21st International Conference on Types for Proofs and Programs

TYPES 2015, May 18–21, 2015, Tallinn, Estonia

Edited by

Tarmo Uustalu

Editor
Tarmo Uustalu
Department of Software Science, Tallinn University of Technology
Akadeemia tee 21B, 12618 Tallinn, Estonia
tarmo@cs.ioc.ee

ACM Classification 1998
F.4.1. Mathematical Logic and Formal Languages: Mathematical Logic – Lambda calculus and related systems

ISBN 978-3-95977-030-9

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/978-3-95977-030-9.

Publication date
March 2018

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.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2015.0

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.

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface
Tarmo Uustalu 0:vii

A Type Theory for Probabilistic and Bayesian Reasoning
Robin Adams and Bart Jacobs 1:1–1:34

Heterogeneous Substitution Systems Revisited
Benedikt Ahrens and Ralph Matthes 2:1–2:23

Towards a Cubical Type Theory without an Interval
Thorsten Altenkirch and Ambrus Kaposi 3:1–3:27

Constrained Polymorphic Types for a Calculus with Name Variables
Davide Ancona, Paola Giannini, and Elena Zucca 4:1–4:29

Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom
Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg 5:1–5:34

Efficient Type Checking for Path Polymorphism
Juan Edi, Andrés Viso, and Eduardo Bonelli 6:1–6:23

A Certified Study of a Reversible Programming Language
Luca Paolini, Mauro Piccolo, and Luca Roversi 7:1–7:21

Functional Kan Simplicial Sets: Non-Constructivity of Exponentiation
Erik Parmann 8:1–8:25

Π-Ware: Hardware Description and Verification in Agda
João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling 9:1–9:27

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editors: Tarmo Uustalu

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

Preface

This volume is the post-proceedings of the 21st International Conference on Types for Proofs and Programs, TYPES 2015, which was held in Tallinn, Estonia, 18–21 May 2015.

The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalized and computer assisted reasoning and computer programming. The meetings from 1990 to 2008 were annual workshops of a sequence of five EU funded networking projects. Since 2009, TYPES has been run as an independent conference series. Prior to the 2015 meeting in Tallinn, TYPES meetings had taken place in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), with post-proceedings published in various outlets and since the Bergen 2011 meeting in LIPIcs.

The TYPES areas of interest include, but are not limited to: foundations of type theory and constructive mathematics; applications of type theory; dependently typed programming; industrial uses of type theory technology; meta-theoretic studies of type systems; proof assistants and proof technology; automation in computer-assisted reasoning; links between type theory and functional programming; formalizing mathematics using type theory.

The TYPES conferences are traditionally of open and informal character. Selection of talks for presentation at the conference is based on short abstracts; reporting work in progress and work presented or published elsewhere is welcome. A formal post-proceedings volume is prepared after the conference. Papers submitted to that must represent unpublished work and are subjected to a full review process; the call for papers is open and not restricted to the authors and presentations of the conference.

The programme of TYPES 2015 included three invited talks by Gilles Barthe (IMDEA Software Institute), Andrej Bauer (University of Ljubljana) and Peter Selinger (Dalhousie University) as well as two tutorials by Joachim Kock (Autonomous University of Barcelona) and Peter LeFanu Lumsdaine (Stockholm University). The contributed part of the programme consisted of 35 talks.

TYPES 2015 was sponsored by the European Regional Development Fund through the CoE project EXCS and the ICT R&D project Coinduction.

The call for contributions to the post-proceedings of TYPES 2015 led to the present volume of nine papers documenting conference presentations at TYPES 2015 and other new work by members of the community. Four of the papers concern homotopy type theory while the rest are on a variety of topics: probabilistic reasoning, reversible programming, hardware description, patterns and dynamic rebinding. I thank both the authors and the reviewers for their hard work.

Tarmo Uustalu
Tallinn, June 2017

Organization

Program Committee

Andrea Asperti (Università di Bologna)
Robert Atkey (University of Edinburgh)
Ulrich Berger (Swansea University)
Jean-Philippe Bernardy (Chalmers University of Technology)
Edwin Brady (University of St Andrews)
Joëlle Despeyroux (INRIA Sophia Antipolis – Méditerranée)
Herman Geuvers (Radboud Universiteit Nijmegen)
Sam Lindley (University of Edinburgh)
Assia Mahboubi (INRIA Saclay – Île de France)
Ralph Matthes (IRIT, CNRS & Université Paul Sabatier)
Aleksandar Nanevski (IMDEA Software)
Christine Paulin-Mohring (LRI, Université Paris-Sud)
Simona Ronchi Della Rocca (Università di Torino)
Ulrich Schöpp (Ludwig-Maximilians-Universität München)
Bas Spitters (Carnegie Mellon University *to* Aarhus Universitet)
Pawel Urzyczyn (University of Warsaw)
Tarmo Uustalu (Institute of Cybernetics, Tallinn) (chair)

Organizing Committee

Tiina Laasma
Monika Perkmann
Tarmo Uustalu

Host Institution

Institute of Cybernetics at Tallinn University of Technology

Sponsor

European Regional Development Fund
through the CoE project EXCS and the ICT R&D project Coinduction

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editors: Tarmo Uustalu

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

List of Authors

Robin Adams
Universitetet i Bergen
Bergen, Norway
robin.adams@uib.no

Benedikt Ahrens
Inria Rennes – Bretagne Atlantique
Nantes, France
benedikt.ahrens@inria.fr

Thorsten Altenkirch
University of Nottingham
Nottingham, UK
txa@cs.nott.ac.uk

Davide Ancona
Università di Genova
Genova, Italy
davide.ancona@unige.it

Eduardo Bonelli
Universidad Nacional de Quilmes
Bernal, Argentina
eabonelli@gmail.com

Cyril Cohen
Inria Sophia Antipolis – Mediterranée
Sophia Antipolis, France
cyril.cohen@inria.fr

Thierry Coquand
University of Gothenburg
Gothenburg, Sweden
thierry.coquand@cse.gu.se

Juan Edi
Universidad de Buenos Aires
Buenos Aires, Argentina
jedi@dc.uba.ar

Paola Giannini
Università del Piemonte Orientale,
Alessandria, Italy
paola.giannini@uniupo.it

Simon Huber
University of Gothenburg
Gothenburg, Sweden
simon.huber@cse.gu.se

Bart Jacobs
Radboud Universiteit
Nijmegen, The Netherlands
bart@cs.ru.nl

Ambrus Kaposi
Eötvös Loránd University
Budapest, Hungari
akaposi@inf.elte.hu

Ralph Matthes
IRIT, CNRS & Univ. Paul Sabatier
Toulouse, France
matthes@irit.fr

Andres Mörtberg
Institute for Advanced Study
Princeton, NJ, USA
amortberg@math.ias.edu

Luca Paolini
Università di Torino
Torino, Italy
lpaolini@unito.it

Erik Parmann
Universitetet i Bergen
Bergen, Norway
eparmann@gmail.com

Mauro Piccolo
Università di Torino
Torino, Italy
mrpiccol@gmail.com

João Paulo Pizani Flor
Universiteit Utrecht
Utrecht, The Netherlands
j.p.pizaniflor@uu.nl

Luca Roversi
Università di Torino
Torino, Italy
lroversi@unito.it

Yorick Sijsling
Universiteit Utrecht
Utrecht, The Netherlands
y.sijsling@uu.nl

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editors: Tarmo Uustalu

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

Wouter Swierstra
Universiteit Utrecht
Utrecht, The Netherlands
w.swierstra@uu.nl

Andrés Viso
Universidad de Buenos Aires
Buenos Aires, Argentina
aeviso@gmail.com

Elena Zucca
Università di Genova
Genova, Italy
elena.zucca@unige.it

A Type Theory for Probabilistic and Bayesian Reasoning*

Robin Adams[1] and Bart Jacobs[2]

1 **Institutt for Informatikk, Universitetet i Bergen,**
Postboks 7803, 5020 Bergen, Norway
robin.adams@uib.no

2 **Institute for Computing and Information Sciences, Radboud Universiteit,**
Postbus 9010, 6500 GL Nijmegen, The Netherlands
bart@cs.ru.nl

Abstract

This paper introduces a novel type theory and logic for probabilistic reasoning. Its logic is quantitative, with fuzzy predicates. It includes normalisation and conditioning of states. This conditioning uses a key aspect that distinguishes our probabilistic type theory from quantum type theory, namely the bijective correspondence between predicates and side-effect free actions (called instrument, or assert, maps). The paper shows how suitable computation rules can be derived from this predicate-action correspondence, and uses these rules for calculating conditional probabilities in two well-known examples of Bayesian reasoning in (graphical) models. Our type theory may thus form the basis for a mechanisation of Bayesian inference.

1998 ACM Subject Classification F.4.1 Mathematical Logic and Formal Languages: Mathematical Logic – Lambda calculus and related systems, G.3 Probability and Statistics: Probabilistic algorithms, F.3.1 Logics and Meanings of Programs: Specifying and Verifying and Reasoning about Programs

Keywords and phrases Probabilistic programming, probabilistic algorithm, type theory, effect module, Bayesian reasoning

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.1

1 Introduction

A probabilistic program is understood (semantically) as a stochastic process. A key feature of probabilistic programs as studied in the 1980s and 1990s is the presence of probabilistic choice, for instance in the form of a weighted sum $x +_r y$, where the number $r \in [0, 1]$ determines the ratio of the contributions of x and y to the result. This can be expressed explicitly as a convex sum $r \cdot x + (1 - r) \cdot y$. Some of the relevant sources are [13, 15, 16, 18], and also [22] for the combination of probability and non-determinism. In the language of category theory, a probabilistic program is a map in the Kleisli category of the distribution monad $\mathcal{D}$ (in the discrete case) or of the Giry monad $\mathcal{G}$ (in the continuous case), see [10] for details.

In recent years, with the establishement of Bayesian machine learning as an important area of computer science, the emphasis of probabilistic programming shifted towards conditional inference. The key feature is no longer probabilistic choice, but normalisation of distributions (states), see *e.g.* [3, 7, 21, 14, 19]. Interestingly, this can be done in basically the same

* This work was supported by ERC Advanced Grant QCLS: Quantum Computation, Logic and Security.

© Robin Adams and Bart Jacobs;
licensed under Creative Commons License CC-BY
21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 1; pp. 1:1–1:34

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

underlying models, where a program still produces a distribution – discrete or continuous – over its output.

This paper contributes to this latest line of work by formulating a novel type theory for probabilistic and Bayesian reasoning. We list the key features of our type theory.

- It includes a logic, which is quantitative in nature. This means that its predicates are best understood as 'fuzzy' predicates, taking values in the unit interval $[0,1]$ of probabilities, instead of in the two-element set $\{0,1\}$ of Booleans.
- As a result, the predicates of this logic do not form Boolean algebras, but effect modules (see *e.g.* [9]). The double negation rule $p^{\perp\perp} = p$ does hold, but the sum $\ovee$ is a partial operation. Moreover, there is a scalar multiplication $s \cdot p$, for a scalar s and a predicate p, which produces a scaled version of the predicate p.
- The type theory includes normalisation (and also probabilistic choice). Abstractly, normalisation means that each non-zero 'substate' in the type theory can be turned into a proper state (like in [11]). This involves, for instance, turning a *sub*distribution $\sum_i r_i x_i$, where the probabilities $r_i \in [0,1]$ satisfy $0 < r \leq 1$ for $r \stackrel{\text{def}}{=} \sum_i r_i$, into a proper distribution $\sum_i \frac{r_i}{r} x_i$ – where, by construction, $\sum_i \frac{r_i}{r} = 1$.
- The type theory also includes conditioning, via the combination of assert maps and normalisation (from the previous two points). Hence, we can calculate conditional probabilities inside the type theory, via appropriate (derived) computation rules. In contrast, in the language of [3], probabilistic (graphical) models can be formulated, but actual computations are done in the underlying mathematical models. Since these computations are done inside our calculus, our type theory can form the basis for mechanisation.

This work concentrates on an integrated type theory and logic for probability, and not so much on the underlying semantics (like in [3, 21]) nor on the programming language aspects (like in [7, 19, 14]), since we do not have a 'while' construct, for instance.

The type theory that we present is based on a new categorical foundation for quantum logic, called effectus theory, see [9, 11, 4, 6]. This theory involves a basic duality between states and effects (predicates), which is implicitly also present in our type theory. A subclass of 'commutative' effectuses can be defined, forming models for probabilistic computation and logic. Our type theory corresponds to these commutative effectuses, and will thus be called **COMET**, as an abbreviation for COMmutative Effectus Theory. This system **COMET** can be seen as an internal language for commutative effectuses.

Effectus theory thus forms the categorical basis for **COMET**. At the same time it forms the basis for an embbeded language **EfProb** in the programming language Python[1]. **EfProb** forms a uniform 'calculator' for discrete, continuous and quantum probability. **EfProb** is an unsafe language, which is in a sense orthogonal to the **COMET** type theory.

The idea that predicates come with an associated action is familiar in mathematics. For instance, in a Hilbert space $\mathcal{H}$, a closed subspace $P \subseteq \mathcal{H}$ (a predicate) can equivalently be described as a linear idempotent operator $p \colon \mathcal{H} \to \mathcal{H}$ (an action) that has P as image. We sketch how these predicate-action correspondences also exist in the models that underlie our type theory.

First, in the category **Sets** of sets and functions, a predicate p on a set X can be identified with a subset of X, but also with a 'characteristic' map $p \colon X \to 1+1$, where $1+1 = 2$ is the two-element set. We prefer the latter view. Such a predicate corresponds bijectively to a 'side-effect free' instrument $\mathsf{instr}_p \colon X \to X + X$, namely to:

[1] See the website `efprob.cs.ru.nl` for details.

$$\mathsf{instr}_p(x) \;=\; \begin{cases} \mathsf{inl}\,(x) & \text{if } p(x) = 1 \\ \mathsf{inr}\,(x) & \text{if } p(x) = 0 \end{cases}$$

Here we write $X + X$ for the sum (coproduct), with left and right coprojections (also called injections) $\mathsf{inl}\,(_)\,, \mathsf{inr}\,(_) : X \to X+X$. Notice that this instrument merely makes a left-right distinction, as described by the predicate, but does not change the state x. It is called side-effect free because it satisfies $\nabla \circ \mathsf{instr}_p = \mathrm{id}$, where $\nabla = [\mathrm{id}, \mathrm{id}] \colon X + X \to X$ is the codiagonal. It is easy to see that each map $f \colon X \to X + X$ with $\nabla \circ f = \mathrm{id}$ corresponds to a predicate $p \colon X \to 1 + 1$, namely to $p = (! + !) \circ f$, where $! \colon X \to 1$ is the unique map to the final (singleton, unit) set 1.

Our next example describes the same predicate-action correspondence in a probabilistic setting. It assumes familiarity with the discrete distribution monad $\mathcal{D}$ – see [9] for details, and also Section 4.1 – and with its Kleisli category $\mathcal{K}\ell(\mathcal{D})$. A predicate map $p \colon X \to 1 + 1$ in $\mathcal{K}\ell(\mathcal{D})$ is (essentially) a fuzzy predicate $p \colon X \to [0, 1]$, since $\mathcal{D}(1 + 1) = \mathcal{D}(2) \cong [0, 1]$. There is also an associated instrument map $\mathsf{instr}_p \colon X \to X + X$ in $\mathcal{K}\ell(\mathcal{D})$, given by the function $\mathsf{instr}_p \colon X \to \mathcal{D}(X + X)$ that sends an element $x \in X$ to the distribution (formal convex combination):

$$\mathsf{instr}_p(x) \;=\; p(x) \cdot \mathsf{inl}\,(x) + (1 - p(x)) \cdot \mathsf{inr}\,(x)\,.$$

This instrument makes a left-right distinction, with the weight of the distinction given by the fuzzy predicate p. Again we have $\nabla \circ \mathsf{instr}_p = \mathrm{id}$, in the Kleisli category, since the instrument map does not change the state. It is easy to see that we get a bijective correspondence.

These instrument maps $\mathsf{instr}_p \colon X \to X + X$ can in fact be simplified further into what we call assert maps. The (partial) map $\mathsf{assert}_p \colon X \to X + 1$ can be defined as $\mathsf{assert}_p = (\mathrm{id} + !) \circ \mathsf{instr}_p$. We say that such a map is side-effect free if there is an inequality $\mathsf{assert}_p \leq \mathsf{inl}\,(_)$, for a suitable order on the homset of partial maps $X \to X + 1$. Given assert maps for p, and for its orthosupplement (negation) $p^\perp$, we can define the associated instrument via a partial pairing operation as $\mathsf{instr}_p = \langle\!\langle \mathsf{assert}_p, \mathsf{assert}_{p^\perp} \rangle\!\rangle$, see below for details. We shall define conditioning via normalisation after assert. More specifically, for a state $\omega \colon X$ and a predicate p on X we define the conditional state $\omega|_p = \mathsf{cond}\,(\omega, p)$ as:

$$\mathsf{cond}\,(\omega, p) \;=\; \mathsf{nrm}\,(\mathsf{assert}_p(\omega))\,,$$

where $\mathsf{nrm}\,(-)$ describes normalisation (of substates to states). This description occurs in semantical form in [11]. Here we formalise it at a type-theoretic level and derive suitable computation rules from it that allow us to do (exact) conditional inference.

The paper is organised as follows. Section 2 provides an overview of the type theory, with some key results, without giving all the details and proofs. Section 3 takes two familiar examples of Bayesian reasoning and formalises them in our type theory **COMET**. Next, Section 4 sketches how our type theory can be interpreted in set-theoretic and probabilistic models. Subsequently, Section 5 explores the type theory in greater depth, and provides justification for the computation rules in the examples. Appendix A contains a formal presentation of the type theory **COMET**.

1.1 Previous Work

The system **COMET** is related to the quantum type theory QPEL described in [1]. The two type theories have a common subsystem. The type theory in [1] extends this subsystem with rules for qubits. The theory **COMET** extends the subsystem with new computation rules for the instrument maps, which provides a bijective correspondence between predicates and side-effect free assert maps (see below for details); as well as normalisation, and the scalar constants $1/n$.

A key feature of quantum theory is that observations have a side-effect: measuring a system disturbs it at the quantum level. In order to perform such measurements, each quantum predicate comes with an associated 'measurement' instrument operation which acts on the underlying space. Probabilistic theories also have such instruments . . . but they are side-effect free!

The key aspect of a probabilistic model, in contrast to a quantum model, is that there is a bijective correspondence between:

- predicates $X \to 1 + 1$
- side-effect free instruments $X \to X + X$ – or equivalently, side-effect free assert maps $X \to X + 1$.

2 Syntax and Rules of Deduction

We present here the terms and types of **COMET**. We shall describe the system at a high level here, giving the intuition behind each construction. The complete list of the rules of deduction of **COMET** is given in Appendix A, and the properties that we use are all proved in Section 5.

2.1 Syntax

Assume we are given a set of *type constants* **C**, representing the base data types needed for each example. (These may typically include for instance **nat** and **real**.) Then the types and terms of **COMET** are the following.

$$\begin{aligned}
\text{Type } A, B &::= \mathbf{C} \mid 0 \mid 1 \mid A + B \mid A \otimes B \\
\text{Term } r, s, t &::= x \mid * \mid s \otimes t \mid \mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t \mid \ ¡t \mid \mathsf{inl}\,(t) \mid \mathsf{inr}\,(t) \mid \\
&\quad (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t) \mid «s, t» \mid \mathsf{left}\,(t) \mid \mathsf{instr}_{\lambda x s}(t) \mid 1/n \\
&\quad \mathsf{nrm}\,(t) \mid s \varovee t
\end{aligned}$$

We explain the intended meaning of these terms in the remaining parts of Section 2.

The variables x and y are bound within s in $\mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t$. The variable x is bound within s and y within t in $\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t$, and x is bound within t in $\mathsf{instr}_{\lambda x t}(s)$. We identify terms up to α-conversion (change of bound variable). We write $t[x := s]$ for the result of substituting s for x within t, renaming bound variables to avoid variable capture. We shall write _ for a vacuous bound variable; for example, we write $\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t$ for $\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t$ when x does not occur free in s.

The typing rules for these terms are given in Figure 1. (Note that some of these rules make use of defined expressions, which will be introduced in the sections below. Note also that, in the rules ($1/n$) and (nrm), the n that occurs is a constant natural number, not a term that may contain free variables.)

The computation rules that these terms obey are given in Figure 2.

$$\text{(var)}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{(unit)}\ \frac{}{\Gamma \vdash * : 1} \qquad (\otimes)\ \frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B}{\Gamma, \Delta \vdash s \otimes t : A \otimes B}$$

$$\text{(let)}\ \frac{\Gamma \vdash s : A \otimes B \qquad \Delta, x : A, y : B \vdash t : C}{\Gamma, \Delta \vdash \mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t : C}$$

$$\text{(magic)}\ \frac{\Gamma \vdash t : 0}{\Gamma \vdash\ ¡t : A} \qquad \text{(inl)}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inl}\,(t) : A + B} \qquad \text{(inr)}\ \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{inr}\,(t) : A + B}$$

$$\text{(case)}\ \frac{\Gamma \vdash r : A + B \qquad \Delta, x : A \vdash s : C \qquad \Delta, y : B \vdash t : C}{\Gamma, \Delta \vdash \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t : C}$$

$$\text{(inlr)}\ \frac{\Gamma \vdash s : A + 1 \qquad \Gamma \vdash t : B + 1 \qquad \Gamma \vdash s \downarrow = t \uparrow : \mathbf{2}}{\Gamma \vdash «s, t» : A + B}$$

$$\text{(left)}\ \frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash \mathsf{inl?}\,(t) = \top : \mathbf{2}}{\Gamma \vdash \mathsf{left}\,(t) : A} \qquad \text{(instr)}\ \frac{x : A \vdash t : \mathbf{n} \qquad \Gamma \vdash s : A}{\Gamma \vdash \mathsf{instr}_{\lambda x t}(s) : n \cdot A}$$

$$(1/n)\ \frac{}{\Gamma \vdash 1/n : \mathbf{2}} \qquad \text{(nrm)}\ \frac{\vdash t : A + 1 \qquad \vdash 1/n \leq t \downarrow : \mathbf{2}}{\Gamma \vdash \mathsf{nrm}\,(t) : A}$$

$$(\oslash)\ \frac{\begin{array}{cc} \Gamma \vdash s : A + 1 & \Gamma \vdash t : A + 1 \\ \Gamma \vdash b : (A + A) + 1 & \Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_1(x) = s : A + 1 \\ \multicolumn{2}{c}{\Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_2(x) = t : A + 1} \end{array}}{\Gamma \vdash s \oslash t : A + 1}$$

Figure 1 Typing rules for **COMET**.

Figures 1 and 2 should be understood simultaneously. So the term $«s, t»$ is well-typed if and only if we can type $s : A + 1$ and $t : B + 1$ (using the rules in Figure 1), *and* derive the equation $s \downarrow = t \uparrow$ using the rules in Figure 2.

The full set of rules of deduction for the system is given in Appendix A.

2.2 Affine Type Theory

Note the form of several of the typing rules in Figure 1, including ($\otimes$) and (let). These rules do not allow a variable to be duplicated. In particular, we cannot derive the judgement $x : A \vdash x \otimes x : A \otimes A$. The *contraction* rule does not hold in our type theory – it is not the case in general that, if $\Gamma, x : A, y : B \vdash \mathcal{J}$, then $\Gamma, z : A \vdash \mathcal{J}[x := z, y := z]$. Our theory is thus an *affine* type theory, which is similar to a *linear* type theory (see for example [2]).

The reason is that these judgements do not behave well with respect to substitution. For example, take the computation $x : \mathbf{2} \vdash x \otimes x : \mathbf{2} \otimes \mathbf{2}$. If we apply this computation to the scalar $1/2$, we presumably wish the result to be $\top \otimes \top$ with probability $1/2$, and $\bot \otimes \bot$ with probability $1/2$. But this is not the semantics for the term $\vdash 1/2 \otimes 1/2 : \mathbf{2} \otimes \mathbf{2}$. This term assigns probability $1/4$ to all four possibilities $\top \otimes \top$, $\top \otimes \bot$, $\bot \otimes \top$, $\bot \otimes \bot$.

We discuss this further in Section 4.3

$\mathsf{let}\ x \otimes y = r \otimes s\ \mathsf{in}\ t = t[x := r, y := s]$	$(\beta\otimes)$
$\mathsf{case}\ \mathsf{inl}\,(r)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t = s[x := r]$	$(\beta{+}_1)$
$\mathsf{case}\ \mathsf{inr}\,(r)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t = t[y := r]$	$(\beta{+}_2)$
$\rhd_1(«s, t») = s$	$(\beta\mathrm{inlr}_1)$
$\rhd_2(«s, t») = t$	$(\beta\mathrm{inlr}_2)$
$\mathsf{inl}\,(\mathsf{left}\,(t)) = t$	$(\beta\mathrm{left})$
$\mathsf{left}\,(\mathsf{inl}\,(t)) = t$	$(\eta\mathrm{left})$
$\mathsf{index}\,(\mathsf{instr}_{\lambda x p}(t)) = p[x := t]$	(instr-test)
$\nabla(\mathsf{instr}_{\lambda x p}(t)) = t$	(∇-instr)
If $\nabla(t) = x$ then $\mathsf{instr}_{\lambda x \mathsf{index}(t)}(s) = t[x := s]$	$(\eta\mathrm{instr})$
If $t : 1$ then $* = t$	$(\eta 1)$
If $t : A \otimes B$ then $\mathsf{let}\ x \otimes y = t\ \mathsf{in}\ x \otimes y = t$	$(\eta\otimes)$
If $t : A + B$ then $\mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inl}\,(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y) = t$	$(\eta+)$
If $t : A + B$ then $«\rhd_1(t), \rhd_2(t)» = t$	$(\eta\mathrm{inlr})$
If t is well-typed then $\mathsf{do}\ _ \leftarrow t; \mathsf{return}\ \mathsf{nrm}\,(t) = t$	$(\beta\mathrm{nrm})$
If $t = \mathsf{do}\ _ \leftarrow t; \mathsf{return}\ \rho$ and $1/n \leq t$, then $\rho = \mathsf{nrm}\,(t)$	$(\eta\mathrm{nrm})$
$n \cdot 1/n = \top$	$(n \cdot 1/n)$
If $n \cdot t = \top$ then $t = 1/n$	(divide)
If $\mathsf{do}\ x \leftarrow b; \rhd_1(x) = s$ and $\mathsf{do}\ x \leftarrow b; \rhd_2(x) = t$	
then $s \ovee t = \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x)$	($\ovee$-def)

Figure 2 Computation rules for **COMET**.

2.3 States, Predicates and Scalars

A closed term $\vdash t : A$ will be called a *state* of type A, and intuitively it represents a probability distribution over the elements of A.

A *predicate* on type A is a term p such that $x : A \vdash p : \mathbf{2}$. These shall be the formulas of the logic of **COMET** (see Section 2.7).

The closed terms t such that $\vdash t : \mathbf{2}$ are called *scalars*, and represent the *probabilities* or *truth values* of our system. In our intended semantics for discrete and continuous probabilities, these denote elements of the real interval $[0, 1]$.

Given a state $\vdash t : A$ and a predicate $x : A \vdash p : \mathbf{2}$, we can find the probability that p is true when measured on t; this probability is simply the scalar $p[x := t]$. This validity is written as $t \models p$ in effectus theory [5].

A term $x : A \vdash c : B$ may be understood as a *channel* from A to B. One can do state transformation and predicate transformation along a channel. In the current setting this is done simply via substitution. A state $\vdash s : A$ is transformed into a state $\vdash c[x := s] : B$. In the other direction, a predicate $y : B \vdash p : \mathbf{2}$ is transformed into a predicate on A as $x : A \vdash p[y := c] : \mathbf{2}$. This predicate is the *weakest precondition* of p with respect to c. State and predicate transformation, together with conditioning, are used in [12] to describe learning in a Bayesian context. These same ideas are used in Section 3 below.

$$\text{(order)}\ \frac{\begin{array}{cc}\Gamma \vdash s : A+1 & \Gamma \vdash t : A+1 \\ \Gamma \vdash b : (A+A)+1 & \Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_1(x) = s : A+1 \\ \multicolumn{2}{c}{\Gamma \vdash \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x) = t : A+1}\end{array}}{\Gamma \vdash s \leq t : A+1}$$

Figure 3 Rule for Ordering in **COMET**.

2.4 Empty Type

The typing rule for the term $¡t$ says that from an inhabitant $t : 0$ we can produce an inhabitant $¡t$ in any type A. Intuitively, this says 'If the empty type is inhabited, then every type is inhabited', which is vacuously true.

2.5 Coproducts and Copowers

Since we have the coproduct $A + B$ of two types, we can construct the disjoint union of n types $A_1 + \cdots + A_n$ in the obvious way. We write $\mathsf{in}_1^n(), \ldots, \mathsf{in}_n^n()$ for its constructors; thus, if $a : A_i$ then $\mathsf{in}_i^n(a) : A_1 + \cdots + A_n$. And given $t : A_1 + \cdots + A_n$, we can eliminate it as:

$$\mathsf{case}\ t\ \mathsf{of}\ \mathsf{in}_1^n(x_1) \mapsto t_1 \mid \cdots \mid \mathsf{in}_n^n(x_n) \mapsto t_n \quad .$$

We abbreviate this expression as $\mathsf{case}_{i=1}^n\ t\ \mathsf{of}\ \mathsf{in}_i^n(x_i) \mapsto t_i$.

The term $\mathsf{left}(t)$ is understood as follows. If we have a term $t : A+B$ and we have derived the judgement $\mathsf{inl?}(t) = \top$, then we know that we know that t always has the form $\mathsf{inl}(a)$ for some term $a : A$. We denote this unique term a by $\mathsf{left}(t)$.

We have a similar $\mathsf{right}()$ consturction, but there is no need to give primitive rules for this one, as it can be defined in terms of $\mathsf{left}()$: $\mathsf{right}(t) \stackrel{\text{def}}{=} \mathsf{left}(\mathsf{swap}(t))$, where $\mathsf{swap}(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}(x) \mapsto \mathsf{inr}(x) \mid \mathsf{inr}(y) \mapsto \mathsf{inl}(y)$.

For the special case where all the types are equal, we write $n \cdot A$ for the type $A + \cdots + A$, where there are n copies of A. In category theory, this is known as the nth *copower* of A. (We include the special cases $0 \cdot A \stackrel{\text{def}}{=} 0$ and $1 \cdot A \stackrel{\text{def}}{=} A$.)

The *codiagonal* $\nabla(t) : A$ for $t : n \cdot A$ is defined by $\nabla(t) \stackrel{\text{def}}{=} \mathsf{case}_{i=1}^n\ t\ \mathsf{of}\ \mathsf{in}_i^n(x) \mapsto x$. In particular, whene $n = 2$ and $t : A + A$, then $\nabla(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}(x) \mapsto x \mid \mathsf{inr}(x) \mapsto x$.

We write $\mathbf{n}$ for $n \cdot 1$. We denote the canonical elements by 1, 2, ..., n, via definitions $i \stackrel{\text{def}}{=} \mathsf{in}_i^n(*) : \mathbf{n}$ for $1 \leq i \leq n$. For $t : n \cdot A$, we define $\mathsf{index}(t) \stackrel{\text{def}}{=} \mathsf{case}_{i=1}^n t\ \mathsf{of}\ \mathsf{in}_i^n(_) \mapsto i : \mathbf{n}$.

2.6 Partial Functions

A term of type A is intended to represent a *total* computation, that always terminates and returns a value of type A. We can think of a term of type $A + 1$ as a *partial* computation that may return a value a of type A (by outputting $\mathsf{inl}(a)$) or diverge (by outputting $\mathsf{inr}(*)$). The judgement $s \leq t$ should be understood as: the probability that s returns $\mathsf{inl}(a)$ is at most the probability that t returns $\mathsf{inl}(a)$, for all a. The rule for this ordering relation is given in Figure 3.

We define:

- If $\Gamma \vdash t : A$ then $\Gamma \vdash \mathsf{return}\ t \stackrel{\text{def}}{=} \mathsf{inl}(t) : A + 1$. This program converges with probability 1.
- $\Gamma \vdash \mathsf{fail} \stackrel{\text{def}}{=} \mathsf{inr}(*) : A + 1$. This program diverges with probability 1.

- If $\Gamma \vdash s : A + 1$ and $\Delta, x : A \vdash t : B + 1$ then
  $\Gamma, \Delta \vdash \mathsf{do}\ x \leftarrow s; t \stackrel{\text{def}}{=} \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t \mid \mathsf{inr}\,(_) \mapsto \mathsf{fail}.$

The term $\mathsf{do}\ x \leftarrow s; t$ should be read as the following computation: Run s. If s returns a value, pass this as input x to the computation t; otherwise, diverge.

These constructions satisfy these computation rules:

$$
\begin{array}{lrl}
\text{(do-return)} & \mathsf{do}\ x \leftarrow \mathsf{return}\ s; t & = t[x := s] \\
\text{(do-fail)} & \mathsf{do}\ x \leftarrow \mathsf{fail}; t & = \mathsf{fail} \\
\text{(return-do)} & \mathsf{do}\ x \leftarrow r; \mathsf{return}\ x & = r \\
\text{(fail-do)} & \mathsf{do}\ _ \leftarrow r; \mathsf{fail} & = \mathsf{fail} \\
\text{(do-do)} & \mathsf{do}\ x \leftarrow r; (\mathsf{do}\ y \leftarrow s; t) & = \mathsf{do}\ y \leftarrow (\mathsf{do}\ x \leftarrow r; s); t
\end{array}
$$

This construction also allows us to define *scalar multiplication*. If we are given a scalar $\vdash s : \mathbf{2}$ and a substate $\vdash t : A + 1$, then the result of multiplying or scaling t by s is $\vdash \mathsf{do}\ _ \leftarrow s; t : A + 1$.

2.6.1 Partial Projections

Given $t : A + B$, the *partial projections* $\rhd_1^{AB}(t) : A + 1$ and $\rhd_2^{AB}(t) : B + 1$ are defined by

$$
\begin{array}{l}
\rhd_1^{AB}(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{return}\ x \mid \mathsf{inr}\,(_) \mapsto \mathsf{fail} \\
\rhd_2^{AB}(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \mathsf{fail} \mid \mathsf{inr}\,(x) \mapsto \mathsf{return}\ x
\end{array}
$$

We shall often omit the superscripts A and B.

Given $t : n \cdot A$, the *partial projection* $\rhd_i^n(t) : A + 1$ is defined to be

$$
\rhd_i^n(t) \stackrel{\text{def}}{=} \mathsf{case}_{j=1}^n t\ \mathsf{of}\ \mathsf{in}_j^n\,(x) \mapsto \begin{cases} \mathsf{return}\ x & \text{if } i = j \\ \mathsf{fail} & \text{otherwise} \end{cases}
$$

We shall often omit the superscript n.

2.6.2 Partial Sum

Let $\Gamma \vdash s, t : A + 1$. If these terms have disjoint domains (*i.e.* given any input x and output a, the sum of the probabilities that s and t return a given x is never greater than 1), then we may form the computation $\Gamma \vdash s \ovee t$, the *partial sum* of s and t. The probability that this program converges with output a is the sum of the probability that s returns a, and the probability that t returns a. The definition is given by the rule ($\ovee$-def). We write $n \cdot t$ for the sum $t \ovee \cdots \ovee t$ with n summands. We include the special cases $0 \cdot t \stackrel{\text{def}}{=} \mathsf{fail}$ and $1 \cdot t \stackrel{\text{def}}{=} t$.

With this operation, the partial functions in $A + 1$ form a *partial commutative monoid* (PCM) (see Lemma 13).

2.7 Logic

The type $\mathbf{2} = 1 + 1$ shall play a special role in this type theory. It is the type of *propositions* or *predicates*, and its objects shall be used as the formulas of our logic.

We define $\top \stackrel{\text{def}}{=} \mathsf{inl}\,(*) : \mathbf{2}$ and $\bot \stackrel{\text{def}}{=} \mathsf{inr}\,(*) : \mathbf{2}$. We also define the *orthosupplement* of a predicate p, which roughly corresponds to negation:

$$
p^{\perp} \stackrel{\text{def}}{=} \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \bot \mid \mathsf{inr}\,(_) \mapsto \top
$$

We immediately have that $p^{\perp\perp} = p$, $\top^{\perp} = \bot$ and $\bot^{\perp} = \top$.

The ordering on $\mathbf{2}$ shall play the role of the *derivability* relation in our logic: $p \leq q$ will indicate that q is derivable from p, or that p implies q. The rules for this logic are not the familiar rules of classical or intuitionistic logic. Rather, the predicates over any context form an *effect algebra* (Proposition 16).

2.7.1 n-tests

An *n-test* in a context Γ is an n-tuple of predicates $(p_1, \ldots, p_n)$ on A such that $\Gamma \vdash p_1 \ovee \cdots \ovee p_n = \top : \mathbf{2}$.

Intuitively, this can be thought of as a set of n fuzzy predicates whose probabilities always sum to 1. We can think of this as a test that can be performed on the types of Γ with n possible outcomes; and, indeed, there is a one-to-one correspondence between the n-tests of Γ and the terms of type $\mathbf{n}$ (Lemma 21).

2.7.2 Instrument Maps

Let $x : A \vdash t : \mathbf{n}$ and $\Gamma \vdash s : A$. The term $\mathsf{instr}_{\lambda x t}(s) : n \cdot A$ is interpreted as follows: we read the computation $x : A \vdash t : \mathbf{n}$ as a test on the type A, with n possible outcomes. The computation $\mathsf{instr}_{\lambda x t}(s)$ runs t on (the output of) s, and returns $\mathsf{in}_i^n(s)$, where i is the outcome of the test.

Given an n-test $(p_1, \ldots, p_n)$ on A, we can write a program that tests which of $p_1, \ldots, p_n$ is true of its input, and performs one of n different calculations as a result. We write this program as $\Gamma \vdash \mathsf{measure}\ p_1 \mapsto t_1 \mid \cdots \mid p_n \mapsto t_n$. It will be defined in Definition 24.

If $x : A \vdash p : \mathbf{2}$ and $\Gamma, x : A \vdash s, t : B$, we define $\Gamma, x : A \vdash (\mathsf{if}\ p\ \mathsf{then}\ s\ \mathsf{else}\ t) \stackrel{\text{def}}{=} \mathsf{measure}\ p \mapsto s \mid p^{\perp} \mapsto t : B$. In the case where s and t do not depend on x, we have the following fact (Lemma 26.2): $\mathsf{if}\ p\ \mathsf{then}\ s\ \mathsf{else}\ t = \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto s \mid \mathsf{inr}\,(_) \mapsto t$.

2.7.3 Assert Maps

If $x : A \vdash p : \mathbf{2}$ is a predicate, we define

$$\Gamma \vdash \mathsf{assert}_{\lambda x p}(t) \stackrel{\text{def}}{=} \mathsf{case}\ \mathsf{instr}_{\lambda x p}(t)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{return}\ x \mid \mathsf{inr}\,(_) \mapsto \mathsf{fail} : A + 1$$

The computation $\mathsf{assert}_{\lambda x p}(t)$ is a partial computation with output type A. It tests whether p is true of t; if so, it leaves t unchanged; if not, it diverges. That is, if $p[x := t]$ returns $\top$, the computation converges and returns t; if not, it diverges.

These constructions satisfy the following computation rules (see Section 5.6 below for the proofs).

(assert$\downarrow$) $(\mathsf{assert}_{\lambda x p}(t)) \downarrow = p[x := t]$
(assert-scalar) For a scalar $\vdash s : \mathbf{2}$: $\mathsf{assert}_{\lambda_ s}(*) = \mathsf{instr}_{\lambda_ s}(*) = s : \mathbf{2}$.
(instr+) For $x : A + B \vdash t : \mathbf{n}$:

$$\begin{aligned} \mathsf{instr}_{\lambda x t}(s) = \mathsf{case}\ s\ \mathsf{of}\ & \mathsf{inl}\,(y) \mapsto \mathsf{case}_{i=1}^{n} \mathsf{instr}_{\lambda a.t[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{in}_i^n(z) \mapsto \mathsf{in}_i^n(\mathsf{inl}\,(z)) \\ & \mathsf{inr}\,(y) \mapsto \mathsf{case}_{i=1}^{n} \mathsf{instr}_{\lambda b.t[x:=\mathsf{inl}(b)]}(y)\ \mathsf{of}\ \mathsf{in}_i^n(z) \mapsto \mathsf{in}_i^n(\mathsf{inr}\,(z)) \end{aligned}$$

(assert+) For $x : A + B \vdash p : \mathbf{2}$:

$$\begin{aligned} \mathsf{assert}_{\lambda x p}(t) = \mathsf{case}\ t\ \mathsf{of}\ & \mathsf{inl}\,(x) \mapsto \mathsf{do}\ z \leftarrow \mathsf{assert}_{\lambda a.p[x:=\mathsf{inl}(a)]}(x); \mathsf{return}\ \mathsf{inl}\,(z) \mid \\ & \mathsf{inr}\,(y) \mapsto \mathsf{do}\ z \leftarrow \mathsf{assert}_{\lambda b.p[x:=\mathsf{inr}(b)]}(y); \mathsf{return}\ \mathsf{inr}\,(z) \end{aligned}$$

(instr m) For $x : \mathbf{m} \vdash t : \mathbf{n}$: $\mathsf{instr}_{\lambda x t}(s) = \mathsf{case}^m_{i=1} s \text{ of } i \mapsto \mathsf{case}^n_{j=1} t[x := i] \text{ of } j \mapsto \mathsf{in}^n_j\,(i)$
(assert m) For $x : \mathbf{m} \vdash p : \mathbf{2}$: $\mathsf{assert}_{\lambda x p}(t) = \mathsf{case}^m_{i=1} t \text{ of } i \mapsto \mathsf{if}\ p[x := i]\ \mathsf{then\ return}\ i\ \mathsf{else\ fail}$

In particular, we have $\mathsf{assert}_{\lambda x \mathsf{inl?}(x)}(t) = \rhd_1(t)$ and $\mathsf{assert}_{\lambda x \mathsf{inr?}(x)}(t) = \rhd_2(t)$.

2.7.4 Sequential Conjunction

Given two predicates $x : A \vdash p(x), q(x) : \mathbf{2}$, we can define their *sequential conjunction*

$$x : A \vdash p \,\&\, q \stackrel{\text{def}}{=} \mathsf{do}\ z \leftarrow \mathsf{assert}_{\lambda y p(y)}(x); q(z) : \mathbf{2} \ .$$

The probability of this predicate being true at x is the product of the probabilities of $p(x)$ and $q(x)$. This operation has many of the familiar properties of conjunction – including commutativity – but not all: in particular, we do not have $p \,\&\, p^\perp = \perp$ in all cases. (For example, $1/2 \,\&\, (1/2)^\perp = 1/4$.)

2.7.5 Coproducts

We can define predicates which, given a term $t : A + B$, test which of A and B the term came from. We write these as $\mathsf{inl?}\,(t)$ and $\mathsf{inr?}\,(t)$. (Compare these with the operators $FstAnd$ and $SndAnd$ defined in [9].) They are defined by

$$\mathsf{inl?}\,(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \top \mid \mathsf{inr}\,(_) \mapsto \perp$$
$$\mathsf{inr?}\,(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \perp \mid \mathsf{inr}\,(_) \mapsto \top$$

2.7.6 Kernels

The predicate $\mathsf{inr?}\,()$ is particularly important for partial maps.

Let $\Gamma \vdash t : A + 1$. The *kernel* of the map denoted by t is

$$t\!\uparrow \stackrel{\text{def}}{=} \mathsf{inr?}\,(t) \stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \perp \mid \mathsf{inr}\,(_) \mapsto \top$$

Intuitively, if we think of t as a partial computation, then $t\!\uparrow$ is the proposition 't does not terminate', or the function that gives the probability that t will diverge on a given input.

Its orthosupplement, $(t\!\uparrow)^\perp = \mathsf{inl?}\,(t)$, which we shall also write as $t\!\downarrow$, is also called the *domain predicate* of t, and represents the proposition that t terminates. We note that it is equal to $\mathsf{do}\ _ \leftarrow t; \top$.

2.8 Partial Pairing

The term $«s, t»$ is understood intuitively as follows. We are given two partial computations s and t, and we have derived the judgement $s\!\downarrow = t\!\uparrow$, which tells us that exactly one of s and t converges on any given input. We may then form the computation $«s, t»$ which, given an input x, returns either $s(x)$ or $t(x)$, whichever of the two converges.

2.9 Scalar Constants

The term $1/n$ represents the probability distribution on $\mathbf{2} = \{\top, \perp\}$ which returns $\top$ with probability $1/n$ and $\perp$ with probability $(n-1)/n$. It can be thought of as a coin toss, with a weighted coin that returns heads with probability $1/n$. In other languages it is sometimes written as the two-element distribution $\mathsf{flip}(1/n)$.

From this, we have a representation of the rational numbers between 0 and 1. Let m/n denote the term $1/n \owedge \cdots \owedge 1/n$, where there are m summands. The usual arithmetic of rational numbers can be carried out in our system (see Section 5.9).

2.10 Normalisation

Let $\vdash t : A + 1$. Then t represents a *substate* of A. As long as the probability $t\downarrow$ is non-zero, we can *normalise* this program over the probability of non-termination. The result is the state denoted by $\mathsf{nrm}\,(t)$. Intuitively, the probability that $\mathsf{nrm}\,(t)$ will output a is the probability that t will output $\mathsf{inl}\,(a)$, conditioned on the event that t terminates.

In order to type $\mathsf{nrm}\,(t)$, we must first prove that t has a non-zero probability of terminating by deriving an inequality of the form $1/n \leq t\downarrow$ for some positive integer $n \geq 2$.

If $\vdash t : A$ and $x : A \vdash p : \mathbf{2}$, we write $\mathsf{cond}\,(t, \lambda xp)$ for

$$\mathsf{cond}\,(t, \lambda xp) \stackrel{\text{def}}{=} \mathsf{nrm}\,(\mathsf{assert}_{\lambda xp}(t)) \quad .$$

The term t denotes a computation whose output is given by a probability distribution over A. Then $\mathsf{cond}\,(t, \lambda xp)$ gives the result of normalising that conditional probability distribution with respect to p.

2.10.1 Note

In **COMET**, we only allow normalisation of closed terms, because we have not been able to find a satisfactory way to express that an open term is non-zero for all inputs. For closed terms, this is done by finding a constant $1/n$ which the term exceeds. For open terms, it is possible that there is no n such that $1/n$ is always less than t, if the probabilities of t are all positive with infimum 0.

2.11 Marginalisation

The tensor product of type $A \otimes B$ comes with two *projections*. Given $\Gamma \vdash t : A \otimes B$, define

$$\Gamma \vdash \pi_1(t) \stackrel{\text{def}}{=} \mathsf{let}\ x \otimes _ = t\ \mathsf{in}\ x : A \qquad \Gamma \vdash \pi_2(t) \stackrel{\text{def}}{=} \mathsf{let}\ _ \otimes y = t\ \mathsf{in}\ y : B$$

If t is a state (*i.e.* Γ is the empty context), then $\pi_1(t)$ denotes the result of *marginalising* t, as a probability distribution over $A \otimes B$, to a probability distribution over A.

2.12 Local Definition

In our examples, we shall make free use of *local definition*. This is not a part of the syntax of **COMET** itself, but part of our metalanguage. We write $\mathsf{let}\ x = s\ \mathsf{in}\ t$ for $t[x := s]$. We shall also locally define functions: we write $\mathsf{let}\ f(x) = s\ \mathsf{in}\ t$ for the result of replacing every subterm of the form $f(r)$ with $s[x := r]$ in t.

