inductive Constructions or the Calculus of Algebraic Constructions [7]. Here, preservation of typing by rewriting, up to *type erasures*, follows from Lemma 4.7.

▶ **Example 3.4.** Here is a simple example with dependent lists of elements of a given type A. We allow ourselves with some OBJ-like mixfix syntax, using "_" for arguments' positions:

$nat, A : \mathrm{TYPE};\ List : \Pi m : nat.\mathrm{TYPE};$
$0 : nat;\ \_ + 1 : \Pi n : nat.nat;\ \_ + \_ : \Pi\{m, n : nat\}.nat$
$cons : \Pi\{n : nat,\ a : A,\ l : List\ n\}.List\ (n + 1)$
$app : \Pi\{m, n : nat,\ k : List\, m,\ l : List\, n\}.\, List\, (m + n)$

$$0 + n \to n \qquad app(0, n, nil, l) \to l$$
$$(m + 1) + n \to (m + n) + 1 \quad app(m+1, n, cons(m, a, k), l) \to cons(m+n, a, app(m, n, k, l))$$

Using LF's typing rules given in Figure 2, we get:

$$\begin{array}{rcrcl}
\{m, n : nat\} & \vdash & n, 0 + n, (m + 1) + n, (m + n) + 1 & : & nat \\
\{n : nat, l : List\, n\} & \vdash & app(0, n, nil, l) & : & List\, (0 + n) \\
\{m, n : nat, k : List\, m, l : List\, n\} & \vdash & app(m+1, n, cons(m, a, k), l) & : & List\, ((m + 1) + n) \\
\{m, n : nat, k : List\, m, l : List\, n\} & \vdash & cons(m+n, a, app(m, n, k, l)) & : & List\, ((m + n) + 1)
\end{array}$$

Typing these rules requires a conversion relation extending $\beta\eta$-conversion on objects with Presburger arithmetic to identify $List\,(0+n), List\, n$ and $List\,((m+1)+n), List\,((m+n)+1)$.

4 Encoding LF in higher-order algebras

We define here a transformation from the source language to a target language which preserves non-termination of arbitrary reductions, not just β-reductions as in LF. Our target language is the simply-typed λ-calculus enriched with function symbols and type constants of Section 2, a choice which has three important advantages: the target vocabulary can be as close as possible from the source vocabulary; the transformation preserves termination as well as non-termination; the transformed rules can be checked for termination by CPO.

The higher-order algebra encoding λ^{Π}_{Σ} will be $\lambda^{\rightarrow}_{\Sigma^{flat}}$, where $\Sigma^{flat} = \mathcal{S}_{flat} \biguplus \Sigma_{flat}$ is a higher-order (non-dependent) signature whose two pieces are described next. The set of types in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ is denoted by $\mathcal{T}^{\rightarrow}_{\mathcal{S}_{flat}}$. Terms in $\lambda^{\rightarrow}_{\Sigma^{flat}}$, whose set is denoted by $\lambda_{\Sigma^{flat}}$, are meant to encode LF objects and types in a way which mimics dependently typed computations. We do not encode LF kinds, since computations in kinds are indeed computations on types or objects. Indeed, dependent types will be encoded in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ both as types and as terms.

4.1 Type erasures

Types. Types in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ are arrow types built from the set of sorts $\mathcal{S}_{flat} = \{*\} \cup \{a \mid a : K \in \Sigma\}$.

The new sort $*$ will serve to encode LF product types as terms of sort $*$ in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. LF objects will be encoded as terms whose types are erasures of LF types, as defined next.

Type erasures. The classical *erasing* transformation from families and kinds in λ^{Π}_{Σ} to types in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ eliminates dependencies from objects by replacing product types by arrow types:

▶ **Definition 4.1.** The *erasing* map $|.| : \mathcal{T} \cup \mathcal{K} \to \mathcal{T}^{\rightarrow}_{\mathcal{S}_{flat}}$ is defined as:

$$(1)|a| = a \qquad (2)|\Pi x : A.B| = |A| \to |B| \qquad (3)|\lambda x : A.B| = |B|$$
$$(4)|\mathrm{TYPE}| = * \qquad (5)|\Pi x : A.K| = |A| \to |K| \qquad (6)|A\, N| = |A|$$

Sorts being constants, an induction shows that variables in types are eliminated in rule (6):

▶ **Lemma 4.2.** *Let $A \in \mathcal{T} \cup \mathcal{K}$. Then $FV(|A|) = \emptyset$.*

▶ **Corollary 4.3.** *For any $E \in \mathcal{K} \cup \mathcal{T}$, $y \in \mathcal{V}$, and $s \in \mathcal{O}^{\rightarrow}_{\Sigma^{flat}}$, $|E\{y \mapsto s\}| = |E|\{y \mapsto |s|\} = |E|$.*

▶ **Lemma 4.4** (Conversion equality)**.** *Let $D, E \in \mathcal{T}$ such that $D \equiv E$. Then, $|D| = |E|$.*

Proof. Conversion is a congruence generated by $\beta\eta$-rewriting and an equivalence $=_{\mathcal{O}}$ on object terms. We show that the property is true of both relations by induction on D.

1. Case $D = a : K$. Then $E = D$.
2. Case $D = \Pi x : A.B : \text{TYPE}$. Then $E = \Pi x : A'.B' : \text{TYPE}$ with $A \equiv A'$ and $B \equiv B'$. By induction hypothesis (and definition of the erasing map).
3. Case $D = \lambda x : A.B : \Pi x : A.K$. There are two cases:
   (a) $E = \lambda x : A'.B' : \Pi x : A'.K$, with $A \equiv A'$ and $B \equiv B'$. By induction hypothesis.
   (b) $D = \lambda x{:}A.(E\ x) \to_\eta E$ where $x \notin \text{FV}(E)$. Then $|D| = |\lambda x{:}A.(E\,x)| = |E\,x| = |E|$.
3. Case $D = A\ M : K$, where M is an object. Again two cases:
   (a) $E = A'\ M'$, $A \equiv A'$ and $M \equiv M'$. Then, $|D| = |A|$ and $|E| = |A'|$. By induction.
   (b) $D = (\lambda x{:}F.G)\ M \to_\beta G\{x \mapsto M\} = E$. By Corol. 4.3, $|D| = |G| = |G\{x \mapsto M\}| = |E|$.

◀

Erasing is extended to environments $\Sigma\,;\Gamma$ by: $|\Gamma| = \{x : |A| \mid x : A \in \Gamma\}$ and $|\Sigma| = \{a : |K| \mid a : K \in \Sigma\} \cup \{f^n : |A_1| \to \ldots |A_n| \to |A| \text{ where } f^n : \Pi\{x_i : A_i\}_n.A \in \Sigma\}$.

Note here that a constructor like $cons : \Pi n : Nat, x : Nat, l : List(n).List(n+1) \in \Sigma$ becomes $cons : Nat \to Nat \to List \to List \in |\Sigma|$. Eliminating the first (type) argument from *cons* would easily allow writing rules for which termination is not preserved by the transformation. This does not mean, however, that writing such rules is actually impossible.

4.2 Term flattening

Besides types and function symbols in $|\Sigma|$, the signature Σ_{flat} will contain algebraic symbols, called *flattening constructors*, used to mimic LF's abstraction and product.

$$\Sigma_{flat} = |\Sigma| \cup \{lo^2_\sigma : * \to \sigma \to \sigma, \quad lf^2_\sigma : * \to \sigma \to \sigma, \quad pif^2_\sigma : * \to (\sigma \to *) \to * \quad | \quad \sigma \in \mathcal{T}^{\to}_{\mathcal{S}_{flat}}\}$$

We may omit subscripts σ, τ in constructor's names, and use lof^2 for $(lo^2_\sigma / lf^2_\sigma)$. The first argument of the flattening constructors, of type $*$, is the flattening of some dependent type A. The second argument of lo^2_σ is the interpretation of some object $M : A$, hence $\sigma = |A|$, while that of lf^2_σ is the interpretation of a type $A : K$, hence $\sigma = |K|$. Since signatures are monomorphic and σ is arbitrary, the flattening constructors must be indexed by these types. We now define a *flattening* transformation for expressions of LF which are typed in some environment left unspecified to expressions of $\lambda^{\to}_{\Sigma^{flat}}$:

▶ **Definition 4.5.** The *flattening* function $\|\_$ from λ_Σ to $\lambda_{\Sigma^{flat}}$ (the context in which the input term is valid is omitted) is defined as

$$\begin{array}{rcl|rcl} \|x : A & = & x : |A| & \|\Pi x : A.B : \text{TYPE} & = & pif^2_{|A|}(\|A, \lambda x : |A|.\|B) \\ \|M\ N : A & = & @(\|M, \|N) & \|\lambda x : A.M : \Pi x : A.B & = & \lambda x : |A|.lo^2_{|B|}(\|A, \|M) \\ \|A\ N : K & = & @(\|A, \|N) & \|\lambda x : A.B : \Pi x : A.K & = & \lambda x : |A|.lf^2_{|K|}(\|A, \|B) \\ \|a : K & = & a & \|f^n(M_1, ..., M_n) : A & = & f^n(\|M_1, ..., \|M_n) \end{array}$$

Since flattening is not surjective, we denote by $\|\lambda_\Sigma \subset \lambda_{\Sigma^{flat}}$ its target.

Note that type symbols in $\mathcal{S}$ become both sorts in $\mathcal{S}_{flat}$ and function symbols in Σ_{flat}, with the same (overloaded) name. This definition obeys the following principles: (i) flattening is a homomorphism, hence commutes with substitutions; (ii) because types may depend on objects, the type information associated with bound variables must be recorded by the encoding to preserve non-termination; (iii) the types of the flattening constructors are compatible with the erasing transformation, which allows to trace the syntactic categories in the transformed world; (iv) the encoding of abstractions and products preserves their variable's binding, but two different encodings are used. The encoding of abstractions is an

abstraction, in order to preserve beta-redexes via the transformation. Nothing like that is needed for product types which cannot be applied, and can therefore be transformed into terms of sort $|\text{TYPE}| = *$. In that case, the abstraction is encapsulated in the flattening constructor. This allows to single out easily products' encodings in the flattened world. Now,

an LF object $M : A$ is translated as a term $\|M$ of type $|A|$,
an LF type $A : K$ is translated as both a type $|A|$ and a term $\|A$ of type $|K|$,
an LF kind $K : \text{KIND}$ is translated as a type $|K|$.

▶ **Lemma 4.6.** *The following properties hold:*

1. *soundness: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A : K$. *Then* $\Sigma_{flat}\,;\,|\Gamma| \vdash \|A : |K|$;
2. *soundness: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : A$. *Then* $\Sigma_{flat}\,;\,|\Gamma| \vdash \|M : |A|$;
3. *preservation: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s : \sigma$. *Then* $FV(s) = FV(\|s)$;
4. *stability: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s, t : \sigma, \tau$ *and* $x : \tau \in \Gamma$. *Then* $\|s\{x \mapsto t\} = \|s\{x \mapsto \|t\}$.

Proof. The first three are proved by induction on the typing derivations of A, M and s respectively, using Lemma 4.4 for the third (translation of products) and for the conversion rule. Stability follows by induction on the typing derivation of s. ◀

4.3 Preservation of reductions by flattening

Our goal now is to show that the reductions on objects and types in λ^{Π}_{Σ} are mimicked in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. Since rewriting a term cannot not increase its set of free variables, rewriting commutes with the η-rule. A consequence is that, given a dependent rewrite system R, $\longrightarrow_{\beta\eta R}$ terminates iff $\longrightarrow_{\beta R}$ terminates: encoding the η-rule will not be necessary. To ease the reading, we use $\rightharpoonup$ for our rewriting symbol in $\lambda^{\rightarrow}_{\Sigma^{flat}}$, and decorate subterms in flattened rules by their type.

$$[\beta_{\mathcal{O}}] \quad @(\lambda x : \sigma.lo^2_\tau(A : *, M : \tau), N : \sigma) \rightharpoonup M\{x \mapsto N\}$$
$$[\beta_{\mathcal{T}}] \quad @(\lambda x : \sigma.lf^2_\tau(A : *, B : \tau), N : \sigma) \rightharpoonup B\{x \mapsto N\}$$

In these rules, A and σ are the term flattening and type erasure of the same dependent type D, a relationship that cannot be expressed in $\lambda^{\rightarrow}_{\Sigma^{flat}}$, hence is not kept in the transformed rules. For example, assuming $A, s, t \in \|\lambda_\Sigma$ and $|\|A^{-1}| \neq \sigma$, then $u = @(\lambda x : \sigma.lof^2(A, s), t)$ is a redex which has no counter-part in λ^{Π}_{Σ}. There are indeed new rewrites in the flattened world, making preservation of non-termination a weaker property.

We use $[R]$, $[\beta]$, $[\beta R]$ and $\beta[\beta R]$ for $\{\|l \rightharpoonup \|r \,|\, l \to r \in R\}$, $\{[\beta_{\mathcal{T}}], [\beta_{\mathcal{O}}]\}$, $[\beta]\cup[R]$, and $\{\beta\}\cup[\beta R]$.

As with A, M, N, σ, τ used in $[\beta]$, variables of R that denote expressions in λ^{Π}_{Σ} keep their name in $[R]$, denoting now expressions of $\lambda^{\rightarrow}_{\Sigma^{flat}}$ belonging to the same syntactic categories as in the dependent world. Then, an instance of a $[\beta]$-rule may not be the encoding of an instance of the β-rule in the dependent world if M/B match flattened terms which are no encoding of dependent expressions. Despite these approximations, reductions are preserved:

▶ **Lemma 4.7.** *Let* $\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s : \sigma$, $s \to_{\beta R} t$ *and* $\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} t : \tau$. *Then,* $|\sigma| = |\tau|$ *and* $\|s \rightharpoonup_{[\beta R]} \|t$.

Proof. The proof is by induction on the typing derivation of s. All cases are by induction except those where the βR redex is at the top. We carry out four typical cases:

1. $[\text{App}]\ \dfrac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A : \Pi x{:}B.K \quad \Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : B}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} s = A\,M : K\{x \mapsto M\}}$ and $M \to_{\beta R} M'$, hence $A\,M \to_{\beta R} A\,M' = t$. By induction hypothesis, $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} M' : B'$, $|B'| = |B|$ and $\|M \rightharpoonup_{[\beta R]} \|M'$. By definition of flattening, $\|A\,M = @(\|A, \|M) : |K\{x \mapsto M\}| = |K|$ by Corollary 4.3. By

definition of erasing, $|\Pi x : B.K| = |B| \to |K|$. By Lemma 4.6 (soundness) $\|A : |B| \to |K|$ and $\|M' : |B'|$. Finally, $\|t = @(\|A, \|M') : |K|$, we are done.

2. $[\textsc{Conv}]\ \dfrac{\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s : \sigma' \quad \Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{K}} \sigma : \text{TYPE/KIND}}{\Sigma\,;\,\Gamma\vdash_{\mathcal{O}} s : \sigma}\ \sigma \equiv \sigma'.$ By induction hypothesis, $\Sigma;\Gamma\vdash_{\mathcal{T}\vee\mathcal{O}} t:\tau$, $|\tau|=|\sigma'|$ and $\|s \rightharpoonup_{[\beta R]} \|t$. By Lemma 4.6, $\|t:|\tau|$. Lemma 4.4 concludes.

3. $[\textsc{App}]\ \dfrac{\Sigma\,;\,\Gamma\vdash_{\mathcal{O}} \lambda x : A.u : \Pi x : A.B \qquad \Sigma\,;\,\Gamma\vdash_{\mathcal{O}} M : A}{\Sigma\,;\,\Gamma\vdash_{\mathcal{O}} s = \lambda x : A.u\,M : B\{x \mapsto M\}}$ and $t = u\{x \mapsto M\}$. By inversion, $\Sigma\,;\Gamma\vdash_{\mathcal{O}} u:B$, thus $t:B\{x\mapsto M\}$ by Lemma 3.1. $\|\lambda x:A.u\,M = @(\lambda x:A.lo^2_{|B|}(\|A, \|u), \|M)$ $\rightharpoonup_{[\beta]} \|u\{x \mapsto \|M\}$ by definition of $[\beta_{\mathcal{O}}]$. Lemma 4.6 and Corollary 4.3 conclude.

4. $\Sigma\,;\,\Gamma\vdash_{\mathcal{T}\vee\mathcal{O}} s\,:\,\sigma$, $s = l\circ\gamma$ and $t = r\circ\gamma$ for some $l \to r \in R$ such that l, r have convertible types μ, ν in their environment. By Lemma 3.1 applied repeatedly, s, t have types $\sigma = \mu\circ\gamma$ and $\tau = \nu \circ \gamma$, and since $\equiv$ is a congruence, $\sigma \equiv \tau$. By Lemma 4.6(stability), $\|l\gamma = \|l\|\gamma$ and $\|r\gamma = \|r\|\gamma$ hence $\|s \rightharpoonup_{[R]} \|t$ by definition of $\rightharpoonup_{[R]}$. ◀

This Lemma contains the analog of the type-preservation property of non-dependent rewriting, equivalence by conversion being here equivalence modulo type erasures: subject reduction holds for dependent rewriting modulo type erasures.

Thus λ^{Π}_{Σ} is terminating for an empty set R, implying strong normalisation of $\beta\eta$-reductions at both object and type level, for any convertibility relation $\equiv$ containing $\beta\eta$-convertibility. In particular, $\equiv$ can contain Presburger arithmetic, an important known extension of LF.

▶ **Example 4.8.** Here are the transformed signature and rules for our example on Lists:

$$\begin{array}{rcl||rcl||rcl}
nat, A & : & * & 0 & : & nat & _+1 & : & nat \to nat \\
List & : & nat \to * & nil & : & List & + & : & nat \to nat \to nat \\
\multicolumn{3}{r||}{cons : nat \to A \to List \to List} & \multicolumn{3}{r||}{} & app & : & nat \to nat \to List \to List \to List \\
\multicolumn{3}{r||}{0 + n \to n} & \multicolumn{6}{l}{(m+1) + n \to (m+n) + 1} \\
\multicolumn{3}{r||}{app(0, n, nil, l) \to l} & \multicolumn{6}{l}{app(m{+}1, n, cons(m, a, k), l) \to cons(m{+}n, a, app(m, n, k, l))}
\end{array}$$

These dependently typed rewrite rules being algebraic, their encoding is the identity. CPO proves their termination with $app >_{\mathcal{F}} cons >_{\mathcal{F}} nil >_{\mathcal{F}} + >_{\mathcal{F}} \_ + 1 >_{\mathcal{F}} 0$, and $List >^{\to} \{nat, A\}$.

▶ **Theorem 4.9.** *Given a signature Σ, a dependent term rewriting system $\beta\eta R$ is terminating in λ^{Π}_{Σ} if its flattening $\beta\|\beta R$ is terminating in $\lambda^{\to}_{\Sigma^{flat}}$.*

Proof. By using Lemma 4.7 and a commutation argument for η. ◀

4.4 Inverse encoding

Lemma 4.7 justifies our method for checking strong normalization by a transformation to a higher-order algebra where we can use standard techniques including CPO. But the flattened world is richer than the dependent world, there are more terms and rewrites. Nonetheless, we can also show that termination is partly preserved by defining an inverse transformation such that composing both is the identity. First, we define the inverse transformation on the subset $\|\lambda_{\Sigma} \subseteq \lambda_{\Sigma^{flat}}$, hence allowing us to show that flattening is injective.

▶ **Definition 4.10.** The inverse $\|\_^{-1} : \|\lambda_{\Sigma} \to \lambda_{\Sigma}$ is defined as:

$$\begin{array}{rcl||rcl||rcl}
\|x^{-1} & = & x & \|a^{-1} & = & a & \|f^n(s_1, ..., s_n)^{-1} & = & f^n(\|s_1{}^{-1}, ..., \|s_n{}^{-1}) \\
\|@(s,\, t)^{-1} & = & \|s^{-1}\ \|t^{-1} & & & & \|pif^2(s, \lambda x : \sigma.t)^{-1} & = & \Pi x : \|s^{-1}.\|t^{-1} \\
\|\lambda x{:}\sigma.lf^2(s,t)^{-1} & = & \lambda x{:}\|s^{-1}.\|t^{-1} & & & & \|\lambda x : \sigma.lo^2_{\tau}(s,t)^{-1} & = & \lambda x : \|s^{-1}.\|t^{-1}
\end{array}$$

▶ **Lemma 4.11** (Reversibility). *Assume* $\Sigma;\Gamma \vdash s:\sigma$ *in* λ^{Π}_{Σ}. *Then* $\Sigma;\Gamma \vdash \|(\|s)^{-1} = s:\sigma$.

Proof. By induction on s. Variables, type constants, applications and pre-algebraic terms are clear. We carry out one remaining case, the others being similar. $\|(\|\Pi x:A.B)^{-1} = \|pif^2_{|A|}(\|A, \lambda x:|A|.\|B)^{-1} = \Pi x:\|(\|A)^{-1}.\|(\|B)^{-1}$. We conclude by induction. ◀

▶ **Corollary 4.12.** *Let* $\sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s:\sigma$ *and* $\|s|_p \in \|\lambda_\Sigma$ *for some* p. *Then* $\|s|_p = \|s|_q$ *for some* q.

Flattening being an injection from λ_Σ to $\lambda_{\Sigma^{flat}}$, hence a bijection between λ_Σ and its target $\|\lambda_\Sigma$, provides strong evidence that λ^{Π}_{Σ} is faithfully encoded by flattening. It follows that, if $\|s = \|t$ for $s,t \in \lambda_\Sigma$, then $s = t$ since $s = \|\|s^{-1} = \|\|t^{-1} = t$. We apply to Lemma 4.6:

▶ **Corollary 4.13** (inverse stability). *Assume* $u\{x \mapsto v\} \in \|\lambda_\Sigma$. *Then,* $\|u\{x \mapsto v\}^{-1} = \|u^{-1}\{x \mapsto \|v^{-1}\}$.

Therefore, rewrites in $\|\lambda_\Sigma$ can be mapped back to λ_Σ, showing both that $\|\lambda_\Sigma$ is closed under rewriting by $[\beta R]$, and that termination in λ_Σ implies termination in $\|\lambda_\Sigma$:

▶ **Lemma 4.14.** *Let* $\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s:\sigma$ *and* $\|s \rightharpoonup_{[\beta R]} u$. *Then* $u = \|t$ *for some* t *st* $s \to_{\beta R} t$.

Proof. By assumption $\|s|_p = \|l\gamma \rightharpoonup_{[\beta R]} u|_p$ for some $p \in \mathcal{P}os(\|s)$, where $l \to r \in \{\beta\} \cup R$, γ a substitution in $\|\lambda_\Sigma$ and $u|_p = \|r\gamma$. By Corollary 4.12, $\|s|_p = \|s|_q$ for some q. By Lemma 4.11, $s|_q = \|\|l\gamma^{-1} = \|\|l^{-1}\|\gamma^{-1}$ (by Corollary 4.13) $= l\theta$ with $\|\theta = \gamma$ by Lemma 4.11. Hence $s \longrightarrow_{\beta R} s[r\theta]_q = t$. Now, since flattening is a homomorphism, $\|t = \|s[\|r\theta]_p$ (by definition of q) $= \|s[\|r\gamma]_p$ (by definition of θ) $= u$. ◀

This does not prove, however, that termination in λ^{Π}_{Σ} implies termination in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. The problem is that flattening is non-surjective, since many terms, like those headed by lof^2, are no flattening of a dependent term. Further, some seemingly good-looking terms, like $pif^2_\sigma(s, \lambda x{:}\sigma.t)$ may not be either, even assuming that s,t are themselves flattening of dependent terms. This is the case because the flattened signature checks that term s has type $*$, hence is the encoding of some dependent type A, but cannot check whether $|A| = \sigma$ as it should. The same happens with applications and pre-algebraic terms: (s,t) may not be typable in λ^{Π}_{Σ} when s,t are typable in λ^{Π}_{Σ} and $@(\|s, \|t)$ is typable in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. Indeed, termination in λ^{Π}_{Σ} does not imply termination in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ as shown by this example: let o : TYPE; $List$: Πx : o.TYPE; $a, b : o$; $la : List\, a$; $lb : List\, b$; $f^2 : \Pi m : o, l : List\, m\, .\, o$; and $\{f(a, la) \to f(b, lb);\ b \to a, lb \to la\}$, whose dependent derivations are all finite since b cannot rewrite to a in $f(b, lb)$, a non-typable term. In the flattened signature, o and $List$ are sorts, and $a, b : o$; $la, lb : List$; $f^2 : o \to List \to o$; and $\{f(a, la) \to f(b, lb);\ b \to a;\ lb \to la\}$. We now have the following infinite derivation: $f(a, la) \to f(b, lb) \to f(a, lb) \to f(a, la) \ldots$. Restricting rewrites on parameters could be a solution.

5 The Dependent Computability Path Ordering

DCPO is an extension of CPO obtained by adding new cases for products. All notations for DCPO are simply and systematically obtained by replacing the CPO ordering notation $>$ by $\succ$. For example, the precedence will be denoted by $\succ_\Sigma$. Type families are compared alternatively as types with $\succeq^{\Pi}$ and as terms with $\succ$. The basic ingredients of DCPO are:

- a *precedence* $\succeq_\Sigma$ on $\mathcal{F} \cup \mathcal{S} \cup \{@, \lambda, \Pi\}$ s.t. $\succ_\Sigma$ is well-founded, and $(\forall f \in \mathcal{F}) f \succ_\Sigma @ \succ_\Sigma \{\lambda, \Pi\}$.
- a status $_f \in \{Mul, Lex\}$ for every symbol $f \in \mathcal{F} \cup \{@\}$ with $_@ \in Mul$.
- a quasi-order $\succeq^{\Pi}$ on $\mathcal{K} \cup \mathcal{T}$, whose strict part $\succ^{\Pi}$ and equivalence $=^{\Pi}$ satisfy:

- **(i)** *compatibility*: $\equiv \subseteq =^{\Pi}$;
- **(ii)** *well-foundedness*: $\succ^{\Pi} \cup \{\langle \Pi x : A.\sigma, A\rangle \mid A \in \mathcal{T},\ \sigma \in \mathcal{T} \cup \mathcal{K}\}$ is well-founded;
- **(iii)** *product body subterm*: $(\forall \sigma \in \mathcal{T} \cup \mathcal{K})\, \Pi x : A.\sigma \succ^{\Pi} \sigma$;
- **(iv)** *product preservation*: $|\Pi x\!:\!A.\sigma| {=^{\Pi}} |\tau|$ implies $\tau = \Pi x\!:\!B.\mu$ for some B, μ, such that $|A| {=^{\Pi}} |B|, |\sigma| {=^{\Pi}} |\mu|$;
- **(v)** *decreasingness*: $\Pi x\!:\!A.\sigma \succeq^{\Pi} \tau$ implies $\sigma \succeq^{\Pi} \tau$ or else $\tau = \Pi x\!:\!B.\nu$, $A {=^{\Pi}} B$ and $\tau \succeq^{\Pi} \nu$.

Building quasi-orders $\succeq^{\Pi}$ on types and kinds with a non-trivial equivalence $=^{\Pi}$ is not hard [14].

We now define a first version of DCPO which can be justified by using CPO [9]. We will then discuss briefly an enhanced version justified by a more elaborated version of CPO [10]. In the following definition, z denotes a fresh variable of type A/B.

▶ **Definition 5.1** (DCPO). Given $\Gamma \vdash_{\Sigma} s : \sigma$ and $\Gamma \vdash_{\Sigma} t : \tau$, then $s \succ^{X} t$ iff either:

1. $t \in X$ and $s \notin \mathcal{X}$ (*var*)
2. $s = (u\ N)$ and $u \succeq^{X} t$ or $N : B \succeq^{X} t : \tau$ (*subt@*)
3. $s = \lambda/\Pi x : A.u$ and $u\{x \mapsto z\} : \mu \succeq^{X} t : \tau$ (*subt*$\lambda\Pi$)
4. $s = f(\overline{s})$, $f \in \Sigma$, $t = (\lambda/\Pi y\!:\!B.v)\!:\!\Pi y\!:\!B.K$, and $s \succ^{X} B$ and $s \succ^{X \cup \{z\}} v\{y \mapsto z\}$ (Σ*prec*$\lambda\Pi$)
5. $s = \lambda x : A.u : \Pi x : A.C$, $t = \lambda y : B.v : \Pi y : B.D$, $|A| = |B|$, $A \succ^{X} B$ and $u\{x \mapsto z\} : C \succ^{X} v\{y \mapsto z\} : D$ (*stat*λ)
6. $s = \Pi x\!:\!A.u$, $t = \Pi y\!:\!B.v$, $|A| = |B|$, $A \succ^{X} B$ and $u\{x \mapsto z\} \succ^{X} v\{y \mapsto z\}$ (*stat*Π)
7. $s = ((\lambda x : A.u) : (\Pi x : A.C)\ \ w : A)$, and $u\{x \mapsto w\} \succeq t$ (*beta*)
8. otherwise $s = f(\overline{s}), t = g(\overline{t})$ with $f, g \in \mathcal{F} \cup \mathcal{S} \cup \{@\} \cup \mathcal{X}$, and either of (*rpo*)
   $\overline{s}\!:\!\overline{\sigma} \succeq t\!:\!\tau$ (*subt*) $\quad f >_{\mathcal{F}} g$ and $s \succ^{X} \overline{t}$ (*prec*) $\quad f =_{\Sigma} g$, $s \succ^{X} \overline{t}$ and $\overline{s}\!:\!\overline{\sigma}\ (\succ)_f\ \overline{t}\!:\!\overline{\tau}$ (*stat*)

All terms built by DCPO are well-typed under the assumption that both starting terms s, t are well-typed. Note (i) the importance of all our assumptions on $\succeq^{\Pi}$, see for example Case stat$\lambda\Pi$; (ii) the order may recursively compare objects with types, even when the input comparison operates on expressions in $\mathcal{T}^2 \cup \mathcal{O}^2$; (iii) compared products cannot be kinds.

5.1 Example

We now consider our list example, skipping the rules on natural numbers which do not have dependencies. We shall use the (user defined) precedence $app \succ_{\Sigma} \{+, cons, nil\}$, multiset status for app, and a quasi-order on types in which, for all $n : nat$, then $List\, n \succeq^{\Pi} A$ and $List\, m {=^{\Pi}} List\, n$ for all m, n of type nat. Such a type order is easy to get by using a restricted RPO on type erasures, which equivalence therefore contains Presburger arithmetic. See [14]. The goal $app(0, n, nil, l) : List\, 0 + n \succ l : List\, n$ is easy, although already requiring identification of $0 + n$ in $List\, 0 + n$ with n in $List\, n$. We proceed with the second goal:
$app(m{+}1, n, cons(m, a, k), l) : List\,(m{+}1){+}n \succ cons(m{+}n, a, app(m, n, k, l))\!:\!List\,(m{+}n){+}1$
The type comparison $List\,(m+1)+n \succeq^{\Pi} List\,(m+n)+1$ succeeds by using our type ordering which indeed equates both types, and we are left with the term comparison: $app(m+1, n, cons(m, a, k), l) \succ cons(m+n, a, app(m, n, k, l))$ Using *(prec)*, we get three subgoals, which are processed in turn, using indentation to identify the dependencies between recursive calls, the used Case being indicated between parentheses:

$app(m+1,n,cons(m,a,k),l) \succ m+n$ $(prec)$
$app(m+1,n,cons(m,a,k),l) \succ m$ $(subt)$
$m+1 : nat \succeq m : nat$ which yields
$nat \succeq^{\Pi} nat$ which succeeds and $m+1 \succeq m$ which succeeds by $(subt)$
$app(m+1,n,cons(m,a,k),l) \succ n$ which succeeds by $(subt)$
$app(m+1,n,cons(m,a,k),l) \succ a$ $(subt)$
$cons(m,a,k) : List\,(m+1) \succ a : A$ which yields
$List\,(m+1) \succeq^{\Pi} A$ which succeeds and $cons(m,a,k) \succ a$ which succeeds by $(subt)$
$app(m+1,n,cons(m,a,k),l) \succ app(m,n,k,l)$ $(stat)$
$\{m+1 : nat, n : nat, cons(m,a,k) : List\ m+1, l : List\ n\}$
$\succ_{mul} \{m : nat, n : nat, k : List\ m, l : List\ n\}$ which yields
$m+1 : nat \succ m : nat$ and $cons(m,a,k) : List\ m+1 \succ k : List\ m$, for the reader.

5.2 Properties of DCPO

▶ **Lemma 5.2** (Monotonicity, stability). *Let $s,t \in \lambda_\Sigma$ st $s : \sigma \succ t : \sigma$, $C(x:\sigma)$ a context, and γ a substitution. Then $C\{x \mapsto s\gamma\} \succ C\{x \mapsto t\gamma\}$.*

Proof. By induction on the definition of $s \succ t$ and stability of the type order for monotonicity. By induction on the context, and use of the status rules for stability. ◀

As anticipated, we now reduce the well-foundedness of DCPO to the well-foundedness of CPO by using Theorem 4.9, termination in the target higher-order algebra being checked by CPO. To this end, we need to show that, whenever $s : \sigma \succ t : \tau$, then $\|s : |\sigma| > \|t : |\tau|$.

We start showing that an order on types and kinds of LF satisfying the requirements for DCPO becomes naturally an order on types of $\lambda^{\rightarrow}_{\Sigma^{flat}}$ satisfying the requirements for CPO.

▶ **Definition 5.3.** Given $\sigma,\tau \in \mathcal{T}^{\rightarrow}_{\mathcal{S}_{flat}}$, let $\sigma \succeq^{\rightarrow}_{\Sigma^{flat}} \tau$ iff $\exists \mu,\nu \in \mathcal{T} \cup \mathcal{K}$, $|\mu| = \sigma$, $|\nu| = \tau$, and $\mu \succeq^{\Pi} \nu$.

Transitivity is clear. Of course, different choices of μ,ν may sometimes lead to contradictory orderings for σ,τ, hence the equality $=^{\rightarrow}_{\Sigma^{flat}}$ may be strictly larger than the corresponding equality $=^{\Pi}$. This has indeed no negative impact since the strict part $\succ^{\Pi}$ is never used in comparisons. Further, the properties of $\succeq^{\Pi}$ transfer naturally to $\succeq^{\rightarrow}_{\Sigma^{flat}}$. Soundness follows:

▶ **Lemma 5.4.** *Assume $\succeq^{\Pi}$ is a DCPO type order. Then $\succeq^{\rightarrow}_{\Sigma^{flat}}$ is a CPO type order.*

The set Σ^{flat} of function symbols of Σ^{flat} is the union of $\mathcal{S}$, $\mathcal{F}$ and the flattening constructors. The precedence $>_{\Sigma^{flat}}$ is obtained by letting the flattening constructors be equivalent minimal symbols. The strict part of $>_{\Sigma^{flat}}$ is clearly well-founded, while its equivalence is increased by the equality of the flattening constructors. We take status *Mul* for lo^2, lf^2, pif^2.

▶ **Lemma 5.5.** *Let $s:\sigma, t:\tau$ and $X \subseteq \mathcal{X}$ such that $s \succ^X t$ by any DCPO rule. Then $\|s >^X \|t$.*

Proof Sketch. The proof is by induction on the definition of DCPO, assuming by induction hypothesis that the property holds at every recursive call. Note that if $s : \sigma$, $t : \tau$, $\sigma \succeq^{\Pi} \tau$ and $\|s >^X \|t$, then $\|s : |\sigma| >^X \|t : |\tau|$ by Definition 5.3. ◀

We then obtain the second main result of the paper as a corollary:

▶ **Theorem 5.6** (Well-foundedness of DCPO). *$\succ^{\Pi}$ is well-founded.*

Proof. By theorem 4.9. Note that we use the full strength of that result, including the need for type-level rules instances of the various DCPO rules instances which compare types. This is possible since we only need preservation of non-termination by the flattening transformation, that is, the if direction of Theorem 4.9. ◀

It follows that $\succ^{\Pi} \cup \eta$ is well-founded. One may wonder why we did not include η-rules in the definition of DCPO, since it is in the definition of CPO. The reason is the comparison of lefthand and righthand sides of $[\eta]$, $\lambda x{:}|A|.lof^2(\|A,\ @(\|B,\ x)) > \|B$ which does not go through: the type comparison of the subgoal $lof^2(\|A,\ @(\|B,\ x)){:}\sigma > \|B{:}|A| \to \sigma$ fails.

5.3 A realistic example

We consider here a more complex, non-algebraic, higher-order example. Given a list l of natural numbers $x_1, ..., x_m$, and a natural number y, a higher-order variable g which is meant to be instantiated by $+$, we define a higher-order function *foldr* such that $(foldr(m, l, y)\ g)$ calculates $g(x_1, g(x_1, ...g(x_m, y)))$, while $(map(m,\ l)\ f)$ calculates the list $f(x_1), ..., f(x_m)$.

$nat : \mathrm{TYPE} \quad | \quad 0 : nat \quad | \quad +1 : \Pi x : nat.nat$
$List : \Pi m{:}nat.\mathrm{TYPE} \quad | \quad nil : List\ 0 \quad | \quad cons : \Pi\{m{:}nat,\ x{:}nat,\ l{:}List\ m\}.List\ m+1$
$map : \Pi\{m : nat,\ l : List\ m\}.(\Pi f : (\Pi x : nat.nat).List\ m)$
$foldr : \Pi\{m : nat,\ l : list\ m,\ y : nat\}.\Pi g : (\Pi\{x_1 : nat,\ x_2 : nat\}.nat).nat$

$map(0, l) \to \lambda f : (\Pi x{:}nat.nat).nil \quad | \quad foldr(0,\ l,\ y) \to \lambda g{:}\Pi\{x_1{:}nat,\ x_2{:}nat\}.nat.y$
$map(m+1,\ cons(m,\ x,\ l)) \to \lambda f : \Pi\{x : nat\}.nat.cons((f\ x),\ map(m,\ l))$
$foldr(m+1,\ cons(m,\ z,\ l),\ y) \to \lambda g : \Pi\{x_1 : nat,\ x_2 : nat\}.nat.(g\ z\ (foldr(m,\ l,\ y)\ g))$

To carry out the example, we let $List >_{\mathcal{T}} nat$, $foldr > map > cons > nil > lf^2 > lo^2 > List > nat$ and $(\forall \sigma >_{\mathcal{T}} \tau)\ lf^2_\sigma > lf^2_\tau$ and $lo^2_\sigma > lo^2_\tau$. We consider the last rule only.

CPO comparison. It generates the following goals (omitting the type comparisons):

$\|l > \lambda g : nat \to nat \to nat.lo^2_{nat}$
$(lf^2_{nat}(nat,\ \lambda x_1 : nat.lf^2_{nat}(nat,\ \lambda x_2 : nat.nat)),\ @(@(g, z), @(foldr(m,\ l,\ y)\ g)))$ $\mathcal{F}\lambda$
$\|l >^{\{g\}} lo^2_{nat}(lf^2_{nat}(nat, \lambda x_1{:}nat.lf^2_{nat}(nat, \lambda x_2{:}nat.nat)), @(@(g, z), @(foldr(m, l, y), g)))$
$\|l >^{\{g\}} lf^2_{nat}(nat,\ \lambda x_1 : nat.lf^2_{nat}(nat,\ \lambda x_2 : nat.nat))$ PREC
$\|l >^{\{g\}} nat$ succeeds by PREC
$\|l >^{\{g\}} \lambda x_1 : nat.lf^2_{nat}(nat,\ \lambda x_2 : nat.nat)$ $\mathcal{F}\lambda$
$\|l >^{\{g,x_1\}} lf^2_{nat}(nat,\ \lambda x_2 : nat.nat)$ PREC
$\|l >^{\{g,x_1\}} nat$ succeeds by PREC
$\|l >^{\{g,x_1\}} \lambda x_2 : nat.nat$ $\mathcal{F}\lambda$
$\|l >^{\{g,x_1,x_2\}} nat$ succeeds by PREC
$\|l >^{\{g\}} @(@(g, z), @(foldr(m,\ l,\ y)\ g))$ PREC
$\|l >^{\{g\}} @(g, z)$ PREC
$\|l >^{\{g\}} g$ succeeds by VAR
$\|l = foldr(m+1,\ cons(m,\ z,\ l),\ y) >^{\{g\}} z$ SUBT
$cons(m, z, l) : List(m+1) >^{\{g\}} z : nat$ which succeeds by SUBT
$\|l >^{\{g\}} @(foldr(m,\ l,\ y), g)$ PREC
$\|l = foldr(m+1,\ cons(m,\ z,\ l),\ y) >^{\{g\}} foldr(m,\ l,\ y)$ STAT
$\{m+1,\ cons(m,\ z,\ l),\ y\}(>^{\{g\}})_{mul}\{m,\ l,\ y\}$ which succeeds by repeated SUBT
$\|l >^{\{g\}} g$ which succeeds by VAR, therefore ending the computation successfully.

DCPO comparison. We now carry out the same computation with DCPO directly:

$foldr(m+1, cons(m, z, l), y) \succ \lambda g{:}\Pi\{x_1{:}nat, x_2{:}nat\}.nat.(g\ z\ (foldr(m, l, y)\ g))$ (Σprec$\lambda\Pi$)
$l \succ \Pi\{x_1 : nat,\ x_2 : nat\}.nat$ (Σprec$\lambda\Pi$)
$l \succ nat$ which succeeds by (prec)
$l \succ^{\{x_1\}} \Pi x_2 : nat.nat$ (Σprec$\lambda\Pi$)
$l \succ^{\{x_1\}} nat$ which succeeds by (prec)

$l \succ^{\{x_1,x_2\}} nat$ which succeeds by (prec)
$l \succ^{\{g\}} (g\ z\ (foldr(m,\ l,\ y)\ g))$ -we keep the @ operator implicit this time- (prec)
$l \succ^{\{g\}} g$ (var)
$l \succ^{\{g\}} z$ (subt)
$cons(m, z, l) : List(m+1) \succ^{\{g\}} z : nat$ succeeds by (subt)
$l \succ^{\{g\}} (foldr(m,\ l,\ y)\ g)\}$ (prec)
$l \succ^{\{g\}} foldr(m,\ l,\ y)$ (stat)
$\{m+1,\ cons(m,\ z,\ l),\ y\} \succ^{\{g\}} \{m, l, y\}$ which succeed by repeated (subt)
$l \succ^{\{g\}} g$ succeeds by (var), therefore ending the computation successfully.

6 Conclusion

The amount of research work targeting automatic termination is vast. Among the most popular techniques are dependency pairs, introduced by Aart and Giesl and the size-changing principle, pioneered by Neil Jones, which have been generalized to dependently-typed rules [5]. Dependent types can also be *used* to store annotations useful for proving termination [22]. Despite these proposals that recent prototypes try to combine, techniques used in Coq and Agda are still poor, as acknowledged by the authors on their websites. Using our techniques would improve this situation.

Our first contribution is a new transformation for eliminating type dependencies using a framework richer than the simply-typed λ-calculus, which provides with a natural encoding, and allows us to consider arbitrary rewrite rules, not only β- and η-reductions. Furthermore, these results hold for a practical dependent type system made richer than LF's via a convertibility relation possibly stronger than $\beta\eta$-convertibility, hence allowing us to type many more terms. This easily implementable transformation allow us using existing implementations targeting termination of rules in presence of simple types. The transformation also allows us to show well-foundedness of DCPO, a version of CPO applying to dependently typed terms directly. This is done by considering pairs ordered by DCPO as dependently typed rewrite rules to which the transformation applies. Note that DCPO will naturally benefit from improvements of CPO, without changing the proof technique. In particular, we could easily accommodate size interpretations by using them as a precedence as in [11], or type level rules such as $s = (A\ u)$, $t = \Pi y : B.v$, with $s \succ^X B$ and $s \succ^{X\cup\{z\}} v\{y \mapsto z\}$ flattened as $\|s = @(\|A, \|u)$, and $\|t = pik_{|B|}(\|B, \lambda y : |B|.\|v)$, with $\|s >^X \|B$ and $\|s >^{X\cup\{z\}} \|v\{y \mapsto z\}$, whose justification requires the extension of CPO with *small* function symbols, here $pik_{|B|}$, which behave as if they were smaller than application and abstraction [10]. We can then break the main goal into the above solvable subgoals.

Our main interest is indeed to prove termination directly at the dependent type level. Using DCPO allows the programmer, in case of failure, to get an error message in her/his own dependently typed syntax, rather than in the transformed syntax as would be the case when using CPO on the transformed rules. To our knowledge, this is the very first general – Coq and Agda's techniques are very limited –, purely syntactic method that allows one to show termination of a set of dependently typed rewrite rules via computations taking place on the user's dependently typed rules.

Acknowledgements. Work supported by NSFC grants 61272002, 91218302, 973 Program 2010CB328003 and Nat. Key Tech. R&D Program SQ2012BAJY4052 of China. This work was done in part during a visit of the first author supported by Tsinghua University.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

2 H. P. Barendregt. Lambda calculus with types. In *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford Univ. Press, 1993.

3 B. Barras, J.-P. Jouannaud, P.-Y. Strub, and Q. Wang. CoqMTU: A higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *LICS*, pages 143–151. IEEE Computer Society, 2011.

4 G. Barthe. The relevance of proof-irrelevance. In *ICALP*, volume 1443 of *LNCS*, pages 755–768. Springer, 1998.

5 F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *RTA*, volume 3091 of *LNCS*, pages 24–39. Springer, 2004.

6 F. Blanqui. Definitions by rewriting in the calculus of constructions. *CoRR*, /abs/cs/0610065, 2006.

7 F. Blanqui, J.-P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *RTA*, volume 1631 of *LNCS*, pages 301–316. Springer, 1999.

8 F. Blanqui, J.-P. Jouannaud, and A. Rubio. HORPO with computability closure : A reconstruction. In *LPAR*, volume 4790 of *LNCS*. Springer, 2007.

9 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *CSL*, *LNCS* 5213. 2008.

10 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering. To appear in LMCS.

11 C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *LPAR*, volume 2250 of *LNCS*, pages 531–547. Springer, 2001.

12 Nachum Dershowitz. Orderings for term-rewriting systems. *TCS*, 17:279–301, 1982.

13 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

14 J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *J. ACM*, 54(1), 2007.

15 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *CoRR*, abs/1109.5468, 2011.

16 U. Norell. Dependently typed programming in Agda. In *TLDI*, pages 1–2. ACM, 2009.

17 C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

18 F. Pfenning. Elf: A language for logic definition and verified metaprogramming. otoIn *In Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 313–322. 1989.

19 P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169. ACM, 2008.

20 P.-Y. Strub. Coq modulo theory. In *CSL*, volume 6247 of *LNCS*, pages 529–543. Springer, 2010.

21 S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *CoRR*, abs/1109.4357, 2011.

22 H. Xi. Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

23 H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, 1999.

Well-Founded Recursion over Contextual Objects

Brigitte Pientka[1] and Andreas Abel[2]

1 **School of Computer Science**
McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

2 **Department of Computer Science and Engineering, Gothenburg University**
Göteborg, Sweden
andreas.abel@gu.se

Abstract

We present a core programming language that supports writing well-founded structurally recursive functions using simultaneous pattern matching on contextual LF objects and contexts. The main technical tool is a coverage checking algorithm that also generates valid recursive calls. To establish consistency, we define a call-by-value small-step semantics and prove that every well-typed program terminates using a reducibility semantics. Based on the presented methodology we have implemented a totality checker as part of the programming and proof environment Beluga where it can be used to establish that a total Beluga program corresponds to a proof.

1998 ACM Subject Classification D.3.1[Programming Languages] Formal Definitions and Languages. F.3.1[Logics and Meaning of Programs] Specifying and Verifying and Reasoning about Programs

Keywords and phrases Type systems, Dependent Types, Logical Frameworks

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.273

1 Introduction

Mechanizing formal systems and their proofs play an important role in establishing trust in formal developments. A key question in this endeavor is how to represent variables and assumptions to which the logical framework LF [8], a dependently typed lambda-calculus, provides an elegant and simple answer: both can be represented uniformly using LF's function space, modelling binders in the object language using binders in LF. This kind of encoding is typically referred to as *higher-order abstract syntax* (HOAS) and provides a general uniform treatment of syntax, rules and proofs.

While the elegance of higher-order abstract syntax encodings is widely acknowledged, it has been challenging to reason inductively about LF specifications and formulate well-founded recursion principles. HOAS specifications are not inductive in the standard sense. As we recursively traverse higher-order abstract syntax trees, we extend our context of assumptions, and our LF object does not remain closed. To tackle this problem, Pientka and collaborators [11, 4] propose to pair LF objects together with the context in which they are meaningful. This notion is then internalized as a contextual type $[\Psi.A]$ which is inhabited by terms M of type A in the context Ψ [9]. Contextual objects are then embedded into a computation language which supports general recursion and pattern matching on contexts and contextual objects. Beluga, a programming environment based on these ideas [13], facilitates the use of HOAS for non-trivial applications such as normalization-by-evaluation [4] and a type-preserving compiler including closure conversion and hoisting [3]. However, nothing in this work enforces or guarantees that a given program is total.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 273–287

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we develop a core functional language for reasoning inductively about context and contextual objects. One can think of this core language as the target of a Beluga program: elaboration may use type reconstruction to infer implicit indices [6] and generate valid well-founded recursive calls that can be made in the body of the function. Type checking will guarantee that we are manipulating well-typed objects and, in addition, that a given set of cases is covering and the given recursive calls are well-founded. To establish consistency, we define a call-by-value small-step semantics for our core language and prove that every well-typed program terminates, using Tait's method of logical relations. Thus, we justify the interpretation of well-founded recursive programs in our core language as inductive proofs. Based on our theoretical work, we have implemented a totality checker for Beluga.

Our approach is however more general: our core language can be viewed as a language for first-order logic proofs by structural induction over a given domain. The domain must only provide answers to three domain-specific questions: (1) how to unify objects in the domain, (2) how to split on a domain object and (3) how to justify that a domain object is smaller according to some measure. The answer to the first and second question allows us to justify that the given program is covering, while the third allows us to guarantee termination. For the domain of contextual LF presented in this paper, we rely on higher-order unification [2] for (1), and our splitting algorithm (2) and subterm ordering (3) builds on previous work [5, 10]. As a consequence, our work highlights that reasoning about HOAS representations via contextual types can be easily accommodated in a first-order theory. In fact, it is a rather straightforward extension of how we reason inductively about simple domains such as natural numbers or lists.

The remainder of the paper is organized as follows. We first present the general idea of writing and verifying programs to be total in Sec. 2 and then describe in more detail the foundation of our core programming language which includes well-founded recursion principles and simultaneous pattern matching in Sec. 3. The operational semantics together with basic properties such as type safety is given in Sec. 4. In Sec. 5, we review contextual LF [4], define a well-founded measure on contextual objects and contexts, and define the splitting algorithm. Subsequently we describe the generation of valid well-founded recursive calls generically, and prove normalization (Sec. 7). We conclude with a discussion of related work, current status and future research directions. Due to space constraints, proofs have been omitted.

2 General Idea

2.1 Example 1: Equality on Natural Numbers

To explain the basic idea of how we write inductive proofs as recursive programs, we consider first a very simple example: reasoning about structural equality on natural numbers (see Listing 1). We encode natural numbers and equality on them in the logical framework LF.

Listing 1 Encoding of an Inductive Proof as a Recursive Function.

```
nat : type.                                    eq   : nat → nat → type.
z   : nat.                                     eq_z : eq z z.
s   : nat → nat.                               eq_s : eq M N → eq (s M) (s N).
.
ref : Π M:nat. [eq M M] = Λ M ⇒ rec-case M of
|                              ref  z     ⇒ [eq_z]
|  M':nat ; ref M':[eq M' M']. ref (s M') ⇒ let  D = ref M' in [eq_s M' M' D];
```

The free variables M and N in the definition of eq_s are implicitly quantified at the outside. Program ref proves reflexivity of eq: for all M:nat we can derive eq M M. Following type-theoretic notation, we write Π for universal quantification; we embed LF objects which denote base predicates via [ ]. Abstraction over LF object M is written Λ M ⇒ in our language. Using **rec-case**, we prove inductively that for all M there is a derivation for [eq M M]. There are two cases to consider: ref z describes the base case where M is zero and the goal refines to [eq z z]. In this case, the proof is simply [eq_z]. In the step case, written as ref (s M'), we also list explicitly the other assumptions: the type of M' and the induction hypothesis written as ref M':[eq M' M']. To establish that [eq (s M') (s M')], we first obtain a derivation D of eq M' M' by induction hypothesis and then extend it to a derivation [eq_s M' M' D] of [eq (s M') (s M')]. We highlight in green redundant information which can be inferred automatically. In the pattern, it is the typing (here: M':nat) of the pattern variables [12, 6] and the listing of the induction hypotheses. The dot "." separates these assumptions from the main pattern. For clarity, we choose to write the pattern as a simultaneous pattern match and make the name of the function explicit; in practice, we only write the main pattern which is left in black, and all other arguments are inferred.

2.2 Example 2: Intrinsically Typed Terms

Next, we encode intrinsically typed λ-terms. This example does exploit the power of LF.

```
tp   : type.                          tm  : tp → type.
bool : tp.                            lam : (tm A → tm B) → tm (arr A B).
arr  : tp → tp → tp.                  app : tm (arr A B) → tm A → tm B.
```

We define base types such as bool and function types, written as arr T S, and represent simply-typed lambda-terms using the constructors lam and app. In particular, we model the binding in the lambda-calculus (our object language) via HOAS, using the LF function space. For example, the identity function is represented as lam λx.x and function composition as lam λg. lam λf. lam λx. app (f (app g x)). As we traverse λ-abstractions we record the variables we are encountering in a context ϕ : cxt. Its shape is given by a *schema declaration* **schema** ctx = tm A stating that it contains only variable bindings of type tm A for some A. To reason about typing derivations, we package the term (or type) together with its context, forming a contextual object (or contextual type, resp.). For example, we write $\phi \vdash$ tm A for an object of type tm A in the context ϕ. Such *contextual types* are embedded into logical statements as [$\phi \vdash$ tm A]. When the context ϕ is empty, we may drop the turnstile and simply write [tm A].

Counting constructors: Induction on (contextual) LF object

As an example, we consider counting constructors in a term. This corresponds to defining the overall size of a typing derivation. We recursively analyze terms M of type tm A in the context ϕ. In the variable case, written as count ϕ B ($\phi \vdash$ p …), we simply return zero. The pattern variable p stands for a variable from the context ϕ. We explicitly associate it with the identity substitution, written as …, to use p which has declared type $\phi \vdash$ tm B in the context ϕ. Not writing the identity substitution would enforce that the pattern variable does not depend on ϕ and forces the type of p to be [$\vdash$ tm B]. While it is certainly legitimate to use p in the context ϕ, since the empty substitution maps variables from the empty context to ϕ, the type of p is empty; since the context is empty, there are no variables of the type [$\vdash$ tm B]. Hence writing ($\phi \vdash$ p) would describe an empty pattern. In contrast, types described by

Listing 2 Counting constructors.

```
count: Π ϕ:ctx. Π A:tp. Π M:(ϕ ⊢ tm A) . [ nat] =
Λ ϕ ⇒ Λ A ⇒ Λ M ⇒ rec-case M of
| B:tp, p:(ϕ ⊢ tm B); . count ϕ B (ϕ ⊢ p …) ⇒ [ z ]                    % Variable Case
| B:tp,C:tp,M:(ϕ,x:tm B ⊢ tm C) ;                                      % Abstraction Case
  count (ϕ,x:tm B) C (ϕ,x:tm B ⊢ M … x) : [nat].                      % IH
  count ϕ (arr B C) (ϕ ⊢ lam  B C λx. M… x) ⇒
  let  X = count (ϕ,x:tm B) C (ϕ,x:tm B ⊢ M… x) in [ s X ]
| B:tp,C:tp,M:(ϕ ⊢ tm (arr B C)), N:(ϕ ⊢ tm B) ;                       % Application Case
  count ϕ (arr B C) (ϕ ⊢ M …):[nat],                                  % IH1
  count ϕ        B  (ϕ ⊢ N …):[nat].                                  % IH2
  count ϕ C (ϕ ⊢ app  B C  (M …) (N …)) ⇒
  let  X = count ϕ (arr B C) (ϕ ⊢ M …) in
  let  Y = count ϕ        B  (ϕ ⊢ N …) in add (s X) Y
```

Listing 3 Computing length of a context.

```
length = Π ϕ:ctx. [nat] =
Λ ϕ ⇒ rec-case ϕ of
|                                . count ∅           ⇒ [ z ]
|  ψ:ctx, A:tp ; count ψ: [nat] . count (ψ, x:tm A) ⇒ let X = count ψ in [ s X ]
```

meta-variables `A` or `B`, for example, are always closed and can be instantiated with any closed object of type `tp` and we do not associate them with an identity substitution.

In the case for lambda-abstractions, `count ϕ (arr B C) (ϕ ⊢ lam λx.M … x)`, we not only list the type of each of the variables occurring in the pattern, but also the induction hypothesis, `count (ϕ,x:tm B) C (ϕ,x:tm B ⊢ M … x):[nat]`. Although the context grows, the term itself is smaller. In the body of the case, we use the induction hypothesis to determine the size `X` of `M … x` in the context `ϕ, x:tm B` and then increment it.

The case for application, `count ϕ C (ϕ ⊢ app B C (M …)(N …))`, is similar. We again list all the types of variables occurring in the pattern as well as the two induction hypotheses. In the body, we determine the size `X` of `(ϕ ⊢M …)` and the size `Y` of `(ϕ ⊢N …)` and then add them.

Computing the length of a context: Induction on the context

As we have the power to abstract and manipulate contexts as first-class objects, we also can reason inductively about them. Contexts are similar to lists and we distinguish between the empty context, written here as $\emptyset$, and a context consisting of at least one element, written as `ψ, x:tm A`. In the latter case, we can appeal to the induction hypothesis on ψ (see Listing 3).

3 Core language with well-founded recursion

In this section, we present the core of Beluga's computational language which allows the manipulation of contextual LF objects by means of higher-order functions and primitive recursion. In our presentation of the computation language we keep however our domain abstract simply referring to U, the type of a domain object, and C, the object of a given domain. In fact, our computational language is parametric in the actual domain. To guarantee totality of a program, the domain needs to provide answers for two main questions: 1) how to split on a domain type U and 2) how to determine whether a domain object C is smaller according to some domain-specific measure. We also need to know how to unify two terms and determine when two terms in our domain are equal. In terms of proof-theoretical strength, the language is comparable to Gödel's T or Heyting Arithmetic where the objects

of study are natural numbers. However in our case, U will stand for a (contextual) LF type and C describes a (contextual) LF object.

Types	$\mathcal{I}, \tau$	$::= [U] \mid \tau_1 \to \tau_2 \mid \Pi X{:}U.\tau$
Expressions	e	$::= y \mid [C] \mid \mathsf{fn}\, y{:}\tau \Rightarrow e \mid e_1\ e_2 \mid \Lambda X{:}U \Rightarrow e \mid e\ C$
		$\mid \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \mid \mathsf{rec{-}case}^{\mathcal{I}} C\ \mathsf{of}\ \overrightarrow{b}$
Branches	b	$::= \Delta; \overrightarrow{r}\ .\ r \Rightarrow e$
Assumptions	r	$::= f\ \overrightarrow{C}\ C$
Contexts	Γ	$::= \cdot \mid \Gamma, y{:}\tau \mid \Gamma, r : \tau$
Meta Context	Δ	$::= \cdot \mid \Delta, X{:}U$

We distinguish between computation variables, simply referred to as variables and written using lower-case letter y; variables that are bound by Π-types and Λ-abstraction are referred to as meta-variables and written using upper-case letter X. Meta-variables occur inside a domain object. For example we saw earlier the object `(ψ ⊢ app B C (M …)(N …))`. Here, ψ, `B`, `C`, `M`, and `N` are referred to as meta-variables.

There are three forms of computation-level types τ. The base type $[U]$ is introduced by wrapping a contextual object C inside a box; an object of type $[U]$ is eliminated by a let-expression effectively unboxing a domain object. The non-dependent function space $\tau_1 \to \tau_2$ is introduced by function abstraction $\mathsf{fn}\, y{:}\tau_1 \Rightarrow e$ and eliminated by application $e_1\ e_2$; finally, the dependent function type $\Pi X{:}U.\tau$ which corresponds to universal quantification in predicate logic is introduced by abstraction $\Lambda X{:}U \Rightarrow e$ over meta-variables X and eliminated by application to a meta objects C written as $e\,C$. The type annotations on both abstractions ensure that every expression has a unique type. Note that we can index computation-level types τ only by meta objects (but this includes LF contexts!), not by arbitrary computation-level objects. Thus, the resulting logic is just first-order, although the proofs we can write correspond to higher-order functional programs manipulating HOAS objects.

Our language supports pattern matching on a meta-object C using $\mathsf{rec{-}case}$-expressions. Note that one cannot match on a computational object e directly; instead one can bind an expression of type $[U]$ to a meta variable X using let and then match on X. We annotate the recursor $\mathsf{rec{-}case}$ with the type of the inductive invariant $\Pi\Delta_0.\tau_0$ which the recursion satisfies. Since we are working in a dependently-typed setting, it is not sufficient to simply state the type U of the scrutinee. Instead, we generalize over the index variables occurring in the scrutinee, since they may be refined during pattern matching. Hence, Δ_0 is $\Delta_1, X_0{:}U_0$ where Δ_1 exactly describes the free meta-variables occurring in U_0. The intention is that we induct on objects of type U_0 which may depend on Δ_1. Δ_0 must therefore contain at least one declaration. We also give the return type τ_0 of the recursor, since it might also depend on Δ_0 and might be refined during pattern matching. This is analogous to Coq's `match_as_in_return_with_end` construct.

One might ask whether this form of inductive invariant is too restrictive, since it seems not to capture, e.g., $\Pi\Delta_0.(\tau \to \Pi X{:}U_0.\tau')$. While allowing more general invariants does not pose any fundamental issues, we simply note here that the above type is isomorphic to $\Pi(\Delta_0, X{:}U_0).\,\tau \to \tau'$ which is treated by our calculus. Forcing all quantifiers at the outside simplifies our theoretical development; however, our implementation is more flexible.

A branch b_i is expressed as $\Delta_i; \overrightarrow{r_i}\ .\ r_{i0} \Rightarrow e_i$. As shown in the examples, we explicitly list all pattern variables (i.e. meta-variables) occurring in the pattern in Δ_i. In practice, they often can be inferred (see for example [12]). We also list all valid well-founded recursive calls $\overrightarrow{r_i}$, i.e. $r_{ik}, \ldots, r_{i1}$, for pattern r_{i0}. In practice, they can be also derived dynamically while we check that a given pattern r_{i0} is covering and we give an algorithm in Section 6.

$\boxed{\Delta;\Gamma \vdash e : \tau}$: Computation e has type τ $\quad \dfrac{\Gamma(y) = \tau}{\Delta;\Gamma \vdash y : \tau} \quad \dfrac{\Gamma(r) = \tau \quad r = r'}{\Delta;\Gamma \vdash r' : \tau}$

$$\frac{\Delta \vdash C : U}{\Delta;\Gamma \vdash [C] : [U]} \quad \frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau} \quad \frac{\Delta;\Gamma \vdash e : \Pi X{:}U.\tau \quad \Delta \vdash C : U}{\Delta;\Gamma \vdash e\ C : [\![C/X]\!]\tau}$$

$$\frac{\Delta;\Gamma, y{:}\tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \mathsf{fn}\ y{:}\tau_1 \Rightarrow e : \tau_1 \to \tau_2} \quad \frac{\Delta, X{:}U;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda X{:}U \Rightarrow e : \Pi X{:}U.\tau} \quad \frac{\Delta;\Gamma \vdash e_1 : [U] \quad \Delta, X{:}U;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\frac{\mathcal{I} = \Pi\Delta_1.\Pi X_0{:}U_0.\tau_0 \quad \Delta \vdash C : [\![\theta]\!]U_0 \quad \Delta \vdash \theta : \Delta_1 \quad b_i : \mathcal{I}\ (\text{for all } i) \quad \overrightarrow{b}\ \mathtt{covers}\ \mathcal{I}}{\Delta;\Gamma \vdash \mathsf{rec{-}case}^{\mathcal{I}}\ C\ \mathsf{of}\ \overrightarrow{b} : [\![\theta, C/X]\!]\tau_0}$$

$\boxed{b : \mathcal{I}}$: Branch b satisfies the invariant $\mathcal{I}$

$$\frac{\text{for all } 0 \le j \le k\ .\ \Delta \vdash_{\mathcal{I}} r_j : \tau_j \qquad \Delta\ ;\ r_k{:}\tau_k, \ldots, r_1{:}\tau_1 \vdash e : \tau_0}{(\Delta; r_k\ \ldots\ r_1\ .\ r_0 \Rightarrow e) : \mathcal{I}}$$

$\boxed{\Delta \vdash_{\mathcal{I}} r : \tau'}$ and $\boxed{\Delta \vdash \overrightarrow{C} : \mathcal{I} > \tau'}$: Assumption r/pattern spine $\overrightarrow{C}$ has type τ' given $\mathcal{I}$

$$\frac{\Delta \vdash \overrightarrow{C} : \mathcal{I} > \tau'}{\Delta \vdash_{\mathcal{I}} f\ \overrightarrow{C} : \tau'} \quad \frac{\Delta \vdash C : U \quad \Delta \vdash \overrightarrow{C} : [\![C/X]\!]\tau > \tau'}{\Delta \vdash C\ \overrightarrow{C} : \Pi X{:}U.\tau > \tau'} \quad \frac{\Delta \vdash C : U}{\Delta \vdash C : \Pi Y{:}U.\tau > [\![C/Y]\!]\tau}$$

Figure 1 Type system for dependently-typed functional computation language.

The identifier f in assumptions r denotes the local function that is essentially introduced by $\mathsf{rec{-}case}$; this notation is inspired by primitive recursion in *Tutch* [1]. Currently, it just improves the readability of call patterns; however, it is vital for extensions to nested recursion.

3.1 Computation-level Type System

In the typing judgement (Fig. 1), we distinguish between the context Δ for meta-variables from our index domain and the context Γ which includes declarations of computation-level variables. We will tacitly rename bound variables, and maintain that contexts declare no variable more than once. Moreover, we require the usual conditions on bound variables. For example in the rule for Λ-abstraction the meta-variable X must be new and cannot already occur in the context Δ. This can always be achieved via α-renaming. Similarly, in the rule for function abstraction, the variable y must be new and cannot already occur in Γ. We have two variable rules to look up a computation-level variable y and an induction hypothesis r. To verify that the induction hypothesis r' has type τ and its use is valid, we simply check whether there exists $r : \tau$ in Γ where $r = r'$. For now it suffices to think of $=$ as syntactically equivalent.

The most interesting rule is the one for recursion: given the invariant $\mathcal{I} = \Pi\Delta_1.\Pi X{:}U_0.\tau_0$ the expression $\mathsf{rec{-}case}^{\mathcal{I}}\ C\ \mathsf{of}\ \vec{b}$ is well-typed under three conditions: First, the meta-object C we are recursing over has some type U and moreover, U is an instance of the type specified in the invariant, i.e. $\Delta_0 = \Delta_1, X{:}U_0$ and $U = [\![\theta]\!]U_0$ for some meta-substitution θ with domain Δ_1. Secondly, all branches b_i are well-typed with respect to the given invariant $\Pi\Delta_0.\tau_0$. Finally, $\vec{b}$ must cover the meta-context Δ_0, i.e., it must be a complete, non-redundant set of patterns covering Δ_0, and all recursive calls are well-founded. Since the coverage check is domain specific, we leave it abstract for now and return to it when we consider (contextual) LF as one possible domain (see Sec. 5).

Note that we drop the meta-context Δ and the computation context Γ when we proceed to check that all branches satisfy the specified invariant. Dropping Δ is fine, since we require

$$\overline{(\mathsf{fn}\, x{:}\tau \Rightarrow e)\ v \longrightarrow [v/x]e} \qquad \overline{(\Lambda X{:}U \Rightarrow e)\ C \longrightarrow [\![C/X]\!]e} \qquad \overline{\mathsf{let}\ X = [C]\ \mathsf{in}\ e \longrightarrow [\![C/X]\!]e}$$

$$\frac{\exists\, \text{unique}\ (\Delta.r_k, \ldots, r_1.r_0 \Rightarrow e) \in \vec{b} \text{ where } r_j = f\ \overrightarrow{C_j}\ C_{j0} \text{ such that } \vdash C \doteq C_{00}/\theta}{\mathsf{rec{-}case}^{\mathcal{I}}\ C\ \mathsf{of}\ \vec{b} \longrightarrow [\![\theta]\!][\,(\mathsf{rec{-}case}^{\mathcal{I}}\ C_{k0}\ \mathsf{of}\ \vec{b})/r_k, \ldots, (\mathsf{rec{-}case}^{\mathcal{I}}\ C_{10}\ \mathsf{of}\ \vec{b})/r_1\,]e}$$

Figure 2 Small-step semantics $e \longrightarrow e'$.

the invariant $\Pi\Delta_0.\tau_0$ to be closed. One might object to dropping Γ; indeed this could be generalized to keeping those assumptions from Γ which do not depend on Δ and generalizing the allowed type of inductive invariant (see our earlier remark).

For a branch $b = \Delta; \vec{r}.r_0 \Rightarrow e$ to be well-typed with respect to a given invariant $\mathcal{I}$, we check the call pattern r_0 and each recursive call r_j against the invariant and synthesize target types τ_j ($j \geq 0$). We then continue checking the body e against τ_0, i.e., the target type of the call pattern r_0, populating the computation context with the recursive calls $\vec{r}$ at their types $\vec{\tau}$. A pattern / recursive call $r_j = f\ \overrightarrow{C_j}$ intuitively corresponds to the given inductive invariant $\mathcal{I} = \Pi\Delta_1.\Pi X_0{:}U_0.\tau_0$, if the spine $\overrightarrow{C}$ matches the specified types in $\Delta_1, X_0{:}U_0$ and it has intuitively the type $[\![C_{jn}/X_n, \ldots, C_{j0}/X_0]\!]\tau_0$ which we denote with τ'_j.

More generally, we write $\Delta \vdash \theta : \Delta_0$ for a well-typed simultaneous substitution where Δ_0 is the domain and Δ is the range of the substitution. It can be inductively defined (see below) and the standard substitution lemmas hold (see for example [4]).

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \theta : \Delta_0 \qquad \Delta \vdash C : [\![\theta]\!]U}{\Delta \vdash \theta, C/X : \Delta_0, X : U}$$

4 Operational Semantics

Fig. 2 specifies the call-by-value (cbv) one-step reduction relation $e \longrightarrow e'$; we have omitted the usual congruence rules for cbv. Reduction is deterministic and does not get stuck on closed terms, due to completeness of pattern matching in $\mathsf{rec{-}case}$. To reduce $(\mathsf{rec{-}case}^{\mathcal{I}}\ C\ \mathsf{of}\ \vec{b})$ we find the branch $(\Delta.r_k, \ldots, r_1.r_0 \Rightarrow e) \in \vec{b}$ such that the principal argument C_{00} of its clause head $r_0 = f\ \overrightarrow{C_0}\ C_{00}$ matches C under meta substitution θ. The reduct is the body e under θ where we additionally replace each place holder r_j of a recursive call by the actual recursive invocation $(\mathsf{rec{-}case}^{\tau}\ [\![\theta]\!]C_{j0}\ \mathsf{of}\ \vec{b})$. The object C_{j0} in fact just denotes the meta-variable on which we are recursing. We also apply θ to the body e. In the rule, we have lifted out θ. *Values* v in our language are boxed meta objects $[C]$, functions $\mathsf{fn}\, x{:}\tau \Rightarrow e$, and $\Lambda X{:}U.e$.

▶ **Theorem 1** (Subject reduction). *If* $\cdot;\cdot \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\cdot;\cdot \vdash e' : \tau$.

Proof. By induction on $e \longrightarrow e'$. ◀

▶ **Theorem 2** (Progress). *If* $\cdot;\cdot \vdash e : \tau$ *then either* e *is a value or* $e \longrightarrow e'$.

Proof. By induction on $\cdot;\cdot \vdash e : \tau$. ◀

5 Contextual LF: Background, Measure, Splitting

If we choose as our domain natural numbers or lists, it may be obvious how to define splitting together with a measure that describes when an object is smaller. Our interest however is to

use the contextual logical framework LF [9] as a general domain language. Contextual LF extends the logical framework LF [8] by packaging an LF objects M of type A together with the context Ψ in which it is meaningful. This allows us to represent rich syntactic structures such as open terms and derivation trees that depend on hypotheses. The core language introduced in Sec. 3 then allows us to implement well-founded recursive programs over these rich abstract syntax trees that correspond to proofs by structural induction.

5.1 Contextual LF

We briefly review contextual LF here. As usual we consider only objects in η-long β-normal form, since these are the only meaningful objects in LF. Further, we concentrate on characterizing well-typed terms; spelling out kinds and kinding rules for types is straightforward.

LF Base Types	P, Q	$::= c \cdot S$
LF Types	A, B	$::= P \mid \Pi x{:}A.B$
Heads	H	$::= c \mid x \mid p[\sigma]$
Neutral Terms	R	$::= H \cdot S \mid u[\sigma]$
Spines	S	$::= \mathsf{nil} \mid M\ S$
Normal Terms	M, N	$::= R \mid \lambda\, x.\, M$
Substitutions	σ	$::= \cdot \mid \mathsf{id}_\psi \mid \sigma, M \mid \sigma; H$
Variable Substitutions	π	$::= \cdot \mid \mathsf{id}_\psi \mid \sigma; x$
LF Contexts	Ψ, Φ	$::= \cdot \mid \psi \mid \Psi, x{:}A$

Normal terms are either lambda-abstractions or neutral terms which are defined using a spine representation to give us direct access to the head of a neutral term. Normal objects may contain *ordinary bound variables* x which are used to represent object-level binders and are bound by λ-abstraction or in a context Ψ. Contextual LF extends LF by allowing two kinds of *contextual variables*: the meta-variable u has type $(\Psi.P)$ and stands for a general LF object that has type P and may use the variables declared in Ψ; the parameter variable p has type $\#(\Psi.A)$ and stands for an LF variable object of type A in the context Ψ.

Contextual variables are associated with a postponed substitution σ which is applied as soon as we instantiate it. More precisely, a meta-variable u stands for a contextual object $\hat{\Psi}.R$ where $\hat{\Psi}$ describes the ordinary bound variables which may occur in R. This allows us to rename the free variables occurring in R when necessary. The parameter variable p stands for a contextual object $\hat{\Psi}.H$ where H must be either an ordinary bound variable from $\hat{\Psi}$ or another parameter variable.

In the simultaneous substitutions σ, we do not make the domain explicit. Rather we think of a substitution together with its domain Ψ and the i-th element in σ corresponds to the i-th declaration in Ψ. We have two different ways of building a substitution entry: either by using a normal term M or a variable x. Note that a variable x is only a normal term M if it is of base type. However, as we push a substitution σ through a λ-abstraction $\lambda x.M$, we need to extend σ with x. The resulting substitution σ, x may not be well-formed, since x may not be of base type and in fact we do not know its type. Hence, we allow substitutions not only to be extended with normal terms M but also with variables x; in the latter case we write $\sigma; x$. Expression id_ψ denotes the identity substitution with domain ψ while $\cdot$ describes the empty substitution.

Application of a substitution σ to an LF normal form B, written as $[\sigma]B$, is *hereditary* [20] and produces in turn a normal form by removing generated redexes on the fly, possibly triggering further hereditary substitutions.

$\boxed{\Delta; \Psi \vdash H \Rightarrow A}$ Synthesize type A for head H

$$\frac{\Psi(x) = A}{\Delta; \Psi \vdash x \Rightarrow A} \qquad \frac{\Sigma(c) = A}{\Delta; \Psi \vdash c \Rightarrow A} \qquad \frac{\Delta(p) = \#\Phi.A \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash p[\sigma] \Rightarrow [\sigma]A}$$

$\boxed{\Delta; \Psi \vdash S : A > P}$ Check spine S against A with target P

$$\frac{}{\Delta; \Psi \vdash \mathsf{nil} : P > P} \qquad \frac{\Delta; \Psi \vdash M \Leftarrow A \quad \Delta; \Psi \vdash S : [M/x]B > P}{\Delta; \Psi \vdash M\ S : \Pi x{:}A.B > P}$$

$\boxed{\Delta; \Psi \vdash M \Leftarrow A}$ Check normal object M against type A

$$\frac{\Delta; \Psi, x{:}A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.\, M \Leftarrow A \to B} \qquad \frac{\Delta(u) = \Phi.P \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad Q = [\sigma]P}{\Delta; \Psi \vdash u[\sigma] \Leftarrow Q}$$

$$\frac{\Delta; \Psi \vdash H \Rightarrow A \quad \Delta; \Psi \vdash S : A > P}{\Delta; \Psi \vdash H \cdot S \Leftarrow P}$$

$\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Phi}$ Check substitution σ against domain Φ

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{}{\Delta; (\psi, \Psi^0) \vdash \mathsf{id}_\psi \Leftarrow \psi}$$

$$\frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]A}{\Delta; \Psi \vdash (\sigma, M) \Leftarrow (\Phi, x{:}A)} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash H \Rightarrow B \quad B = [\sigma]A}{\Delta; \Psi \vdash (\sigma; H) \Leftarrow (\Phi, x{:}A)}$$

Figure 3 Bi-directional typing for contextual LF.

An LF context Ψ is either a list of bound variable declarations $\overrightarrow{x : A}$ or a context variable ψ followed by such a list. We write Ψ^0 for contexts that do not start with a context variable. We write Ψ, Φ^0 or sometimes Ψ, Φ for the extension of context Ψ by the variable declarations of Φ^0 or Φ, resp. The operation $\mathsf{id}(\Psi)$ that generates an identity substitution for a given context Ψ is defined inductively as follows: $\mathsf{id}(\cdot) = \cdot$, $\mathsf{id}(\psi) = \mathsf{id}_\psi$, and $\mathsf{id}(\Psi, x{:}A) = \mathsf{id}(\Psi); x$.

We summarize the bi-directional type system for contextual LF in Fig. 3. LF objects may depend on variables declared in the context Ψ and a fixed meta-context Δ which contains contextual variables such as meta-variables u, parameter variables p, and context variables ψ. All typing judgments have access to both contexts and a fixed well-typed signature Σ where we store constants c together with their types and kinds.

5.2 Meta-level Terms and Typing Rules

We lift contextual LF objects to meta-objects to have a uniform definition of all meta-objects. We also define context schemas G that classify contexts.

$$\begin{array}{llll} \text{Context Schemas } G & ::= \exists \Phi^0.B \mid G + \exists \Phi^0.B & & \\ \text{Meta Types} & U, V ::= \Psi.P \mid G \mid \#\Psi.A & \text{Meta Objects} & C, D \ ::= \hat{\Psi}.R \mid \Psi \end{array}$$

A consequence of the uniform treatment of meta-terms is that the design of the computation language is modular and parametrized over meta-terms and meta-types. This has two main advantages: First, we can in principle easily extend meta-terms and meta-types without affecting the computation language; second, it will be key to a modular, clean design.

The above definition gives rise to a compact treatment of meta-context Δ. A meta-variable X can denote a meta-variable u, a parameter variable p, or a context variable ψ.

Meta substitution C/X can represent $\hat{\Psi}.R/u$, or Ψ/ψ, or $\hat{\Psi}.x/p$, or $\hat{\Psi}.p'[\pi]/p$ (where π is a variable substitution so that $p[\pi]$ always produces a variable). A meta declaration $X{:}U$ can stand for $u : \Psi.P$, or $p : \#\Psi.A$, or $\psi : G$. Intuitively, as soon as we replace u with $\hat{\Psi}.R$ in $u[\sigma]$, we apply the substitution σ to R hereditarily. The simultaneous meta-substitution, written as $[\![\theta]\!]$, is a straightforward extension of the single substitution. For a full definition of meta-substitutions, see [9, 4]. We summarize the typing rules for meta-objects below.

$\boxed{\Delta \vdash C : U}$ Check meta-object C against meta-type U $\qquad \dfrac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \hat{\Psi}.R : \Psi.P}$

$$\frac{}{\Delta \vdash \cdot : G} \qquad \frac{\Delta(\psi) = G}{\Delta \vdash \psi : G} \qquad \frac{\Delta \vdash \Psi : G \quad \exists \Phi^0.B \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi^0 \quad [\sigma]B = B'}{\Delta \vdash \Psi, x{:}B' : G}$$

We write $\hat{\Psi}$ for a list of variables obtained by erasing the types from the context Ψ. We have omitted the rules for parameter types $\#\Psi.A$ because they are not important for the further development. Intuitively an object R has type $\#\Psi.A$ if R is either a concrete variable x of type A in the context Ψ or a parameter variable p of type A in the context Ψ. This can be generalized to account for re-ordering of variables allowing the parameter variable p to have some type A' in the context Ψ' s.t. there exists a permutation substitution π on the variables such that $\Psi \vdash \pi : \Psi'$ and $A = [\pi]A'$.

5.3 Well-founded Structural Subterm Order

There are two key ingredients to guarantee that a given function is total: we need to ensure that all the recursive calls are on smaller arguments according to a well-founded order and the function covers all possible cases. We define here a well-founded structural subterm order on contexts and contextual objects similar to the subterm relations for LF objects[10]. For simplicity, we only consider here non-mutual recursive type families; those can be incorporated using the notion of subordination [19].

We first define an ordering on contexts: $\boxed{\Psi \preceq \Phi}$, read as "context Ψ is a subcontext of Φ", shall hold if all declarations of Ψ are also present in the context Φ, i.e., $\Psi \subseteq \Phi$. The strict relation $\boxed{\Psi \prec \Phi}$, read as "context Ψ is strictly smaller than context Φ" holds if $\Psi \preceq \Phi$ but Ψ is strictly shorter than Φ.

Further, we define three relations on contextual objects $\hat{\Psi}.M$: a strict subterm relation $\prec$, an equivalence relation $\equiv$, and an auxiliary relation $\preceq$.

$$\frac{\hat{\Psi} \subseteq \hat{\Phi} \text{ or } \hat{\Phi} \subseteq \hat{\Psi} \quad \pi \text{ is a variable subst. s.t. } M = [\pi]N}{\hat{\Psi}.M \equiv \hat{\Phi}.N}$$

$$\frac{\hat{\Psi}.M \preceq \hat{\Phi}.N_i \ \text{ for some } 1 \le i \le n}{\hat{\Psi}.M \prec \hat{\Phi}.\, h \cdot N_1 \ \ldots \ N_n \ \mathsf{nil}} \qquad \frac{\hat{\Psi}.M \prec \hat{\Phi}.N}{\hat{\Psi}.M \preceq \hat{\Phi}.N} \qquad \frac{\hat{\Psi}.M \equiv \hat{\Phi}.N}{\hat{\Psi}.M \preceq \hat{\Phi}.N} \qquad \frac{\hat{\Psi}.M \preceq \hat{\Phi}, x.N}{\hat{\Psi}.M \preceq \hat{\Phi}.\lambda x.N}$$

$\hat{\Psi}.M$ is a strict subterm of $\hat{\Phi}.N$ if M is a proper subterm of N modulo α-renaming and weakening. Two terms $\hat{\Psi}.M$ and $\hat{\Phi}.N$ are structurally equivalent, if they describe the same term modulo α-renaming and possible weakening. To allow mutual recursive definitions and richer subterm relationships, we can incorporate subordination information and generalize the variable substitution π (see for example [10] for such a generalization). Using the defined subterm order, we can easily verify that the recursive calls in the examples are structurally smaller.