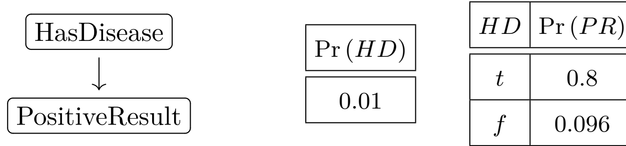
3 Examples

This section describes two examples of (Bayesian) reasoning in our type theory **COMET**. Since this kind of reasoning is not very intuitive, a formal calculus is very useful. The first example is a typical exercise in Bayesian probability theory. The second example involves a simple graphical model.

▶ **Example 1.** (See also [23, 3]) Consider the following situation.

> 1% of a population have a disease. 80% of subjects with the disease test positive, and 9.6% without the disease also test positive. If a subject is positive, what are the odds he/she has the disease?

This situation can be described as a very simple graphical model, with associated (conditional) probabilities.



$\Pr(HD)$
0.01

HD	$\Pr(PR)$
t	0.8
f	0.096

In our type theory **COMET**, we use the following description.

$$\begin{array}{l} \mathsf{let\ subject} = 0.01\ \mathsf{in} \\ \quad \mathsf{let\ positive_result}(x) = (\mathsf{if}\ x\ \mathsf{then}\ 0.8\ \mathsf{else}\ 0.096)\ \mathsf{in} \\ \mathsf{cond}\,(\mathsf{subject}, \mathsf{positive_result}) \end{array}$$

We thus obtain a state $\mathsf{subject} : \mathbf{2}$, conditioned on the predicate $\mathsf{positive_result}$ on $\mathbf{2}$. We calculate the outcome in semi-formal style. The conditional state $\mathsf{cond}\,(\mathsf{subject}, \mathsf{positive_result})$ is defined via normalisation of assert, see Section 2.10. We first calculate what this assert term is:

$$\begin{aligned} \mathsf{assert}_{\lambda x \mathsf{positive_result}(x)}(x) &= \mathsf{if}\ x\ \ \mathsf{then\ if\ positive_result}(\top)\ \mathsf{then\ return}\ \top\ \mathsf{else\ fail} \\ &\qquad\qquad \mathsf{else\ if\ positive_result}(\bot)\ \mathsf{then\ return}\ \bot\ \mathsf{else\ fail} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by (assert } m) \\ &= \mathsf{if}\ x\ \ \mathsf{then\ if}\ 0.8\ \mathsf{then\ return}\ \top\ \mathsf{else\ fail} \\ &\qquad\qquad \mathsf{else\ if}\ 0.096\ \mathsf{then\ return}\ \bot\ \mathsf{else\ fail} \end{aligned}$$

Conditioning requires that the domain of the substate $\mathsf{assert}_{\lambda x \mathsf{positive_result}(x)}(\mathsf{subject})$ is non-zero. We compute this domain as:

$$\begin{aligned} \mathsf{assert}_{\lambda x \mathsf{positive_result}(x)}(\mathsf{subject}){\downarrow} &= \mathsf{positive_result}(\mathsf{subject}) && \text{(Rule (assert}\downarrow\text{))} \\ &= \mathsf{if}\ 0.01\ \mathsf{then}\ 0.8\ \mathsf{else}\ 0.096 \\ &= (0.01\ \&\ 0.8) \ovee (0.99\ \&\ 0.096) && \text{(Lemma 26.2)} \\ &= 0.10304 && \text{(Lemma 28)} \end{aligned}$$

Hence we can choose (e.g.) $n = 10$, to get $\frac{1}{n} \leq 0.10304 = \mathsf{assert}_{\lambda x \mathsf{positive_result}(x)}(\mathsf{subject}){\downarrow}$.

We now proceed to calculate the result, answering the question in the beginning of this example.

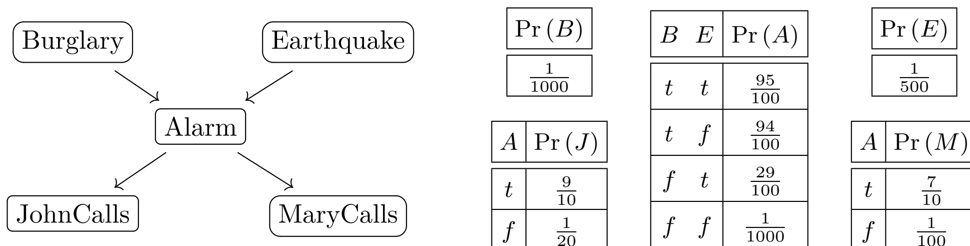
$$
\begin{aligned}
\mathsf{assert}_{\lambda x \mathsf{positive_result}(x)}(\mathsf{subject}) &= \mathsf{if}\ 0.01\ \ \mathsf{then}\ \mathsf{if}\ 0.8\ \mathsf{then}\ \mathsf{return}\ \top\ \mathsf{else}\ \mathsf{fail} \\
&\qquad\qquad \mathsf{else}\ \mathsf{if}\ 0.096\ \mathsf{then}\ \mathsf{return}\ \bot\ \mathsf{else}\ \mathsf{fail} \\
&= \mathsf{measure}\ \ 0.01\ \&\ 0.8 \mapsto \mathsf{return}\ \top \quad \text{(Lemma 25.3)} \\
&\qquad\qquad 0.01\ \&\ 0.8^{\perp} \mapsto \mathsf{fail} \\
&\qquad\qquad 0.01^{\perp}\ \&\ 0.096 \mapsto \mathsf{return}\ \bot \\
&\qquad\qquad 0.01^{\perp}\ \&\ 0.096^{\perp} \mapsto \mathsf{fail} \\
&= \mathsf{measure}\ \ 0.008 \mapsto \mathsf{return}\ \top \quad \text{(Lemma 25.5)} \\
&\qquad\qquad 0.09504 \mapsto \mathsf{return}\ \bot \\
&\qquad\qquad 0.89696 \mapsto \mathsf{fail} \\
\mathsf{cond}\,(\mathsf{subject}, \mathsf{positive_result}) &\stackrel{\text{def}}{=} \mathsf{nrm}\left(\mathsf{assert}_{\lambda x \mathsf{positive_result}(x)}(\mathsf{subject})\right) \\
&= \mathsf{measure}\ 0.0776 \mapsto \top \quad \text{(Corollary 30)} \\
&\qquad\qquad 0.9224 \mapsto \bot \\
&= 0.0776. \quad \text{(Lemma 26.3)}
\end{aligned}
$$

Hence the probability of having the disease after a positive test result is approximately 7.8%.

▶ **Example 2** (Bayesian Network)**.** The following is a standard example of a problem in Bayesian networks, created by [20, Chap. 14].

> I'm at work, neighbor John calls to say my alarm is ringing. Sometimes it's set off by minor earthquakes. Is there a burglar?

We are given that the situation is as described by the following Bayesian network.



$\Pr(B)$
$\frac{1}{1000}$

A	$\Pr(J)$
t	$\frac{9}{10}$
f	$\frac{1}{20}$

B	E	$\Pr(A)$
t	t	$\frac{95}{100}$
t	f	$\frac{94}{100}$
f	t	$\frac{29}{100}$
f	f	$\frac{1}{1000}$

$\Pr(E)$
$\frac{1}{500}$

A	$\Pr(M)$
t	$\frac{7}{10}$
f	$\frac{1}{100}$

The probability of each event given its preconditions is as given in the tables – for example, the probability that the alarm rings given that there is a burglar but no earthquake is 0.94.

We model the above question in **COMET** as follows.

$$
\begin{aligned}
&\mathsf{let}\ b = 0.01\ \mathsf{in}\ \mathsf{let}\ e = 0.002\ \mathsf{in}\\
&\mathsf{let}\ a(x,y) = (\mathsf{if}\ x\ \ \mathsf{then}\ (\mathsf{if}\ y\ \mathsf{then}\ 0.95\ \mathsf{else}\ 0.94)\\
&\qquad\qquad\qquad\quad \mathsf{else}\ (\mathsf{if}\ y\ \mathsf{then}\ 0.29\ \mathsf{else}\ 0.001))\ \mathsf{in}\\
&\mathsf{let}\ j(z) = (\mathsf{if}\ z\ \mathsf{then}\ 0.9\ \mathsf{else}\ 0.05)\ \mathsf{in}\\
&\mathsf{let}\ m(z) = (\mathsf{if}\ z\ \mathsf{then}\ 0.7\ \mathsf{else}\ 0.01)\ \mathsf{in}\\
&\pi_1\big(\mathsf{cond}\,(b \otimes e, j \circ a)\,\big)
\end{aligned}
$$

We first elaborate the predicate $j \circ a$, given in context as $x\colon \mathbf{2}, y\colon \mathbf{2} \vdash j(a(x,y))\colon \mathbf{2}$. It is:

$$
\begin{aligned}
j(a(x,y)) &= \mathsf{if}\ a(x,y)\ \mathsf{then}\ 0.90\ \mathsf{else}\ 0.05\\
&= \mathsf{if}\ x\ \ \mathsf{then}\ (\mathsf{if}\ y\ \ \mathsf{then}\ (\mathsf{if}\ 0.95\ \mathsf{then}\ 0.90\ \mathsf{else}\ 0.05)\\
&\qquad\qquad\qquad\qquad\ \mathsf{else}\ (\mathsf{if}\ 0.94\ \mathsf{then}\ 0.90\ \mathsf{else}\ 0.05)\\
&\qquad\quad\ \mathsf{else}\ (\mathsf{if}\ y\ \ \mathsf{then}\ (\mathsf{if}\ 0.29\ \mathsf{then}\ 0.90\ \mathsf{else}\ 0.05)\\
&\qquad\qquad\qquad\qquad\ \mathsf{else}\ (\mathsf{if}\ 0.001\ \mathsf{then}\ 0.90\ \mathsf{else}\ 0.05)\\
&= \mathsf{if}\ x\ \ \mathsf{then}\ (\mathsf{if}\ y\ \ \mathsf{then}\ (0.95\ \&\ 0.90) \owedge (0.95^{\perp}\ \&\ 0.05)\\
&\qquad\qquad\qquad\qquad\ \mathsf{else}\ (0.94\ \&\ 0.90) \owedge (0.94^{\perp}\ \&\ 0.05))\\
&\qquad\quad\ \mathsf{else}\ (\mathsf{if}\ y\ \ \mathsf{then}\ (0.29\ \&\ 0.90) \owedge (0.29^{\perp}\ \&\ 0.05)\\
&\qquad\qquad\qquad\qquad\ \mathsf{else}\ (0.001\ \&\ 0.90) \owedge (0.001^{\perp}\ \&\ 0.05))\\
&= \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{if}\ y\ \mathsf{then}\ 0.8575\ \mathsf{else}\ 0.849)\ \mathsf{else}\ (\mathsf{if}\ y\ \mathsf{then}\ 0.2965\ \mathsf{else}\ 0.05085)
\end{aligned}
$$

Let us write $\mathsf{assert}_{j \circ a}$ for $\mathsf{assert}_{\lambda t \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ j(a(x,y))}$. Then the associated assert map is:

$$
\begin{aligned}
\mathsf{assert}_{j \circ a}(b,e) &= \mathsf{measure}\ \ 0.001\ \&\ 0.002\ \&\ 0.8575 \mapsto \mathsf{return}\ \top \otimes \top\\
&\qquad\qquad\quad\ 0.001\ \&\ 0.998\ \&\ 0.849 \mapsto \mathsf{return}\ \top \otimes \bot\\
&\qquad\qquad\quad\ 0.999\ \&\ 0.002\ \&\ 0.2965 \mapsto \mathsf{return}\ \bot \otimes \top\\
&\qquad\qquad\quad\ 0.999\ \&\ 0.998\ \&\ 0.05085 \mapsto \mathsf{return}\ \bot \otimes \bot\\
&\qquad\qquad\quad\ 0.052138976^{\perp} \mapsto \mathsf{fail}\\
&= \mathsf{measure}\ \ 0.000001715 \mapsto \mathsf{return}\ \top \otimes \top\\
&\qquad\qquad\quad\ 0.000847302 \mapsto \mathsf{return}\ \top \otimes \bot\\
&\qquad\qquad\quad\ 0.000592407 \mapsto \mathsf{return}\ \bot \otimes \top\\
&\qquad\qquad\quad\ 0.050697552 \mapsto \mathsf{return}\ \bot \otimes \bot\\
&\qquad\qquad\quad\ 0.052138976^{\perp} \mapsto \mathsf{fail}
\end{aligned}
$$

Hence by Corollary 30 we obtain the marginalised conditional:

$$
\begin{aligned}
\pi_1\big(\mathsf{cond}\,(b \otimes e, j \circ a)\big) &= \pi_1\big(\mathsf{nrm}\,(\mathsf{assert}_{j\circ a}(b,e))\big) \\
&= \pi_1\big(\mathsf{measure}\ \ {}^{0.000001715}\!/_{0.052138976} \mapsto \top \otimes \top \\
&\qquad\qquad\qquad {}^{0.000847302}\!/_{0.052138976} \mapsto \top \otimes \bot \\
&\qquad\qquad\qquad {}^{0.000592407}\!/_{0.052138976} \mapsto \bot \otimes \top \\
&\qquad\qquad\qquad {}^{0.050697552}\!/_{0.052138976} \mapsto \bot \otimes \bot\big) \\
&= \mathsf{measure}\ \ 0.000032893 \mapsto \pi_1(\top \otimes \top) \\
&\qquad\qquad 0.016250837 \mapsto \pi_1(\top \otimes \bot) \\
&\qquad\qquad 0.011362078 \mapsto \pi_1(\bot \otimes \top) \\
&\qquad\qquad 0.972354194 \mapsto \pi_1(\bot \otimes \bot) \\
&= \mathsf{measure}\ \ 0.000032893 \mapsto \top \\
&\qquad\qquad 0.016250837 \mapsto \top \\
&\qquad\qquad 0.011362076 \mapsto \bot \\
&\qquad\qquad 0.972354194 \mapsto \bot \\
&= \mathsf{measure}\ \ 0.01628373 \mapsto \top \\
&\qquad\qquad 0.98371627 \mapsto \bot \\
&= 0.01628373
\end{aligned}
$$

We conclude that there is an approximately 1.6% chance of a burglary when John calls.

4 Semantics

The terms of **COMET** are intended to represent probabilistic programs. We show how to give semantics to our system using discrete probability distributions.

4.1 Discrete Probabilistic Computation

We give an interpretation that assigns, to each term, a discrete probability distribution over its output type.

▶ **Definition 3.** Let A be a set.

- The *support* of a function $\phi : A \to [0,1]$ is $\mathrm{supp}\,\phi = \{a \in A : \phi(a) \neq 0\}$.
- A *(discrete) probability distribution* over A is a function $\phi : A \to [0,1]$ with finite support such that $\sum_{a \in A} \phi(a) = 1$.
- Let $\mathcal{D}A$ be the set of all probability distributions on A.

We shall interpret every type A as a set $[\![A]\!]$. Assume we are given a set $[\![\mathbf{C}]\!]$ for each type constant $\mathbf{C}$. Define a set $[\![A]\!]$ for each type A thus:

$$[\![0]\!] = \emptyset \qquad [\![1]\!] = \{*\} \qquad [\![A + B]\!] = [\![A]\!] \uplus [\![B]\!] \qquad [\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!]$$

where $A \uplus B = \{\kappa_1(a) : a \in A\} \cup \{\kappa_2(b) : b \in B\}$. We extend this to contexts by defining $[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$.

Now, to every term $\Gamma \vdash t : B$, where $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, we assign a function $[\![t]\!] : [\![\Gamma]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to \mathcal{D}\,[\![B]\!]$. The value $[\![t]\!]\,(g_1, \ldots, g_n)(b) \in [0,1]$ will be written

$$
\begin{aligned}
P(x_i(\vec{g}) = a) &= \begin{cases} 1 \text{ if } a = g_i \\ 0 \text{ if } a \neq g_i \end{cases} \\
P(*(\vec{g}) = *) &= 1 \\
P((s \otimes t)(\vec{g}, \vec{d}) = (a, b)) &= P(s(\vec{g}) = a)P(t(\vec{d}) = b) \\
P((\mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t)(\vec{g}, \vec{d}) = c) &= \textstyle\sum_a \sum_b P(s(\vec{g}) = (a, b))P(t(\vec{d}, a, b) = c) \\
P((\text{¡}\, t)(\vec{g}) = a) &= 0 \\
P(\mathsf{inl}\,(t)\,(\vec{g}) = \kappa_1(a)) &= P(t(\vec{g}) = a) \\
P(\mathsf{inl}\,(t)\,(\vec{g}) = \kappa_2(b)) &= 0 \\
P(\mathsf{inr}\,(t)\,(\vec{g}) = \kappa_1(a)) &= 0 \\
P(\mathsf{inr}\,(t)\,(\vec{g}) = \kappa_2(b)) &= P(t(\vec{g}) = b) \\
((\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t)(\vec{g}, \vec{d}) = c) & \\
&= \textstyle\sum_a P(r(\vec{g}) = \kappa_1(a))P(s(\vec{d}, a) = c) + \\
&\quad \textstyle\sum_b P(r(\vec{g}) = \kappa_2(b))P(t(\vec{d}, b) = c) \\
P(«s, t»(\vec{g}) = \kappa_1(a)) &= P(s(\vec{g}) = \kappa_1(a)) \\
P(«s, t»(\vec{g}) = \kappa_2(b)) &= P(t(\vec{g}) = \kappa_1(b)) \\
P(\mathsf{left}\,(t)\,(\vec{g}) = a) &= P(t(\vec{g}) = \kappa_1(a)) \\
P(\mathsf{instr}_{\lambda xt}(s)(\vec{g}) = \kappa_i(a)) &= P(s(\vec{g}) = a)P(t(a) = \kappa_i(*)) \\
P(1/n(\vec{g}) = \kappa_1(*)) &= 1/n \\
P(1/n(\vec{g}) = \kappa_2(*)) &= (n-1)/n \\
P(\mathsf{nrm}\,(t)\,(\vec{g}) = a) &= P(t(\vec{g}) = \kappa_1(a))/(1 - P(t(\vec{g}) = \kappa_2(*))) \\
P((s \ovee t)(\vec{g}) = \kappa_1(a)) &= P(s(\vec{g}) = \kappa_1(a)) + P(t(\vec{g}) = \kappa_1(a)) \\
P((s \ovee t)(\vec{g}) = \kappa_2(*)) &= P(s(\vec{g}) = \kappa_2(*)) + P(t(\vec{g}) = \kappa_2(*)) - 1
\end{aligned}
$$

Figure 4 Semantics for **COMET** in $\mathcal{K}\ell(\mathcal{D})$.

as $P(t(g_1, \ldots, g_n) = b)$, and should be thought of as the probability that b will be the output if $g_1, \ldots, g_n$ are the inputs. The clauses are given in Figure 4.

The sums involved here are all well-defined because the function $P(t(\vec{g}) = -)$ has finite support for all t, $\vec{g}$.

▶ **Example 4** (Assert Maps). This definition gives the following semantics to the assert maps. Recall that we define

$$\Gamma \vdash \mathsf{assert}_{\lambda xp}(t) \stackrel{\text{def}}{=} \rhd_1(\mathsf{instr}_{\lambda xp}(t)) : A + 1 \ ,$$

where $\Gamma \vdash t : A$ and $x : A \vdash p : \mathbf{2}$. We therefore have

$$
\begin{aligned}
P(\mathsf{assert}_{\lambda xp}(t)(\vec{g}) = \kappa_1(a)) &= P(t(\vec{g}) = a)P(p(a) = \kappa_1(*)) \\
P(\mathsf{assert}_{\lambda xp}(t)(\vec{g}) = \kappa_2(*)) &= \sum_{a \in A} P(t(\vec{g}) = a)P(p(a) = \kappa_2(*))
\end{aligned}
$$

▶ **Theorem 5** (Soundness).

1. *If* $\Gamma \vdash t : A$ *is derivable, then for all* $\vec{g} \in [\![\Gamma]\!]$*, we have* $P(t(\vec{g}) = -)$ *is a probability distribution on* $[\![A]\!]$*.*
2. *If* $\Gamma \vdash s = t : A$*, then* $P(s(\vec{g}) = a) = P(t(\vec{g}) = a)$*.*

Proof. The proof is by induction on derivations. First prove $P(t[x := s](\vec{g}, \vec{a}) = b) = \sum_{a \in [\![A]\!]} P(s(\vec{g}) = a) P(t(\vec{d}, a) = b)$ whenever $t[x := s]$ is well-typed. ◀

As a corollary, we know that **COMET** is non-degenerate:

▶ **Corollary 6.** *Not every judgement is derivable; in particular, the judgement* $\vdash \top = \bot : \mathbf{2}$ *is not derivable.*

4.2 Alternative Semantics

It is also possible to give semantics to **COMET** using continuous probabilities. We assign a measurable space $[\![A]\!]$ to every type A. Each term then gives a measurable function $[\![A_1]\!] \times \cdots \times [\![A_n]\!] \to \mathcal{G}\,[\![B]\!]$, where $\mathcal{G}X$ is the space of all probability distributions over the measurable space X. ($\mathcal{G}$ here is the *Giry monad* [8, 10].)

If we remove the constants $1/n$ from the system, we can give *deterministic* semantics to the subsystem, in which we assign a set to every type, and a function $[\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![B]\!]$ to every term.

More generally, we can give an interpretation of **COMET** in any *commutative monoidal effectus with normalisation* in which there exists a scalar s such that $n \cdot s = 1$ for all positive integers n [5]. The three ways of giving semantics to **COMET** that we have described are three instances of this interpretation.

4.3 Note on Affine Type Theory

The diagonals or copiers in $\mathcal{K}\ell(\mathcal{D})$ are not natural. It is easy to see that the only arrow $\delta_A : A \to A \otimes A$ that satisfies

$$\pi_1 \circ \delta_A = \pi_2 \circ \delta_A = \mathrm{id}\,A$$

is given by $\delta_A(a) = 1|(a,a)\rangle$; that is, $\delta_A(a)$ gives probability 1 to (a,a), and probability 0 to all other pairs.

However, this family of arrows δ_A is not natural in A. Let $f : A \to B$ be any morphism in $\mathcal{K}\ell(\mathcal{D})$.

$$\begin{array}{ccc} A & \xrightarrow{\delta} & A \otimes A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f \otimes f} \\ B & \xrightarrow[\delta]{} & B \otimes B \end{array}$$

We have

$$((f \otimes f) \circ \delta_A)(a)(b, b') = f(a)(b) \cdot f(a)(b')$$

$$(\delta_B \circ f)(a)(b, b') = \begin{cases} f(a)(b) & \text{if } b = b' \\ 0 & \text{if } b \neq b' \end{cases}$$

There is therefore no way to give semantics to a type theory with contraction in $\mathcal{K}\ell(\mathcal{D})$ in such a way that the following substitution property holds, which was needed for the proof of the Soundness Theorem.

$P(t[x := s](\vec{g}, \vec{a}) = b) = \sum_{a \in [\![A]\!]} P(s(\vec{g}) = a) P(t(\vec{d}, a) = b)$ whenever $t[x := s]$ is well-typed.

In particular, the rule ($\beta\otimes$) becomes unsound. For our semantics give:

$$
\begin{aligned}
& P((\mathsf{let}\ x \otimes y = 1/2 \otimes * \ \mathsf{in}\ x \otimes x)(\vec{g}) = (b_1, b_2)) \\
&= \sum_b P(1/2(\vec{g}) = b) P((x \otimes x)(b) = (b_1, b_2)) \\
&= \sum_b P(1/2(\vec{g}) = b) P(x(b) = b_1) P(x(b) = b_2) \\
&= \begin{cases} 1/2 & \text{if } b_1 = b_2 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

and so $\mathsf{let}\ x \otimes y = 1/2 \otimes * \ \mathsf{in}\ x \otimes x$ and $1/2 \otimes 1/2$ receive different semantics.

5 Metatheorems

We presented an overview of the system in Section 2, and gave the intuitive meaning of the terms of **COMET**. In this section, we proceed to a more formal development of the theory, and investigate what can be proved within the system.

The type theory we have presented enjoys the following standard properties.

▶ **Lemma 7.**
1. ***Weakening*** *If* $\Gamma \vdash \mathcal{J}$ *and* $\Gamma \subseteq \Delta$ *then* $\Delta \vdash \mathcal{J}$.
2. ***Substitution*** *If* $\Gamma \vdash t : A$ *and* $\Delta, x : A \vdash \mathcal{J}$ *then* $\Gamma, \Delta \vdash \mathcal{J}[x := t]$.
3. ***Equation Validity*** *If* $\Gamma \vdash s = t : A$ *then* $\Gamma \vdash s : A$ *and* $\Gamma \vdash t : A$.
4. ***Inequality Validity*** *If* $\Gamma \vdash s \leq t : A + 1$ *then* $\Gamma \vdash s : A + 1$ *and* $\Gamma \vdash t : A + 1$.
5. ***Functionality*** *If* $\Gamma \vdash r = s : A$ *and* $\Delta, x : A \vdash t : B$ *then* $\Gamma, \Delta \vdash t[x := r] = t[x := s] : B$.

Proof. The proof in each case is by induction on derivations. Each case is straightforward.

(Note in particular the form that the rule (nrm) takes. Even though we only apply normalisation to states, we allow an arbitrary context in the conclusion so that the Weakening property shall hold.) ◀

The following lemma shows that substituting within our binding operations works as desired.

▶ **Lemma 8.** 1. *If* $\Gamma \vdash r : A \otimes B$*;* $\Delta, x : A, y : B \vdash s : C$*; and* $\Theta, z : C \vdash t : D$ *then* $\Gamma, \Delta, \Theta \vdash t[z := \mathsf{let}\ x \otimes y = r\ \mathsf{in}\ s] = \mathsf{let}\ x \otimes y = r\ \mathsf{in}\ t[z := s] : D$.
2. *If* $\Gamma \vdash r : A + B$*;* $\Delta, x : A \vdash s : C$*;* $\Delta, y : B \vdash s' : C$*; and* $\Theta, z : C \vdash t : D$ *then*

$$
\begin{aligned}
\Gamma, \Delta, \Theta \vdash\ & t[z := \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto s'] \\
&= \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t[z := s] \mid \mathsf{inr}\,(y) \mapsto t[z := s'] : D
\end{aligned}
$$

Proof. For part 1, we us the following 'trick' to simulate local definition (see [1]):

$$
\begin{aligned}
& t[z := \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto s'] \\
&= \mathsf{let}\ z \otimes _ = (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto s') \otimes * \ \mathsf{in}\ t && (\beta\otimes) \\
&= \mathsf{let}\ z \otimes _ = \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \otimes * \mid \mathsf{inr}\,(y) \mapsto s' \otimes * \ \mathsf{in}\ t && (\text{case-}\otimes) \\
&= \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{let}\ z \otimes _ = s \otimes * \ \mathsf{in}\ t \mid \mathsf{inr}\,(y) \mapsto \mathsf{let}\ z \otimes _ = s' \otimes * \ \mathsf{in}\ t && (\text{let-case}) \\
&= \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t[z := s] \mid \mathsf{inr}\,(y) \mapsto t[z := s'] && (\beta\otimes)
\end{aligned}
$$

Part 2 is proven similarly using (let-$\otimes$) and (let-let) . ◀

▶ **Corollary 9.**
1. *If* $\Gamma \vdash s : A \otimes B$ *and* $\Delta \vdash t : C$ *then* $\Gamma, \Delta \vdash \mathsf{let}\ _ \otimes _ = s\ \mathsf{in}\ t = t : C$.
2. *If* $\Gamma \vdash s : A + B$ *and* $\Delta \vdash t : C$ *then* $\Gamma, \Delta \vdash \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto t \mid \mathsf{inr}\,(_) \mapsto t = t : C$.

Proof. These are both the special case where z does not occur free in t. ◀

5.1 Coproducts

We generalise the $\mathsf{inl?}\,()$ and $\mathsf{inr?}\,()$ constructions as follows. Define the predicate $\mathsf{in}_i?\,()$ on $n \cdot A$, which tests whether a term comes from the ith component, as follows.

$$\mathsf{in}_i?\,(t) \stackrel{\text{def}}{=} \mathsf{case}^n_{j=1} t\ \mathsf{of}\ \mathsf{in}^n_j\,(_) \mapsto \begin{cases} \top & \text{if } i = j \\ \bot & \text{if } i \neq j \end{cases}$$

5.2 The right() Construction

▶ **Lemma 10.** *The* $\mathsf{right}\,()$ *construction satisfies analagous rules to the* $\mathsf{left}\,()$ *constructor:*

1. *If* $t : A + B$ *and* $\mathsf{inr?}\,(t) = \top : \mathbf{2}$ *then* $\mathsf{right}\,(t) : B$.
2. *If* $t = t' : A + B$ *and* $\mathsf{inr?}\,(t) = \top$ *then* $\mathsf{right}\,(t) = \mathsf{right}\,(t') : B$.
3. *If* $t : A + B$ *and* $\mathsf{inr?}\,(t) = \top$ *then* $t = \mathsf{inr}\,(\mathsf{right}\,(t)) : A + B$.
4. *If* $t : B$ *then* $t = \mathsf{right}\,(\mathsf{inr}\,(t))$.

Proof. We prove parts 1 and 3 here. Suppose $\mathsf{inr?}\,(t) = \top$. Then we have

$$\begin{aligned}
\mathsf{inl?}\,(\mathsf{swap}\,(t)) &\stackrel{\text{def}}{=} \mathsf{inl?}\,(\mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inr}\,(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inl}\,(y)) \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inl?}\,(\mathsf{inr}\,(x)) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inl?}\,(\mathsf{inl}\,(y)) && \text{(case-case)} \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \bot \mid \mathsf{inr}\,(y) \mapsto \top && (\beta{+}_1), (\beta{+}_2) \\
&\stackrel{\text{def}}{=} \mathsf{inr?}\,(t) \\
&= \top && \text{(by hypothesis)}
\end{aligned}$$

Therefore, $\mathsf{right}\,(t) \stackrel{\text{def}}{=} \mathsf{left}\,(\mathsf{swap}\,(t))$ is well-typed with type B, and

$$\begin{aligned}
\mathsf{inr}\,(\mathsf{right}\,(t)) &= \mathsf{swap}\,(\mathsf{inl}\,(\mathsf{right}\,(t))) && (\beta{+}_1) \\
&\stackrel{\text{def}}{=} \mathsf{swap}\,(\mathsf{inl}\,(\mathsf{left}\,(\mathsf{swap}\,(t)))) \\
&= \mathsf{swap}\,(\mathsf{swap}\,(t)) && (\beta\mathsf{left}) \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{swap}\,(\mathsf{inr}\,(x)) \mid \\
&\quad \mathsf{inr}\,(y) \mapsto \mathsf{swap}\,(\mathsf{inl}\,(y)) && \text{(case-case)} \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inl}\,(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y) && \text{(case-eq)}, (\beta{+}_1), (\beta{+}_2) \\
&= t && (\eta{+})
\end{aligned}$$

◀

5.3 The swap() Operation

▶ **Lemma 11.**
1. *Let* $\Gamma \vdash t : A + B$. *Then*

$$\Gamma \vdash \rhd_1(\mathsf{swap}\,(t)) = \rhd_2(t) : B + 1$$
$$\Gamma \vdash \rhd_2(\mathsf{swap}\,(t)) = \rhd_1(t) : A + 1$$

2. *Let* $\Gamma \vdash t : A + A$. *Then* $\Gamma \vdash \nabla(\mathsf{swap}\,(t)) = \nabla(t) : A$.

Proof. We prove the first of these here. We have

$$\begin{aligned}
\rhd_1(\mathsf{swap}\,(t)) &\stackrel{\text{def}}{=} \mathsf{case}\ (\mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inr}\,(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inl}\,(y))\ \mathsf{of} \\
&\quad \mathsf{inl}\,(y) \mapsto \mathsf{inl}\,(y) \mid \mathsf{inr}\,(_) \mapsto \mathsf{inr}\,(*) \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto (\mathsf{case}\ \mathsf{inr}\,(x)\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl}\,(y) \mid \mathsf{inr}\,(_) \mapsto \mathsf{inr}\,(*)) \mid \\
&\quad \mathsf{inr}\,(y) \mapsto (\mathsf{case}\ \mathsf{inl}\,(y)\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl}\,(y) \mid \mathsf{inr}\,(_) \mapsto \mathsf{inr}\,(*)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(case-case)} \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inr}\,(*) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inl}\,(y) \qquad (\beta{+}_1), (\beta{+}_2) \\
&\stackrel{\text{def}}{=} \rhd_2(t)
\end{aligned}$$

◀

5.4 Kernels

▶ **Lemma 12.**

1. *Let* $\Gamma \vdash t : A + 1$. *Then* $\Gamma \vdash t{\downarrow} = \bot : \mathbf{2}$ *if and only if* $\Gamma \vdash t = \mathsf{fail} : A + 1$.
2. *Let* $\Gamma \vdash s : A + 1$ *and* $\Delta, x : A \vdash t : B + 1$. *Then* $\Gamma, \Delta \vdash (\mathsf{do}\ x \leftarrow s; t){\downarrow} = \mathsf{do}\ x \leftarrow s; t{\downarrow} : \mathbf{2}$.

Proof.

1. We have

$$\begin{aligned}
\mathsf{fail}{\downarrow} &\stackrel{\text{def}}{=} \mathsf{case}\ \mathsf{inr}\,(*)\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \top \mid \mathsf{inr}\,(_) \mapsto \bot \\
&= \bot \qquad\qquad\qquad (\beta{+}_1)
\end{aligned}$$

For the converse, if $t{\downarrow} = \bot$ then

$$\begin{aligned}
t{\uparrow} &\stackrel{\text{def}}{=} \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \bot \mid \mathsf{inr}\,(_) \mapsto \top \\
&= \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \top^{\perp} \mid \mathsf{inr}\,(_) \mapsto \bot^{\perp} \qquad \text{(case-eq)}, (\beta{+}_1), (\beta{+}_2) \\
&= (\mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto \top \mid \mathsf{inr}\,(_) \mapsto \bot)^{\perp} \qquad \text{(Lemma 2)} \\
&\stackrel{\text{def}}{=} t{\downarrow}^{\perp} \\
&= \bot^{\perp} \qquad \text{(case-eq)} \\
&= \top \qquad (\beta{+}_2)
\end{aligned}$$

and so

$$\begin{aligned}
t &= \mathsf{inr}\,(\mathsf{right}\,(t)) \qquad \text{(Lemma 10.3)} \\
&= \mathsf{inr}\,(*) \qquad (\eta 1)
\end{aligned}$$

2. $$\begin{aligned}
&(\mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t \mid \mathsf{inr}\,(_) \mapsto \mathsf{fail}){\downarrow} \\
&= \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t{\downarrow} \mid \mathsf{inr}\,(_) \mapsto \mathsf{fail}{\downarrow} \qquad \text{(Lemma 2)} \\
&= \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t{\downarrow} \mid \mathsf{inr}\,(_) \mapsto \bot \qquad (\beta{+}_1)
\end{aligned}$$

◀

5.5 Ordering on Partial Maps and the Partial Sum

Note that, from the rules ($\ovee$) and ($\ovee$-def) , we have $\Gamma \vdash s \ovee t : A + 1$ if and only if there exists $\Gamma \vdash b : (A + A) + 1$ such that

$$\Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_1(x) = s : A + 1, \qquad \Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_2(x) = t : A + 1 \ ,$$

in which case $\Gamma \vdash s \otimes t = \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x) : A + 1$. We say that such a term b is a *bound* for $s \otimes t$. By the rule (JM) (see Appendix A.12), this bound is unique if it exists.

The set of *partial* maps $A \to B + 1$ between any two types A and B form a *partial commutative monoid* (PCM) with least element fail, as shown by the following results.

▶ **Lemma 13.**
1. *If* $\Gamma \vdash t : A + 1$ *then* $\Gamma \vdash t \otimes \mathsf{fail} = t : A + 1$.
2. *(**Commutativity**) If* $\Gamma \vdash s \otimes t : A+1$ *then* $\Gamma \vdash t \otimes s : A+1$ *and* $\Gamma \vdash s \otimes t = t \otimes s : A+1$.
3. *(**Associativity**)* $\Gamma \vdash (r \otimes s) \otimes t : A + 1$ *if and only if* $\Gamma \vdash r \otimes (s \otimes t) : A + 1$*, in which case* $\Gamma \vdash r \otimes (s \otimes t) = (r \otimes s) \otimes t : A + 1$.

Proof. We prove part 2 here. Let b be a bound for $s \otimes t$. We shall prove that $\mathsf{do}\ x \leftarrow b; \mathsf{return}\ \mathsf{swap}\,(x)$ is a bound for $t \otimes s$. We have

$$\begin{aligned}
&\mathsf{do}\ y \leftarrow \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \mathsf{swap}\,(x); \rhd_1(y) \\
&= \mathsf{do}\ x \leftarrow b; \mathsf{do}\ y \leftarrow \mathsf{return}\ \mathsf{swap}\,(x); \rhd_1(y) && \text{(do-do)} \\
&= \mathsf{do}\ x \leftarrow b; \rhd_1(\mathsf{swap}\,(x)) && \text{(do-return)} \\
&= \mathsf{do}\ x \leftarrow b; \rhd_2(x) && \text{(Lemma 11.1)} \\
&= t && \text{(by hypothesis)}
\end{aligned}$$

Similarly, $\mathsf{do}\ y \leftarrow \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \mathsf{swap}\,(b); \rhd_2(y) = s$.

Now, we have

$$\begin{aligned}
t \otimes s &= \mathsf{do}\ y \leftarrow (\mathsf{do}\ x \leftarrow b; \mathsf{return}\ \mathsf{swap}\,(x)); \mathsf{return}\ \nabla(y) && (\otimes\text{-def}) \\
&= \mathsf{do}\ x \leftarrow b; \mathsf{do}\ y \leftarrow \mathsf{return}\ \mathsf{swap}\,(x); \mathsf{return}\ \nabla(y) && \text{(do-do)} \\
&= \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(\mathsf{swap}\,(x)) && \text{(do-return)} \\
&= \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x) && \text{(Lemma 11.2)} \\
&= s \otimes t && (\otimes\text{-def})
\end{aligned}$$

◀

▶ **Lemma 14.** *Let* $\Gamma \vdash r : A + 1$ *and* $\Gamma \vdash s : A + 1$*. Then* $\Gamma \vdash r \leq s : A + 1$ *if and only if there exists* t *such that* $\Gamma \vdash r \otimes t = s : A + 1$.

Proof. Suppose $r \leq s$. If b is such that $\mathsf{do}\ x \leftarrow b; \rhd_1(x) = r$ and $\mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x) = s$ then take $t = \mathsf{do}\ x \leftarrow b; \rhd_2(x)$. We have $r \otimes t = \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x)$ by ($\otimes$-def), and so $r \otimes t = s$.

Conversely, if $r \otimes t = s$, then inverting the derivation of $\Gamma \vdash r \otimes t : A + 1$ we have that there exists b such that $r = \mathsf{do}\ x \leftarrow b; \rhd_1(x)$, $t = \mathsf{do}\ x \leftarrow b; \rhd_2(x)$ and $s = r \otimes t = \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x)$. Therefore, $r \leq s$ by (order) . ◀

In this case, the bound for $r \otimes t$ will also be called a bound for $r \leq s$.

▶ **Lemma 15.**
1. *If* $\Gamma \vdash s \otimes t : A + 1$ *then* $\Gamma \vdash s \leq s \otimes t : A + 1$ *and* $\Gamma \vdash t \leq s \otimes t : A + 1$.
2. *If* $\Gamma \vdash t : A + 1$ *then* $\Gamma \vdash t \leq t : A + 1$.
3. *If* $\Gamma \vdash t : A + 1$ *then* $\Gamma \vdash \mathsf{fail} \leq t : A + 1$.
4. *If* $\Gamma \vdash r \leq s : A + 1$ *and* $\Gamma \vdash s \leq t : A + 1$ *then* $\Gamma \vdash r \leq t : A + 1$.
5. *If* $\Gamma \vdash r \leq s : A + 1$ *and* $\Gamma \vdash s \otimes t : A + 1$ *then* $\Gamma \vdash r \otimes t \leq s \otimes t : A + 1$.

Proof. Parts 1–4 follow by applying Lemma 14 to the appropriate part of Lemma 13. For part 5, let $r \ovee x = s$. Then $r \ovee x \ovee t = s \ovee t$ and so $r \ovee t \leq s \ovee t$. ◀

On the predicates, we have the following structure, which shows that they form an *effect algebra*. (In fact, they have more structure: they form an *effect module* over the scalars, as we will prove in Proposition 20.)

▶ **Proposition 16.**
1. *If* $\Gamma \vdash p : \mathbf{2}$ *then* $\Gamma \vdash p \ovee p^\perp = \top : \mathbf{2}$.
2. *If* $\Gamma \vdash p \ovee q = \top : \mathbf{2}$ *then* $\Gamma \vdash q = p^\perp : \mathbf{2}$.
3. ***(Zero-One Law)*** *If* $\Gamma \vdash p \ovee \top : \mathbf{2}$ *then* $\Gamma \vdash p = \bot : \mathbf{2}$.
4. $\Gamma \vdash p \ovee q : \mathbf{2}$ *if and only if* $\Gamma \vdash p \leq q^\perp : \mathbf{2}$.
5. *Suppose* $\Gamma \vdash r : A + B$ *and* $\Delta, x : A \vdash s \ovee t : C + 1$ *and* $\Delta, y : B \vdash s' \ovee t' : C + 1$. *Then*

$$\begin{aligned}\Gamma, \Delta \vdash\ & \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \ovee t \mid \mathsf{inr}\,(y) \mapsto s' \ovee t' \\ &= (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto s') \ovee (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto t \mid \mathsf{inr}\,(y) \mapsto t') : C + 1\end{aligned}$$

6. *If* $\Gamma \vdash r : A + 1$ *and* $\Delta, x : A \vdash s \ovee t : B + 1$ *then* $\Gamma, \Delta \vdash \mathsf{do}\ x \leftarrow r; s \ovee t = (\mathsf{do}\ x \leftarrow r; s) \ovee (\mathsf{do}\ x \leftarrow r; t) : B + 1$.

Proof. We prove part 2 here. Let b be a bound for $p \ovee q$. We have

$$\begin{aligned}
\top &= p \ovee q && \text{(by hypothesis)} \\
&= \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x) && (\ovee\text{-def}) \\
&= \mathsf{do}\ x \leftarrow b; \top && (\eta 1) \\
&\overset{\text{def}}{=} b{\downarrow} \\
\therefore b &= \mathsf{return}\ \mathsf{left}\,(b) && (\beta\text{left}) \\
\therefore p &= \mathsf{do}\ x \leftarrow b; \rhd_1(x) && \text{(by hypothesis)} \\
&= \rhd_1(\mathsf{left}\,(b)) && \text{(do-return)} \\
q &= \rhd_2(\mathsf{left}\,(b)) && \text{(similarly)} \\
&= \rhd_1(\mathsf{left}\,(b))^\perp && (\beta{+_1}), (\beta{+_2}), \text{(case-case)} \\
&= p^\perp
\end{aligned}$$

◀

▶ **Corollary 17.**
1. ***(Cancellation)*** *If* $\Gamma \vdash p \ovee q = p \ovee r : \mathbf{2}$ *then* $\Gamma \vdash q = r : \mathbf{2}$.
2. ***(Positivity)*** *If* $\Gamma \vdash p \ovee q = \bot : \mathbf{2}$ *then* $\Gamma \vdash p = \bot : \mathbf{2}$ *and* $\Gamma \vdash q = \bot : \mathbf{2}$.
3. *If* $\Gamma \vdash p : \mathbf{2}$ *then* $\Gamma \vdash p \leq \top : \mathbf{2}$.
4. *If* $\Gamma \vdash p \leq q : \mathbf{2}$ *then* $\Gamma \vdash q^\perp \leq p^\perp : \mathbf{2}$.
5. *If* $\Gamma \vdash p \leq q : \mathbf{2}$ *and* $\Gamma \vdash q \leq p : \mathbf{2}$ *then* $\Gamma \vdash p = q : \mathbf{2}$.

Proof. We prove part 1 here. We have

$$\begin{aligned}
p \ovee r \ovee (p \ovee q)^\perp &= p \ovee q \ovee (p \ovee q)^\perp && \text{(by hypothesis)} \\
&= \top && \text{(Proposition 16.1)} \\
\therefore q = r &= (p \ovee (p \ovee q)^\perp)^\perp && \text{(Proposition 16.2)}
\end{aligned}$$

◀

5.6 Assert Maps

Recall that, for $x : A \vdash p : \mathbf{2}$ and $\Gamma \vdash t : A$, we define $\Gamma \vdash \mathsf{assert}_{\lambda xp}(t) \stackrel{\text{def}}{=} \rhd_1(\mathsf{instr}_{\lambda xp}(t)) : A + 1$.

We now give rules for calculating $\mathsf{instr}_{\lambda xp}$ and $\mathsf{assert}_{\lambda xp}$ directed by the type.

▶ **Lemma 18** ((assert-scalar))**.** *If* $\vdash s : \mathbf{2}$ *then*

$$\vdash \mathsf{assert}_{\lambda_s}(*) = \mathsf{instr}_{\lambda_s}(*) = s : \mathbf{2}$$

Proof. We have $\nabla(s) = *$ by $(\eta 1)$ and $s{\downarrow}= s$ by $(\eta+)$. The result follows by (ηinstr) . ◀

▶ **Lemma 19.** *The rules* (instr+) *and* (assert+) *are admissible (see Section 2.7.3).*

Proof. We shall prove the case $n = 2$ of (instr+) here. Let $x : A + B \vdash p : \mathbf{2}$.

For $x : A + B$, let us write $f(x) : (A + B) + (A + B)$ for

$$\begin{aligned}
&\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \mathsf{inl}\,(\mathsf{inl}\,(t))\mid \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \mathsf{inr}\,(\mathsf{inl}\,(t))\mid \\
&\qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \mathsf{inl}\,(\mathsf{inr}\,(t))\mid \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \mathsf{inr}\,(\mathsf{inr}\,(t))
\end{aligned}$$

We shall prove $f(x) = \mathsf{instr}_{\lambda xp}(x)$.

We have

$$\begin{aligned}
\nabla(f(x)) = {} & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \nabla(\mathsf{inl}\,(\mathsf{inl}\,(t)))\mid \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \nabla(\mathsf{inr}\,(\mathsf{inl}\,(t)))\mid \\
& \qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \nabla(\mathsf{inl}\,(\mathsf{inr}\,(t)))\mid \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \nabla(\mathsf{inr}\,(\mathsf{inr}\,(t))) \\
& \hfill \text{(case-case)} \\
= {} & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \mathsf{inl}\,(t)\mid \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \mathsf{inl}\,(t)\mid \\
& \qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \mathsf{inr}\,(t)\mid \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \mathsf{inr}\,(t) \\
& \hfill (\beta{+}_1)\ ,\ (\beta{+}_2) \\
= {} & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl}(\mathsf{case}\ \mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto t\mid \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto t)\mid \\
& \qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{inr}(\mathsf{case}\ \mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto t\mid \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto t) \\
& \hfill \text{(Lemma 2)} \\
\stackrel{\text{def}}{=} {} & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl}\,\big(\nabla(\mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y))\big) \\
& \qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,\big(\nabla(\mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y))\big) \\
= {} & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl}\,(y)\mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y) \qquad (\nabla\text{-instr}) \\
= {} & x \qquad (\eta+)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{inl?}\,(f(x)) \\
&= \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \mathsf{inl?}\,(\mathsf{inl}\,(\mathsf{inl}\,(t))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \mathsf{inl?}\,(\mathsf{inr}\,(\mathsf{inl}\,(t))) \\
&\qquad\qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \mathsf{inl?}\,(\mathsf{inl}\,(\mathsf{inr}\,(t))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \mathsf{inl?}\,(\mathsf{inr}\,(\mathsf{inr}\,(t))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Lemma 2)} \\
&= \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \top \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \bot \\
&\qquad\qquad\quad \mathsf{inr}\,(y) \mapsto \mathsf{case}\ \mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\ \mathsf{of}\ \mathsf{inl}\,(t) \mapsto \top \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\,(t) \mapsto \bot \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\beta{+}_1), (\beta{+}_2) \\
&\stackrel{\text{def}}{=} \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl?}\left(\mathsf{instr}_{\lambda ap[x:=\mathsf{inl}(a)]}(y)\right) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inl?}\left(\mathsf{instr}_{\lambda bp[x:=\mathsf{inr}(b)]}(y)\right) \\
&= \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto p[x := \mathsf{inl}\,(y)] \mid \mathsf{inr}\,(y) \mapsto p[x := \mathsf{inr}\,(y)] \qquad \text{(instr-test)} \\
&= p[x := \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\,(y) \mapsto \mathsf{inl}\,(y) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y)] \qquad \text{(Lemma 2)} \\
&= p \qquad (\eta{+})
\end{aligned}$$

Hence $f(x) = \mathsf{instr}_{\lambda xp}(x)$ by (ηinstr) . ◀

The rules (instr m) and (assert m) follow easily.