The given subterm relation is well-founded. We define the measure $||\Psi||$ of a ground context Ψ^0 or its erasure $\hat{\Psi}^0$ as its length $|\Psi|$. The measure $||\hat{\Psi}.M||$ of a contextual object

$\hat{\Psi}.M$, is the measure $||M||$ of M. The latter is defined inductively by:

$$\begin{array}{lcl} ||h \cdot M_1 \ldots M_n \ \mathsf{nil}|| & = & 1 + \max(||M_1||, \ldots, ||M_n||) \\ ||\lambda x.M|| & = & ||M|| \end{array}$$

▶ **Theorem 3** (Order on contextual objects is well-founded). *Let θ be a grounding meta-substitution.*

1. *If $C \prec C'$ then $|||[\![\theta]\!]C|| \ < \ |||[\![\theta]\!]C'||$.*
2. *If $C \equiv C'$ then $|||[\![\theta]\!]C|| \ = \ |||[\![\theta]\!]C'||$.*
3. *If $C \preceq C'$ then $|||[\![\theta]\!]C|| \ \leq \ |||[\![\theta]\!]C'||$.*

5.4 Case Splitting

Our language allows pattern matching and recursion over contextual objects. For well-formed recursors $(\mathsf{rec{-}case}^{\mathcal{I}}\ C\ \ \mathsf{of}\ \vec{b})$ with invariant $\mathcal{I} = \Pi\Delta.\Pi X{:}U.\tau$, branches $\vec{b}$ need to cover all different cases for the argument C of type U. We only take the shape of U into account and generate the unique complete set $\mathcal{U}_{\Delta \vdash U}$ of non-overlapping shallow patterns by splitting meta-variable X of type U.

If $U = \Psi.P$ is a base type, then intuitively the set $\mathcal{U}_{\Delta \vdash U}$ contains all neutral terms $R = H \cdot S$ where H is a constructor c, a concrete variable x from Ψ or a parameter variable $p[\mathsf{id}_\psi]$ denoting a variable from the context variable ψ, and S is a most general spine s.t. the type of R is an instance of P in the context Ψ. We note that when considering only closed terms it suffices to consider only terms with $H = c$. However, when considering terms with respect to a context Ψ, we must generate additional cases covering the scenario where H is a variable – either a concrete variable x if $x{:}A$ is in Ψ or a parameter variable if the context is described abstractly using a context variable ψ.

If U denotes a context schema G, we generate all shallow context patterns of type G. This includes the empty context and a context extended with a declaration formed by $\Psi, x{:}A$.

From $\mathcal{U}_{\Delta \vdash U}$ we generate the complete minimal set $\mathcal{C} = \{\Delta_i; r_{ik}, \ldots, r_{i1}.r_{i0} \mid 1 \leq i \leq n\}$ of possible, non-overlapping cases where the i-th branch shall have the well-founded recursive calls $r_{ik}, \ \ldots, \ r_{i1}$ for the case r_{i0}. For the given branches $\vec{b}$ to be covering, each element in $\mathcal{C}$ must correspond to one branch b_i.

Splitting on a Contextual Type

Following [5, 17], the patterns R of type $\Psi.P$ are computed by brute force: We first synthesize a set $\mathcal{H}_{\Delta;\Psi}$ of *all* possible heads together with their type: constants $c \in \Sigma$, variables $x \in \Psi$, and parameter variables if Ψ starts with a context variable ψ.

$$\begin{array}{rl} \mathcal{H}_{\Delta;\Psi} = & \{(\Delta; \Psi \vdash c : A) \mid (c{:}A) \in \Sigma\} \\ & \cup \{(\Delta; \Psi \vdash x : A) \mid (x{:}A) \in \Psi\} \\ & \cup \{(\Delta, \overrightarrow{X{:}U}, p{:}\#(\psi.B'); \ \Psi \vdash p[\mathsf{id}_\psi] : B') \mid \Psi = \psi, \Psi^0 \text{ and } \psi{:}G \in \Delta \text{ and } \exists \overrightarrow{x{:}A}.B \in G \\ & \qquad \text{and } \mathsf{genMV}\ (\psi.A_i) = (X_i{:}U_i,\ M_i) \text{ for all } i, \text{ and } B' = [\overrightarrow{M/x}]B \ \} \end{array}$$

See Fig. 4. Using a head H of type A from the set $\mathcal{H}_{\Delta;\Psi}$, we then generate, if possible, the most general pattern $H \cdot S$ whose target type is unifiable with P in the context Ψ. We describe unification using the judgment $\boxed{\Delta; \Psi \vdash Q \doteq P \ / \ (\Delta' \ , \ \theta)}$. If unification succeeds then $[\![\theta]\!]Q = [\![\theta]\!]P$ and $\Delta' \vdash \theta : \Delta$.

$\boxed{\mathsf{genMV}\ (\Psi.A) = (X{:}U, M)}$ Generation of a lowered meta variable

$\mathsf{genMV}\ (\Psi.\,\Pi\overrightarrow{x{:}A}.P) = (u : (\Psi, \overrightarrow{x{:}A}.\,P),\ \lambda\vec{x}.u[\mathsf{id}(\Psi, \overrightarrow{x{:}A})])$ for a fresh meta variable u

$\boxed{\Delta; \Psi \vdash R : A \Leftarrow P/(\Delta', \theta, R_0)}$ Extending $R{:}A$ to most general normal term $R_0 : [\![\theta]\!]P$.

$$\frac{\Delta; \Psi \vdash Q \doteq P \ /\ (\Delta_0,\ \theta)}{\Delta; \Psi \vdash R : Q \Leftarrow P \ /\ (\Delta_0\ ,\ \theta\ ,\ [\![\theta]\!]R)}$$

$$\frac{\mathsf{genMV}\ (\Psi.A) = (X{:}U,\ M) \quad \Delta, X{:}U; \Psi \vdash R\ M : [M/x]B \Leftarrow P \ /\ (\Delta_0,\ \theta,\ R')}{\Delta; \Psi \vdash R : \Pi x{:}A.B \Leftarrow P \ /\ (\Delta_0,\ \theta,\ R')}$$

Figure 4 Generation of most general normal objects and call patterns.

$\boxed{\Delta; \Psi \vdash R : A \Leftarrow P \ /\ (\Delta', \theta, R_0)}$ describes the generation of a normal pattern where all the elements on the left side of / are inputs and the right side is the output, which satisfies $\Delta' \vdash \theta : \Delta$ and $\Delta'; [\![\theta]\!]\Psi \vdash R \Rightarrow [\![\theta]\!]A$ and $\Delta'; [\![\theta]\!]\Psi \vdash R_0 \Leftarrow [\![\theta]\!]P$. To generate a normal term R_0 of the expected base type, we start with head $H : A$. As we recursively analyze A, we generate all the arguments H is applied to until we reach an atomic type Q. If Q unifies with the expected type P, then generating a most general neutral term with head H succeeds.

$$\mathcal{U}_{\Delta \vdash \Psi.P} = \{\ (\Delta'' \vdash \hat{\Phi}.R : \Phi.Q)\ \mid\ (\Delta'; \Psi \vdash H : A) \in \mathcal{H}_{\Delta;\Psi} \text{ and } \Delta'; \Psi \vdash H : A \Leftarrow P \ /\ (\Delta'',\ \theta,\ R) \text{ and } \Phi = [\![\theta]\!]\Psi \text{ and } Q = [\![\theta]\!]P\}$$

Splitting on a Context Schema

Spitting a context variable of schema G generates the empty context and the non-empty contexts $(\phi, x{:}B')$ for each possible form of context entry $\exists\Phi^0.B \in G$.

$$\mathcal{U}_{\Delta \vdash G} = \{\ (\Delta \vdash \cdot : G)\ \} \cup \{\ (\Delta, \phi{:}G, \overrightarrow{X{:}U} \vdash (\phi, x{:}[\overrightarrow{M/x}]B) : G) \mid \phi \text{ a fresh context variable and for any } \exists\overrightarrow{x : A}.B \in G \text{ . } \mathsf{genMV}\ (\psi.A_i) = (X_i{:}U_i, M_i) \text{ for all } i\}$$

▶ **Theorem 4** (Splitting on meta-types). *The set $\mathcal{U}_{\Delta \vdash U}$ of meta-objects generated is non-redundant and complete.*

Proof. $\mathcal{U}_{\Delta \vdash G}$ is obviously non-redundant. $\mathcal{U}_{\Delta \vdash \Psi.P}$ is non-redundant since all generated neutral terms have distinct heads. Completeness is proven by cases. ◀

6 Generation of Call Patterns and Coverage

Next, we explain the generation of call patterns, i.e. well-founded recursive calls as well as the actual call pattern being considered.

▶ **Definition 5** (Generation of call patterns). Given the invariant $\mathcal{I} = \Pi(\Delta_1, X_0{:}U_0).\tau_0$ where $\Delta_1 = X_n{:}U_n, \ldots, X_1{:}U_1$, the set $\mathcal{C}$ of call patterns $(\Delta_i\ ;\ r_{ik}{:}\tau_{ik}, \ldots, r_{i1}{:}\tau_{i1}\ .\ r_{i0})$ is generated as follows:

- For each meta-object $\Delta_i \vdash C_{i0} : V_i$ in $\mathcal{U}_{\Delta_0 \vdash U_0}$, we generate using unification, if possible, a call pattern $r_{i0} = f\ C_{in} \ldots C_{i1}\ C_{i0}$ s.t. $\tau_{i0} = [\![C_{in}/X_n, \ldots, C_{i1}/X_1, C_{i0}/X_0]\!]\tau_0$ and

$\Delta_i \vdash r_{i0} : \tau_{i0}$. This may fail if V_i is not an instance of the scrutinee type U_0; then, the case C_{i0} is impossible.

- Further, for all $1 \leq j \leq k$, $\Delta_i = Y_k{:}V_k, \ldots, Y_1{:}V_1$, we generate a recursive call $r_{ij} = f\ C_{jn} \ldots C_{j1}\ Y_j$ s.t. $\tau_{ij} = [\![C_{jn}/X_n, \ldots, C_{j1}/X_1, Y_j/X_0]\!]\tau_0$ and $\Delta_i \vdash r_{ij} : \tau_{ij}$, if $Y_j \prec C_{i0}$. This may also fail, if V_i is not an instance with U_0; in this case V_i does not give rise to recursive call.

▶ **Theorem 6** (Pattern generation). *The set $\mathcal{C}$ of call patterns generated is non-redundant and complete and the recursive calls are well-founded.*

Proof. Using Theorem 4 and the properties of unification. ◀

▶ **Definition 7** (Coverage). We say $\boxed{\vec{b} \text{ covers } \mathcal{I}}$ iff for every Δ_i ; $\overrightarrow{r_i{:}\tau_i} \,.\, r_{i0} \in \mathcal{C}$ where $\mathcal{C}$ is the set of call patterns given $\mathcal{I}$, we have one corresponding Δ_i ; $\overrightarrow{r_i{:}\tau_i} \,.\, r_{i0} \Rightarrow e_i \in \vec{b}$ and vice versa.

7 Termination

We now prove that every well-typed closed program e terminates (halts) by a standard reducibility argument; closely related is [21]. The set $\mathcal{R}_\tau$ of reducible closed programs $\cdot;\cdot \vdash e : \tau$ is defined by induction on the size of τ.

$$
\begin{array}{lllll}
\text{Contextual Type} & \mathcal{R}_{[U]} & = & \{e \mid \cdot;\cdot \vdash e : [U] \text{ and } e \text{ halts}\} \\
\text{Function Type} & \mathcal{R}_{\tau' \to \tau} & = & \{e \mid \cdot;\cdot \vdash e : \tau' \to \tau \text{ and } e \text{ halts and } \forall e' \in \mathcal{R}_{\tau'}.\, e\ e' \in \mathcal{R}_\tau\} \\
\text{Dependent Type} & \mathcal{R}_{\Pi X{:}U.\tau} & = & \{e \mid \cdot;\cdot \vdash e : \Pi X{:}U.\tau \text{ and} \\
 & & & \quad e \text{ halts and } \forall C s.t. \cdot \vdash C : U.\, e\ C \in \mathcal{R}_{[\![C/X]\!]\tau}\} \\
\text{Context} & \mathcal{R}_\Gamma & = & \{\eta \mid \cdot;\cdot \vdash \eta : \Gamma \text{ and } \eta(x) \in \mathcal{R}_\tau \text{ for all } (x{:}\tau) \in \Gamma\}
\end{array}
$$

For the size of τ all meta types U shall be disregarded, thus, the size is invariant under meta substitution C/X. We also note that since reduction $e \longrightarrow e'$ is deterministic, e halts if and only if e' halts.

▶ **Lemma 8** (Expansion closure).

1. *If $\cdot;\cdot \vdash e : \tau$ and $e \longrightarrow e'$ and $e' \in \mathcal{R}_\tau$, then $e \in \mathcal{R}_\tau$.*
2. *If $\cdot;\cdot \vdash e : \tau$ and $e \longrightarrow^* e'$ and $e' \in \mathcal{R}_\tau$, then $e \in \mathcal{R}_\tau$.*

Proof. The first statement, by induction on the size of type τ. The second statement, inductively on $\longrightarrow^*$. ◀

▶ **Lemma 9** (Fundamental Lemma).
If $\Delta;\Gamma \vdash e : \tau$ and grounding substitution θ s.t. $\cdot \vdash \theta : \Delta$ and $\eta \in \mathcal{R}_{[\![\theta]\!]\Gamma}$ then $[\eta][\![\theta]\!]e \in \mathcal{R}_{[\![\theta]\!]\tau}$.

Proof. By induction on $\Delta;\Gamma \vdash e : \tau$. In the interesting case of recursion rec−case, we make essential use of coverage and structural descent in the recursive calls. ◀

▶ **Theorem 10** (Termination). *If $\cdot;\cdot \vdash e : \tau$ then e halts.*

Proof. Taking the empty meta-context Δ and empty computation-level context Γ, we obtain $e \in \mathcal{R}_\tau$ by the fundamental lemma, which implies that e halts by definition of τ. ◀

8 Related Work

Our work is most closely related to [16] where the authors propose a modal lambda-calculus with iteration to reason about closed HOAS objects. In their work the modal type $\Box$ describes closed LF objects. Our work extends this line to allow open LF objects and define functions by pattern matching and well-founded recursion.

Similar to our approach, Schürmann [15] presents a meta-logic $\mathcal{M}^2$ for reasoning about LF specifications and describes the generation of splits and well-formed recursive calls. However, $\mathcal{M}^2$ does not support higher-order computations. Moreover, the foundation lacks first-class contexts, but all assumptions live in an ambient context. This makes it less direct to justify reasoning with assumptions, but maybe more importantly complicates establishing meta-theoretic results such as proving normalization.

Establishing well-founded induction principles to support reasoning about higher-order abstract syntax specifications has been challenging and a number of alternative approaches have been proposed. These approaches have led to new reasoning logics – either based on nominal set theory [14] or on nominal proof theory [7]. Proving consistency of these theories in the presence of induction is often significantly more complicated [18]. Our work shows that reasoning about HOAS representations can be supported using first-order logic by modelling HOAS objects as contextual objects. As a consequence, we can directly take advantage of established proof and mechanization techniques. This also opens up the possibility of supporting contextual reasoning as a domain in other systems.

9 Conclusion

We have developed a core language with structural recursion for implementing total functions about LF specification. We describe a sound coverage algorithm which, in addition to verifying that there exists a branch for all possible contexts and contextual objects, also generates and verifies valid primitive recursive calls. To establish consistency of our core language we prove termination using reducibility semantics.

Our framework can be extended to handle mutual recursive functions: By annotating a given rec−case-expression with a list of invariants using the subordination relation, we can generate well-founded recursive calls matching each of the invariants. Based on these ideas, we have implemented a totality checker in Beluga. We also added reasoning principles for inductive types [4] that follow well-trodden paths; we must ensure that our inductive type satisfies the positivity restriction and define generation of patterns for them.

Our language not only serves as a core programming language but can be interpreted by the Curry-Howard isomorphism as a proof language for interactively developing proofs about LF specifications. In the future, we plan to implement and design such a proof engine and to generalize our work to allow lexicographic orderings and general well-founded recursion.

Acknowledgements. We thank Sherry Shanshan Ruan for her work during her Summer Undergraduate Research Internship in 2013 at the beginning of this project.

References

1 Andreas Abel. *Tutch User's Guide.* Carnegie-Mellon University, Pittsburgh, PA, 2002. Section 7.1: Proof terms for structural recursion.

2 Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In TLCA'11, volume 6690 of *LNCS*, pages 10–26. Springer, 2011.

3 Olivier Savary Belanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In *CPP'13*, volume 8307 of *LNCS*, pages 243–258. Springer, 2013.

4 Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *POPL'12*, pages 413–424. ACM, 2012.

5 Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. *ENTCS*, 228:69–84, 2009.

6 Francisco Ferreira and Brigitte Pientka. Bidirectional elaboration of dependently typed languages. In *PPDP'14*. ACM, 2014.

7 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *LICS'08*, pages 33–44. IEEE CS Press, 2008.

8 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.

9 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM TOCL*, 9(3):1–49, 2008.

10 Brigitte Pientka. Verifying termination and reduction properties about higher-order logic programs. *JAR*, 34(2):179–207, 2005.

11 Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL'08*, pages 371–382. ACM, 2008.

12 Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *JFP*, 1(1–37), 2013.

13 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR'10*, volume 6173 of *LNCS*, pages 15–21. Springer, 2010.

14 Andrew Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

15 Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.

16 Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *TCS*, 266(1-2):1–57, 2001.

17 Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In *TPHOLS'03*, volume 2758 of *LNCS*, pages 120–135, Rome, Italy, 2003. Springer.

18 Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *J. Applied Logic*, 10(4):330–367, 2012.

19 Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. CMU-CS-99-167.

20 Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.

21 Hongwei Xi. Dependent types for program termination verification. *HOSC*, 15(1):91–131, 2002.

Polynomial Time in the Parametric Lambda Calculus

Brian F. Redmond

Department of Computing, Mathematics and Statistical Sciences
Grande Prairie Regional College
10726 – 106 Avenue, Grande Prairie, AB, T8V 4C4, Canada
bredmond@GPRC.ab.ca

Abstract

In this paper we investigate Implicit Computational Complexity via the parametric lambda calculus of Ronchi Della Rocca and Paolini [13]. We show that a particular instantiation of the set of input values leads to a characterization of polynomial time computations in a similar way to Lafont's Soft Linear Logic [9]. This characterization is manifestly type-free and does not require any ad hoc extensions to the pure lambda calculus. Moreover, there is a natural extension to nondeterminism with the addition of explicit products.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Parametric Lambda Calculus, Polynomial Time Complexity, Combinators, Nondeterminism and Explicit Products, Implicit Computational Complexity

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.288

1 Introduction

There is an inherent problem in studying computational complexity within the lambda calculus in that a single beta reduction step can effectively square the size of a term, thus leading to an exponential size term after only a linear number of steps. It therefore seems unreasonable to simply count the number of beta reduction steps as a measure of the computational complexity of reduction. Indeed, there have been studies, most recently in [1], which give more reasonable measures of the complexity of a given λ-term using explicit substitutions and notions of sharing. These delicate issues are entirely avoided here as all beta reduction steps, except for a constant number (which does not depend on the size of the input), do not increase the size of the lambda term. Therefore, we feel justified in taking a very simplistic approach to measuring the complexity of reduction of a given lambda term. We define a simple "by-value" operational semantics and define the complexity of reduction as the size of its corresponding evaluation tree, which we show is polynomial in the size of the input binary word. We claim that any such reduction can be simulated on a Turing machine with polynomial overhead.

Most studies of complexity within the (pure) lambda calculus rely on typing restrictions to ensure that terms are strongly normalizing. For example, it is well known that terms in the simply typed lambda calculus are strongly normalizing and, moreover, that the class of representable numerical functions is precisely the class of *extended polynomials* [15]. Adding an (impredicative) operation of abstraction on types, as in Girard's system **F** [6], greatly increases the class of representable functions to the class of functions provably total in second order Peano arithmetic [7]. Nevertheless, the system remains strongly normalizing. A system somewhere in the middle of these two extremes is obtained by stratifying type abstraction into a finite number of levels. Indeed, in this case, the class of representable functions

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 288–301

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is precisely the class of super-elementary functions, i.e. the class of $\mathcal{E}_4$ in Grzegorczyk's subrecursive hierarchy [10]. Finally, there are also well known studies on the complexity of beta equivalence in the simply typed lambda calculus [16], as well as related results when restricting to low functional orders [14].

Another fruitful approach to investigating complexity in the lambda calculus is through linear logic and related type systems. In these studies, it is not typically type abstraction that is limited, but instead duplication is controlled via the modified exponentials ! and § (see [5] and [2], for example). These type systems were derived from their corresponding systems of (light) linear logic and proof nets, and illustrate the computational power of duplication in the lambda calculus. Unfortunately, type checking and type inference are undecidable in the presence of unrestricted polymorphism [4].

In this paper we take an entirely different approach to studying complexity within the lambda calculus via the parametric lambda calculus [13]. With a frugal choice of so-called input values, we show that strong normalization is guaranteed, yet the system remains expressive enough to capture polynomial time computations. Moreover, there is a natural extension to nondeterminism with the addition of explicit products. In contrast to the above studies, however, this approach does not rely on typing restrictions as we work in an entirely type-free setting. Nevertheless, we believe a system of intersection types can be introduced post hoc if desired.

This work is closely related to the author's work on Bounded Combinatory Logic [12]. In that work, the usual Curry combinators B, C, K, W are introduced, but the duplication combinator W has one of its arguments restricted to a proper subset of combinators, namely the BCK-combinators. This ensures that only *affine linear* terms are duplicated, and leads to a simple characterization of polynomial time computations. The "moral" analogue in the lambda calculus corresponds to a particular instantiation of the parametric lambda calculus, which is the focus of the current paper. However, the systems are not equivalent and in fact use completely different reduction strategies and encodings. For this reason, we chose to present this work independently and investigate the relationship between the two system in future work.

2 An Instance of the Parametric Lambda Calculus

One of the aims of the parametric lambda calculus is to study in a uniform way various systems of the pure lambda calculus, in particular its call-by-name and call-by-value versions [13]. This is done by restricting beta reduction to subsets of lambda terms, called input values, that satisfy certain closure conditions. These closure conditions guarantee important properties like confluence are satisfied. In this paper we study various instantiations of the parametric lambda calculus in the context of Implicit Computational Complexity (ICC). We refer the reader to [13] for much of the notation used as well as some of the basic definitions in the lambda calculus. However, all nonstandard notation, terminology, and definitions will be explicitly stated.

The set of lambda terms, Λ, is defined in the usual manner. We assume a countably infinite set of variables, Var, and define the set of lambda terms Λ as follows:

$$M, N ::= x \mid (MN) \mid (\lambda x.M)$$

where $x \in \text{Var}$. For notational convenience we tacitly assume the standard conventions regarding parentheses and for contracting multiple lambda abstractions. We use the symbol $\equiv$ to denote the syntactical identity of terms up to α-congruence, and $M[N/x]$ denotes the

capture-free substitution of N for x in M. Finally, we need the notion of a context $C[.]$ as defined in [13], Def. 1.1.11.

Let M be a generic lambda term. For each subterm of M of the form $\lambda x.P$, let $\#(\lambda x.P)$ denote the number of times x occurs free in P. The **size**[1] of a lambda term M, denoted $s(M)$, is defined by induction on M as follows: $s(x) = 1$ if x is a variable, $s(\lambda x.P) = 1 + s(P)$ and $s(PQ) = s(P) + s(Q)$. Recall from [13] that a subset Δ of Λ is called a set of **input values** if it satisfies the following three conditions:

1. $\text{Var} \subseteq \Delta$
2. If P and Q are in Δ, then so is $P[Q/x]$, for each $x \in \text{Var}$
3. If $M \in \Delta$ and $M \to_\Delta N$, then $N \in \Delta$

In condition 3, the symbol $\to_\Delta$ denotes a one-step Δ-reduction [13].

In this paper we introduce a set of input values, denoted Φ, consisting of "pseudo-affine" terms. This is made precise in the following definition: Let Φ denote the smallest subset of Λ satisfying the following closure properties:

- $x \in \text{Var}$ implies $x \in \Phi$,
- If $M, N \in \Phi$, then $(MN) \in \Phi$,
- If $M \in \Phi$ and $x \in \text{Var}$ and x occurs free at most once in M, then $(\lambda x.M) \in \Phi$.

We shall often refer to Φ as the set of **player** terms and to $\Lambda \backslash \Phi$ as the set of **opponent** terms. For example, the closed terms $I \equiv \lambda x.x, B \equiv \lambda xyz.x(yz), C \equiv \lambda xyz.xzy$, and $K \equiv \lambda xy.x$ are all player terms, but $D \equiv \lambda x.xx$ is an opponent term. Note that *free* variables may appear more than once in a player term. This follows terminology introduced in [12].

The one-step Φ-reduction $\to_\Phi$ is defined as the contextual closure of the following rule:

$$(\lambda x.M)N \to_\Phi M[N/x] \quad \text{if and only if} \quad N \in \Phi$$

Let $\to_\Phi^*$ denote the reflexive and transitive closure of $\to_\Phi$. A redex $R \equiv (\lambda x.P)Q$ with $(\lambda x.P)$ an opponent term and $Q \in \Phi$ is called an **opponent redex**. A redex $R \equiv (\lambda x.P)Q \in \Phi$ is called a **player redex**. This terminology extends to their respective reductions as well.

Define the **degree** of a term M, denoted $d(M)$, as follows: $d(M) = 0$ if $M \in \Phi$ and $d(MN) = d(M) + d(N)$ if $MN \in \Lambda \backslash \Phi$ and $d(\lambda x.P) = 1 + d(P)$ if $\lambda x.P \in \Lambda \backslash \Phi$. For example, the degree of the term $\lambda x.(\lambda y.yy)x$ is 2.

▶ **Theorem 1** (confluence). *The subset Φ as defined above is a set of input values. Therefore, the reduction relation $\to_\Phi^*$ is confluent by Theorem 1.2.5 in [13].*[2]

Proof. Condition 1 is obvious. For condition 2, we use induction on the structure of P:

- If $P \equiv y$, then $P[Q/x]$ is either $Q \in \Phi$ if $y \equiv x$, or $y \in \Phi$ if $y \not\equiv x$.
- If $P \equiv P_1 P_2 \in \Phi$, then $P_1 \in \Phi$ and $P_2 \in \Phi$. By the induction hypothesis, we have $P_1[Q/x], P_2[Q/x] \in \Phi$. Thus, $P[Q/x] \equiv P_1[Q/x]P_2[Q/x] \in \Phi$.
- If $P \equiv \lambda z.M \in \Phi$, then $M \in \Phi$ and z occurs free at most once in M. If $z \equiv x$, then $P[Q/x] \equiv P \in \Phi$. Otherwise, by α-congruence, we may assume that $z \notin \text{FV}(Q)$, so z occurs free at most once in $M[Q/x]$. By the induction hypothesis we have $M[Q/x] \in \Phi$. Thus, $P[Q/x] \equiv \lambda z.M[Q/x] \in \Phi$.

Finally, for condition 3, we proceed by induction on the structure of M:

- If M is a variable, then the result is vacuously true.

---

[1] This is called *length* in [8].

[2] The set $\text{Var} \cup \Phi^0$, where Φ^0 denotes the set of closed terms in Φ, also forms a set of input values. However, as we shall see in Section 3.1, pseudo-affine terms provide more flexibility when defining programs.

- If $M \equiv P_1P_2 \in \Phi$ and the Φ-redex/reduction is entirely in P_i ($i \in \{1,2\}$), then $P_i \to_\Phi P_i'$. By the induction hypothesis $P_i' \in \Phi$, so either $N \equiv P_1'P_2 \in \Phi$ or $N \equiv P_1P_2' \in \Phi$. On the other hand, if $M \equiv (\lambda x.P)Q$ and $(\lambda x.P)Q \to_\Phi P[Q/x] \equiv N$, then $N \in \Phi$ by condition 2.
- If $M \equiv \lambda x.P \in \Phi$, then x occurs free in $P \in \Phi$ at most once, and $P \to_\Phi P'$. By the induction hypothesis, $P' \in \Phi$. Moreover, since the redex contracted in $P \in \Phi$ must be a player redex, the number of times x occurs free in P' is no more than 1. Thus, $\lambda x.P' \in \Phi$.

Therefore, Φ is a valid set of input values. ◀

Note that any term in the $\lambda\Phi$-calculus has the form: $\lambda x_1 \cdots x_n.\zeta M_1 \cdots M_m$ $(n, m \geq 0)$ where ζ is either a variable or a Φ-redex or a head block. (A head block is a term $(\lambda x.P)Q$, where $Q \notin \Phi$.) We say that a term is in Φ-normal form if it has the form $\lambda x_1 \cdots x_n.\zeta M_1 \cdots M_m$, where M_i is in Φ-normal form $(1 \leq i \leq m)$ and ζ is either a variable or a head block $(\lambda x.P)Q$, where both P and Q are in Φ-normal form. Note that a lambda term is in Φ-normal form iff it contains no Φ-redexes.

▶ **Lemma 2.** *If P is any term and Q is a player term, then $d(P) = d(P[Q/x])$.*

Proof. First note that if P is a player term, then so is $P[Q/x]$ and both have degree 0. So assume P is an opponent term and argue by induction on P. If $P \equiv P_1P_2$, then by the induction hypothesis, $d(P_1[Q/x]) = d(P_1)$ and $d(P_2[Q/x]) = d(P_2)$. Thus, $d(P[Q/x]) = d(P_1[Q/x]P_2[Q/x]) = d(P_1[Q/x]) + d(P_2[Q/x]) = d(P_1) + d(P_2) = d(P)$. If $P \equiv \lambda z.P_1$, then $d(P) = 1 + d(P_1)$. If $z \equiv x$, then $P[Q/x] \equiv P$ and the result follows. If $z \not\equiv x$, then $P[Q/x] \equiv \lambda z.P_1[Q/x]$ where we may assume by α-congruence that $z \notin \mathrm{FV}(Q)$. Then by the induction hypothesis, we have $d(P_1) = d(P_1[Q/x])$. Thus, $d(P) = 1 + d(P_1) = 1 + d(P_1[Q/x]) = d(P[Q/x])$. ◀

▶ **Theorem 3** (strong normalization). *The $\lambda\Phi$-calculus is strongly normalizing.*

Proof. Let M be an arbitrary term. Let $R \equiv (\lambda x.P)Q$ be a redex in M and let M change to M' by contracting R. If R is an opponent redex, then by induction on M we show that $d(M') < d(M)$:

- If $M \equiv N_1N_2$, and R is contained in N_i, then let N_i change to N_i' by contracting R. By the induction hypothesis, $d(N_i') < d(N_i)$. Thus, $d(M') < d(M)$. If $M \equiv R$, then $M' \equiv P[Q/x]$. Then $d(M') = d(P) < 1 + d(P) = d(M)$, where the first equality follows from Lemma 2.
- If $M \equiv \lambda z.N$, then R must be contained in N. Let N change to N' by contracting R, so $M' \equiv \lambda z.N'$. By the induction hypothesis, we have $d(N') < d(N)$. Thus, $d(M') \leq 1 + d(N') < 1 + d(N) = d(M)$.

On the other hand, if R is a player redex, then $s(M') < s(M)$ and $d(M') \leq d(M)$. Therefore, in between each of the at most $d(M)$ opponent reductions, there can be at most a finite number of player reductions, and reduction always terminates. ◀

It is useful to have a rough estimate on the complexity of reduction. Note that since each opponent term can at most square the size of the term, the size of any reduct is bounded by $s(M)^{2^d}$, where $d = d(M)$. Therefore, in between each of the at most $d = d(M)$ opponent reductions, there can be at most $s(M)^{2^d}$ player reductions. This leads to normalization in *double exponential time*, $s(M)^{2^{O(d)}}$.

There is some flexibility in what to take as the set of **output values**, Θ (see [13]). At the very least, the set of Φ-normal forms, denoted Φ-NF, should be contained in Θ. For the purposes of this paper, it suffices to assume that $\Theta = \Phi$-NF. In the next section we shall represent polynomial time algorithms by programs in the $\lambda\Phi$-calculus.

3 The Parametric Lambda Calculus and Polynomial Time

To begin we must represent some basic data structures like booleans and boolean strings in the $\lambda\Phi$-calculus, but we shall not require them to be encoded as player terms, so they are not necessarily input values[3]. Other data structures will be introduced as needed.

Booleans

Booleans are represented by the set $\{True, False\}$ and the conditional $Cond$, where:

$$True \equiv \lambda xy.x, \qquad False \equiv \lambda xy.y, \qquad Cond \equiv I$$

Note that $CondTrueMN \rightarrow_\Phi^* M$ and $CondFalseMN \rightarrow_\Phi^* N$ for any $M, N \in \Phi$, and that the terms $True, False, Cond$ belong to $\Phi \cap \Theta$.

Boolean Strings

Boolean strings $w = b_1 b_2 \cdots b_n \in \{0,1\}^*$, are represented using a Church-style encoding as follows:

$$W \equiv \lambda fx.fa_1(fa_2(\cdots fa_{n-1}(fa_n x)\cdots)), \qquad a_i \equiv \begin{cases} zero \equiv \lambda xyz.y, & b_i = 0 \\ one \equiv \lambda xyz.z, & b_i = 1 \end{cases}$$

For example, the boolean string $1011 \in \{0,1\}^*$ is encoded by the opponent term:

$$\lambda fx.fone(fzero(fone(fonex)))$$

Note that $d(W) \leq 1$ for all string encodings W; this fact will be important in the proof of Theorem 5. Note that boolean strings are in Φ-normal form, but they are not necessarily input values. Thus, boolean strings are not in general duplicable in our setting. For this reason a slightly more general notion of representability is required:

▶ **Definition 4.** A predicate $A \subseteq \{0,1\}^*$ is representable in the $\lambda\Phi$-calculus if there exists a context $C[.]$ such that, for all $w \in \{0,1\}^*$, $C[W] \rightarrow_\Phi^* Bool$, where W is the encoding of w (as defined above) and $Bool \equiv True$ if $w \in A$ and $Bool \equiv False$ if $w \notin A$. In this case, the context $C[.]$ is said to represent the predicate in the $\lambda\Phi$-calculus.

We now prove one of the main results of this paper:

▶ **Theorem 5** (soundness). *Let $C[.]$ represent a predicate in the $\lambda\Phi$-calculus. Then, for all $w \in \{0,1\}^*$, the term $C[W]$, where W is the encoding of w, reduces to $True$ or $False$ in time polynomial in the size/length, denoted $|w|$, of w.*

Proof. We define a simple call-by-value operational semantics that proves judgements of the

[3] An alternative system in which all of the required basic data structures are input values will be sketched in the conclusion.

form $M \Downarrow_{\mathbf{V}} N$, where M is any lambda term and N is a term in Φ-normal form:

$$\frac{(M_i \Downarrow_{\mathbf{V}} N_i)_{i \leq m}}{xM_1 \cdots M_m \Downarrow_{\mathbf{V}} xN_1 \cdots N_m} \ (var)$$

$$\frac{M \Downarrow_{\mathbf{V}} N}{\lambda x.M \Downarrow_{\mathbf{V}} \lambda x.N} \ (abs)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \in \Phi \quad P[Q'/x]M_1 \cdots M_m \Downarrow_{\mathbf{V}} N}{(\lambda x.P)QM_1 \cdots M_m \Downarrow_{\mathbf{V}} N} \ (head)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \notin \Phi \quad P \Downarrow_{\mathbf{V}} P' \quad (M_i \Downarrow_{\mathbf{V}} N_i)_{i \leq m}}{(\lambda x.P)QM_1 \cdots M_m \Downarrow_{\mathbf{V}} (\lambda x.P')Q'N_1 \cdots N_m} \ (block)$$

An easy induction on the evaluation tree shows that if $M \Downarrow_{\mathbf{V}} N$, then $M \to_{\Phi}^{*} N$, where N is in Φ-normal form, and the length of this reduction sequence is bounded by the size of the evaluation tree.

Our runtime bound is therefore obtained by bounding the size of the (rooted) evaluation tree – i.e. the total number of rules in the evaluation tree for the judgement $M \Downarrow_{\mathbf{V}} N$. Let n denote the total number of (*head*)-rules contained in such a tree. We show by induction on the height of the tree that the size of the tree is bounded by $(n+1)h$, where h denotes the maximum size of any reduct of M. To this end, we define a partial order on the vertices (proof rules) of the evaluation tree such that $u \leq v$ iff the unique path from the root to v passes through u. Consider the subtree obtained by removing the *right* branch of each of the $p \geq 0$ occurences of minimal (with respect to the above partial order) (*head*)-rules. This subtree has size bounded by $s(M) \leq h$ as each rule in the subtree decreases the size of the term. By the induction hypothesis, each of the removed branches, which end with judgements of the form $P[Q'/x]M_1 \cdots M_m \Downarrow_{\mathbf{V}} N$, has size bounded by $(n_i+1)h$, where n_i $(1 \leq i \leq p)$ denotes the number of (*head*)-rules in branch i. Therefore, the total size of the evaluation tree is bounded by $h + \sum_{i \leq p}(n_i+1)h = (1 + \sum_{i \leq p} n_i + p)h = (1+n)h$, as claimed.

Therefore, it follows by the discussion immediately following the proof of Theorem 3 that the size of the evaluation tree is bounded by $s(C[W])^{2^{O(d)}}$, where d is the degree of $C[W]$, as both n are h are so bounded. And since $s(C[W]) = O(|w|)$ and d remains fixed, the bound is in fact polynomial in $|w|$ (albeit with possibly large degree). Therefore, our reduction machine, as defined by the operational semantics above, runs in polynomial time. ◀

3.1 Polynomial Time Completeness

In this section we shall use the notation $M^n N \equiv N$, if $n = 0$, and $M^{n+1}N \equiv M(M^n N)$, if $n > 0$.

We begin by defining a context $Iter_{P,M}[.]$ which is used to iterate a player term $M \in \Phi$ a polynomial $P(n)$ number of times. More precisely, we claim:

$$Iter_{P,M}[W] \to_{\Phi}^{*} \lambda y.M^{P(n)}y \in \Phi \tag{1}$$

where $n = |w|$. Recall that any polynomial with natural number coefficients can be represented in *Horner normal form*. For example, the polynomial $2n^3 + 4n^2 + 3n + 5$ is represented in Horner normal form as $(((2)n+4)n+3)n+5$. Given a polynomial $P(n)$, which is either a constant a_0 or has the form $P(n) = P_1(n)n + a_0$, where $P_1(n)$ is in Horner normal form, the context $Iter_{P,M}[.]$ is defined inductively on the structure of P as follows:

$$\begin{aligned} Iter_{a_0,M}[.] &\equiv \lambda y.M^{a_0}y \\ Iter_{P,M}[.] &\equiv \lambda y.M^{a_0}([.](K\,Iter_{P_1,M}[.])y) \qquad (K \equiv \lambda xy.x) \end{aligned}$$

We argue by induction on the structure of the polynomial that this context satisfies reduction (1). Indeed, the base case is clear. Otherwise we suppose $P(n) = (P_1(n))n + a_0$. By the induction hypothesis we have that $Iter_{P_1,M}[W] \to_\Phi^* \lambda y.M^{P_1(n)}y \in \Phi$, where $n = |w|$. Then:

$$\begin{array}{rcl} Iter_{P,M}[W] & \equiv & \lambda x.M^{a_0}(W(KIter_{P_1,M}[W])x) \\ & \to_\Phi^* & \lambda x.M^{a_0}(W(K\lambda y.M^{P_1(n)}y)x) \\ & \to_\Phi^* & \lambda x.M^{a_0}((\lambda y.M^{P_1(n)}y)^n x) \\ & \to_\Phi^* & \lambda x.M^{a_0}(M^{(P_1(n))n}x) \\ & \equiv & \lambda x.M^{(P_1(n))n+a_0}x \\ & \equiv & \lambda x.M^{P(n)}x \in \Phi \end{array}$$

This finishes the induction. Therefore, for any player term N, we have:

$$Iter_{P,M}[W]N \to_\Phi^* M^{P(n)}N \in \Phi$$

Note, in particular, that $\lambda f.Iter_{P,f}[W] \to_\Phi^* \lambda fx.f^{P(n)}x$, which is the Church representation of $P(n)$. We shall use this iteration combinator to iterate the transition function of a space-bounded Turing machine a polynomial number of times. But first we need the following preliminaries.

Affine Tensor Products

Given player terms $N_1, \ldots, N_m \in \Phi$, we write $N_1 \otimes \cdots \otimes N_m$ for the term $\lambda x.xN_1 \cdots N_m$ and Prj_i for the term $\lambda f.f(\lambda x_1 \ldots x_m.x_i)$ for each $1 \leq i \leq m$. Note that $N_1 \otimes \cdots \otimes N_m, Prj_i \in \Phi$ and satisfy: $Prj_iN_1 \otimes \cdots \otimes N_m \to_\Phi^* N_i$ for each $1 \leq i \leq m$. Occasionally we shall also use the notation $\lambda x_1 \otimes \cdots \otimes x_m.M$ for the term $\lambda f.f(\lambda x_1 \ldots x_m.M)$. This satisfies $(\lambda x_1 \otimes \cdots \otimes x_m.M)(N_1 \otimes \cdots \otimes N_m) \to_\Phi^* M[N_1/x_1] \cdots [N_m/x_m]$ for any terms $M, N_1, \ldots, N_m \in \Phi$. For example, Prj_i could be written instead as $\lambda x_1 \otimes \cdots \otimes x_m.x_i$ for each $1 \leq i \leq m$.

Lists

Lists are encoded using player terms for the constructors *nil* and *cons* as follows:

$$nil \equiv \lambda xy.y \qquad H :: T \equiv \lambda x.xHT \equiv H \otimes T$$

Note that $nil \in \Phi$ and $H :: T \in \Phi$ iff $H, T \in \Phi$. We shall assume that "::" associates to the right.

Space-Bounded Turing Machines

Suppose we are given a Turing machine with k states, tape alphabet $\{\sqcup, 0, 1\}$, and input alphabet $\{0, 1\}$, where $\sqcup$ is a special symbol for "blank". These three symbols are encoded, respectively, by the terms $blank \equiv \lambda xyz.x$, $zero \equiv \lambda xyz.y$ and $one \equiv \lambda xyz.z$. For convenience, we shall assume that the set of states always contains (distinct) special *accepting* and *rejecting* states, and that the machine cannot change states once it reaches one of these two terminating states. Moreover, we assume that the tape is sufficiently large so that either the accepting or rejecting state is always reached before the machine encounters either end of the tape.

A configuration of the Turing machine is encoded as an affine triple tensor product, $S \otimes L \otimes R$, where S encodes the current state, L encodes the left part of the tape (in reverse

order), and R encodes the right part of the tape. The head of the Turing machine is always assumed to be positioned on the head of R. The left and right parts of the tape are encoded as lists of the tape alphabet. The k states S_i are encoded as follows. Suppose, for example, in state i, the machine's instructions, upon reading the symbol on the head, are:

$$\begin{aligned} \sqcup &\mapsto (s_{j1}, 0, \mathit{move\_right}) \\ 0 &\mapsto (s_{j2}, 1, \mathit{move\_left}) \\ 1 &\mapsto (s_{j3}, \sqcup, \mathit{move\_left}) \end{aligned}$$

Then state S_i is encoded as follows:

$$S_i \equiv \lambda x_1 \dots x_k, h_1 \otimes t_1, h_2 \otimes t_2.h_2(F_1)(F_2)(F_3)x_1 \cdots x_k h_1 t_1 t_2$$

$$\begin{aligned} F_1 &\equiv \lambda x_1 \dots x_k h_1 t_1 t_2.x_{j1} \otimes (\mathit{zero} :: h_1 :: t_1) \otimes t_2 \\ F_2 &\equiv \lambda x_1 \dots x_k h_1 t_1 t_2.x_{j2} \otimes t_1 \otimes (h_1 :: \mathit{one} :: t_2) \\ F_3 &\equiv \lambda x_1 \dots x_k h_1 t_1 t_2.x_{j3} \otimes t_1 \otimes (h_1 :: \mathit{blank} :: t_2) \end{aligned}$$

The transition function is then defined as $T \equiv \lambda z \otimes l \otimes r.zS_1 \cdots S_k lr$ and satisfies $T(S_i \otimes L \otimes R) \to_\Phi^* S_j \otimes L' \otimes R'$, where S_j is the new current state and L' and R' are encodings of the updated left and right parts of the tape after one iteration of the machine. The special accepting and rejecting states simply remain in the same state and write *one* or *zero*, respectively, on the head of the tape. Finally, observe that there is a term *out* (encoded via appropriate projections) that returns the head of the right tape from a given configuration of the machine. Note that all the terms in this encoding belong to Φ.

▶ **Theorem 6** (completeness). *If a predicate is computable by a Turing machine in polynomial time $P(n)$ and polynomial space $Q(n)$[4], then it is representable in the $\lambda\Phi$-calculus by a context $C[.]$.*

Proof. Observe that there is a player term, denoted *pad*, which satisfies $\mathit{pad}(S \otimes L \otimes R) \to_\Phi^* S \otimes (\mathit{blank} :: L) \otimes (\mathit{blank} :: R)$. Then $\mathit{Iter}_{Q,\mathit{pad}}[W](S_1 \otimes \mathit{nil} \otimes \mathit{nil})$ pads out the tape sufficiently for the full computation of the machine and puts it in the initial state S_1. Next, apply a player context *write* such that $\mathit{write}[W](S \otimes L \otimes R) \to_\Phi^* S \otimes L \otimes (W(\lambda xy.(x :: y))R)$, which writes the binary string w onto the right part of the tape. Now apply $\mathit{Iter}_{P,T}[W]$, where T is the encoding of the transition function of the machine (as described above), to iterate the transition function $P(|w|)$ times, which suffices for the machine to reach a terminating state. Finally, use the player term *out* (mentioned above) to return the head of the right tape R and apply it to the arguments $I, \mathit{False}, \mathit{True}$ to get a boolean value which indicates whether the machine is in an accepting or a rejecting state. ◀

4 Various Extensions

In this section we investigate a few natural extensions of the $\lambda\Phi$-calculus. The first extends the set of input values to include the so-called Φ-valuable terms (defined below). This larger set of input values allows for further flexibility in defining programs, but at the expense of strong normalization. The second extension adds explicit products to the language for the purpose of characterizing nondeterministic polynomial computations.

[4] Of course, the explicit space bound $Q(n)$ is not necessary here since one may simply take $Q(n) = P(n)$. The explicit accounting of both time and space resources in the encoding of TMs is similar to that given in [9].

4.1 Φ-valuability

In order to reduce a redex $(\lambda x.P)Q$ in the $\lambda\Phi$-calculus, it is first necessary to reduce Q to an input value. However, suppose we relax this condition as follows. Recall that a term M is called **Φ-valuable** if there is an $N \in \Phi$ such that $M \to_\Phi^* N$. Let Φ_v denote the set of Φ-valuable terms.

▶ **Theorem 7.** *Φ_v forms a set of input values such that $\Phi \subset \Phi_v$. Therefore, the reduction relation $\to_{\Phi_v}^*$ is confluent by Theorem 1.2.5 in [13].*

Proof. Condition 1 is immediate. For condition 2, let $P, Q \in \Phi_v$, so there are terms $P', Q' \in \Phi$ such that $P \to_\Phi^* P'$ and $Q \to_\Phi^* Q'$. Then $P[Q/x] \to_\Phi^* P'[Q'/x]$ (by Lemmas 1.2.21 and 1.2.22 in [13]). But then $P'[Q'/x]$ is in Φ by the substitution closure of Φ. Hence $P[Q/x] \in \Phi_v$. For condition 3, let $M \equiv C[(\lambda x.P)Q]$ and $N \equiv C[P[Q/x]]$. Then $Q \in \Phi_v$ in order for this reduction to happen. So there is a term $Q' \in \Phi$ such that $Q \to_\Phi^* Q'$. Then $M \to_\Phi^* C[(\lambda x.P)Q'] \to_\Phi C[P[Q'/x]]$. But since $M \in \Phi_v$, there is a term $M' \in \Phi$ such that $M \to_\Phi^* M'$. So by the confluence of $\to_\Phi^*$ there is a term $M'' \in \Phi$ such that $C[P[Q'/x]] \to_\Phi^* M''$. But then $N \to_\Phi^* C[P[Q'/x]] \to_\Phi^* M''$, which shows that $N \in \Phi_v$. ◀

The $\lambda\Phi_v$-calculus is *not* strongly normalizing. For example, consider the term $D \equiv \lambda y.KI(yy)I$, which is Φ-valuable since it reduces to $\lambda yx.x \in \Phi$. However, $DD \to_{\Phi_v} KI(DD)I \to_{\Phi_v} KI(KI(DD)I)I \to_{\Phi_v} \cdots$, which leads to an infinite reduction sequence. Nevertheless, every term in the $\lambda\Phi_v$-calculus has a (unique) normal form:

▶ **Theorem 8** (weak normalization)**.** *Every term in the $\lambda\Phi_v$-calculus has a unique normal form.*

Proof. Let M be a term. If there are no Φ_v-redexes, then M is a Φ_v-nf and we are done. Otherwise, let $(\lambda x.P)Q$ be a Φ_v-redex in M. Then $Q \in \Phi_v$, and thus $Q \to_\Phi^* Q'$, where $Q' \in \Phi$. Consider the Φ-reduction of M: $M \equiv C[(\lambda x.P)Q] \to_\Phi^* C[(\lambda x.P)Q'] \to_\Phi C[P[Q'/x]]$. If $C[P[Q'/x]]$ is a Φ_v-nf, then we are done. Otherwise, repeat this procedure and extend the Φ-reduction of M. This process must terminate since there are no infinite Φ-reductions. Therefore, M is weakly normalizable. Uniqueness comes from the confluence property (Theorem 7). ◀

Of course, by including the Φ-valuable terms in the set of input values, we have not obtained any new complexity results. However, Theorem 7 is an interesting general result about the parametric lambda calculus that is true of any set of input values and may have future applications. Theorem 8 is less applicable as it requires the additional fact that the $\lambda\Phi$-calculus is strongly normalizing.

4.2 Explicit Products and Nondeterminism

In this section we study nondeterminism in the parametric lambda calculus with the use of explicit products. Of course, one could simply add a new term constructor $M + N$ (*sum*) together with nondeterministic projections, but the resulting system would not be confluent (by construction). Here we present an alternative approach based on the idea of a polynomial verifier.

Let $\Lambda_\times$ denote the set of lambda calculus with explicit products:

$$M, N ::= x \mid MN \mid \lambda x.M \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M$$

where $x \in \text{Var}$. The notions of substitution and α-equivalence are extended in the obvious manner. Here, the term $\langle M, N\rangle$ is called an **explicit product** and $\pi_i M$ are called **projections**. For a given set of input values $\Delta \subseteq \Lambda_\times$, we extend $\to_\Delta$ as the contextual closure of the following rules:

$$\begin{array}{lll} (\lambda x.M)N & \to_\Delta & M[N/x] \quad \text{iff } N \in \Delta \\ \pi_1\langle M, N\rangle & \to_\Delta & M \\ \pi_2\langle M, N\rangle & \to_\Delta & N \end{array}$$

As before, we let $\to_\Delta^*$ denote the reflexive and transitive closure of $\to_\Delta$. The following is a straightforward generalization of Theorem 1.2.5 in [13].

▶ **Theorem 9.** *The $\lambda\Delta$-calculus with explicit products and $\to_\Delta^*$ as defined above is confluent.*

Proof. The definitions of deterministic parallel reduction $\hookrightarrow_\Delta$ and nondeterminsitic parallel reduction $\Rightarrow_\Delta$, Definition 1.2.19 in [13], are extended with the following clauses:

5. $M \hookrightarrow_\Delta M'$, $N \hookrightarrow_\Delta N'$ imply $\langle M, N\rangle \hookrightarrow_\Delta \langle M', N'\rangle$;
6. $M \hookrightarrow_\Delta M'$ and $M \not\equiv \langle M_1, M_2\rangle$ imply $\pi_i M \hookrightarrow_\Delta \pi_i M'$, for $i \in \{1, 2\}$;
7. $M_1 \hookrightarrow_\Delta M_1'$, $M_2 \hookrightarrow_\Delta M_2'$ imply $\pi_i\langle M_1, M_2\rangle \hookrightarrow_\Delta M_i'$, for $i \in \{1, 2\}$.

5. $M \Rightarrow_\Delta M'$, $N \Rightarrow_\Delta N'$ imply $\langle M, N\rangle \Rightarrow_\Delta \langle M', N'\rangle$;
6. $M \Rightarrow_\Delta M'$ implies $\pi_i M \Rightarrow_\Delta \pi_i M'$, for $i \in \{1, 2\}$;
7. $M_1 \Rightarrow_\Delta M_1'$, $M_2 \Rightarrow_\Delta M_2'$ imply $\pi_i\langle M_1, M_2\rangle \Rightarrow_\Delta M_i'$, for $i \in \{1, 2\}$.

Then Lemmas 1.2.21, 1.2.22, Property 1.2.23 and Lemma 1.2.24 in [13] all have straightforward generalizations by checking the extra cases. The details are left to the reader. Finally, the proof of Lemma 1.2.25 and the rest of the proof of Theorem 1.2.5 are unchanged. ◀

We expand the set of input/player values to include affine linear terms with explicit products: Let $\Phi_\times$ be defined the smallest subset of $\Lambda_\times$ satisfying the following closure properties:

- $x \in \text{Var}$ implies $x \in \Phi_\times$,
- If $M, N \in \Phi_\times$, then $(MN) \in \Phi_\times$,
- If $M \in \Phi_\times$, then $\pi_i M \in \Phi_\times$ for $i \in \{1, 2\}$,
- If $M, N \in \Phi_\times$, then $\langle M, N\rangle \in \Phi_\times$,
- If $M \in \Phi_\times$ and $x \in \text{Var}$ and x occurs free at most once in M, then $(\lambda x.M) \in \Phi_\times$.

Note that if $M \in \Phi_\times$ and is closed, then M is affine linear even with (additive) explicit products. One could define a more general notion of input values based on the notion of *slice* (as defined in [11], for example) which includes terms like $\lambda x.\langle x, x\rangle$ and is strongly normalizing. However, such a system would require a lazy reduction strategy for explicit products as well as pointers to avoid an exponential explosion in the size of a term (cf. [12]). We don't believe this added complication is necessary here.[5]

The following two theorems are straightforward generalizations of Theorems 1 and 3, so the proofs have been omitted.

▶ **Theorem 10.** *$\Phi_\times$ forms a set of input values such that $\Phi \subset \Phi_\times$. Therefore, the reduction relation $\to_{\Phi_\times}^*$ is confluent by Theorem 9.*

▶ **Theorem 11.** *The $\lambda\Phi_\times$-calculus is strongly normalizing.*

[5] However, we do believe this more general set of input values is the starting point for a characterization of PSPACE (see [12]).

We extend the simple call-by-value operational semantics introduced in the proof of Theorem 5 with the following rules:

$$\frac{P_1 \Downarrow_{\mathbf{V}} P_1' \quad P_2 \Downarrow_{\mathbf{V}} P_2' \quad (M_i \Downarrow_{\mathbf{V}} N_i)_{i \leq m}}{\langle P_1, P_2\rangle M_1 \cdots M_m \Downarrow_{\mathbf{V}} \langle P_1', P_2'\rangle N_1 \cdots N_m} \ (pair)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \equiv \langle Q_1', Q_2'\rangle \quad Q_i' M_1 \cdots M_m \Downarrow_{\mathbf{V}} N}{\pi_i Q M_1 \cdots M_m \Downarrow_{\mathbf{V}} N} \ (proj_i)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \not\equiv \langle Q_1', Q_2'\rangle \quad (M_i \Downarrow_{\mathbf{V}} N_i)_{i \leq m}}{\pi_i Q M_1 \cdots M_m \Downarrow_{\mathbf{V}} \pi_i Q' N_1 \cdots N_m} \ (block_i)$$

Once again, an easy induction on the evaluation tree shows that if $M \Downarrow_{\mathbf{V}} N$, then $M \to_{\Phi_\times}^* N$, where N is in $\Phi_\times$-normal form, and the length of this reduction sequence is bounded by the size of the evaluation tree. On the other hand, by Theorem 10, if $M \to_{\Phi_\times}^* N$, where N is in $\Phi_\times$-normal form, then M evaluates to N according to the operational semantics defined above.

Additive Booleans

Explicit products allow for an alternative definition of booleans and conditional:

$$Proj_1 \equiv \lambda f.\pi_1 f \quad Proj_2 \equiv \lambda f.\pi_2 f \quad if\ b\ then\ M\ else\ N \equiv \lambda b.b\langle M, N\rangle$$

A term M is called **eventually true** if there exists a sequence of additive booleans $Proj_{i_1}, \ldots, Proj_{i_k}$ such that $M Proj_{i_1} \cdots Proj_{i_k} \to_{\Phi_\times}^* True$.

▶ **Definition 12.** A predicate $A \subseteq \{0,1\}^*$ is representable in the $\lambda\Phi_\times$-calculus if there is a context $C[.]$ such that, for all $w \in \{0,1\}^*$, $w \in A$ iff $C[W]$ is eventually true.

We have the following result:

▶ **Theorem 13.** *A predicate $A \subseteq \{0,1\}^*$ is representable in the $\lambda\Phi_\times$-calculus by a context $C[.]$ iff it is computable in nondeterministic polynomial time (NP).*

Proof. ($\Rightarrow$) Suppose a predicate A is representable by a context $C[.]$ in the $\lambda\Phi_\times$-calculus. If $w \in A$, then a straightforward generalization of Theorem 5 shows that, for any choice of projections $Proj_{i_1}, Proj_{i_2}, \ldots, Proj_{i_k}$, the term $C[W]Proj_{i_1}Proj_{i_2}\cdots Proj_{i_k}$ reduces in time bounded by a polynomial in $s(C[W]Proj_{i_1}Proj_{i_2}\cdots Proj_{i_k})$ to $True$.[6] Moreover, note that k must be bounded by a polynomial in $|w|$. Indeed, each projection input requires a head lambda abstraction. This head lambda abstraction cannot itself be a projection term because otherwise the normal form would have a (projection) block. And there can only be a polynomial in $|w|$ such head reductions. Therefore, the entire reduction is polynomial time in $|w|$ only.

($\Leftarrow$) Conversely, let A be a predicate computable on a nondeterministic Turing machine in polynomial time $P(n)$ and polynomial space $Q(n)$ (i.e. the maximum time and space used by any computational branch). The encoding of nondeterministic Turing machines is based on the encoding of deterministic Turing machines found in Section 3.1. However, for a nondeterministic machine, we assume a pair of transition functions T_l and T_r instead of just

[6] As noted above, we may assume, by Theorem 10, that this reduction sequence is determined by the operational semantics.

one. The following player term can be iterated $P(n)$ times using the iteration combinator from Section 3.1:

$$Branch \equiv \lambda f z b.b\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle f z$$

Let w be any binary word and let $n = |w|$. Let $Initial[W] \to^*_{\Phi_\times} Config$ initialize the machine by padding the tape out to size $Q(n)$, writing w on the initial segment of the tape, and putting it in the start state. Finally, let out be a term that reduces to $True$ if a given configuration is accepting and reduces to $False$ otherwise.

Let Z_k denote the normal form of $Branch^k out$, which has the form:

$$\begin{aligned} Z_k &\equiv \lambda z b.b\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle Z_{k-1} z, \qquad k > 0 \\ Z_0 &\equiv out \end{aligned}$$

If $w \in A$, then there exists a sequence of $i_1, \ldots, i_k$, with $i_j \in \{1, 2\}$ and $k = P(n)$, specifying a path down the nondeterministic evaluation tree to an accepting leaf. This path is encoded by the series of projections $Proj_{i_1}, \ldots, Proj_{i_k}$ and verified as follows:

$$\begin{aligned} & Iter_{P,Branch}[W]\, out\, Config\, Proj_{i_1} \cdots Proj_{i_k} \\ \to^*_{\Phi_\times} \; & Branch^k\, out\, Config\, Proj_{i_1} \cdots Proj_{i_k} \\ \to^*_{\Phi_\times} \; & (\lambda b.b\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle Z_{k-1}\, Config) Proj_{i_1} \cdots Proj_{i_k} \\ \to^*_{\Phi_\times} \; & Proj_{i_1} \langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle Z_{k-1}\, Config\, Proj_{i_2} \cdots Proj_{i_k} \\ \to^*_{\Phi_\times} \; & Z_{k-1}\, Config_1\, Proj_{i_2} \cdots Proj_{i_k} \\ \to^*_{\Phi_\times} \; & \cdots \\ \to^*_{\Phi_\times} \; & Z_1\, Config_{k-1}\, Proj_{i_k} \\ \to^*_{\Phi_\times} \; & out\, Config_k \\ \to^*_{\Phi_\times} \; & True \end{aligned}$$

Moreover, this reduction proceeds according to the operational semantics defined above. Thus, $(Iter_{P,Branch}[W]\, out\, Config) Proj_{i_1} \cdots Proj_{i_k} \Downarrow_{\mathbf{V}} True$. On the other hand, if $w \notin A$, then all such reductions reduce to $False$. ◀

5 Conclusion

In this paper we have demonstrated that a characterization of polynomial time computations can be obtained in the lambda calculus without requiring any typing information and/or ad hoc extensions to the language. Indeed, the characterization is obtained simply by restricting the set of input values to the so-called pseudo-affine terms. Moreover, a characterization of nondeterministic polynomial time is obtained with the addition of explicit products.

It would be interesting to investigate other (decidable) instantiations of the parametric lambda calculus in the context of Implicit Computational Complexity. For example, the choice to allow weakening in the language was made simply because it made the encoding of polynomial time TMs much more natural. However, if we change the final clause in the definition of Φ to specify that x occurs *exactly* once in P, we conjecture that this smaller set of input values also characterizes polynomial time. In this case, the encoding of polynomial bounded TMs might follow that in [11].

Finally, one unfortunate aspect of our characterization is that binary words (and Church numerals) are not in general input values. For this reason our definitions of representability use contexts instead, which is not standard. We consider two possibilites for dealing with

this situation. First, one could use a player encoding of binary strings instead (in the style of the Barendregt numerals [3, 13]), together with a more natural definition of representability. However, it is not difficult to show that this leads to a system for linear time computations only. A second possibility is to use a system of *abstract binary numerals* (see [8]) instead of pure terms to represent binary strings. In this case, four new atomic constants, denoted $\epsilon, \sigma_1, \sigma_2$ and Z, are added to the system such that ϵ, σ_1 and σ_2 belong to the set of input values, but Z does not. Then any binary string $w \in \{0,1\}^*$ can be represented by an input value $\widehat{w}$ in the obvious way. Furthermore, we add the contextual closure of the following reduction rule:

$$Z\widehat{w} \rightarrow_\Phi \overline{w}$$

where $\overline{w}$ is the Church-style representation of w described in Section 3.[7] This *arithmetical extension* [8] of the $\lambda\Phi$-calculus leads to an *applied* system for polynomial time computations, similar to the pure system presented in this paper. It is an alternative if a standard notion of representability is desired.

Acknowledgements.

The author would like to thank Simona Ronchi Della Rocca for reading a preliminary draft of this paper and offering some useful suggestions for improvement. The author would also like to thank the anonymous referees for their valuable comments and suggestions regarding the presentation of the results of this paper.

References

1 Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS'14, Vienna, Austria, July 14–18, 2014*, page 8. ACM, 2014.

2 Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 266–275. IEEE Computer Society, 2004.

3 Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, New-York, Oxford, 1981.

4 J. Chrząszcz and A. Schubert. The role of polymorphism in the characterisation of complexity by soft types. (To appear).

5 Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for *lambda*-calculus. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2007.

6 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

[7] Note that if Z is permitted to be an input value, then $\overline{w}$ must be as well. Otherwise, the system would not satisfy reduction closure (i.e. condition 3 in the definition of input values). This leads to (at least) elementary time complexity.

7 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

8 J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008.

9 Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004.

10 Daniel Leivant. Finitely stratified polymorphism. *Inf. Comput.*, 93(1):93–113, 1991.

11 Harry G. Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*, volume 2841 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2003.

12 B. Redmond. Bounded combinatory logic and lower complexity. (To appear).

13 Simona Ronchi Della Rocca and Luca Paolini. *The parametric lambda calculus: a metamodel for computation*. Texts in theoretical computer science. Springer-Verlag, New York, 2004.

14 Aleksy Schubert. The complexity of beta-reduction in low orders. In *TLCA*, pages 400–414, 2001.

15 H. Schwichtenberg. Definierbare Funktionen im λ-Kalkül mit Typen. *Arkhiv für mathematische Logik und Grundlagenforschung*, 17:113–114, 1976.

16 R. Statman. The typed lambda calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–82, 1979.

Fibrations of Tree Automata

Colin Riba

ENS de Lyon, Université de Lyon, LIP*, France
colin.riba@ens-lyon.fr

Abstract

We propose a notion of morphisms between tree automata based on game semantics. Morphisms are winning strategies on a synchronous restriction of the linear implication between acceptance games. This leads to split indexed categories, with substitution based on a suitable notion of synchronous tree function. By restricting to tree functions issued from maps on alphabets, this gives a fibration of tree automata. We then discuss the (fibrewise) monoidal structure issued from the synchronous product of automata. We also discuss how a variant of the usual projection operation on automata leads to an existential quantification in the fibered sense. Our notion of morphism is correct (it respects language inclusion), and in a weaker sense also complete.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Formal Languages

Keywords and phrases Tree automata, Game semantics, Categorical logic

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.302

1 Introduction

This paper proposes a notion of morphism between tree automata based on game semantics. We follow the Curry-Howard-like slogan: *Automata as objects, Executions as morphisms.*

We consider general alternating automata on infinite ranked trees. These automata encompass Monadic Second-Order Logic (MSO) and thus most of the logics used in verification [6]. Tree automata are traditionally viewed as positive objects: one is primarily interested in satisfaction or satisfiability, and the primitive notion of quantification is existential. In contrast, Curry-Howard approaches tend to favor proof-theoretic oriented and negative approaches, *i.e.* approaches in which the predominant logical connective is the implication, and where the predominant form of quantification is universal.

We consider full infinite ranked trees, built from a non-empty finite set of directions D and labeled in non-empty finite alphabets Σ. The base category **Tree** has alphabets as objects and morphisms from Σ to Γ are $(\Sigma \to \Gamma)$-labeled D-ary trees.

The fibre categories are based on a generalization of the usual acceptance games, where for an automaton $\mathcal{A}$ on alphabet Γ (denoted $\Gamma \vdash \mathcal{A}$), input characters can be precomposed with a tree morphism $M \in \mathbf{Tree}[\Sigma, \Gamma]$, leading to substituted acceptance games of type $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$. Usual acceptance games, which correspond to the evaluation of $\Sigma \vdash \mathcal{A}$ on a Σ-labeled input tree, are substituted acceptance games $\mathbf{1} \vdash \mathcal{G}(\mathcal{A}, t)$ with $t \in \mathbf{Tree}[\mathbf{1}, \Sigma]$. Games of the form $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$ are the objects of the fibre category over Σ.

For morphisms, we introduce a notion of "synchronous" simple game between acceptance games. We rely on Hyland & Schalk's functor (denoted HS) from simple games to **Rel** [9]. A synchronous strategy $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A}, M) \multimap\!\!\circledast\, \mathcal{G}(\mathcal{B}, N)$ is a strategy in the simple game $\mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)$ required to satisfy (in **Set**) a diagram of the form of (1) below,

* LIP is associated with UMR CNRS – ENS Lyon – UCB Lyon 1 – INRIA 5668.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 302–316
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

expressing that $\mathcal{A}$ and $\mathcal{B}$ are evaluated along the same path of the tree and read the same input characters:

$$\begin{array}{ccc} \mathrm{HS}(\sigma) & \longrightarrow & \mathcal{G}(\mathcal{B}, N) \\ \downarrow & & \downarrow \\ \mathcal{G}(\mathcal{A}, M) & \longrightarrow & (D + \Sigma)^* \end{array} \tag{1}$$

This gives a split fibration game of tree automata and acceptance games. When restricting the base to *alphabet* morphisms (*i.e.* functions $\Sigma \to \Gamma$), substitution can be internalized in automata. By change-of-base of fibrations, this leads to a split fibration aut. In the fibers of aut, the substituted acceptance games have finite-state winning strategies, whose existence can be checked by trivial adaptation of usual algorithms.

Each of these fibrations is monoidal in the sense of [16], by using a natural synchronous product of tree automata. We also investigate a linear negation, as well as existential quantifications, obtained by adapting the usual projection operation on non-deterministic automata to make it a left-adjoint to weakening, the adjunction satisfying the usual Beck-Chevalley condition.

Our linear implication of acceptance games seems to provide a natural notion of prenex universal quantification on automata not investigated before. As expected, if there is a synchronous winning strategy $\sigma \Vdash \mathcal{A} \multimap \mathcal{B}$, then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ (*i.e.* each input tree accepted by $\mathcal{A}$ is also accepted by $\mathcal{B}$). Under some assumptions on $\mathcal{A}$ and $\mathcal{B}$ the converse holds: $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ implies $\sigma \Vdash \mathcal{A} \multimap \mathcal{B}$ for some σ.

At the categorical level, thanks to (1), the constructions mimic relations in slices categories $\mathbf{Set}/(D + \Sigma)^*$ of the co-domain fibration: substitution is given by a (well chosen) pullback, and the monoidal product of automata is issued from the Cartesian product of plays in $\mathbf{Set}/(D + \Sigma)^*$ (*i.e.* also by a well chosen pullback).

The paper is organized as follows. Section 2 presents notations for trees and tree automata. Our notions of substituted acceptance games and synchronous arrow games are then discussed in Sect. 3. Substitution functors and the corresponding fibrations are presented in Sect. 4, and Section 5 overviews the monoidal structure. We then state our main correctness results in Sect. 6. Section 7 presents existential quantifications and quickly discusses non-deterministic automata.

2 Preliminaries

Fix a singleton set $\mathbf{1} = \{\bullet\}$ and a finite non-empty set D of (tree) *directions*.

Alphabets and Trees. We write $\Sigma, \Gamma, \ldots$ for *alphabets*, *i.e.* finite non-empty sets. We let $\mathbf{Alph}$ be the category whose objects are alphabets and whose morphisms $\beta \in \mathbf{Alph}[\Sigma, \Gamma]$ are functions $\beta : \Sigma \to \Gamma$.

We let $\mathbf{Tree}[\Sigma]$ be the set of Σ-labeled full D-ary trees, *i.e.* the set of maps $T : D^* \to \Sigma$. Let $\mathbf{Tree}$ be the category with *alphabets* as objects and with morphisms $\mathbf{Tree}[\Sigma, \Gamma] := \mathbf{Tree}[(\Sigma \to \Gamma)]$, *i.e.* $(\Sigma \to \Gamma)$-labeled trees. Maps $M \in \mathbf{Tree}[\Sigma, \Gamma]$ and $L \in \mathbf{Tree}[\Gamma, \Delta]$ are composed as $L \circ M : p \in D^* \mapsto (a \in \Sigma \mapsto L(p)(M(p)(a)))$ and the identity $\mathrm{Id}_\Sigma \in \mathbf{Tree}[\Sigma, \Sigma]$ is defined as $\mathrm{Id}_\Sigma(p)(a) := a$. Note that $\mathbf{Tree}[\mathbf{1}, \Sigma]$ is in bijection with $\mathbf{Tree}[\Sigma]$.

There is a faithful functor from $\mathbf{Alph}$ to $\mathbf{Tree}$, mapping $\beta \in \mathbf{Alph}[\Sigma, \Gamma]$ to the constant tree morphism $(\_ \mapsto \beta) \in \mathbf{Tree}[\Sigma, \Gamma]$ that we simply write β.

Tree Automata. Alternating tree automata [14] are finite state automata running on full infinite Σ-labeled D-ary trees. Their distinctive feature is that transitions are given by *positive Boolean* formulas with atoms pairs (q, d) of a state q and a tree direction $d \in D$ ((q, d) means that one copy of the automaton should start in state q from the d-th son of the current tree position).

Acceptance for alternating tree automata can be defined either *via* run trees or *via* the existence of winning strategies in *acceptance games* [14]. In both cases, we can w.l.o.g. restrict to transitions given by formulas in (irredundant) disjunctive normal form [15]. In our setting, it is quite convenient to follow the presentation of [18], in which disjunctive normal forms with atoms in $Q \times D$ are represented as elements of $\mathcal{P}(\mathcal{P}(Q \times D))$.

An alternating tree automaton $\mathcal{A}$ on alphabet Σ has the form $(Q, q^\imath, \delta, \Omega)$ where Q is the finite set of states, $q^\imath \in Q$ is the *initial state*, the *acceptance condition* is $\Omega \subseteq Q^\omega$ and following [18], the *transition function* δ has the form

$$\delta \quad : \quad Q \times \Sigma \quad \longrightarrow \quad \mathcal{P}(\mathcal{P}(Q \times D))$$

We write $\Sigma \vdash \mathcal{A}$ if $\mathcal{A}$ is a tree automaton on Σ. Usual acceptance games are described in Sec. 3.1. It is customary to put restrictions on the acceptance condition $\Omega \subseteq Q^\omega$, typically by assuming it is generated from a *Muller* family $\mathcal{F} \in \mathcal{P}(\mathcal{P}(Q))$ as the set of $\pi \in Q^\omega$ such that $\mathrm{Inf}(\pi) \in \mathcal{F}$. We call such automata *regular*[1]. They have decidable emptiness checking and the same expressive power as MSO on D-ary trees (see e.g. the survey [17]).

3 Categories of Acceptance Games and Automata

We present in this Section the categories $\mathbf{SAG}^{(\mathrm{W})}_\Sigma$ of *substituted acceptance games*. Their objects will be *substituted acceptance games* (to be presented in Sect. 3.1) and their morphisms will be strategies in corresponding *synchronous arrow games* (to be presented in Sect. 3.2). Substituted acceptance games and synchronous arrow games are the two main notions we introduce in this paper. Our categories of $\mathbf{Aut}^{(\mathrm{W})}_\Sigma$ of automata will be full subcategories of $\mathbf{SAG}^{(\mathrm{W})}_\Sigma$, while $\mathbf{SAG}^{(\mathrm{W})}_\Sigma$ and $\mathbf{Aut}^{(\mathrm{W})}_\Sigma$ will be the total categories of our fibrations

$$\mathsf{game}^{(\mathrm{W})} \quad : \quad \mathbf{SAG}^{(\mathrm{W})} \quad \longrightarrow \quad \mathbf{Tree} \qquad\qquad \mathsf{aut}^{(\mathrm{W})} \quad : \quad \mathbf{Aut}^{(\mathrm{W})} \quad \longrightarrow \quad \mathbf{Alph}$$

to be presented in Sect. 4.

3.1 Substituted Acceptance Games

Consider a tree automaton $\mathcal{A} = (Q, q^\imath, \delta, \Omega)$ on Γ and a morphism $M \in \mathbf{Tree}[\Sigma, \Gamma]$. The *substituted acceptance game* $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$ is the positive game

$$\mathcal{G}(\mathcal{A}, \vec{M}) \quad := \quad (D^* \times (A_\mathsf{P} + A_\mathsf{O}), E, *, \lambda, \xi, \mathcal{W})$$

whose positions are given by $A_\mathsf{P} := Q$ and $A_\mathsf{O} := \Sigma \times \mathcal{P}(Q \times D)$, whose polarized root is $* := (\varepsilon, q^\imath)$ with $\xi(*) = \mathsf{P}$, whose polarized moves (E, λ) are given by

$$\begin{array}{llcll} \text{from } (D^* \times A_\mathsf{P}) \text{ to } (D^* \times A_\mathsf{O}) : & (p, q) & \xrightarrow{\mathsf{P}} & (p, a, \gamma) & \text{iff } \gamma \in \delta(q, \vec{M}(p)(a)) \\ \text{from } (D^* \times A_\mathsf{O}) \text{ to } (D^* \times A_\mathsf{P}) : & (p, a, \gamma) & \xrightarrow{\mathsf{O}} & (p.d, q) & \text{iff } (q, d) \in \gamma \end{array}$$

[1] By adding states to $\mathcal{A}$ if necessary, one can describe Ω by an equivalent *parity* condition.

and whose winning condition is given by

$$(\varepsilon, q_0) \cdot (\varepsilon, a_0, \gamma_0) \cdot (p_1, q_1) \cdot \ldots \cdot (p_n, q_n) \cdot (p_n, a_n, \gamma_n) \cdot \ldots \quad \in \mathcal{W} \qquad \text{iff} \quad (q_i)_{i\in\mathbb{N}} \in \Omega$$

The input alphabet of $\Gamma \vdash \mathcal{A}$ is Γ, and we use the tree morphism $M \in \mathbf{Tree}[\Sigma, \Gamma]$ in a contravariant way to obtain a game with "input alphabet" Σ, that we emphasize by writing $\Sigma \vdash \mathcal{G}(\mathcal{A}, \vec{M})$. Input characters $a \in \Sigma$ are chosen by P, directions $d \in D$ are chosen by O.

Write $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, \vec{M})$ if σ is a winning P-strategy on $\Sigma \vdash \mathcal{G}(\mathcal{A}, \vec{M})$, and $\Sigma \Vdash \mathcal{G}(\mathcal{A}, M)$ if $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, M)$ for some σ.

Correspondence with usual Acceptance Games. Usual acceptance games model the evaluation of automata $\Sigma \vdash \mathcal{A}$ on input trees $t \in \mathbf{Tree}[\Sigma]$. They correspond to games of the form $\mathbf{1} \vdash \mathcal{G}(\mathcal{A}, \dot{t})$, where $\dot{t} \in \mathbf{Tree}[\mathbf{1}, \Sigma]$ is the tree morphism corresponding to $t \in \mathbf{Tree}[\Sigma]$.

Note that in these cases, A_{O} is of the form $\mathbf{1} \times \mathcal{P}(Q \times D) \simeq \mathcal{P}(Q \times D)$, so that the games $\mathbf{1} \vdash \mathcal{G}(\mathcal{A}, \dot{t})$ are isomorphic to the acceptance games of [18].

▶ **Definition 3.1.** Let $\Sigma \vdash \mathcal{A}$.

(i) $\mathcal{A}$ *accepts* the tree $t \in \mathbf{Tree}[\Sigma]$ if there is a strategy σ such that $\mathbf{1} \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, \dot{t})$.

(ii) Let $\mathcal{L}(\mathcal{A}) \subseteq \mathbf{Tree}[\Sigma]$, the *language* of $\mathcal{A}$, be the set of trees accepted by $\mathcal{A}$.

3.2 Synchronous Arrow Games

Consider games $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$ and $\Sigma \vdash \mathcal{G}(\mathcal{B}, N)$ with $\mathcal{A} = (Q_{\mathcal{A}}, q^{\imath}_{\mathcal{A}}, \delta_{\mathcal{A}}, \Omega_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, q^{\imath}_{\mathcal{B}}, \delta_{\mathcal{B}}, \Omega_{\mathcal{B}})$. Similarly as in Sect. 3.1 above, write

$$A_{\mathsf{P}} := Q_{\mathcal{A}} \qquad A_{\mathsf{O}} := \Sigma \times \mathcal{P}(Q_{\mathcal{A}} \times D) \qquad B_{\mathsf{P}} := Q_{\mathcal{B}} \qquad B_{\mathsf{O}} := \Sigma \times \mathcal{P}(Q_{\mathcal{B}} \times D)$$

We define the *synchronous arrow game*

$$\Sigma \vdash \mathcal{G}(\mathcal{A}, \vec{M}) \multimap\!\circledast\, \mathcal{G}(\mathcal{B}, \vec{N})$$

as the negative game $(V, E, *, \lambda, \xi, \mathcal{W})$ whose positions are given by

$$V \quad := \quad (D^* \times A_{\mathsf{P}}) \times (D^* \times B_{\mathsf{P}}) \;+\; (D^* \times A_{\mathsf{O}}) \times (D^* \times B_{\mathsf{P}}) \;+\; (D^* \times A_{\mathsf{O}}) \times (D^* \times B_{\mathsf{O}})$$

whose polarized root is $* := ((\varepsilon, q^{\imath}_{\mathcal{A}}), (\varepsilon, q^{\imath}_{\mathcal{B}}))$ with $\xi(*) := \mathsf{O}$, whole polarized edges (E, λ) are given in Table 1, and whose winning condition is given by

$$((\varepsilon, q^0_{\mathcal{A}}) \;,\; (\varepsilon, q^0_{\mathcal{B}})) \;\cdot\; \ldots \;\cdot\; ((\varepsilon, q^n_{\mathcal{A}}) \;,\; (\varepsilon, q^n_{\mathcal{B}})) \;\cdot\; \ldots \quad \in \mathcal{W}$$
$$\text{iff} \qquad \big((q^i_{\mathcal{A}})_{i\in\mathbb{N}} \in \Omega_{\mathcal{A}} \implies (q^i_{\mathcal{B}})_{i\in\mathbb{N}} \in \Omega_{\mathcal{B}}\big)$$

Note that P-plays end in positions of the form

$$\begin{array}{lrcl} & ((p, q_{\mathcal{A}}) \;,\; (p, q_{\mathcal{B}})) & \in & (D^* \times A_{\mathsf{P}}) \times (D^* \times B_{\mathsf{P}}) \\ \text{and} & ((p, a, \gamma_{\mathcal{A}}) \;,\; (p, a, \gamma_{\mathcal{B}})) & \in & (D^* \times A_{\mathsf{O}}) \times (D^* \times B_{\mathsf{O}}) \end{array}$$

Each of these position is of homogeneous type, and moreover in each case the D^* and Σ components coincide. On the other hand, O-plays end in positions of the form

$$\begin{array}{lrcl} & ((p, a, \gamma_{\mathcal{A}}) \;,\; (p, q_{\mathcal{A}})) & \in & (D^* \times A_{\mathsf{O}}) \times (D^* \times B_{\mathsf{P}}) \\ \text{and} & ((p, a, \gamma_{\mathcal{A}}) \;,\; (p \cdot d, q_{\mathcal{B}})) & \in & (D^* \times A_{\mathsf{O}}) \times (D^* \times B_{\mathsf{P}}) \end{array}$$

Each of these intermediate position is of heterogeneous type, and in the second one, the D^* components do not coincide.

λ	$\mathcal{G}(\mathcal{A},\vec{M})$	$\multimap\!\circledast$	$\mathcal{G}(\mathcal{B},\vec{N})$	
	$((p,q_{\mathcal{A}})$	,	$(p,q_{\mathcal{B}}))$	
O	$\downarrow$			
	$((p,a,\gamma_{\mathcal{A}})$	,	$(p,q_{\mathcal{B}}))$	if $\gamma_{\mathcal{A}} \in \delta_{\mathcal{A}}(q_{\mathcal{A}},\vec{M}(p)(a))$
P			$\downarrow$	
	$((p,a,\gamma_{\mathcal{A}})$	,	$(p,a,\gamma_{\mathcal{B}}))$	if $\gamma_{\mathcal{B}} \in \delta_{\mathcal{B}}(q_{\mathcal{B}},\vec{N}(p)(a))$
O			$\downarrow$	
	$((p,a,\gamma_{\mathcal{A}})$	,	$(p.d,q'_{\mathcal{B}}))$	if $(q'_{\mathcal{B}},d) \in \gamma_{\mathcal{B}}$
P	$\downarrow$			
	$((p.d,q'_{\mathcal{A}})$	,	$(p.d,q'_{\mathcal{B}}))$	if $(q'_{\mathcal{A}},d) \in \gamma_{\mathcal{A}}$

Figure 1 Moves of $\mathcal{G}(\mathcal{A},\vec{M}) \multimap\!\circledast \mathcal{G}(\mathcal{B},\vec{N})$.

We write $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A},M) \multimap\!\circledast \mathcal{G}(\mathcal{B},N)$ if σ is a P-strategy on $\mathcal{G}(\mathcal{A},M) \multimap\!\circledast \mathcal{G}(\mathcal{B},N)$, and $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A},M) \multimap\!\circledast \mathcal{G}(\mathcal{B},N)$ if σ is moreover winning. Finally, we write

$$\Sigma \Vdash \mathcal{G}(\mathcal{A},M) \multimap\!\circledast \mathcal{G}(\mathcal{B},N)$$

if there is a winning P-strategy σ on $\mathcal{G}(\mathcal{A},M) \multimap\!\circledast \mathcal{G}(\mathcal{B},N)$.