5.7 Sequential Conjunction

We do not have conjunction or disjunction in our language for predicates over the same type, as this would involve duplicating variables. However, we do have the following *sequential conjunction*. (This was called the 'and-then' test operator in Section 9 in [9].)

Let $x : A \vdash p, q : \mathbf{2}$. We define the *sequential conjunction* $p \mathbin{\&} q$ by

$$x : A \vdash p \mathbin{\&} q \stackrel{\text{def}}{=} \mathsf{do}\ x \leftarrow \mathsf{assert}_{\lambda xp}(x); q : \mathbf{2} \ .$$

▶ **Proposition 20.** *Let $x : A \vdash p, q : \mathbf{2}$.*

1. $\mathsf{instr}_{\lambda x(p \& q)}(x) = \mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{instr}_{\lambda xq}(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y)$
2. $\mathsf{assert}_{\lambda x(p \& q)}(x) = \mathsf{do}\ x \leftarrow \mathsf{assert}_{\lambda xp}(x); \mathsf{assert}_{\lambda xq}(x)$
3. ***(Commutativity)*** $p \mathbin{\&} q = q \mathbin{\&} p$.
4. $(p \ovee q) \mathbin{\&} r = (p \mathbin{\&} r) \ovee (q \mathbin{\&} r)$ *and* $p \mathbin{\&} (q \ovee r) = (p \mathbin{\&} q) \ovee (p \mathbin{\&} r)$.
5. $p \mathbin{\&} \bot = \bot \mathbin{\&} q = \bot$
6. $p \mathbin{\&} \top = p$ *and* $\top \mathbin{\&} q = q$
7. $p \mathbin{\&} (q \mathbin{\&} r) = (p \mathbin{\&} q) \mathbin{\&} r$
8. *If x does not occur free in q, then* $p \mathbin{\&} q = \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto q \mid \mathsf{inr}\,(_) \mapsto \bot$.

Proof. We shall prove the first three parts here.

1. $$\begin{aligned}
&\mathsf{inl?}\,(\mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{instr}_{\lambda xq}(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y)) \\
&= \mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inl?}\,(\mathsf{instr}_{\lambda xq}(x)) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inl?}\,(\mathsf{inr}\,(y)) && \text{(case-case)} \\
&= \mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto q \mid \mathsf{inr}\,(y) \mapsto \bot && \text{(instr-test)}, (\beta{+}_2) \\
&\stackrel{\text{def}}{=} \mathsf{do}\ x \leftarrow \mathsf{assert}_{\lambda xp}(x); q \\
&\stackrel{\text{def}}{=} p \mathbin{\&} q \\
&\nabla(\mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{instr}_{\lambda xq}(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y)) \\
&= \mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \nabla(\mathsf{instr}_{\lambda xq}(x)) \mid \mathsf{inr}\,(y) \mapsto \nabla(\mathsf{inr}\,(y)) && \text{(case-case)} \\
&= \mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto x \mid \mathsf{inr}\,(y) \mapsto y && (\nabla\text{-instr}), (\beta{+}_2) \\
&\stackrel{\text{def}}{=} \nabla(\mathsf{instr}_{\lambda xp}(x)) \\
&= x && (\nabla\text{-instr})
\end{aligned}$$
   so the result follows by (ηinstr) .
2. This follows immediately from the previous part.
3. This follows from the previous part and the rule (comm) (Appendix A.12).

◀

These results show that the scalars form an *effect monoid*, and the predicates on any type form an *effect module* over that effect monoid (see [9]).

5.8 n-tests

Recall that an *n-test* on a type A is an n-tuple $(p_1, \dots, p_n)$ such that $x : A \vdash p_1 \owedge \cdots \owedge p_n = \top : \mathbf{2}$.

The following lemma shows that there is a one-to-one correspondance between the n-tests on A, and the maps $A \to \mathbf{n}$.

▶ **Lemma 21.** *For every n-test $(p_1, \dots, p_n)$ on A, there exists a term $x : A \vdash t(x) : \mathbf{n}$, unique up to equality, such that $x : A \vdash p_i(x) = \rhd_i(t(x)) : \mathbf{2}$.*

Proof. The proof is by induction on n. The case $n = 1$ is trivial.

Suppose the result is true for n. Take an $n+1$-test $(p_1, \dots, p_{n+1})$. Then $(p_1, p_2, \dots, p_n \owedge p_{n+1})$ is an n-test. By the induction hypothesis, there exists $t : \mathbf{n}$ such that $\rhd_i(t) = p_i$ for $i < n$ and $\rhd_n(t) = p_n \owedge p_{n+1}$. Let $b : \mathbf{3}$ be the bound for $p_n \owedge p_{n+1}$. Reading t and b as partial functions in $\mathbf{n-1}+1$ and $\mathbf{2}+1$, we have that $t{\uparrow}= b{\downarrow}= p_n \owedge p_{n+1}$. Hence «$b, t$» $: \mathbf{2} + \mathbf{n-1}$ exists. Reading it as a term of type $\mathbf{n+1}$, we have that

$$\begin{aligned}
\rhd_1^{n+1}(«b,t») &= \rhd_1^3(\rhd_1^{\mathbf{2},\mathbf{n-1}}(«b,t»)) && \text{(case-case)}, (\beta{+}_1), (\beta{+}_2) \\
&= \rhd_1^3(b) && (\beta\text{inlr}_1) \\
&= p_n && (b \text{ is a bound for } p_n \owedge p_{n+1}) \\
\rhd_2^{n+1}(«b,t») &= \rhd_2^3(\rhd_1^{\mathbf{2},\mathbf{n-1}}(«b,t»)) && \text{(case-case)}, (\beta{+}_1), (\beta{+}_2) \\
&= \rhd_2^3(b) && (\beta\text{inlr}_1) \\
&= p_{n+1} && (b \text{ is a bound for } p_n \owedge p_{n+1})
\end{aligned}$$

and for $i < n$:

$$\begin{aligned} \rhd^{n+1}_{i+2}(«b,t») &= \rhd^{n-1}_{i}(\rhd^{\mathbf{2},\mathbf{n-1}}_{2}(«b,t»)) \\ &= \rhd^{n+1}_{i}(t) && (\beta\text{inlr}_2) \\ &= p_i && (\text{induction hypothesis}) \end{aligned}$$

From this it is easy to construct the term of type $\mathbf{n+1}$ required. ◀

We write $\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(s)$ for $\mathsf{instr}_t(s)$, where t is the term such that $\rhd_i(t) = p_i$ for each i.

▶ **Lemma 22.** $\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x)$ *is the unique term such that* $\mathsf{in}_i?\big(\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x)\big) = p_i$ *for all* i *and* $\nabla(\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x)) = x$*.*

▶ **Lemma 23.**

$$\mathsf{instr}_{\lambda x p_i}(x) = \mathsf{case}^n_{j=1}\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x) \text{ of } \mathsf{in}^n_j(x) \mapsto \begin{cases} \mathsf{inl}(x) & \text{if } i = j \\ \mathsf{inr}(x) & \text{if } i \neq j \end{cases}$$

$$\mathsf{assert}_{\lambda x p_i}(x) = \mathsf{case}^n_{j=1}\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x) \text{ of } \mathsf{in}^n_j(x) \mapsto \begin{cases} \mathsf{return}\ x & \text{if } i = j \\ \mathsf{fail} & \text{if } i \neq j \end{cases}$$

Proof. Let t be the right-hand side of the first formula. Then

$$\begin{aligned} \mathsf{inl}?(t) &= \mathsf{in}_i?\big(\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x)\big) && (\text{case-case}), (\beta+_1), (\beta+_2) \\ &= p_i && (\text{Lemma 22}) \\ \nabla(t) &= \mathsf{case}^n_{j=1}\mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x) \text{ of } \mathsf{in}^n_j(x) \mapsto x && (\text{case-case}), (\beta+_1), (\beta+_2) \\ &= x && (\text{Lemma 2}) \end{aligned}$$

The second formula follows easily from the first. ◀

We can now define the program that divides into n branches depending on the outcome of an n-test:

▶ **Definition 24.** Given $x : A \vdash p_1(x) \owedge \cdots \owedge p_n(x) = \top : \mathbf{2}$, define

$$\begin{aligned} &x : A \vdash \mathsf{measure}\ p_1(x) \mapsto t_1(x) \mid \cdots \mid p_n(x) \mapsto t_n(x) \\ &\quad \stackrel{\text{def}}{=} \mathsf{case}\ \mathsf{instr}_{\lambda x(p_1,\ldots,p_n)}(x) \text{ of } \mathsf{in}^n_1(x) \mapsto t_1(x) \mid \cdots \mid \mathsf{in}^n_n(x) \mapsto t_n(x) \end{aligned}$$

▶ **Lemma 25.** *The* measure *construction satisfies the following laws.*

1. $(\mathsf{measure}\ \top \mapsto t) = t$
2. $(\mathsf{measure}\ p_1 \mapsto t_1 \mid \cdots \mid p_n \mapsto t_n \mid \bot \mapsto t_{n+1}) = (\mathsf{measure}\ p_1 \mapsto t_1 \mid \cdots \mid p_n \mapsto t_n)$
3. $(\mathsf{measure}_i\ p_i \mapsto \mathsf{measure}_j\ q_{ij} \mapsto t_{ij}) = (\mathsf{measure}_{i,j}\ (p_i \mathbin{\&} q_{ij}) \mapsto t_{ij})$
4. *For any permutation* π *of* $\{1,\ldots,n\}$, $\mathsf{measure}_i\ p_i \mapsto t_i = \mathsf{measure}_i\ p_{\pi(i)} \mapsto t_{\pi(i)}$.
5. *If* $t_n = t_{n+1}$ *then* $\mathsf{measure}^n_{i=1} p_i \mapsto t_i = \mathsf{measure}\ p_1 \mapsto t_1 \mid \cdots \mid p_{n-1} \mapsto t_{n-1} \mid p_n \owedge p_{n+1} \mapsto t_n$.

Proof. We shall prove part 3. The proof for the other parts follows the same pattern.

Let us write $\mathsf{in}_{i,j}()$ $(1 \leq i \leq m,\ 1 \leq j \leq n_i)$ for the constructors of $(n_1 + \cdots + n_m) \cdot A$, and $\mathsf{in}_{i,j}?()$ for the corresponding predicates.

We shall first prove

$$\begin{aligned}&\mathsf{instr}_{\lambda x(p_i \& q_{ij})_{i,j}}(x) \\ &= \mathsf{case}^m_{i=1}\ \mathsf{instr}_{\lambda x \vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_i\,(x) \mapsto \mathsf{case}^{n_i}_{j=1}\ \mathsf{instr}_{\lambda x \vec{q_i}}(x)\ \mathsf{of}\ \mathsf{in}^{n_i}_j\,(x) \mapsto \mathsf{in}_{i,j}\,(x)\quad .\end{aligned} \tag{1}$$

Let R denote the right-hand side of (1). We have

$$\begin{aligned}
\mathsf{in}_{i,j}?\,(R) &= \mathsf{case}^m_{i'=1}\mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_{i'}\,(x) \mapsto \\
&\qquad \mathsf{case}^{n_i}_{j'=1}\mathsf{instr}_{\lambda x\vec{q_i}}(x)\ \mathsf{of}\ \mathsf{in}^{n_i}_j\,(x) \mapsto \begin{cases}\top & \text{if } i=i' \text{ and } j=j' \\ \bot & \text{otherwise}\end{cases} \\
&\hfill \text{(case-case)}, (\beta{+}_1), (\beta{+}_2) \\
&= \mathsf{case}^m_{i'=1}\ \mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_{i'}\,(x) \mapsto \begin{cases}\mathsf{in}_j?\,(\mathsf{instr}_{\lambda x\vec{q_i}}(x)) & \text{if } i=i' \\ \bot & \text{if } i\neq i'\end{cases} \\
&\hfill \text{(Lemma 2)} \\
&= \mathsf{case}^m_{i'=1}\ \mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_{i'}\,(x) \mapsto \begin{cases} q_{ij} & \text{if } i=i' \\ \bot & \text{if } i\neq i'\end{cases} \qquad \text{(instr-test)} \\
&= \mathsf{do}\ x \leftarrow \left(\mathsf{case}^m_{i'=1}\ \mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_{i'}\,(x) \mapsto \begin{cases}\mathsf{return}\ x & \text{if } i=i' \\ \mathsf{fail} & \text{if } i\neq i'\end{cases}\right); q_{ij} \\
&\hfill \text{(case-case)}, (\beta{+}_1), (\beta{+}_2) \\
&= \mathsf{do}\ x \leftarrow \mathsf{assert}_{\lambda x p_i}(x); q_{ij} \qquad \text{(by Lemma 23)} \\
&\stackrel{\text{def}}{=} p_i \ \&\ q_{ij} \\
\nabla(R) &= \mathsf{case}^m_{i=1}\ \mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_i\,(x) \mapsto \nabla(\mathsf{instr}_{\lambda x\vec{q_i}}(x)) \qquad \text{(case-case)} \\
&= \mathsf{case}^m_{i=1}\ \mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_i\,(x) \mapsto x \qquad (\nabla\text{-instr}) \\
&\stackrel{\text{def}}{=} \nabla(\mathsf{instr}_{\lambda x\vec{p}}(x)) = x \qquad (\nabla\text{-instr})
\end{aligned}$$

Equation (1) follows by (ηinstr).

Now, we have

$$\begin{aligned}
&\mathsf{measure}_{ij}(p_i \ \&\ q_{ij}) \mapsto t_{ij} \\
&\stackrel{\text{def}}{=} \mathsf{case}_{ij}\ \mathsf{instr}_{\lambda x(p_i \& q_{ij})_{ij}}(x)\ \mathsf{of}\ \mathsf{in}_{i,j}\,(x) \mapsto t_{ij} \\
&= \mathsf{case}_{ij}\ R\ \mathsf{of}\ \mathsf{in}_{i,j}\,(x) \mapsto t_{ij} \qquad \text{by (1)} \\
&= \mathsf{case}_i\ \mathsf{instr}_{\lambda x\vec{p}}(x)\ \mathsf{of}\ \mathsf{in}^m_i\,(x) \mapsto \mathsf{case}_j \mathsf{instr}_{\lambda x\vec{q_i}}(x)\ \mathsf{of}\ \mathsf{in}^{m_i}_j\,(x) \mapsto t_{ij} \\
&\hfill \text{(case-case)}, (\beta{+}_1), (\beta{+}_2) \\
&\stackrel{\text{def}}{=} \mathsf{measure}_i\ p_i \mapsto \mathsf{measure}_j\ q_{ij} \mapsto t_{ij}
\end{aligned}$$

◀

Let $x : A \vdash p : \mathbf{2}$ and $\Gamma, x : A \vdash s, t : B$. We define $\Gamma, x : A \vdash \mathsf{if}\ p\ \mathsf{then}\ s\ \mathsf{else}\ t \stackrel{\text{def}}{=} \mathsf{measure}\ p \mapsto s \mid p^\perp \mapsto t : B$.

▶ **Lemma 26.**

1. *If* $x : A \vdash p_1 \otimes \cdots \otimes p_n = \top : \mathbf{2}$ *and* $x : A \vdash q_1, \ldots, q_n : \mathbf{2}$, *then*

$$(\mathsf{measure}\ p_1 \mapsto q_1 \mid \cdots \mid p_n \mapsto q_n) = (p_1 \ \&\ q_1) \otimes \cdots \otimes (p_n \ \&\ q_n) \quad .$$

2. *Let* $x : A \vdash p : \mathbf{2}$ *and* $\Gamma \vdash q, r : B$ *where* $x \notin \Gamma$. *Then*

$$\Gamma, x : A \vdash \mathsf{if}\ p\ \mathsf{then}\ q\ \mathsf{else}\ r = \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto q \mid \mathsf{inr}\,(_) \mapsto r : B \quad .$$

3. *Let* $x : A \vdash p : \mathbf{2}$*. Then* $x : A \vdash \mathsf{if}\ p\ \mathsf{then}\ \top\ \mathsf{else}\ \bot = p : \mathbf{2}$*.*

Proof. We prove part 2 here. We have

$$\begin{aligned}\mathsf{measure}\ p \mapsto q \mid p^\perp \mapsto r &\stackrel{\text{def}}{=} \mathsf{case}\ \mathsf{instr}_{\lambda xp}(x)\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto q \mid \mathsf{inr}\,(_) \mapsto r \\ &= \mathsf{case}\ \mathsf{inl?}\,(\mathsf{instr}_{\lambda xp}(x))\ \ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto q \mid \mathsf{inr}\,(_) \mapsto r \\ &\qquad\qquad (\text{case-case}), (\beta{+}_1), (\beta{+}_2) \\ &= \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,(_) \mapsto q \mid \mathsf{inr}\,(_) \mapsto r \qquad (\text{instr-test})\end{aligned}$$

◀

5.9 Scalars

From the rules given in Figure 2, the usual algebra of the rational interval from 0 to 1 follows.

▶ **Lemma 27.** *If* $p/q = m/n$ *as rational numbers, then* $\vdash p \cdot (1/q) = m \cdot (1/n) : \mathbf{2}$*.*

Proof. We first prove that $\vdash a \cdot (1/ab) = 1/b : \mathbf{2}$ for all a, b. This holds because $ab \cdot (1/ab) = \top$ by $(n \cdot 1/n)$, hence $a \cdot (1/ab) = 1/b$ by (divide) .

Hence we have $p \cdot (1/q) = pn \cdot (1/nq) = qm \cdot (1/nq) = m \cdot (1/n)$. ◀

Recall that within **COMET**, we are writing m/n for the term $m \cdot (1/n)$. Similar reasoning leads us to

▶ **Lemma 28.** *Let* q *and* r *be rational numbers in* $[0, 1]$*.*

1. *If* $q \leq r$ *in the usual ordering, then* $\vdash q \leq r : \mathbf{2}$*.*
2. $\vdash q \ovee r : \mathbf{2}$ *iff* $q + r \leq 1$*, in which case* $\vdash q \ovee r = q + r : \mathbf{2}$*.*
3. $\vdash q \mathbin{\&} r = qr : \mathbf{2}$*.*

5.10 Normalisation

The following lemma gives us a rule that allows us to calculate the normalised form of a substate in many cases, including the examples in Section 3.

▶ **Lemma 29.** *Let* $\vdash t : A + 1$, $\vdash p_1 \ovee \cdots \ovee p_n = \top : \mathbf{2}$*, and* $\vdash q : \mathbf{2}$*. Let* $\vdash s_1, \ldots, s_n : A$*. Suppose* $\vdash 1/m \leq q : \mathbf{2}$*. If*

$$\vdash t = \mathsf{measure}\ p_1 \mathbin{\&} q \mapsto \mathsf{return}\ s_1 \mid \cdots \mid p_n \mathbin{\&} q \mapsto \mathsf{return}\ s_n \mid q^\perp \mapsto \mathsf{fail} : A + 1 \ \ , \textit{ then}$$

$$\vdash \mathsf{nrm}\,(t) = \mathsf{measure}\ p_1 \mapsto s_1 \mid \cdots \mid p_n \mapsto s_n : A$$

Proof. Let $\rho \stackrel{\text{def}}{=} \mathsf{measure}_{i=1}^n p_i \ \mapsto\ s_i$. By the rule (ηnrm) , it is sufficient to prove that $t = \mathsf{do}\ _ \leftarrow t; \mathsf{return}\ \rho$. We have

$$\begin{aligned}\mathsf{do}\ _ \leftarrow t; \mathsf{return}\ \rho &= \mathsf{measure}\ p_1 \mathbin{\&} q \mapsto \mathsf{return}\ \rho \mid \cdots \mid p_n \mathbin{\&} q \mapsto \mathsf{return}\ \rho \mid q^\perp \mapsto \mathsf{fail} \\ &\qquad\qquad (\text{case-case}), (\beta{+}_1), (\beta{+}_2) \\ &= \mathsf{measure}\ q \mapsto \mathsf{return}\ \rho \mid q^\perp \mapsto \mathsf{fail} \qquad (\text{Lemma 25}) \\ &= \mathsf{measure}_{i=1}^n\ q \mathbin{\&} p_i \mapsto \mathsf{return}\ s_i \mid q^\perp \mapsto \mathsf{fail} \qquad (\text{Lemma 25}) \\ &= t \qquad (\text{Proposition 20})\end{aligned}$$

◀

▶ **Corollary 30.** *Let* $\alpha_1, \ldots, \alpha_n, \beta$ *be rational numbers that sum to 1, with* $\beta \neq 1$*. If*

$$\vdash t = \mathsf{measure}\ \alpha_1 \mapsto \mathsf{return}\ s_1 \mid \cdots \mid \alpha_n \mapsto \mathsf{return}\ s_n \mid \beta \mapsto \mathsf{fail} : A + 1 \ \ , \textit{ then}$$

$$\vdash \mathsf{nrm}\,(t) = \mathsf{measure}\ \alpha_1/(\alpha_1 + \cdots + \alpha_n) \mapsto s_1 \mid \cdots \mid \alpha_n/(\alpha_1 + \cdots + \alpha_n) \mapsto s_n : A \ .$$

6 Conclusion

The system **COMET** allows for the specification of probabilistic programs and reasoning about their properties, both within the same syntax.

There are several avenues for further work and research.

- The type theory that we describe can be interpreted both in discrete and in continuous probabilistic models, that is, both in the Kleisli category $\mathcal{K}\ell(\mathcal{D})$ of the distribution monad $\mathcal{D}$ and in the Kleisli category $\mathcal{K}\ell(\mathcal{G})$ of the Giry monad $\mathcal{G}$. On a finite type each distribution is discrete. The discrete semantics were exploited in the current paper in the examples in Section 3. In a follow-up version we intend to elaborate also continuous examples.
- The normalisation and conditioning that we use in this paper can in principle also be used in a quantum context, using the appropriate (non-side-effect free) assert maps that one has there. This will give a form of Bayesian quantum theory, as also explored in [17].
- A further ambitious follow-up project is to develop tool support for **COMET**, so that the computations that we carry out here by hand can be automated. This will provide a formal language for Bayesian inference.

Acknowledgements Thanks to Kenta Cho for discussion and suggestions during the writing of this paper, and very detailed proofreading. Thanks to Bas Westerbaan for discussions about effectus theory.

References

1 R. Adams. QPEL: Quantum program and effect language. In B. Coecke, I. Hasuo, and P. Panangaden, editors, *Proc. of 11th Int. Workshop on Quantum Physics and Logic, QPL 2014*, volume 172 of *Electron. Proc. Theor. Comput. Sci.*, pages 133–153. Open Publishing Assoc., 2014. `doi:10.4204/eptcs.172.10`.

2 N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proc. of Int. Conf. on Typed Lambda Calculi and Applications, TLCA '93*, volume 664 of *Lect. Notes in Comput. Sci.*, pages 75–90. Springer, 1993. `doi:10.1007/bfb0037099`.

3 Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for Bayesian machine learning. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 77–96. Springer, 2011. `doi:10.1007/978-3-642-19718-5_5`.

4 K. Cho. Total and partial computation in categorical quantum foundations. In C. Heunen, P. Selinger, and J. Vicary, editors, *Proc. of 12th Int. Workshop on Quantum Physics and Logic, QPL 2015*, volume 195 of *Electron. Proc. in Theor. Comput. Sci.*, pages 116–135. Open Publishing Assoc., 2015. `doi:10.4204/eptcs.195.9`.

5 K. Cho, B. Jacobs, A. Westerbaan, and B. Westerbaan. An introduction to effectus theory, 2015. arXiv preprint 1512.05813. URL: `https://arxiv.org/abs/1512.05813`.

6 K. Cho, B. Jacobs, A. Westerbaan, and B. Westerbaan. Quotient comprehension chains. In C. Heunen, P. Selinger, and J. Vicary, editors, *Proc. of 12th Int. Workshop on Quantum Physics and Logic, QPL 2015*, volume 195 of *Electron. Proc. in Theor. Comput. Sci.*, pages 136–147. Open Publishing Assoc., 2015. `doi:10.4204/eptcs.195.10`.

7 Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In James D. Herbsleb and Matthew B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 167–181. ACM, 2014. doi:10.1145/2593882.2593900.

8 B. Jacobs. Measurable spaces and their effect logic. In *Proc. of 28th Annual ACM/IEEE Symp. on Logic in Computer Science, LICS '13*, pages 83–92. IEEE, 2013. doi:10.1109/lics.2013.13.

9 B. Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *Log. Methods Comput. Sci.*, 11(3):article 24, 2015. doi:10.2168/lmcs-11(3:24)2015.

10 Bart Jacobs. From probability monads to commutative effectuses. *J. Log. Algebr. Meth. Program.*, 94:200–237, 2018. doi:10.1016/j.jlamp.2016.11.006.

11 Bart Jacobs, Bas Westerbaan, and Bram Westerbaan. States of convex sets. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2015. doi:10.1007/978-3-662-46678-0_6.

12 Bart Jacobs and Fabio Zanasi. A predicate/state transformer semantics for bayesian learning. *Electr. Notes Theor. Comput. Sci.*, 325:185–200, 2016. doi:10.1016/j.entcs.2016.09.038.

13 C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proc. of 4th Ann. IEEE Symp. on Logic in Computer Science, LICS '89*, pages 186–195. IEEE, 1989. doi:10.1109/lics.1989.39173.

14 Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 364–389. Springer, 2016. doi:10.1007/978-3-662-49498-1_15.

15 Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981. doi:10.1016/0022-0000(81)90036-2.

16 Dexter Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985. doi:10.1016/0022-0000(85)90012-1.

17 M. S. Leifer and R. W. Spekkens. Towards a formulation of quantum theory as a causally neutral theory of Bayesian inference. *Phys. Rev. A*, 88(5):article 052130, 2013. doi:10.1103/physreva.88.052130.

18 Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996. doi:10.1145/229542.229547.

19 Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Reasoning about recursive probabilistic programs. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 672–681. ACM, 2016. doi:10.1145/2933575.2935317.

20 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

21 Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer*

Science, LICS '16, New York, NY, USA, July 5-8, 2016, pages 525–534. ACM, 2016. doi:10.1145/2933575.2935313.

22 Regina Tix, Klaus Keimel, and Gordon D. Plotkin. Semantic domains for combining probability and non-determinism. *Electr. Notes Theor. Comput. Sci.*, 222:3–99, 2009. doi:10.1016/j.entcs.2009.01.002.

23 E. S. Yudkowsky. An intuitive explanation of Bayesian reasoning. Essay, 2003. URL: http://yudkowsky.net/rational/bayes.

A Formal Presentation of COMET

The full syntax of **COMET** is given by the grammar:

$$\begin{array}{lll} \text{Type} & A, B & ::= \mathbf{C} \mid 0 \mid 1 \mid A + B \mid A \otimes B \\ \text{Term} & r, s, t & ::= x \mid * \mid s \otimes t \mid \mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t \mid \text{¡} t \mid \mathsf{inl}\,(t) \mid \mathsf{inr}\,(t) \mid \\ & & (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t) \mid \langle\!\langle s, t \rangle\!\rangle \mid \mathsf{left}\,(t) \mid \mathsf{instr}_{\lambda xs}(t) \mid 1/n \mid b_{mn} \\ & & \mathsf{nrm}\,(t) \mid s \owedge t \end{array}$$

We have a constant $1/n$ for every natural number $n \geq 2$, and a constant b_{mn} for all natural numbers $1 \leq m < n$.

The full set of rules of deduction for **COMET** are given below.

A.1 Structural Rules

$$(\text{exch})\ \frac{\Gamma, x : A, y : B, \Delta \vdash \mathcal{J}}{\Gamma, y : B, x : A, \Delta \vdash \mathcal{J}} \qquad (\text{var})\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(\text{ref})\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad (\text{sym})\ \frac{\Gamma \vdash s = t : A}{\Gamma \vdash t = s : A} \qquad (\text{trans})\ \frac{\Gamma \vdash r = s : A \qquad \Gamma \vdash s = t : A}{\Gamma \vdash r = t : A}$$

A.2 The Unit Type

$$(\text{unit})\ \frac{}{\Gamma \vdash * : 1} \qquad (\eta 1)\ \frac{\Gamma \vdash t : 1}{\Gamma \vdash t = * : 1}$$

A.3 Tensor Product

$$(\otimes)\ \frac{\Gamma \vdash s : A \qquad \Delta \vdash t : B}{\Gamma, \Delta \vdash s \otimes t : A \otimes B} \qquad (\text{let})\ \frac{\Gamma \vdash s : A \otimes B \qquad \Delta, x : A, y : B \vdash t : C}{\Gamma, \Delta \vdash \mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t : C}$$

$$(\text{paireq})\ \frac{\Gamma \vdash s = s' : A \qquad \Delta \vdash t = t' : B}{\Gamma, \Delta \vdash s \otimes t = s' \otimes t' : A \otimes B}$$

$$(\text{leteq})\ \frac{\Gamma \vdash s = s' : A \otimes B \qquad \Delta, x : A, y : B \vdash t = t' : C}{\Gamma, \Delta \vdash (\mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t) = (\mathsf{let}\ x \otimes y = s'\ \mathsf{in}\ t') : C}$$

$$(\beta\otimes)\ \frac{\Gamma \vdash r : A \qquad \Delta \vdash s : B \qquad \Theta, x : A, y : B \vdash t : C}{\Gamma, \Delta, \Theta \vdash (\mathsf{let}\ x \otimes y = r \otimes s\ \mathsf{in}\ t) = t[x := r, y := s] : C}$$

$$(\eta\otimes)\ \frac{\Gamma \vdash t : A \otimes B}{\Gamma \vdash t = (\mathsf{let}\ x \otimes y = t\ \mathsf{in}\ x \otimes y) : A \otimes B}$$

$$\text{(let-let)}\quad \frac{\Gamma \vdash r : A \otimes B \qquad \Delta, x : A, y : B \vdash s : C \otimes D \qquad \Theta, z : C, w : D \vdash t : E}{\begin{array}{l}\Gamma, \Delta, \Theta \vdash \ \mathsf{let}\ x \otimes y = r\ \mathsf{in}\ (\mathsf{let}\ z \otimes w = s\ \mathsf{in}\ t) \\ \qquad = \mathsf{let}\ z \otimes w = (\mathsf{let}\ x \otimes y = r\ \mathsf{in}\ s)\ \mathsf{in}\ t : E\end{array}}$$

$$\text{(let-}\otimes\text{)}\quad \frac{\Gamma \vdash r : A \otimes B \qquad \Delta, x : A, y : B \vdash s : C \qquad \Theta \vdash t : D}{\Gamma, \Delta, \Theta \vdash \mathsf{let}\ x \otimes y = r\ \mathsf{in}\ (s \otimes t) = (\mathsf{let}\ x \otimes y = r\ \mathsf{in}\ s) \otimes t : D}$$

A.4 Empty Type

$$\text{(magic)}\quad \frac{\Gamma \vdash t : 0}{\Gamma \vdash\ ¡\, t : A} \qquad (\eta 0)\quad \frac{\Gamma \vdash s : 0 \qquad \Gamma \vdash t : A}{\Gamma \vdash\ ¡\, s = t : A}$$

A.5 Binary Coproducts

$$\text{(inl)}\quad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inl}\,(t) : A + B} \qquad \text{(inr)}\quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{inr}\,(t) : A + B}$$

$$\text{(inl-eq)}\quad \frac{\Gamma \vdash t = t' : A}{\Gamma \vdash \mathsf{inl}\,(t) = \mathsf{inl}\,(t') : A + B} \qquad \text{(inr-eq)}\quad \frac{\Gamma \vdash t = t' : B}{\Gamma \vdash \mathsf{inr}\,(t) = \mathsf{inr}\,(t') : A + B}$$

$$\text{(case)}\quad \frac{\Gamma \vdash r : A + B \qquad \Delta, x : A \vdash s : C \qquad \Delta, y : B \vdash t : C}{\Gamma, \Delta \vdash \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t : C}$$

$$\text{(case-eq)}\quad \frac{\Gamma \vdash r = r' : A + B \qquad \Delta, x : A \vdash s = s' : C \qquad \Delta, y : B \vdash t = t' : C}{\Gamma, \Delta \vdash \begin{array}{l}\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t \\ = \mathsf{case}\ r'\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s' \mid \mathsf{inr}\,(y) \mapsto t' : C\end{array}}$$

$$(\beta{+_1})\quad \frac{\Gamma \vdash r : A \qquad \Delta, x : A \vdash s : C \qquad \Delta, y : B \vdash t : C}{\Gamma, \Delta \vdash \mathsf{case}\ \mathsf{inl}\,(r)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t = s[x := r] : C}$$

$$(\beta{+_2})\quad \frac{\Gamma \vdash r : B \qquad \Delta, x : A \vdash s : C \qquad \Delta, y : B \vdash t : C}{\Gamma, \Delta \vdash \mathsf{case}\ \mathsf{inr}\,(r)\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto t = t[y := r] : C}$$

$$(\eta{+})\quad \frac{\Gamma \vdash t : A + B}{\Gamma \vdash t = \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{inl}\,(x) \mid \mathsf{inr}\,(y) \mapsto \mathsf{inr}\,(y) : A + B}$$

$$\text{(case-case)}\quad \frac{\begin{array}{c}\Gamma \vdash r : A + B \qquad \Delta, x : A \vdash s : C + D \qquad \Delta, y : B \vdash s' : C + D \\ \Theta, z : C \vdash t : E \qquad \Theta, w : D \vdash t' : E\end{array}}{\begin{array}{l}\Gamma, \Delta, \Theta \vdash\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\,(z) \mapsto t \mid \mathsf{inr}\,(w) \mapsto t' \mid \\ \qquad\qquad \mathsf{inr}\,(y) \mapsto \mathsf{case}\ s'\ \mathsf{of}\ \mathsf{inl}\,(z) \mapsto t \mid \mathsf{inr}\,(w) \mapsto t' \\ \qquad = \mathsf{case}\ (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto s') \\ \qquad\qquad \mathsf{of}\ \mathsf{inl}\,(z) \mapsto t \mid \mathsf{inr}\,(w) \mapsto t' : E\end{array}}$$

$$\text{(case-}\otimes\text{)}\quad \frac{\Gamma \vdash r : A + B \qquad \Delta, x : A \vdash s : C \qquad \Delta, y : B \vdash s' : C \qquad \Theta \vdash t : D}{\begin{array}{l}\Gamma, \Delta, \Theta \vdash\ (\mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \mid \mathsf{inr}\,(y) \mapsto s') \otimes t = \\ \qquad \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(x) \mapsto s \otimes t \mid \mathsf{inr}\,(y) \mapsto s' \otimes t : C \otimes D\end{array}}$$

$$\text{(let-case)}\ \frac{\begin{array}{c}\Gamma \vdash r : A + B \qquad \Delta, z : A \vdash s : C \otimes D \\ \Delta, w : B \vdash s' : C \otimes D \qquad \Theta, x : C, y : D \vdash t : E\end{array}}{\begin{array}{c}\Gamma, \Delta, \Theta \vdash \ \mathsf{let}\ x \otimes y = \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(z) \mapsto s \mid \mathsf{inr}\,(w) \mapsto s'\ \mathsf{in}\ t = \\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{inl}\,(z) \mapsto \mathsf{let}\ x \otimes y = s\ \mathsf{in}\ t \mid \\ \mathsf{inr}\,(w) \mapsto \mathsf{let}\ x \otimes y = s'\ \mathsf{in}\ t : E\end{array}}$$

A.6 Partial Pairing

$$\text{(inlr)}\ \frac{\Gamma \vdash s : A + 1 \qquad \Gamma \vdash t : B + 1 \qquad \Gamma \vdash s \downarrow = t \uparrow : \mathbf{2}}{\Gamma \vdash «s, t» : A + B}$$

$$\text{(inlr-eq)}\ \frac{\Gamma \vdash s = s' : A + 1 \qquad \Gamma \vdash t = t' : B + 1 \qquad \Gamma \vdash s \downarrow = t \uparrow : \mathbf{2}}{\Gamma \vdash «s, t» = «s', t'» : A + B}$$

$$(\beta\text{inlr}_1)\ \frac{\Gamma \vdash s : A + 1 \qquad \Gamma \vdash t : B + 1 \qquad \Gamma \vdash s \downarrow = t \uparrow : \mathbf{2}}{\Gamma \vdash \rhd_1(«s, t») = s : A + 1}$$

$$(\beta\text{inlr}_2)\ \frac{\Gamma \vdash s : A + 1 \qquad \Gamma \vdash t : B + 1 \qquad \Gamma \vdash s \downarrow = t \uparrow : \mathbf{2}}{\Gamma \vdash \rhd_2(«s, t») = t : B + 1}$$

$$(\eta\text{inlr})\ \frac{\Gamma \vdash t : A + B}{\Gamma \vdash t = «\rhd_1(t), \rhd_2(t)» : A + B}$$

A.7 The left() Construction

$$\text{(left)}\ \frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash \mathsf{inl?}\,(t) = \top : \mathbf{2}}{\Gamma \vdash \mathsf{left}\,(t) : A}$$

$$\text{(left-eq)}\ \frac{\Gamma \vdash t = t' : A + B \qquad \Gamma \vdash \mathsf{inl?}\,(t) = \top : \mathbf{2}}{\Gamma \vdash \mathsf{left}\,(t) = \mathsf{left}\,(t') : A}$$

$$(\beta\text{left})\ \frac{\Gamma \vdash t : A + B \qquad \Gamma \vdash \mathsf{inl?}\,(t) = \top : \mathbf{2}}{\Gamma \vdash \mathsf{inl}\,(\mathsf{left}\,(t)) = t : A + B} \qquad (\eta\text{left})\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{left}\,(\mathsf{inl}\,(t)) = t : A}$$

A.8 Instruments

$$\text{(instr)}\ \frac{x : A \vdash t : \mathbf{n} \qquad \Gamma \vdash s : A}{\Gamma \vdash \mathsf{instr}_{\lambda x t}(s) : n \cdot A} \qquad (\nabla\text{-instr})\ \frac{x : A \vdash t : \mathbf{n} \qquad \Gamma \vdash s : A}{\Gamma \vdash \nabla(\mathsf{instr}_{\lambda x t}(s)) = s : A}$$

$$\text{(instr-eq)}\ \frac{x : A \vdash t = t' : \mathbf{n} \qquad \Gamma \vdash s = s' : A}{\Gamma \vdash \mathsf{instr}_{\lambda x t}(s) = \mathsf{instr}_{\lambda x t'}(s') : n \cdot A}$$

$$\text{(instr-test)}\ \frac{x : A \vdash t : \mathbf{n} \qquad \Gamma \vdash s : A}{\Gamma \vdash \mathsf{case}\ {}_{i=1}^{n}\mathsf{instr}_{\lambda x t}(s)\ \mathsf{of}\ \mathsf{in}_i^n\,(_) \mapsto i = t[x := s] : \mathbf{n}}$$

$$(\eta\text{instr})\ \frac{x : A \vdash r : n \cdot A \qquad x : A \vdash \nabla(r) = x : A \qquad \Gamma \vdash s : A}{\Gamma \vdash \mathsf{instr}_{\lambda x.\mathsf{case}\ {}_{i=1}^{n} r\ \mathsf{of}\ \mathsf{in}_i^n(_) \mapsto i}(s) = r[x := s] : n \cdot A}$$

A.9 Scalar Constants

For any natural number $n \geq 2$, we have the following rules.

$$(1/n)\ \frac{}{\Gamma \vdash 1/n : \mathbf{2}} \qquad (n \cdot 1/n)\ \frac{}{\Gamma \vdash n \cdot 1/n = \top : \mathbf{2}}$$

$$(\text{divide})\ \frac{\Gamma \vdash n \cdot t = \top : \mathbf{2}}{\Gamma \vdash t = 1/n : \mathbf{2}} \qquad (b_{mn})\ \frac{}{\Gamma \vdash b_{mn} : \mathbf{3}}\ (1 \leq m < n)$$

$$(\rhd_1 - b_{mn})\ \frac{}{\Gamma \vdash \mathsf{do}\ x \leftarrow b_{mn}; \rhd_1(x) = 1/n : \mathbf{2}}\ (1 \leq m < n)$$

$$(\rhd_2 - b_{mn})\ \frac{}{\Gamma \vdash \mathsf{do}\ x \leftarrow b_{mn}; \mathsf{return}\ \nabla(x) = m \cdot 1/n : \mathbf{2}}\ (1 \leq m < n)$$

These ensure that $1/n$ is the unique scalar whose sum with itself n times is $\top$. The term b_{mn} ensures that the term $(m+1) \cdot 1/n$ is well-typed.

A.10 Normalisation

$$(\text{nrm})\ \frac{\vdash t : A + 1 \qquad \vdash 1/n \leq t \downarrow : \mathbf{2}}{\Gamma \vdash \mathsf{nrm}\,(t) : A}$$

$$(\beta\text{nrm})\ \frac{\vdash t : A + 1 \qquad \vdash 1/n \leq t \downarrow : \mathbf{2}}{\Gamma \vdash t = \mathsf{do}\ _ \leftarrow t; \mathsf{return}\ \mathsf{nrm}\,(t) : A + 1}$$

$$(\eta\text{nrm})\ \frac{\vdash t : A + 1 \qquad \vdash 1/n \leq t \downarrow : \mathbf{2} \qquad \vdash \rho : A \qquad \vdash t = \mathsf{do}\ _ \leftarrow t; \mathsf{return}\ \rho : A + 1}{\Gamma \vdash \rho = \mathsf{nrm}\,(t) : A}$$

A.11 Partial Sum

$$(\ovee)\ \frac{\begin{array}{cc} \Gamma \vdash s : A + 1 & \Gamma \vdash t : A + 1 \\ \Gamma \vdash b : (A + A) + 1 & \Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_1(x) = s : A + 1 \\ \multicolumn{2}{c}{\Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_2(x) = t : A + 1} \end{array}}{\Gamma \vdash s \ovee t : A + 1}$$

$$(\ovee\text{-def})\ \frac{\begin{array}{cc} \Gamma \vdash s : A + 1 & \Gamma \vdash t : A + 1 \\ \Gamma \vdash b : (A + A) + 1 & \Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_1(x) = s : A + 1 \\ \multicolumn{2}{c}{\Gamma \vdash \mathsf{do}\ x \leftarrow b; \rhd_2(x) = t : A + 1} \end{array}}{\Gamma \vdash s \ovee t = \mathsf{do}\ x \leftarrow b; \mathsf{return}\ \nabla(x) : A + 1}$$

A.12 Miscellaneous

$$(\text{JM})\ \frac{\begin{array}{cc} \Gamma \vdash s : (A + A) + 1 & \Gamma \vdash t : (A + A) + 1 \\ \multicolumn{2}{c}{\Gamma \vdash \mathsf{do}\ x \leftarrow s; \rhd_1(x) = \mathsf{do}\ x \leftarrow t; \rhd_1(x) : A + 1} \\ \multicolumn{2}{c}{\Gamma \vdash \mathsf{do}\ x \leftarrow s; \rhd_2(x) = \mathsf{do}\ x \leftarrow t; \rhd_2(x) : A + 1} \end{array}}{\Gamma \vdash s = t : (A + A) + 1}$$

$$(\text{comm})\ \frac{x : A \vdash p : \mathbf{2} \qquad x : A \vdash q : \mathbf{2} \qquad \Gamma \vdash t : A}{\Gamma \vdash\ \mathsf{do}\ y \leftarrow \mathsf{assert}_{\lambda x p}(t); \mathsf{assert}_{\lambda x q}(y) = \mathsf{do}\ y \leftarrow \mathsf{assert}_{\lambda x q}(t); \mathsf{assert}_{\lambda x p}(y) : A + 1}$$

Heterogeneous Substitution Systems Revisited*

Benedikt Ahrens[1] and Ralph Matthes[2]

1 **Inria Rennes – Bretagne Atlantique,**
4 rue Alfred Kastler, 44307 Nantes Cedex 3, France
`benedikt.ahrens@inria.fr`

2 **Institut de Recherche en Informatique de Toulouse (IRIT),**
CNRS & Université Toulouse 3 Paul Sabatier,
118 route de Narbonne, 31062 Toulouse Cedex 9, France
`matthes@irit.fr`

Abstract

Matthes and Uustalu (TCS 327(1–2):155–174, 2004) presented a categorical description of substitution systems capable of capturing syntax involving binding which is independent of whether the syntax is made up from least or greatest fixed points. We extend this work in two directions: we continue the analysis by creating more categorical structure, in particular by organizing substitution systems into a category and studying its properties, and we develop the proofs of the results of the cited paper and our new ones in `UniMath`, a recent library of univalent mathematics formalized in the Coq theorem prover.

1998 ACM Subject Classification F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages

Keywords and phrases formalization of category theory, nested datatypes, Mendler-style recursion schemes, representation of substitution in languages with variable binding

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.2

1 Introduction

Given a first-order signature over some supply of variables, substitution is nearly a homomorphism: the substitution function commutes with all term-forming operations (however, at leaf positions, variables may get replaced by terms). But substitution also gives rise to a monad structure. For this, it is useful to see the variable supply of the terms as a parameter: writing TA for the set of terms over variable supply A (those variables that may occur free in the terms), parallel substitution associates with each substitution rule f, which is a function from A to TB, a substitution function $[f] : TA \to TB$, and for a given term $t : TA$, the term $t[f] : TB$ (notice the post-fix notation for function $[f]$) is the result of the parallel substitution that replaces each occurrence of a variable $x : A$ in t by $fx : TB$. In fact, the function T, the function that injects variables into terms, and the operation of parallel substitution together form a monad in the format of a Kleisli triple over the category of sets and functions. Notice that the types serve as a means of tracking the

* 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. This material is based upon work supported by the National Science Foundation under agreement Nos. DMS-1128155 and CMU 1150129-338510. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work has partly been funded by the CoqHoTT ERC Grant 637339.

© Benedikt Ahrens and Ralph Matthes;
licensed under Creative Commons License CC-BY

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 2; pp. 2:1–2:23

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

(names of) variables that may occur free in a term, the object syntax itself is untyped. The parameter A plays a more prominent role as soon as variable binding is allowed in the object syntax: for pure λ-calculus, bound and free variable occurrences have to be distinguished, and even the constructors of the object language relate terms with different variable supply, in particular λ-abstraction assumes an argument term where the newly bound variable is added to the variable supply (this will be seen with more details in Section 8.). Although parallel substitution $t[f]$ has to be defined with extra care to avoid capture of free variables of some fx by binders in t, it is still (modulo α-equivalence) nearly a homomorphism, and it still yields a monad [9]. However, the monad laws by themselves do not express the (nearly) "homomorphic nature" of substitution.