▶ Remark. Recall that if $\Omega_{\mathcal{A}}$ and $\Omega_{\mathcal{B}}$ are Borel sets, then $\mathcal{W}$ is a Borel set and by Martin's Theorem [12], either P or O has a winning strategy. Moreover, if the automata $\mathcal{A}$ and $\mathcal{B}$ are regular (in the sense of Sect. 2), then $\mathcal{W}$ is an ω-regular language. If in addition the trees M and N are regular (in the usual sense), then the game is equivalent to a finite regular game. By Büchi-Landweber Theorem, the existence of a winning strategy for a given player is decidable, and the winning player has *finite state* winning strategies (see e.g. [17]).

3.3 Characterization of the Synchronous Arrow Games

We now give a characterization of synchronous arrow games in traditional games semantics. Our characterization involve relations in slices categories $\mathbf{Set}/J$, that will give rise to a strong analogy between our fibrations $\mathsf{game}^{(\mathrm{W})}$ and $\mathsf{aut}^{(\mathrm{W})}$ and substitution (a.k.a *change-of-base*) in the codomain fibration $\mathsf{cod} : \mathbf{Set}^{\to} \to \mathbf{Set}$.

Simple Games. Recall the usual notion of *simple games* (see e.g. [1, 7]). Simple games are usually negative, but given positive games A and B, their *negative* linear arrow $A \multimap B$ can still be defined. Moreover, simple games, with linear arrows $A \multimap B$ between games A and B of the same polarity, form a category that we write $\mathbf{SGG}$. When equipped with winning conditions, winning strategies compose, giving rise to a category that we write $\mathbf{SGG}^{\mathrm{W}}$.

A P-strategy $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A},M) \multimap\!\circledast \mathcal{G}(\mathcal{B},N)$ is a morphism of $\mathbf{SGG}$ from the substituted acceptance game $\mathcal{G}(\mathcal{A},M)$ to the substituted acceptance game $\mathcal{G}(\mathcal{B},N)$. If σ is moreover winning, then it is a morphism of $\mathbf{SGG}^{\mathrm{W}}$.

The Hyland & Schalk Functor. Hyland & Schalk have presented in [9] a faithful functor, that we denote HS, from simple games to the category $\mathbf{Rel}$ of sets and relations. This functor can easily be extended to a functor $\mathrm{HS} : \mathbf{SGG}^{(\mathrm{W})} \to \mathbf{Rel}$.

Given a play $s \in \wp(A \multimap B)$ we let $s{\restriction}A \in \wp(A)$ be its projection on A and similarly for

B,[2] so that $\mathrm{HS}(s) := (s\restriction A, s\restriction B)$. Given a P-strategy $\sigma : A \multimap B$ we have $\sigma \subseteq \wp^{\mathsf{P}}(A \multimap B)$ and thus

$$\mathrm{HS}(\sigma) \quad := \quad \{\mathrm{HS}(s) \mid s \in \sigma\} \quad \subseteq \quad \wp(A) \times \wp(B)$$

We write $\wp_\Sigma(\mathcal{A}, M)$ for the plays of the substituted acceptance game $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$. Given $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A}, M) \multimap\!\circledast\, \mathcal{G}(\mathcal{B}, N)$, we thus have

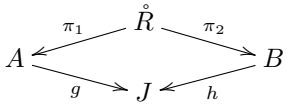
$$\mathrm{HS}(\sigma) \quad \subseteq \quad \wp_\Sigma(\mathcal{A}, M) \times \wp_\Sigma(\mathcal{B}, N)$$

Synchronous Relations. We will now see that P-strategies on a synchronous arrow game can be seen as relations in slice categories $\mathbf{Set}/J$. We call such relations *synchronous*.

Given a set J, define the category $\mathbf{Rel}(\mathbf{Set}/J)$ as follows:

Objects are indexed sets $A \xrightarrow{g} J$, written simply A when g is understood from the context.

Morphisms from $A \xrightarrow{g} J$ to $B \xrightarrow{h} J$ are given by relations $\mathring{R} : A \nrightarrow B$ such that the following commutes:



Traces. For the synchronous arrow games, synchronization is performed using the following notion of *trace*. Given $\Gamma \vdash \mathcal{A}$ and $M \in \mathbf{Tree}[\Gamma, \Sigma]$, define

$$\mathrm{tr} \quad : \quad \wp_\Sigma(\mathcal{A}, M) \quad \longrightarrow \quad (D + \Sigma)^*$$

inductively as follows

$$\mathrm{tr}(\varepsilon) \;:=\; \varepsilon \qquad \mathrm{tr}(s \to (p, a, \gamma)) \;:=\; \mathrm{tr}(s) \cdot a \qquad \mathrm{tr}(s \to (p \cdot d, q)) \;:=\; \mathrm{tr}(s) \cdot d$$

The image of tr is the set $\mathrm{Tr}_\Sigma \;:=\; (\Sigma \cdot D)^* + (\Sigma \cdot D)^* \cdot \Sigma$.

Characterization of the Synchronous Arrow. We can now characterize the synchronous arrow games.

▶ **Proposition 3.2.** *Strategies on the synchronous arrow game $\mathcal{G}(\mathcal{A}, M) \multimap\!\circledast\, \mathcal{G}(\mathcal{B}, N)$ are exactly the strategies $\sigma : \mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)$ such that*

$$\begin{array}{ccc} \mathrm{HS}(\sigma) & \longrightarrow & \wp_\Sigma(\mathcal{B}, \vec{N}) \\ \downarrow & & \downarrow{\scriptstyle \mathrm{tr}} \\ \wp_\Sigma(\mathcal{A}, \vec{M}) & \xrightarrow[\mathrm{tr}]{} & \mathrm{Tr}_\Sigma \end{array} \tag{2}$$

▶ **Proposition 3.3.** *Let $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$ and $\Sigma \vdash \mathcal{G}(\mathcal{B}, N)$. The following is a pullback in* **Set***:*

$$\begin{array}{ccc} \wp^{\mathsf{P}}_\Sigma(\mathcal{G}(\mathcal{A}, M) \multimap\!\circledast\, \mathcal{G}(\mathcal{B}, N)) & \xrightarrow{(-)\restriction \mathcal{G}(\mathcal{B}, N)} & \wp_\Sigma(\mathcal{B}, N) \\ {\scriptstyle (-)\restriction \mathcal{G}(\mathcal{A}, M)}\downarrow & & \downarrow{\scriptstyle \mathrm{tr}} \\ \wp_\Sigma(\mathcal{A}, M) & \xrightarrow[\mathrm{tr}]{} & \mathrm{Tr}_\Sigma \end{array}$$

We write $\mathrm{tr}^{\multimap\!\circledast}$ for any of two equal maps $\mathrm{tr} \circ (-)\restriction \mathcal{G}(\mathcal{A}, M)$, $\mathrm{tr} \circ (-)\restriction \mathcal{G}(\mathcal{B}, N)$.

[2] We write $\wp(A)$ for the set of plays on A, and $\wp^{\mathsf{P}}(A)$ for the set of P-plays.

3.4 Categories of Substituted Acceptance Games and Automata

We now define our categories $\mathbf{SAG}_\Sigma^{(\mathrm{W})}$ of substituted acceptance games and their full subcategories $\mathbf{Aut}_\Sigma^{(\mathrm{W})}$ of tree automata. That they indeed form categories follows from the characterization Prop. 3.2, together with the fact that $\mathbf{Rel}(\mathbf{Set}/J)$ and $\mathbf{SGG}^{(\mathrm{W})}$ are categories, and the fact that the identity strategies $\mathrm{id} : \mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)$ are synchronous.

Objects of $\mathbf{SAG}_\Sigma$ and $\mathbf{SAG}_\Sigma^{\mathrm{W}}$ are games $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$,
Morphisms of $\mathbf{SAG}_\Sigma$ are synchronous strategies $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A}, M) \multimap\!\!\circledast\, \mathcal{G}(\mathcal{B}, N)$,
Morphisms of $\mathbf{SAG}_\Sigma^{\mathrm{W}}$ are synchronous winning strategies $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, M) \multimap\!\!\circledast\, \mathcal{G}(\mathcal{B}, N)$.

Objects of $\mathbf{Aut}_\Sigma$ and $\mathbf{Aut}_\Sigma^{\mathrm{W}}$ are automata $\Sigma \vdash \mathcal{A}$,
Morphisms of $\mathbf{Aut}_\Sigma$ are synchronous strategies $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A}, \mathrm{Id}_\Sigma) \multimap\!\!\circledast\, \mathcal{G}(\mathcal{B}, \mathrm{Id}_\Sigma)$,
Morphisms of $\mathbf{Aut}_\Sigma^{\mathrm{W}}$ are synchronous winning strategies $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, \mathrm{Id}_\Sigma) \multimap\!\!\circledast\, \mathcal{G}(\mathcal{B}, \mathrm{Id}_\Sigma)$.

A Lifting Property. Among the useful consequences of Prop. 3.3, we state the following lifting property.

▶ **Proposition 3.4.** *Consider $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$ and $\Sigma \vdash \mathcal{G}(\mathcal{B}, N)$. Assume that, in $\mathbf{Rel}(\mathbf{Set}/\mathrm{Tr}_\Sigma)$ we have an isomorphism $\mathring{R} : (\wp_\Sigma(\mathcal{A}, M) \xrightarrow{\mathrm{tr}} \mathrm{Tr}_\Sigma) \mathrel{-\!\!\!\rightarrow\!\!\!\!\rightarrow}_{/\mathrm{Tr}_\Sigma} (\wp_\Sigma(\mathcal{B}, N) \xrightarrow{\mathrm{tr}} \mathrm{Tr}_\Sigma)$.*

There is a (unique, total) isomorphism $\sigma : \mathcal{G}(\mathcal{A}, M) \longrightarrow_{\mathbf{SAG}_\Sigma} \mathcal{G}(\mathcal{B}, N)$ s.t. $\mathrm{HS}(\sigma) = R$.

In general we can not ask σ to be winning, and in particular to be a morphism of $\mathbf{SAG}_\Sigma^{\mathrm{W}}$.

4 Fibrations of Acceptance Games and Automata

A tree morphism $L \in \mathbf{Tree}[\Sigma, \Gamma]$ defines a map L^* from the objects of $\mathbf{SAG}_\Gamma$ to the objects of $\mathbf{SAG}_\Sigma$: we let $L^*(\Gamma \vdash \mathcal{G}(\mathcal{A}, M)) := \Sigma \vdash \mathcal{G}(\mathcal{A}, M \circ L)$.

In this Section, we show that L^* extends to functors $L^* : \mathbf{SAG}_\Gamma^{(\mathrm{W})} \longrightarrow \mathbf{SAG}_\Sigma^{(\mathrm{W})}$ and that the operation $(-)^*$ is itself functorial and thus leads to split indexed categories $(-)^* : \mathbf{Tree}^{\mathrm{op}} \longrightarrow \mathbf{Cat}$. By applying Groethendieck completion, we obtain our split fibrations of acceptance games $\mathsf{game}^{(\mathrm{W})} : \mathbf{SAG}^{(\mathrm{W})} \longrightarrow \mathbf{Tree}$.

On the other hand, by restricting substitution to tree morphisms generated by alphabet morphisms $\beta \in \mathbf{Alph}[\Sigma, \Gamma]$, we obtain functors $\beta^* : \mathbf{Aut}_\Gamma^{(\mathrm{W})} \longrightarrow \mathbf{Aut}_\Sigma^{(\mathrm{W})}$ giving rise to split fibrations of tree automata $\mathsf{aut}^{(\mathrm{W})} : \mathbf{Aut}^{(\mathrm{W})} \longrightarrow \mathbf{Alph}$.

Our substitution functors L^* are build in strong analogy with change-of-base functors $\mathbf{Set}/\mathrm{Tr}_\Gamma \to \mathbf{Set}/\mathrm{Tr}_\Sigma$ of the codomain fibration $\mathsf{cod} : \mathbf{Set}^{\to} \to \mathbf{Set}$. We refer to [10] for basic material about fibrations.

4.1 Substitution Functors

Change-of-Base in $\mathbf{Set}^{\to}$. A morphism $L \in \mathbf{Tree}[\Sigma, \Gamma]$ induces a map $\mathrm{Tr}(L) : \mathrm{Tr}_\Sigma \longrightarrow \mathrm{Tr}_\Gamma$ inductively defined as follows (where $(-)_D$ is the obvious projection $\mathrm{Tr}_\Sigma \to D^*$):

$$\mathrm{Tr}(L)(\varepsilon) := \varepsilon \qquad \mathrm{Tr}(L)(w \cdot a) := \mathrm{Tr}(L)(w) \cdot L(w_D)(a) \qquad \mathrm{Tr}(L)(w \cdot d) := \mathrm{Tr}(L)(w) \cdot d$$

The map $\mathrm{Tr}(L)$ gives rise to the usual change-of-base functor $L^\bullet : \mathbf{Set}/\mathrm{Tr}_\Gamma \to \mathbf{Set}/\mathrm{Tr}_\Sigma$, defined using chosen pullbacks in $\mathbf{Set}$:

$$\begin{array}{ccc} L^\bullet(\wp_\Gamma(\mathcal{A}, M)) & \longrightarrow & \wp_\Gamma(\mathcal{A}, M) \\ {\scriptstyle L^\bullet(\mathrm{tr})}\downarrow & & \downarrow{\scriptstyle \mathrm{tr}} \\ \mathrm{Tr}_\Sigma & \xrightarrow{\mathrm{Tr}(L)} & \mathrm{Tr}_\Gamma \end{array}$$

Substitution on Plays. The action of the substitution L^* on plays can be described, similarly as the action of $L^\bullet$ on objects of $\mathbf{Set}/\mathrm{Tr}_\Gamma$, by a pullback property.

Consider $\Gamma \vdash \mathcal{G}(\mathcal{A}, M)$, so that $\Sigma \vdash \mathcal{G}(\mathcal{A}, M \circ L)$. A position $(p, a, \gamma_\mathcal{A})$ of the game $\Sigma \vdash \mathcal{G}(\mathcal{A}, M \circ L)$ can be mapped to the position $(p, L(p)(a), \gamma_\mathcal{A})$ of the game $\Gamma \vdash \mathcal{G}(\mathcal{A}, M)$. Moreover, since $\delta_\mathcal{A}(q_\mathcal{A}, (M \circ L)(p)(a)) = \delta_\mathcal{A}(q_\mathcal{A}, M(p)(L(p)(a)))$, we have

$$(p, q_\mathcal{A}) \to (p, a, \gamma_\mathcal{A}) \qquad \text{if and only if} \qquad (p, q_\mathcal{A}) \to (p, L(p)(a), \gamma_\mathcal{A})$$

This gives a map

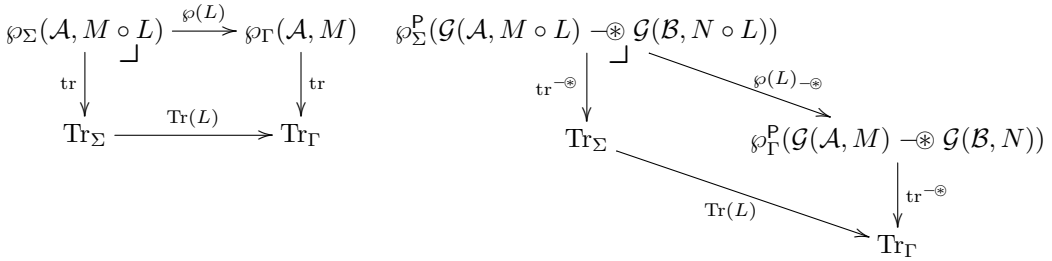
$$\wp(L) \quad : \quad \wp_\Sigma(\mathcal{A}, M \circ L) \quad \longrightarrow \quad \wp_\Gamma(\mathcal{A}, M)$$

If we are also given $\Gamma \vdash \mathcal{G}(\mathcal{B}, N)$, then we similarly obtain

$$\wp(L)_{-\circledast} \quad : \quad \wp_\Sigma(\mathcal{G}(\mathcal{A}, M \circ L) \multimap \mathcal{G}(\mathcal{B}, N \circ L)) \quad \longrightarrow \quad \wp_\Gamma((\mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N))$$

These two maps are related *via* HS as expected: $\mathrm{HS} \circ \wp(L)_{-\circledast} = (\wp(L) \times \wp(L)) \circ \mathrm{HS}$. Moreover,

▶ **Proposition 4.1.** *We have, in* **Set***:*



Substitution on Strategies. The action of L^* on strategies is defined using Prop. 4.1: Given $\Gamma \vdash \sigma : \mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)$, so that $\sigma \subseteq \wp^{\mathsf{P}}_\Gamma(\mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N))$, we define

$$L^*(\sigma) \quad := \quad \wp(L)^{-1}_{-\circledast}(\sigma) \quad \subseteq \quad \wp^{\mathsf{P}}_\Sigma(\mathcal{G}(\mathcal{A}, M \circ L) \multimap \mathcal{G}(\mathcal{B}, N \circ L))$$

▶ **Proposition 4.2.** *$L^*(\sigma)$ is a strategy. If moreover σ is winning, then $L^*(\sigma)$ is also winning.*

Functoriality of Substitution. Proposition 4.1 can be formulated by saying that the maps $\langle \mathrm{tr}, \wp(L) \rangle$ and $\langle \mathrm{tr}^{-\circledast}, \wp(L)_{-\circledast} \rangle$ are bijections, respectively:

$$\begin{array}{rcl} \wp_\Sigma(\mathcal{A}, M \circ L) & \xrightarrow{\simeq} & \mathrm{Tr}_\Sigma \times_{\mathrm{Tr}_\Sigma} \wp_\Gamma(\mathcal{A}, M) \\ \wp^{\mathsf{P}}_\Sigma(\mathcal{G}(\mathcal{A}, M \circ L) \multimap \mathcal{G}(\mathcal{B}, N \circ L)) & \xrightarrow{\simeq} & \mathrm{Tr}_\Sigma \times_{\mathrm{Tr}_\Sigma} \wp^{\mathsf{P}}_\Gamma(\mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)) \end{array}$$

These bijections are crucial to prove that

▶ **Proposition 4.3.** *L^* is a functor from $\mathbf{SAG}^{(\mathrm{W})}_\Gamma$ to $\mathbf{SAG}^{(\mathrm{W})}_\Sigma$.*

4.2 Fibrations of Acceptance Games

Consider now $L \in \mathbf{Tree}[\Sigma, \Gamma]$ and $K \in \mathbf{Tree}[\Gamma, \Delta]$. Since $\mathrm{Tr}(K \circ L) = \mathrm{Tr}(K) \circ \mathrm{Tr}(L)$ and $\wp(K \circ L)_{(-\circledast)} = \wp(K)_{(-\circledast)} \circ \wp(L)_{(-\circledast)}$ we immediately get

▶ **Proposition 4.4.** *The operations $(-)^* : \mathbf{Tree}^{\mathrm{op}} \to \mathbf{Cat}$, mapping Σ to $\mathbf{SAG}^{(\mathrm{W})}_\Sigma$, and mapping $L \in \mathbf{Tree}[\Sigma, \Gamma]$ to $L^* : \mathbf{SAG}^{(\mathrm{W})}_\Gamma \to \mathbf{SAG}^{(\mathrm{W})}_\Sigma$ are functors.*

By using Groethendieck completion (see e.g. [10, §1.10]), this gives us split fibrations of acceptance games $\mathsf{game}^{(\mathrm{W})} : \mathbf{SAG}^{(\mathrm{W})} \longrightarrow \mathbf{Tree}$ that we do not detail here by lack of space.

4.3 Fibrations of Automata

In order to obtain fibrations of automata, we restrict substitution to tree morphisms generated by alphabet morphisms $\beta \in \mathbf{Alph}[\Sigma, \Gamma]$. The crucial point is that these restricted substitutions can be internalized in automata.

Given $\Gamma \vdash \mathcal{A}$ with $\mathcal{A} = (Q, q^\imath, \delta, \Omega)$, and $\beta \in \mathbf{Alph}[\Sigma, \Gamma]$, define the automaton $\Sigma \vdash \mathcal{A}[\beta]$ as $\mathcal{A}[\beta] := (Q, q^\imath, \delta_\beta, \Omega)$ where $\delta_\beta(q, a) := \delta(q, \beta(a))$.

▶ **Proposition 4.5.** $\Sigma \vdash \mathcal{G}(\mathcal{A}[\beta], \mathrm{Id}_\Sigma) \;=\; \Sigma \vdash \mathcal{G}(\mathcal{A}, \beta)$.

It is easy to see that $(-)^*$ restricts to a functor from $\mathbf{Alph}^{\mathrm{op}}$ to $\mathbf{Cat}$, so that we get fibrations

$$\mathsf{aut}^{(\mathrm{W})} \;:\; \mathbf{Aut}^{(\mathrm{W})} \longrightarrow \mathbf{Alph} \qquad \text{with, in } \mathbf{Cat}\text{:} \quad \begin{array}{ccc} \mathbf{Aut}^{(\mathrm{W})} & \longrightarrow & \mathbf{SAG}^{(\mathrm{W})} \\ {\scriptstyle \mathsf{aut}^{(\mathrm{W})}}\downarrow & \lrcorner & \downarrow{\scriptstyle \mathsf{game}^{(\mathrm{W})}} \\ \mathbf{Alph} & \longrightarrow & \mathbf{Tree} \end{array}$$

5 Symmetric Monoidal Structure

We now consider a synchronous product of automata. When working on *complete* automata (to be defined in Sect. 5.1 below), it gives rise to split symmetric monoidal fibrations, in the sense of [16].

According to [16, Thm. 12.7], split symmetric monoidal fibrations can equivalently be obtained from split symmetric monoidal indexed categories. In our context, this means that the functors $(-)^*$ extend to

$$(-)^* \;:\; \mathbf{Tree}^{\mathrm{op}} \longrightarrow \mathbf{SymMonCat} \qquad (-)^* \;:\; \mathbf{Alph}^{\mathrm{op}} \longrightarrow \mathbf{SymMonCat}$$

where **SymMonCat** is the category of symmetric monoidal categories and strong monoidal functors. Hence, we equip our categories of (complete) acceptance games and automata with a symmetric monoidal structure. Substitution turns out to be *strict* symmetric monoidal.

We refer to [13] for background on symmetric monoidal categories.

5.1 Complete Tree Automata

An automaton $\mathcal{A}$ is *complete* if for every $(q, a) \in Q \times \Sigma$, the set $\delta(q, a)$ is not empty and moreover for every $\gamma \in \delta(q, a)$ and every $d \in D$, we have $(q', d) \in \gamma$ for some $q' \in Q$.

Given an automaton $\mathcal{A} = (Q, q^\imath, \delta, \Omega)$ its *completion* is the automaton $\widehat{\mathcal{A}} := (\widehat{Q}, q^\imath, \widehat{\delta}, \widehat{\Omega})$ with states $\widehat{Q} := Q + \{\mathsf{true}, \mathsf{false}\}$, with acceptance condition $\widehat{\Omega} := \Omega + Q^* \cdot \mathsf{true} \cdot \widehat{Q}^\omega$, and with transition function $\widehat{\delta}$ defined as

$$\begin{array}{lll} \widehat{\delta}(\mathsf{true}, q) := \{\{(\mathsf{true}, d) \mid d \in D\}\} & & \widehat{\delta}(\mathsf{false}, q) := \{\{(\mathsf{false}, d) \mid d \in D\}\} \\ \widehat{\delta}(q, a) := \{\{(\mathsf{false}, d) \mid d \in D\}\} & & \text{if } q \in Q \text{ and } \delta(q, a) = \emptyset \\ \widehat{\delta}(q, a) := \{\widehat{\gamma} \mid \gamma \in \delta(q, a)\} & & \text{otherwise} \end{array}$$

where, given $\gamma \in \delta(q, a)$, we let $\widehat{\gamma} := \gamma \cup \{(\mathsf{true}, d) \mid \text{there is no } q \in Q \text{ s.t. } (q, d) \in \gamma\}$.

▶ **Proposition 5.1.** $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\widehat{\mathcal{A}})$.

Restricting to complete automata gives rise to full subcategories $\widehat{\mathbf{SAG}}^{(\mathrm{W})}_\Sigma$ and $\widehat{\mathbf{Aut}}^{(\mathrm{W})}_\Sigma$ of resp. $\mathbf{SAG}^{(\mathrm{W})}_\Sigma$ and $\mathbf{Aut}^{(\mathrm{W})}_\Sigma$, and thus induces fibrations

$$\widehat{\mathsf{game}} \;:\; \widehat{\mathbf{SAG}}^{(\mathrm{W})} \longrightarrow \mathbf{Tree} \qquad \widehat{\mathsf{aut}} \;:\; \widehat{\mathbf{Aut}}^{(\mathrm{W})} \longrightarrow \mathbf{Alph}$$

5.2 The Synchronous Product

Assume given complete automata $\Sigma \vdash \mathcal{A}$ and $\Sigma \vdash \mathcal{B}$. Define $\Sigma \vdash \mathcal{A} \circledast \mathcal{B}$ as

$$\mathcal{A} \circledast \mathcal{B} \quad := \quad (Q_{\mathcal{A}} \times Q_{\mathcal{B}}, (q^\iota_{\mathcal{A}}, q^\iota_{\mathcal{B}}), \delta_{\mathcal{A}\circledast\mathcal{B}}, \Omega_{\mathcal{A}\circledast\mathcal{B}})$$

where $(q^n_{\mathcal{A}}, q^n_{\mathcal{B}})_{n\in\mathbb{N}} \in \Omega_{\mathcal{A}\circledast\mathcal{B}}$ iff $((q^n_{\mathcal{A}})_{n\in\mathbb{N}} \in \Omega_{\mathcal{A}}$ and $(q^n_{\mathcal{B}})_{n\in\mathbb{N}} \in \Omega_{\mathcal{B}})$, and where we let $\delta_{\mathcal{A}\circledast\mathcal{B}}((q_{\mathcal{A}}, q_{\mathcal{B}}), a)$ be the set of all the $\gamma_{\mathcal{A}} \circledast \gamma_{\mathcal{B}}$ for $\gamma_{\mathcal{A}} \in \delta_{\mathcal{A}}(q_{\mathcal{A}}, a)$ and $\gamma_{\mathcal{B}} \in \delta_{\mathcal{B}}(q_{\mathcal{B}}, a)$, with $\gamma_{\mathcal{A}} \circledast \gamma_{\mathcal{B}} := \{((q'_{\mathcal{A}}, q'_{\mathcal{B}}), d) \mid d \in D$ and $(q'_{\mathcal{A}}, d) \in \gamma_{\mathcal{A}}$ and $(q'_{\mathcal{B}}, d) \in \gamma_{\mathcal{B}}\}$.

Note that since $\mathcal{A}$ and $\mathcal{B}$ are complete, each $\gamma_{\mathcal{A}\circledast\mathcal{B}} \in \delta_{\mathcal{A}\circledast\mathcal{B}}((q_{\mathcal{A}}, q_{\mathcal{B}}), a)$ uniquely decomposes as $\gamma_{\mathcal{A}\circledast\mathcal{B}} = \gamma_{\mathcal{A}} \circledast \gamma_{\mathcal{B}}$.

Action on Plays. The unique decomposition property of $\gamma_{\mathcal{A}\circledast\mathcal{B}}$ allows to define projections

$$\begin{array}{rcccc} \varpi_i & : & \wp_\Sigma(\mathcal{A}_1 \circledast \mathcal{A}_2, M) & \longrightarrow & \wp_\Sigma(\mathcal{A}_i, M) \\ \varpi_i^{-\circledast} & : & \wp_\Sigma\left(\mathcal{G}(\mathcal{A}_1 \circledast \mathcal{B}_1, M) \multimap \mathcal{G}(\mathcal{A}_2 \circledast \mathcal{B}_2, N)\right) & \longrightarrow & \wp_\Sigma\left(\mathcal{G}(\mathcal{A}_i, M) \multimap \mathcal{G}(\mathcal{B}_i, N)\right) \end{array}$$

We write $\mathrm{SP} := \langle \varpi_1, \varpi_2 \rangle$ and $\mathrm{SP}_{-\circledast} := \langle \varpi_1^{-\circledast}, \varpi_2^{-\circledast} \rangle$.

▶ **Proposition 5.2.** *We have, in* **Set***:*

$$\begin{array}{ccc} \wp_\Sigma(\mathcal{A} \circledast \mathcal{B}, M) & \xrightarrow{\varpi_2} & \wp_\Sigma(\mathcal{B}, M) \\ {\scriptstyle \varpi_1}\downarrow & & \downarrow{\scriptstyle \mathrm{tr}} \\ \wp_\Sigma(\mathcal{A}, M) & \xrightarrow{\mathrm{tr}} & \mathrm{Tr}_\Sigma \end{array} \qquad \begin{array}{ccc} \wp^{\mathsf{P}}_\Sigma(\mathcal{G}(\mathcal{A} \circledast \mathcal{B}, M) \multimap \mathcal{G}(\mathcal{C} \circledast D, N)) & & \\ {\scriptstyle \varpi_1^{-\circledast}}\downarrow & \searrow^{\varpi_2^{-\circledast}} & \\ \wp^{\mathsf{P}}_\Sigma(\mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{C}, N)) & & \wp^{\mathsf{P}}_\Sigma(\mathcal{G}(\mathcal{B}, M) \multimap \mathcal{G}(\mathcal{D}, N)) \\ & & \downarrow{\scriptstyle \mathrm{tr}} \\ & \xrightarrow[\mathrm{tr}]{} & \mathrm{Tr}_\Sigma \end{array}$$

Action on Synchronous Games. The action of $\circledast$ on the objects of $\widehat{\mathbf{SAG}}^{(\mathrm{W})}_\Sigma$ is given by

$$(\Sigma \vdash \mathcal{G}(\mathcal{A}, M)) \circledast (\Sigma \vdash \mathcal{G}(\mathcal{B}, N)) \quad := \quad \Sigma \vdash \mathcal{G}(\mathcal{A}[\pi] \circledast \mathcal{B}[\pi'], \langle M, N \rangle)$$

where π and π' are suitable projections. For morphisms, let $\Sigma \vdash \sigma : \mathcal{G}(\mathcal{A}_0, M_0) \multimap \mathcal{G}(\mathcal{A}_1, M_1)$ and $\Sigma \vdash \tau : \mathcal{G}(\mathcal{B}_0, N_0) \multimap \mathcal{G}(\mathcal{B}_1, N_1)$. Then since $\Sigma \vdash \mathcal{G}(\mathcal{A}_i[\pi_i], \langle M_i, N_i \rangle) = \Sigma \vdash \mathcal{G}(\mathcal{A}_i, M_i)$ and $\Sigma \vdash \mathcal{G}(\mathcal{B}_i[\pi'_i], \langle M_i, N_i \rangle) = \Sigma \vdash \mathcal{G}(\mathcal{B}_i, N_i)$, thanks to Prop. 5.2 we can simply let $\sigma \circledast \tau := \mathrm{SP}^{-1}_{-\circledast}(\sigma, \tau)$.

▶ **Proposition 5.3.** *The product* $\_ \circledast \_$ *gives functors* $\widehat{\mathbf{SAG}}^{(\mathrm{W})}_\Sigma \times \widehat{\mathbf{SAG}}^{(\mathrm{W})}_\Sigma \longrightarrow \widehat{\mathbf{SAG}}^{(\mathrm{W})}_\Sigma$.

5.3 Symmetric Monoidal Structure

Thanks to Prop. 5.2 and Prop. 3.4 the symmetric monoidal structure of $\circledast$ in $\widehat{\mathbf{SAG}}^{(\mathrm{W})}_\Sigma$ can be directly obtained from the symmetric monoidal structure of the tensorial product of $\mathbf{Rel}(\mathbf{Set}/\mathrm{Tr}_\Sigma)$.

Symmetric Monoidal Structure in $\mathbf{Rel}(\mathbf{Set}/J)$. We define a product $\otimes$ in $\mathbf{Rel}(\mathbf{Set}/J)$:

On Objects: for (A, g) and (B, h) objects in $\mathbf{Rel}(\mathbf{Set}/J)$ the product $A \otimes B$ is $A \times_J B$ with the corresponding map, that is

$$A \otimes B \quad := \quad \{(a, b) \in A \times B \mid g(a) = h(b)\} \quad \xrightarrow{g\circ\pi_1 = h\circ\pi_2} \quad J$$

On Morphisms: given $R \in \mathbf{Rel}(\mathbf{Set}/J)[A, C]$ and $P \in \mathbf{Rel}(\mathbf{Set}/J)[B, D]$, we define $R \otimes P \in \mathbf{Rel}(\mathbf{Set}/J)[A \otimes B, C \otimes D]$ as

$$R \otimes P \quad := \quad \{((a,b),(c,d)) \in (A \otimes B) \times (C \otimes D) \mid (a,c) \in R \text{ and } (b,d) \in P\}$$

For the unit, we *choose* some $\mathbf{I} = (\jmath : I \xrightarrow{\simeq} J)$. Note that $\jmath$ is required to be a bijection. The natural isomorphisms are given by:

$$\begin{array}{rcl} \mathring{\alpha}_{A,B,C} & := & \{(((a,b),c)\ ,\ (a,(b,c))) \mid g_A(a) = g_B(b) = g_C(c)\} \\ \mathring{\lambda}_A & := & \{((e,a)\ ,\ a) \mid \jmath(e) = g_A(a)\} \\ \mathring{\rho}_A & := & \{((a,e)\ ,\ a) \mid g_A(a) = \jmath(e)\} \\ \mathring{\gamma}_{A,B} & := & \{((a,b)\ ,\ (b,a)) \mid g_A(a) = g_B(b)\} \end{array}$$

▶ **Proposition 5.4.** *The category* $\mathbf{Rel}(\mathbf{Set}/J)$*, equipped with the above data, is symmetric monoidal.*

Unit Automata. The requirement that the monoidal unit $\jmath : I \to J$ of $\mathbf{Rel}(\mathbf{Set}/J)$ should be a bijection leads us to the following unit automata. We let $\mathcal{I} := (Q_\mathcal{I}, q_\mathcal{I}, \delta_\mathcal{I}, \Omega_\mathcal{I})$ where $Q_\mathcal{I} := \mathbf{1}$, $q_\mathcal{I} := \bullet$, $\Omega_\mathcal{I} = Q_\mathcal{I}^\omega$ and $\delta_\mathcal{I}(q_\mathcal{I}, a) := \{\{(q_\mathcal{I}, d) \mid d \in D\}\}$.

Note that since $\delta_\mathcal{I}$ is constant, we have $\Sigma \vdash \mathcal{G}(\mathcal{I}, M) = \Sigma \vdash \mathcal{G}(\mathcal{I}, \mathrm{Id})$. Moreover,

▶ **Proposition 5.5.** *Given* $M \in \mathbf{Tree}[\Sigma, \Gamma]$*, we have, in* $\mathbf{Set}$*, a bijection*

$$\mathrm{tr} \quad : \quad \wp_\Sigma(\mathcal{I}, M) \quad \xrightarrow{\simeq} \quad \mathrm{Tr}_\Sigma$$

Symmetric Monoidal Structure. Using Prop. 3.4, the structure isos of $\mathbf{Rel}(\mathbf{Set}/\mathrm{Tr}_\Sigma)$ can be lifted to $\widehat{\mathbf{SAG}}_\Sigma^{(\mathrm{W})}$ (winning is trivial). Moreover, the required equations (naturality and coherence) follows from Prop. 3.3, Prop 5.2, and the fact that $((\mathrm{SP} \times \mathrm{SP}) \circ \mathrm{HS})(\sigma \circledast \tau) = \mathrm{HS}(\sigma) \otimes \mathrm{HS}(\tau)$ (where composition on the left is in $\mathbf{Set}$, and the expression denotes the actions of the resulting function on the set of plays $(\sigma \circledast \tau)$).

▶ **Proposition 5.6.** *The categories* $\widehat{\mathbf{SAG}}_\Sigma^{(\mathrm{W})}$ *and* $\widehat{\mathbf{Aut}}_\Sigma^{(\mathrm{W})}$ *equipped with the above data, are symmetric monoidal.*

5.4 Symmetric Monoidal Fibrations

In order to obtain symmetric monoidal fibrations, by [16, Thm. 12.7], it remains to check that substitution is strong monoidal. It is actually *strict* monoidal: it directly commutes with $\circledast$ and preserves the unit, as well as all the structure maps.

▶ **Proposition 5.7.**

(i) *Given* $L \in \mathbf{Tree}[\Sigma, \Gamma]$*, the functors* $L^* : \widehat{\mathbf{SAG}}_\Gamma^{(\mathrm{W})} \to \widehat{\mathbf{SAG}}_\Sigma^{(\mathrm{W})}$ *are strict monoidal.*

(ii) *Given* $\beta \in \mathbf{Alph}[\Sigma, \Gamma]$*, the functors* $\beta^* : \widehat{\mathbf{Aut}}_\Gamma^{(\mathrm{W})} \to \widehat{\mathbf{Aut}}_\Sigma^{(\mathrm{W})}$ *are strict monoidal.*

6 Correctness w.r.t. Language Operations

This Section gathers several properties stating the correctness of our constructions w.r.t. operations on recognized languages. We begin in Sect. 6.1 by properties on the symmetric monoidal structure, the most important one being that the synchronous arrow is *correct*, in the sense that $\Sigma \vdash \mathcal{A} \multimap\!\!\circledast\, \mathcal{B}$ implies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Then, in Sect. 6.2, we discuss complementation of automata, and its relation with the synchronous arrow.

6.1 Correctness of the Symmetric Monoidal Structure

We begin by a formal correspondence between acceptance games and synchronous games of a specific form. This allows to show that the synchronous arrow is *correct*, in the sense that $\Sigma \vdash \mathcal{A} \multimap \mathcal{B}$ implies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. We then briefly discuss the correctness of the synchronous product w.r.t. language intersection.

▶ **Proposition 6.1.** *Given $\Sigma \vdash \mathcal{A}$ and $t \in \mathbf{Tree}[\Sigma]$, there is a bijection:*

$$\{\sigma \mid \mathbf{1} \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, \dot{t})\} \quad \simeq \quad \{\theta \mid \mathbf{1} \vdash \theta \Vdash \mathcal{G}(\mathcal{I}, \mathrm{Id}_{\mathbf{1}}) \multimap \mathcal{G}(\mathcal{A}, \dot{t})\}$$

▶ Remark. The above correspondence is only possible for acceptance games over $\mathbf{1}$:

- In $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, M)$, σ is a positive P-strategy, hence chooses the input characters in Σ.
- In $\Sigma \vdash \theta \Vdash \mathcal{G}(\mathcal{I}_\Sigma, \mathrm{Id}_\Sigma) \multimap \mathcal{G}(\mathcal{A}, M)$, the strategy θ is a negative. It plays positively in $\Sigma \vdash \mathcal{G}(\mathcal{A}, M)$, but must follow the input characters chosen by O in $\Sigma \vdash \mathcal{G}(\mathcal{I}_\Sigma, \mathrm{Id}_\Sigma)$.

We now check that the arrow $\mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)$ is correct w.r.t. language inclusion:

▶ **Proposition 6.2** (Correctness of the Arrow). *Assume given $\Sigma \vdash \sigma \Vdash \mathcal{G}(\mathcal{A}, M) \multimap \mathcal{G}(\mathcal{B}, N)$.*

(i) *For all $t \in \mathbf{Tree}[\Sigma]$, we have $\dot{t}^*(\sigma) \Vdash \mathcal{G}(\mathcal{A}, M \circ \dot{t}) \multimap \mathcal{G}(\mathcal{B}, N \circ \dot{t})$.*
(ii) *If $\mathbf{1} \Vdash \mathcal{G}(\mathcal{A}, M \circ \dot{t})$ then $\mathbf{1} \Vdash \mathcal{G}(\mathcal{B}, N \circ \dot{t})$.*
(iii) *For all tree $t \in \mathbf{Tree}[\Sigma]$, if $M(t) \in \mathcal{L}(\mathcal{A})$ then $N(t) \in \mathcal{L}(\mathcal{B})$.*

The converse property will be discussed in Sect. 7. We finally check that the synchronous product is correct.

▶ **Proposition 6.3.** $\mathcal{L}(\widehat{\mathcal{A}} \circledast \widehat{\mathcal{B}}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$.

6.2 Complementation and Falsity

Complementation. Given an automaton $\mathcal{A} = (Q, q^\imath, \delta, \Omega)$, following [18], we let its complement be $\sim\mathcal{A} := (Q, q^\imath, \delta_{\sim\mathcal{A}}, \Omega_{\sim\mathcal{A}})$, where $\Omega_{\sim\mathcal{A}} := Q^\omega \setminus \Omega$ and

$$\delta_{\sim\mathcal{A}}(q, a) \quad := \quad \{\gamma_\sim \in \mathcal{P}(Q \times D) \mid \forall \gamma \in \delta(q, a),\ \gamma_\sim \cap \gamma \neq \emptyset\}$$

The idea is that P on $\sim\mathcal{A}$ simulates O on $\mathcal{A}$, so that the correctness of $\sim\mathcal{A}$ relies on determinacy of acceptance games. In particular, thanks to Borel determinacy [12], we have:

▶ **Proposition 6.4** ([18]). *Given $\mathcal{A}$ with $\Omega_\mathcal{A}$ a Borel set, we have $\mathcal{L}(\sim\mathcal{A}) = \mathbf{Tree}[\Sigma] \setminus \mathcal{L}(\mathcal{A})$.*

Note that if $\mathcal{A}$ is complete, then $\sim\mathcal{A}$ is not necessarily complete, but $\delta_{\sim\mathcal{A}}$ is always not empty and so are the γ's in its image.

The Falsity Automaton $\bot$. We let $\bot := (Q_\bot, q_\bot, \delta_\bot, \Omega_\bot)$ where $Q_\bot := \mathbf{1}$, $q_\bot := \bullet$, $\Omega_\bot = \emptyset$ and $\delta_\bot(q_\bot, a) := \{\{(q_\bot, d)\} \mid d \in D\}$. Note that $\mathcal{I} = \sim\bot$. In particular, it is actually P who guides the evaluation of $\bot$, by choosing the tree directions.