In previous work, Matthes and Uustalu [24] define a notion of "heterogeneous substitution system", the purpose of which is to axiomatize substitution and its desired properties. Such a substitution system is given by an algebra of a signature functor, equipped with an operation—which is to be thought of as substitution—that is compatible with the algebra structure map in a suitable sense. The term "heterogeneous" refers to the fact that the underlying notion of signature encompasses variable binding constructions and also explicit substitution a. k. a. flattening. The authors then prove that any heterogeneous substitution system gives rise to a monad; multiplication of the monad is derived from the "substitution" operation.

Furthermore, it is shown there that, under some assumptions on the underlying category, "substitution is for free" for both initial algebras as well as—maybe more surprisingly—for (the inverse of) final coalgebras: if the initial algebra, resp. terminal coalgebra, of a given signature functor exists, then it, resp. its inverse, can be augmented to a substitution system. Indeed, it was one of the design goals of the axiomatic framework of heterogeneous substitution systems to be applicable to *non-wellfounded* syntax as well as to wellfounded syntax, whereas related work (e.g., [15]) frequently only applies to wellfounded syntax.

Examples of substitution systems are thus given by the lambda calculus, with and without explicit flattening, but also by languages involving typing and *infinite* terms.

The goal of the present work is twofold:

Firstly, we extend the work by Matthes and Uustalu [24]; in particular, we introduce a natural notion of *morphisms* of heterogeneous substitution systems, thus arranging them into a category. We then show that the construction of a monad from a heterogeneous substitution system from [24] extends functorially to morphisms. Moreover, we prove that the substitution system obtained in [24] by equipping the initial algebra with a substitution operation, is initial in the corresponding category of substitution systems. This makes use of a general fusion law for generalized iteration [12]. As an example of the usefulness of our results, we express the resolution of explicit flattening of the lambda calculus as a(n initial) morphism of substitution systems.

A second part of our work is the formalization of some of our results in univalent type theory, more specifically, building upon the `UniMath` library [32]. This basis of our formalization is suitable in that it provides extensionality in a natural way and hereby avoids the use of setoids that would otherwise be inevitable; indeed, since our results are not about categories *in abstracto* but use general categorical concepts in more concrete instances such as the endofunctor category over a given category or its extension by a "point", we need extensionality axioms for the instantiation. We profit from the existing category theory library [4] in `UniMath`.

1.1 Related work

Related work is extensively discussed in Matthes and Uustalu's article [24].

In the meantime, monads and modules over monads, have been used by Hirschowitz and Maggesi [18, 19] to define models of syntax, and to give a categorical characterization thereof.

The notion of signature introduced in [24] and formalized in the present work is similar to that employed in Hirschowitz and Maggesi's most recent work [17]. One difference is that we do not, in the present work, insist on our signature functor to be ω-cocontinuous, since we do not worry about the existence of initial algebras, but assume them to exist. In our follow-up work with Mörtberg [6, 5] on the construction of initial algebras in sets, however, this condition is of the essence.

Voevodsky [31] constructs a C-system from a module over a relative monad on sets, which in turn can be obtained from a monad on Set^2 and a choice of a set. Of particular interest as input to this construction are "term monads" generated by 2-sorted binding signatures. The present work does not directly allow for the construction of such monads. The follow-up work [5] describes a variant (alluded to in Remark 9) of the necessary results formalized in the current work that can be used for the construction of such monads.

1.2 Synopsis

In Section 2 we first give an overview of the system we work in: `UniMath`. Afterwards, we review the definition of categories in `UniMath`, and finally, we show how the foundations are realized in the proof assistant Coq.

In Section 3 we define a few basic concepts and introduce notation.

In Section 4 we present "Generalized Iteration in Mendler-style", and a fusion law satisfied by this form of iteration. The presented results will be used in Section 7.

In Section 5 we review the notion of heterogeneous substitution system. Afterwards, we define a category of substitution systems and prove a few properties about that category.

In Section 6 we state one of the main results of [24], the construction of a monad from a substitution system. We then prove that the map thus constructed extends to morphisms and yields a faithful functor.

In Section 7 we state another of the important results of [24]: the construction of a substitution system from an initial algebra via Generalized Iteration in Mendler-style as presented in Section 4. We show that the obtained substitution system is again initial, using the fusion law stated in 4.

In Section 8, we construct a particular morphism of substitution systems, the underlying map of which "computes away" explicit substitution of lambda calculus.

Most of the results presented in this article, both by Matthes and Uustalu [24] and our new results, have been formalized, based on the `UniMath` library [32]. More precisely, all results except for Theorem 22 and Lemmas 25 and 21 are proved in our formalization; Section 9 provides some technical details about our library.

2 Univalent Mathematics

The original article [24] is written without referring to a specific foundation of mathematics. Indeed, the authors use purely categorical methods to derive their results.

Our analysis and continuation of that article takes place in a *type-theoretic* foundation augmented by Voevodsky's *Univalence Axiom*. Specifically, we are working in the `UniMath` language and library, based on Voevodsky's *Foundations* [30].

2.1 About `UniMath`

`UniMath` is based on an intensional type theory augmented by Voevodsky's univalence axiom. In the following, we give a brief overview of the type constructors available in `UniMath`:

For a dependent type B over A there is the dependent pair type $\sum_{x:A} B(x)$, elements of which are dependent pairs (a, p) where $a : A$ and $p : B(a)$. The type $\prod_{x:A} B(x)$ is the type of dependent functions from A to B, that is, a function $f : \prod_{x:A} B(x)$ maps $a : A$ into the type $B(a)$. Special, non-dependent, cases of the aforementioned constructors are the cartesian product $A \times B$ and the function type $A \to B$.

For any type A and $a, b : A$ elements of A, there is the Martin-Löf identity type $a =_A b$ of "(propositional) equalities" between a and b. We often omit the subscript A and hence simply write $a = b$.

The Univalence Axiom identifies identities between types with equivalences between types, see [29, Axiom 2.10.3]. In this work, we do not use the full strength of the Univalence Axiom, but only function extensionality, a consequence of the Univalence Axiom.

In `UniMath`, there is an internal notion of propositions and sets. A type A is called a *proposition* if it satisfies the (propositional) "proof irrelevance" principle, that is, if one can construct a term of type

$$\mathsf{isProp}(A) := \prod_{x,y:A} x = y \enspace .$$

Furthermore, a type A is called a *set* if all of its identity types are propositions, that is, if one can construct a term of type

$$\mathsf{isSet}(A) := \prod_{x,y:A} \mathsf{isProp}(x = y) \enspace .$$

These two definitions are actually special cases of a more general definition of **homotopy levels** of types. However, the general definition will not be of use in this article, and can be consulted in [29]. We call **proposition** any type that is a proposition in this sense, that is, any element of $\mathsf{Prop} := \sum_{X:\mathcal{U}} \mathsf{isProp}(X)$, and similarly for **sets**.

Technically, the `UniMath` language is a subset of the language of the Coq proof assistant [13]: In order to simulate working in the theory described above, we do not use the full language Coq provides, but restrict ourselves to the language constructors mentioned there. In particular, there is no use of inductive types besides that of the natural numbers, and of the identity type and the type of dependent pairs, both of which are not primitives in Coq, but instead implemented via the general `Inductive` vernacular. Furthermore, record types are not used in `UniMath`; bundling of structures is instead implemented via (iterated) Sigma types.

The proof assistant Coq has recently gained a new form of universe polymorphism [28]. Unfortunately, this universe management is not powerful enough for our purposes. In particular, it does not implement a form of *resizing* rule that is needed for some impredicative encodings of constructions—propositional truncation in particular, as described by Voevodsky [30, Section 4]. To implement this resizing rule in Coq, we disable its checking of universes via a flag `-type-in-type` passed to the program. We hence work in a formally inconsistent system, and we have to check manually that we do not actually exploit that inconsistency.

Another difference to standard Coq is our use of the `-indices-matter` flag. This flag ensures that the identity type associated to a type A, lives in the same universe as the type A itself. By default, without that flag, Coq would put the identity type into the universe `Prop` (not to be confounded with the **homotopy level** of propositions).

"Higher Inductive Types" (HITs), described, e. g., in the HoTT book [29], are not part of the axiomatically given type constructors of `UniMath`.

The Univalence Axiom is implemented in `UniMath` via the `Axiom` vernacular of Coq. This leads to potentially non-normalizing terms, when using the axiom or any of its consequences—such as function extensionality. We do not experience any problems related to non-normalization, since we only use the univalence axiom (indirectly by using function extensionality) for proving propositions, not for specifying operations.

2.2 Category Theory in Univalent Type Theory

Category theory in univalent type theory has been developed in [4]. A category $\mathcal{C}$ is given by

- a type $\mathcal{C}_0$ of objects;
- for any $a, b : \mathcal{C}_0$, a set $\mathcal{C}(a, b)$ of morphisms from a to b;
- for any $a : \mathcal{C}_0$, an identity morphism $\mathsf{id}(a) : \mathcal{C}(a, a)$;
- for any $a, b, c : \mathcal{C}_0$, a composition function $\mathcal{C}(a, b) \to \mathcal{C}(b, c) \to \mathcal{C}(a, c)$, written $f \mapsto g \mapsto g \circ f$;
- for any $a, b : \mathcal{C}_0$ and $f : \mathcal{C}(a, b)$, we have $f \circ \mathsf{id}(a) = f$ and $\mathsf{id}(b) \circ f = f$;
- for any $a, b, c, d : A$ and $f : \mathcal{C}(a, b)$, $g : \mathcal{C}(b, c)$, $h : \mathcal{C}(c, d)$, we have $h \circ (g \circ f) = (h \circ g) \circ f$.

Note that we ask the hom-types $\mathcal{C}(a, b)$ of a category to be **sets**. This requirement enforces that the categorical axioms—which talk about equality of arrows—form propositions.

▶ Nota bene. There is an important difference between categories as usually formalized in intensional type theory and categories as considered in [4]: in intensional type theory, categories are usually defined to come with a custom equivalence relation on the types of morphisms, which is to be read as equality relation on morphisms, specified for each category individually (see, e. g., [21]). These categories are sometimes referred to as "E-categories" [27].

In [4], however, the authors consider morphisms of a category modulo equality as given by the identity type. That this is feasible is due to the extensional features that the univalence axiom adds to type theory, in particular, function extensionality.

In [4], an additional property of categories is studied: for any category $\mathcal{C}$, define a family of maps

$$\mathsf{idtoiso} : \prod_{a,b:\mathcal{C}_0} (a = b) \to \mathsf{iso}(a, b) \quad .$$

This family of maps is defined by identity elimination, mapping $\mathsf{refl}_a : a = a$ to the identity isomorphism on a.

A category $\mathcal{C}$ is called **univalent**, if for any $a, b : \mathcal{C}_0$, the map $\mathsf{idtoiso}_{a,b}$ is an equivalence.

An important remark about naming: in [4], the univalence condition above is part of the definition of a category—the term "precategory" is employed for categories that are not necessarily univalent. That is, the authors of [4] use the terms "precategory" and "category" for what we call "category" and "univalent category" in the present article, respectively.

For the purposes of the present article, the univalence condition on categories is not essential. Indeed, no other result depends on Theorem 22. We thus choose to de-emphasize the importance of the univalence condition for categories by deviating from the naming of [4], and instead to make it explicit when considering categories that satisfy univalence.

3 Preliminaries

Categories, functors and natural transformations are defined in [4]. Some more concepts and notation are defined in the following:

For functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$, we write $G \cdot F : \mathcal{C} \to \mathcal{E}$ for their composition. We use the same notation for composition of a functor with a natural transformation (sometimes called "whiskering"), as in $\tau \cdot F$ and $G \cdot \tau$.

▶ **Definition 1** (Pointed functors)**.** Let $\mathcal{C}$ be a category. We denote by $\mathsf{Ptd}(\mathcal{C})$ the category of pointed endofunctors on $\mathcal{C}$, an object of which is a pair (X, η) of an endofunctor X on $\mathcal{C}$ and a natural transformation $\eta : \mathsf{Id} \to X$, called a "point" of X, where Id is the identity functor on $\mathcal{C}$. Morphisms of pointed functors are natural transformations between the underlying endofunctors that are compatible with the chosen points. Call U the forgetful functor from $\mathsf{Ptd}(\mathcal{C})$ to the underlying endofunctor category $[\mathcal{C}, \mathcal{C}]$ (in particular, for a morphism f, Uf is f, but its compatibility with the points is not taken into account in the type information—justifying to confuse Uf and f in the rest of the paper).

▶ **Definition 2** (Monoidal structure on functor categories)**.** The monoidal structure on the endofunctor category $[\mathcal{C}, \mathcal{C}]$ given by composition extends to $\mathsf{Ptd}(\mathcal{C})$. We denote by $\alpha_{X,Y,Z}$: $X \cdot (Y \cdot Z) \simeq (X \cdot Y) \cdot Z$, $\rho_X : \mathsf{Id} \cdot X \simeq X$ and $\lambda_X : X \cdot \mathsf{Id} \simeq X$ the monoidal isomorphisms.

Note that the associator and unitor isomorphisms are given by families of identity morphisms, and thus do not carry any information at all; they are merely needed to formally adjust the type of source and target functors of the natural transformations involved.

▶ **Remark 3.** In [24], the authors implicitly assume the monoidal structures of composition on $[\mathcal{C}, \mathcal{C}]$ and $\mathsf{Ptd}(\mathcal{C})$ to be strict. In univalent type theory, we have, e. g., that $F \cdot \mathsf{Id}$ is not convertible to F as a functor, but the two functors are convertible pointwise on objects and morphisms. This in turn entails that for $\rho_F : \mathsf{Id} \cdot F \to F$, the type $\rho_F = 1_F$ is well-typed. Note, however, that for an abstract functor on endofunctors H, the type $H(\rho_F) = H(1_F)$ is not well-typed.

In our definition of signatures (Definition 12) the associators and unitors do occur "under a functor application", where we cannot pretend (or even state) that they are identity morphisms. For reasons of symmetry, we hence decide to consider the monoidal structure of composition as non-strict, inserting the associator and unitors also in cases where this would not be necessary. In particular, we explicitly insert them in the strength laws of Definition 12 on the right-most position on the right hand side, respectively.

▶ **Definition 4** (Algebras of a functor)**.** For an endofunctor $F : \mathcal{C} \to \mathcal{C}$, the category $\mathsf{Alg}(F)$ of **algebras** has, as objects, pairs (X, α) of an object $X : \mathcal{C}_0$ and a morphism $\alpha : \mathcal{C}(FX, X)$. For a given algebra (X, α), we call X the **(algebra) carrier** of the algebra. A morphism $f : \mathsf{Alg}(F)\big((X, \alpha), (X', \alpha')\big)$ is given by a morphism $f : \mathcal{C}(X, X')$ such that $f \circ \alpha = \alpha' \circ Ff$.

▶ **Remark 5.** We are using the arrow symbol "$\to$" for three different things:

1. morphisms $f : c \to d$ in a category, as shorthand for $f : \mathcal{C}(c, d)$ (hence in particular for natural transformations as morphisms in functor categories);
2. functors $F : \mathcal{C} \to \mathcal{D}$ between categories; and
3. type-theoretic functions $f : A \to B$.

Information on what the arrow denotes in each occurrence will be deducible from the context.

▶ **Definition 6** (Monads)**.** For a category $\mathcal{C}$, the category $\mathsf{Mon}(\mathcal{C})$ of **monads** has, as objects, triples (T, η, μ) of an endofunctor T of $\mathcal{C}$, and natural transformations $\eta : \mathsf{Id} \to T$ and

$\mu : T \cdot T \to T$ (using our convention on natural transformations), subject to the usual monad laws. A morphism $f : \mathsf{Mon}(\mathcal{C})\big((T,\eta,\mu),(T',\eta',\mu')\big)$ is given by a natural transformation $f : T \to T'$, subject to the usual compatibility conditions.

Notice that we follow [24] in taking monad multiplication μ as third component of a monad and not the Kleisli extension operation that is more widespread in computer science literature.

▶ **Definition 7.** Given $d : \mathcal{D}$ and a category $\mathcal{C}$, we call $\underline{d} : \mathcal{C} \to \mathcal{D}$ the functor that is constantly d and id_d on objects and morphisms, respectively. This notation hides the category $\mathcal{C}$, which will usually be deducible from the context. In this article, $\mathcal{C}$ will always be $\mathcal{D}$.

4 Generalized Iteration in Mendler-style and Fusion Law

In this section we discuss "generalized iteration in Mendler-style" and a fusion law that one can prove for this iteration scheme. Both the iteration scheme and the fusion law are used in Section 7.

▶ **Lemma 8** (Generalized iteration in Mendler-style (Theorem 2 of [12] by Bird and Paterson))**.** *Let $\mathcal{C}$ be a category, and let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor on $\mathcal{C}$. Suppose $(\mu F, \mathsf{in})$ is the initial algebra of F. Let $\mathcal{D}$ be another category, and let $\mathcal{C} : L \dashv R : \mathcal{D}$ be an adjunction. Let $X : \mathcal{D}_0$ be an object of $\mathcal{D}$, and let*

$$\Psi : \mathcal{D}(L-, X) \to \mathcal{D}(L(F-), X)$$

be a natural transformation. Then there is exactly one morphism $h : L(\mu F) \to X$ such that the following diagram commutes:

$$\begin{array}{ccc} L(F(\mu F)) & \xrightarrow{L\mathsf{in}} & L(\mu F) \\ & \searrow^{\Psi_{\mu F}(h)} & \downarrow h \\ & & X \end{array}$$

We call $\mathsf{It}^L_F(\,\Psi\,) := h$ the unique morphism thus specified.

The link with the work by Mendler [25] is not made in the original proof [12, Thm. 2] of the lemma. The presentation in [12] is very much oriented towards functional programming. In their notation, the natural transformation Ψ would be typed as

$$\Psi :: \forall A. \, (LA \to X) \to (L(FA) \to X) \ .$$

▶ **Remark 9.** The existence of the right adjoint R for L is rather a matter of technical convenience: it can be replaced by asking for the preservation of colimits of ω-chains by F and L and the preservation of initial objects by L [12, Theorem 1]. We do not pursue that alternative in the present work.

In [24], only a specialized form of generalized iteration in Mendler-style is used that is called "generalized iteration" (again with no hint to Mendler's work—see our remarks in Section 7 on the connection). The specialization consists in taking only natural transformations Ψ of a specific form, so that Ψ disappears from the formulation (as explained in [24]). In fact, we do not need the fuller generality of generalized iteration in Mendler-style in Sections 7 and 8. However, the formulation of the fusion law to come next is more natural in the more general setting. No fusion law was needed in [24] since no *morphisms* of heterogeneous substitution systems were considered there.

The next lemma shows a sufficient condition for two applications of the iterator $\mathsf{It}(\,-\,)$ to be related:

▶ **Lemma 10** (Fusion law). *Suppose the data as given in Lemma 8. Additionally, let $L' : \mathcal{C} \to \mathcal{D}$ be a functor, $X' : \mathcal{D}_0$ be an object of $\mathcal{D}$, let*

$$\Psi' : \mathcal{D}(L'-, X') \to \mathcal{D}(L'(F-), X')$$

be a natural transformation with type analogous to that of Ψ, and let

$$\Phi : \mathcal{D}(L-, X) \to \mathcal{D}(L'-, X')$$

be a natural transformation. Then we have

$$\Phi_{\mu F}\big(\mathsf{It}^{L}_{F}(\,\Psi\,)\big) = \mathsf{It}^{L'}_{F}(\,\Psi'\,) \qquad \textit{if} \qquad \Phi_{F\mu F} \circ \Psi_{\mu F} = \Psi'_{\mu F} \circ \Phi_{\mu F} \ .$$

The name "fusion law" is wide-spread in functional programming for means to eliminate the creation of some extra datastructure. Here, the subsequent calculation of $\Phi_{\mu F}$ for the result $\mathsf{It}^{L}_{F}(\,\Psi\,)$ of the iteration over μF is "fused" into one single iteration over μF—the right-hand side of the conclusion.

The version of this fusion law with X and X' the same object of $\mathcal{D}$ and instantiated to the special situation of generalized folds (see Section 7) has been found by Bird and Paterson [12] (see right before their Theorem 1). While we will only use the fusion law for generalized folds (in Section 7), it is necessary to have the freedom to choose X and X' separately. The proof itself is a matter of verifying that the left-hand side satisfies the defining equation (embodied in the commuting diagram in Lemma 8) of the right-hand side. This also settles existence of the right-hand side—thus avoiding the need for a right adjoint for L', which would have allowed us to invoke Lemma 8 also for Ψ'. (In our formalization, we did not implement this subtlety. Instead, we require a right adjoint for L', in order to use the definition of the $\mathsf{It}(\,-\,)$ operator underlying the formalization of Lemma 8.)

5 The Category of Heterogeneous Substitution Systems

In [24], implicitly there is a notion of signature. Here, we make this definition explicit and adapt it to the lack of strictness of our monoidal structures on endofunctors (see Definition 2).

▶ **Definition 11** (Relative strength). Let $(\mathcal{V}, \otimes, I)$ and $(\mathcal{W}, \bullet, E)$ be monoidal categories, and let

$$(U, \varphi, \varphi_0) : (\mathcal{W}, \bullet, E) \to (\mathcal{V}, \otimes, I)$$

be a strong monoidal functor, that is, $\varphi_{w,w'} : Uw \otimes Uw' \cong U(w \bullet w')$ and $\varphi_I : I \cong UE$. Let $F : \mathcal{V} \to \mathcal{V}$ be a functor. A **tensorial strength for F relative to** (U, φ, φ_I) is a natural transformation

$$\beta_{w,v} : Uw \otimes Fv \to F(Uw \otimes v)$$

such that the following diagrams commute for any $w, w' : \mathcal{W}_0$ and $v : \mathcal{V}_0$:

$$\begin{array}{ccccc}
U(w \bullet w') \otimes Fv & \xrightarrow{\beta_{w \bullet w', v}} & F(U(w \bullet w') \otimes v) & \xrightarrow{F(\varphi^{-1}_{w,w'} \otimes 1_v)} & F((Uw \otimes Uw') \otimes v) \\
\Big\downarrow{\scriptstyle \varphi^{-1}_{w,w'} \otimes 1_{Fv}} & & & & \\
(Uw \otimes Uw') \otimes Fv & & & & \Big\downarrow{\scriptstyle F(\alpha_{Uw,Uw',v})} \\
\Big\downarrow{\scriptstyle \alpha_{Uw,Uw',Fv}} & & & & \\
Uw \otimes (Uw' \otimes Fv) & \xrightarrow[1_{Uw} \otimes \beta_{w',v}]{} & Uw \otimes F(Uw' \otimes v) & \xrightarrow[\beta_{w,Uw' \otimes v}]{} & F(Uw \otimes (Uw' \otimes v))
\end{array}$$

and

$$\begin{array}{ccc} UE \otimes Fv & \xrightarrow{\beta_{E,v}} & F(UE \otimes v) \\ \downarrow{\scriptstyle \varphi_I^{-1} \otimes 1_{Fv}} & & \downarrow{\scriptstyle F(\varphi_I^{-1} \otimes 1_v)} \\ I \otimes Fv & & F(I \otimes v) \\ & \searrow^{\lambda_{Fv}} \quad \swarrow_{F(\lambda_v)} & \\ & Fv & \end{array}$$

This definition is an instance of a broader definition of strength by Fiore [14, I.1.2]. Modulo order of arguments, our relative strength is a $\mathcal{W}$-strength of type $(\mathcal{V}, \odot) \to (\mathcal{V}, \odot)$, with the action $\odot$ induced by U, a construction that is also described by Fiore in the cited section.

▶ Nota bene. We find it important to mention two possible sources of confusion:

1. The notion of tensorial strength *relative to a strong monoidal functor* of Definition 11 is inspired by the notion of monad *relative to a functor* [7]. However, it is not the same as the concept of a *strength for a monad T relative to a functor $J : \mathcal{C} \to \mathcal{D}$*.
2. Note that the adjective "strong" is used in two different ways in the literature:
   - A *strong functor (or monad)* is a functor (or monad) equipped with a strength.
   - A *strong monoidal functor* is a monoidal functor for which the commutator morphisms $\varphi_{w,w'}$ and φ_I are isomorphisms, as recalled above.

We are interested in tensorial strengths for functors H relative to the forgetful functor $U : \mathsf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$ of Definition 1 that "forgets" the points of pointed functors. That particular functor is strict in the sense that φ and φ_I are identities. We hence set $(Z, e) \bullet (Z', e') := (Z' \cdot Z, e' \cdot e)$ and $X \otimes X' := X' \cdot X$ for the purpose of the following definition. Unfortunately, there is a mismatch between the order of the arguments of β in Definition 11 on the one hand—which is the order naturally arising when generalizing the traditional definition of strength—and the order in which Matthes and Uustalu [24] give the arguments to their instance of such a relative tensorial strength—called θ—in the following definition. We choose to retain compatibility with [24]:

▶ **Definition 12** (Signature)**.** Given a category $\mathcal{C}$, a **signature with strength** is a pair (H, θ) of an endofunctor H on $[\mathcal{C}, \mathcal{C}]$ and a tensorial strength for H relative to $U : \mathsf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$, that is, a natural transformation $\theta : (H-) \cdot U{\sim} \to H(- \cdot U{\sim})$ between functors $[\mathcal{C}, \mathcal{C}] \times \mathsf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$ such that

$$\theta_{X,\mathsf{id}} = H(\lambda_X^{-1}) \circ \lambda_{HX} \quad \text{and}$$

$$\theta_{X,(Z' \cdot Z, e' \cdot e)} = H(\alpha^{-1}_{X,Z',Z}) \circ \theta_{X \cdot Z',(Z,e)} \circ (\theta_{X,(Z',e')} \cdot Z) \circ \alpha_{HX,Z',Z} \ .$$

We loosely refer to θ as the **strength of the signature** (H, θ).

In practice, a signature is given by a family of *arities*, each arity specifying the type of a term constructor. The above definition of signature is modular in the sense that building a signature from arities corresponds to taking an amalgamated sum. This is explained in detail in Section 8, to which we refer for an example of signature.

Note that while the definition of signature with strength does not require the base category $\mathcal{C}$ to have coproducts, this is a requirement for most signatures with strength that we consider in practice, and in particular for the example of Section 8. It also is a requirement for the definition of "models" of signatures with strength, see Definition 15.

▶ **Convention 13.** From now on, we assume the category $\mathcal{C}$ to have (specified) coproducts. We denote by $\mathsf{inl}_{A,B} : A \to A + B$ and $\mathsf{inr}_{A,B} : B \to A + B$ the maps into the coproduct. We omit the subscripts of inl and inr when possible without ambiguity.

▶ **Remark 14.** The notion of signature with strength introduced in Definition 12 encompasses "polynomial" signatures like the ones described in [15] and [26]. In fact, it is strictly more general in that it also encompasses the arity of explicit flattening—the Example 35 we discuss in detail in Section 8—that is not captured by the other works mentioned above.

For a given signature (H, θ), we are interested in $(\underline{\mathsf{Id}} + H)$-algebras (T, α). For such an algebra, the natural transformation $\alpha : \mathsf{Id} + HT \to T$ decomposes into two $[\mathcal{C}, \mathcal{C}]$-morphisms $\eta : \mathsf{Id} \to T$, $\tau : HT \to T$ defined by

$$\eta = \alpha \circ \mathsf{inl}_{\mathsf{Id},HT} \qquad \text{and} \qquad \tau = \alpha \circ \mathsf{inr}_{\mathsf{Id},HT} \ . \tag{1}$$

The pair (T, η) is an object in the category of pointed functors (see Definition 1).

Intuitively, in the case where $\mathcal{C} = \mathsf{Set}$, the transformation η corresponds to viewing variables $x : X$ as "terms", that is, as elements of TX, whereas $\tau : HT \to T$ represents the operations specified by the signature functor H.

▶ **Definition 15** (Def. 5 of [24], Heterogeneous substitution system of a signature)**.** We call (T, α) a *heterogeneous substitution system* for (H, θ), if, for every $\mathsf{Ptd}(\mathcal{C})$-morphism $f : (Z, e) \to (T, \eta)$, there exists a unique $[\mathcal{C}, \mathcal{C}]$-morphism $h : T \cdot Z \to T$, denoted $\{f\}$, satisfying

$$\begin{array}{ccc} Z + (HT)\cdot Z & \xrightarrow{\alpha\cdot Z} & T\cdot Z \\ \downarrow{\scriptstyle \mathsf{id}+\theta_{T,(Z,e)}} & & \\ Z + H(T\cdot Z) & & \downarrow{\scriptstyle h} \\ \downarrow{\scriptstyle \mathsf{id}+Hh} & & \\ Z + HT & \xrightarrow{[f,\tau]} & T \end{array} \quad \text{i.e.,} \quad \begin{array}{ccccc} Z & \xrightarrow{\eta\cdot Z} & T\cdot Z & \xleftarrow{\tau\cdot Z} & (HT)\cdot Z \\ & {\scriptstyle f}\searrow & & & \downarrow{\scriptstyle \theta_{T,(Z,e)}} \\ & & \downarrow{\scriptstyle h} & & H(T\cdot Z) \\ & & & & \downarrow{\scriptstyle Hh} \\ & & T & \xleftarrow{\tau} & HT \end{array}$$

For a heterogeneous substitution system $(T, \alpha, \{-\})$, we call T its **carrier**, thus extending the convention of Definition 4.

Notice that the quantification is implicitly also over all pointed endofunctors (Z, e) on $\mathcal{C}$.

▶ Nota bene. Having freedom in the choice of parameter f (and its domain) is particularly important for Theorem 26, see Section 6. In its proof (not shown in this paper), monad multiplication and one of the monad laws is obtained from the existence of a solution in the case that f is the identity, while the other monad laws are derived from uniqueness for two other choices of f.

In the following, we sometimes omit the word "heterogeneous" when talking about heterogeneous substitution systems. We refer to the operation $\{-\}$ by "substitution".

▶ **Remark 16.** Being equipped with a substitution operation $\{-\}$ is a proposition on $(\underline{\mathsf{Id}} + H)$-algebras.

The statement of the following lemma is mentioned, but not proven in [24]:

▶ **Lemma 17.** *The operation $\{-\}$ is a natural transformation*

$$\mathsf{Ptd}(-, (T, \eta)) \to [\mathcal{C}, \mathcal{C}](T \cdot U-, T) \ .$$

▶ **Definition 18** (Category of substitution systems). Given (H, θ) as before, the category $\mathsf{hss}(H, \theta)$ has, as objects, heterogeneous substitution systems as in Definition 15. A morphism of substitution systems is an algebra morphism that is compatible with the substitution $\{-\}$ on either side. In terms of η and τ as defined in Equation (1), a morphism from $(T, \eta, \tau, \{\})$ to $(T', \eta', \tau', \{\}')$ is a natural transformation $\beta : T \to T'$ such that the following diagrams commute:

$$\begin{array}{ccc} \mathsf{Id} & \xrightarrow{\eta} & T \\ & \searrow^{\eta'} & \downarrow{\scriptstyle \beta} \\ & & T' \end{array} \qquad \begin{array}{ccc} HT & \xrightarrow{\tau} & T \\ {\scriptstyle H\beta}\downarrow & & \downarrow{\scriptstyle \beta} \\ HT' & \xrightarrow[\tau']{} & T' \end{array} \qquad \begin{array}{ccc} T \cdot Z & \xrightarrow{\{f\}} & T \\ {\scriptstyle \beta \cdot Z}\downarrow & & \downarrow{\scriptstyle \beta} \\ T' \cdot Z & \xrightarrow[\{\beta \circ f\}']{} & T' \end{array}$$

Here, the first and second diagram express the property of β being an algebra morphism, and the third diagram expresses compatibility of β with substitution on either side.

Note that the composite $\beta \circ f$ in the last diagram is the composite in the category of **pointed** endofunctors, that is, the definition of that composite uses commutativity of the first diagram.

▶ **Remark 19.** Similarly to Remark 16, being compatible with the substitution on either side is a proposition on algebra morphisms.

We now study the category $\mathsf{hss}(H, \theta)$ of substitution systems associated to a signature with strength in more detail, in particular with respect to the particular foundations we are working in. The main objective of the rest of the section is Theorem 22: the category $\mathsf{hss}(H, \theta)$ is univalent if the base category $\mathcal{C}$ is.

Remarks 16 and 19 together show that the category of $\mathsf{hss}(H, \theta)$ can be obtained as a *subcategory* of the category of $(\underline{\mathsf{Id}} + H)$-algebras in the following sense:

▶ **Definition 20.** A **subcategory** of a category $\mathcal{C}$ is given by a predicate $P : \mathcal{C}_0 \to \mathsf{Prop}$ and a family of predicates $P_{a,b} : P(a) \times P(b) \times \mathcal{C}(a, b) \to \mathsf{Prop}$ that is closed under identity and composition in the sense that

- for any $a : \mathcal{C}_0$ satisfying P, we have a proof of $P_{a,a}(\mathsf{id}(a))$ and
- for any $a, b, c : \mathcal{C}_0$ satisfying P, and for any $f : \mathcal{C}(a, b)$ and $g : \mathcal{C}(b, c)$, we have a map $P_{a,b}(f) \to P_{b,c}(g) \to P_{a,c}(g \circ f)$.

We suppress the arguments of type $P(a)$ and $P(b)$ when discussing the predicate $P_{a,b}(f)$, since those arguments are unique.

A subcategory of $\mathcal{C}$ is—better, gives rise to—a category $\mathcal{C}_P$; objects are of the form $\sum_{x:\mathcal{C}_0} P(x)$, and morphisms $(f, p_f) : \mathcal{C}_P\big((a, p_a), (b, p_b)\big)$ are pairs of a morphism $f : \mathcal{C}(a, b)$ of $\mathcal{C}$ together with a proof $p : P_{a,b}(f)$.

Given a signature (H, θ), define a subcategory of the category of $(\underline{\mathsf{Id}} + H)$-algebras via the predicates of Remarks 16 and 19. The resulting category is clearly isomorphic to $\mathsf{hss}(H, \theta)$ in the sense of [4, Definition 6.9].

Note that isomorphic categories are propositionally equal [4, Definition 6.16], and hence share all properties definable in type theory. We thus give up the distinction between the category $\mathsf{hss}(H, \theta)$ and the subcategory of $(\underline{\mathsf{Id}} + H)$-algebras it is isomorphic to.

A subcategory is called **replete**, when it is closed under isomorphism, that is, when, for $f : \mathsf{iso}_{\mathcal{C}}(a, b)$ and $P(a)$, it follows that $P(b)$ and $P_{a,b}(f)$.

▶ **Lemma 21.** *The category $\mathsf{hss}(H, \theta)$ is a replete subcategory of the category of $(\underline{\mathsf{Id}} + H)$-algebras.*

Proof. Given a substitution system $(T, \alpha, \{-\})$, an algebra (T', α') and an algebra isomorphism $\beta : (T, \alpha) \to (T', \alpha')$, we define substitution $\{-\}'$ on (T', α') as follows: for a given pointed morphism $f : (Z, e) \to (T', \eta')$, we define $\{f\}'$ as the composition

$$\{f\}' := \beta \circ \{\beta^{-1} \circ f\} \circ \beta^{-1} \cdot Z \quad : \quad T' \leftarrow T \leftarrow T \cdot Z \leftarrow T' \cdot Z$$

The morphism $\{f\}'$ thus defined satisfies the equations of Definition 15,

$$f = \{f\}' \circ \eta' \cdot Z$$
$$\{f\}' \circ \tau' \cdot Z = \tau' \circ H(\{f\}') \circ \theta_{T',(Z,e)} \;\; ;$$

the calculation is routine. Concerning the uniqueness of $\{f\}'$, suppose h such that these equations with h in place of $\{f\}'$ are satisfied. We have to show that $h = \beta \circ \{\beta^{-1} \circ f\} \circ \beta^{-1} \cdot Z$. Equivalently, one can show that

$$\{\beta^{-1} \circ f\} = \beta^{-1} \circ h \circ \beta \cdot Z \;\; , \tag{2}$$

which follows from the uniqueness of $\{-\}$: it suffices to show that the right-hand side of (2) satisfies the equations involving η and τ. We thus have equipped (T', α') with a (necessarily unique) substitution operation.

The fact that β is compatible with $\{-\}$ and $\{-\}'$, and hence in the subcategory, is a routine calculation. ◀

▶ **Theorem 22.** *The category* $\mathsf{hss}(H, \theta)$ *is univalent if* $\mathcal{C}$ *is.*

Proof. Combine Lemmas 23, 25, 24 below and Lemma 21 above. More precisely, if $\mathcal{C}$ is univalent, so is $[\mathcal{C}, \mathcal{C}]$, and thus also the category of $(\underline{\mathsf{Id}} + H)$-algebras on $[\mathcal{C}, \mathcal{C}]$. Finally, the category $\mathsf{hss}(H, \theta)$ is univalent as a replete subcategory of the category of $(\underline{\mathsf{Id}} + H)$-algebras. ◀

The following lemmas state closure properties of the property of being univalent:

▶ **Lemma 23.** *The category of algebras of a functor* $F : \mathcal{C} \to \mathcal{C}$ *is univalent if* $\mathcal{C}$ *is.*

Proof. This lemma is proved in the file `CategoryTheory/FunctorAlgebras.v` of the `UniMath` library. ◀

The next lemma is originally due to Hofmann and Streicher [20]; and is also proved in Thm. 4.5 of [4]:

▶ **Lemma 24.** *The category of functors* $[\mathcal{C}, \mathcal{D}]$ *is univalent if the target category* $\mathcal{D}$ *is.*

The category of substitution systems contains all the isomorphisms of the category of $(\underline{\mathsf{Id}} + H)$-algebras, for which source and target are substitution systems.

This is sufficient to inherit univalence from the category of algebras:

▶ **Lemma 25.** *Let* $\mathcal{C}$ *be a univalent category and let* $P : \mathcal{C}_0 \to \mathsf{Prop}$ *and* $P_{a,b} : \mathcal{C}(a, b) \to \mathsf{Prop}$ *define a subcategory* $\mathcal{C}_P$ *of* $\mathcal{C}$*. Then* $\mathcal{C}_P$ *is univalent if, for any objects* (a, p_a) *and* (b, p_b) *of* $\mathcal{C}_P$*, and for any* iso*morphism* $f : \mathsf{iso}_{\mathcal{C}}(a, b)$ *from* a *to* b*, we have* $P_{a,b}(f)$*.*

In particular, replete subcategories of univalent categories are univalent.

Proof. For (a, p_a) and (b, p_b) objects of $\mathcal{C}_P$, we have

$$(a, p_a) =_{\mathcal{C}_P} (b, p_b) \;\; \simeq \;\; a =_{\mathcal{C}} b \;\; \simeq \;\; \mathsf{iso}_{\mathcal{C}}(a, b) \;\; \simeq \;\; \mathsf{iso}_{\mathcal{C}_P}((a, p_a), (b, p_b))$$

and this equivalence, from left to right, is equal to $\mathsf{idtoiso}$. ◀

This concludes our study of the category of substitution systems associated to a signature with strength.

6 From Substitution Systems to Monads

One of the most important results of Matthes and Uustalu's work [24] is the construction of a monad from any substitution system:

▶ **Theorem 26** ([24], Thm. 10)**.** *If an* $(\underline{\mathsf{Id}} + H)$*-algebra* (T, α) *forms a heterogeneous substitution system for* (H, θ) *for some* θ*, then* $(T, \eta, \{\mathsf{id}_{(T,\eta)}\})$ *is a monad.*

See Section 9 for some comments on technical challenges we had to overcome for the formalization of its proof.

It is natural to ask whether this map extends to *morphisms*, and indeed it does:

▶ **Theorem 27.** *The map from heterogeneous substitution systems to monads defined in [24, Thm. 10] is the object map of a functor* $\mathsf{hss}(H, \theta) \to \mathsf{Mon}(\mathcal{C})$*.*

Proof. Given any morphism $\beta : (T, \eta, \tau, \{\}) \to (T', \eta', \tau', \{\}')$ of substitution systems, the underlying natural transformation $\beta : T \to T'$ needs to be proven compatible with the multiplications $\mu^T := \{\mathsf{id}_{(T,\eta)}\}$ and $\mu^{T'}$ of the monadic structures on T and T' defined in [24, Thm. 10]. This is an easy consequence of the compatibility of β with $\{\}$ and $\{\}'$. ◀

▶ Nota bene. Hirschowitz and Maggesi [17] observe that any signature with strength (H, θ) yields a module transformer: given a module (M, ρ) over a monad R, then HM is again a module over R. The module multiplication of HM is defined as $H\rho \circ \theta_{M,R}$.

Given a heterogeneous substitution system $(T, [\eta, \tau], \{-\})$, the monad $(T, \eta, \{\mathsf{id}_{(T,\eta)}\})$ (from now on denoted just T) constructed in Theorem 26 can be viewed as a module over itself. Applying the aforementioned module transformer yields a module (with underlying functor) HT over T.

Further along those lines, we note that $\tau : HT \to T$ is a module morphism in the sense of [19] between the modules over T thus defined. The equation establishing this is an instance of the family of equations on heterogeneous substitution systems ruling the case of constructors—that is, of the family of squares concerning the morphism τ in Definition 15: the instance where $(Z, e) := (T, \eta)$ and $f := \mathsf{id}_{(T,\eta)}$.

One might thus say that being a substitution system includes by definition that τ is a module morphism. However, this only concerns the equation to be fulfilled. It does not suggest a modification of the notion of substitution systems: when defining substitution systems, the modules for which τ is a morphism are not available, and thus, it could not even be stated in the definition of substitution systems that τ ought to be a morphism between these modules.

The functor from substitution systems to monads is faithful, but not full. Intuitively, the lack of fullness stems from the fact that the axioms of a monad morphism do not specify compatibility of the mapping with the "inner nodes" of an expression, but only at the leaves, that is, in the case of a variable.

▶ **Lemma 28.** *The functor of Theorem 27 is faithful.*

Proof. Two parallel monad morphisms are equal if their underlying natural transformations are, and the analogous statement is true for morphisms of substitution systems. ◀

▶ **Remark 29.** The functor of Theorem 27 is not full. For instance, choose $\mathcal{C} = \mathsf{Set}$, and take a signature with two copies app and app' (of the same arity) of an "application" constructor, see Definition 33 in Section 8. Take the initial substitution system associated to that signature with strength (as constructed via Theorems 30 and 31 in Section 7), and define

an endomorphism on it that maps app to app′ recursively, and is the identity on the other constructors. This yields a monad morphism, but not a morphism of substitution systems; indeed, the second diagram of Def. 18 does not commute—any endomorphism on that substitution system must be the identity morphism.

▶ Nota bene. The question may arise if we could modify our results on the construction of monads to obtain relative monads [7]. However, non-relativized monads are the more general outcome, since, by composing the constituents of a monad (in the presentation with Kleisli extension instead of a monad multiplication) with a given functor J, we would obtain monads relative to J (called "restriction" [7, Proposition 2.3(1)]).

7 Lifting Initiality Through a Fusion Law

The starting point of this section is a result from [24], which gives one way to define substitution systems and which comes from a very specific instance of Lemma 8. As a first instantiation step, take in that lemma $[\mathcal{C},\mathcal{C}]$ for $\mathcal{C}$ and $\mathcal{D}$ and the *reduction functor* $-\cdot Z$ for L, for any endofunctor Z of $\mathcal{C}$. This is the general situation of the "gfolds" of Bird and Paterson [12], and (the carriers of) the corresponding initial F-algebras are called "nested datatypes" [10]. As Bird and Paterson recall, the assumption of having a right adjoint to the reduction functor means that right Kan extensions along those Z exist. In the context of functional programming with impredicative polymorphism, these right Kan extensions can be defined syntactically: the syntactic right Kan extension of type transformer G along type transformer Z is defined as $\lambda A \forall B.\,(A \to ZB) \to GB$, which is monotone in A for syntactic reasons regardless of G and Z (A occurs non-strictly positively in the body of the abstraction). This construction is essential for relating different formulations of iteration over nested datatypes [3]. However, the full categorical properties of Kan extensions are not ensured by the computation rules of the polymorphic language. Still, they are satisfied in parametric models of higher-order polymorphism [16, Thm. 6.10 (i)]. We will not further develop the categorical semantics of those programming languages. The previous remarks should make it plausible that the following theorem rests on "reasonable" technical conditions. If program verification is aimed at in an intensional setting, replacements for the categorical notions have to be found, and yet different schemes of generalized iteration have to be studied in order to combine expressivity, termination guarantees and program verification in the same framework [23] (using Coq very differently from the `UniMath` approach).

▶ **Theorem 30** ([24], Thm. 15). *Let (H,θ) be a signature. If $[\mathcal{C},\mathcal{C}]$ has an initial $(\underline{\mathsf{Id}}+H)$-algebra and a right adjoint for the functor $-\cdot Z : [\mathcal{C},\mathcal{C}] \to [\mathcal{C},\mathcal{C}]$ exists for every* $\mathsf{Ptd}(\mathcal{C})$*-object (Z,e), then (T,α) defined by*

$$(T,\alpha) = (\mu(\underline{\mathsf{Id}}+H), \mathsf{in}_{\underline{\mathsf{Id}}+H})$$

is a heterogeneous substitution system for (H,θ).

The proof of this theorem is by identifying, for a given $f : (Z,e) \to (T,\eta)$, the morphism $\{f\}$ as an instance of Lemma 8, both for the existence and uniqueness property. The obvious part of the instantiation is the choice of parameters mentioned above, and by setting $F := \underline{\mathsf{Id}} + H$. The essential ingredient for getting a morphism $\{f\}$ of type $\mu F \cdot Z \to T$ (here, T is even μF) is a natural transformation Ψ_f whose typing could sloppily be written as

$$\Psi_f :: \forall X : [\mathcal{C},\mathcal{C}].\,(X \cdot Z \to T) \to (FX \cdot Z \to T) \ .$$

The type of Ψ_f suggests the following problem-solving method: The original problem is that of finding a morphism of type $\mu F \cdot Z \to T$. We abstract away from μF and replace it by an arbitrary endofunctor $X : [\mathcal{C}, \mathcal{C}]$. For this arbitrary X, we have to extend a purported solution for parameter X, hence of type $X \cdot Z \to T$, to a solution for parameter FX, hence of type $FX \cdot Z \to T$. Of course, this has to be done naturally in X, as required in Lemma 8. So, using the construction that extends solutions for parameter X naturally to solutions for parameter FX, the lemma even yields a (unique) solution for the least fixed-point of F as parameter. The continuity properties behind this method were deeply explored by Abel [2] for (co-)inductive types and extended to nested datatypes later [1].

This is the essence of schemes in Mendler's style [25]: being able to advance from a solution in parameter X to a solution in parameter FX *uniformly* (in Mendler's original work, this was plainly universal quantification over a type variable X; in the categorical setting, this is achieved by naturality), one is guaranteed a solution in parameter μF. Lemma 8 is an instance of that idea, hence the name "generalized iteration in Mendler-style".

Mendler-style gives great liberty: were are free in choosing Ψ_f of the required type (implicitly asking for naturality), but there is little guidance in finding the right one for our purpose. Guidance would, e. g., come from asking for an algebra structure on the target endomorphism T. Therefore, we instantiate the lemma further to obtain what is called "a special case of generalized iteration" by Matthes and Uustalu [24].[1] It consists in requiring an endofunctor F' on $[\mathcal{C}, \mathcal{C}]$, a natural transformation $\theta' : (F-) \cdot Z \to F'(- \cdot Z)$ and an F'-algebra $\varphi : F'T \to T$ on T, and in putting them together to obtain

$$\Psi_f(X)(h : X \cdot Z \to T) := \varphi \circ F'h \circ \theta'_X : FX \cdot Z \to T \ .$$

Its use in our present situation is then with $F' := \underline{Z} + H$, $\theta'_X := \mathsf{id} + \theta_{X,(Z,e)}$ and $\varphi := [f, \tau]$, using the strength θ of the signature and the H-algebra τ that is generically derived from α (see before Definition 15).

▶ Nota bene. We remark that all of this is not optimal from a progammer's point of view, where the question is not only of soundness but of efficiency of the traversals through the data structures. There is the more refined notion of "generalized Mendler iteration" [3] (called GMIt^ω) as an efficient way out. The crucial idea is to generalize the problem further than finding a solution of $X \cdot Z \to T$ for parameter $X = \mu F$. An $h : X \cdot Z \to T$ consists of morphisms $h_A : X(ZA) \to TA$ for every $A : \mathcal{C}_0$, and generalized Mendler iteration asks even for operations $h_f : XB \to TA$ for any $B : \mathcal{C}_0$ and $f : B \to ZA$. Taking for f the identity morphism on ZA, one gets the desired components of the solution in the end. The gain in efficiency comes from the combination of a fold and a map in this scheme—enforced just by these types in the polymorphic formulation of [3].

Also for generalized Mendler iteration, there is a formulation in more conventional terms of algebras, called "generalized refined conventional iteration" [3], which captures in particular the efficient folds of Martin, Gibbons and Bayley [22]. For generalized Mendler iteration, there is also a means of verification in usual intensional Coq, using category theory only as a motivation and not as the mathematical framework [23].

We augment the previous theorem by showing that the constructed substitution system is initial:

[1] The instantiation with $- \cdot Z$ for L can also be formulated in a less homogeneous setting where not only endofunctor categories intervene [24, Section 2.3].

▶ **Theorem 31.** *The substitution system* $(T, \alpha, \{\})$ *constructed in Lemma 30 is initial in* $\mathsf{hss}(H, \theta)$.

In order to prove Theorem 31, it suffices to show that, for any given substitution system $(T', \alpha', \{\}')$, the initial morphism **of algebras**

$$! : (T, \alpha) \to (T', \alpha')$$

is compatible with the operations $\{\}$ (defined in the proof of Lemma 30) and $\{\}'$. That is, we need to show that, for any $f : (Z, e) \to (T, \eta)$,

$$! \circ \{f\} = \{! \circ f\}' \circ (! \cdot Z) \ . \tag{3}$$

Using the fusion law (Lemma 10), we show that both sides of (3) are equal to the application of an iterator. More precisely, we use the fusion law for the left-hand side, knowing the explicit definition of $\{f\}$ as an iterator, described above, to establish equality with $\mathsf{It}_F^{- \cdot Z}(\,\Psi_f\,)$, where we define

$$\Psi_f(X)(h : X \cdot Z \to T') := [! \circ f, \tau' \circ Hh \circ \theta_{X,(Z,e)}] : FX \cdot Z \to T' \ .$$

Once the premisses of the fusion law established, we can show equality with the right-hand side of (3) by verifying that the defining equations of $\mathsf{It}_F^{- \cdot Z}(\,\Psi_f\,)$ are fulfilled by the right-hand side.

8 A Worked Example: Flattening of Explicit Substitution

In practice, a signature is often a family of arities, each arity specifying the type of one term constructor. A typical example is a typeful version of de Bruijn indices for pure (untyped) λ-calculus, where, intuitively, the equation

$$TA = A + TA \times TA + T(1 + A)$$

has to be solved, giving in TA the set of λ-terms having free variables among A (cf. the introduction), where the last summand represents λ-abstraction that abstracts the variable corresponding to the extra element of $1 + A$. This example is developed in [24] but originates from [8, 11].

We can "glue" signatures with strength together to obtain a new signature with strength:

▶ **Lemma 32** (Sum of signatures)**.** *Let* (H, θ) *and* (H', θ') *be two signatures. Then* $(H + H', \theta + \theta')$ *is a signature.*

This lemma is important for our main example: indeed, we consider two signatures, where one is obtained from the other by extending the language (better: its signature) by one additional term constructor (better: arity).

To this end, we need the base category $\mathcal{C}$ to come equipped with some extra structure: for the remainder of this section, we assume $\mathcal{C}$ to have (specified) products, coproducts and a terminal object. An example of such a category is the (univalent) category Set of sets (see Section 2), which has all limits and colimits.

We continue the case study in [24] on λ-calculus without and with a form of explicit substitution—"explicit flattening". In order to do so, we first present the signatures (H, θ) corresponding to application, abstraction, and explicit flattening, respectively:

▶ **Definition 33** (Application). The signature of application is given by pointwise product, inherited from the base category $\mathcal{C}$:

$$H^{\mathsf{App}}(T) := T \times T \ .$$

The strength θ^{App} is given pointwise by the identity,

$$\theta^{\mathsf{App}}_{X,(Z,e)} : (X \times X) \cdot Z \to (X \cdot Z) \times (X \cdot Z) \ .$$

The fact that the identity suffices here corresponds to the triviality of first-order operations in substitution (which is plainly homomorphic on those operations).

▶ **Definition 34** (Abstraction). Abstraction in our context is defined by precomposition with a coproduct, corresponding to "context extension":

$$H^{\mathsf{Abs}}(T) := T \cdot \mathsf{option} \ ,$$

where $\mathsf{option}(X) := 1 + X$ represents the context X extended by one distinguished element $\mathsf{inl}_{1,X}(\star)$. The strength θ is defined as

$$\theta^{\mathsf{Abs}}_{X,(Z,e)}(A) := X[\,e_{1+A} \circ \mathsf{inl}_{1,A}, Z\mathsf{inr}_{1,A}\,] : X(1 + ZA) \to X(Z(1 + A)) \ .$$

The defined strength embodies the usual lifting needed for substitution in de Bruijn representations of λ-abstraction.

▶ **Definition 35** (Explicit flattening). The flattening signature is defined by selfcomposition,

$$H^{\mathsf{Flatten}}(T) := T \cdot T \ ,$$

and the corresponding strength requires the unit e of the pointed endofunctor (Z, e) to be inserted in the right place:

$$\theta^{\mathsf{Flatten}}_{X,(Z,e)} := X \cdot e \cdot X \cdot Z : X \cdot X \cdot Z \to X \cdot Z \cdot X \cdot Z \ .$$

Note that the flattening signature cannot be dealt with in a framework with a fixed enumeration of variable names and shows, already on the syntactic side, the most simple case of "true nesting" in nested datatypes (see, e. g., [3]). Notice that the highly parameterized type already suggests the right definition. For its mainly used instance $\theta^{\mathsf{Flatten}}_{T,(T,\eta)}$, with T and η components of the obtained substitution system, its type $T^3 \to T^4$ hardly suggests a canonical definition.

These signatures are now combined, as per Lemma 32, to obtain the signatures we are mainly interested in:

▶ **Definition 36** (Signature of λ-calculus). The signature Λ is obtained as the sum of the signatures of Defs. 33 and 34.

▶ **Definition 37** (Signature of λ-calculus with explicit flattening). The signature Λ^μ is obtained as the sum of the signatures of Defs. 36 and 35.

For the purpose of this example, we assume the signatures Λ and Λ^μ to have initial substitution systems. By Lemma 30 we get those if we assume that their underlying initial algebras exist. (For a remark on the construction of initial algebras, see Section 10.) We denote the initial substitution systems by $(\mathsf{Lam}, \alpha, \{\})$ and $(\mathsf{Lam}^\mu, \alpha^\mu, \{\}^\mu)$, respectively.

Intuitively, they solve the equation in T given in the first paragraph of this section, and the following equation in T', respectively:

$$T'A = A + T'A \times T'A + T'(\mathsf{option}\,A) + T'(T'A) \ .$$

Why is Lam^μ supposed to represent λ-calculus with explicit flattening? Coming back to parallel substitution on T (= Lam), as mentioned in the introduction, we may study the substitution rule $f := \lambda x^{TB}.x$ of type $TB \to TB$. Then, $\mu_B := [f] : T(TB) \to TB$ can be interpreted as doing the following: in a term whose free variables have as names terms over B, those names are replaced by themselves, but now integrated into the given term. In other words, μ_B removes the "cross section" between the trunk of the term and the term-like variable leaves. Invoking Theorem 26 for $(\mathsf{Lam}, \alpha, \{\})$, one obtains $\mu := \{\mathsf{id}_{(\mathsf{Lam},\eta)}\} : \mathsf{Lam}\cdot\mathsf{Lam} \to \mathsf{Lam}$ as monad multiplication on the monad of λ-terms, and the above-mentioned parallel substitution can then be derived generically, so as to obtain its components μ_B with the described behaviour. In other words, the generic notion of monad multiplication appears to have the behaviour of "flattening" a nested term structure of type $T(TB)$ into one of type TB (for every B). Now, Lam^μ even has a term constructor, corresponding to the injection of the last summand of the above equation into the left-hand side. Hence, the constructor is of type $\mathsf{Lam}^\mu \cdot \mathsf{Lam}^\mu \to \mathsf{Lam}^\mu$, which is the same type as monad multiplication. As a constructor, this operation does *not* denote the result of the flattening (here, even for the extended syntax), but is a formal syntactic element and is therefore called an explicit flattening operation. (Cf. explicit substitution; in fact, explicit flattening is a variant of explicit substitution.) Already in [24], it was shown that those explicit flattenings can be resolved by evaluating any term with explicit flattenings (from $\mathsf{Lam}^\mu A$ for some A) into a term without explicit flattenings (in $\mathsf{Lam}A$). We continue this case study by using our extra categorical structure on substitution systems.

In the following, our goal is to construct a morphism of substitution systems from Lam^μ to Lam. This is not quite precise and needs refinement, since a priori, those two substitution systems are not in the same category. More precisely, we are going to build a substitution system for the signature Λ^μ, the underlying carrier of which is the carrier Lam. To this end, we need to construct two ingredients: firstly, we need a natural transformation $\mu^{\mathsf{Lam}} : H^{\mathsf{Flatten}}(\mathsf{Lam}) \to \mathsf{Lam}$ in order to obtain a structure of $\underline{\mathsf{Id}}+\Lambda^\mu$-algebra on Lam. Secondly, we equip this $\underline{\mathsf{Id}}+\Lambda^\mu$-algebra with a substitution operation—which, of course, must be shown compatible with the $\underline{\mathsf{Id}}+\Lambda^\mu$-algebra structure in the sense of the diagram of Definition 15.

Once this is done, we obtain, by initiality, a morphism of substitution systems from the initial substitution system of Λ^μ to the newly constructed one, the underlying algebra morphism of which is a morphism from Lam^μ to Lam that "does the right thing": mapping explicit substitution to substitution.

▶ **Definition 38** (Representation of flattening on Lam)**.** Let $\mu^{\mathsf{Lam}} : H^{\mathsf{Flatten}}(\mathsf{Lam}) \to \mathsf{Lam}$ be given by

$$\mu^{\mathsf{Lam}} := \{\mathsf{id}_{\mathsf{Lam}}\} : \mathsf{Lam}\cdot\mathsf{Lam} \to \mathsf{Lam} \ .$$

▶ **Lemma 39** (Substitution system of Λ^μ on Lam)**.** *The pair* $(\mathsf{Lam}, [\,\alpha, \mu^{\mathsf{Lam}}\,])$ *is an* $\underline{\mathsf{Id}}+\Lambda^\mu$*-algebra. (Here, we have implicitly used associativity of the coproduct.)*

We define substitution $\{\}^{\mathsf{Flatten}}$ *on this algebra by setting, for* (Z,e) *and* $f : (Z,e) \to (\mathsf{Lam},\eta)$*,*

$$\{f\}^{\mathsf{Flatten}} := \{f\} \ .$$

This assignment defines substitution on that algebra, and hence a substitution system $(\mathsf{Lam}, [\,\alpha, \mu^{\mathsf{Lam}}\,], \{\}^{\mathsf{Flatten}})$ *for the signature* Λ^{μ}*.*

Proof. We need to show that $\{-\}^{\mathsf{Flatten}}$ satisfies the equations of substitution, see Definition 15. The diagrams can be checked for any "arity" individually, and for η, App and Abs, the equations to check are exactly those satisfied by Lam as a substitution system for the signature Λ. The only non-trivial equation to check states that $\{-\}^{\mathsf{Flatten}}$ is compatible with μ^{Lam}; we have to check that

$$\{f\}^{\mathsf{Flatten}} \circ \mu^{\mathsf{Lam}} \cdot Z = \mu^{\mathsf{Lam}} \circ \mathsf{Lam}(\{f\}^{\mathsf{Flatten}}) \circ \{f\}^{\mathsf{Flatten}} \cdot \mathsf{Lam} \cdot Z \circ \mathsf{Lam} \cdot e \cdot \mathsf{Lam} \cdot Z$$

We omit the details of this calculation here, and refer instead to the formal proof. ◀

We thus have two objects in the category $\mathsf{hss}(\Lambda^{\mu})$, an initial object with underlying carrier Lam^{μ}, and the object constructed in Lemma 39, with underlying carrier Lam. By initiality, we obtain a unique morphism of substitution systems in this category.

▶ **Definition 40.** We call $\mathsf{eval} : \mathsf{Lam}^{\mu} \to \mathsf{Lam}$ the morphism of substitution systems obtained by initiality. This map sends application and abstraction to themselves, respectively, and it sends the explicit flattening operator to its "evaluation", that is, to a "flattened" term.

This morphism of substitution systems gives rise, via functoriality of the monad construction (Theorem 27), to a monad morphism; it is this morphism that is studied in Example 16 of [24]. Here, we have shown how that monad morphism arises from a morphism of substitution systems.

9 About the Formalization

Most of the results presented in this article have been formalized, based on the `UniMath` library [32]. More precisely, all results except for Theorem 22 and Lemmas 25 and 21 are proved in our formalization.

Our formalization started out as an independent repository, but has since been integrated into `UniMath`, as a package (subdirectory) called `SubstitutionSystems`. The formalization can be inspected by cloning the `UniMath` repository on Github, `https://github.com/UniMath/UniMath`, following the installation procedure described there.

The `UniMath` library being under active development, the organization of the packages is going to change: some code will be moved to other, more fundamental, packages. For the purpose of inspection of the package `SubstitutionSystems` as described here, it is hence convenient to stick with a particular commit of the git repository, e.g., `commit 1ead81a`. The sections of this article roughly correspond to files in the formalization:

`GenMendlerIteration.v` corresponds to Section 4;

`SubstitutionSystems.v` corresponds to Section 5;

`MonadsFromSubstitutionSystems` corresponds to Section 6;

`LiftingInitial.v` corresponds to Section 7.

The code corresponding to Section 8 is spread over several files:

`SumOfSignatures.v` corresponds to Lemma 32;

`LamSignature.v` corresponds to Definitions 33, 34, 35;

`Lam.v` corresponds to the rest of Section 8.

Table 1 Lines of code of the library `SubstitutionSystems`.

spec	proof	comments	
32	59	10	AdjunctionHomTypesWeq.v
90	165	102	Auxiliary.v
28	14	8	EndofunctorsMonoidal.v
70	124	27	FunctorsPointwiseCoproduct.v
70	113	7	FunctorsPointwiseProduct.v
91	116	30	GenMendlerIteration.v
28	21	7	HorizontalComposition.v
79	407	72	LamSignature.v
106	249	57	Lam.v
236	518	61	LiftingInitial.v
123	423	76	MonadsFromSubstitutionSystems.v
26	0	12	Notation.v
15	4	9	PointedFunctorsComposition.v
36	61	11	PointedFunctors.v
42	81	11	ProductPrecategory.v
22	0	10	RightKanExtension.v
82	211	40	Signatures.v
155	326	53	SubstitutionSystems.v
69	170	13	SumOfSignatures.v
1400	3062	616	total

To account for the ongoing work on the `UniMath` library, we provide an "interface" file
`UniMath/SubstitutionSystems/SubstitutionSystems_Summary.v`
containing pointers to the most important formalized theorems. So, while this paper is best studied with an eye on the commit mentioned above, the interface file allows to locate all the notions, constructions and results in their respective current state of evolution.

9.1 Statistics

Our library consists of a bit more than 4400 loc, plus 600 lines of comments[2]. Details are given in Table 1—numbers are taken from `commit 1ead81a`. For comparison, for the same commit, the whole of `UniMath`, including our library, consists of about 37000 lines of code:

spec	proof	comments	
15053	22389	3987	total

9.2 About Performance: Transparency vs. Opacity

One important aspect of computer proof assistants that are based on type theory is **computation**. Computation enables us to obtain some equalities for free. For instance, in our formalization of (co)products in a functor category $[\mathcal{C}, \mathcal{D}]$ from (co)products in the

[2] Note that the organization of the files is going to change over time, due to reorganization of the library. In particular, contents may get moved to other parts of `UniMath` in the future.

target category $\mathcal{D}$, the (co)product of two functors F and G **computes** pointwise to the (co)product of the images, that is, for instance $(F \oplus_{[\mathcal{C},\mathcal{D}]} G)(c) \equiv Fc \oplus_{\mathcal{D}} Gc$. Here, the notation $\equiv$ denotes definitional equality a.k.a. computation. This is only true for a specific construction of (co)products in functor categories, of course; in general, one can only expect $(F \oplus_{[\mathcal{C},\mathcal{D}]} G)(c) \simeq_{\mathcal{D}} Fc \oplus_{\mathcal{D}} Gc$. However, in order to keep the complexity of our proofs manageable for us, having definitional equality instead of isomorphism was crucial. We hence had to keep many category-theoretic constructions, such as (co)products in functor categories, **transparent**. Technically, this amounts to closing a proof using `Defined.` instead of `Qed.` in the Coq proof assistant.

This lack of opacification, however, results in terms getting very large, making type checking more costly for the machine. The transparency vs. opacity issue can hence be restated as an issue of human vs. machine friendliness.

Our approach to this issue was to make opaque all the terms that we could afford making opaque, either by moving them into lemmas by themselves, closing with `Qed.`, or by enclosing the corresponding sequence of tactics producing that term into an `abstract (...)` block. The inconvenience of the latter method is that the block enclosed by `abstract` must be **one** tactic (composed using the semicolon), not a sequence of tactics. This method is hence only feasible for small subproofs.

Our library is quite slow to compile, due to the rather large proof terms arising when working with multiple stacked constructions in category theory: some `Qed.` take very long to check. A significant speedup was obtained in the file `MonadsFromSubstitutionSystems.v` by setting the option `Unset Kernel Term Sharing.`, the workings of which are unknown to us. However, this option proved useless or even increased compile time in other files, and is hence only used in that one file. It is unclear to us why this option is beneficial in that file and only there, and whether there is a guiding principle saying when this option is useful.

In our library, there is a slight duplication of code: the `UniMath` library contains a proof that colimits lift to functor categories from the target category, formalized by Ahrens and Mörtberg [6]. This result could in principle be applied to lift coproducts and products, both of which are formalized as specific colimits. However, it turned out that this approach made typechecking unfeasibly slow: indeed, the first files making use of coproducts in functor categories would stop compiling when that construction of coproducts in functor categories was plugged in. Instead, we provide a manual lifting of (co)products into functor categories in the files `FunctorsPointwiseProduct.v` and `FunctorsPointwiseCoproduct.v`, with which typechecking is reasonably fast. The latter construction applies similar principles of opacification as the general lifting of colimits; it is hence unclear to us why the latter does perform so much better than the former.

Added in print: the use of primitive projections in Coq, via the option `Set Primitive Projections`, reduced the compilation time for our project dramatically (much better than the 56% drop in time that were observed for the whole `UniMath` library on average). This was adopted for the Σ-types used in `UniMath` in `commit 6b044cc`.

10 Conclusions

We presented, in univalent type theory, some new results about the heterogeneous substitution systems introduced by Matthes and Uustalu [24], and showed how to obtain initial substitution systems (such as lambda calculi) from initial algebras using generalized iteration in Mendler-style.

We have not studied, in the present work, the construction of initial algebras in univalent type theory; this is the subject of joint work with Mörtberg [6, 5].

Acknowledgements Thanks to Paige North for discussion of the subject matter, and to Anders Mörtberg for providing feedback to a draft of this article. Thanks to the rest of the UniMath team, for providing a sound base for formalization, and, specifically, to Dan Grayson and Anders Mörtberg for helping maintain the code described in this article.

References

1 A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. Doktorarbeit (PhD thesis), LMU München, 2006.

2 Andreas Abel. Termination checking with types. *ITA*, 38(4):277–319, 2004. doi:10.1051/ita:2004015.

3 Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005. doi:10.1016/j.tcs.2004.10.017.

4 B. Ahrens, K. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015. doi:10.1017/s0960129514000486.

5 B. Ahrens, R. Matthes, and A. Mörtberg. From signatures to monads in UniMath, 2016. arXiv preprint 1612.00693. URL: https://arxiv.org/abs/1612.00693.

6 Benedikt Ahrens and Anders Mörtberg. Some wellfounded trees in UniMath - extended abstract. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew J. Sommese, editors, *Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, volume 9725 of *Lecture Notes in Computer Science*, pages 9–17. Springer, 2016. doi:10.1007/978-3-319-42432-3_2.

7 T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. *Log. Methods Comput. Sci.*, 11(1):article 3, 2015. doi:10.2168/lmcs-11(1:3)2015.

8 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0_32.

9 Françoise Bellegarde and James Hook. Subsitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2-3):287–311, 1994. doi:10.1016/0167-6423(94)00022-0.

10 R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Proc. of 4th Int. Conf. on Mathematics of Program Construction, MPC '98*, volume 1422 of *Lect. Notes in Comput. Sci.*, pages 52–67. Springer, 1998. doi:10.1007/bfb0054285.

11 R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999. doi:10.1017/s0956796899003366.

12 Richard S. Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2):200–222, 1999. doi:10.1007/s001650050047.

13 The Coq Development Team. The Coq proof assistant reference manual, version 8.6, 2016. URL: https://coq.inria.fr/distrib/current/refman/.

14 M. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. of 23rd Ann. IEEE Symp. on Logic in Computer Science, LICS 2008*. IEEE, 2008. doi:10.1109/lics.2008.38.

15 M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Ann. IEEE Symp. on Logic in Computer Science, LICS '99*, pages 193–202. IEEE, 1999. doi:10.1109/lics.1999.782615.

16 Ryu Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September*

24-27, 1991, Proceedings, volume 526 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 1991. doi:10.1007/3-540-54415-1_61.

17 A. Hirschowitz and M. Maggesi. Initial semantics for strengthened signatures. In D. Miller and Z. Ésik, editors, *Proc. of 8th Wksh. on Fixed Points in Computer Science, FICS 2012*, volume 77 of *Electron. Proc. in Theor. Comput. Sci.*, pages 31–38. Open Publishing Assoc., 2012. doi:10.4204/eptcs.77.5.

18 André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 14th International Workshop, WoLLIC 2007, Rio de Janeiro, Brazil, July 2-5, 2007, Proceedings*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007. doi:10.1007/978-3-540-73445-1_16.

19 André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010. doi:10.1016/j.ic.2009.07.003.

20 M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In G. Sambin and J. M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Clarendon Press, 1998.

21 G. Huet and A. Saïbi. Constructive category theory: Essays in honour of Robin Milner. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, Foundations of Computing Series, pages 239–275. MIT Press, 2000.

22 Clare E. Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Asp. Comput.*, 16(1):19–35, 2004. doi:10.1007/s00165-003-0013-6.

23 Ralph Matthes. Map fusion for nested datatypes in intensional type theory. *Sci. Comput. Program.*, 76(3):204–224, 2011. doi:10.1016/j.scico.2010.05.008.

24 Ralph Matthes and Tarmo Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theor. Comput. Sci.*, 327(1-2):155–174, 2004. doi:10.1016/j.tcs.2004.07.025.

25 N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Log.*, 51(1–2):159–172, 1991. doi:10.1016/0168-0072(91)90069-x.

26 Marino Miculan and Ivan Scagnetto. A framework for typed HOAS and semantics. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 184–194. ACM, 2003. doi:10.1145/888251.888269.

27 E. Palmgren and O. Wilander. Constructing categories and setoids of setoids in type theory. *Log. Methods Comput. Sci.*, 10(3):article 25, 2014. doi:10.2168/lmcs-10(3:25)2014.

28 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014. doi:10.1007/978-3-319-08970-6_32.

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

30 V. Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Math. Struct. Comput. Sci.*, 25:1278–1294, 2015. doi:10.1017/s0960129514000577.

31 V. Voevodsky. C-system of a module over a Jf-relative monad, 2016. arXiv preprint 1602.00352. URL: https://arxiv.org/abs/1602.00352.

32 V. Voevodsky, B. Ahrens, D. Grayson, and Others. UniMath: Univalent mathematics. URL: https://github.com/UniMath.

Towards a Cubical Type Theory without an Interval*

Thorsten Altenkirch[1] and Ambrus Kaposi[2]

1 **School of Computer Science, University of Nottingham, Wollaton Road, Nottingham NG8 1BB, United Kingdom**
txa@cs.nott.ac.uk

2 **Faculty of Informatics, Eötvös Loránd University, Pázmány P. stny. 1/C, 1117 Budapest, Hungary**
akaposi@inf.elte.hu

Abstract

Following the cubical set model of type theory which validates the univalence axiom, cubical type theories have been developed that interpret the identity type using an interval pretype. These theories start from a geometric view of equality. A proof of equality is encoded as a term in a context extended by the interval pretype. Our goal is to develop a cubical theory where the identity type is defined recursively over the type structure, and the geometry arises from these definitions. In this theory, cubes are present explicitly, e.g., a line is a telescope with 3 elements: two endpoints and the connecting equality. This is in line with Bernardy and Moulin's earlier work on internal parametricity. In this paper we present a naive syntax for internal parametricity and by replacing the parametric interpretation of the universe, we extend it to univalence. However, we do not know how to compute in this theory. As a second step, we present a version of the theory for parametricity with named dimensions which has an operational semantics. Extending this syntax to univalence is left as further work.

1998 ACM Subject Classification F.4.1 Mathematical Logic and Formal Languages: Mathematical Logic

Keywords and phrases homotopy type theory, parametricity, univalence

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.3

1 Introduction

Homotopy Type Theory [18] introduces the *univalence axiom*, which basically identifies propositional equality with equivalence of types[1]. We can understand the *univalence axiom* as a powerful extensionality principle which allows us to replace one type by another which has the same behaviour. It also entails functional extensionality.

The usual formulation of Homotopy Type Theory (given in the appendix of [18]) lacks a computational understanding of univalence. With some simplification it adds the following axiom to Martin-Löf Type Theory:

$$\mathsf{univ} : A \simeq B \to A =_{\mathsf{U}} B$$

* This research was supported by EPSRC grant EP/M016951/1, USAF grant FA9550-16-1-0029 and COST Action EUTypes CA15123.

[1] Equivalence of types is a proof-relevant refinement of isomorphism. Naively it is sufficient to think of isomorphism since this notion is logically equivalent (but not isomorphic as a type).

© Thorsten Altenkirch and Ambrus Kaposi;
licensed under Creative Commons License CC-BY

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 3; pp. 3:1–3:27

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

saying that an equality in the universe is provided given an equivalence between the types A and B. Here, equality is defined as an inductive family with the constructor refl and eliminator J. The axiom univ adds another constructor without providing an elimination rule for this constructor resulting in stuck terms. For example, we can define the equivalence $(\mathsf{not}, \dots)$ between Bool and Bool and thus $\mathsf{univ}\,(\mathsf{not}, \dots) : \mathsf{Bool} =_{\mathsf{U}} \mathsf{Bool}$, and then using it, define a boolean b:

$$\mathsf{coe} : (A =_{\mathsf{U}} B) \to A \to B$$
$$\mathsf{coe}\,(\mathsf{refl}\,A)\,a :\equiv a$$
$$b : \mathsf{Bool}$$
$$b :\equiv \mathsf{coe}\,(\mathsf{univ}\,(\mathsf{not}, \dots))\,\mathsf{true}$$

We know that b should be false as we coerce along not, but coe was only defined[2] for the case refl so the term b does not compute further. One may think that it would be enought to add the equation $\mathsf{coe}\,(\mathsf{univ}\,(f, \dots)) \equiv f$ but this is not sufficient as shown by the following example. Here we use the ap (apply path) function for a $P : \mathsf{U} \to \mathsf{U}$:

$$\mathsf{ap}_P : A =_{\mathsf{U}} B \to P\,A =_{\mathsf{U}} P\,B$$
$$\mathsf{ap}_P\,(\mathsf{refl}\,A) \ \equiv \mathsf{refl}\,(P\,A)$$
$$x : P\,\mathsf{Bool} =_{\mathsf{U}} P\,\mathsf{Bool}$$
$$x :\equiv \mathsf{ap}_P\,(\mathsf{univ}\,(\mathsf{not}, \dots))$$

To reduce this term we need to know how to transport an equivalence along an arbitrary type.

We observe that the problem is caused by viewing equality as an inductive type with only one constructor. Alternatively, we may want to exploit the characterisation of equalities for each type (as described in chapter 2 of [18]): equality of pairs is a pair of equalities, equality of functions is given by a function and equality of types is given by equivalence. That is we want to define the equality relation inductively on the type structure. The theory of logical relations, as known from parametricity ([10, 13, 5]) describes such relations: a logical relation for pairs is a pair of relations, a logical relation for functions is a function which maps related inputs to related outputes. When we view equality as a logical relation, the fundamental lemma of logical relations which says that every term preserves the relation becomes congruence for equality (usually denoted ap or cong). Our work is closely related to the earlier work by Bernardy and Moulin on internal parametricity [6, 15, 8]. Their later [7, 16] work uses as interval type just as [11]. The work on Observational Type Theory [3] is related, however it uses a different heterogeneous equality with built-in proof-irrelevance. Another work very similar to ours is [17], however it only targets functional extensionality.

Our approach is an alternative to introducing an interval pretype as in [11, 4]. Clearly, their work is more complete in that it provides a computational understanding of univalence. While the introduction of the interval is very elegant it is interesting to consider an alternative approach where equality is defined recursively. However, in the moment it is not clear how to interpret univalence in this context. The basic idea is to say that equality in the universe is given by a symmetric notion of equivalence but the problem is that in this definition we refer back to equality for any type in the universe. Hence we do not succeed in justifying the

[2] coe was defined by pattern matching, which can be seen as a usage of the eliminator J in this case.

usual rules for equality, in particular the eliminator J. However, justifying univalence is also technically complicated in [11].

Our approach is also closely related to the basic cubical model of type theory as described in [9, 14] (without connections). That is the structure of the presheaf model emerges in our syntax. Hence, we conjecture that there is a natural interpretation of our syntax in this presheaf category. The main question is how a univalent universe of fibrant types can be defined in the syntax.

1.1 Structure of the Paper

We develop a naive approach to cubical type theory by defining equality in section 2. Our main tool is to introduce heterogenous logical relations similar to Bernardy and Moulin [6]. However, they focus on logical predicates while we are using logical relations and our approach is based on a calculus with explicit subsitutions. We also show how to interpret univalence and derive the eliminator for equality in this calculus (section 2.8). One issue with this calculus as already observed by Bernardy and Moulin is the computational interpretation of the swap operation which swaps dimensions. To address this we introduce a new system with named dimensions in section 3. This is similar to Type Theory in Colour [8] but with explicit substitutions and without the need to annotate all judgements with a list of colours or dimensions. We present an operational semantics for this calculus (appendix A) but do not prove completeness. Also the question how to interpret univalence is left open. We conclude in section 4.

2 Naive Cubical Type Theory

In this section we introduce a naive syntax for cubical type theory. After presenting a core substitution calculus (section 2.1) and the rules for dependent function space (section 2.2) we show how to extend the theory by rules for external parametricity (section 2.3), internal parametricity (section 2.4) and finally we replace relations by equality relations and admit univalence (2.8). In each section, we introduce new derivation rules which extend the theory given in the previous sections. We discuss the metatheoretic properties of the theory in section 2.9.

2.1 Core Substitution Calculus

We start with an explicit substitution calculus with variable names and universes à la Russell. Weakening is implicit. While we present the system using named variables for readability we assume that terms are identified up to α-conversion and that name capture is always avoided by appropriate renaming. We use $\mathsf{U} : \mathsf{U}$ for presentation purposes, but mean $\mathsf{U}_i : \mathsf{U}_j$ only if $i < j$ officially. We only define well-typed terms (no preterms) and we write equality judgements without context and type information for brevity. For a formal account on such an approach, see [2].

Notations:

Γ, Δ, Θ	contexts
x, y, z, X	variables
t, u, v, f	terms
A, B, C	types
ρ, σ, ν	substitutions

Judgment kinds:

$\Gamma \vdash$	Γ is a valid context
$\Gamma \vdash t : A$	t is a term of type A in context Γ
$\rho : \Delta \Rightarrow \Gamma$	a substitution from Δ to Γ
$\Gamma \equiv \Gamma'$	definitional equality of contexts
$\Gamma \vdash t \equiv t' : A$	definitional equality of terms
$\Gamma \vdash \rho \equiv \rho' : \Delta \Rightarrow \Gamma$	definitional equality of substitutions

Rules:

$$\frac{}{\cdot \vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash A : \mathsf{U}}{\Gamma.x : A \vdash}$$

$$\frac{\Gamma \vdash}{\epsilon : \Gamma \Rightarrow \cdot} \qquad \frac{\Gamma \vdash A : \mathsf{U} \quad \rho : \Delta \Rightarrow \Gamma \quad \Delta \vdash t : A[\rho]}{(\rho, x \mapsto t) : \Delta \Rightarrow \Gamma.x : A} \qquad \frac{\Gamma \vdash}{\mathsf{id}_\Gamma : \Gamma \Rightarrow \Gamma}$$

$$\frac{\rho : \Delta \Rightarrow \Gamma \quad \sigma : \Theta \Rightarrow \Delta}{\rho\sigma : \Theta \Rightarrow \Gamma} \qquad \frac{\Delta \vdash A : \mathsf{U} \quad \rho : \Delta \Rightarrow \Gamma}{\rho : \Delta.x : A \Rightarrow \Gamma}$$

$$\mathsf{id}\rho \equiv \rho \equiv \rho\mathsf{id} \qquad \nu(\rho\sigma) \equiv (\nu\rho)\sigma \qquad (\rho, x \mapsto t)\sigma \equiv (\rho\sigma, x \mapsto t[\sigma])$$

$$(\underbrace{\mathsf{id}_\Gamma}_{:\Gamma.x:A \Rightarrow \Gamma}, x \mapsto x) \equiv \mathsf{id}_{\Gamma.x:A} \qquad \frac{\sigma : \Gamma \Rightarrow \cdot}{\sigma \equiv \epsilon}$$

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma \vdash t : B}{\Gamma.x : A \vdash t : B} \qquad \frac{\Gamma \vdash t : A \quad \rho : \Delta \Rightarrow \Gamma}{\Delta \vdash t[\rho] : A[\rho]} \qquad t[\mathsf{id}] \equiv t \qquad t[\rho][\sigma] \equiv t[\rho\sigma]$$

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma.x : A \vdash x : A} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U} : \mathsf{U}} \qquad \mathsf{U}[\rho] \equiv \mathsf{U}$$

We assume all coercion rules, congruence rules and reflexivity, symmetry, transitivity of $\equiv$. We usually omit the starting $\cdot$ from contexts and ϵ from substitutions.

2.2 Function Space

We list the rules for dependent function space here.

$$\frac{\Gamma.x : A \vdash B : \mathsf{U}}{\Gamma \vdash \Pi(x : A).B : \mathsf{U}} \qquad \frac{\Gamma.x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi(x : A).B} \qquad \frac{\Gamma \vdash f : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash f\, u : B[(\mathsf{id}, x \mapsto u)]}$$

$$(\lambda x.t)\, u \equiv t[x \mapsto u] \qquad f \equiv (\lambda x.f\, x)$$

$$(\Pi(x : A).B)[\rho] \equiv \Pi(x : A[\rho]).B[(\rho, x \mapsto x)]$$

$$(\lambda x.t)[\rho] \equiv \lambda x.t[(\rho, x \mapsto x)] \qquad (f\, u)[\rho] \equiv f[\rho]\, u[\rho]$$

$A \to B$ is an abbreviation for $\Pi(x : A).B$ where B does not depend on x. $\Pi(x : A, y : B).C$ is an abbreviation for $\Pi(x : A).\Pi(y : B).C$.

2.3 External Parametricity

Binary parametricity says that logically related interpretations of a term are logically related. We can express this syntactically in the following way: given a term $\Gamma \vdash t : A$, and two substitutions into Γ denoted ρ_0 and ρ_1 which are pointwise related at each type in Γ, then $t[\rho_0]$ and $t[\rho_1]$ will be related at A. We formalise this idea by adding the following rules to our theory:

$$\frac{\Gamma \vdash}{\Gamma^= \vdash} \qquad \mathsf{0}_\Gamma, \mathsf{1}_\Gamma : \Gamma^= \Rightarrow \Gamma \qquad \frac{\Gamma \vdash t : A}{\Gamma^= \vdash t^* : A^*\, t[\mathsf{0}_\Gamma]\, t[\mathsf{1}_\Gamma]} \qquad \frac{\rho : \Gamma \Rightarrow \Delta}{\rho^= : \Gamma^= \Rightarrow \Delta^=}$$

$\Gamma^=$ contains two copies of Γ (ρ_0 and ρ_1 in the above example) and a logical relation between them. A^* is the logical relation at A. It relates the two interpretations of the term t, in the two different copies of Γ, which are projected out by the substitutions $\mathsf{0}_\Gamma$, $\mathsf{1}_\Gamma$. t^* is the witness of this relation. The parametricity rule for a term t says that the two versions of the term are related at A^* and this is witnessed by t^*. Also, substitutions can be lifted to the $-^=$-d contexts. We use the notation $-^=$ because (from section 2.8) we will view the relation A^* as the heterogeneous equality relation on A.

The following rule for relations is admissible as it follows from the rule for terms and the equality rule for U^* below.

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma^= \vdash A^* : A[\mathsf{0}_\Gamma] \to A[\mathsf{1}_\Gamma] \to \mathsf{U}}$$

The operation $-^=$ duplicates a context and adds witnesses that the two new subcontexts are pointwise related. We add lower indices to the variable names to get new variable names.

$$\begin{aligned} \cdot^= &\equiv \cdot \\ (\Gamma.x : A)^= &\equiv \Gamma^=.x_0 : A[\mathsf{0}_\Gamma].x_1 : A[\mathsf{1}_\Gamma].x_2 : A^*\, x_0\, x_1 \end{aligned}$$

The substitutions $\mathsf{0}$ and $\mathsf{1}$ project out the corresponding components ($b = \mathsf{0}, \mathsf{1}$).

$$\begin{aligned} b. &\equiv \epsilon && : \cdot \Rightarrow \cdot \\ b_{(\Gamma.x:A)} &\equiv (b_\Gamma, x \mapsto x_b) && : (\Gamma.x : A)^= \Rightarrow \Gamma.x : A \end{aligned}$$

The relation A^* is generated by induction on A: for function types, it says that related inputs are mapped to related outputs, for the universe, it is relation space[3]. As types are just terms, we define t^* for every term t uniformly, including the cases when it is a type:

$$\begin{aligned} (\Pi(x : A).B)^* &\equiv \lambda f_0\, f_1.\Pi(x_0 : A[\mathsf{0}], x_1 : A[\mathsf{1}], x_2 : A^*\, x_0\, x_1).B^*\, (f_0\, x_0)\, (f_1\, x_1) \\ \mathsf{U}^* &\equiv \lambda X_0\, X_1.X_0 \to X_1 \to \mathsf{U} \\ x^* &\equiv x_2 \\ (t[\rho])^* &\equiv t^*[\rho^=] \\ (\lambda x.t)^* &\equiv \lambda x_0\, x_1\, x_2.t^* \\ (f\, u)^* &\equiv f^*\, u[\mathsf{0}]\, u[\mathsf{1}]\, u^* \end{aligned}$$

Note that U^* validates the rule $\mathsf{U}^* : \mathsf{U}^*\, \mathsf{U}\, \mathsf{U}$ by reducing its type. The cases for variables, substituted terms, abstraction and application can be seen as the proof of the fundamental theorem of the logical relation.

---

[3] In section 2.8 we will replace relation space by equivalence

$-^{=}$ on substitutions is defined componentwise.

$$\begin{aligned}
\epsilon^{=} &\equiv \epsilon \\
\mathsf{id}^{=} &\equiv \mathsf{id} \\
(\sigma\rho)^{=} &\equiv \sigma^{=}\rho^{=} \\
(\rho, x \mapsto t)^{=} &\equiv (\rho^{=}, x_0 \mapsto t[\mathsf{0}], x_1 \mapsto t[\mathsf{1}], x_2 \mapsto t^{*})
\end{aligned}$$

Finally, we add naturality of b for $b = \mathsf{0}, \mathsf{1}$.

$$\frac{\rho : \Gamma \Rightarrow \Delta}{\rho b_{\Gamma} \equiv b_{\Delta} \rho^{=}}$$

The above rules add external parametricity to the theory. For example, using the new rules, we can derive that a particular inhabitant of the type $\Pi(A : \mathsf{U}).A \to A$ is the identity function. Using the parametricity rule for the term f and expanding its type using the computation rules we get

$$\frac{f : \Pi(A : \mathsf{U}).A \to A}{f^{*} : \Pi\left(\begin{array}{l} A_0 : \mathsf{U}, A_1 : \mathsf{U}, A_2 : A_0 \to A_1 \to \mathsf{U}, \\ x_0 : A_0, x_1 : A_1, x_2 : A_2\, x_0\, x_1 \end{array}\right).A_2\,(f\,A_0\,x_0)\,(f\,A_1\,x_1)} \; .$$

Now we define a function that for any type A and element $y : A$ shows that $f\,A\,y$ is equal to y. For this example we need an identity type Id_A and its constructor refl to be part of our theory. The binary relation on A says that the first component is equal to y. Obviously, this y witnesses this relation by $\mathsf{refl}\,y$.

$$\lambda A : \mathsf{U}\, y : A.f^{*}\,A\,A\,(\lambda x_0\,x_1.\mathsf{Id}_A\,x_0\,y)\,y\,y\,(\mathsf{refl}\,y) : \Pi(A : \mathsf{U}, y : A).\mathsf{Id}_A\,(f\,A\,y)\,y$$

Another example of using parametricity is given for the following term.

$$A : \mathsf{U}.z : A.s : A \to A \vdash t : A$$

We can interpret the context by the following two substitutions. For this example, we need natural numbers and booleans in our type theory.

$$\begin{aligned}
\rho_0 &= (A := \mathbb{N}, \quad z := \mathsf{zero}, s := \mathsf{suc}) \\
\rho_1 &= (A := \mathsf{Bool}, z := \mathsf{true}, s := \mathsf{not})
\end{aligned}$$

That is, in the first case we use natural numbers with the Peano constructors for z and s and in the second case we have booleans with true for z and negation for successor. We would like to prove that whatever t is, it is not possible to have $t[\rho_0] = \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})$ and $t[\rho_1] = \mathsf{false}$. Binary parametricity says in this case that if we have a relation between $\mathbb{N}$ and Bool such that if $z[\rho_0]$ is related to $z[\rho_1]$ and $s[\rho_0]$ is related to $s[\rho_1]$ then $t[\rho_0]$ is related to $t[\rho_1]$. We can define the relation $A_2 : \mathbb{N} \to \mathsf{Bool} \to \mathsf{U}$ to be $A_2\,x\,b := \mathsf{if}\,b\,\mathsf{then}\,\mathsf{Even}\,x\,\mathsf{else}\,\mathsf{Odd}\,x$. We can show that zero and true are related as zero is even and that the successor of an even number is odd and the successor of an odd number is even. Parametricity tells us that $t[\rho_0]$ and $t[\rho_1]$ need to be related by A_2, hence it cannot happen that $t[\rho_0]$ is 2 and $t[\rho_1]$ is false. Collecting together ρ_0, ρ_1 and the proof of their relatedness into ρ, we can express this in our theory as follows.

$$\cfrac{\cfrac{A : \mathsf{U}.z : A.s : A \to A \vdash t : A}{(A : \mathsf{U}.z : A.s : A \to A)^{=} \vdash t^{*} : A_2\,(t[\mathsf{0}])\,(t[\mathsf{1}])}}{\cdot \vdash t^{*}[\rho] : A_2[\rho]\,(t[\rho_0])\,(t[\rho_1])}$$

We call this theory *external* because even though we can show for every instance of $f : \Pi(A : \mathsf{U}).A \to A$ that it behaves like the identity but we cannot show this *internally* for an assumption of the form $f : \Pi(A : \mathsf{U}).A \to A$.

2.4 Internal Parametricity

To internally derive that every function of type $\Pi(A : \mathsf{U}).A \to A$ is the identity, we would need a term t expressing parametricity for f assuming it in the context containing f:

$$f : \Pi(A : \mathsf{U}).A \to A \vdash t : \Pi(A_0, A_1 : \mathsf{U}, A_2 : A_0 \to A_1 \to \mathsf{U})\,.\\ \Pi(x_0 : A_0, x_1 : A_1, x_2 : A_2\,x_0\,x_1)\,.\,A_2\,(f\,A_0\,x_0)\,(f\,A_1\,x_1).$$

We can try to choose t to be f^*, but that lives in the context $(f : \Pi(A : \mathsf{U}).A \to A)^=$.

This motivates the definition of a substitution that goes from a context Γ into $\Gamma^=$. Using such a substitution R_Γ, we can choose the above t to be $f^*[\mathsf{R}_{f:\Pi(A:\mathsf{U}).A\to A}]$.

We add the substitution R by the following rule.

$$\frac{\Gamma \vdash}{\mathsf{R}_\Gamma : \Gamma \Rightarrow \Gamma^=}$$

We also add the following computation rules.

$$\mathsf{R}_\cdot \equiv \epsilon \qquad \mathsf{R}_{\Gamma.x:A} \equiv (\mathsf{R}_\Gamma, x_0 \mapsto x, x_1 \mapsto x, x_2 \mapsto x^*[\mathsf{R}_{\Gamma.x:A}]) \qquad \frac{\rho : \Delta \Rightarrow \Gamma}{\mathsf{R}_\Gamma \rho \equiv \rho^= \mathsf{R}_\Delta}$$

The first two rules explain how R is duplicating the elements in the context, while the last one is a naturality rule making it possible to commute substitutions with R.

Note that $x^*[\mathsf{R}_{\Gamma.x:A}]$ is equal to $x_2[\mathsf{R}_{\Gamma.x:A}]$ however we do not have any rules to compute such an expression any further. Hence, this becomes a new normal form. We can use the naturality rule to substitute into such a normal form.

2.5 Geometry Arising from the Syntax

A context $(x : A)$ can be viewed as a context of points of type A. The context $(x : A)^=$ is a context of lines where x_2 is a line between the edges x_0 and x_1 (3 elements), A^* is a relation (the type of lines). The $(x : A)^{==}$ is a type of squares with 9 components: 4 points, 4 lines and a filler of. A^{**} is a two-dimensional relation defining the type of squares: it has 8 arguments, the 4 points and the 4 lines of the square.