▶ **Proposition 6.5.** *Let $\mathcal{A}$ and $\mathcal{B}$ be complete. Then $\Sigma \Vdash \mathcal{A} \circledast \mathcal{B} \multimap \widehat{\bot}$ iff $\Sigma \Vdash \mathcal{A} \multimap \widehat{\sim\mathcal{B}}$.*

▶ **Corollary 6.6.** *Let $\mathcal{A}$ be a complete automaton on Σ. Then $\mathbf{1} \Vdash \widehat{\sim\mathcal{A}}$ iff $\mathbf{1} \Vdash \mathcal{A} \multimap \widehat{\bot}$.*

7 Projection and Fibred Simple Coproducts

We now check that automata can be equipped with existential quantifications in the fibered sense. Namely, given a projection $\pi \in \mathbf{Alph}[\Sigma \times \Gamma, \Sigma]$, the induced weakening functor $\pi^* : \widehat{\mathbf{Aut}}^{(\mathrm{W})}_{\Sigma} \to \widehat{\mathbf{Aut}}^{(\mathrm{W})}_{\Sigma\times\Gamma}$ has a left-adjoint $\amalg_{\Sigma,\Gamma}$, and moreover this structure is preserved by substitution, in the sense of the Beck-Chevalley condition (see e.g. [10]). This will lead to a (weak) completeness property of the synchronous arrow on *non-deterministic* automata, to be discussed below.

Recall from [11, Thm. IV.1.2.(ii)] than an adjunction $\amalg_{\Sigma,\Gamma} \dashv \pi^*$, with π^* a functor, is completely determined by the following data: To each object $\Sigma\times\Gamma \vdash \mathcal{A}$, an object $\Sigma \vdash \amalg_{\Sigma,\Gamma}\mathcal{A}$, and a map $\eta_{\mathcal{A}} : \Sigma \times \Gamma \vdash \mathcal{A} \longrightarrow \Sigma \times \Gamma \vdash (\amalg_{\Sigma,\Gamma}\mathcal{A})[\pi]$ satisfying the following universal lifting property:

$$\begin{array}{l}\text{For every}\\ \sigma \quad : \quad \Sigma \times \Gamma \vdash \mathcal{A} \quad\longrightarrow\quad \Sigma \times \Gamma \vdash \mathcal{B}[\pi]\\ \text{there is a unique}\\ \qquad \tau \quad : \quad \Sigma \vdash \amalg_{\Sigma,\Gamma}\mathcal{A} \quad\longrightarrow\quad \Sigma \vdash \mathcal{B}\end{array} \qquad \text{s.t.} \qquad \begin{array}{ccc}\mathcal{A} & \xrightarrow{\eta_{\mathcal{A}}} & (\amalg_{\Sigma,\Gamma}\mathcal{A})[\pi]\\ & {\scriptstyle\sigma}\searrow & \downarrow{\scriptstyle \pi^*(\tau)}\\ & & \mathcal{B}[\pi]\end{array} \qquad (3)$$

In our context, the Beck-Chevalley condition amounts to the equalities

$$\Delta \vdash (\amalg_{\Sigma,\Gamma}\mathcal{A})[\beta] \;=\; \Delta \vdash \amalg_{\Delta,\Gamma}(\mathcal{A}[\beta \times \mathrm{Id}_\Gamma]) \qquad \eta_{\mathcal{A}[\beta\times\mathrm{Id}_\Gamma]} \;=\; (\beta \times \mathrm{Id}_\Gamma)^*(\eta_{\mathcal{A}}) \qquad (4)$$

It turns out that the usual projection operation on automata (see e.g. [18]) is not functorial. Surprisingly, this is independent from whether automata are non-deterministic or not[3]. We devise a *lifted* projection operation, which indeed leads to a fibered existential quantification, and which is correct, on non-deterministic automata, w.r.t. the recognized languages.

The Lifted Projection. Consider $\Sigma \times \Gamma \vdash \mathcal{A}$ with $\mathcal{A} = (Q, q^\imath, \delta, \Omega)$. Define $\Sigma \vdash \amalg_{\Sigma,\Gamma}\mathcal{A}$ as $\amalg_{\Sigma,\Gamma}\mathcal{A} \;:=\; (Q \times \Gamma + \{q^\imath\}, q^\imath, \delta_{\amalg\mathcal{A}}, \Omega_{\amalg\mathcal{A}})$ where

$$\begin{array}{rcl}\delta_{\amalg\mathcal{A}}(q^\imath, a) & := & \bigcup_{b\in\Gamma}\{\gamma^{+b} \mid \gamma \in \delta(q^\imath, (a, b))\}\\ \delta_{\amalg\mathcal{A}}((q, \_), a) & := & \bigcup_{b\in\Gamma}\{\gamma^{+b} \mid \gamma \in \delta(q, (a, b))\}\end{array}$$

and, given $\gamma \in \mathcal{P}(Q \times D)$ and $b \in \Gamma$, we let $\gamma^{+b} := \{((q^{+b}, d) \mid (q, d) \in \gamma\}$ with $q^{+b} := (q, b)$.

For the acceptance condition, we let $q^\imath \cdot (q_0, b_0) \cdot \ldots \cdot (q_n, b_n) \cdot \ldots$ in $\Omega_{\amalg\mathcal{A}}$ iff $q^\imath \cdot q_0 \cdot \ldots \cdot q_n \cdot \ldots \;\in\; \Omega$.

Action on Plays of The Lifted Projection. The action on plays of $\amalg_{\Sigma,\Gamma}$ is characterized by the map $\wp(\amalg) : \wp_{\Sigma\times\Gamma}(\mathcal{A}) \longrightarrow \wp_\Sigma(\amalg_{\Sigma,\Gamma}\mathcal{A})$ inductively defined as $\wp(\amalg)(\varepsilon, q^\imath) := (\varepsilon, q^\imath)$ and

$$\begin{array}{rcl}\wp(\amalg)((\varepsilon, q^\imath) \to^* (p, q) \to (p, (a, b), \gamma)) & := & \wp(\amalg)((\varepsilon, q^\imath) \to^* (p, q)) \to (p, a, \gamma^{+b})\\ \wp(\amalg)((\varepsilon, q^\imath) \to^* (p, (a, b), \gamma) \to (p.d, q)) & := & \wp(\amalg)((\varepsilon, q^\imath) \to^* (p, (a, b), \gamma)) \to (p.d, q^{+b})\end{array}$$

▶ **Proposition 7.1.** *If $\mathcal{A}$ is a complete automaton, then $\wp(\amalg)$ is a bijection.*

[3] It is well-known that the projection operation is correct w.r.t. the recognized languages only on *non-deterministic automata.*

The Unit Maps $\eta_{(-)}$. Consider the injection $\iota_{\Sigma,\Gamma} : \wp_\Sigma(\mathrm{II}_{\Sigma,\Gamma}\mathcal{A}) \longrightarrow \wp_{\Sigma\times\Gamma}((\mathrm{II}_{\Sigma,\Gamma}\mathcal{A})[\pi])$ inductively defined as $\iota_{\Sigma,\Gamma}((\varepsilon, q^i_{\mathcal{A}})) := (\varepsilon, q^i_{\mathcal{A}})$ and $\iota_{\Sigma,\Gamma}(s \to (p, q^{+b})) := \iota_{\Sigma,\Gamma}(s) \to (p, q^{+b})$ and $\iota_{\Sigma,\Gamma}(s \to (p, a, \gamma^{+b})) := \iota_{\Sigma,\Gamma}(s) \to (p, (a, b), \gamma^{+b})$.

If $\Sigma \times \Gamma \vdash \mathcal{A}$ is complete, we let the unit $\eta_{\mathcal{A}}$ be the unique strategy of $\widehat{\mathbf{SAG}}^{\mathrm{W}}_{\Sigma\times\Gamma}$ such that $\mathrm{HS}(\eta_{\mathcal{A}}) = \{(t, \iota_{\Sigma,\Gamma} \circ \wp(\mathrm{II})(t)) \mid t \in \wp_{\Sigma\times\Gamma}(\mathcal{A})\}$. We do not detail the B.-C. condition (4).

The Unique Lifting Property (3). Consider some $\Sigma \times \Gamma \vdash \sigma : \mathcal{A} \multimap \mathcal{B}[\pi]$ with $\mathcal{A}$ complete. We let τ be the unique strategy such that $\mathrm{HS}(\tau) = \{(\wp(\mathrm{II})(s), \wp(\pi)(t)) \mid (s, t) \in \mathrm{HS}(\sigma)\}$. It is easy to see that τ is winning whenever σ is winning. Moreover

▶ **Lemma 7.2.** *$\sigma = \pi^*(\tau) \circ \eta_{\mathcal{A}}$.*

For the unicity part of the lifting property of $\eta_{\mathcal{A}}$, it is sufficient to check:

▶ **Lemma 7.3.** *If $\pi^*(\theta) \circ \eta_{\mathcal{A}} = \pi^*(\theta') \circ \eta_{\mathcal{A}}$ then $\theta = \theta'$.*

Non-Deterministic Tree Automata. An automaton $\mathcal{A}$ is *non-deterministic* if for every γ in the image of δ and every direction $d \in D$, there is at most one state q such that $(q, d) \in \gamma$.

▶ Remark. If $\mathcal{A}$ and $\mathcal{B}$ are non-deterministic, then so are $\mathcal{A} \circledast \mathcal{B}$ and $\mathrm{II}(\mathcal{A})$.

▶ **Proposition 7.4** ([4, 15, 18])**.** *For each regular automaton $\Sigma \vdash \mathcal{A}$ there is a complete non-deterministic automaton $\Sigma \vdash \mathrm{ND}(\mathcal{A})$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathrm{ND}(\mathcal{A}))$.*

▶ **Proposition 7.5.** *If $\Sigma \times \Gamma \vdash \mathcal{A}$ is non-deterministic and complete, then $\mathcal{L}(\mathrm{II}_{\Sigma,\Gamma}\mathcal{A}) = \pi_{\Sigma,\Gamma}(\mathcal{L}(\mathcal{A}))$ where $\pi_{\Sigma,\Gamma} \in \mathbf{Alph}[\Sigma \times \Gamma, \Sigma]$ is the first projection.*

▶ **Proposition 7.6.** *Consider complete regular automata $\Sigma \vdash \mathcal{A}$ and $\Sigma \vdash \mathcal{B}$.*
If $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ then $\Sigma \Vdash \mathrm{ND}(\mathcal{A}) \multimap \widehat{\sim\mathcal{C}}$ with $\mathcal{C} := \mathrm{ND}(\sim\mathcal{B})$.

8 Conclusion

We presented monoidal fibrations of tree automata and acceptance games, in which the fibre categories are based on a synchronous restriction of linear simple games.

For technical simplicity, we did not yet consider monoidal closure, but strongly expect that it holds. One of the main question is whether suitable restrictions of these categories are Cartesian closed, so as to interpret proofs from intuitionistic variants of MSO. Among other questions are the status of non-determinization (*i.e.* whether it can be made functorial, or even co-monadic), as well as relation with the Dialectica interpretation (in the vein of e.g. [8]). Our result of weak completeness (Prop. 7.6) suggests strong connections with the notion of *guidable* non-deterministic automata of [2]. On a similar vein, connections with *game automata* [3, 5] might be relevant to investigate.

Acknowledgments. This work benefited from numerous discussions with Pierre Clairambault and Thomas Colcombet.

References

1 S. Abramsky. Semantics of Interaction. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14 of *Publications of the Newton Institute*, page 1. Cambridge University Press, 1997.

2 T. Colcombet and C. Löding. The Non-deterministic Mostowski Hierarchy and Distance-Parity Automata. In *ICALP 2008*, volume 5126 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 2008.

3 J. Duparc, F. Facchini, and F. Murlak. Definable Operations On Weakly Recognizable Sets of Trees. In *FSTTCS*, volume 13 of *LIPIcs*, pages 363–374. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

4 E. A. Emerson and C. S. Jutla. Tree Automata, Mu-Calculus and Determinacy (Extended Abstract). In *FOCS*, pages 368–377. IEEE Computer Society, 1991.

5 A. Facchini, F. Murlak, and M. Skrzypczak. Rabin-Mostowski Index Problem: A Step beyond Deterministic Automata. In *LICS*, pages 499–508. IEEE Computer Society, 2013.

6 E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

7 J. M. E. Hyland. Game Semantics. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14 of *Publications of the Newton Institute*, page 131. Cambridge University Press, 1997.

8 J. M. E. Hyland. Proof theory in the abstract. *Ann. Pure Appl. Logic*, 114(1-3):43–78, 2002.

9 J. M. E. Hyland and A. Schalk. Abstract Games for Linear Logic. *Electr. Notes Theor. Comput. Sci.*, 29:127–150, 1999.

10 B. Jacobs. *Categorical Logic and Type Theory*. Studies in logic and the foundations of mathematics. Elsevier, 2001.

11 S. Mac Lane. *Categories for the Working Mathematician*. Springer, 2 edition, 1998.

12 D. A. Martin. Borel Determinacy. *The Annals of Mathematics, Second Series*, 102(2):363–371, 1975.

13 P.-A. Melliès. Categorical semantics of linear logic. In *Interactive models of computation and program behaviour*, volume 27 of *Panoramas et Synthèses*. SMF, 2009.

14 D. E. Muller and P. E. Schupp. Alternating Automata on Infinite Trees. *Theor. Comput. Sci.*, 54:267–276, 1987.

15 D. E. Muller and P. E Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theor. Comput. Sci.*, 141(1&2):69–107, 1995.

16 M. Shulman. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories*, 20(18):650–738, 2008.

17 W. Thomas. Languages, Automata, and Logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 389–455. Springer, 1997.

18 I. Walukiewicz. Monadic second-order logic on tree-like structures. *Theor. Comput. Sci.*, 275(1-2):311–346, 2002.

Multi-Focusing on Extensional Rewriting with Sums

Gabriel Scherer

Gallium, INRIA, France

Abstract

We propose a logical justification for the rewriting-based equivalence procedure for simply-typed lambda-terms with sums of Lindley [8]. It relies on maximally multi-focused proofs, a notion of canonical derivations introduced for linear logic. Lindley's rewriting closely corresponds to *preemptive rewriting* [5], a technical device used in the meta-theory of maximal multi-focus.

1998 ACM Subject Classification F.4.1 Lambda calculus and related systems

Keywords and phrases Maximal multi-focusing, extensional sums, rewriting, natural deduction

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.317

1 Introduction

Deciding observational equality of pure typed lambda-terms in presence of sum types is a difficult problem. After several solutions based on complex syntactic [6] or semantic [1, 2] techniques, Sam Lindley presented a surprisingly simple rewriting solution [8]. While the underlying intuition (extrude contexts to move pattern-matchings as high as possible in the term) makes sense, the algorithm is still mysterious in many aspects: even though they were synthesized from the previous highly-principled approach, the rewriting rules may feel strangely ad-hoc.

In this paper, we will propose a *logical* justification of this algorithm. It is based on recent developments in proof search, maximally multi-focused proofs [5]. The notion of *preemptive rewriting* was introduced in the meta-theory of multi-focusing as a purely technical device; we claim that it is in fact strongly related to Lindley's rewriting, and formally establish the correspondence.

The reference work on multi-focused systems [5] has been carried in a sequent calculus for linear logic. We will first establish the meta-theory of maximal multi-focusing for intuitionistic logic (Section 2). We start from a sequent calculus presentation, which is closest to the original system. Our first contribution is to propose an equivalent multi-focusing system in natural deduction 2.2. We then define preemptive rewriting in this natural deduction 2.4 and establish canonicity of maximally multi-focused proofs 2.6.

In Section 3, we transpose the preemptive rewrite rules into a relation on proof terms. We can then formally study the correspondence between rewriting a multi-focused proof into a canonical maximally multi-focused one, and Lindley's γ-reduction on lambda-terms. We demonstrate that they compute the same normal forms, modulo a form of redundancy elimination that is missing in the multi-focused system.

We finally introduce redundancy-elimination rewriting and equivalence for the proof terms of the multi-focused natural deduction (Section 4). The resulting notion of canonical proofs, *simplified maximal proofs*, precisely corresponds to normal forms of Lindley's rewriting relation. The natural notion of local equivalence between simplified maximal proofs therefore captures extensional equality.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 317–331

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Intuitionistic multi-focusing

The space of proofs in sequent calculus or natural deduction exhibits a lot of redundancy: many proofs that are syntactically distinct really encode the same semantics. In particular, it is often possible to permute two inference rules in a way that preserves the validity of proofs, but also the reduction semantics of the corresponding proof terms. If a permutation transforms a proof with rule A applied above rule B into a proof with rule B applied above rule A, we say that it is an A/B permutation (A is above the slash, as in the source proof).

Focusing is a general discipline that can be imposed upon proof system, based on the separation of inference rules into two classes. *Invertible* rules (called as such because their inverse is admissible) always preserve provability, and can thus be applied as early as possible. *Non-invertible* rules may result in dead ends if they are applied too early (consider proving $A+B \vdash A+B$ by first introducing the sum on the right-hand side)). In focusing calculi, derivations are structured in "sequences" or "phases", that either only apply invertible rules or only non-invertible rules. Focusing imposes that phases be as long as possible. During invertible phases, one must apply any valid invertible rule. During non-invertible phases, one *focuses* on a set of formulas, and applies non-invertible operations on those formulas as long as possible – if the phase is started too early, this may result in a dead end.

Invertibility determines a notion of polarity of logical connectives: we call *positive* those whose right-hand-side rule is non-invertible (they are "only interesting in positive position"), and *negative* those whose left-hand-side rule is non-invertible. In single-succedent intuitionistic logic, $(\to)$ is negative, $(+)$ is positive, and the product $(\times)$ may actually be assigned either polarity.

In single-sided calculi, non-invertible rules are those that introduce positive connectives, and are called "positive". For continuity of vocabulary, we will also call non-invertible rules *positive*, and invertible rules *negative*. In particular, a permutation that moves a non-invertible rule below an invertible rule is a "pos/neg permutation".

2.1 Multi-focused sequent calculus

Multi-focusing ([9, 5]) is an extension of focusing calculi where, instead of focusing on a single formula of the sequent (either on the left or on the right), we allow to simultaneously focus on several formulas at once. The multiple foci do not interact during the focusing phase, and this allows to express the fact that several focusing sequences are in fact independent and can be performed in parallel, condensing several distinct focused proofs into a single multi-focused derivation.

We start with a multi-focused variant of the intuitionistic sequent calculus, presented in Fig. 1. We denote focus using brackets: the rules with no brackets are invertible. This notation will change in natural deduction calculi.

In particular, we write A_n or Δ_p for formula or contexts that must be all negative or positive, and X, Y or Z for atoms. We write B_{pa} and Γ_{na} when either a positive (resp. negative) or an atom is allowed. For readability reasons, we only add polarity annotations when necessary; if we consider only derivations whose end conclusion is unfocused, then the invariant holds that the unfocused left-hand-side context is always all-negative, while the unfocused right-hand-side formula is always positive.

Our intuitionistic calculi are, as is most frequent, single-succedent. The notation $A \mid B$ on the right does not denote a real disjunction but a single formula, one of the two variables being empty. The focusing rule SEQ-FOCUS with conclusion $\Gamma, \Delta \vdash A \mid B$ can be instantiated in two ways, one when A is empty, and the premise is $\Gamma, [\Delta] \vdash [B]$ (the succedent is part of

$$\textsc{seq-atom}\ \frac{X \text{ atomic}}{\Gamma_n, X \vdash X} \qquad \textsc{seq-inv-sum-l}\ \frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A + B \vdash C} \qquad \textsc{seq-inv-prod-r}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \textsc{seq-inv-arr-r}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\textsc{seq-focus}\ \frac{\Gamma_{na}, [\Delta_n] \vdash A_{pa} \mid [B_{pa}]}{\Gamma_{na}, \Delta_n \vdash A_{pa} \mid B_{pa}} \qquad \textsc{seq-release}\ \frac{\Gamma, \Delta_{pa} \vdash A \mid B_{na}}{\Gamma, [\Delta_{pa}] \vdash A \mid [B_{na}]} \qquad \textsc{seq-foc-arr-l}\ \frac{\Gamma \vdash [A] \qquad \Gamma, [\Delta, B] \vdash C \mid [D]}{\Gamma, [\Delta, A \to B] \vdash C \mid [D]}$$

$$\textsc{seq-foc-prod-l}\ \frac{\Gamma, [\Delta, A_i] \vdash B \mid [C]}{\Gamma, [\Delta, A_1 \times A_2] \vdash B \mid [C]} \qquad \textsc{seq-foc-sum-l}\ \frac{\Gamma, [\Delta] \vdash [A_i]}{\Gamma, [\Delta] \vdash [A_1 + A_2]}$$

Figure 1 Multifocused sequent calculus for intuitionistic logic.

the multi-focus), and one when B is empty, and the premise is $\Gamma, [\Delta] \vdash A$ (the succedent is not part of the multi-focus). Note that Δ is a set and may be empty, in which case the focus only happens on the right.

As a minor presentation difference to the reference work on multi-focusing [5], our contexts are unordered multi-sets, and all the formulas under focus are released at once – by SEQ-RELEASE, which releases positives (resp. negatives) or atoms.

This multi-focused calculus proves exactly the same formulas as the singly-focused sequent calculus. The latter is trivially included in the former, and conversely one can turn a multi-focus into an arbitrarily ordered sequence of single foci. As a corollary, relying on non-trivial proofs from the literature (e.g., [11]), it is equivalent in provability to the (non-focused) sequent calculus for intuitionistic logic.

2.2 Multi-focused natural deduction

While the multi-focusing sequent calculus closely corresponds to existing focused presentations, its natural deduction presentation in Fig. 2 is new. We took inspiration from the presentation of focused linear logic in natural deduction of [3], in particular the $\Uparrow$ and $\Downarrow$ notations coming from intercalation calculi.

There are three main judgments. $\Gamma \vdash A$ is the unfocused judgment with the invertible rules. $\Gamma; A \Downarrow B$ is the "elimination-focused" judgment, and $A \Uparrow B$ is the "introduction-focused" judgment (focused on A). $\Gamma; A \Downarrow B$ means that the assertion B can be produced from the hypothesis A by non-invertible elimination rules; the context Γ is used in any non-focused subgoal. $A \Uparrow B$ means that proving the goal A can be reduced, by applying non-invertible introduction rules, to proving the goal B. Those two judgments do not come separately, they are introduced by the focusing rule NAT-FOCUS.

In Fig. 2, we used auxiliary rules (NAT-START-INTRO, NAT-START-NO-INTRO, NAT-START-ELIM) to present the focusing compactly (this is important when rewriting proofs); those rules can only happen immediately above NAT-FOCUS, and can thus be considered definitional syntactic sugar – we used a double bar to reflect this. If we inlined these auxiliary rules, the focusing rule would read (equivalently):

$$\frac{(A_n^i)^{i \in I} \subseteq \Gamma_{na} \qquad (\Gamma_{na}; A_n^i \Downarrow A_{pa}'^i)^{i \in I} \qquad (B_p \Uparrow B'_{na} \mid B = B') \qquad \Gamma_{na}, (A_{pa}'^i)^{i \in I} \vdash B'}{\Gamma_{na} \vdash B_{pa}}$$

This rule can only be used when all invertible rules have been performed: the context must be negative or atomic, and the goal positive or atomic. It selects set of foci on the

$$\textsc{nat-atom}\ \frac{X \text{ atomic}}{\Gamma_{na}, X \vdash X} \qquad \textsc{nat-inv-sum-l}\ \frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A + B \vdash C} \qquad \textsc{nat-inv-prod-r}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \textsc{nat-inv-arr-r}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\textsc{nat-focus}\ \frac{\Gamma_{na} \Downarrow \Gamma' \qquad A_{pa} \Uparrow^? A' \qquad \Gamma_{na}, \Gamma' \vdash A'}{\Gamma_{na} \vdash A_{pa}} \qquad \textsc{nat-end-elim}\ \frac{}{\Gamma; A_{na} \Downarrow A_{na}} \qquad \textsc{nat-end-intro}\ \frac{}{A_{na} \Uparrow A_{na}}$$

$$\textsc{nat-elim-arr}\ \frac{\Gamma; A \Downarrow B \to C \qquad B \Uparrow B' \qquad \Gamma \vdash B'}{\Gamma; A \Downarrow C} \qquad \textsc{nat-elim-prod}\ \frac{\Gamma; A \Downarrow B_1 \times B_2}{\Gamma; A \Downarrow B_i} \qquad \textsc{nat-intro-sum}\ \frac{A_i \Uparrow B}{A_1 + A_2 \Uparrow B}$$

$$\textsc{nat-start-no-intro}\ \frac{}{A \Uparrow^? A} \qquad \textsc{nat-start-intro}\ \frac{A_p \Uparrow B_{na}}{A_p \Uparrow^? B_{na}} \qquad \textsc{nat-start-elim}\ \frac{(A_n^i)^{i \in I} \subseteq \Gamma \qquad (\Gamma; A_n^i \Downarrow A_{pa}'^i)^{i \in I}}{\Gamma \Downarrow (A_{pa}'^i)^{i \in I}}$$

Figure 2 Multifocused natural deduction for intuitionistic logic.

left, the family of strictly negative assumptions $(A_n^i)^{i \in I}$ (we consistently use the superscript notation for family indices), and optionally a focus on the right; if the goal is focused it must be strictly positive. All foci must be as long as possible: elimination foci go from a variable down to a positive or atomic $A_{pa}'^i$, and the introduction focus goes up until it encounters a negative or atomic B_{na}'.

In comparison to the sequent calculus, the positive or atomic formulas $(A_{pa}'^i)^{i \in I}$ appearing at the *start* of the elimination-focus correspond to the formulas *released* at the *end* of a multi-focus in a sequent proof; natural deduction, when compared to the sequent calculus, has elimination rules "upside down". Also characteristic of natural deduction is the horizontal parallelism between eliminations and introductions; for example, the following two partial derivations correspond to the same natural deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A_{pa} \times B, A_{pa} \vdash C_{na}}{A_{pa} \times B, [A_{pa}] \vdash [C_{na}]}}{A_{pa} \times B, [A_{pa}] \vdash [C_{na} + D]}}{A_{pa} \times B, [A_{pa} \times B] \vdash [C_{na} + D]}}{A_{pa} \times B \vdash C_{na} + D} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A_{pa} \times B, A_{pa} \vdash C_{na}}{A_{pa} \times B, [A_{pa}] \vdash [C_{na}]}}{A_{pa} \times B, [A_{pa} \times B] \vdash [C_{na}]}}{A_{pa} \times B, [A_{pa} \times B] \vdash [C_{na} + D]}}{A_{pa} \times B \vdash C_{na} + D}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_{pa} \times B; A_{pa} \times B \Downarrow A_{pa} \times B}}{A_{pa} \times B; A_{pa} \times B \Downarrow A_{pa}} \qquad \dfrac{\dfrac{}{C_{na} \Uparrow C_{na}}}{C_{na} + D \Uparrow C_{na}} \qquad A_{pa} \times B, A_{pa} \vdash C_{na}}{A_{pa} \times B \vdash C_{na} + D}$$

On the other hand, we kept the less important invertible rules in sequent style: the sum elimination is a left introduction. Invertible rules being morally "automatically" applied, the sequent-style left introduction, which is directed by the type of its conclusion, is more natural in this context. Ironically, this brings us rather close to the sequent calculus of Krishnaswami [7] which, for presentation purposes, preserved a function-elimination rule in natural deduction style.

▶ **Lemma 1.** *The multi-focused natural deduction system proves exactly the same non-focused judgments as the multi-focused sequent calculus.*

$$\textsc{preempt-focus}\ \frac{\Gamma_{npa} \Downarrow \Delta'_{pa} \qquad B_p \Uparrow^? B'_{na} \qquad \Gamma_{npa}, \Delta'_{pa} \vdash A_{npa} \mid B'_{na}}{\Gamma_{npa} \vdash A_{npa} \mid B_p} \qquad \textsc{preempt-elim}\ \frac{\Gamma_{npa} \Downarrow \Delta'_{pa} \qquad \Gamma_{npa}, \Delta'_{pa}; A \Downarrow A'}{\Gamma_{npa}; A \Downarrow A'}$$

Figure 3 Preemptive rules for intuitionistic multifocused natural deduction.

2.3 A preemptive variant of multi-focused natural deduction

Multi-focusing was introduced to express the idea of *parallelism* between non-invertible rules on several independent foci. A proof has more parallelism than another if two sequential foci of the latter are merged (through rule permutations) in a single multi-focus in the former. A natural question is whether there exists "maximally parallel proofs". To answer it (affirmatively), the original article on multi-focusing ([5]) introduced a rewriting relation that permutes non-invertible phases down in proof derivations, until they cannot go any further without losing provability – neighboring phases can then be merged into a maximally focused proof.

In the process of moving down, a non-invertible phase will traverse invertible phases below. The intermediary states of this reduction sequence may break the invariant that invertible rules must be applied as early as possible; we say that the non-invertible phase *preempts* (a part of) the invertible phase. As this intermediary state is not a valid proof in off-the-shelf multi-focusing systems, the original article introduced a relaxed variant called a preemptive system, in which the phase-sinking transformation, called *preemptive rewriting*, can be defined following [5].

We present in Fig. 3 a preemptive variant of multi-focused natural deduction, except for the invertible and focused-introduction rules that are strictly unchanged from the previous multi-focusing rules in Fig. 2. There are two important differences:

- Preemption of invertible phases. To allow the start of a focusing phase when some invertible rules could still be applied, we lifted the polarity constraints for starting focusing. In the rule PREEMPT-FOCUS, the goal $\Gamma_{npa} \vdash A_{npa}$ may be of any polarity. We use a tautological Γ_{npa} annotation to emphasize this change.
- Preemption of non-invertible phases. This is expressed by the rule PREEMPT-ELIM, where an ongoing focus on A is preempted by a complete focus on Δ'_{pa}. Note that stored contexts are *not* available during the current elimination phase (they are unused in NAT-END-ELIM); they are only available to non-focused phases that appear as subgoals (in the arrow elimination rule). This preserves the central idea that the simultaneous foci of a single focusing rule are *independent*.

2.4 Preemptive rewriting

We can then define in Fig. 4 the rewriting relation on the preemptive calculus, that lets any non-invertible phase move as far as possible down the derivation tree. Maximally multi-focused proofs, which can be characterized on permutation-equivalence classes of multi-focused proofs, correspond to normal forms of this rewriting relation.

A focused phase cannot move below an inference rule if some of the foci depend on this inference rule. Instead of expressing the non-dependency requirement by implicit absence of the foci, we have explicitly canceled out the foci that *must be absent* to improve readability. In the first rule for example, $\Gamma, \cancel{A} \Downarrow \Delta$ means that the A hypothesis must be weakened (not used) in the derivation of $\Gamma \Downarrow \Delta$, orelse it cannot move below the introduction of A.

$$\frac{\dfrac{\Gamma, \cancel{A} \Downarrow \Delta \quad \cancel{B \Uparrow B'} \quad \Gamma, A, \Delta \vdash B}{\Gamma, A \vdash B}}{\Gamma \vdash A \to B} \quad \to \quad \frac{\Gamma \Downarrow \Delta \quad \dfrac{\Gamma, \Delta, A \vdash B}{\Gamma, \Delta \vdash A \to B}}{\Gamma \vdash A \to B}$$

$$\frac{\dfrac{\Gamma \Downarrow \Delta \quad \cancel{A \Uparrow A'} \quad \Gamma, \Delta \vdash A}{\Gamma \vdash A} \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \to \quad \frac{\Gamma \Downarrow \Delta \quad \dfrac{\Gamma, \Delta \vdash A \quad \Gamma \vdash B}{\Gamma, \Delta \vdash A \times B}}{\Gamma \vdash A \times B}$$

$$\frac{\left(\dfrac{\Gamma, \cancel{A} \Downarrow \Delta \quad C \Uparrow^? D \quad \Gamma, A, \Delta \vdash D}{\Gamma, A \vdash C}\right) \quad \left(\dfrac{\Gamma, \cancel{B} \Downarrow \Delta \quad C \Uparrow^? D \quad \Gamma, B, \Delta \vdash D}{\Gamma, B \vdash C}\right)}{\Gamma, A + B \vdash C} \quad \to \quad \frac{\Gamma \Downarrow \Delta \quad C \Uparrow^? D \quad \dfrac{\Gamma, A, \Delta \vdash D \quad \Gamma, B, \Delta \vdash D}{\Gamma, A + B, \Delta \vdash D}}{\Gamma, A + B \vdash C, D}$$

$$\frac{\dfrac{\Gamma \Downarrow \Gamma' \quad \Gamma, \Gamma'; A \Downarrow B_1 \times B_2}{\Gamma; A \Downarrow B_1 \times B_2}}{\Gamma, \Gamma'; A \Downarrow B_i} \quad \to \quad \frac{\Gamma \Downarrow \Gamma' \quad \dfrac{\Gamma, \Gamma'; A \Downarrow B_1 \times B_2}{\Gamma, \Gamma'; A \Downarrow B_i}}{\Gamma; A \Downarrow B_i}$$

$$\frac{\dfrac{\Gamma \Downarrow \Gamma' \quad \Gamma, \Gamma'; A \Downarrow B \to C}{\Gamma; A \Downarrow B \to C} \quad B \Uparrow B' \quad \Gamma \vdash B'}{\Gamma; A \Downarrow C} \quad \to \quad \frac{\Gamma \Downarrow \Gamma' \quad \dfrac{\Gamma, \Gamma'; A \Downarrow B \to C \quad B \Uparrow B' \quad \Gamma \vdash B'}{\Gamma, \Gamma' \Downarrow B \to C}}{\Gamma; A \Downarrow C}$$

$$\frac{\Gamma; A \Downarrow B \to C \quad B \Uparrow B'_{na} \quad \dfrac{\Gamma \Downarrow \Gamma' \quad \Gamma, \Gamma' \vdash B'_{na}}{\Gamma \vdash B'_{na}}}{\Gamma; A \Downarrow C} \quad \to \quad \frac{\Gamma \Downarrow \Gamma' \quad \dfrac{\Gamma; A \Downarrow B \to C \quad B \Uparrow B'_{na} \quad \Gamma, \Gamma' \vdash B'_{na}}{\Gamma, \Gamma'; A \Downarrow C}}{\Gamma; A \Downarrow C}$$

$$\frac{\dfrac{\Gamma \Downarrow \Gamma' \quad A_n \in \Gamma \quad \dfrac{\Gamma \Downarrow \Delta \quad \Gamma, \Delta; A_n \Downarrow A'}{\Gamma; A_n \Downarrow A'}}{\Gamma \Downarrow \Gamma', A'} \quad B \Uparrow^? B' \quad \Gamma, \Gamma', A' \vdash B'}{\Gamma \vdash B}$$

$$\to \quad \frac{\Gamma \Downarrow \Delta \quad \dfrac{\dfrac{\Gamma \Downarrow \Gamma' \quad A_n \in \Gamma \quad \Gamma, \Delta; A_n \Downarrow A'}{\Gamma, \Delta \Downarrow \Gamma', A'} \quad B \Uparrow^? B' \quad \Gamma, \Gamma', A' \vdash B'}{\Gamma, \Delta \vdash B}}{\Gamma \vdash B}$$

$$\frac{\Gamma \Downarrow \Delta \quad A \Uparrow^? B \quad \dfrac{\Gamma \Downarrow \Delta' \quad B \Uparrow^? C \quad \Gamma, \Delta, \Delta' \vdash C}{\Gamma, \Delta \vdash B}}{\Gamma \vdash A} \quad \leftrightarrow \quad \frac{\Gamma \Downarrow \Delta, \Delta' \quad A \Uparrow^? C \quad \Gamma, \Delta, \Delta' \vdash C}{\Gamma \vdash A}$$

Figure 4 Preemptive rewriting for multifocused natural deduction for intuitionistic logic.

In this situation, it may be the case that other parts of the multi-focus do not depend on the rule below, and those should not be blocked. To allow rewriting to continue, the last rewrite of our system is bidirectional. It allows to separate the foci of a multi-focus, in particular separate the foci that depend on the rule below from those that do not – and can thus permute again. This corresponds to the first rule of the original preemptive rewriting system [5], which splits a multi-focus in two. We only need to apply this rule when the result can make one more unidirectional rewrite step – this strategy ensures termination.

In the left-to-right direction, this rule relies on the possibility of merging together two elimination-focused derivations, or two optional introduction-focused derivations, with the implicit requirement that at least one of them is empty.

2.5 Reinversion

After the preemptive rewriting rules have been applied, the result is not, in general, a valid derivation in the non-preemptive system. Consider for example the following rewriting process:

$$\begin{bmatrix} & \nu_3 \\ & \pi_3 \\ \nu_2 & \\ \pi_2 & \\ \nu_1 & \\ \pi_1 & \end{bmatrix} \to^* \begin{bmatrix} & \nu_3 \\ \nu_2 & \\ & \pi_3;\nu_2 \\ \pi_2 & \\ \nu_1 & \\ \pi_1 & \end{bmatrix} \to^* \begin{bmatrix} \nu_2 & \nu_3 \\ \pi_2 & \pi_3 \\ \nu_1 & \\ \pi_1 & \end{bmatrix} \to^* \begin{bmatrix} \nu_2 & \nu_3 \\ \pi_2;\nu_3 & \\ \nu_1 & \\ & \pi_3;\nu_1 \\ \pi_1 & \end{bmatrix} \to^* \begin{bmatrix} \nu_2 & \nu_3 \\ \pi_2;\nu_3 & \\ \nu_1 & \\ \pi_1 & \pi_3 \end{bmatrix}$$

We are here representing derivations from a high-level point of view, by naming complete sequences of rules of the same polarity. Sequences of positive (non-invertible) are named π_n, and sequences of negative (invertible) rules ν_m. We use horizontal position to denote parallelism, or dependencies between phases: each dipole (π_k, ν_k) is vertically aligned as the invertibles of ν_k have been produced by the foci of π_k, but we furthermore assume that the second dipole depends on formulas released by the first, while the third dipole is independent.

The third dipole is independent from the others, and its foci in π_3 move downward in the derivation as expected in the preemptive system. After the first step, its negative phase has preempted the invertible phase ν_2, and it is thus written $\pi_3;\nu_2$ to emphasize that any rule of this sequence will have all the invertible formulas of ν_2 in non-focused positions (positives in the hypotheses, and negatives in the succedent). It can then be merged with the foci of π_2, in which case it does not see the invertibles of ν_2 anymore. When it moves further down, the invertible formulas in its topmost sequent, those consumed by ν_3, are present/preempted by all the non-invertible rules of π_2. It is eventually merged with π_1.

The normal form of this rewrite sequence could be considered a maximally multi-focused proof, in the sense that the foci happen as soon as possible in the derivation – which was not the case in the initial proof, where π_3 was delayed. However, while the initial proof is a valid proof in the non-preemptive system, the last derivation is not: the invertible formulas produced by π_3 are not consumed as early as possible, but only at the very end of the derivation, and the foci of π_2 therefore happen while there are still invertible rules to be applied.

We introduce a *reinversion* relation between proofs, written $\mathcal{D} \triangleright \mathcal{E}$, that turns the proof $\mathcal{D}$ with possible preemption into a proof $\mathcal{E}$ valid in the non-preemptive system, by doing the inversions where they are required, without changing the structure of the negative phases –

the foci are exactly the same. In our example, we have:

$$\begin{bmatrix} \nu_2 & \nu_3 \\ \pi_2;\nu_3 & \\ \nu_1 & \\ \pi_1 & \pi_3 \end{bmatrix} \triangleright \begin{bmatrix} \nu_2 & \\ \pi_2 & \\ \nu_1 & \nu_3 \\ \pi_1 & \pi_3 \end{bmatrix}$$

▶ **Definition 2** (Rewriting relation)**.** If $\mathcal{D}$ and $\mathcal{E}$ are proofs of the non-preemptive system, we write $\mathcal{D} \Rightarrow \mathcal{E}$ if there exists a $\mathcal{E}'$ such that $\mathcal{D} \to^* \mathcal{E}' \triangleright \mathcal{E}$.

Reinversion was not discussed directly in the original multi-focusing work [5], but it plays an important role and can be described and understood in several fairly different ways. For lack of space, we omit this discussion from this short article, and will only formally define reinversion as a relation on the (more concise) proof terms in Section 3.1, Definition 4.

2.6 Maximal multi-focusing and canonicity

Now that we have defined the focusing-lowering rewrite ($\Rightarrow$) between non-preemptive proof, we can define the notion of *maximal* multi-focusing and its meta-theory. It is defined by looking at the width of multi-focus phases in equivalence classes of rule permutations; but it can also be characterized as the normal forms of the ($\Rightarrow$) relation.

For lack of space, we have omitted this development (which is a mere adaptation of the previous work [5]) from this short article. The central result is summarized below.

▶ **Definition.** *We say that two proofs $\mathcal{D}$ and $\mathcal{E}$ are* locally equivalent*, or* iso-polar*, written $\mathcal{D} \approx_{loc} \mathcal{E}$, if one can be rewritten into the other using only local positive/positive and negative/negative permutations, preserving their initial sequents.*

▶ **Definition.** *We say that two proofs $\mathcal{D}$ and $\mathcal{E}$ are* globally equivalent*, or* iso-initial*, written $\mathcal{D} \simeq_{glob} \mathcal{E}$, when one can be rewritten into the other using local permutations of any polarity (so when seen as proofs in a non-focused system), preserving their initial sequents.*

▶ **Fact.** *Two multi-focused proofs are globally equivalent if and only if they are rewritten by ($\Rightarrow$) in locally equivalent maximal proofs.*

3 On the side of proof terms

3.1 Preemption and reinversion as term rewriting

Now that we have a notion of maximally multi-focused proofs in natural deduction, we can cross the second bridge between multi-focusing and Lindley's work by moving to a term system. We define in Figure 5 a term syntax for multi-focused derivations in natural deduction.

As the distinction between the preemptive and the non-preemptive systems are mostly about invariants of the focusing rule, the same term calculus is applicable to both. The only syntactic difference is that preemptive terms allow a multi-focusing $f[n]$ to preempt an ambient elimination focus n'.