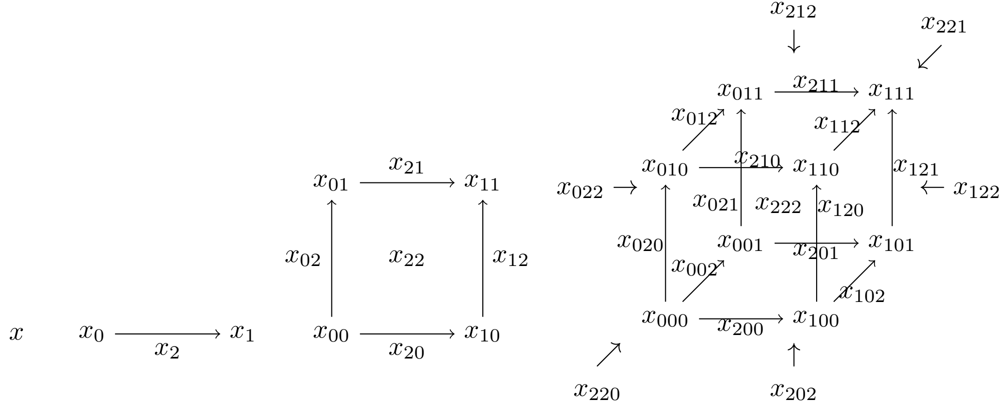
When iterating $–^=$ for a context with more than one type, we get a series of cubes. The later cubes can depend on the previous ones, hence the lines are heterogeneous (the endpoints do not necessarily have the same types). As an example, we expand the first two iterations.

$$\begin{aligned}
(\Gamma.x : A)^= &\equiv \Gamma^=.x_0 : A[\mathsf{0}_\Gamma].x_1 : A[\mathsf{1}_\Gamma].x_2 : A^*\,x_0\,x_1\\
(\Gamma.x : A)^{==} &\equiv \Gamma^{==}.x_{00} : A[\mathsf{0}_\Gamma\mathsf{0}_{\Gamma^=}] \quad .x_{01} : A[\mathsf{0}_\Gamma\mathsf{1}_{\Gamma^=}] \quad .x_{02} : A[\mathsf{0}_\Gamma]^*\,x_{00}\,x_{01}\\
&\qquad .x_{10} : A[\mathsf{1}_\Gamma\mathsf{0}_{\Gamma^=}] \quad .x_{11} : A[\mathsf{1}_\Gamma\mathsf{1}_{\Gamma^=}] \quad .x_{12} : A[\mathsf{1}_\Gamma]^*\,x_{10}\,x_{11}\\
&\qquad .x_{20} : A^*[\mathsf{0}_{\Gamma^=}]\,x_{00}\,x_{10}.x_{21} : A^*[\mathsf{1}_{\Gamma^=}]\,x_{01}\,x_{11}\,.x_{22} : (A^*\,x_0\,x_1)^*\,x_{20}\,x_{21}
\end{aligned}$$

Note that e.g. the type of x_{20} was computed from $(A^*\,x_0\,x_1)[\mathsf{0}_{\Gamma^=.x_0:A[\mathsf{0}].x_1:A[\mathsf{1}]}] \equiv A^*[\mathsf{0}_{\Gamma^=}]\,x_{00}\,x_{10}$. Also, the type of x_{22} is equal to $A^{**}\,x_{00}\,x_{01}\,x_{02}\,x_{10}\,x_{11}\,x_{12}\,x_{20}\,x_{21}$ by the computation rule of $–^*$ for applications.

More generally, the context $(x : A)^n$ ($–^=$ iterated n times) has 3^n components which make an n-dimensional cube. We can depict the first three iterations as follows. The first

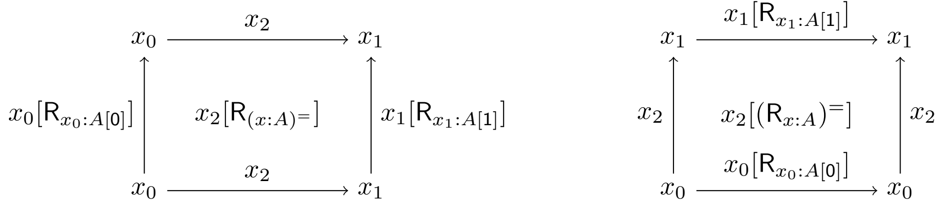
dimension is drawn horizontal, the second vertical and the third is perpendicular to the plain of the paper.



As an example we describe how to view a variable in a 3-dimensional context. If we name the dimensions d_0, d_1, d_2, the variable x_{ijk} represents a face of the 3-dimensional cube which has coordinate i in dimension d_0, coordinate j in dimension d_1, and coordinate k in dimension d_2. By face we mean any 0,1,2 or 3-dimensional face, a 3-dimensional cube only has one 3-dimensional face, the filler. A coordinate can be 0 or 1, in this case we can interpret it as a usual cartesian coordinate. If the coordinate is 2, it means that this is a face spanning through that dimension. Eg. x_{011} is the $d_0 = 0, d_1 = 1, d_2 = 1$ point of a 3-dimensional cube (left upper back if the dimensions are oriented in the standard way); x_{021} is a line in the vertical dimension d_1, it connects the points x_{001} and x_{011} on the left and in the back of the cube; x_{221} is the back face of the cube; x_{222} is the filler of the cube.

For a two-dimensional cube $(x : A)^{==}$, we have four ways to project out the lines: $\mathsf{0}_{(x:A)^=}, (\mathsf{0}_{x:A})^=, \mathsf{1}_{(x:A)^=}$ and $(\mathsf{1}_{x:A})^=$. If we expand $\mathsf{0}_{(x:A)^=}$, we get $x_0 \mapsto x_{00}, x_1 \mapsto x_{10}, x_2 \mapsto x_{20}$, the bottom line of the square. However expanding $(\mathsf{0}_{x:A})^=$ gives $x_0 \mapsto x_{00}, x_1 \mapsto x_{01}, x_2 \mapsto x_{02}$, the left line. In general, $(b_{\Gamma^i})^{n-i}$ projects dimension i to b while decreasing the dimension, i.e. it consists of $x_{j_0 \dots j_{n-1}} \mapsto x_{j_0 \dots j_{i-1} b j_i \dots j_{n-1}}$ maps.

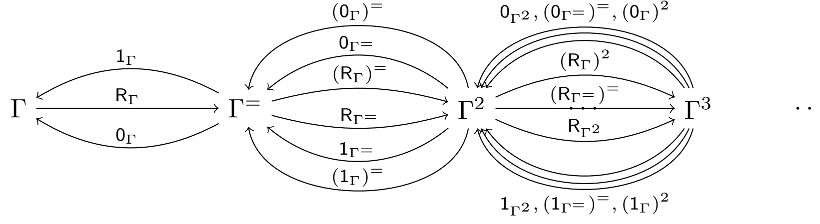
Similarly, from a one-dimensional cube $(x : A)^=$ there are two ways to go to the two-dimensional $(x : A)^{==}$: $\mathsf{R}_{(x:A)^=}$ and $(\mathsf{R}_{x:A})^=$, depicted as the left and right square, respectively.



In general, $(\mathsf{R}_{\Gamma^i})^{n-i}$ adds a degenerate dimension at index i, i.e. $x_{j_0 \dots j_{i-1} b j_i \dots j_{n-1}} \mapsto x_{j_0 \dots j_{n-1}}$ for $b = 0, 1$ and $x_{j_0 \dots j_{i-1} 2 j_i \dots j_{n-1}} \mapsto x_{j_0 \dots j_{n-1}}[(\mathsf{R}_{\Gamma^i})^{n-i}]$. This can be shown as follows: $(\mathsf{R}_{(x:A)^i})^{n-i}$ for arbitrary indices $j_0 \dots j_{i-1}$ includes $(x_{j_0 \dots j_{i-1} 0} \mapsto x_{j_0 \dots j_{i-1}}, x_{j_0 \dots j_{i-1} 1} \mapsto x_{j_0 \dots j_{i-1}}, x_{j_0 \dots j_{i-1} 2} \mapsto x_{j_0 \dots j_{i-1}}[R_{(x:A)^i}])^{n-i}$. We need to apply $-^=$ $(n-i)$ times to the part in parentheses. Applying $-^=$ once means substituting each component with $\mathsf{0}$ and $\mathsf{1}$ and applying $-^*$ while adding indices $0, 1$ and 2, respectively. For the first two components in the substitution we simply get $x_{j_0 \dots j_{i-1} bk} \mapsto x_{j_0 \dots j_{i-1} k}$ this way for $k = 0, 1, 2$. If the third component is substituted by b, we can use the naturality rule for b: $x_{j_0 \dots j_{i-1}}[\mathsf{R}_{(x:A)^i} b_{(x:A)^i}] \equiv x_{j_0 \dots j_{i-1}}[b_{(x:A)^{i+1}} (\mathsf{R}_{(x:A)^i})^=] \equiv x_{j_0 \dots j_{i-1} b}[(\mathsf{R}_{(x:A)^i})^=]$. And using the rule $(t[\rho])^* \equiv t^*[\rho^=]$ we

get $(x_{j_0\ldots j_{i-1}}[\mathsf{R}_{(x:A)^i} b_{(x:A)^i}])^* \equiv x_{j_0\ldots j_{i-1}2}[(\mathsf{R}_{(x:A)^i})^=]$. By induction on $n-i$, we obtain the above rule.

The substitutions coming from iterating $-^=$ on 0_Γ, 1_Γ and R_Γ can be depicted as follows.



2.6 A Swapping Substitution

We add another substitution to explain the relationship between $\mathsf{R}_{\Gamma^=}$ and $(\mathsf{R}_\Gamma)^=$. This is a two-dimensional substitution which swaps the two dimensions of the square. We specify it with the following rule: it does not change the dimension of the context.

$$\frac{\Gamma \vdash}{\mathsf{S}_\Gamma : \Gamma^{==} \Rightarrow \Gamma^{==}}$$

The first two computation rules explain how it acts on the empty and extended contexts. For the extended context, it swaps the dimensions i.e. maps x_{ij} to x_{ji} while the swapped x_{22} needs to be substituted with S because it has a different type now: the arguments of A^{**} have been swapped.

$$\begin{aligned}
\mathsf{S}_\cdot &\equiv \epsilon && : \cdot \Rightarrow \cdot \\
\mathsf{S}_{\Gamma.x:A} &\equiv (\mathsf{S}_\Gamma, x_{00} \mapsto x_{00}, x_{01} \mapsto x_{10}, x_{02} \mapsto x_{20}, \\
&\qquad x_{10} \mapsto x_{01}, x_{11} \mapsto x_{11}, x_{10} \mapsto x_{01}, \\
&\qquad x_{20} \mapsto x_{02}, x_{21} \mapsto x_{12}, x_{22} \mapsto x_{22}[\mathsf{S}_{\Gamma.x:A}]) && : (\Gamma.x:A)^2 \Rightarrow (\Gamma.x:A)^2
\end{aligned}$$

The following three computation rules express that S is an isomorphism (applied twice it gives the identity), it has a naturality rule and it can be used to express $(\mathsf{R}_\Gamma)^=$: we swap the results of $\mathsf{R}_{\Gamma^=}$.

$$\mathsf{S}_\Gamma \mathsf{S}_\Gamma \equiv \mathsf{id}_{\Gamma^{==}} \qquad \frac{\rho : \Gamma \Rightarrow \Delta}{\mathsf{S}_\Delta \rho^{==} \equiv \rho^{==} \mathsf{S}_\Gamma} \qquad \mathsf{S}_\Gamma \mathsf{R}_{\Gamma^=} \equiv (\mathsf{R}_\Gamma)^=$$

We can depict the effect of $\mathsf{S}_{x:A}$ as follows.

$$\begin{array}{ccccccccc}
x_{01} & \xrightarrow{x_{21}} & x_{11} & & & x_{10} & \xrightarrow{x_{12}} & x_{11} \\
x_{02}\uparrow & x_{22} & \uparrow x_{12} & \overset{\mathsf{S}_{x:A}}{\longmapsto} & & x_{20}\uparrow & x_{22}[\mathsf{S}_{x:A}] & \uparrow x_{21} \\
x_{00} & \xrightarrow[x_{20}]{} & x_{10} & & & x_{00} & \xrightarrow[x_{02}]{} & x_{01}
\end{array}$$

Just as $x_2[\mathsf{R}_{\Gamma.x:A}]$ introduced a new normal form, $x_{22}[\mathsf{S}_{\Gamma.x:A}]$ is a new normal form, however unfortunately we only know how to commute $-^{==}$-d substitutions with S using the naturality rule. For arbitrary substitutions, we are stuck. We come back to this problem in section 2.9.

By iterating $-^=$ on S we observe that at dimension n there are $n-1$ different swap substitutions. $(\mathsf{S}_{\Gamma^i})^{n-2-i}$ swaps dimensions i and $i+1$ of an n-dimensional cube ($n \geq 2$). With the same line of thought as for $(\mathsf{R}_{\Gamma^i})^{n-i}$, we can show that $(\mathsf{S}_{\Gamma^i})^{n-2-i}$ constitutes of

maps $x_{j_0 \dots j_{i-1} k l j_{i+2} \dots j_{n-1}} \mapsto x_{j_0 \dots j_{i-1} l k j_{i+2} \dots j_{n-1}}$ for $k, l = 0, 1, 2$ when k and l are not both 2 and has $x_{j_0 \dots j_{i-1} 22 j_{i+2} \dots j_{n-1}} \mapsto x_{j_0 \dots j_{i-1} 22 j_{i+2} \dots j_{n-1}}[(\mathsf{S}_{\Gamma^i})^{n-2-i}]$ for the other case.

Note that as we do not know how to commute arbitrary substitutions and swap, we also do not have all the rules for computing with compositions of swap substitutions. For example, we would like to have the equality $\mathsf{S}_{\Gamma^=} \mathsf{S}_{\Gamma}{}^{=} \mathsf{S}_{\Gamma^=} \equiv \mathsf{S}_{\Gamma}{}^{=} \mathsf{S}_{\Gamma^=} \mathsf{S}_{\Gamma}{}^{=}$ (having 3 dimensions, swapping dimensions 1–2, then 0–1, then 1–2 gives the same result as swapping 0–1, then 1–2, then 0–1), however this is not yet justified by our rules.

2.7 Sigma Types

We will need Σ types to express the constructs in the next section, we add them using the following rules in the usual way. We have both β and η computation rules.

$$\frac{\Gamma.x : A \vdash B : \mathsf{U}}{\Gamma \vdash \Sigma(x : A).B : \mathsf{U}} \qquad \big(\Sigma(x : A).B\big)[\rho] \equiv \Sigma(x : A[\rho]).B[\rho, x \mapsto x]$$

$$\frac{\Gamma \vdash u : A \qquad \Gamma \vdash v : B[x \mapsto u]}{\Gamma \vdash (u, v) : \Sigma(x : A).B} \qquad \frac{\Gamma \vdash w : \Sigma(x : A).B}{\Gamma \vdash \mathsf{proj}_1\, w : A} \qquad \frac{\Gamma \vdash w : \Sigma(x : A).B}{\Gamma \vdash \mathsf{proj}_2\, w : B[x \mapsto \mathsf{proj}_1\, w]}$$

$$(u, v)[\rho] \equiv (u[\rho], v[\rho]) \qquad (\mathsf{proj}_1\, w)[\rho] \equiv \mathsf{proj}_1\,(w[\rho]) \qquad (\mathsf{proj}_2\, w)[\rho] \equiv \mathsf{proj}_2\,(w[\rho])$$

$$\mathsf{proj}_1\,(u, v) \equiv u \qquad \mathsf{proj}_2\,(u, v) \equiv v \qquad (\mathsf{proj}_1\, w, \mathsf{proj}_2\, w) \equiv w$$

$-^*$ for Σ encodes that pairs are related if they are componentwise related. The relation for the second components depends on the relatedness of the first components (B^* only makes sense in context $(\Gamma.x : A)^=$), this is why we need to substitute it.

$$\begin{aligned}\big(\Sigma(x : A).B\big)^* \equiv \lambda w_0\, w_1.\Sigma\big(&x_2 : A^*\,(\mathsf{proj}_1\, w_0)\,(\mathsf{proj}_1\, w_1)\big)\\ &.B^*[x_0 \mapsto \mathsf{proj}_1\, w_0, x_1 \mapsto \mathsf{proj}_1\, w_1, x_2 \mapsto x_2]\,(\mathsf{proj}_2\, w_0)\,(\mathsf{proj}_2\, w_1)\end{aligned}$$

$-^*$ on the constructor is componentwise and it commutes with the projections.

$$(u, v)^* \equiv (u^*, v^*) \qquad (\mathsf{proj}_1\, w)^* \equiv \mathsf{proj}_1\,(w^*) \qquad (\mathsf{proj}_2\, w)^* \equiv \mathsf{proj}_2\,(w^*)$$

2.8 Univalence

For a type A, $A^*[\mathsf{R}] : A \to A \to \mathsf{U}$ can be viewed as the identity type of A. We have reflexivity of this identity by defining

$$\mathsf{refl}\, a :\equiv a^*[\mathsf{R}],$$

however we cannot show transitivity and symmetry, let alone the J rule. To address this, as a first step we replace the definition of U^*. Instead of relation space, we would like to encode what equality should really be for the universe: equivalence. We use a notion of equivalence which includes a relation. Such a notion is given in exercise 4.2 in [18] where an equivalence of types A and B is given by the following Σ type where $\mathsf{isContr}\, X$ says that the type X is contractible (this is equivalent to the other notions of equivalence in [18]).

$$\Sigma(\sim : A \to B \to \mathsf{U}).\Pi(x : A).\mathsf{isContr}\,\big(\Sigma(y : B).x \sim y\big) \times \Pi(y : B).\mathsf{isContr}\,\big(\Sigma(x : A).x \sim y\big)$$

We replace the definition of U^* by an expansion of this definition. For a type A, A^* will now not only contain the relation, but also the $\mathsf{isContr}$ proofs, hence we need to adjust the

parametricity rule to use the relation part. We can call this rule congruence rule with the view of $\sim_{A^*}$ as the heterogeneous equality for A. $\sim$ projects out the relation part from A^*.

$$\frac{\Gamma \vdash t : A}{\Gamma^= \vdash t^* : t[0] \sim_{A^*} t[1]}$$

Similarly, we need to adjust the computation rule for $-^=$ on context extensions to add the projection $\sim$.

$$(\Gamma.x : A)^= \equiv \Gamma^=.x_0 : A[0_\Gamma].x_1 : A[1_\Gamma].x_2 : x_0 \sim_{A^*} x_1$$

We define equality in the universe by the following iterated Σ-type. The first component is an infix relation and we use $(x \sim)$ to denote $\lambda y.x \sim y$ and $(\sim y)$ to denote $\lambda x.x \sim y$.

$$\begin{aligned}
\sim_{\mathsf{U}^*} \equiv \lambda\, A\, B\,.\, \Sigma - \sim - \quad & : A \to B \to \mathsf{U} \\
\mathsf{coe}^0 \quad & : A \to B \\
\mathsf{coh}^0 \quad & : \Pi(x : A).x \sim \mathsf{coe}^0\, x \\
\mathsf{uncoe}^0 \quad & : \Pi(x : A, y : B, p : x \sim y).\mathsf{coe}^0\, x \sim_{\mathsf{refl}\, B} y \\
\mathsf{uncoh}^0 \quad & : \Pi(x : A, y : B, p : x \sim y).\mathsf{coh}^0\, x \sim_{\mathsf{refl}\, (x\sim)\, (\mathsf{coe}^0\, x)\, y\, (\mathsf{uncoe}^0\, x\, y\, p)} p \\
\mathsf{coe}^1 \quad & : B \to A \\
\mathsf{coh}^1 \quad & : \Pi(y : B).\mathsf{coe}^1\, y \sim y \\
\mathsf{uncoe}^1 \quad & : \Pi(x : A, y : B, p : x \sim y).\mathsf{coe}^1\, y \sim_{\mathsf{refl}\, A} x \\
\mathsf{uncoh}^1 \quad & : \Pi(x : A, y : B, p : x \sim y).\mathsf{coh}^1\, y \sim_{\mathsf{refl}\, (\sim y)\, (\mathsf{coe}^1\, y)\, x\, (\mathsf{uncoe}^1\, x\, y\, p)} p \\
: \mathsf{U} \to \mathsf{U} \to \mathsf{U} &
\end{aligned}$$

An equality between two types $A \sim_{\mathsf{U}^*} B$ is a relation between them, coercion functions (coe) from A to B and backwards, together with coherence proofs (coh) that the coercions respect the relations and uniqueness proofs (uncoe, uncoh) stating that any equality is equal to a coercion. We can depict these conditions as follows. The components with 0 indices give us the left square, the components with 1 indices give the middle square, while the right square shows the types of the other two squares.

$$\begin{array}{ccc}
x & \xrightarrow{\;p\;} & y \\
\mathsf{refl}\, x \uparrow & \mathsf{uncoh}^0\, x\, y\, p & \uparrow \mathsf{uncoe}^0\, x\, y\, p \\
x & \xrightarrow[\mathsf{coh}^0\, x]{} & \mathsf{coe}^0\, x
\end{array}
\qquad
\begin{array}{ccc}
x & \xrightarrow{\;p\;} & y \\
\mathsf{uncoe}^1\, x\, y\, p \uparrow & \mathsf{uncoh}^1\, x\, y\, p & \uparrow \mathsf{refl}\, y \\
\mathsf{coe}^1\, y & \xrightarrow[\mathsf{coh}^1\, y]{} & y
\end{array}
\quad : \quad
\begin{array}{ccc}
A & \xrightarrow{\;\sim\;} & B \\
\sim_{\mathsf{refl}\, A} \uparrow & R & \uparrow \sim_{\mathsf{refl}\, B} \\
A & \xrightarrow[\sim]{} & B
\end{array}$$

$$R\, x_{00}\, x_{01}\, x_{02}\, x_{10}\, x_{11}\, x_{12}\, x_{20}\, x_{21} \equiv x_{20} \sim_{\mathsf{refl} \sim x_{00}\, x_{01}\, x_{02}\, x_{10}\, x_{11}\, x_{12}} x_{21}$$

Note that the type of the squares is degenerate in the vertical dimension. uncoe says that any y which is equal to x (by p) is equal to the coercion of x, while uncoh gives the equality of the coherence and the equality proof p. Using the definition of refl, we can unfold the types of the uncoh components as follows.

$$\begin{aligned}
&\mathsf{uncoh}^0\, x\, y\, p : \mathsf{coh}^0\, x \sim_{(x\sim y)^*[\mathsf{R}_{A:\mathsf{U}.B:\mathsf{U}.\sim:A\to B\to\mathsf{U}.x:A}, y_0 \mapsto \mathsf{coe}^0\, x, y_1 \mapsto y, y_2 \mapsto \mathsf{uncoe}^0\, x\, y\, p]} p \\
&\mathsf{uncoh}^1\, x\, y\, p : \mathsf{coh}^1\, y \sim_{(x\sim y)^*[\mathsf{R}_{A:\mathsf{U}.B:\mathsf{U}.\sim:A\to B\to\mathsf{U}.y:B}, x_0 \mapsto \mathsf{coe}^1\, y, x_1 \mapsto x, x_2 \mapsto \mathsf{uncoe}^1\, x\, y\, p]} p
\end{aligned}$$

As we replaced the relation space for U^* by the above definition, we need to define the new components $\sim$, (un)coe and (un)coh for the type formers U, Π and Σ. The $\sim$ component

for U^* is defined above, and for Π and Σ we just use the relations given in the previous sections, i.e.

$$\sim_{(\Pi(x:A).B)^*} \equiv \lambda f_0\, f_1.\Pi(x_0 : A[0], x_1 : A[1], x_2 : x_0 \sim_{A^*} x_1).f_0\, x_0 \sim_{B^*} f_1\, x_1$$
$$\sim_{(\Sigma(x:A).B)^*} \equiv \lambda w_0\, w_1.\Sigma(x_2 : \mathsf{proj}_1\, w_0 \sim_{A^*} \mathsf{proj}_1\, w_1)$$
$$.\mathsf{proj}_2\, w_0 \sim_{B^*[x_0 \mapsto \mathsf{proj}_1\, w_0, x_1 \mapsto \mathsf{proj}_2\, w_1, x_2 \mapsto x_2]} \mathsf{proj}_2\, w_1.$$

First we show how to define coe and coh for U and Σ (section 2.8.1), then we show how to derive coercion and coherence for squares (2-dimensional version of the 1-dimensional coe and coh) and show how to use this to derive uncoe and uncoh (section 2.8.2). Then we derive coe and coh for Π (section 2.8.3). Finally, we show how to derive the usual elimination rule for homogeneous equality (section 2.8.4).

2.8.1 coe and coh for U and Σ

We use the following abbreviation to concisely express $\sim$ in both directions.

$$x \overset{0}{\sim}_e y \equiv x \sim_e y \qquad y \overset{1}{\sim}_e x \equiv x \sim_e y$$

For U we simply use the identity function as coercion and reflexivity for coherence.

$$\Gamma^= \vdash \mathsf{coe}^b_{\mathsf{U}^*} \equiv \mathsf{id} : \mathsf{U} \to \mathsf{U}$$
$$\Gamma^= \vdash \mathsf{coh}^b_{\mathsf{U}^*} \equiv \mathsf{refl} : \Pi(A : \mathsf{U})\,.\,A \overset{b}{\sim}_{\mathsf{U}^*} A$$

For Σ types, coercion and coherence are pointwise.

$$\Gamma^= \vdash \mathsf{coe}^b_{(\Sigma(x:A).B)^*} \equiv \lambda(u,v).(\mathsf{coe}^b_{A^*}\, u, \mathsf{coe}^b_{B^*}[x_2 \mapsto \mathsf{coh}^b_{A^*}\, u]\, v)$$
$$: (\Sigma(x : A).B)[b] \to (\Sigma(x : A).B)[1-b]$$
$$\Gamma^= \vdash \mathsf{coh}^b_{(\Sigma(x:A).B)^*} \equiv \lambda(u,v).(\mathsf{coh}^b_{A^*}\, u, \mathsf{coh}^b_{B^*}[x_2 \mapsto \mathsf{coh}^b_{A^*}\, u]\, v)$$
$$: \Pi(w : (\Sigma(x : A).B)[b])\,.\,w \overset{b}{\sim}_{(\Sigma(x:A).B)^*} \mathsf{coe}^b_{(\Sigma(x:A).B)^*}\, w$$

We used pattern matching lambdas for ease of notation and in the substitution $[x_2 \mapsto \mathsf{coh}^b_{A^*}\, u]$ we left the x_0, x_1 components implicit. We need the $[x_2 \mapsto \mathsf{coh}^b_{A^*}\, u]$ substitution when we using coe_{B^*} and coh_{B^*} because B is in the context $\Gamma, x : A$, hence B^* is in the context $(\Gamma, x : A)^=$ which includes the components $x_0 : A[0], x_1 : A[1], x_2 : x_0 \sim_{A^*} x_1$ and we need to provide these components.

2.8.2 Deriving coe, coh for Squares from coe, coh for Lines

A square with a missing top line and a filler can be depicted as follows.

$$\begin{array}{ccc} x_{01} & & x_{11} \\ {\scriptstyle x_{02}}\uparrow & & \uparrow{\scriptstyle x_{12}} \\ x_{00} & \xrightarrow[x_{20}]{} & x_{10} \end{array}$$

This corresponds to the last 7 components of the context $(\Gamma^=.x_0 : A[0].x_1 : A[1])^=.x_{20} : x_{00} \sim_{A^*[0]} x_{10}$. Coercion for the type $x_0 \sim_{A^*} x_1$ in this context has the following type.

$$(\Gamma^=.x_0 : A[0].x_1 : A[1])^= \vdash \mathsf{coe}^0_{(x_0 \sim_{A^*} x_1)^*} : x_{00} \sim_{A^*[0]} x_{10} \to x_{01} \sim_{A^*[1]} x_{11}$$

We can use this coercion to get the top of the square from x_{20} and similarly, we can use coherence to obtain the filler. Putting them together we get a substitution which fills in the last two components of the square.

$$\begin{aligned}
\mathsf{K}^{0,0}_{\Gamma.x:A} :\equiv \big(&\mathsf{id}_{(\Gamma^=.x_0:A[0].x_1:A[1])^=},\\
&x_{20} \mapsto x_{20}, x_{21} \mapsto \mathsf{coe}^0_{(x_0 \sim_{A^*} x_1)^*}\, x_{20}, x_{22} \mapsto \mathsf{coh}^0_{(x_0 \sim_{A^*} x_1)^*}\, x_{20}\big)\\
&: (\Gamma^=.x_0 : A[0].x_1 : A[1])^=.x_{20} : x_{00} \sim_{A^*[0]} x_{10} \Rightarrow (\Gamma.x : A)^{==}
\end{aligned}$$

If the right side of a square is missing, we can use the swap substitution to obtain a filler. First we swap the components we have, then apply the above $\mathsf{K}^{0,0}_{\Gamma.x:A}$, then swap the components back so that everything stays in its original place. We denote this substitution $\mathsf{K}^{1,0}_{\Gamma.x:A}$.

$$(\Gamma^=.x : A[0])^=.x_{10} : A[10].x_{11} : A[11].x_{20} : x_{00} \sim_{A^*[0]} x_{10}.x_{21} : x_{01} \sim_{A^*[1]} x_{11}$$

$$\Big\downarrow\; (\mathsf{S}_\Gamma, x_{00} \mapsto x_{00}, x_{01} \mapsto x_{10}, x_{02} \mapsto x_{20}, x_{10} \mapsto x_{01}, x_{11} \mapsto x_{11}, x_{12} \mapsto x_{21}, x_{20} \mapsto x_{02})$$

$$(\Gamma^=.x_0 : A[0].x_1 : A[1])^=.x_{20} : x_{00} \sim_{A^*[0]} x_{10}$$

$$\Big\downarrow\; \mathsf{K}^{0,0}_{\Gamma.x:A}$$

$$(\Gamma.x : A)^{==}$$

$$\Big\downarrow\; \mathsf{S}_{\Gamma.x:A}$$

$$(\Gamma.x : A)^{==}$$

With the help of $\mathsf{K}^{1,0}$ we can define uncoe^0 and uncoh^0 for a given type $\Gamma \vdash A : \mathsf{U}$. We use a substitution ρ to define the common parts.

$$\begin{aligned}
\rho :\equiv \big(&(\mathsf{R}_\Gamma)^=, x_{00} \mapsto x, x_{01} \mapsto x, x_{02} \mapsto \mathsf{refl}\, x,\\
&x_{10} \mapsto \mathsf{coe}_{A^*}\, x, x_{11} \mapsto y, x_{20} \mapsto \mathsf{coh}_{A^*}\, x, x_{21} \mapsto p\big)\\
&: (\Gamma^=.x : A[0].y : A[1].p : x \sim_{A^*} y) \Rightarrow (\Gamma.x : A)^2\\
\Gamma^= \vdash\ &\mathsf{uncoe}_{A^*} \equiv \lambda\, x\, y\, p.x_{12}[\mathsf{K}^{1,0}_{\Gamma.x:A}\rho] : \Pi(x : A, y : B, p : x \sim y).\mathsf{coe}^0\, x \sim_{\mathsf{refl}\, B}\, y\\
\Gamma^= \vdash\ &\mathsf{uncoh}_{A^*} \equiv \lambda\, x\, y\, p.x_{22}[\mathsf{K}^{1,0}_{\Gamma.x:A}\rho] : \Pi(x : A, y : B, p : x \sim y)\\
&\qquad .\mathsf{coh}^0\, x \sim_{\mathsf{refl}\,(x\sim)\,(\mathsf{coe}^0\, x)\, y\, (\mathsf{uncoe}^0\, x\, y\, p)}\, p
\end{aligned}$$

We can analogously define $\mathsf{K}^{0,1}$ and $\mathsf{K}^{1,1}$ which go in the opposite directions (and $\mathsf{K}^{1,1}$ can be used to define uncoe^1 and uncoh^1). As a summary, we list here the directions of filling for a square that we get using these substitutions.

$\mathsf{K}^{0,0}$	bottom to top	$\uparrow$
$\mathsf{K}^{0,1}$	top to bottom	$\downarrow$
$\mathsf{K}^{1,0}$	left to right	$\rightarrow$
$\mathsf{K}^{1,1}$	right to left	$\leftarrow$

2.8.3 coe and coh for Π

The coerce operation for Π types takes a function of type $(\Pi(x : A).B)[0]$ and needs to return a function of type $(\Pi(x : A).B)[1]$. We define the latter function by first coercing backwards from $A[1]$ to $A[0]$, then applying the original function, then coercing forward the result from

$B[0]$ to $B[1]$. We define it for both directions as follows.

$$\begin{aligned}\Gamma^{=} \vdash \mathsf{coe}^{b}_{(\Pi(x:A).B)^*} \equiv\, & \lambda f.\lambda x.\mathsf{coe}^{b}_{B^*}[x_2 \mapsto \mathsf{coh}^{1-b}_{A^*}\, x]\,(f\,(\mathsf{coe}^{1-b}_{A^*}\, x)) \\ & : (\Pi(x:A).B)[b] \to (\Pi(x:A).B)[1-b]\end{aligned}$$

We left the other two components of the substitution $[x_2 \mapsto \mathsf{coh}^{1-b}_{A^*}\, x]$ implicit.

The coherence operation for Π needs to have the following type.

$$\begin{aligned}\Gamma^{=} \vdash \mathsf{coh}^{b}_{(\Pi(x:A).B)^*} :& \Pi\big(f : (\Pi(x:A).B)[b], x_0 : A[0], x_1 : A[1], x_2 : x_0 \sim_{A^*} x_1\big) \\ & .\, f\, x_b \overset{b}{\sim}_{B^*} \mathsf{coe}^{b}_{B^*}[x_2 \mapsto \mathsf{coh}^{1-b}_{A^*}\, x_{1-b}]\,(f\,(\mathsf{coe}^{1-b}_{A^*}\, x_{1-b}))\end{aligned}$$

We explain in detail how to define $\mathsf{coh}^{0}_{(\Pi(x:A)B)^*}\, f\, x_0\, x_1\, x_2$, the other direction is symmetric.

We start in the context $\Gamma^{=}.f : (\Pi(x:A).B)[0].x_0 : A[0].x_1 : A[1].x_2 : x_0 \sim_{A^*} x_1$. First we will fill the left incomplete square thereby obtaining the dashed line r and using this r we construct the right incomplete square and filling it gives us the line in the bottom which is exactly what we need. Note that the left square has telescope type $(x : A)^2[(\mathsf{R}_\Gamma)^{=}]$ while the right square has telescope type $(y : B)^2[(\mathsf{R}_\Gamma)^{=}]$ and depends on the first square (by $(x : A)^2$ we mean the telescope $x_{00} : A[00], x_{01} : A[01], \ldots, x_{22} : x_{20} \sim_{(x_0 \sim_{A^*} x_1)^*} x_{21}$ which contains the last 9 elements of the context $(\Gamma, x : A)^{==}$).

$$\begin{array}{ccc} \mathsf{coe}^{1}_{A^*}\, x_1 & \xrightarrow{\mathsf{coh}^{1}_{A^*}\, x_1} & x_1 \\ {\scriptstyle r}\uparrow & & \uparrow{\scriptstyle \mathsf{refl}\, x_1} \\ x_0 & \xrightarrow[x_2]{} & x_1 \end{array} \qquad \begin{array}{ccc} f\,(\mathsf{coe}^{1}_{A^*}\, x_1) & \xrightarrow{\mathsf{coh}^{0}_{B^*}[\ldots]\,(f\,(\mathsf{coe}^{1}_{A^*}\, x_1))} & \mathsf{coe}^{0}_{B^*}[\ldots]\,(f\,(\mathsf{coe}^{1}_{A^*}\, x_1)) \\ {\scriptstyle \mathsf{refl}\, f\, x_0\, (\mathsf{coe}^{1}_{A^*}\, x_1)\, r}\uparrow & & \uparrow{\scriptstyle \mathsf{refl} \ldots} \\ f\, x_0 & & \mathsf{coe}^{0}_{B^*}[\ldots]\,(f\,(\mathsf{coe}^{1}_{A^*}\, x_1)) \end{array}$$

First we define the incomplete first square as a substitution ρ.

$$\begin{aligned}\rho \equiv & \big((\mathsf{R}_\Gamma)^{=}, x_{00} \mapsto x_0, x_{01} \mapsto \mathsf{coe}^{1}_{A^*}\, x_1, x_{10} \mapsto x_1, x_{11} \mapsto x_1, x_{12} \mapsto \mathsf{refl}\, x_1, \\ & x_{20} \mapsto x_2, x_{21} \mapsto \mathsf{coh}^{1}_{A^*}\, x_1\big) \\ & : \big(\Gamma^{=}.f : (\Pi(x:A).B)[0].x_0 : A[0].x_1 : A[1].x_2 : x_0 \sim_{A^*} x_1\big) \\ & \Rightarrow \Gamma^2.x_{00} : A[00].x_{01} : A[01].x_{10} : A[10].x_{11} : A[11].x_{12} : x_{10} \sim_{(A[1])^*} x_{11} \\ & \quad .x_{20} : x_{00} \sim_{A^*[0]} x_{10}.x_{21} : x_{01} \sim_{A^*[1]} x_{11}\end{aligned}$$

The following σ substitution defines the (complete) first square and the incomplete second square.

$$\begin{aligned}\sigma \equiv \big((\mathsf{R}_\Gamma)^{=}, & \ x_{00} \mapsto x_0, x_{01} \mapsto \mathsf{coe}^{1}_{A^*}\, x_1, x_{02} \mapsto x_{02}[\mathsf{K}^{1,1}][\rho], \\ & x_{10} \mapsto x_1, x_{11} \mapsto x_1, x_{12} \mapsto \mathsf{refl}\, x_1 \\ & x_{20} \mapsto x_2, x_{21} \mapsto \mathsf{coh}^{1}_{A^*}\, x_1, x_{22} \mapsto x_{22}[\mathsf{K}^{1,1}][\rho], \\ & y_{00} \mapsto f\, x_0, y_{01} \mapsto f\,(\mathsf{coe}^{1}_{A}\, x_1), y_{02} \mapsto f^*[\mathsf{R}_{\Gamma^=}]\, x_0\,(\mathsf{coe}^{1}_{A}\, x_1)\,(x_{02}[\mathsf{K}^{1,1}][\rho]) \\ & y_{10}, y_{11} \mapsto \mathsf{coe}^{0}_{B^*}[x_2 \mapsto \mathsf{coh}^{1}_{A^*}\, x]\,(f\,(\mathsf{coe}^{1}_{A^*}\, x)), \\ & y_{12} \mapsto \mathsf{refl}\,(\mathsf{coe}^{0}_{B^*}[x_2 \mapsto \mathsf{coh}^{1}_{A^*}\, x]\,(f\,(\mathsf{coe}^{1}_{A^*}\, x)))\big) \\ : \big(\Gamma^{=}.&f : (\Pi(x:A).B)[0].x_0 : A[0].x_1 : A[1].x_2 : x_0 \sim_{A^*} x_1\big) \\ \Rightarrow ((&\Gamma.x : A)^{=}.y_0 : B[0].y_1 : B[1])^{=}.y_{21} : y_{01} \sim_{B^*[1]} y_{11}\end{aligned}$$

Using σ we can define the coherence operation for Π as follows.

$$\begin{aligned}&\Gamma^{=} \vdash \mathsf{coh}^0_{(\Pi(x:A).B)^*} \equiv \lambda f\, x_0\, x_1.\, x_2.\mathsf{coe}^1_{(y_0 \sim_{B^*} y_1)^*[\sigma]}\big(\mathsf{coh}^0_{B^*}[x_2 \mapsto \mathsf{coh}^1_{A^*}\, x]\,(f\,(\mathsf{coe}^1_{A^*}\, x))\big)\\ &\quad : \Pi\big(f : (\Pi(x : A).B)[0], x_0 : A[0], x_1 : A[1], x_2 : x_0 \sim_{A^*} x_1\big).f\, x_0 \sim_{B^*} \mathsf{coe}_{(\Pi(x:A).B)^*}\, f\, x_1\end{aligned}$$

2.8.4 The Elimination Rule for Equality

We define the homogeneous equality $\Gamma \vdash a =_A b : \mathsf{U}$ as $\Gamma \vdash a \sim_{A^*[\mathsf{R}_\Gamma]} b : \mathsf{U}$.

We express the eliminator for equality as the $\mathsf{transport}$ function and the fact that singletons are contractible. From these two principles the eliminator J can be derived.

$$\frac{\Gamma \vdash P : A \to \mathsf{U} \quad \Gamma \vdash r : a =_A b \quad \Gamma \vdash u : P\, a}{\Gamma \vdash \mathsf{transport}_P\, r\, u : P\, b} \qquad \frac{\Gamma \vdash a, b : A \quad \Gamma \vdash r : a =_A b}{\Gamma \vdash\ (s, t) : (a, \mathsf{refl}\, a) =_{\Sigma(x:A).a =_A x} (b, r)}$$

We validate transport using coe^0 and $-^*$ on the predicate P.

$$\Gamma \vdash \mathsf{transport}_P\, r\, u \equiv \mathsf{coe}^0_{P^*[\mathsf{R}_\Gamma]\, a\, b\, r}\, u : P\, b$$

The computation rule for transport is validated by the corresponding coherence law (up to equality, but not definitional equality). $\mathsf{coh}^0_{P^*[\mathsf{R}_\Gamma]\, a\, a\, (\mathsf{refl}\, a)}\, u : u =_{P\, a} \mathsf{transport}_P\, (\mathsf{refl}\, a)\, u\, u$.

The type of (s, t) in the second principle is equal to

$$\Sigma(s : a \sim_{A^*[\mathsf{R}_\Gamma]} b).\mathsf{refl}\, a \sim_{(a \sim_{A^*[\mathsf{R}_\Gamma]} x)^*[\mathsf{R}_\Gamma, a, b, s]} r.$$

We construct s by filling the following incomplete square from bottom to top.

$$\begin{array}{ccc} a & \overset{s}{\dashrightarrow} & b \\ {\scriptstyle \mathsf{refl}\, a}\uparrow & & \uparrow{\scriptstyle r} \\ a & \underset{\mathsf{refl}\, a}{\longrightarrow} & a \end{array}$$

We spell it out explicitly below.

$$\begin{aligned}&\rho :\equiv (\mathsf{R}_{\Gamma^{=}}\mathsf{R}_\Gamma, x_{00} \mapsto a, x_{01} \mapsto a, x_{02} \mapsto \mathsf{refl}\, x_{10} \mapsto a, x_{11} \mapsto b, x_{12} \mapsto r)\\ &\quad : \Gamma \Rightarrow (\Gamma^{=}.x_0 : A[0].x_1 : A[1])^{=}\\ &\Gamma \vdash s :\equiv \mathsf{coe}^0_{(x_0 \sim_{A^*} x_1)^*}[\rho](\mathsf{refl}\, a) : a \sim_{A^*[\mathsf{R}_\Gamma]} b\end{aligned}$$

The coherence gives us the filler, however we need to swap it to get the type that we need. Without swapping we get the following type.

$$\Gamma \vdash \mathsf{coh}^0_{(x_0 \sim_{A^*} x_1)^*}[\rho](\mathsf{refl}\, a) : \mathsf{refl}\, a \sim_{(x_0 \sim_{A^*} x_1)^*[\rho]} s$$

We define a substitution σ and compose it with $\mathsf{S}_{\Gamma.x:A}$ and we define t as the last term in the resulting substitution.

$$\Gamma \xrightarrow{\quad \sigma \quad} (\Gamma.x : A)^{==} \xrightarrow{\mathsf{S}_{\Gamma.x:A}} (\Gamma.x : A)^{==}$$

$$\begin{aligned}\sigma :\equiv \big(\mathsf{R}_{\Gamma^{=}}\mathsf{R}_\Gamma, &x_{00} \mapsto a, x_{01} \mapsto a, x_{02} \mapsto \mathsf{refl}\, a,\\ &x_{10} \mapsto a, x_{11} \mapsto b, x_{12} \mapsto r,\\ &x_{20} \mapsto \mathsf{refl}\, a, x_{21} \mapsto \mathsf{coe}^0_{(x_0 \sim_{A^*} x_1)^*}[\rho](\mathsf{refl}\, a), x_{22} \mapsto \mathsf{coh}^0_{(x_0 \sim_{A^*} x_1)^*}[\rho](\mathsf{refl}\, a)\big)\end{aligned}$$

$$\Gamma \vdash t :\equiv x_{22}[\mathsf{S}_{\Gamma.x:A}\sigma] : \mathsf{refl}\, a \sim_{(a \sim_{A^*[\mathsf{R}_\Gamma]} x)^*[\mathsf{R}_\Gamma, a, b, s]} r$$

2.9 Metatheoretic Properties

The operational semantics for the theory of external parametricity (section 2.3) is clear, we can always eliminate the new constructs $-^{=}$, $-^{*}$, $\mathsf{0}$, $\mathsf{1}$. We formalised the unary version of this theory as a syntactic translation, see [2].

After adding reflexivities (section 2.4), we lose the property that we can translate everything back to the original theory as we have new normal forms such as $x_2[\mathsf{R}_{\Gamma.x:A}]$. And even worse, we do not know how to substitute arbitrary substitutions into $x_{22}[(\mathsf{R}_{\Gamma.x:A})^{=}]$, i.e. $x_{22}[(\mathsf{R}_{\Gamma.x:A})^{=}\rho]$ is stuck. We can defer the problem by introducing S and expressing $(\mathsf{R}_{\Gamma})^{=}$ as $\mathsf{S}_{\Gamma}\mathsf{R}_{\Gamma^{=}}$, but then we need to commute S and arbitrary substitutions which we do not know how to do. This is a problem indeed in practice, e.g. we do not know how to compute with $\mathsf{K}^{1,0}$ from section 2.8.2 which is defined by postcomposing S with another substitution. We describe a possible solution in section 3, but this is notationally quite heavy.

We believe however that the syntax up to section 2.6 has a presheaf model with base category the category of renamings from [14]. This category does not include connections [12], only face maps (corresponding to our $\mathsf{0}$ and $\mathsf{1}$), degeneracies (corresponding to R) and renamings (corresponding to S). The idea of the construction is as follows. We denote the set with n elements (an object of the base category) $\overline{n}$. An interpretation of a context Γ is a covariant presheaf $[\![\Gamma]\!] : \mathcal{C} \to \mathbf{Set}$, and for lifted contexts we define $[\![\Gamma^{=}]\!]\,\overline{n} := [\![\Gamma]\!]\,\overline{n+1}$. The interpretations of $\mathsf{0}$ and $\mathsf{1}$ are natural transformations and we set $[\![\mathsf{0}_{\Gamma}]\!]_{\overline{n}} := [\![\Gamma]\!]\,(d_n = 0)$ i.e. we use the action on morphisms for $[\![\Gamma]\!]$ at the morphism which sends dimension n to 0. A lifted substitution is the unlifted substitution at a higher dimensional object, $[\![\rho^{=}]\!]_{\overline{n}} := [\![\rho]\!]_{\overline{n+1}}$. A type $\Gamma \vdash A$ is interpreted as a family of presheaves over $[\![\Gamma]\!]$. A lifted type $A^{*}\,x_0\,x_1$ (which depends on the last two elements in the context being $x_0 : A[\mathsf{0}], x_1 : A[\mathsf{1}]$) is interpreted as the element of the higher type where the projections give the two faces in the context.

$$[\![A^{*}\,x_0\,x_1]\!]_{\overline{n}}\,(\gamma, a_0, a_1) := \big(a : [\![A]\!]_{\overline{n+1}}\,\gamma\big) \times [\![A]\!]\,(d_n = 0)\,a = a_0 \times [\![A]\!]\,(d_n = 1)\,a = a_1$$

Terms are interpreted as sections, lifted terms are interpreted as triples corresponding to the above sum types: $[\![t^{*}]\!]_{\overline{n}}\,\gamma := ([\![t]\!]_{\overline{n+1}}\,\gamma, \mathsf{refl}, \mathsf{refl})$. Some equalities however are only validated up to isomorphism in this model. E.g. $-^{=}$ on context extension is only validated up to an isomorphism which follows from the fact that singletons are contractible.

$$\begin{aligned}
& [\![(\Gamma.x : A)^{=}]\!]_{\overline{n}} \\
= {} & (\gamma : [\![\Gamma]\!]_{\overline{n+1}}) \times [\![A]\!]_{\overline{n+1}}\,\gamma \\
\simeq {} & \quad (\gamma : [\![\Gamma]\!]_{\overline{n+1}}) \times \big(a_0 : [\![A]\!]\,([\![\Gamma]\!]\,(d_n = 0)\,\gamma)\big) \times \big(a_1 : [\![A]\!]\,([\![\Gamma]\!]\,(d_n = 1)\,\gamma)\big) \\
& \times \big(a : [\![A]\!]_{\overline{n+1}}\,\gamma\big) \times [\![A]\!]\,(d_n = 0)\,a = a_0 \times [\![A]\!]\,(d_n = 1)\,a = a_1 \\
= {} & [\![\Gamma^{=}.x_0 : A[\mathsf{0}].x_1 : A[\mathsf{1}].x_2 : A^{*}\,x_0\,x_1]\!]_{\overline{n}}
\end{aligned}$$

This can be remedied using a univalent metatheory (where isomorphic sets are equal) or by a refinement of the presheaf model as in [7] where presheaves return sets with additional structure (called I-sets for an object I (set of dimension names) in the base category; an I-set is a set of I-indexed tuples).