Structural constraints on the multi-focusing system (preemptive or not) guarantee that strong typing invariants are verified. In particular, in a focused term ($\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?t$), the $\bar{n}$ are typed by the formulas in Δ at the end of a $\Gamma \Downarrow \Delta$ elimination phase: by our release discipline they have a positive or atomic type, so the `let`-introduced $\bar{x}$ are always bound to positive types. The rewriting rules corresponding to the preemptive rewriting relation are defined in Figure 6.

$$
\begin{array}{lll}
t ::= & & \text{terms} \\
& \mid x, y, z & \text{variable} \\
& \mid \lambda(x)\, t & \text{lambda} \\
& \mid (t, t) & \text{pair} \\
& \mid \delta(x, x.t, x.t) & \text{case} \\
& \mid f[t] & \text{focusing}
\end{array}
\qquad
\frac{X \text{ atomic}}{\Gamma_{na}, x : X \vdash x : X}
$$

$$
\frac{\Gamma, x : A \vdash t : C \qquad \Gamma, x : B \vdash u : C}{\Gamma, x : A + B \vdash \delta(x, x.t, x.u) : C}
\qquad
\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \times B}
\qquad
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x)\, t : A \to B}
$$

$f[\Box] ::= \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?\Box$ multi-focusing

$$
\begin{array}{lll}
p^? ::= & & \text{optional introduction focus} \\
& \mid \emptyset & \text{no introduction} \\
& \mid p & \text{introduction focus}
\end{array}
$$

$$
\frac{\Gamma_{na} \Downarrow \mathtt{let}\, \bar{x} = \bar{n} : \Gamma' \qquad A_{pa} \Uparrow^? p^? : A' \qquad \Gamma_{na}, \Gamma' \vdash t : A'}{\Gamma_{na} \vdash \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^? t : A_{pa}}
$$

$$
\frac{}{A \Uparrow^? \emptyset : A}
\qquad
\frac{A_p \Uparrow p : B_{na}}{A_p \Uparrow^? p : B_{na}}
\qquad
\frac{(x^i : A^i_n)^{i \in I} \subseteq \Gamma \qquad (\Gamma; x^i : A^i_n \Downarrow n^i : A'^i_{pa})^{i \in I}}{\Gamma \Downarrow \mathtt{let}\, (x^i)^{i \in I} = (n^i)^{i \in I} : (A'^i_{pa})^{i \in I}}
$$

$$
\begin{array}{lll}
n ::= & & \text{negatives} \\
& \mid x, y, z & \text{variable} \\
& \mid \pi_i\ n & \text{pair projection} \\
& \mid n\ p(t) & \text{function application} \\
& \mid \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ n & \text{focusing (only in the preemptive calculus)}
\end{array}
\qquad
\frac{}{\Gamma; x : A_{na} \Downarrow x : A_{na}}
$$

$$
\frac{\Gamma; A \Downarrow n : B \to C \qquad B \Uparrow p : B' \qquad \Gamma \vdash t : B'}{\Gamma; A \Downarrow n\ p(t) : C}
\qquad
\frac{\Gamma; A \Downarrow n : B_1 \times B_2}{\Gamma; A \Downarrow \pi_i\ n : B_i}\ i \in \{1, 2\}
$$

$$
\frac{\Gamma_{npa} \Downarrow \mathtt{let}\ (x^i)^{i \in I} = (n^i)^{i \in I} : \Delta'_{pa} \qquad \Gamma_{npa}, \Delta'_{pa}; A \Downarrow n' : A'}{\Gamma_{npa}; A \Downarrow \mathsf{let}\ (x^i)^{i \in I} = (n^i)^{i \in I}\ \mathsf{in}\ n' : A'}
$$

$$
\begin{array}{lll}
p ::= & & \text{positives} \\
& \mid \star & \text{identity} \\
& \mid \sigma_i\ p & \text{sum injection}
\end{array}
\qquad
\frac{}{A_{na} \Uparrow \star : A_{na}}
\qquad
\frac{A_i \Uparrow p : B}{A_1 + A_2 \Uparrow \sigma_i\ pB}\ i \in \{1, 2\}
$$

Figure 5 Preemptive term calculus.

▶ **Lemma 3.** *If t is a proof term for the preemptive derivation $\mathcal{D}$, then $t \to u$ if and only if u is a proof term for a preemptive derivation $\mathcal{E}$ with $\mathcal{D} \to \mathcal{E}$.*

The reinversion relation also has a corresponding term-rewriting interpretation. To perform each invertible rule as early as it should be, it suffices to let any invertible rule skip over a non-invertible phase it does not depend on. Depending on the order of the invertible rules after this phase, the invertible rule we want to move may be after a series of invertible rules that cannot be moved.

We "skip" over invertible contexts, we reduce invertible rules happening inside contexts of

$$
\begin{array}{rcl}
\lambda(y)\ \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ t & \overset{y \notin \bar{n}}{\to} & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ \lambda(y)\, t \\
((\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ t_1), t_2) & \to & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ (t_1, t_2) \\
(t_1, (\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ t_2)) & \to & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ (t_1, t_2) \\
\delta(y, y_1.\, (\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?t_1), y_2.\, (\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?t_2)) & \overset{y_1, y_2 \notin \bar{n}}{\to} & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?\delta(y, y_1.t_1, y_2.t_2) \\
\pi_i\ (\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ n') & \to & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ \pi_i\ n' \\
(\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ n')\ t & \to & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ n'\ t \\
n'\ p(\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ t) & \to & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ n'\ p(t) \\
\mathsf{let}\ y = (\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ n')\ \mathsf{in}\ p^?t & \to & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ \mathsf{let}\ y = n'\ \mathsf{in}\ p^?t \\
\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?(\mathsf{let}\ \bar{y} = \bar{n'}\ \mathsf{in}\ q^?\, t) & \overset{\bar{x} \notin \bar{n'}}{\leftrightarrow} & \mathsf{let}\ \bar{x}, \bar{y} = \bar{n}, \bar{n'}\ \mathsf{in}\ (p^?.q^?)t
\end{array}
$$

$$
\begin{array}{rclcrcl}
p^?.\emptyset & = & p^? & \qquad & \star.q & = & q \\
\emptyset.q^? & = & q^? & & (\sigma_i\ p).q & = & \sigma_i\ (p.q)
\end{array}
$$

Figure 6 Preemptive rewriting on proof terms.

$$
\begin{array}{rcl}
F_i[\Box] & ::= & \lambda(x)\,\Box \\
& | & \delta(x, x_1.\Box, x_2.t) \\
& | & \delta(x, x_1.t, x_2.\Box) \\
& | & (t, \Box) \\
& | & (\Box, t)
\end{array}
\qquad
\begin{array}{rcl}
C_{neg}[\Box] & ::= & n\ p(\Box) \\
& | & C_{neg}[\Box]\ p(t) \\
& | & \pi_i\ C_{neg}[\Box] \\
& | & C_{ni}[C_{neg}[\Box]]
\end{array}
$$

$$
C_i[\Box] \quad ::= \quad \Box \quad | \quad F_i[C_i[\Box]]
\qquad\qquad
\begin{array}{rcl}
C_{ni}[\Box] & ::= & \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?\Box \\
& | & \mathsf{let}\ \bar{x}, y = \bar{n}, C_{neg}[\Box]\ \mathsf{in}\ p^?t
\end{array}
$$

Figure 7 Invertible frames and contexts, non-invertible contexts and elimination contexts.

the form $C_{ni}[C_i[\ ]]$, where $C_i[t]$ is a notation for invertible contexts (defined using invertible frames $F_i[t]$), and $C_{ni}[t]$ for non-invertible contexts. Defining the latter requires describing negative/elimination contexts $C_{neg}[t]$, with holes where a term may appear in a series of elimination-focused terms.

▶ **Definition 4.** Reinversion can be precisely defined as the transitive congruence closure of the rewrite rules listed in Figure 8.

The rewrite conditions are expressed in terms of a $C[\Box] \prec c$ relation (read "context C *blocks* term-constructor c") that indicates a dependency of an invertible construction c on a given context $C[\Box]$. For example, it would make no sense to extrude a λ in argument position in a destructor, or move a sum-elimination $\delta(x)$ across the frame that defined the variable x. This blocking relation is defined in Figure 9 – $(A \mid B)$ in a rule means that the rule holds with either A or B in place of $(A \mid B)$.

It may at first seem surprising that reinversion rules have instances that are the *opposite* of some of the preemptive rewriting rules – those about pos/neg permutations. But that is precisely one of the purposes of reinversion: after preemptive rewriting rules have been fully applied, we undo those that have gone "too far", in the sense that they let a non-invertible phase preempt a portion of an invertible phase below, but were blocked by dependencies without reaching the next non-invertible phase. This blocked phase does not increase the parallelism of multi-focusing in the proof, but stops the derivation from being valid in the original multi-focusing system, so reinversion undoes its preemption.

$$C_{ni}[C_i[\lambda(x)\,t]] \overset{C_{ni}[C_i[\Box]] \not\prec \lambda}{\triangleright} \lambda(x)\,C_{ni}[C_i[t]]$$

$$C_{ni}[C_i[(t_1,t_2)]] \overset{C_{ni}[C_i[\Box]] \not\prec (,)}{\triangleright} (C_{ni}[C_i[t_1]], C_{ni}[C_i[t_2]])$$

$$C_{ni}[C_i[\delta(x,x_1.t_1,x_2.t_2)]] \overset{C_{ni}[C_i[\Box]] \not\prec \delta(x)}{\triangleright} \delta(x,x_1.C_{ni}[C_i[t_1]],x_2.C_{ni}[C_i[t_2]])$$

Figure 8 Reinversion rewrite rules.

$$c ::= (,) \mid \lambda \mid \delta(x)$$

$$\frac{y \in \bar{x}}{\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^?\Box \prec \delta(y)} \qquad \frac{p \neq \emptyset}{\mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p\,\Box \prec (,) \mid \lambda} \qquad \frac{C_{neg}[\Box] \prec c}{\mathsf{let}\ \bar{x}, y = \bar{n}, C_{neg}[\Box]\ \mathsf{in}\ p^?t \prec c}$$

$$((\Box,t) \mid (t,\Box) \mid \lambda(x)\,\Box) \prec ((,) \mid \lambda) \qquad \lambda(x)\,\Box \prec \delta(x) \qquad (\delta(x,y.\Box,z.t) \mid \delta(x,z.t,y.\Box)) \prec \delta(y)$$

$$n\ p(\Box) \prec (,) \mid \lambda \qquad \frac{C_{neg}[\Box] \prec c}{C_{neg}[\Box]\ p(t) \mid \pi_i\ C_{neg}[\Box] \mid C_{ni}[C_{neg}[\Box]] \prec c}$$

$$\frac{C_{ni}[\Box] \prec c \mid C_{neg}[\Box] \prec c}{C_{ni}[C_{neg}[\Box]] \prec c} \qquad \frac{F_i[\Box] \prec c \mid C_i[\Box] \prec c}{F_i[C_i[\Box]] \prec c} \qquad \frac{C_{ni}[\Box] \prec c \mid C_i[\Box] \prec c}{C_{ni}[C_i[\Box]] \prec c}$$

Figure 9 Reinversion blocking relation.

Remark, in relation to this situation, that preemptive rewriting cannot be easily defined on equivalence classes of neg/neg permutations (or other presentations of focusing that crush the invertible phase in one not-so-interesting step, such as higher-order focusing), as the order of the invertible rules in a single phase may determine where a non-invertible phase stops its preemption and is blocked in the middle of the invertible phase. Reinversion restores this independence on invertible ordering. This explains why the meta-theory of maximal multi-focusing was conducted in the non-preemptive system, using the relation between proofs that always applies reinversion after preemptive rewriting.

The other interesting case is a non-invertible phase π_0 having traversed a family of non-invertible phases $(\pi'_i)_{i\in I}$, before merging into some non-invertible phase π_1. Reinversion will move its negative phase ν_0, reverting the preemption of the (π'_i) on the invertible formulas introduced by π_0. But the important preemptions that happened, namely the traversal by π_0 of each of the invertible phases $(\nu'_i)_{i\in I}$, are not reverted: each ν'_i is blocked by the π'_i below and thus cannot be reverted below π_0. As π_0 traversed *both* the ν'_i and the π'_i, it does not have the corresponding invertible formulas in its context anymore, and is well-positioned even in a non-preemptive proof.

▶ **Lemma 5.** *If t is the proof term of the preemptive derivation $\mathcal{D} : \Gamma \vdash A$, and u is such that $t \triangleright u$, then u is a valid (preemptive) proof term for $\Gamma \vdash A$.*

▶ **Lemma 6.** *If u is a valid proof term in the preemptive system, and a normal form of the relation $(\triangleright)$, then u is also a valid proof term for the non-preemptive system.*

▶ **Theorem 7.** *If t is a proof term for $\mathcal{D}$ and u for $\mathcal{E}$, then $\mathcal{D} \Rightarrow \mathcal{E}$ if and only there is a u' such that $t \to^* u' \triangleright u$, and u is a normal form for $(\triangleright)$.*

3.2 Multi-focused terms as lambda-terms

There is a natural embedding $\lfloor t \rfloor$ of a multi-focused term t into the standard lambda-calculus, generated by the following transformation, where $t[\bar{x} := \bar{u}]$ represents simultaneous substitution:

$$\lfloor \mathsf{let}\ \bar{x} = \bar{n}\ \mathsf{in}\ p^? t \rfloor := \lfloor p^? \rfloor (\lfloor t \rfloor [\bar{x} := \bar{\lfloor n \rfloor}])$$

$$\lfloor \emptyset \rfloor (t) := t \qquad \lfloor \star \rfloor (t) := t \qquad \lfloor \sigma_i\ p \rfloor (t) := \sigma_i\ \lfloor p \rfloor (t)$$

The substitutions break the invariant that the scrutinee of a sum-elimination construct is always a variable. However, as only negative terms are substituted, sum-elimination scrutinee are always neutrals – embedding of negative terms. In particular, this embedding does not create any β-redex. Proof terms coming from non-preemptive multi-focusing are also always in η-long form, and this is preserved by the embedding; with the restriction present in Lindley's work that only neutral terms (eliminations) are expanded – this avoids issues of commuting conversions. We mean here the *weak* η-long form, determined by the weak equation $(m : A + B) =_{\text{weak-}\eta} \delta(m, x_1.\sigma_1\ x_1, x_2.\sigma_2\ x_2)$.

▶ **Lemma 8.** *If $\Gamma \vdash t : A$ in the preemptive multi-focused system, then $\Gamma \vdash \lfloor t \rfloor : A$ in simply-typed lambda-calculus, and $\lfloor t \rfloor$ is in β-normal form. If t is valid in the non-preemptive system, then the* pure neutral subterms *of $\lfloor t \rfloor$ are also in* weak *η-long form.*

3.3 Lindley's rewriting relation

The strong η-equivalence for sums makes lambda-term equivalence a difficult notion. For any term $m : A+B$ and well-typed context $C[\Box]$, it dictates that $C[m] \approx \delta(m, x_1.C[x_1], x_2.C[x_2])$. In his article [8], Sam Lindley breaks it down in four simpler equations, including in particular the "weak", non-local η-rule (where F represents a frame, that is a context of term-size exactly 1):

$$m \approx \delta(m, x_1.\sigma_1\ x_1, x_2.\sigma_2\ x_2) \quad (+.\eta)$$

$$F[\delta(p, x_1.t_1, x_2.t_2)] \approx \delta(p, x_1.F[t_1], x_2.F[t_2]) \quad (\textit{move-case})$$

$$\delta\left(p,\ \begin{array}{l} x_1.\delta(p, y_1.t_1, y_2.t_2), \\ x_2.\delta(p, z_1.u_1, z_2.u_2) \end{array}\right) \approx \delta(p, x_1.t_1[y_1 := x_1], x_2.u_2[z_2 := x_2]) \quad (\textit{repeated-guard})$$

$$\delta(p, x_1.t, x_2.t) \overset{x_1, x_2 \notin t}{\approx} t \quad (\textit{redundant-guard})$$

Lindley further refines the *move-case* equivalence into a less-local *hoist-case* rule. Writing D for a frame that is either $\delta(p, x_1.\Box, x_2.t)$ or $\delta(p, x_1.t, x_2.\Box)$, D^* for an arbitrary (possibly empty) sequence of them, and H any frame that *is not* of this from, *hoist-case* is defined as:

$$H[D^*[\delta(t, x_1.t_1, x_2.t_2)]] \to \delta(t, x_1.H[D^*[t_1]], x_2.H[D^*[t_2]])$$

Lindley's equivalence algorithm (Theorem 36, p. 13) proceeds in three steps: rewriting terms in $\beta\eta\gamma_E$-normal forms (using the weak $(+.\eta)$ on sums), then rewriting them in γ-normal form, and finally using a decidable redundancy-eliminating equivalence relation called $\sim$. The rewriting relation γ is defined as the closure of *repeated-guard*, *redundant-guard* (when read left-to-right) and *hoist-case*; γ_E is a weak restriction of it defined below. The equivalence $\sim$ is the equivalence closure of the equivalence *repeated-guard*, *redundant-guard*, and *move-case* restricted to D-frames – clauses of a sum elimination.

We discuss redundancy elimination, that is aspects related to *repeated-guard* and *redundant guard*, in Section 4, and focus here on explanation of the other rewriting processes ($\beta\eta\gamma_E$ and *hoist-case*) in logical terms. We show that multi-focused terms in $(\Rightarrow)$-normal form embed into $\beta\eta\gamma_E\gamma$-normal forms. As we ignore redundancy elimination, this is modulo $\sim$.

The β and η rewriting rules are standard – for sums, this is the weak, local η-relation, and not the strong η-equivalence. As explained in the previous subsection, embeddings of proof terms valid in the non-preemptive system – as are $(\Rightarrow)$-normal forms – are in $\beta\eta$-normal form. The rewriting γ_E is defined as the extrusion of a sum-elimination out of an elimination context: $\Box\ t \mid \pi_i\ \Box \mid \delta(\Box, x_1.t, x_2.t)$.

▶ **Lemma 9.** *Terms for valid preemptive multi-focusing derivations are in γ_E-normal form.*

This rigid structure of focused proofs is well-known, just as $\beta\eta$-normality or commuting conversions are not the interesting points of Lindley's work. The crux of the correspondence is between the transformation to maximal proofs, computed by $(\Rightarrow)$, and his γ-rewriting relation. There is an interesting dichotomy:

- Preemptive rewriting, which merges non-invertible phases, is where most of the work happens from a logical point of view. Yet this transformation, on the embeddings of the multi-focused proof terms, corresponds to the identity!
- Reinversion, which is obvious logically as it only concerns invertible rules which commute easily, corresponds to γ-rewriting on the embeddings.

Of course, preemptive rewriting is in fact crucial for γ-rewriting. It is the one that determines upto where negative terms can move in the derivation, and in particular the scrutinees of sum eliminations. Reinversion would not work without the first preemptive rewriting step, and applying reinversion on a proof term that is not in preemptive-normal form may not give a γ-normal embedding. Note that the proof of the last theorem in this section makes essential use of the confluence of γ-rewriting, one of Lindley's key results.

▶ **Lemma 10.** *If $t \to u$, then $\lfloor t \rfloor =_\alpha \lfloor u \rfloor$.*

▶ **Lemma 11.** *If $t \triangleright u$, then $\lfloor t \rfloor \to_\gamma^* \lfloor u \rfloor$.*

▶ **Lemma 12.** *If u is in $(\Rightarrow)$-normal form, then for some $u' \approx_{loc} u$, $\lfloor u' \rfloor$ is in γ-normal form modulo $\sim$.*

▶ **Theorem 13** (γ-normal forms are embeddings of maximally-focused proofs)**.** *If $\lfloor t \rfloor \to_\gamma^* n$ and n is γ-normal, then there are $u \approx_{loc} u'$ such that $t \Rightarrow u$ and $\lfloor u' \rfloor \sim n$. In particular, u is maximally multi-focused.*

4 Redundancy elimination

In the previous section, we have glossed over the fact that Lindley's γ-reduction also simplifies redundant and duplicated sum-eliminations. Those simplifications are *not* implied by multi-focusing – they are not justified by proof theory alone. Our understanding is that they correspond to purity assumptions that are stronger than the natural equational theory of focused proofs. On the other hand, starting from maximally multi-focused forms is essential to being able to define those extra simplifications. We do so in this section, to obtain a system that is completely equivalent to Lindley's.

We simply have to add the following simplifications on proof terms:

REDUNDANT-FOCUS
$\mathsf{let}\ \bar{x}, y, z = \bar{n}, n', n'\ \mathsf{in}\ p^?t \to_s \mathsf{let}\ \bar{x}, y = \bar{n}, n'\ \mathsf{in}\ p^?t[z := y]$

REDUNDANT-GUARD
$\delta(x, x_1.t, x_2.t) \overset{x_1, x_2 \notin t}{\approx_{\mathrm{loc}}} t$

REPEATED-CASE-1
$\delta(x, x_1.\delta(x, y_1.u_1, y_2.u_2), x_2.t_2) \approx_{\mathrm{loc}} \delta(x, x_1.u_1[y_1 := x_1], x_2.t_2)$

REPEATED-CASE-2
$\delta(x, x_1.t_1, x_2.\delta(x, y_1.u_1, y_2.u_2)) \approx_{\mathrm{loc}} \delta(x, x_1.y_1, x_2.u_2[y_2 := x_2])$

While those rules are not implied by focusing, they are reasonable in a focused setting, as they respect the phase separation. As the redundancy-elimination rules test for equality of subterms, they have an unpleasant non-atomic aspect (repeated cases only test variables), but this seems unavoidable to handle sum equivalence (Lindley [8], or Balat, Di Cosmo and Fiore [2], have a similar test in their normal form judgments), and have also been used previously in the multi-focusing literature, for other purposes; in Alexis Saurin's PhD thesis [10], an equality test is used to give a convenient $\otimes$/& permutation rule (p. 231).

▶ **Definition 14.** We define the relation $t \Rightarrow_s u$ between proof terms of the (preemptive) multi-focusing calculus as follows, where t_1 is a preemptive normal form, t_2 is a redundant-foci normal form, and u_0 is a ($\triangleright$)-normal form: $t \to^* t_1 \to_s^* t_2 \triangleright^* u_0 \approx_{\mathrm{loc}} u$

▶ **Definition 15.** We call the u in the target of the ($\Rightarrow_s$) relation *simplified maximal forms.*

▶ **Theorem 16** (Simplified maximal forms are γ-normal)**.** *Given a multi-focused term t, there exists some u such that $t \Rightarrow_s u$, $\lfloor t \rfloor \to_\gamma^* \lfloor u \rfloor$, and $\lfloor u \rfloor$ is in γ-normal form. This u is unique modulo local equivalence.*

▶ **Corollary 17.** *Two multi-focused proof terms are extensionally equivalent if their maximally multi-focused normal forms are locally equivalent (modulo redundancy elimination).*

Related and Future work

Maximally multi-focused proofs were previously used to bridge the gap between sequent calculus, as a rather versatile way of defining proof systems, and specialized proof structures designed to minimize redundancy for a fixed logic. The original paper on multi-focusing [5] demonstrated an isomorphism between maximal proofs and proof nets for a subset of linear logic. In recent work [4], maximally multi-focused proof of a sequent calculus for first-order logic have been shown isomorphic to *expansion proofs*, a compact description of first-order classical proofs.

There are some recognized design choices in the land of equivalence-checking presentation that can now be linked to design choices of focused system. For example, Altenkirch et al. [1] proposed to make the syntax more canonical with respect to redundancy-elimination by using a n-ary sum elimination construct, while Lindley prefers to quotient over local reorderings of unary sum-eliminations. This sounds similar to the choice between higher-order focusing ([12]), where all invertible rules are applied at once, or quotienting of concrete proofs by neg/neg permutations as used here.

When we started this work, we planned to also study the proof-term presentation of preemptive rewriting, in a term language for sequent calculus. We have been collaborating with Guillaume Munch-Maccagnoni to study the normal forms of an intuitionistic restriction of System L, with sums. In this untyped calculus, syntactic phases appear that closely

resemble a focusing discipline, and equivalence relations can be defined in a more uniform way, thanks to the symmetric status of the (non)-invertible rules that "change the type of the result" (terms, values) and those that only manipulate the environments (co-terms, stacks).

Conclusion

We propose a multi-focused calculus for intuitionistic logic in natural deduction, and establish the canonicity of maximally multi-focused proofs by transposing the preemptive rewriting technique [5] in our intuitionistic, natural deduction setting. By studying the computational effect of preemptive rewriting on proof terms, we demonstrate the close correspondence with the rewriting on lambda-terms with sums proposed by Lindley [8] to compute extensional equivalence. Adding a notion of redundancy elimination to our multi-focused system makes preemptive rewriting precisely equivalent to Lindley's γ-rules. In particular, the resulting canonical forms, *simplified maximal proofs*, capture extensional equality.

Acknowledgements. The author thanks his advisor, Didier Rémy for the freedom of making an overly long detour through the proof search literature; Alexis Saurin for his advice, and Pierre-Évariste Dagand, Sam Lindley, and anonymous reviewers for their helpful comments.

References

1 Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, 2001.
2 Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, 2004.
3 Taus Brock-Nannestad and Carsten Schürmann. Focused natural deduction. In *LPAR (Yogyakarta)*, 2010.
4 Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A Systematic Approach to Canonicity in the Classical Sequent Calculus. In *21st EACSL Annual Conference on Computer Science Logic*, volume 16, September 2012.
5 Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *IFIP TCS*, 2008.
6 N Ghani. Beta-eta equality for coproducts. In *Proceedings of TLCA'95*, number 902 in Lecture Notes in Computer Science. Springer-Verlag, 1995.
7 Neelakantan R. Krishnaswami. Focusing on pattern matching. In *POPL*, 2009.
8 Sam Lindley. Extensional rewriting with sums. In *TLCA*, 2007.
9 Dale Miller and Alexis Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In *CSL*, 2007.
10 Alexis Saurin. *Une étude logique du contrôle (appliquée à la programmation fonctionnelle et logique)*. PhD thesis, École Polytechnique, 2008.
11 Robert J. Simmons. Structural focalization. *CoRR*, abs/1109.6273, 2011.
12 Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A Proof-theoretic Characterization of Independence in Type Theory

Yuting Wang[1] and Kaustuv Chaudhuri[2]

1 **University of Minnesota, USA**
`yuting@cs.umn.edu`
2 **Inria & LIX/École polytechnique, France**
`kaustuv.chaudhuri@inria.fr`

Abstract

For λ-terms constructed freely from a type signature in a type theory such as LF, there is a simple inductive *subordination* relation that is used to control type-formation. There is a related – but not precisely complementary – notion of *independence* that asserts that the inhabitants of the function space $\tau_1 \to \tau_2$ depend vacuously on their arguments. Independence has many practical reasoning applications in logical frameworks, such as pruning variable dependencies or transporting theorems and proofs between type signatures. However, independence is usually not given a formal interpretation. Instead, it is generally implemented in an *ad hoc* and uncertified fashion. We propose a formal definition of independence and give a proof-theoretic characterization of it by: (1) representing the inference rules of a given type theory and a closed type signature as a theory of intuitionistic predicate logic, (2) showing that typing derivations in this signature are adequately represented by a focused sequent calculus for this logic, and (3) defining independence in terms of *strengthening* for intuitionistic sequents. This scheme is then formalized in a meta-logic, called $\mathcal{G}$, that can represent the sequent calculus as an inductive definition, so the relevant strengthening lemmas can be given explicit inductive proofs. We present an algorithm for automatically deriving the strengthening lemmas and their proofs in $\mathcal{G}$.

1998 ACM Subject Classification F.4.2. Mathematical logic: proof theory

Keywords and phrases subordination, independence, sequent calculus, focusing, strengthening

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.332

1 Introduction

In logical frameworks such as LF [6] or $\mathcal{G}$ [4] that are designed to reason about typed λ-terms *qua* data, one notion that appears again and again is *dependency*: when can the structure of one group of λ-terms depend essentially on that of another group of λ-terms? The most widely studied general notion of dependency is *subordination* [15, 17, 7], which is best explained using an example. Consider λ-terms built out of the following *type signature* of constants, where `nat` and `bt` respectively denote natural numbers and binary trees with natural numbers in the leaves.

$$\texttt{z} : \texttt{nat}. \qquad \texttt{s} : \texttt{nat} \to \texttt{nat}. \qquad \texttt{leaf} : \texttt{nat} \to \texttt{bt}. \qquad \texttt{node} : \texttt{bt} \to \texttt{bt} \to \texttt{bt}.$$

From this signature, it is immediately evident that a closed β-normal term of type `bt` can – indeed, *must* – contain a subterm of type `nat`, so we say that `nat` is *subordinate* to `bt`. The subordination relation $\leq$ on types can be derived from a type signature as follows (adapting [7, Definition 2.14]). For any type τ, write $\mathcal{H}(\tau)$ for its *head*, which is the basic type that occurs rightmost in the chain of $\to$s (or dependent products in the case of dependent types) in

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 332–346

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

τ. Then, for every type $\tau_1 \to \cdots \to \tau_n \to a$ (here, a is the head) that occurs anywhere in the signature, set $\mathcal{H}(\tau_i) \leq a$ for every $i \in 1..n$. Finally, define $\tau_1 \leq \tau_2$ generally as $\mathcal{H}(\tau_1) \leq \mathcal{H}(\tau_2)$ and close $\leq$ under reflexivity and transitivity.

With this definition, and the above signature, we have that $\texttt{nat} \leq \texttt{bt}$ but $\texttt{bt} \not\leq \texttt{nat}$. This notion of subordination, when strictly enforced such as in canonical LF [17, 7], enables a kind of modularity of reasoning: inductive theorems about the λ-terms of a given type can be proved in the *smallest* relevant signature and imported into larger signatures that do not contain subordinate types. For instance, meta-theorems about `nat`s, proved in a context of `nat` assumptions, can be transported to contexts of `bt` assumptions since $\texttt{bt} \not\leq \texttt{nat}$. It is indeed this complement of subordination that is most useful in reasoning: intuitively, $\texttt{bt} \not\leq \texttt{nat}$ means that the inhabitants of $\texttt{bt} \to \texttt{nat}$ are functions whose arguments cannot occur in their bodies in the β-normal form. This negative reading of subordination can be used to prune dependencies during unification, which may bring an unsolvable higher-order problem into a solvable pattern problem [9], or to prevent *raising* a variable over non-subordinate arguments, producing more concise proofs [3].

However, this pleasingly simple notion of subordination has a somewhat obscure formal interpretation: the definition is independent of the typing rules and it is unclear how they are related. We set out to formalize such a relation in terms of an inductive characterization of the (β-normal) inhabitants of types, and in the process we discovered a curious aspect of the above definition of subordination that manifests for higher-order constructors. Take, for instance, the alternative type $(\texttt{nat} \to \texttt{bt}) \to \texttt{bt}$ for `leaf`. Nothing changes as far as the $\leq$ relation is concerned: $\texttt{nat} \to \texttt{bt}$ still occurs in the signature, so $\texttt{nat} \leq \texttt{bt}$ is still true.[1] Yet, if we look at all β-normal terms of type `bt` now, there can be no subterms of type `nat` since there does not exist a constructor for injecting them into terms of type `bt` in the base case. The definition of $\leq$ above clearly over-approximates possible dependencies, for reasons that are not at all obvious, so its complement is too conservative.

In this paper we propose an alternative view of dependency that is not based on the subordination relation. We directly characterize when one type is *independent* of another with a *proof-theoretic* view of independence: for every claimed independence, we establish, by means of an induction over all typing derivations based on a given signature, that indeed a certain dependency is not strict. This view has several benefits:

- First, our notion of independence is larger than non-subordination, which means that it can be used for more aggressive pruning of dependencies.
- More important is that we have strong confidence in the correctness of our definition of independence, since it is now a derived property of the type system. Indeed, we propose an algorithm that extracts formal inductive proofs of independence that can be verified without needing a built-in notion of subordination or independence. This changes independence from a trusted framework-level procedure to a certifying procedure.
- Finally, we use only standard and simple proof-theoretic machinery in our definition. We require neither rich type systems nor sophisticated meta-theoretic tools.

Our view of independence has the following outline, which is also the outline of the paper.

1. We start (Section 2) by defining independence as a property of a given type theory.
2. We then (Section 3) describe a *specification language* built around the logic of higher-order hereditary Harrop formulas (HH) [10]. This is a fragment of first-order intuitionistic logic

[1] In canonical LF [7, Definition 2.14], well-formedness of signatures requires subordination of argument types of all dependent products.

with a particularly simple focused proof system [16]. It can also be seen as the minimal logical core of every logical framework.

3. We then (Section 4) use the language to give an *adequate* encoding of the inference system that underlies a given type system. To keep the presentation simple, we have chosen to use the simply typed λ-calculus, but the technique generalizes at least to LF [13, 14]. In terms of this encoding, we characterize independence as a particular kind of *strengthening*.
4. We then (Section 5) take the focused HH sequent calculus as an object logic in the reasoning logic $\mathcal{G}$ that adds the remaining crucial ingredients: a *closed world* reading, induction, $\beta\eta$-equality of λ-terms, and generic reasoning. This is the *two-level logic approach* [5] that underlies the Abella theorem prover [18].
5. Lastly (Section 6) we show how strengthening for typing derivations is formalized in $\mathcal{G}$, and give an algorithm for automatically deriving these lemmas from a given signature. We show an application of the formalization to pruning unnecessary dependence between variables in $\mathcal{G}$.

The Abella development of examples in this paper is available at:

`http://abella-prover.org/independence`.

2 Independence in Type Theory

Intuitively, in a given type theory, τ_2 is independent of τ_1 if and only if the type $\tau_1 \to \tau_2$ is only inhabited by abstractions whose bodies in the β-normal form do not contain the arguments. We can write this as a property on typing derivations:

▶ **Definition 1** (Independence)**.** Let $\Gamma \vdash t : \tau$ be the typing judgment in the given type theory. The type τ_2 is *independent* of τ_1 in Γ if whenever $\Gamma, x{:}\tau_1 \vdash t : \tau_2$ holds for some t, the β-normal form of t does not contain x, *i.e.*, $\Gamma \vdash t : \tau_2$ holds. ◀

A straightforward way to prove such a property is to perform inductive reasoning on the first typing derivation. For this, we need to know not only what are the possible ways to prove a typing judgment, but also that they are the only ways. This is the *closed-world assumption* that underlies reasoning about a fixed type signature. However, even with this assumption, the inductive proofs for independence can be hard to establish: since the target type τ_2 is fixed, the inductive hypothesis will not be applicable to new types encountered during the induction. To see this, suppose we are working with the simply-typed λ-calculus (STLC). Typing rules for STLC derive judgments of the form $\Gamma \vdash t : \tau$ where Γ is a context that assigns unique types to distinct variables, t is a λ-term and τ is its type. The typing rules are standard:

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \texttt{t-bas} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau}\ \texttt{t-app} \qquad \frac{\Gamma, x{:}\tau_1 \vdash t : \tau \quad (x \notin \Gamma)}{\Gamma \vdash \lambda x{:}\tau_1.\, t : \tau_1 \to \tau}\ \texttt{t-abs}$$

A direct induction on $\Gamma \vdash t : \tau$ using the typing rules produces three cases. The case when t is an application results in two premises where the premise for the argument has a new target type τ_1 (as shown in `t-app`). Since τ_1 can be any arbitrary type that is not necessarily related to τ, it is not possible to appeal to the inductive hypothesis.

The key observation is that the signature must be fixed for the dependence between types to be fully characterized. We propose an approach to formalize independence by (1) giving an encoding of a given type theory and a closed signature in a specification logic which finitely characterizes the dependence between types, (2) proving that the encoding is faithful to the

$$\frac{\Sigma, \bar{x}{:}\bar{\tau}; \Gamma, F_1, \ldots, F_n \vdash A}{\Sigma; \Gamma \vdash \Pi\bar{x}{:}\bar{\tau}.\, F_1 \Rightarrow \cdots \Rightarrow F_n \Rightarrow A}\ \textsf{reduce} \qquad \frac{(F \in \Gamma) \quad \Sigma; \Gamma, [F] \vdash A}{\Sigma; \Gamma \vdash A}\ \textsf{focus}$$

$$\frac{\Sigma \vdash \theta : \bar{\tau} \quad A'[\theta] = A \quad \left\{\Sigma; \Gamma \vdash F_i[\theta]\right\} \text{ for } i \in 1..n}{\Sigma; \Gamma, [\Pi x_1{:}\tau_1, \ldots, x_n{:}\tau_n.\, F_1 \Rightarrow \ldots \Rightarrow F_n \Rightarrow A'] \vdash A}\ \textsf{backchain}$$

Figure 1 Inference rules for HH. For reduce, we assume that $\bar{x} \notin \Sigma$. In the backchain rule, θ stands for a *substitution* $[t_1/x_1, \ldots, t_n/x_n]$; we write $\Sigma \vdash \theta : \bar{\tau}$ to mean $\Sigma \vdash t_i : \tau_i$ for each $i \in 1..n$.

original type theory and signature, and (3) formally stating and proving the independence as lemmas in a reasoning logic that gives an inductive reading of the encoding. The first two tasks are covered by Section 3 and 4. The last task is the topic of Section 5 and 6. We use STLC as the example throughout the paper; extending to other type theories such as LF is left for future work.

3 The Specification Language

Let us begin with a sketch of a specification language for encoding rule-based systems. This language will be used to encode type theories in later sections.

3.1 The HH Proof System

The logic of *higher-order hereditary Harrop formulas* (HH) is an intuitionistic and predicative fragment of Church's simple theory of types. Expressions in HH are simply typed λ-terms. Types are freely formed from built-in or user-defined atomic types containing at least the built-in type o (of formulas) and the arrow type constructor $\to$ (right-associative). The *head type* of τ is the atomic type that occurs rightmost in a chain of $\to$s. Terms are constructed from a *signature* (written Σ); we write $\Sigma \vdash t : \tau$ if a λ-term t has type τ in Σ with the usual rules. Logic is built into HH by means of logical constants including $\Rightarrow : \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ (written infix and right-associative) for implications and $\Pi_\tau : (\tau \to \mathsf{o}) \to \mathsf{o}$ for universal quantification over values of a type τ that does not contain o. Predicates are constants in the signature with head type o. For readability, we will often write $t_1 \Leftarrow t_2$ for $t_2 \Rightarrow t_1$, and abbreviate $\Pi_\tau(\lambda x{:}\tau.\, t)$ as $\Pi x{:}\tau.\, t$ and $\Pi x_1{:}\tau_1 \ldots \Pi x_n{:}\tau_n.\, t$ as $\Pi\bar{x}{:}\bar{\tau}.\, t$ where $\bar{x} = x_1, \ldots, x_n$ and $\bar{\tau} = \tau_1, \ldots, \tau_n$. We also omit type subscripts when they are obvious from context.

An *atomic formula* (written A) is a term of type o that head-normalizes to a term that is not an application of one of the logical constants $\Rightarrow$ or Π_τ. Every HH formula is equivalent to a *normalized formula* (written F) of the form $\Pi\bar{x}{:}\bar{\tau}.\, F_1 \Rightarrow \cdots \Rightarrow F_n \Rightarrow A$. In the rest of this paper we will assume that all formulas are normalized unless explicitly excepted. Given a normalized formula $F = \Pi\bar{x}{:}\bar{\tau}.\, F_1 \Rightarrow \cdots \Rightarrow F_n \Rightarrow A$, we write: $\mathcal{H}(F)$, the *head of* F, for A; $\mathcal{L}(F)$, the *body of* F, for the multiset $\{F_1, \ldots, F_n\}$; and $\mathcal{B}(F)$, the *binders of* F, for the typing context $\bar{x}{:}\bar{\tau}$.

The inference system for HH is a negatively polarized focused sequent calculus, also known as a uniform proof system, that contains two kinds of *sequents*: the *goal reduction* sequent $\Sigma; \Gamma \vdash F$ and the *backchaining* sequent $\Sigma; \Gamma, [F] \vdash A$ with F *under focus*. In either case, Γ is a multiset of formulas called *program clauses* and the right hand side formula of $\vdash$ is called the *goal formula*. Figure 1 contains the inference rules.

An HH proof usually starts (reading conclusion upwards) with a goal-reduction sequent $\Sigma; \Gamma \vdash F$. If F is not atomic, then the reduce rule is applied to make it atomic; this rule extends the signature with $\mathcal{B}(F)$ and the context with $\mathcal{L}(F)$. Then the focus rule is applied which selects a formula from the context Γ. This formula is then decomposed in the backchain

rule that produces fresh goal reduction sequents for (instances of the) body, assuming that the instantiated head of the focused formula matches the atomic goal formula. This process repeats unless the focused formula has no antecedents.

The three rules of HH can be combined together into one *synthetic* rule for goal reduction sequents that have an atomic goal formula.

$$\frac{F \in \Gamma \quad \Sigma \vdash \theta : \bar{\tau} \quad \mathcal{H}(F)[\theta] = A \quad \left\{\Sigma, \mathcal{B}(G[\theta]); \Gamma, \mathcal{L}(G[\theta]) \vdash \mathcal{H}(G[\theta])\right\} \text{ for } G \in \mathcal{L}(F)}{\Sigma; \Gamma \vdash A}\ \mathsf{bcred}$$

Every premise of this rule is a goal reduction sequent with an atomic goal formula. In the rest of this paper we will limit our attention to this fragment of HH.

3.2 Encoding Rule-based Systems in HH

Because the expressions of HH are λ-terms, we can use the *λ-tree approach to syntax* (λTS), sometimes known as *higher-order abstract syntax* (HOAS), to represent the rules of deductive systems involving binding using the binding structure of λ-terms. Binding in object-language syntax is represented explicitly by meta-language λ-abstraction, and recursion over such structures is realized by introducing fresh new constants using universal goals and recording auxiliary properties for such constants via hypothetical goals. This kind of encoding is concise and, as we shall see in later sections, has logical properties that we can use in reasoning.

We present the encoding of typing rules for STLC as described in Section 2 as a concrete example of specifying in HH. Two basic types, `ty` and `tm`, are used for classifying types and terms in STLC. We then use the following signature defining the type and term constructors:

$$\texttt{b} : \texttt{ty}. \quad \texttt{arr} : \texttt{ty} \to \texttt{ty} \to \texttt{ty}. \qquad \texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm}. \quad \texttt{abs} : \texttt{ty} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm}.$$

The type of the `abs` uses λTS to encode binding. To illustrate, $(\lambda y{:}b \to b.\, \lambda x{:}b.\, y\ x)$ is encoded as $\texttt{abs}\ (\texttt{arr}\ \texttt{b}\ \texttt{b})\ (\lambda y.\, \texttt{abs}\ \texttt{b}\ (\lambda x.\, \texttt{app}\ y\ x))$.