We also believe that the presheaf model of the syntax with the Kan operations (section 2.8) has Kan operations with the uniformity condition as in [14].

3 Cubical Type theory with Named Dimensions

In this section we remedy the problem of the previous approach which was that we did not know how to compute with the S substitutions. We define a new syntax where each usage of

the $-^{=}$ and $-^{*}$ operators is decorated with a dimension name. A two-dimensional context $\Gamma^{==}$ will be denoted Γ^{ij} where i and j are dimension names. They are assumed to be fresh in every rule. With named dimensions at our hand, we will equate the contexts Γ^{ij} and Γ^{ji} as they contain the same information in different order. A context $(x : A)^{ij}$ can be viewed as a collection of types indexed by $i = 0, 1, 2$ and $j = 0, 1, 2$ regardless of the order; i and j are the dimension names, and given a coordinate for each dimension, we get a type. The information about which dimension has which index will be stored in the variable names, e.g. x_{i0j1}, x_{i1j2}. This eliminates the need of the swapping substitution S and the problem with its computation rules. However the theory becomes more involved as we need Π types to remember their dimensions as well. As higher dimensional contexts are order-agnostic, lifted Π types need to be order-agnostic too, hence they need to carry information about their dimensions. Instead of defining the lifted relation $(\Pi(x : A).B)^i$ as an iterated Π type $\Pi(x_{i0} : A[\mathsf{0}]).\Pi(x_{i1} : A[\mathsf{1}].\Pi(x_{i2} : x_{i0} \sim_{A^i} x_{i1}) \ldots$, we will use $\Pi(x : A)^i \ldots$ which shows that this type has arguments of dimension i.

We start with the core substitution calculus given in section 2.1. First we add telescopes to the calculus in section 3.1. Then we define the function space with dimensional information in section 3.2. Sections 3.3 and 3.4 show how to define the parametricity operation (called $-^{=}$ before) for this calculus. Section 3.5 adds a definitional quotient thereby e.g. equating Γ^{ij} and Γ^{ji}. We need this quotient to internalise parametricity in section 3.6. This version corresponds to section 2.4 of the naive calculus. We stop here and do not interpret univalence in the nominal calculus. In appendix A we define an operational semantics.

3.1 Telescopes

We add new judgment types to the core substitution calculus defined in section 2.1.

$\Gamma \vdash \Omega$	Ω is a telescope context in context Γ
$\Gamma \vdash \omega : \Omega$	ω is a telescope substitution of type Ω in context Γ
$\Gamma \vdash \Omega \equiv \Omega'$	definitional equality of telescope contexts
$\Gamma \vdash \omega \equiv \omega' : \Omega$	definitional equality of telescope substitutions

Explanation:

$$\begin{array}{llll} \text{Just as} & \Gamma \vdash t : A & \text{can be viewed as} & (\mathsf{id}_\Gamma, x \mapsto t) : \Gamma \Rightarrow \Gamma.x : A, \\ & \Gamma \vdash \Omega & \text{can be viewed as} & \Gamma +\!\!+ \Omega \vdash \\ \text{and} & \Gamma \vdash \omega : \Omega & \text{can be viewed as} & (\mathsf{id}_\Gamma +\!\!+ \omega) : \Gamma \Rightarrow \Gamma +\!\!+ \Omega. \end{array}$$

Telescope contexts are generalisations of types into lists of types which might depend on each other. They can also be thought of as named iterated Σ-types.

$$\frac{\Gamma \vdash}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash \Omega \quad \Gamma +\!\!+ \Omega \vdash A : \mathsf{U}}{\Gamma \vdash \Omega.x : A}$$

We explain how to extend contexts with telescope contexts:

$$\frac{\Gamma \vdash \Omega}{\Gamma +\!\!+ \Omega \vdash} \qquad \Gamma +\!\!+ \cdot \equiv \Gamma \qquad \Gamma +\!\!+ (\Omega.x : A) \equiv (\Gamma +\!\!+ \Omega).x : A$$

Substitution of telescope contexts is pointwise:

$$\frac{\Gamma \vdash \Omega \quad \rho : \Delta \Rightarrow \Gamma}{\Delta \vdash \Omega[\rho]} \qquad \cdot[\rho] \equiv \cdot \qquad (\Omega.x : A)[\rho] \equiv \Omega[\rho].x : A[\rho]$$

Telescope substitutions are generalisations of terms into lists of terms.

$$\frac{\Gamma \vdash}{\Gamma \vdash \epsilon : \cdot} \qquad \frac{\Gamma \vdash \omega : \Omega \quad \Gamma +\!\!+ \Omega \vdash A : \mathsf{U} \quad \Gamma \vdash t : A[(\mathsf{id}_\Gamma +\!\!+ \omega)]}{\Gamma \vdash (\omega, x \mapsto t) : \Omega.x : A}$$

We explain how to extend normal substitutions with telescope substitutions:

$$\frac{\rho : \Delta \Rightarrow \Gamma \quad \Delta \vdash \omega : \Omega[\rho]}{\rho +\!\!+ \omega : \Delta \Rightarrow \Gamma +\!\!+ \Omega} \qquad \rho +\!\!+ \epsilon \equiv \rho \qquad \rho +\!\!+ (\omega, x \mapsto t) \equiv (\rho +\!\!+ \omega, x \mapsto t)$$

Substitution of telescope substitutions is pointwise:

$$\frac{\Gamma \vdash \omega : \Omega \quad \rho : \Delta \Rightarrow \Gamma}{\Delta \vdash \omega[\rho] : \Omega[\rho]} \qquad \epsilon[\rho] \equiv \epsilon \qquad (\omega, x \mapsto t)[\rho] \equiv (\omega[\rho], x \mapsto t[\rho])$$

Extending telescope contexts with telescope contexts and telescope substitutions with telescope substitutions is done in the obvious way. Specification:

$$\frac{\Gamma \vdash \Omega \quad \Gamma +\!\!+ \Omega \vdash \Omega'}{\Gamma \vdash \Omega +\!\!+ \Omega'} \qquad \Omega +\!\!+ \cdot \equiv \Omega \qquad \Omega +\!\!+ (\Omega'.x : A) \equiv \Omega +\!\!+ \Omega'.x : A$$

$$\frac{\Gamma \vdash \omega : \Omega \quad \Gamma \vdash \omega' : \Omega'[\omega]}{\Gamma \vdash \omega +\!\!+ \omega' : \Omega +\!\!+ \Omega'} \qquad \omega +\!\!+ \epsilon \equiv \omega \qquad \omega +\!\!+ (\omega', x \mapsto t) \equiv (\omega +\!\!+ \omega', x \mapsto t)$$

The pr telescope substitution generalises the projection $\Gamma.x : A \vdash x : A$.

$$\frac{\Gamma \vdash \Omega}{\Gamma +\!\!+ \Omega \vdash \mathsf{pr}_\Omega : \Omega} \qquad \Gamma +\!\!+ \cdot \vdash \mathsf{pr}_\cdot \equiv \epsilon : \cdot \qquad \Gamma +\!\!+ (\Omega.x : A) \vdash \mathsf{pr}_{\Omega.x:A} \equiv (\mathsf{pr}_\Omega, x \mapsto x) : (\Omega.x : A)$$

Note that we have not added new types in the universe, telescopes live outside the world of types.

3.2 Function Space

We would like to have $\Gamma^{ij} \equiv \Gamma^{ji}$. This forces us to provide a new type former for $\Pi(x : A)^{ij}.B$ instead of using the iterated $\Pi(x_{i0j0} : A[\mathsf{0}_i\mathsf{0}_j]).\Pi(x_{i0j1} : A[\mathsf{0}_i\mathsf{1}_j]).\ldots.B$. When applying arguments to such a function, we need to supply all of them at the same time, because the order of arguments will not matter, only their dimension indices.

For this reason, we add functions where the domain can be a telescope context. However, we only allow special telescope contexts, ones which arise from applying the lifting operator zero or more times. Hence, a function will be able to have only 3^k number of arguments where k is the number of dimensions. We will denote such a telescope context $(x : A)^I$, where I is a set of dimensions which might be empty.

Also, to express the type of the relation $A^i : A[\mathsf{0}_i] \to A[\mathsf{1}_i] \to \mathsf{U}$, we need functions with incomplete cube arguments, $\{x : A\}_I$ will denote $(x : A)^I$ without the last element, so $A^i : \Pi\{x : A\}_i.\mathsf{U}$. Hence, we will also have functions with $3^k - 1$ number of arguments.

We will add rules to compute $(x : A)^I$ and $\{x : A\}_I$ in the next section.

Our function space also needs to be stable under substitution. For example, consider the type $(X : \mathsf{U})^i \vdash \Pi(x : X)^i.B : \mathsf{U}$. Here all the 3 types in the domain of the function are variables, hence they can be given arbitrary values by a substitution $\rho : \cdot \Rightarrow (X : \mathsf{U})^i$. Performing the substitution $(\Pi(x : X)^i.B)[\rho]$ results in $\Pi(x_{i0} : X_{i0}[\rho], x_{i1} : X_{i1}[\rho], x_{i2} : X_{i2}[\rho]\, x_{i0}\, x_{i1}).B[\rho]$ which does not have the form $\Pi(x : A)^i.B$ for some A anymore. This is

we decorate the domain of the function with a telescope substitution providing the types and A will be fixed to be a variable X. The formation rules:

$$\frac{\Gamma \vdash \xi : (X : \mathsf{U})^I \quad \Gamma +\!\!+ (x : X)^I[\xi] \vdash B : \mathsf{U}}{\Gamma \vdash \Pi(x : X)^I[\xi].B : \mathsf{U}} \qquad \frac{\Gamma \vdash \xi : \{X : \mathsf{U}\}_{Ii}}{\Gamma \vdash \Pi\{x : X\}_{Ii}[\xi].\mathsf{U} : \mathsf{U}}$$

These are just the usual rules for telescope functions, restricted on the domain (and for relations, in the codomain too). Note that the domain for relation space cannot be zero-dimensional. Also, we have not defined substituting a telescope context with a telescope substitution, so we mean $(x : X)^I[\mathsf{id} +\!\!+ \xi]$ when we write $(x : X)^I[\xi]$.

Most rules have the same shape for functions and relations, for these, we introduce the notation $⦃x : A⦄_I$ which can mean $(x : A)^I$ or $\{x : A\}_I$. We use the notation $\mathsf{app}^I(f, \omega)$ for function application, $\mathsf{app}_I(R, \omega)$ for relation application and $\mathsf{app}_I(f, \omega)$ can be either.

$$\frac{\Gamma +\!\!+ ⦃x : X⦄_I[\xi] \vdash t : B}{\Gamma \vdash \lambda⦃x : X⦄_I[\xi].t : \Pi⦃x : X⦄_I[\xi].B} \qquad \frac{\Gamma \vdash f : \Pi⦃x : X⦄_I[\xi].B \quad \Gamma \vdash \omega : ⦃y : X⦄_I[\xi]}{\Gamma \vdash \mathsf{app}_I(f, \omega) : B[\mathsf{id} +\!\!+ \omega]}$$

The computation rules include η:

$$\begin{aligned}
\mathsf{app}_I(\lambda⦃x⦄_I.t, \omega) &\equiv t[\mathsf{id} +\!\!+ \omega] \\
f &\equiv \lambda⦃x : X⦄_I[\xi].\mathsf{app}_I(f, \mathsf{pr}_{⦃x:X⦄_I[\xi]}) \\
(\Pi⦃x : X⦄_I[\xi].B)[\sigma] &\equiv \Pi⦃x : X⦄_I[\xi[\sigma]].B[\sigma +\!\!+ \mathsf{pr}_{⦃x:X⦄_I[\xi[\sigma]]}] \\
(\lambda⦃x : X⦄_I[\xi].t)[\sigma] &\equiv \lambda⦃x : X⦄_I[\xi[\sigma]].t[\sigma +\!\!+ \mathsf{pr}_{⦃x:X⦄_I[\xi[\sigma]]}] \\
\mathsf{app}_I(f, \omega)[\sigma] &\equiv \mathsf{app}_I(f[\sigma], \omega[\sigma])
\end{aligned}$$

If the telescope substitution ξ is the projection pr, we omit it, eg. we write $\Pi(x : X)^I.B$ for $\Pi(x : X)^I[\mathsf{pr}].B$. Also, we write $\Pi(x : A)^I.B$ for $\Pi(x : X)^I[(X \mapsto A)^I].B$ and $\Pi\{x : A\}_I.B$ for $\Pi\{x : X\}_I[\{X \mapsto A\}_I].B$.

Note that in the case when I is empty, $\Pi(x : A)^I.B$ becomes the usual function space, we denote it as $\Pi(x : A).B$.

3.3 The Operations $(\text{–})^i$ and $(\text{–})_i$ on Contexts and Substitutions

We define $(\text{–})^i$ and $\{\text{–}\}_i$ on contexts, telescope contexts, substitutions, telescope substitutions. The definitions for contexts are the same as in the naive version.

Specification:

$$\frac{\Gamma \vdash}{(\Gamma)^i \vdash} \qquad \frac{\rho : \Delta \Rightarrow \Gamma}{(\rho)^i : \Delta^i \Rightarrow \Gamma^i} \qquad \frac{\Gamma \vdash \Omega}{(\Gamma)^i \vdash \Omega^i} \qquad \frac{\Gamma \vdash \omega : \Omega}{(\Gamma)^i \vdash \omega^i : \Omega^i}$$

$$\frac{\Gamma.x : A \vdash}{\{\Gamma.x : A\}_i \vdash} \qquad \frac{\rho : \Delta \Rightarrow \Gamma.x : A}{\{\rho\}_i : \Delta^i \Rightarrow \{\Gamma.x : A\}_i} \qquad \frac{\Gamma \vdash \Omega.x : A}{\Gamma^i \vdash \{\Omega.x : A\}_i} \qquad \frac{\Gamma \vdash \omega : \Omega.x : A}{\Gamma^i \vdash \{\omega\}_i : \{\Omega.x : A\}_i}$$

Implementation:

$$
\begin{array}{lll}
(\cdot)^i & \equiv \cdot & \\
(\Gamma.x : A)^i & \equiv \Gamma^i + (x : A)^i & \\
\epsilon^i & \equiv \epsilon & : \Gamma^i \Rightarrow \cdot^i \\
(\rho, x \mapsto t)^i & \equiv (\rho)^i + (x \mapsto t)^i & : \Delta^i \Rightarrow (\Gamma.x : A)^i \\
\Gamma^i \vdash (\cdot)^i & \equiv \cdot & \\
\Gamma^i \vdash (\Omega.x : A)^i & \equiv \Omega^i.x_{i0} : A[0_{i\Gamma}].x_{i1} : A[1_{i\Gamma}].x_{i2} : \mathsf{app}_i(A^i, (x)_i) & \\
\Gamma^i \vdash \epsilon^i & \equiv \epsilon & : \cdot^i \\
\Gamma^i \vdash (\omega, x \mapsto t)^i & \equiv (\omega^i, x_{i0} \mapsto t[0_{i\Gamma}], x_{i1} \mapsto t[1_{i\Gamma}], x_{i2} \mapsto t^i) & : (\Omega.x : A)^i \\
\{\Gamma.x : A\}_i & \equiv \Gamma^i + \{x : A\}_i & \\
\{\rho, x \mapsto t\}_i & \equiv \rho^i + \{x \mapsto t\}_i & : \Delta^i \Rightarrow \{\Omega.x : A\}_i \\
\Gamma^i \vdash \{\Omega.x : A\}_i & \equiv \Omega^i.x_{i0} : A[0_{i\Gamma}].x_{i1} : A[1_{i\Gamma}] & \\
\Gamma^i \vdash \{\omega, x \mapsto t\}_i & \equiv (\omega^i, x_{i0} \mapsto t[0_{i\Gamma}], x_{i1} \mapsto t[1_{i\Gamma}]) & : \{\Omega.x : A\}_i
\end{array}
$$

The operation $\{-\}_i$ does the same as $(-)^i$ but omits the very last element (because of this, it does not work on empty contexts or substitutions).

The substitutions 0 and 1 project out the corresponding components ($b = 0, 1$) while losing the dimension i.

$$
\begin{array}{ll}
b_{i\cdot} \equiv \epsilon & : \cdot \Rightarrow \cdot \\
b_{i(\Gamma.x:A)} \equiv (b_{i\Gamma}, x \mapsto x_{ib}) & : (\Gamma.x : A)^i \Rightarrow \Gamma.x : A
\end{array}
$$

We also add how $(-)^i$ operates on special substitutions:

$$(\rho\sigma)^i \equiv \rho^i\sigma^i \qquad (\mathsf{id}_\Gamma)^i \equiv \mathsf{id}_{\Gamma^i}$$

$(-)^I$ is an iterated version of $(-)^i$. $\{-\}_{Ii}$ is just $(-)^I$ composed by $\{-\}_i$. For contexts we express this by the following rules, for the substitutions etc. we have analogous rules.

$$\Gamma^\emptyset \equiv \Gamma \qquad \Gamma^{Ij} \equiv (\Gamma^I)^j \qquad \{\Gamma\}_{Ij} \equiv \{\Gamma^I\}_j$$

An element of a (telescope) context produced by multiple applications of $(-)^i$ can be viewed as an $|I|$-dimensional cube.

3.4 $(-)^i$ on Terms

The operation which maps a term to the witness of parametricity is the same as in the naive version, but with the additional information added for the function space. We need to handle functions and relations separately because performing $(-)^i$ on a full cube gives a full cube, but on incomplete cubes it does not even give an incomplete cube. We need to fill in the omitted two elements in the latter case. Note that $(x : A)^I$ has $3^{|I|}$ elements and $\{x : A\}_I$ has $3^{|I|} - 1$ elements.

Specification:

$$\frac{\Gamma \vdash t : A}{\Gamma^i \vdash t^i : \mathsf{app}_i(A^i, \{x \mapsto t\}_i)} \tag{1}$$

Implementation:

$$
\begin{aligned}
(t[\rho])^i &\equiv t^i[\rho^i] \\
x^i &\equiv x_{i2} \\
\mathsf{U}^i &\equiv \lambda\{X\}_i.\Pi\{x : X\}_i.\mathsf{U} \\
(\Pi(x : X)^I[\xi].B)^i &\equiv \lambda\{f\}_i.\Pi(x : X)^{Ii}[\xi^i].\mathsf{app}_i(B^i, \{y \mapsto \mathsf{app}^I(f, (x)^I)\}_i) \\
(\lambda(x)^I.t)^i &\equiv \lambda(x)^{Ii}.t^i \\
(\mathsf{app}^I(f, \omega))^i &\equiv \mathsf{app}^{Ii}(t^i, \omega^i) \\
(\Pi\{x : X\}_I[\xi].\mathsf{U})^i &\equiv \lambda\{y\}_i.\Pi\{x : X\}_{Ii}[\xi^i + \{X_{I2} \mapsto y\}_i].\mathsf{U} \\
(\lambda\{x\}_I.B)^i &\equiv \lambda\{x\}_{Ii}.\mathsf{app}_i(B^i, \{y \mapsto x_{I2}\}_i) \\
(\mathsf{app}_I(R, \omega))^i &\equiv \lambda\{y\}_i.\mathsf{app}_{Ij}(R^i, \omega^i + \{x_{I2} \mapsto y\}_i)
\end{aligned}
$$

x_{I2} is a notation for the variable x where all the dimensions have index 2, eg. x_{ijk2} is x_{i2j2k2}.

3.5 A Definitional Quotient

Now we can add rules equating $(\text{–})^{ij}$ and $(\text{–})^{ji}$.

$$
\frac{\Gamma \vdash}{(\Gamma^i)^j \equiv (\Gamma^j)^i} \qquad \frac{\Gamma \vdash \Omega}{\Gamma^{ij} \vdash (\Omega^i)^j \equiv (\Omega^j)^i} \qquad \frac{\Gamma \vdash \omega : \Omega}{\Gamma^{ij} \vdash (\omega^i)^j \equiv (\omega^j)^i : \Omega^{ij}}
$$

$$
\frac{\rho : \Delta \Rightarrow \Gamma}{(\rho^i)^j \equiv (\rho^j)^i : \Delta^{ij} \Rightarrow \Gamma^{ij}} \qquad \frac{\Gamma \vdash t : A}{\Gamma^{ij} \vdash (t^i)^j \equiv (t^j)^i : \mathsf{app}_{ij}(A^{ij}, \{x \mapsto t\}_{ij})}
$$

$$
\frac{\Gamma \vdash}{\{\Gamma^i\}_j \equiv \{\Gamma^j\}_i} \qquad \frac{\Gamma \vdash \Omega}{\Gamma^{ij} \vdash \{\Omega^i\}_j \equiv \{\Omega^j\}_i} \qquad \frac{\Gamma \vdash \omega : \Omega}{\Gamma^{ij} \vdash \{\omega^i\}_j \equiv \{\omega^j\}_i : \{\Omega\}_{ij}}
$$

$$
\frac{\rho : \Delta \Rightarrow \Gamma}{\{\rho^i\}_j \equiv \{\rho^j\}_i : \Delta^{ij} \Rightarrow \{\Gamma\}_{ij}}
$$

So we can treat the dimensions of a context, substitution etc. as a set and write them without parentheses.

Now we have $(b_{i\Gamma})^j \equiv b_{i\Gamma^j}$ for $b = 0, 1$.

3.6 Internalisation of Parametricity

We add a substitution R that adds a dimension to a context.

$$
\frac{\Gamma \vdash}{\mathsf{R}_{i\Gamma} : \Gamma \Rightarrow \Gamma^i} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash t^{ⓘ} : \mathsf{app}_i(A^i[\mathsf{R}_{i\Gamma}], (x_{i0} \mapsto t, x_{i1} \mapsto t))}
$$

$$
\begin{aligned}
&\mathsf{R}_{i\cdot} \equiv \epsilon && : \cdot \Rightarrow \cdot \\
&\mathsf{R}_{i(\Gamma.x:A)} \equiv (\mathsf{R}_{i\Gamma}, x_{i0} \mapsto x, x_{i1} \mapsto x, x_{i2} \mapsto x^{ⓘ}) && : (\Gamma.x : A) \Rightarrow (\Gamma.x : A)^i
\end{aligned}
$$

$$
t^{ⓘ} \equiv t^i[\mathsf{R}_{i\Gamma}] \qquad \frac{\rho : \Delta \to \Gamma}{\mathsf{R}_{i\Gamma}\rho \equiv \rho^i \mathsf{R}_{i\Delta}}
$$

Note that we can derive $a^{ⓘ}[\rho] \equiv (a[\rho])^{ⓘ}$ from the above rules. Also, the computation rule does not give us anything new if a is a variable:

$$
\Gamma.x : A \vdash x^{ⓘ} \equiv x^i[\mathsf{R}_{i(\Gamma.x:A)}] \equiv x_{i2}[\mathsf{R}_{i(\Gamma.x:A)}] \equiv x^{ⓘ}.
$$

We finally add the rule describing how $(-)^i$ works on the substitution R and how it interacts with the projections $b = 0, 1$:

$$(\mathsf{R}_{j\Gamma})^i \equiv \mathsf{R}_{j\Gamma^i} \qquad b_{i\Gamma}\mathsf{R}_{i\Gamma} \equiv \mathsf{id}_\Gamma$$

The first rule needs the definitional quotient to typecheck.

With this rule, we defined a type theory with internal parametricity. We describe an operational semantics in appendix A.

4 Conclusions and Further Work

We defined a naive version of cubical type theory in which univalence is admissible however we do not yet know how to compute in this theory. An advantage of this theory is that it is close to the usual syntax of Martin-Löf's type theory, hence it might be a step towards a translation from cubical type theory into intensional type theory. We conjecture that it has a presheaf model in a univalent metatheory (as some equations are only validated up to isomorphism – note that this problem does not appear for Cohen et al's cubical type theory [11], however it does appear in [7]).

We also defined a nominal version of the theory for parametricity together with a big-step normalisation function. We still have to establish that this normalisation function is sound and complete following [1]. However, as far as we know the computational behaviour of the operational semantics suggested in [11] has not been fully investigated yet, either.

In our approach the cubical structure arises naturally by iterating the $-^=$ and $-^i$ operators and it is not a consequence of extending the theory with a pretype for the interval as in [11, 4]. One interesting difference is that we directly define a family of equality types while the approach based on the interval type defines the family by fixing the endpoints of functions on the interval. On the other hand the notational overhead in the interval type based approach seems to be less than in our approach. Introducing an univalent universe is hard and is the most difficult aspect in [11], hence it is maybe unsurprising that we have not achieved this yet in this more low level approach.

While in the moment the interval based approach presented in [11] seems to offer many advantages it may be a good idea to consider alternative approaches as the ones suggested here. We may also wonder whether we should need to explain an interval type when we want to introduce type theory?

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1 Thorsten Altenkirch and James Chapman. Big-step normalisation. *J. Funct. Program.*, 19(3-4):311–333, 2009. doi:10.1017/S0956796809007278.

2 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016. doi:10.1145/2837614.2837638.

3 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming*

Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007, pages 57–68. ACM, 2007. `doi:10.1145/1292597.1292608`.

4 C. Angiuli, R. Harper, and T. Wilson. Computational higher-dimensional type theory. In *Proc. of 44th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2017*, pages 680–693. ACM, 2017. `doi:10.1145/3009837.3009861`.

5 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014. `doi:10.1145/2535838.2535852`.

6 J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proc. of 27th Ann. IEEE Symp. on Logic in Computer Science, LICS '12*, pages 135–144. IEEE, 2012. `doi:10.1109/lics.2012.25`.

7 Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electr. Notes Theor. Comput. Sci.*, 319:67–82, 2015. `doi:10.1016/j.entcs.2015.12.006`.

8 Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 61–72. ACM, 2013. `doi:10.1145/2500365.2500577`.

9 M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In R. Matthes and A. Schubert, editors, *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES 2013*, volume 26 of *Leibniz Int. Proc. in Inform.*, pages 107–128. Dagstuhl Publishing, 2014. `doi:10.4230/lipics.types.2013.107`.

10 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985. `doi:10.1145/6041.6042`.

11 C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. In T. Uustalu, editor, *Proc. of 21st Int. Conf. on Types for Proofs and Programs, TYPES 2015*, volume 69 of *Leibniz Int. Proc. in Inform.* Dagstuhl Publishing, 2017.

12 T. Coquand. Variation on cubical sets. Note, 2014. URL: `http://www.cse.chalmers.se/~coquand/comp.pdf`.

13 Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity - A Reynolds programme for category theory and programming languages. *Electr. Notes Theor. Comput. Sci.*, 303:149–180, 2014. `doi:10.1016/j.entcs.2014.02.008`.

14 S. Huber. *A Model of Type Theory in Cubical Sets.* Licentiate thesis, Chalmers University of Technology, 2015. URL: `http://www.cse.chalmers.se/~simonhu/misc/lic.pdf`.

15 G. Moulin. *Pure Type Systems with an Internalized Parametricity Theorem.* Licentiate thesis, Chalmers University of Technology, 2013. URL: `http://publications.lib.chalmers.se/publication/191873`.

16 G. Moulin. *Internalizing Parametricity.* PhD thesis, Chalmers University of Technology, 2016. URL: `http://publications.lib.chalmers.se/publication/235758`.

17 A. Polonsky. Extensionality of lambda-*. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *Proc. of 20th Int. Conf. on Types for Proofs and Programs, TYPES 2014*, volume 39 of *Leibniz Int. Proc. in Inform.*, pages 221–250. Dagstuhl Publishing, 2015. `doi:10.4230/lipics.types.2014.221`.

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

A An Operational Semantics for the Nominal Calculus

A.1 Evaluation

We describe a call by name operational semantics for the theory defined in section 3. We conjecture that the semantics defined below is terminating, and when viewed as a syntactic model construction, sound and complete (we refer to [1] for the meaning of these properties); decidability of definitional would equality.

We define values (weak-head normal forms), and show how to evaluate terms into values. Next we define normal forms, and show that by recursively applying evaluation, we can normalise terms.

In what follows, $\{\!\!\{-\}\!\!\}_I$ can mean both $(-)^I$ and $\{-\}_I$. Note that in the latter case I is nonempty. We define the following inductive sets by listing their constructors.

t, A	$::= t[\rho] \mid x \mid \mathsf{U} \mid \Pi(x : X)^I[\omega].A \mid \Pi\{x : X\}_I[\omega].\mathsf{U}$	
	$\mid \lambda\{\!\!\{x\}\!\!\}_I.t \mid \mathsf{app}_I(t, \omega) \mid t^i \mid t^{ⓘ}$	terms
ρ	$::= \mathsf{id} \mid \epsilon \mid (\rho, x \mapsto t) \mid \rho\rho' \mid \rho +\!\!+ \omega \mid \mathsf{0}_i \mid \mathsf{1}_i \mid \mathsf{R}_i \mid \rho^i \mid \{\rho\}_i$	substitutions
ω	$::= \epsilon \mid (\omega, x \mapsto t) \mid \omega[\rho] \mid \omega +\!\!+ \omega' \mid \mathsf{pr}_\Omega \mid \omega^i \mid \{\omega\}_i$	telescope substitutions
x, f, X	$::= \ldots$	variables
i	$::= \ldots$	dimension names
I	$::= \emptyset \mid Ii$	sets of dimension names
v	$::= \mathsf{U} \mid (\Pi\{\!\!\{x : X\}\!\!\}_I[\omega].A)[\nu] \mid (\lambda\{\!\!\{x\}\!\!\}_I.t)[\nu] \mid n$	values (whnfs)
ν	$::= \epsilon \mid (\nu, x \mapsto g) \mid (\nu, x \mapsto t[\nu])$	environments
n	$::= g \mid \mathsf{app}_I(n, \psi)$	neutral values
g	$::= x \mid g^{ⓘ}$	generic values
ψ	$::= \epsilon \mid (\psi, x \mapsto g) \mid (\psi, x \mapsto t[\nu])$	telescope environments
Ω	$::= \cdot \mid \Omega.x : A \mid \Omega[\rho] \mid \Omega +\!\!+ \Omega' \mid \Omega^i \mid \{\Omega\}_i$	telescope contexts
Ξ	$::= \cdot \mid \Xi.x : A$	linear telescope contexts

We have the following judgment types for evaluation.

$t[\nu] \Downarrow v$	evaluate a closure to a value
$\rho\nu \Downarrow \nu'$	evaluate a substitution closure to an environment
$\omega[\nu] \Downarrow \psi$	evaluate a telescope substitution closure into a telescope environment
$\Omega \Downarrow \Xi$	evaluate a telescope context into a linear telescope context
$\nu(x) \rightsquigarrow v$	look up the value of a variable in an environment
$t^i \rightsquigarrow t'$	one step in calculating the meaning of a lifted term
$\rho^i \rightsquigarrow \rho'$	one step in calculating the meaning of a lifted substitution
$\omega^i \rightsquigarrow \omega'$	one step in calculating the meaning of a lifted telescope substitution
$\nu +\!\!+ \psi \rightsquigarrow \nu'$	composition of environments and telescope environments
$\psi +\!\!+ \psi' \rightsquigarrow \psi''$	composition of telescope environments
$\mathsf{pr}_\Xi[\nu] \rightsquigarrow \psi$	evaluating a telescope projection

We give a case for each judgment type for each constructor. Evaluation of terms is standard, the interesting cases are for t^i and $t^{ⓘ}$: for the former we use the one-step evaluation relation $t^i \rightsquigarrow t'$, for the latter we use the computation rule for $-^{ⓘ}$.

$\boxed{t[\nu] \Downarrow v}$

$$\frac{\rho\nu \Downarrow \nu' \quad t[\nu'] \Downarrow v}{t[\rho][\nu] \Downarrow v} \qquad \frac{\nu(x) \rightsquigarrow v}{x[\nu] \Downarrow v} \qquad \mathsf{U}[\nu] \Downarrow \mathsf{U} \qquad (\Pi\{x : X\}_I[\omega].A)[\nu] \Downarrow (\Pi\{x : X\}_I[\omega].A)[\nu]$$

$$(\lambda\{x\}_I.t)[\nu] \Downarrow (\lambda\{x\}_I.t)[\nu] \qquad \frac{t[\nu] \Downarrow (\lambda(x)_I.t')[\nu'] \quad \omega[\nu] \Downarrow \psi \quad \nu' + \psi \rightsquigarrow \nu'' \quad t'[\nu''] \Downarrow v}{\mathsf{app}_I(t,\omega)[\nu] \Downarrow v}$$

$$\frac{t[\nu] \Downarrow n \quad \omega[\nu] \Downarrow \psi}{\mathsf{app}_I(t,\omega)[\nu] \Downarrow \mathsf{app}_I(n,\psi)} \qquad \frac{t^i \rightsquigarrow t' \quad t'[\nu] \Downarrow v}{t^i[\nu] \Downarrow v} \qquad \frac{t^i[\mathsf{R}_i][\nu] \Downarrow v}{t^{ⓘ}[\nu] \Downarrow v}$$

For substitutions, the interesting cases are again for our special constructions: the substitutions $0_i, 1_i$ (denoted b_i) project out the corresponding components, while the substitution R_i duplicates the content of the environment. Its substitution rule ensures that we can move the $-^{ⓘ}$ through the environment ν'.

$\boxed{\rho\nu \Downarrow \nu'}$

$$\mathsf{id}\nu \Downarrow \nu \qquad \epsilon\nu \Downarrow \epsilon \qquad \frac{\rho\nu \Downarrow \nu'}{(\rho, x \mapsto t)\nu \Downarrow (\nu', x \mapsto t[\nu])} \qquad \frac{\rho'\nu \Downarrow \nu' \quad \rho\nu' \Downarrow \nu''}{(\rho\rho')\nu \Downarrow \nu''}$$

$$\frac{\rho\nu \Downarrow \nu' \quad \omega[\nu] \Downarrow \psi \quad \nu' + \psi \rightsquigarrow \nu''}{(\rho + \omega)\nu \Downarrow \nu''}$$

$$b_i\epsilon \Downarrow \epsilon \qquad \frac{b_i\nu \Downarrow \nu'}{b_i(\nu, x_{ib} \mapsto t) \Downarrow (\nu', x \mapsto t)} \qquad \frac{b_i\nu \Downarrow \nu'}{b_i(\nu, x_{ic} \mapsto t) \Downarrow \nu'}\ c\,\mathsf{not}{=}b$$

$$\mathsf{R}_i\epsilon \Downarrow \epsilon \qquad \frac{\mathsf{R}_i\nu \Downarrow \nu'}{\mathsf{R}_i(\nu, x \mapsto g) \Downarrow (\nu', x_{i0} \mapsto g, x_{i1} \mapsto g, x_{i2} \mapsto g^{ⓘ})}$$

$$\frac{\mathsf{R}_i\nu \Downarrow \nu''}{\mathsf{R}_i(\nu, x \mapsto t[\nu']) \Downarrow (\nu'', x_{i0} \mapsto t[\nu'], x_{i1} \mapsto t[\nu'], x_{i2} \mapsto t^{ⓘ}[\nu'])}$$

$$\frac{\rho^i \rightsquigarrow \rho' \quad \rho'\nu \Downarrow \nu'}{(\rho)^i\nu \Downarrow \nu'} \qquad \frac{\rho^i \rightsquigarrow \rho' \quad \rho'\nu \Downarrow (\nu', x \mapsto t)}{\{\rho\}_i\nu \Downarrow \nu'}$$

$\boxed{\omega[\nu] \Downarrow \psi}$

$$\epsilon[\nu] \Downarrow \epsilon \qquad \frac{\omega[\nu] \Downarrow \psi}{(\omega, x \mapsto t)[\nu] \Downarrow (\psi, x \mapsto t[\nu])} \qquad \frac{\rho\nu \Downarrow \nu' \quad \omega[\nu'] \Downarrow \psi}{\omega[\rho][\nu] \Downarrow \psi}$$

$$\frac{\omega[\nu] \Downarrow \psi \quad \omega'[\nu] \Downarrow \psi \quad \psi + \psi' \rightsquigarrow \psi''}{(\omega + \omega')[\nu] \Downarrow \psi''} \qquad \frac{\Omega \Downarrow \Xi \quad \mathsf{pr}_\Xi \rightsquigarrow \psi}{\mathsf{pr}_\Omega[\nu] \Downarrow \psi} \qquad \frac{\omega^i \rightsquigarrow \omega' \quad \omega'[\nu] \Downarrow \psi}{\omega^i[\nu] \Downarrow \psi}$$

$$\frac{\omega^i \rightsquigarrow \omega' \quad \omega'[\nu] \Downarrow (\psi, x \mapsto t)}{\{\omega\}_i[\nu] \Downarrow \psi}$$

$\boxed{\Omega \Downarrow \Xi}$

$$\cdot \Downarrow \cdot \qquad \frac{\Omega \Downarrow \Xi}{\Omega.x : A \Downarrow \Xi.x : A} \qquad \frac{\Omega \Downarrow \cdot}{\Omega[\rho] \Downarrow \cdot} \qquad \frac{\Omega \Downarrow \Xi.x : A \quad \Xi[\rho] \Downarrow \Xi'}{\Omega[\rho] \Downarrow \Xi'.x : A[\rho]}$$

$$\frac{\Omega \Downarrow \cdot \quad \Omega' \Downarrow \Xi}{\Omega + \Omega' \Downarrow \Xi} \qquad \frac{\Omega' \Downarrow (\Xi.x : A) \quad \Omega + \Xi \Downarrow \Xi'}{\Omega + \Omega' \Downarrow \Xi'.x : A} \qquad \frac{\Omega \Downarrow \cdot}{\Omega^i \Downarrow \cdot}$$

$$\frac{\Omega \Downarrow \Xi.x : A \qquad \Xi^i \Downarrow \Xi'}{\Omega^i \Downarrow \Xi'.x_{i0} : A[0_i].x_{i1} : A[1_i].x_{i2} : \mathsf{app}_i(A^i, (x \mapsto x)_i)} \qquad \frac{\Omega \Downarrow \Xi.x : A \qquad \Xi^i \Downarrow \Xi'}{\{\Omega\}_i \Downarrow \Xi'.x_{i0} : A[0_i].x_{i1} : A[1_i]}$$

$\boxed{\nu(x) \rightsquigarrow v}$

$$\frac{\nu(x) = g}{\nu(x) \rightsquigarrow g} \qquad \frac{\nu(x) = t[\nu'] \quad t[\nu'] \Downarrow v}{\nu(x) \rightsquigarrow v}$$

The one-step part of the evaluation is given by following the rules in section 3.4.

$\boxed{t^i \rightsquigarrow t'}$

$$\begin{aligned}
(t[\rho])^i &\rightsquigarrow t^i[\rho^i] \\
x^i &\rightsquigarrow x_{i2} \\
\mathsf{U}^i &\rightsquigarrow \lambda\{X\}_i.\Pi(x : X)_i[\{X \mapsto X\}_i].\mathsf{U} \\
(\Pi(x : X)^I[\omega].A)^i &\rightsquigarrow \lambda\{f\}_i.\Pi(x : X)^{Ii}[\omega^i].\mathsf{app}_i(A^i, \{x' \mapsto \mathsf{app}^I(f, (x'' \mapsto x)^I)\}_i) \\
(\lambda(x)^I.t)^i &\rightsquigarrow \lambda(x)^{Ii}.t^i \\
\mathsf{app}^I(t, \omega)^i &\rightsquigarrow \mathsf{app}^{Ii}(t^i, \omega^i) \\
(\Pi\{x : X\}_I[\omega].\mathsf{U})^i &\rightsquigarrow \lambda(X')_i.\Pi\{x : X\}_{Ii}[(\omega^i, \{X_{I2} \mapsto X'\}_i)].\mathsf{U} \\
(\lambda\{x\}_I.A)^i &\rightsquigarrow \lambda\{x\}_{Ii}.\mathsf{app}_i(A^i, \{x' \mapsto x_{I2}\}_i) \\
\mathsf{app}_I(t, \omega)^i &\rightsquigarrow \lambda\{x'\}_i.\mathsf{app}_{Ii}(t^i, \{\omega^i, (x_{I2} \mapsto x'\}_i))
\end{aligned}$$

$$\frac{t^j \rightsquigarrow t'}{(t^j)^i \rightsquigarrow t'^i} \qquad (t^{\textcircled{j}})^i \rightsquigarrow (t^j[\mathsf{R}_j])^i$$

The definition of $\rho^i \rightsquigarrow \rho'$ for substitutions follows a similar pattern, it uses the functor laws for substitutions given in section 3.3. $\omega^i \rightsquigarrow \omega'$ describes one-step evaluation of telescope substitutions.

$\boxed{\rho^i \rightsquigarrow \rho'}$

$$\begin{aligned}
\mathsf{id}^i &\rightsquigarrow \mathsf{id} \\
\epsilon^i &\rightsquigarrow \epsilon \\
(\rho, x \mapsto t)^i &\rightsquigarrow (\rho^i, x_{i0} \mapsto t[0_i], x_{i1} \mapsto t[1_i], x_{i2} \mapsto t^i) \\
(\rho\rho')^i &\rightsquigarrow \rho^i\rho'^i \\
(\rho + \omega)^i &\rightsquigarrow (\rho^i + \omega^i) \\
(b_j)^i &\rightsquigarrow b_j \\
(\mathsf{R}_j)^i &\rightsquigarrow \mathsf{R}_j
\end{aligned}$$

$$\frac{\rho^j \rightsquigarrow \rho'}{(\{\!|\rho|\!\}_j)^i \rightsquigarrow \rho'^i}$$

$\boxed{\omega^i \rightsquigarrow \omega'}$

$$\begin{aligned}
\epsilon^i &\rightsquigarrow \epsilon \\
(\omega, x \mapsto t)^i &\rightsquigarrow (\omega^i, x_{i0} \mapsto t[0_i], x_{i1} \mapsto t[1_i], x_{i2} \mapsto t^i) \\
(\omega[\rho])^i &\rightsquigarrow \omega^i[\rho^i] \\
(\omega + \omega')^i &\rightsquigarrow (\omega^i + \omega'^i)
\end{aligned}$$

$$\frac{\omega^j \rightsquigarrow \omega'}{(\{\!\!\{\omega\}\!\!\}_j)^i \rightsquigarrow \omega'^i}$$

$\boxed{\nu +\!\!+ \psi \rightsquigarrow \nu'}$

$$\nu +\!\!+ \epsilon \rightsquigarrow \nu \qquad \frac{\nu +\!\!+ \psi \rightsquigarrow \nu'}{\nu +\!\!+ (\psi, x \mapsto t) \rightsquigarrow (\nu', x \mapsto t)}$$

$\boxed{\psi +\!\!+ \psi' \rightsquigarrow \psi''}$

$$\psi +\!\!+ \epsilon \rightsquigarrow \psi \qquad \frac{\psi +\!\!+ \psi' \rightsquigarrow \psi''}{\psi +\!\!+ (\psi', x \mapsto t) \rightsquigarrow (\psi'', x \mapsto t)}$$

$\boxed{\mathsf{pr}_\Xi[\nu] \rightsquigarrow \psi}$

$$\mathsf{pr}_\cdot[\nu] \Downarrow \epsilon \qquad \frac{\mathsf{pr}_\Xi[\nu] \Downarrow \psi}{\mathsf{pr}_{\Xi.x:A}[(\nu, x \mapsto t)] \Downarrow (\psi, x \mapsto t)}$$

A.2 Normalisation

$$\begin{array}{lll} v_n & ::= \mathsf{U} \mid \Pi\{\!\!\{x : X\}\!\!\}_I[\psi_n].v_n \mid \lambda\{\!\!\{x\}\!\!\}_I.v_n \mid n_n & \text{normal forms} \\ n_n & ::= g \mid \mathsf{app}_I(n, \psi_n) & \text{neutral normal forms} \\ \psi_n & ::= \epsilon \mid (\psi_n, x \mapsto v_n) & \text{normal telescopes} \end{array}$$

$\boxed{v \Rrightarrow v_n}$

$$\mathsf{U} \Rrightarrow \mathsf{U}$$

$$\frac{\{\!\!\{x : X\}\!\!\}_I[\mathsf{id} +\!\!+ \omega] \Downarrow \Xi \quad \mathsf{pr}_\Xi \rightsquigarrow \psi \quad \nu +\!\!+ \psi \rightsquigarrow \nu' \quad A[\nu'] \Downarrow v \quad v \Rrightarrow v_n \quad \omega[\nu] \Downarrow \psi' \quad \psi' \Rrightarrow \psi'_n}{(\Pi\{\!\!\{x : X\}\!\!\}_I[\omega].A)[\nu] \Rrightarrow \Pi\{\!\!\{x : X\}\!\!\}_I[\psi'_n].v_n}$$

$$\frac{\{\!\!\{x : X\}\!\!\}_I[\mathsf{id} +\!\!+ \omega] \Downarrow \Xi \quad \mathsf{pr}_\Xi \rightsquigarrow \psi \quad \nu +\!\!+ \psi \rightsquigarrow \nu' \quad t[\nu'] \Downarrow v \quad v \Rrightarrow v_n}{(\lambda\{\!\!\{x\}\!\!\}_I.t)[\nu] \Rrightarrow \lambda\{\!\!\{x\}\!\!\}_I.v_n} \qquad \frac{n \Rrightarrow n_n}{n \Rrightarrow n_n}$$

$\boxed{n \Rrightarrow n_n}$

$$g \Rrightarrow g \qquad \frac{n \Rrightarrow n_n \quad \psi \Rrightarrow \psi_n}{\mathsf{app}_I(n, \psi) \Rrightarrow \mathsf{app}_I(n_n, \psi_n)}$$

$\boxed{\psi \Rrightarrow \psi_n}$

$$\epsilon \Rrightarrow \epsilon \qquad \frac{\psi \Rrightarrow \psi_n}{(\psi, x \mapsto g) \Rrightarrow (\psi_n, x \mapsto g)} \qquad \frac{\psi \Rrightarrow \psi_n \quad t[\nu] \Downarrow v \quad v \Rrightarrow v_n}{(\psi, x \mapsto t[\nu]) \Rrightarrow (\psi_n, x \mapsto v_n)}$$

To generate the identity substitution, we need to linearize contexts, it can be done in the same way as for telescope contexts before ($\Omega \Downarrow \Xi$). We denote this by $\Gamma \Downarrow \Delta$.