We define a predicate $\texttt{of} : \texttt{tm} \to \texttt{ty} \to \texttt{o}$ to encode the typing judgments in STLC, with the context implicitly represented by the context of HH sequents. The typing rules are encoded by the following program clauses (where the outermost universal quantifiers are omitted):

$$\texttt{of}\ (\texttt{app}\ M_1\ M_2)\ T \Leftarrow \texttt{of}\ M_1\ (\texttt{arr}\ T_1\ T) \Leftarrow \texttt{of}\ M_2\ T_1$$
$$\texttt{of}\ (\texttt{abs}\ T_1\ R)\ (\texttt{arr}\ T_1\ T) \Leftarrow (\Pi x.\, \texttt{of}\ (R\ x)\ T \Leftarrow \texttt{of}\ x\ T_1)$$

Here, we are following the usual logic programming convention of writing universally quantified variables using upper-case identifiers.

To see that these clauses accurately represent the typing rules of STLC, consider deriving a HH sequent $\Sigma; \Gamma \vdash \texttt{of}\ M\ T$, where Γ contains the clauses above and the possible assignments of types to variables introduced by the second clause corresponding to the abstraction rule `t-abs`. The only way to proceed is by focusing and backchaining on one of the clauses. Backchaining on the first clause unifies the goal with the head formula and produces two premises that corresponds to the premises of `t-app`. Backchaining on the second clause followed by goal reduction results in $\Sigma, x : \texttt{tm}; \Gamma, \texttt{of}\ x\ T_1 \vdash \texttt{of}\ (R\ x)\ T$. Note that x is a fresh variable introduced by reducing the universal goal and $\texttt{of}\ x\ T_1$ is the typing assignment of x from reducing the hypothetical goal, which exactly captures the side condition of `t-abs` that x must be fresh for Γ. The rule `t-bas` is modeled by backchaining on assumptions in Γ that assigns types to variables introduced by `t-abs`.

$\hat{\texttt{ty}}\ \texttt{b}.\qquad \hat{\texttt{ty}}\ (\texttt{arr}\ T_1\ T_2) \Leftarrow \hat{\texttt{ty}}\ T_1 \Leftarrow \hat{\texttt{ty}}\ T_2.\qquad \hat{\texttt{tm}}\ (\texttt{app}\ M_1\ M_2) \Leftarrow \hat{\texttt{tm}}\ M_1 \Leftarrow \hat{\texttt{tm}}\ M_2.$
$\hat{\texttt{tm}}\ (\texttt{abs}\ T\ R) \Leftarrow (\Pi x{:}\texttt{tm}.\, \hat{\texttt{tm}}\ (R\ x) \Leftarrow \hat{\texttt{tm}}\ x) \Leftarrow \hat{\texttt{ty}}\ T.$

Figure 2 An example encoding of an STLC signature into HH clauses

4 Independence via Strengthening

This section presents an encoding of STLC in HH that finitely captures the dependence between types in a closed signature. Then the independence property can be stated as strengthening lemmas and proved by induction.

4.1 An Adequate Encoding of STLC

The encoding is based on the *types-as-predicates* principle: every type is interpreted as a predicate that is true of its inhabitants. The atomic types and constants of STLC are imported directly into the HH signature. For every atomic type b, we define a predicate $\hat{b} : b \to \texttt{o}$. We then define a mapping $[\![-]\!]$ from STLC types τ to a function $\tau \to \texttt{o}$ as follows:

$$[\![b]\!] = \lambda t.\, \hat{b}\ t \quad \text{if } b \text{ is an atomic type.} \qquad [\![\tau_1 \to \tau_2]\!] = \lambda t.\, \Pi x{:}\tau_1.\, [\![\tau_1]\!]\ x \Rightarrow [\![\tau_2]\!]\ (t\ x)$$

The mapping $[\![-]\!]$ is extended to typing contexts: for $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, we write $[\![\Gamma]\!]$ for the multiset of HH formulas $\{[\![\tau_1]\!]\ x_1, \ldots, [\![\tau_n]\!]\ x_n\}$. A typing judgment $\Gamma \vdash t : \tau$ is encoded as an HH sequent $\Gamma; [\![\Gamma]\!] \vdash [\![\tau]\!]\ t$.

As an example, consider the signature with STLC as an object language as described in Section 3.2 (not to be confused with STLC we are encoding). We have two predicates $\hat{\texttt{tm}} : \texttt{tm} \to \texttt{o}$ and $\hat{\texttt{ty}} : \texttt{ty} \to \texttt{o}$. Let Γ be the STLC signature containing `b`, `arr`, `app` and `abs`. It is translated into $[\![\Gamma]\!]$ containing the program clauses in Figure 2 (in normalized form and with outermost quantifiers elided): The typing judgment $\Gamma, y : \texttt{tm} \to \texttt{tm} \vdash \texttt{abs b}\ y : \texttt{tm}$ which is provable in STLC is encoded as $\Gamma, y{:}\texttt{tm} \to \texttt{tm}; [\![\Gamma]\!], (\Pi x{:}\texttt{tm}.\, \hat{\texttt{tm}}\ x \Rightarrow \hat{\texttt{tm}}\ (y\ x)) \vdash \hat{\texttt{tm}}\ (\texttt{abs b}\ y)$ which is also provable in HH.

This encoding generalizes to richer type systems than STLC. An encoding of LF into HH was presented in [2, 13, 14], which is essentially a superset of the encoding we are doing. The soundness and completeness of the STLC encoding follows easily from the results in [13].

▶ **Theorem 2.** *Let Γ be a well-formed context and τ be a type in* STLC. *If $\Gamma \vdash t : \tau$ has a derivation for a $\beta\eta$-normal term t, then there is a derivation of $\Gamma; [\![\Gamma]\!] \vdash [\![\tau]\!]\ t$ in* HH. *Conversely, if $\Gamma; [\![\Gamma]\!] \vdash [\![\tau]\!]\ t$ for any term t that is well-typed in* HH, *then $\Gamma \vdash t' : \tau$ where t' is the canonical (β-normal η-long) form of t.*

Proof. This theorem is a special case of [13, Theorem 1]. The proof is almost the same. ◀

4.2 Independence as Strengthening Lemmas

By Definition 1 and Theorem 2, τ_2 is independent from τ_1 in Γ iff the following *strengthening* lemma is true: if $\Gamma, x{:}\tau_1; [\![\Gamma]\!], [\![\tau_1]\!]\ x \vdash [\![\tau_2]\!]\ t$ holds for some t, then the β-normal form of t does not contain x and $\Gamma; [\![\Gamma]\!] \vdash [\![\tau_2]\!]\ t$ holds. Because the typing information in formulas generated by $[\![-]\!]$ is statically determined by predicates and the translated program is finite, it is possible to determine the dependence between types finitely. Thus, the independence argument can be proved by induction.

Using the previous encoding example, let's see how to prove that `ty` is independent of `tm` in Γ which is the signature with STLC in Section 3.2. We need to show that $\Gamma, x{:}\texttt{tm} \vdash t : \texttt{ty}$

implies $\Gamma \vdash t : \mathtt{ty}$, which by Theorem 2 is equivalent to $\Gamma, x{:}\mathtt{tm}; [\![\Gamma]\!], \hat{\mathtt{tm}}\ x \vdash \hat{\mathtt{ty}}\ t$ implying $\Gamma; [\![\Gamma]\!] \vdash \hat{\mathtt{ty}}\ t$. The proof proceeds by induction on the first assumption. Apparently, $\hat{\mathtt{ty}}\ t$ can only be proved by backchaining on clauses whose heads start with $\hat{\mathtt{ty}}$. The case when $\hat{\mathtt{ty}}\ t$ is proved by backchaining on $\hat{\mathtt{ty}}\ \mathtt{b}$ is immediate. For the case when $t = \mathtt{arr}\ t_1\ t_2$, backchaining produces two premises $\Gamma, x{:}\mathtt{tm}; [\![\Gamma]\!], \hat{\mathtt{tm}}\ x \vdash \hat{\mathtt{ty}}\ t_i$ $(i \in \{1, 2\})$. Applying the inductive hypothesis we get $\Gamma; [\![\Gamma]\!] \vdash \hat{\mathtt{ty}}\ t_i$, from which the conclusion $\Gamma; [\![\Gamma]\!] \vdash \hat{\mathtt{ty}}\ (\mathtt{arr}\ t_1\ t_2)$ follows easily.

5 The Two-Level Logic Approach

5.1 The Reasoning Logic $\mathcal{G}$

When a relation is described as an inductive inference system, the rules are usually understood as fully characterizing the relation. When interpreting the relation as a computation, we can just give the inference rules a *positive* interpretation. However, when reasoning about the properties of the inference system, the inductive definition must be seen in a *negative* interpretation, *i.e.*, as a prescription of the *only* possible ways to establish the encoded property. Concretely, given the rules for typing λ-terms in STLC, we not only want to identify types with typable terms, but also to argue that a term such as $\lambda x.\, x\ x$ does not have a type. We sketch the logic $\mathcal{G}$ that supports this complete reading of rule-based specifications by means of *fixed-point definitions* [4].

To keep things simple, $\mathcal{G}$ uses the same term language as HH and is also based on Church's simple theory of types. At the type level, the only difference is that $\mathcal{G}$ formulas have type $\mathtt{prop}$ instead of $\mathtt{o}$. The non-atomic formulas of $\mathcal{G}$ include formulas of ordinary first-order intuitionistic logic, built using the constants $\top, \bot : \mathtt{prop}$, $\wedge, \vee, \supset$: $\mathtt{prop} \to \mathtt{prop} \to \mathtt{prop}$ (written infix), and $\forall_\tau, \exists_\tau : (\tau \to \mathtt{prop}) \to \mathtt{prop}$ for types τ that do not contain $\mathtt{prop}$. To this, $\mathcal{G}$ adds intensional equality at all types, $=_\tau$: $\tau \to \tau \to \mathtt{prop}$, which is interpreted as $\beta\eta$-convertibility. Like with HH earlier, we will drop the explicit types for these constants and write quantifiers using more familiar notation. The proof system of $\mathcal{G}$ is a sequent calculus with the standard derivation rules for logical constants [5], which we will not repeat here.

The next crucial ingredient in $\mathcal{G}$ is a mechanism for *defining* predicates by means of *fixed-point definitions*. Such a definition is characterized by a collection of *definitional clauses* each of which has form $\forall \bar{x}.A \triangleq B$ where A is an atomic formula all of whose free variables are bound by $\bar{x}$ and B is a formula whose free variables must occur in A. A is called the head of such a clause and B is called its body. (For a fixed-point definition to be well-formed, it must satisfy certain stratification conditions [8] that we do not elaborate on here.) To illustrate definitions, let $\mathtt{olist}$ be a new basic type for lists of HH formulas, built from the type signature $\mathtt{nil} : \mathtt{olist}$ and $(\mathtt{::}) : \mathtt{o} \to \mathtt{olist} \to \mathtt{olist}$. Then, list membership ($\mathtt{member}$) and list concatenation ($\mathtt{append}$) may be specified in $\mathcal{G}$ using the following definitional clauses:

$$\mathtt{member}\ X\ (X :: L) \triangleq \top. \qquad \mathtt{member}\ X\ (Y :: L) \triangleq \mathtt{member}\ X\ L.$$

$$\mathtt{append}\ \mathtt{nil}\ L\ L \triangleq \top. \qquad \mathtt{append}\ (X :: L_1)\ L_2\ (X :: L_3) \triangleq \mathtt{append}\ L_1\ L_2\ L_3.$$

As before, we use the convention of indicating universally closed variables with upper-case identifiers. Read positively, these clauses can be used in the standard backchaining style to derive atomic formulas: the goal must match with the head of a clause and the backchaining step reduces the goal to deriving the instances of corresponding body. Read negatively, they are used to do case analysis on an assumption: if an atomic formula holds, then it must be the case that it unifies with the head of some clause defining it and the body of the clause is derivable. It therefore suffices to show that the conclusion follows from each such possibility.

Fixed-point definitions can also be interpreted inductively or coinductively, leading to corresponding reasoning principles. We use an annotated style of reasoning to illustrate how induction works in $\mathcal{G}$. Its interpretation into the proof theory of $\mathcal{G}$ is described in [3]. When proving a formula $(\forall \bar{x}.\, D_1 \supset \cdots \supset A \supset \cdots \supset D_n \supset G)$ by induction on the atom A, the proof reduces to proving G with the inductive hypothesis $(\forall \bar{x}.\, D_1 \supset \cdots \supset A^* \supset \cdots \supset D_n \supset G)$ and assumptions $D_1, \ldots, A^@, \ldots, D_n$. Here, A^* and $A^@$ are simply annotated versions of A standing for *strictly smaller* and *equal sized* measures respectively. When $A^@$ is *unfolded* by using a definitional clause, the predicates in the body of the corresponding clause are given the * annotation. Thus, the inductive hypothesis can only be used on A^* that results from unfolding $A^@$ at least once.

The negative reading of fixed-point definitions in $\mathcal{G}$ requires some care. In the negative view, universally quantified variables are interpreted *extensionally*, *i.e.*, as standing for all their possible instances. To illustrate, if `nat` were defined by $\texttt{nat z} \triangleq \top$ and $\texttt{nat (s } X\texttt{)} \triangleq \texttt{nat } X$, then one can derive $\forall x.\, \texttt{nat } x \supset G$ by case-analysis of the assumption $\texttt{nat } x$, which amounts to proving $[\texttt{z}/x]G$ and $\forall y.\, \texttt{nat } y \supset [\texttt{s } y/x]G$. This extensional view of universal variables is not appropriate when reasoning about binding structures viewed as syntax, where the syntactic variables are not stand-ins for all terms but rather for *names*. To see this clearly, consider the formula $\forall w.\, (\lambda x.\, w) = (\lambda x.\, x) \supset \bot$. If equality of λ-terms were interpreted extensionally with $\forall$, we would be left with $\forall w.\, (\forall x.\, w = x) \supset \bot$ which is not provable. Yet, the λ-terms $(\lambda x.\, w)$ and $(\lambda x.\, x)$ denote the constant and identity functions, respectively, and are therefore intensionally different – neither is $\beta\eta$-convertible to the other.

To achieve this intensional view, we come to the final ingredient of $\mathcal{G}$: *generic reasoning*. Every type of $\mathcal{G}$ is endowed with an infinite number of *nominal constants*, and there is a quantifier $\nabla_\tau : (\tau \to \texttt{prop}) \to \texttt{prop}$ (for τ not containing `prop`) to abstract over such constants. The critical features of nominal constants are: (1) they are not $\beta\eta$-convertible, so $\nabla x.\, \nabla y.\, x = y \supset \bot$ is derivable; and (2) formulas on the left and right of a $\mathcal{G}$ sequent interact up to *equivariance*, which allows the nominal constants in one formula to be systematically permuted to match those of the other. This latter property allows the ∇ quantifier to permute with all other connectives, including disjunction. The rules for introducing ∇ quantified formulas both as assumption and conclusion are similar: a formula $\nabla x.A$ is reduced to $[c/x]A$ where c is a nominal constant that is not already present in A.

The general form of a $\mathcal{G}$ definitional clause therefore allows the ∇ quantifier in heads: $\forall \bar{x}.\, (\nabla \bar{z}.\, A) \triangleq B$. An instance of the clause must have $\bar{z}$ replaced by distinct nominal constants that are not already present in the formula. Since $\bar{x}$ is quantified outside of $\bar{z}$, their instantiations cannot contain $\bar{z}$, which is used to encode structural properties of terms in the definitions. For example, $(\nabla x.\, \texttt{name } x) \triangleq \top$ defines `name` such that $\texttt{name } X$ holds only if X is a nominal constant. Another example is $\forall T.\, (\nabla x.\, \texttt{fresh } x\ T) \triangleq \top$ which defines `fresh` such that $\texttt{fresh } X\ T$ holds only if X is a nominal constant that does not occur in T.

5.2 An Encoding of HH in $\mathcal{G}$

The HH proof system can be encoded as a fixed-point definition in $\mathcal{G}$. The logical constants, signatures, and terms are imported into $\mathcal{G}$ transparently from HH – this is possible since $\mathcal{G}$ and HH use the same simple type system. The sequents of HH are then encoded in $\mathcal{G}$ using the predicates $\texttt{seq} : \texttt{olist} \to \texttt{o} \to \texttt{prop}$ and $\texttt{bch} : \texttt{olist} \to \texttt{o} \to \texttt{o} \to \texttt{prop}$. The encodings and their abbreviated notations are as follows:

HH	$\mathcal{G}$	notation
$\Sigma; \Gamma \vdash F$	$\texttt{seq } \Gamma\ F$	$\{\Gamma \vdash F\}$
$\Sigma; \Gamma, [F] \vdash A$	$\texttt{bch } \Gamma\ F\ A$	$\{\Gamma, [F] \vdash A\}$

$$\texttt{seq}\ L\ (F \Rightarrow G) \triangleq \texttt{seq}\ (F :: L)\ G. \qquad \texttt{bch}\ L\ (G \Rightarrow F)\ A \triangleq \texttt{seq}\ L\ G \wedge \texttt{bch}\ L\ F\ A.$$
$$\texttt{seq}\ L\ (\Pi x{:}\tau.\ F\ x) \triangleq \nabla x{:}\tau.\ \texttt{seq}\ L\ (F\ x). \qquad \texttt{bch}\ L\ (\Pi x{:}\tau.\ F\ x)\ A \triangleq \exists t{:}\tau.\ \texttt{bch}\ L\ (F\ t)\ A.$$
$$\texttt{seq}\ L\ A \triangleq \texttt{atom}\ A \wedge \texttt{member}\ F\ L \wedge \texttt{bch}\ L\ F\ A. \qquad \texttt{bch}\ L\ A\ A \triangleq \top.$$

Figure 3 Encoding of HH rules as inductive definitions in $\mathcal{G}$.

The definitional clause for `seq` and `bch` are listed in Figure 3. Note that we use a list of formulas to represent the multiset context in HH. This is not a problem since we can prove that the ordering of clauses in the list in `seq` and `bch` formulas does not affect their derivability as theorems about `seq` and `bch` in $\mathcal{G}$.

The encoding of HH in $\mathcal{G}$ is adequate. The first two clauses defining `seq` exactly capture the reduce rule. Note that in the second clause we use ∇ to encode Π in HH, since Π introduces fresh and unanalyzable eigenvariables into the HH signature. The third clause encodes the focus rule, where `atom` is defined as

$$\texttt{atom}\ F \triangleq \big(\forall G.\ (F = \Pi x{:}\tau.\ G\ x) \supset \bot\big) \wedge \big(\forall G_1, G_2.\ (F = (G_1 \Rightarrow G_2)) \supset \bot\big).$$

The clauses defining `bch` exactly capture the backchain rule. Note that the two rules for introducing Π actually represents a collection of rules for each instance of τ. In other words, these rule are actually schematic rules. This is possible since the proof theory allows inductive definitions to have infinitely many clauses. We will often write elements in `olist` in reverse order separated by commas; the exact meaning of the comma depends on its context. For example, given $L_1, L_2 : \texttt{olist}$ and $A : \texttt{o}$, $\{L_1, L_2, A \vdash G\}$ stands for $\texttt{seq}\ (A :: L)\ G$ for some $L : \texttt{olist}$ such that $\texttt{append}\ L_1\ L_2\ L$ holds.

Theorems of HH specifications can be proved through this encoding in $\mathcal{G}$. As an example, consider proving the uniqueness of the encoding of typing in STLC shown in Section 3.2. The theorem can be stated as follows:

$$\forall L, M, T, T'.\ \{\Gamma, L \vdash \texttt{of}\ M\ T\} \supset \{\Gamma, L \vdash \texttt{of}\ M\ T'\} \supset T = T'.$$

where Γ represents the program clauses defining `of` in Section 3.2.[2]

6 Formalizing Independence

This section first describes the formalization of independence in terms of *strengthening* for HH sequents of a certain shape in $\mathcal{G}$. Then a general algorithm for automatically deriving such strengthening lemmas is presented. Lastly, an application of the formalization for pruning variable dependencies in $\mathcal{G}$ is described.

6.1 Independence as Strengthening Lemmas in $\mathcal{G}$

A strengthening lemma has the following form in $\mathcal{G}$: $\forall \bar{x}.\ \{\Gamma, F \vdash G\} \supset \{\Gamma \vdash G\}$. As we have discussed in Section 3, an HH derivation always starts with applying reduce to turn the right hand side to atomic form. Thus, it suffices to consider instances of strengthening lemmas where G is atomic. This lemma is usually proved by induction on the only assumption. The proof proceeds by inductively checking that a derivation of $\{\Gamma', F \vdash A\}$ cannot contain any application of focus rule on F. In Section 3 we have shown that an HH derivation starting

[2] This is not precisely correct, since the typing context L also needs to be characterized by some inductive property; a complete exposition on this encoding can be found in the Abella tutorial [1].

with an atomic goal can be seen as repeatedly applying bcred. At every point in such a proof, we check whether the atomic goal A matches the head of F. If such a match never occurs, we can drop the assumption F from every HH sequent.

We formalize this intuition to establish independence for STLC in terms of a kind of strengthening lemma in $\mathcal{G}$. Concretely, by the adequacy of the encoding of STLC in HH and the adequacy of the encoding of HH in $\mathcal{G}$, τ_2 is independent of τ_1 in Γ if the following strengthening lemma is provable in $\mathcal{G}$:

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], [\![\tau_1]\!]\ x \vdash [\![\tau_2]\!]\ (t\ x)\} \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash [\![\tau_2]\!]\ t'\}.$$

Note that the variables in the context Γ become nominal constants of appropriate types which are absorbed into formulas. The term $(t\ x)$ indicates the *possible* dependence of t on x. The conclusion $t = (\lambda y.\, t')$ asserts that t is $\beta\eta$-convertible to a term with a vacuous abstraction; this is indicated by the fact that t' is bound outside the scope of λ. To prove any instance of this lemma for particular types τ_1 and τ_2, we proceed by induction on the only assumption. The conclusion $\exists t'.\, t = (\lambda y.\, t')$ is immediately satisfied when the atomic goals in the derivation do not match $[\![\tau_1]\!]\ x$, since this is the only hypothesis where x occurs.

As an example, `ty` is independent of `tm` in $[\![\Gamma]\!]$ which contains the clauses in Figure 2 is formalized as the following lemma:

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t\ x)\} \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t'\}$$

By induction on the assumption and introducing the goals, we get an inductive hypothesis

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t\ x)\}^* \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t'\}$$

and a new hypothesis $\{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t\ x)\}^{@}$. Unfolding this hypothesis amounts to analyzing all possible ways to derive this using the definitional clauses of Figure 3. Since $\hat{\texttt{ty}}\ (t\ x)$ cannot match $\hat{\texttt{tm}}\ x$, the selected focus cannot be $\hat{\texttt{tm}}\ x$, so the only options are clauses selected from $[\![\Gamma]\!]$, of which only two clauses have heads compatible with $\hat{\texttt{ty}}\ (t\ x)$. In the first case, for the clause $\hat{\texttt{ty}}\ \texttt{b}$, we unite $t\ x$ with $\texttt{b}$, instantiating the eigenvariable t with $\lambda y.\, \texttt{b}$; this in turn gives us the witness for t' to finish the proof. In the other case, $t\ x$ is united with $\texttt{arr}\ t_1\ t_2$ (for fresh eigenvariables t_1 and t_2). Here, t_1 and t_2 are first *raised* over x to make the two terms have the same nominal constants, and bcred then reduces the hypothesis to the pair of hypotheses, $\{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t_1\ x)\}^*$ and $\{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t_2\ x)\}^*$. The inductive hypothesis applies to both of these, so we conclude that $\{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t_1'\}^*$ and $\{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t_2'\}^*$ for suitable t_1' and t_2' that are independent of x. We can then finish the proof by forward-chaining on the definitional clauses for HH. Observe that this proof follows the informal proof described in Section 4.2

As another example, let's see why `tm` is not independent of `ty`, since abstractions contain their argument types. The relevant lemma would be:

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], \hat{\texttt{ty}}\ x \vdash \hat{\texttt{tm}}\ (t\ x)\} \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash \hat{\texttt{tm}}\ t'\}.$$

Here, a direct induction on the assumption would not work because we can now focus on the fourth clause of Figure 2 that would extend the context of the HH sequent with a fresh assumption of the form $\hat{\texttt{tm}}\ x'$. The inductive hypothesis is prevented from being used because the context has *grown*. This is a standard feature of HH derivations, and we therefore need to give an inductive characterization of the structure of the *dynamically growing* context. We use the technique of defining these dynamic extensions in terms of *context definitions*; for example, for the $\hat{\texttt{tm}}$ predicate, this definition has the following clauses:

$$\texttt{ctx}\ \texttt{nil} \triangleq \top; \qquad (\nabla x.\, \texttt{ctx}\ (\hat{\texttt{tm}}\ x :: L)) \triangleq \texttt{ctx}\ L.$$

Then we generalize the lemmas as follows:

$$\forall t, L. \nabla x. \texttt{ctx}\ L \supset \{[\![\Gamma]\!], L, \hat{\texttt{ty}}\ x \vdash \hat{\texttt{tm}}\ (t\ x)\} \supset \exists t'. t = (\lambda y. t') \wedge \{[\![\Gamma]\!], L \vdash \hat{\texttt{tm}}\ t'\}.$$

Now, by induction, when the fourth clause of Figure 2 is selected for focus, one of the hypotheses of the resulting bcred is $\{[\![\Gamma]\!], L, \hat{\texttt{ty}}\ x \vdash \hat{\texttt{ty}}\ (t_1\ x)\}$ (for a new eigenvariable t_1). It is entirely possible that $\hat{\texttt{ty}}\ (t_1\ x)$ is proved by selecting $\hat{\texttt{ty}}\ x$. Thus we cannot conclude the original assumption does not depend on $\hat{\texttt{ty}}\ x$ – so tm is not independent of ty.

6.2 Automatically Deriving Strengthening Lemmas

The above illustrations show that these strengthening lemmas have a predictable form and proof structure. We will now give an algorithm that extracts these proofs automatically. The key insight is that the strengthening lemmas can be provable because of a *failure* to match the heads of the encoded clauses against the right hand sides of the HH sequent. Therefore, we simply need to enumerate the possible forms of the right hand sides and generate a mutually inductive lemma to cover all possibilities. This intuition is not entirely trivial to implement, since the HH contexts can potentially grow on every bcred step, which must then be accounted for.

For $F = (\Pi\bar{x}. F_1 \Rightarrow \ldots \Rightarrow F_n \Rightarrow A)$, let $\mathcal{H}_p(F)$ stand for the head predicate in A and $\mathcal{L}_p(F)$ for $\{\mathcal{H}_p(F_i) \mid 1 \in i..n\}$. To prove $\forall\bar{x}. \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$, we proceed as follows:

1. For every predicate a in Γ, we compute the possible dynamic contexts that arise in proofs of atomic formulas of head a.
2. For every predicate a in Γ and A, we compute a collection of predicates $S(a)$ which contains the head predicates of atomic formulas that may occur as the goal in a derivation starting with an atomic goal formula A' for which $\mathcal{H}_p(A') = a$. That is, the provability of any goal A' of head a only depends on formulas whose heads are in $S(a)$.
3. If $\mathcal{H}_p(A) = a$ and $\mathcal{H}_p(F) \notin S(a)$, then for every predicate $a' \in S(a)$ and any atomic goal A' *s.t.* $\mathcal{H}_p(A') = a'$, $\forall\bar{x}. \{\Gamma, F \vdash A'\} \supset \{\Gamma \vdash A'\}$ is provable. The proof proceeds by a simultaneously induction on all these formulas.
4. Since $a \in S(a)$, the required lemma is just one of the cases of the simultaneous induction.

Before we elaborate on these steps in the following subsections, note that our algorithm is sound and terminating, but not complete. The existence of a decision procedure for strengthening lemmas is outside the scope of this paper.

6.2.1 Calculating the dynamic contexts

Let Γ be a context that contains only finite distinct clauses, A be an atomic goal and F be some program clause. We would like to prove the strengthening lemma $\forall\bar{x}. \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$ in $\mathcal{G}$. Let Δ be the set of predicates occurring in Γ. For every predicate $a \in \Delta$, let $C(a)$ denote a set of formulas that can possibly occur in the dynamic contexts of atomic formulas of head a. The only way formulas can be introduced into dynamic contexts is by applying bcred. Given a program clause $\Pi\bar{x}. G_1 \Rightarrow \cdots \Rightarrow G_n \Rightarrow A$, the dynamic context of $\mathcal{H}(G_i)$ is obtained by extending the context of A with program clauses in $\mathcal{L}(G_i)$ for $1 \leq i \leq n$. Algorithm 1 derives a set of constraints $\mathcal{C}$ on C based on this observation. It traverses all the program clauses and their sub-program clauses in formulas in Γ and collects a set $\mathcal{C}$ of constraints for dynamic contexts which must be satisfied by derivations starting with Γ as the context.

To compute a set of dynamic contexts that satisfies $\mathcal{C}$, we start with $C(a) = \emptyset$ for all $a \in \Delta$ and iteratively apply the constraint equations until the constraint is satisfied. It is

Algorithm 1 Collecting constraints on dynamic contexts

Let Γ' be a finite set equal to Γ and $\mathcal{C} \leftarrow \emptyset$
while $\Gamma' \neq \emptyset$ **do**
  pick some $D = (\Pi \bar{x}.\, G_1 \Rightarrow \cdots \Rightarrow G_n \Rightarrow A)$ from Γ'
  add equations $\{C(\mathcal{H}_p(G_i)) = C(\mathcal{H}_p(G_i)) \cup C(\mathcal{H}_p(A)) \cup \mathcal{L}(G_i) \mid i \in 1..n\}$ to $\mathcal{C}$
  remove D from Γ' and add clauses in $\bigcup_{i \in 1..n} \mathcal{L}(G_i)$ to Γ'
end while

Algorithm 2 Collecting constraints on the dependency relation

let Γ' be a finite set equal to Γ and $\mathcal{S} \leftarrow \emptyset$
for all $a \in \Delta$ **do**
  for all $D \in \Gamma' \cup C(a)$ where $D = (\Pi \bar{x}.\, G_1 \Rightarrow \ldots \Rightarrow G_n \Rightarrow A)$ and $\mathcal{H}_p(A) = a$ **do**
    add $(S(a) = S(a) \cup \bigcup_{i \in 1..n} S(\mathcal{H}_p(G_i)))$ to $\mathcal{S}$
  end for
end for

easy to see that this algorithm terminates: since the iterations never shrink $C(a)$ for $a \in \Delta$, the only way the algorithm goes on forever is to keep adding new clauses in every iteration. This is impossible since there are only finitely many distinct program clauses. In the end, we get a finite set of dynamic clauses $C(a)$ for every $a \in \Delta$ that satisfies constraints in $\mathcal{C}$. Suppose $C(a) = \{D_1, \ldots, D_n\}$ for $a \in \Delta$. We define the context relation $\mathtt{ctx}_a$ as follows:

$$\mathtt{ctx}_a\ \mathtt{nil} \triangleq \top; \qquad \mathtt{ctx}_a\ (D_1\ \mathtt{::}\ L) \triangleq \mathtt{ctx}_a\ L; \qquad \cdots \qquad \mathtt{ctx}_a\ (D_n\ \mathtt{::}\ L) \triangleq \mathtt{ctx}_a\ L.$$

6.2.2 Generating the dependency relation between predicates

By the bcred rule, given a program clause $\Pi \bar{x}.\, G_1 \Rightarrow \ldots \Rightarrow G_n \Rightarrow A$, the provability of A will depend on the provability of $\mathcal{H}(G_i)$ for $1 \in 1..n$. For any $a \in \Delta \cup \{\mathcal{H}_p(A)\}$, a can proved by backchaining on either some clause in Γ or in the dynamic contexts $C(a)$. Since both Γ and $C(a)$ are known and finite, we can derive a set $S(a)$ containing predicates that a depends on.

The steps for computing S are similar to that for C. First, we derive a set of constraints $\mathcal{S}$, by Algorithm 2. To generate the dependency relations, we start with $S(a) = \{a\}$ for all $a \in \Delta \cup \{\mathcal{H}_p(A)\}$ and iteratively apply the constraint equations until the constraints in $\mathcal{S}$ are satisfied. The algorithm terminates by an analysis similar to the previous one.

6.2.3 Constructing proofs for the strengthening lemmas

Now we are in a position to prove the strengthening theorem $\forall \bar{x}.\, \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$ in $\mathcal{G}$. Since the proof of A may depend on formulas with different heads, we generalize the strengthening lemma to take into account of related predicates.

▶ **Theorem 3.** *Let $S(\mathcal{H}_p(A)) = \{a_1, \ldots, a_n\}$, The generalized strengthening lemma is*

$$(\forall \Gamma', \bar{x}_1.\, \mathtt{ctx}_{a_1} \Gamma' \supset \{\Gamma, \Gamma', F \vdash a_1\ \bar{x}_1\} \supset \{\Gamma, \Gamma' \vdash a_1\ \bar{x}_1\}) \wedge \cdots \wedge$$
$$(\forall \Gamma', \bar{x}_n.\, \mathtt{ctx}_{a_n} \Gamma' \supset \{\Gamma, \Gamma', F \vdash a_n\ \bar{x}_n\} \supset \{\Gamma, \Gamma' \vdash a_n\ \bar{x}_n\})$$

If $\mathcal{H}_p(F) \notin S(\mathcal{H}_p(A))$, then this lemma has a proof in $\mathcal{G}$.

Proof. By simultaneous induction on $\{\Gamma, \Gamma', F \vdash a_i\ \bar{x}_i\}$ for $1 \leq i \leq n$. ◀

The original strengthening lemma is an immediate corollary of Theorem 3:

▶ **Corollary 4.** *If $\mathcal{H}_p(F) \notin S(\mathcal{H}_p(A))$, then $\forall \bar{x}.\, \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$ has a proof in $\mathcal{G}$.* ◀

Since the proofs for Theorem 3 is constructive, we obtain a certifying algorithm to state and prove strengthening lemmas in $\mathcal{G}$ on demand.

6.3 Application

This section describes an application of the formalization of independence: to prune unnecessary dependence on variables in $\mathcal{G}$. When analyzing the terms containing nominal constants in $\mathcal{G}$, it is often necessary to introduce variable dependencies on such constants. As an example, the $\forall$ introduction rule creates a new eigenvariable that is raised over the nominal constants in the principal formula. However, if we can show that the type of the eigenvariable is independent from the types of the nominal constants, then we can suppress this dependency. In Abella, the theorem prover based on $\mathcal{G}$, an ad-hoc algorithm based on checking of subordination relations is used to prune such dependencies, but the exact logical basis of the algorithm has never been adequately formulated.

Now that we can derive independence lemmas, we can recast this pruning of nominal constants in terms of the closed-world reading of types. To give it a formal treatment in the logic, we reflect the closed-world reading of types into the proofs. This can be done by encoding the type theory of HH (which is STLC) into an HH specification as described in Section 4 and then to use the derived strengthening lemmas directly.

As an example, suppose we want to prove the following theorem

$$\forall X, T.\, \mathtt{name}\ X \supset \{\mathtt{of}\ X\ T\} \supset \cdots$$

where X has type $\mathtt{tm}$ and T has type $\mathtt{ty}$. By introducing the assumptions and case analysis on $\mathtt{name}\ X$, X will be unified with a nominal constant n and the dependence of T on n will be introduced, resulting in the hypothesis $\{\mathtt{of}\ n\ (T\ n)\}$. This dependence of T on n is vacuous as we have already seen – types cannot depend on terms in STLC. However, to formally establish the independence, we require T to be a well-formed type, so we need to change the theorem to

$$\forall X, T, L.\, \mathtt{ctx}\ L \supset \{\Gamma, L \vdash \hat{\mathtt{ty}}\ T\} \supset \mathtt{name}\ X \supset \{\mathtt{of}\ X\ T\} \supset \cdots$$

where Γ contains the program clauses described in Figure 2. For the purpose of demonstration, we assume the context definition $\mathtt{ctx}$ contains infinitely many nominal constants of type $\hat{\mathtt{tm}}$, which is defined as follows:

$$\mathtt{ctx}\ \mathtt{nil} \triangleq \top; \qquad (\nabla x.\, \mathtt{ctx}\ (\hat{\mathtt{tm}}\ x :: L)) \triangleq \mathtt{ctx}\ L.$$

We perform the same introduction and case analysis on $\mathtt{name}\ X$ and get hypotheses $\mathtt{ctx}\ (L\ n)$, $\{\Gamma, L\ n \vdash \hat{\mathtt{ty}}\ (T\ n)\}$ and $\{\mathtt{of}\ n\ (T\ n)\}$. By the definition of $\mathtt{ctx}$, when treated as a multiset, $L\ n$ must be equivalent to $(\hat{\mathtt{tm}}\ n :: L')$ where L' does not contain n. Thus $\{\Gamma, L', \hat{\mathtt{tm}}\ n \vdash \hat{\mathtt{ty}}\ (T\ n)\}$ holds. At this point, we can use the algorithm in Section 6.2 to derive and prove the following strengthening lemma

$$\forall T.\, \nabla x.\, \{\Gamma, L', \hat{\mathtt{tm}}\ x \vdash \hat{\mathtt{ty}}\ (T\ x)\} \supset \exists T'.\, T = (\lambda y.\, T') \wedge \{\Gamma, L' \vdash \hat{\mathtt{ty}}\ T'\}.$$

By applying it to $\{\Gamma, L', \hat{\mathtt{tm}}\ n \vdash \hat{\mathtt{ty}}\ (T\ n)\}$, we get $T = \lambda y.\, T'$ for some T' not containing y. Hence $\{\mathtt{of}\ n\ (T\ n)\}$ becomes $\{\mathtt{of}\ n\ T'\}$. Note that we choose a particular definition of $\mathtt{ctx}$ for demonstration; the exact context of well-formed terms can vary in practice, based on the signature. Nevertheless, pruning of nominal constants can always be expressed by proving and applying the strengthening lemmas derived from independence.

7 Related Work

The earliest formulation of dependency that we were able to find is by Miller [9, Definition 11.3], where it is defined in terms of derivability of a certain fragment of strict logic for use in a unification procedure. Although we do not have a proof of this, this notion appears to coincide with the negation of our notion of independence. It is unclear why this definition was never adopted in subsequent work on logical frameworks, but we can speculate that one reason is its inherent cost, since it requires proving an arbitrary theorem of relevant logic. Indeed, using independence in the core unification engine is probably inadvisable if the unification engine is to be trusted.

It is more popular to define subordination relations independently of proof-theory, as has been done for many variants of LF. In [15], Virga proposed a *dependence relation* between types and type families in LF to constraint higher-order rewritings to well-behaved expressions. Later, this relation was popularized as *subordination* and used in the type theory of canonical LF to show that canonical terms of one type τ is not affected by introduction of terms of another type that is not subordinate to τ [17, 7]. The subordination relations in these cases are defined to be strong enough so that the type theory only deals with well-formed instances.

For reasoning applications, in order to move from one context to another, it is often necessary to check if a term of type τ_1 can occur in the normal form of another type τ_2. Traditionally, the complement of subordination has been thought to be the right interpretation of independence. However, it is unclear how exactly it can be translated into evidence in the theory that supports the reasoning. Thus, *ad hoc* algorithms have been developed in systems like Twelf [11], Beluga [12] and Abella [18], which all lack formal definitions.

8 Conclusion and Future Work

We proposed a proof-theoretic characterization of independence in a two-level logic framework, and gave an example of such characterization by encoding the type theory of STLC in the logic HH and interpreting the independence relation as strengthening lemmas in a reasoning logic $\mathcal{G}$. We developed an algorithm to automatically establish the independence relation and strengthening lemmas and showed its application to pruning variable dependence in $\mathcal{G}$.

Interpreting independence as strengthening should be realizable in other logical frameworks that support inductive reasoning. We chose the two-level logic framework because it provides a first-class treatment of contexts, which makes proofs of strengthening lemmas easy. It would be worth investigating a similar formal development in logical frameworks such as Twelf and Beluga where contexts are either implicit or built into the type system.

The characterization of independence can be extended to more sophisticated type theories. Recently, it was shown that the encoding of LF in $\mathcal{G}$ [13] can be used transparently to perform inductive reasoning over LF specifications [14]. We plan to develop a characterization of independence of LF based on this approach, which will formalize the important concept of *world subsumption* for migrating LF meta-theorems between different LF context schema.

Finally, one benefit of a logical characterization that is almost too obvious to state is that it opens up independence to both external validation and user-guidance. In LF, where inhabitation is undecidable, the notion of independence proposed in this paper will generally not be automatically derivable. Presenting the user with unsolved independence obligations may be an interesting interaction mode worth investigating.

Acknowledgements. This work has been partially supported by the NSF Grant CCF-0917140 and the ERC grant ProofCert. The first author has been partially supported by the

Doctoral Dissertation Fellowship from the University of Minnesota, Twin Cities. Opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the University of Minnesota, or Inria.

References

1 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

2 Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In *Conference on Automated Deduction*, volume 449 of *LNAI*, pages 221–235. Springer, 1990.

3 Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.

4 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

5 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.

6 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

7 Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

8 Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

9 Dale Miller. Unification under a mixed prefix. *J. of Symbolic Comp.*, 14(4):321–358, 1992.

10 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

11 Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206. Springer, 1999.

12 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.

13 Zachary Snow, David Baelde, and Gopalan Nadathur. A meta-programming approach to realizing dependently typed logic programming. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.

14 Mary Southern and Kaustuv Chaudhuri. A two-level logic approach to reasoning about typed specification languages. In *34th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 557–569, December 2014.

15 Roberto Virga. *Higher-order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

16 Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In *15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, pages 157–168, September 2013.

17 Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2003. Revised, May 2003.

18 The Abella web-site. `http://abella-prover.org/`, 2015.