$\boxed{\mathsf{id}_\Delta \Downarrow \nu}$

$$\mathsf{id}_\cdot \Downarrow \epsilon \qquad \frac{\mathsf{id}_\Delta \Downarrow \nu}{\mathsf{id}_{\Delta.x:t} \Downarrow (\nu, x \mapsto x)}$$

Normalisation can be performed as follows:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \Downarrow \Delta \qquad \mathsf{id}_\Delta \Downarrow \nu \qquad t[\nu] \Downarrow v \qquad v \Rrightarrow v_n}{t \text{ normalises to } v_n}$$

Constrained Polymorphic Types for a Calculus with Name Variables*

Davide Ancona[1], Paola Giannini[†2], and Elena Zucca[3]

1 **DIBRIS, Università di Genova, via Dodecaneso 35, Genova, Italy**
`davide.ancona@unige.it`
2 **DISIT, Università del Piemonte Orientale,**
via Teresa Michel 11, Alessandria, Italy
`paola.giannini@uniupo.it`
3 **DIBRIS, Università di Genova, via Dodecaneso 35, Genova, Italy**
`elena.zucca@unige.it`

Abstract

We extend the simply-typed lambda-calculus with a mechanism for dynamic rebinding of code based on parametric nominal interfaces. That is, we introduce values which represent single fragments, or families of named fragments, of open code, where free variables are associated with *names* which do not obey α-equivalence. In this way, code fragments can be passed as function arguments and manipulated, through their nominal interface, by operators such as rebinding, overriding and renaming. Moreover, by using *name variables*, it is possible to write terms which are parametric in their nominal interface and/or in the way it is adapted, greatly enhancing expressivity. However, in order to prevent conflicts when instantiating name variables, the name-polymorphic types of such terms need to be equipped with simple inequality constraints. We show soundness of the type system.

1998 ACM Subject Classification D.3.1 Programming Languages: Formal Definitions and Theory, D.3.3 Programming Languages: Language Constructs and Features

Keywords and phrases open code, incremental rebinding, name polymorphism, metaprogramming

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.4

1 Introduction

We propose an extension of the simply-typed lambda-calculus with a mechanism for dynamic and incremental rebinding of code based on parametric nominal interfaces. That is, we introduce values which represent single fragments, or families of named fragments, of open code, where free variables are associated with *names* which do not obey α-equivalence. Moreover, by using *name variables*, it is possible to write terms which are parametric in their nominal interface and/or in the way it is adapted, greatly enhancing expressivity. For instance, it is possible to write a term which corresponds to the selection of an arbitrary component of a module. We summarize here below the language features.

* This work has been partially supported by MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.
† This original research has the financial support of the Università del Piemonte Orientale.

© Davide Ancona, Paola Giannini and Elena Zucca;
licensed under Creative Commons License CC-BY
21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 4; pp. 4:1–4:29

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

- *Unbound terms*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid t\rangle$, and *rebindings*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid Y_n \mapsto t_1, \ldots, Y_n \mapsto t_n\rangle$, are values representing a single fragment, and a family of named fragments, respectively, of *open* code. That is, t, $t_1, \ldots, t_n$, may contain free occurrences of variables $x_1, \ldots, x_m$ to be dynamically bound through the global nominal interface $X_1, \ldots, X_m$. Unbound terms can be "unboxed" and executed through the *run* operator only after their open code has been completed through one or more applications of rebindings, so that they do not contain unbound variables; for instance, the unbound term $\langle x \mapsto X \mid x\texttt{+}1\rangle$ can be made self-contained with the rebinding $\langle \mid X \mapsto 0, Z \mapsto 1\rangle$.
- Rebinding application is *incremental*, that is, an unbound term can be partially rebound, leading to still open code. For instance, the term $\langle x \mapsto X, y \mapsto Y \mid x\texttt{+}y\rangle$ can be combined with the rebinding $\langle \mid X \mapsto 1, Z \mapsto 2\rangle$, getting $\langle y \mapsto Y \mid 1\texttt{+}y\rangle$. This allows code specialization, similarly to what partial application achieves for positional binding.
- Moreover, rebindings can be open, hence rebinding application is incremental even in the sense that it can introduce new variables to be dynamically bound. For instance, the term $\langle y \mapsto Y \mid 1\texttt{+}y\rangle$ can be combined with the rebinding $\langle x \mapsto X, z \mapsto Z \mid Y \mapsto x\texttt{+}z\rangle$, getting $\langle x \mapsto X, z \mapsto Z \mid 1\texttt{+}x\texttt{+}z\rangle$. As illustrated in [1, 2], this allows a form of generative programming.
- Finally, rebindings are first-class values, and can be manipulated by operators such as overriding and renaming.

A *name* X can be either a *name constant* N or a name variable α, and *name abstraction* $\Lambda\alpha.t$ and *name application* $t\ X$ can be used analogously to lambda-abstraction and application to define and instantiate name-parametric terms.

The types of such name-parametric terms need, correspondingly, to be polymorphic on names, and, moreover, must be equipped with simple inequality constraints. Formally, we get *constrained name-polymorphic types* of shape $\forall\alpha{:}c.T$, where c is a set of constraints of shape $X{\neq}Y$ among names. Such constraints are necessary to guarantee that for each possible instantiation of α we get well-formed terms and types. For instance, the term $\Lambda\alpha{:}\alpha \neq N.\langle \mid N{:}\texttt{int} \mapsto 0, \alpha{:}\texttt{int} \mapsto 1\rangle$ is a rebinding parametric in the name of one of its two components, which, however, must be different from the constant name N of the other component.

We refer to our previous work [1, 2], where we presented a monomorphic version with no name variables, for more examples and details on the features which are not novel here. This paper is a revised and improved version of [4]. Notably, we have added an algorithmic presentation of the type system, specified in full the notion of well-formedness and provided detailed proofs of the type soundness results.

The rest of the paper is structured as follows. In Section 2 we provide the formal definition of an untyped version of the calculus and an example motivating the introduction of name polymorphism. We then define a typed version of the calculus in Section 3, and a complete example of typing in Section 4. In Section 5 we prove the relevant results about the type system. We show the algorithmic presentation of the type system in Section 6, and finally in the Conclusion we discuss related and future work.

2 Untyped Calculus

The syntax and reduction rules of the untyped calculus are given in Figure 1, where we leave unspecified constructs of primitive types such as integers, which we will use in the examples.

t	$::=$	$\ldots \mid v \mid x$	term
		$\mid t_1\ t_2$	application
		$\mid t\ X$	name application
		$\mid t_1 \succ t_2$	rebinding operator
		$\mid\ !t$	run
		$\mid t_1 \lhd t_2$	overriding
		$\mid \sigma_1 \ltimes t \rtimes \sigma_2$	renaming operator
u	$::=$	$x_1 \mapsto X_1, \ldots, x_m \mapsto X_m$	unbinding map
r	$::=$	$X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$	rebinding map
σ	$::=$	$X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m$	renaming
X, Y	$::=$	$N \mid \alpha$	name
v	$::=$	$\ldots \mid \lambda x.t \mid \langle u \mid t \rangle \mid \langle u \mid r \rangle \mid \Lambda\alpha.t$	value
$\mathcal{E}$	$::=$	$[\,] \mid \ldots \mid \mathcal{E}\ t \mid v\ \mathcal{E} \mid \mathcal{E}\ X \mid \mathcal{E} \succ t \mid v \succ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} \lhd t$	evaluation context
		$\mid v \lhd \mathcal{E} \mid \sigma_1 \ltimes \mathcal{E} \rtimes \sigma_2$	
s	$::=$	$x_1 \mapsto t_1, \ldots, x_m \mapsto t_m$	substitution

$$(\textsc{Ctx})\ \frac{t \longrightarrow t'}{\mathcal{E}[t] \longrightarrow \mathcal{E}[t']} \qquad (\textsc{App})\ \frac{}{(\lambda x.t)\ v \longrightarrow t\{x \mapsto v\}}$$

$$(\textsc{Name-App})\ \frac{}{(\Lambda\alpha.t)\ N \longrightarrow t\{\alpha \mapsto N\}}$$

$$(\textsc{Reb-App})\ \frac{}{\langle u | r \rangle \succ \langle u_1, u_2 | t \rangle \longrightarrow \langle u, u_2 | t\{x \mapsto r(u_1(x)) | x \in dom(u_1)\} \rangle}\quad rng(u_2) \cap dom(r) = \emptyset$$

$$(\textsc{Run})\ \frac{}{!\langle\ \mid t \rangle \longrightarrow t} \qquad (\textsc{Over})\ \frac{}{\langle u_1 \mid r_1 \rangle \lhd \langle u_2 \mid r_2 \rangle \longrightarrow \langle u_1, u_2 \mid r_1[r_2] \rangle}$$

$$(\textsc{Rename})\ \frac{}{\sigma_1 \ltimes \langle u \mid r \rangle \rtimes \sigma_2 \longrightarrow \langle \sigma_1 \circ u \mid r \circ \sigma_2 \rangle}$$

Figure 1 Untyped calculus: syntax and reduction rules

We assume infinite sets of *variables* x, *name constants* N and *name variables* α. We use X, Y to range over *names* which are either name constants or name variables.

We use various kinds of (sequences which represent) finite maps: *unbinding maps* u from variables to names, *rebinding maps* r from names to terms, *renamings* σ from names to names, and *substitutions* s from variables to terms. Order and repetitions are immaterial in such sequences. Moreover, in well-formed terms, they are assumed to be actually maps, that is, e.g., given a rebinding, $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$, if $X_i = X_j$ then $t_i = t_j$. Hence, we can use the following notations: *dom* and *rng* for the domain and range, respectively, $r_1 \circ r_2$ for map composition, assuming $rng(r_2) \subseteq dom(r_1)$, r_1, r_2 for the union of two maps, and $r_1[r_2]$ for the map coinciding with r_2 wherever the latter is defined, with r_1 elsewhere.

Besides lambda-abstractions and values of primitive types, there are three new kinds of values in the calculus: *unbound terms* $\langle u \mid t \rangle$, *rebindings* $\langle u \mid r \rangle$ and *name abstractions* $\Lambda\alpha.t$.

An unbound term, e.g., $\langle x \mapsto N \mid x{+}1 \rangle$, represents code which is not directly used but, rather, "boxed", as the brackets suggest. This boxed code is possibly open, and can be dynamically rebound through a nominal interface.

Conversely, a rebinding represents code which can be used to dynamically rebind open code. A rebinding can be unbound as well, that is, its code can be open, as in $\langle x \mapsto N \mid N_1 \mapsto 0, N_2 \mapsto 1{+}x\rangle$. According to the sequence notation, an unbound term with an empty unbinding map is simply written $\langle\ \mid t\rangle$, and analogously for a rebinding.

Name abstractions can be used to write terms which are parametric w.r.t. the nominal interface, e.g., $\Lambda\alpha.\langle x \mapsto \alpha \mid x{+}1\rangle$ is the parametric version of the above unbound term. Note that, differently from, e.g., [12], we take a stratified approach where *names are not terms*, to keep separate the conventional language, which is here lambda-calculus for simplicity, from the meta-level constructs, whose semantics is in principle independent. Hence, we have ad-hoc constructs for name abstraction and name application.

Besides values and variables, terms include compound terms constructed by the following operators: application, name application, rebinding, run, overriding, and renaming. They are illustrated together with reduction rules given in Figure 1.

Rule (Ctx) is the usual contextual closure.

Rule (App) is standard. The *application of a substitution to a term*, $t\{s\}$, is defined in the standard way. Note that a variable occurrence in the domain of an unbinding map behaves like a λ-binder. Hence, the variables in $dom(u)$ are not free in $\langle u \mid t\rangle$, and not subject to substitution.

In a name application $t\ X$, t and X are expected to reduce to a name abstraction, and a name constant, respectively. The name abstraction is applied to the name constant, as modeled by rule (Name-App). The *application of a name substitution to a term*, $t\{\alpha \mapsto N\}$, that is, substitution of a name variable with a name constant, is defined in the standard way. In particular, the only construct that introduces binders is name abstraction, whereas name substitution has to be propagated also to unbinding maps, rebinding maps, and renamings. Note that, by name substitution, we could obtain ill-formed terms, e.g., $\langle\ \mid \alpha \mapsto 0, N \mapsto 1\rangle\{\alpha \mapsto N\}$ gives $\langle\ \mid \langle\ \mid N \mapsto 0, N \mapsto 1\rangle\rangle$. Since reduction is defined on well-formed terms, in this case the rule cannot be applied.

In a term $t_1 \succ t_2$, the arguments of the rebinding operator t_1 and t_2 are expected to reduce to a rebinding and to an unbound term, respectively. When the rebinding is applied to the unbound term, rule (Reb-App), all the variables associated with names provided by the rebinding (side condition $rng(u_2) \cap dom(r) = \emptyset$) are replaced by the corresponding terms, and are therefore removed from the unbinding map of the unbound term. However, the unbinding map of the resulting unbound term is augmented with the unbinding map of the rebinding term. The condition $dom(u) \cap dom(u_2) = \emptyset$, implicitly required for the well-formedness of u, u_2, can be always satisfied by applying a suitable α-renaming to one of the two terms. We also tacitly assume that the rule is applicable only when $r(u_1(x))$ is defined for all $x \in dom(u_1)$, that is, $rng(u_1) \subseteq dom(r)$. For instance,

$$\langle y \mapsto N_2 \mid N_1 \mapsto y{+}2, N_3 \mapsto y\rangle \succ \langle x \mapsto N_1, y \mapsto N_2 \mid x{+}y\rangle$$

reduces to $\langle y \mapsto N_2, y' \mapsto N_2 \mid (y{+}2){+}y'\rangle$.

In a term $!t$, the argument of the run operator is expected to reduce to an unbound term with no names to be rebound, which can be "unboxed", rule (Run). For instance, $!\langle\ \mid 0{+}1\rangle$ reduces to $0{+}1$, which can then be evaluated. Unbound terms can be unboxed and executed through the run operator only after their open code has been completed through one or more applications of rebindings so that they do not contain unbound variables; for instance, the unbound term $\langle x \mapsto N \mid x{+}1\rangle$ can be made self-contained with the rebinding $\langle\ \mid N \mapsto 0, N' \mapsto 1\rangle$.

In a term $t_1 \lhd t_2$, the arguments of the overriding operator are expected to reduce to two rebindings. Rule (Over) allows one to merge the two rebindings giving preference to the right

one in case of conflict. Unbinding maps u_1 and u_2 are simply merged together (hence, names are shared). As it happens for rule (Reb-App), the implicit condition $dom(u_1) \cap dom(u_2) = \emptyset$ can be always satisfied by applying a suitable α-renaming to one of the two terms. For instance,

$$\langle x \mapsto N_1 \mid N_2 \mapsto x\ 1, N_3 \mapsto 1\rangle \lhd \langle x \mapsto N_1 \mid N_3 \mapsto 2, N_4 \mapsto x\ 2\rangle$$

reduces to $\langle x \mapsto N_1, x' \mapsto N_1 \mid N_2 \mapsto x\ 1, N_3 \mapsto 2, N_4 \mapsto x'\ 2\rangle$.

In a term $\sigma_1 \ltimes t \rtimes \sigma_2$, the argument of the renaming operator is expected to reduce to a rebinding $\langle u \mid r\rangle$. The renaming operator is used for adapting the nominal interfaces of the unbinding and rebinding map u and r, respectively, rule (Rename). With the renaming σ_1 it is possible to merge names, while with σ_2 one can duplicate and remove terms; for instance

$$(N_1 \mapsto N_2, N_2 \mapsto N_2) \ltimes \langle x \mapsto N_1, y \mapsto N_2 \mid N_1 \mapsto 0, N_3 \mapsto 1\rangle \rtimes (N_1 \mapsto N_1, N_2 \mapsto N_1)$$

reduces to $\langle x \mapsto N_2, y \mapsto N_2 \mid N_1 \mapsto 0, N_2 \mapsto 0\rangle$. As for rule (Reb-App), we tacitly assume that $rng(u) \subseteq dom(\sigma_1)$ and $rng(\sigma) \subseteq dom(r_2)$ respectively hold.

Renamings and name abstractions can be used together to favor dynamic software adaptation and reuse. For instance, the term

$$t = \Lambda\alpha_1.\Lambda\alpha_2.\lambda x_r.(\ltimes x_r \rtimes (N_1 \mapsto \alpha_1, N_2 \mapsto \alpha_2)) \succ \langle x_1 \mapsto N_1, x_2 \mapsto N_2 \mid x_1\ x_2\rangle$$

is expected to take a rebinding x_r with generic shape $\langle \mid \alpha_1 \mapsto t_1, \alpha_2 \mapsto t_2, \ldots\rangle$, to adapt it by renaming and then to apply it to the unbound term $\langle x_1 \mapsto N_1, x_2 \mapsto N_2 \mid x_1\ x_2\rangle$; as an example, $t\ N_3\ N_4\ \langle \mid N_3 \mapsto \lambda x.x{+}1, N_4 \mapsto 1\rangle$ reduces (in some steps) to 2.

To make the paper self-contained, we briefly recall some examples which show the role of our calculus as unifying foundation for dynamic scoping, rebinding, and meta-programming features, referring to [1, 2] for other examples and more explanations. Then, we illustrate in more detail two examples, that is, *selection of an arbitrary component of a module*, and *adaptation of mixins* (also used in Section 4), which illustrate the expressive power of the name variables introduced in this paper. In the examples we use the let construct, `let` $x = t_1$ `in` t_2, as syntactic sugar for $(\lambda x.t_2)\ t_1$.

Dynamic Scoping

In our calculus, names play the role of dynamic variables, and dynamic scoping can be encoded by unbinding and rebinding, e.g., in the traditional example

```
let x=3 in
  let f=lambda y.x+y in
    let x=5 in
      f 1
```

dynamic scoping, which leads to result 6 rather than 4, can be encoded as follows:

```
let x=3 in
  let f=lambda y.<x ↦ X | x+y> in
    let x=5 in
      !(< | X ↦ x> ≻ (f 1))
```

Rebinding of marshalled computations

Assuming to enrich the calculus with primitives for concurrency, we can model exchange of mobile code, which may contain unbound variables to be rebound by the receiver, by the parallel composition $t_{snd} \mid\mid t_{rcv}$ where t_{snd} is defined by

```
let x = t_x in
  let y = t_y in
    let f_code = < x ↦ X, y ↦ Y | t(x,y)> in
      let f = !(< | X ↦ x, Y ↦ y> ≻ f_code) in
      t(f) //f is used locally
      send(f_code).nil
```

and t_{rcv} is defined by

```
let x = t_x^new in
  receive(f_code).send (< | X ↦ x> ≻ f_code).nil
```

In this example, open code `f_code` is first used locally in the process on the left-hand side of the parallel operator, binding resources `x` and `y` to their local versions, and then sent to the process on the right-hand side. Note that incremental rebinding allows this process to receive the code, to provide a new version of the resource `x`, and to resend still open code. Here `t(x,y)` and `t(f)` are terms with free variables `x,y` and `f`, respectively.

Multi-stage programming

First of all, note that a rebinding of shape $\langle y \mapsto Y \mid Y \mapsto f\ y\rangle$, where f is some function, acts as a filter which, applied to an open code of shape $\langle y \mapsto Y \mid y\rangle$, transforms it in $\langle y \mapsto Y \mid f\ y\rangle$. Hence, a repeated application obtained, e.g., by recursion, transforms the original open code in $\langle y \mapsto Y \mid f^n\ y\rangle$.

This "recursive rebinding pattern" is used in the example below, one of the most typically used in literature for illustrating program specialization via generative programming: the power function `pow` which, taken the integer n, returns the optimized function `lambda x.x*...*x` computing x^n.

```
let rec aux_pow = lambda n.
   if n>0 then <x ↦ X, y ↦ Y | Y ↦ x*y> ≻ aux_pow (n-1)
   else <y ↦ Y | y>
let pow = lambda n.
   let f = < | Y ↦ 1> ≻ (aux_pow n) in
      lambda x. !(< | X ↦ x> ≻ f)
```

Multi-staging is obtained by incrementally rebinding unbound terms; the recursive function `aux_pow` returns an unbound term which depends on the two names `X` and `Y`: the former corresponds to the base, whereas the latter is used as a hook to generate the desired specialization, and then it is bound to 1 in the `pow` function. We refer to [4] for more details and an example of computation.

We now turn to show more in details two examples which illustrate the expressive power of the notion of name variable introduced in this paper.

Module/component selection

Rebinding terms directly support the notion of module/component. We have already shown [2] how member selection of closed (that is, where all dependencies have been resolved)

modules/components can be encoded. For instance, the following term encodes an operator which selects the Y member of a (closed) module represented by a rebinding:

```
ts =lambda x. !(x ≻ < y ↦ Y | y >)
```

For instance the term t_s `< | X↦0, Y↦42>` evaluates to 42. However, in this way selection can be encoded only for a single fixed name constant (`Y` in this specific case).

With the newly introduced construct of name abstraction, a generic definition of the selection operator can be provided by a single term of the calculus.

```
t's =Lambda α. lambda x. !(x ≻ < y ↦ α | y >)
```

In this way, the same term t'_s can be used for selecting members associated with arbitrary names. For instance, if t =`< | F↦lambda n.n+1, N↦ 41>`, then (t'_s `F` t) (t'_s `N` t) evaluates to 42.

In mainstream object-oriented languages such meta-programming facilities are supported either by specific libraries for reflection, or by more flexible constructs, as the JavaScript bracket notation. In all cases, no static checking is performed to ensure that the selected names will be always defined at runtime.

For instance, with the use of the bracket notation in JavaScript[1] it is possible to define the following function:

```
function select(name,object){return object[name]}
```

The notation e_1`[`e_2`]` allows programmers to access properties of the object denoted by e_1 whose name is defined by the arbitrary expression e_2. Therefore, `select ("val",{val:42})` returns 42, whereas `select ("foo",{val:42})` is undefined.

Adaptation of mixins

Mixin classes [5] and mixin modules [3] are notions commonly employed in generic programming to support software reuse.

Among statically typed mainstream object-oriented programming languages, mixins are only supported by C++, with templates, see [15]. The following class template defines class `CheckedMixin` which is parametric in its base class, represented by the template parameter `B`.

```
template <class B>
class CheckedMixin : public B {
public:
  static int checked_op(int value) {
    if(B::in_bounds(value))
      return B::op(value);
    else
      throw std::logic_error("Illegal argument");
  }
};
```

The mixin adds the static method `checked_op`, and can be instantiated with classes defining `op(int)` and `in_bounds(int)`, as in the following code fragment:

[1] All examples presented here are compliant with the ECMAScript 5 syntax, although some of them could be written in a slightly more concise way by using the new features and shorthands introduced with the recently released specification of ECMAScript 6.

```
class Sqrt {
public:
  static int op(int value) { return sqrt(value); }
  static bool in_bounds(int value){ return value >= 0; }
};

class Checked_sqrt : public CheckedMixin<Sqrt> { };

int main() {
  assert(Checked_sqrt::checked_op(4)==2) ;
  assert(Checked_sqrt::op(-4)!=2) ;
  assert(Checked_sqrt::checked_op(-4)!=2) ; // throws logic_error
}
```

Thanks to the generic code defined by `CheckedMixin`, class `Sqrt` is extended with the static method `checked_op` which checks whether the argument is non negative, before applying the static method `op` which, in turn, applies the library function[2] `sqrt`.

The main limitation of mixins implemented with C++ class templates is their inability to be adapted to classes where methods have names different from those chosen in the mixin. In the case of `CheckedMixin`, the parametric base class must provide static methods named `op(int)` and `in_bounds(int)`. Furthermore, typechecking of C++ templates is not compositional, therefore such constraints are checked every time the template is instantiated.

Dynamic languages, as JavaScript [10], allow, instead, *adaptation* of mixins, in the sense that they can be parameterized not only on the implementation, but also on the name, of a required method.

In this case the mixin is defined by a function[3] taking three arguments that are expected to contain strings: `op` denotes the name of the operation that has to be checked, `in_bounds` denotes the name of the operation that performs the check, and `new_op` denotes the name of the newly added operation corresponding to the checked version of `op`.

```
function CheckedMixin(op,in_bounds,new_op){
   this[new_op] = function(x){
      if(!this[in_bounds](x))
         throw new Error('Illegal argument')
      return this[op](x)
   }
}
```

Thanks to the bracket notation the programmer can pass to the `CheckedMixin` function the proper strings to adapt the instances of `CheckedMixin`.

```
var sqrt={ // a new object with two properties
   sqrt:Math.sqrt,
   check_arg:function(x){return x>=0}
}
var chk_sqrt=new CheckedMixin('sqrt','check_arg','checked_sqrt')
Object.setPrototypeOf(chk_sqrt,sqrt) // sqrt prototype of chk_sqrt
chk_sqrt.sqrt(-4) // evaluates to NaN
```

[2] Function `sqrt` does not perform any check, unless `math_errhandling` has the constant `MATH_ERREXCEPT` set.

[3] We recall that JavaScript is a prototype-based language where objects are dynamically created through functions, although an equivalent class-based notation has been introduced in ECMAScript 6.

```
chk_sqrt.checked_sqrt(4) // evaluates to 2
chk_sqrt.checked_sqrt(-4) // throws Error: Illegal argument
```

The same function `CheckedMixin` can be used to extend an object which computes the `log` function.

```
var log={ // a new object with two properties
   log:Math.log10,
   check_arg:function(x){return x>=0}
}
var chk_log=new CheckedMixin('log','check_arg','safe_log')
Object.setPrototypeOf(chk_log,log) // log prototype of chk_log
chk_log.log(-10) // evaluates to NaN
chk_log.safe_log(10) // evaluates to 1
chk_log.safe_log(-10) // throws Error: Illegal argument
```

Thanks to the support for name manipulation, mixin adaptation and application can be expressed in our calculus; furthermore, as will be shown in Section 3, compositional typechecking ensures the type correctness of mixin adaptation and application. The JavaScript example given above can be recast[4] in our calculus as follows:

```
tm =Lambda α_op.Lambda α_in_b.Lambda α_n_op.lambda r.
 let n_op =
  !(r ≻ < op ↦ α_op, in_b ↦ α_in_b|lambda x. if (not in_b(x)) -1 else op(x)>)
 in r ◁ < | α_n_op ↦ n_op >
```

As in the previous example, the mixin takes three names α_{op}, α_{in_b}, and α_{n_op}, corresponding to the name of the operation that has to be checked, the name of the operation that performs the check, and the name of the newly added operation which is the checked version of the operation α_{op}. Then it takes a rebinding r, which is expected to provide a definition for the operations α_{op} and α_{in_b}, and that is applied to an unbound term which defines the new operation in terms of α_{op} and α_{in_b}. The result of the application of the rebinding is run to get the value corresponding to the new operation, and, finally, the rebinding is extended with the new component by means of the overriding operator.

3 Typed Calculus

Figure 2 shows the syntax of the typed calculus, which is extended by annotating variables and names with types, and name variables with constraints, as explained in detail below.

Constraints c are sequences of inequalities $X \neq Y$. We assume that c is a set, that is, order and repetitions are immaterial, and, moreover, inequalities of shape $N_1 \neq N_2$ for N_1 and N_2 different names are immaterial as well, that is, we can always assume that c does not contain such inequalities.

Types includes function types, constrained name-polymorphic types, unbound types $\langle \Delta \mid T \rangle$, and rebinding types $\langle \Delta_1 \mid \Delta_2 \rangle^{\nu}$. For simplicity we omit basic types for primitive values such as integers or booleans. In the explanations that follow, we illustrate in more detail the new feature of the type system, that is, constrained name-polymorphic types. The reader can refer to our previous work [1, 2] for more explanations and examples on unbound types and open/closed rebinding types.

[4] Since the calculus does not support exceptions, in case the bounds are not verified the function simply returns the conventional value -1.

t	$::=$	$\ldots \mid \lambda x{:}T.t \mid \langle u \mid t\rangle \mid \langle u \mid r\rangle \mid \Lambda\alpha{:}c.t \mid x \mid t_1\ t_2 \mid t\ X \mid$	
		$t_1 \succ t_2 \mid !t \mid t_1 \lhd t_2 \mid \sigma_1 \ltimes t \rtimes \sigma_2$	term
u	$::=$	$x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m$	unbinding map
r	$::=$	$X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m$	rebinding map
σ	$::=$	$X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m$	renaming
X, Y	$::=$	$N \mid \alpha$	name
T	$::=$	$\ldots \mid T_1 \to T_2 \mid \forall\alpha{:}c.T \mid \langle\Delta \mid T\rangle \mid \langle\Delta_1 \mid \Delta_2\rangle^\nu$	type
c	$::=$	$X_1{\neq}Y_1 \ldots X_m{\neq}Y_m$	constraints
Δ	$::=$	$X_1{:}T_1, \ldots, X_m{:}T_m$	name context
ν	$::=$	$\circ \mid +$	(variance) annotation
Σ	$::=$	$A; c; \Gamma$	typing context
A	$::=$	$\alpha_1 \ldots \alpha_n$	name variables
Γ	$::=$	$x_1{:}T_1, \ldots, x_m{:}T_m$	variable context

Figure 2 Typed calculus: syntax

Function types correspond to lambda abstractions, where the variable is now annotated with a type.

Constrained name-polymorphic types correspond to name abstractions, where the name variable is now annotated with constraints. Constraints are necessary to guarantee that for each possible instantiation of the name variable we get well-formed terms and types. For instance, the term $\Lambda\alpha{:}\alpha \neq N.\langle \mid N{:}\mathtt{int} \mapsto 0, \alpha{:}\mathtt{int} \mapsto 1\rangle$ is a rebinding parametric in the name of one of its two components, which, however, must be different from the constant name N of the other component.

Unbound types $\langle\Delta \mid T\rangle$ correspond to open code: Δ is a sequence $X_1{:}T_1, \ldots, X_m{:}T_m$ called *name context*. The type specifies that the open code needs the rebinding of the names X_i to terms of type T_i $(1 \leq i \leq m)$ in order to correctly produce a term of type T.

Rebinding types $\langle\Delta_1 \mid \Delta_2\rangle^\nu$ correspond to rebindings; the name context Δ_1 specifies the names which the rebinding depends on, while the name context $\Delta_2 = X_1{:}T_1, \ldots, X_m{:}T_m$ specifies that the rebinding map associates each name X_i with a term of type T_i $(1 \leq i \leq m)$. If the type is annotated with $\nu = +$, then we say that the type is *open* (or *non-exact*), and the rebinding map is allowed to contain more associations than those specified in the name context. The annotation $\nu = \circ$ is used for *closed* (or *exact*) types, to enforce that the domain of the rebinding map exactly coincides with the domain of Δ_2. In the typing rules we will use the binary operator $\sqcup$ over annotations, defined by $\circ \sqcup \nu = \nu \sqcup \circ = \nu$, and $+ \sqcup + = +$.

Renamings, as well as values, evaluation contexts, and substitutions are defined as for the untyped language.

3.1 Well-Formedness

Figure 3 defines well-formed names, constraints, types, name contexts, rebinding maps and renamings. We say that X *could be equal to* Y *under* c, written $c \models X \stackrel{?}{=} Y$, if $X \neq Y \notin c$ and $Y \neq X \notin c$, and that all constraints in c *refer to* α, written $\alpha \Vdash c$, if for all $Y \neq X$ in c, either $X = \alpha$ or $Y = \alpha$.

A name X is well-formed under name variables A (written $A \vdash X$) if it is either a name constant, rule (WF-NAME-CONST), or a name variable in A, rule (WF-NAME-VAR).

A set of constraints c is well-formed under name variables A, written $A \vdash c$, if variables occurring in c belong to A.

Well-formedness of a type T under name variables A and constraints c is written $A; c \vdash T\ \mathtt{OK}$.

$$\text{(WF-NAME-CONST)}\ \frac{}{A \vdash N} \qquad \text{(WF-NAME-VAR)}\ \frac{}{A \vdash \alpha}\ \alpha \in A$$

$$\text{(WF-EMPTY-CONS)}\ \frac{}{A \vdash} \qquad \text{(WF-NON-EMPTY-CONS)}\ \frac{A \vdash X_1 \quad A \vdash X_2 \quad A \vdash c}{A \vdash X_1 \neq X_2, c}\ \text{not } X_1 = X_2$$

$$\text{(WF-ARROW-TYPE)}\ \frac{A; c \vdash T\ \texttt{OK} \quad A; c \vdash T'\ \texttt{OK}}{A; c \vdash T \to T'\ \texttt{OK}} \qquad \text{(WF-NAME-ARROW-TYPE)}\ \frac{\begin{array}{c}A \cup \{\alpha\} \vdash c' \quad \alpha \Vdash c' \\ A \cup \{\alpha\}; c, c' \vdash T\ \texttt{OK}\end{array}}{A; c \vdash \forall\alpha{:}c'.T\ \texttt{OK}}\ \alpha \notin A$$

$$\text{(WF-UNB-TYPE)}\ \frac{A; c \vdash \Delta\ \texttt{OK} \quad A; c \vdash T\ \texttt{OK}}{A; c \vdash \langle \Delta \mid T \rangle\ \texttt{OK}} \qquad \text{(WF-REB-TYPE)}\ \frac{A; c \vdash \Delta'\ \texttt{OK} \quad A; c \vdash \Delta\ \texttt{OK}}{A; c \vdash \langle \Delta' \mid \Delta \rangle^{\nu}\ \texttt{OK}}$$

$$\text{(WF-NAME-CTX)}\ \frac{\begin{array}{c}A; c \vdash T_k\ \texttt{OK}\ (1 \leq k \leq m) \quad A \vdash X_k\ (1 \leq k \leq m) \\ c \models X_i \stackrel{?}{=} X_j \Rightarrow T_i = T_j\ (1 \leq i, j \leq m)\end{array}}{A; c \vdash X_1{:}T_1, \ldots, X_n{:}T_m\ \texttt{OK}}$$

$$\text{(WF-REB-MAP)}\ \frac{A \vdash X_k\ (1 \leq k \leq m) \quad c \models X_i \stackrel{?}{=} X_j \Rightarrow t_i = t_j\ (1 \leq i, j \leq m)}{A; c \vdash X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\ \texttt{OK}}$$

$$\text{(WF-REN)}\ \frac{A \vdash X_k\ (1 \leq k \leq m) \quad A \vdash Y_k\ (1 \leq k \leq m) \quad c \models X_i \stackrel{?}{=} X_j \Rightarrow Y_i = Y_j\ (1 \leq i, j \leq m)}{A; c \vdash X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m\ \texttt{OK}}$$

Figure 3 Well-formed names, constraints, types, name contexts, rebinding maps, and renamings

The side condition $\alpha \notin A$ in (WF-NAME-ARROW-TYPE) avoids unsoundness caused by conflicts between name variables; otherwise, for instance, the type $\forall\alpha{:}\alpha \neq N.\forall\alpha{:}.\langle N{:}\texttt{int}, \alpha{:}\texttt{bool} \mid \texttt{int}\rangle$ would be considered well-formed, because the constraint $\alpha \neq N$ referring to the outer binder could be erroneously used also for the inner binder; however, in the unbound type $\langle N{:}\texttt{int}, \alpha{:}\texttt{bool} \mid \texttt{int}\rangle$, α is bound to the inner binder α for which the constraint $\alpha \neq N$ required for guaranteeing that the type is well-formed (see below) is missing. The side condition $\alpha \notin A$ can be always satisfied by renaming the type variable; for instance, given the type $\forall\alpha{:}.\forall\alpha{:}\alpha \neq N.\langle N{:}\texttt{int}, \alpha'\texttt{bool} \mid \texttt{int}\rangle$, it is possible to derive that the equivalent type $\forall\alpha{:}.\forall\alpha'{:}\alpha' \neq N.\langle N{:}\texttt{int}, \alpha'{:}\texttt{bool} \mid \texttt{int}\rangle$ is well-formed, with α' any name variable different from α.

An unbound type is well-formed under name variables A and constraints c only if types occurring in the sequence are well-formed, name variables occurring in the sequence belong to A, and names which could be equal under c are mapped to the same type, as specified by rules (WF-UNB-TYPE) and (WF-NAME-CTX) in Figure 3.

A rebinding type is well-formed under name variables A and constraints c only if types occurring in the sequences Δ_1 and Δ_2 are well-formed, name variables occurring in the sequences belong to A, and names which could be equal under c are mapped in the same type, analogously to what is required for an unbound term, as specified by rules (WF-REB-TYPE) and (WF-NAME-CTX) in Figure 3.

(Untyped) rebinding maps are well-formed if names which could be equal under c are mapped in the same term, and name variables belong to A, as specified by rule (WF-REB-MAP).

Well-formedness of renamings requires that name variables belong to A, and names which could be equal under c are mapped in the same name, as specified by rule (WF-REN).

$$\frac{A; c \vdash T\ \texttt{OK} \quad A; c \vdash t\ \texttt{OK}}{A; c \vdash \lambda x{:}T.t\ \texttt{OK}} \qquad \frac{A \vdash X_i\ (1 \leq i \leq m) \quad A; c \vdash T_i\ \texttt{OK}\ (1 \leq i \leq m) \quad A; c \vdash t\ \texttt{OK}}{A; c \vdash \langle x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m \mid t\rangle\ \texttt{OK}}$$

$$\frac{A \vdash X_i\ (1 \leq i \leq m) \quad A; c \vdash T_i\ \texttt{OK}\ (1 \leq i \leq m) \quad A; c \vdash r\ \texttt{OK}}{A; c \vdash \langle x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m \mid r\rangle\ \texttt{OK}}$$

$$\frac{A \cup \{\alpha\} \vdash c' \quad \alpha \Vdash c' \quad A \cup \{\alpha\}; c, c' \vdash t\ \texttt{OK}}{A; c \vdash \Lambda\alpha{:}c'.t\ \texttt{OK}}\ \alpha \notin A$$

$$\frac{A; c \vdash t_1\ \texttt{OK} \quad A; c \vdash t_2\ \texttt{OK}}{A; c \vdash t_1\ t_2\ \texttt{OK} \qquad A; c \vdash t_1 \succ t_2\ \texttt{OK} \qquad A; c \vdash t_1 \lhd t_2\ \texttt{OK}} \qquad \frac{A \vdash c\{\alpha \mapsto X\} \quad A; c \vdash t\ \texttt{OK}}{A; c \vdash t\ X\ \texttt{OK}}$$

$$\frac{}{A; c \vdash x\ \texttt{OK}} \qquad \frac{A; c \vdash t\ \texttt{OK}}{A; c \vdash\ !t\ \texttt{OK}} \qquad \frac{A; c \vdash \sigma_i\ \texttt{OK}\ (1 \leq i \leq 2) \quad A; c \vdash t\ \texttt{OK}}{A; c \vdash \sigma_1 \ltimes t \rtimes \sigma_2\ \texttt{OK}}$$

Figure 4 Well-formed terms

$$(\textsc{Sub-arr})\ \frac{\vdash T_1' \leq T_1 \quad \vdash T_2 \leq T_2'}{\vdash T_1 \to T_2 \leq T_1' \to T_2'} \qquad (\textsc{Sub-name-arr})\ \frac{c_2 \vdash c_1 \quad \vdash T_1 \leq T_2}{\vdash \forall\alpha{:}c_1.T_1 \leq \forall\alpha{:}c_2.T_2}$$

$$(\textsc{Sub-unb})\ \frac{\vdash \Delta' \leq \Delta \quad \vdash T \leq T'}{\vdash \langle\Delta \mid T\rangle \leq \langle\Delta' \mid T'\rangle} \qquad (\textsc{Sub-open-reb})\ \frac{\vdash \Delta_1' \leq \Delta_1 \quad \vdash \Delta_2 \leq \Delta_2'}{\vdash \langle\Delta_1 \mid \Delta_2\rangle^{\nu} \leq \langle\Delta_1' \mid \Delta_2'\rangle^{+}}$$

$$(\textsc{Sub-closed-reb})\ \frac{\vdash \Delta_1' \leq \Delta_1 \quad \vdash T_i \leq T_i'\ (1 \leq i \leq n)}{\vdash \langle\Delta_1 \mid X_1{:}T_1, \ldots, X_n{:}T_n\rangle^{\circ} \leq \langle\Delta_1' \mid X_1{:}T_1', \ldots, X_n{:}T_n'\rangle^{\circ}}$$

$$(\textsc{Sub-name-ctx})\ \frac{\forall i\ (1 \leq i \leq n)\ \exists j\ (1 \leq j \leq m)\ X_i' = X_j \wedge\ \vdash T_j \leq T_i'}{\vdash X_1{:}T_1, \ldots, X_m{:}T_m \leq X_1'{:}T_1', \ldots, X_n'{:}T_n'}$$

Figure 5 Typed calculus: subtyping rules

The notion of well-formedness is extended to typed terms in Figure 4. Note that, if $\emptyset; \emptyset \vdash t\ \texttt{OK}$, then the erasure of t obtained by removing the type annotations is well-formed in the sense of Section 2.

3.2 Subtyping

The subtyping relation $\vdash T \leq T'$ is defined in Figure 5.

Subtyping between function types is standard. A constrained polymorphic type can be made more specific by relaxing the constraints (constraint entailment is defined in Figure 6) or making more specific the type obtained by instantiation, while the two binders can always be made equal by a suitable α-renaming. For instance, $\vdash \forall\alpha_1{:}\alpha_1 \neq \alpha_2.T \leq \forall\alpha_1{:}\alpha_1 \neq N, \alpha_2 \neq \alpha_1.T$ is derivable.

Subtyping between unbound types obeys a rule similar to that for function types: the relation is contravariant in the name context, and covariant in the type returned after rebinding. Subtyping between name contexts is defined by the usual rule for record subtyping: both width and depth subtyping are allowed. Width and depth subtyping are also allowed between rebinding types, in case the right-hand side (rhs for short) type in the relation is open, because a closed type can always be considered as an open type, but not the other way around. This is a consequence of the fact that closed types express more restrictive constraints on rebinding maps. For instance, the rebinding $\langle\ \mid X{:}T_X \mapsto t_x, Y{:}T_Y \mapsto t_y\rangle$ has, for any Δ,

$$\text{(Ent-empty)}\ \frac{}{c \vdash \emptyset} \qquad \text{(Ent-var)}\ \frac{c_1 \vdash c_2}{c_1 \vdash X_1 \neq X_2, c_2}\ X_1 \neq X_2 \in c_1 \text{ or } X_2 \neq X_1 \in c_1$$

Figure 6 Constraint entailment

type $\langle \Delta \mid X{:}T_X, Y{:}T_Y \rangle^\nu$ for both $\nu = +$ and $\nu = \circ$, whereas it has type $\langle \Delta \mid X{:}T_X \rangle^\nu$ only for $\nu = +$; note also that the most precise type for this term is $\langle \mid X{:}T_X, Y{:}T_Y \rangle^\circ$. When the rhs type in the subtyping relation is a closed rebinding type, then the lhs type must be closed as well, and, therefore, it must define the same set of names; in this case only depth subtyping is allowed.

Figure 6 defines constraint entailment.

Rule (Ent-empty) states that the empty set of constraints is always entailed; rule (Ent-var) states that $X_1 \neq X_2$ is entailed from c if it is contained in c, up to symmetry. Since set of constraints must be satisfiable, as specified in rule (wf-non-empty-cons) in Figure 3, the case $c_1 \vdash c_2$ where c_1 contains $X \neq X$ is not considered.

3.3 Typing Rules

The typing judgment has shape $A; c; \Gamma \vdash t : T$, meaning that the term t has type T under the name variables A, constraints c, and context Γ providing types for the free variables. The typing rules are given in Figure 7.

The type system supports subsumption, as specified by rule (T-Sub); T' is required to be well-formed, whereas the fact that T is well-formed can be derived from the premise $A; c; \Gamma \vdash t : T$, as we will state in Lemma 5; indeed, it can be proved by induction on the typing rules that if $A; c; \Gamma \vdash t : T$ is derivable, then T is well-formed.

Rule (T-Abs) for lambda abstractions is standard.

In rule (T-Name-Abs), the term $\Lambda\alpha'{:}c'.t$ is well-typed if the introduced constraints c' are consistent under the current name variables augmented by α', t is well-typed taking the union of the constraints, and the set of constraints c' refer to α'. At the end of Section 4, we show an example of how this last requirement is necessary for the proof of correctness.

In rule (T-Unb), the term $\langle u \mid t \rangle$ is well-typed if the name context extracted from u by the auxiliary function *name_ctx*, say, $X_1{:}T_1, \ldots, X_m{:}T_m$, is well-formed under the current name variables and constraints, that is, X_i belongs to A if it is a name variable, and, if X_i could be equal to X_j under c, then they are mapped in the same type. The resulting type T is obtained by typing t in the context updated by that extracted from u by the auxiliary function *ctx*. Both auxiliary functions are defined at the bottom of Figure 7.

In rule (T-Reb), the term $\langle u \mid r \rangle$ is well-typed if the name contexts extracted from u and r are well-formed under the current name variables and constraints. Moreover, r must be well-formed under the current constraints, that is, names which could be equal are mapped in the same term. Finally, for each name in the domain of r, annotated with type, say, T, the associated term must have type T in the context updated by that extracted from u by the auxiliary function *ctx*. Note that an exact type can be always deduced.

Rules (T-Var) and (T-App) are standard.

In rule (T-Name-App), the term $t\ X$ is well-typed if X belongs to A if it is a name variable, t has a constrained polymorphic type $\forall\alpha{:}c'.T$, and by replacing α by X in the constraints c' we do not get inequalities of shape $Y{\neq}Y$. In this case, the resulting type is obtained by replacing α by X in T. The obvious definitions of replacing a name variable by a name in constraints and types are omitted.

$$\text{(T-SUB)}\ \frac{A; c; \Gamma \vdash t : T \quad A; c \vdash T' \ \texttt{OK} \quad \vdash T \leq T'}{A; c; \Gamma \vdash t : T'} \qquad \text{(T-ABS)}\ \frac{A; c \vdash T_1 \ \texttt{OK} \quad A; c; \Gamma[x{:}T_1] \vdash t : T_2}{A; c; \Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2}$$

$$\text{(T-NAME-ABS)}\ \frac{A \cup \{\alpha'\} \vdash c' \quad A \cup \{\alpha'\}; c, c'; \Gamma \vdash t : T \quad \alpha' \Vdash c'}{A; c; \Gamma \vdash \Lambda\alpha'{:}c'.t : \forall\alpha'{:}c'.T}\ \alpha' \notin A$$

$$\text{(T-UNB)}\ \frac{A; c \vdash \Delta \ \texttt{OK} \quad A; c; \Gamma[\Gamma'] \vdash t : T}{A; c; \Gamma \vdash \langle u \mid t \rangle : \langle \Delta \mid T \rangle} \quad \begin{array}{l} name_ctx(u) = \Delta \\ ctx(u) = \Gamma' \end{array}$$

$$\text{(T-REB)}\ \frac{\begin{array}{c} A; c \vdash X_1 \mapsto t_1, \ldots, X_m \mapsto t_m \ \texttt{OK} \\ A; c \vdash \langle \Delta_1 \mid \Delta_2 \rangle^\circ \ \texttt{OK} \quad A; c; \Gamma[\Gamma'] \vdash t_i : T_i \quad (1 \leq i \leq m) \end{array}}{A; c; \Gamma \vdash \langle u \mid X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m \rangle : \langle \Delta_1 \mid \Delta_2 \rangle^\circ} \quad \begin{array}{l} name_ctx(u) = \Delta_1 \\ ctx(u) = \Gamma' \\ \Delta_2 = X_1{:}T_1, \ldots, X_m{:}T_m \end{array}$$

$$\text{(T-VAR)}\ \frac{}{A; c; \Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-APP)}\ \frac{\Sigma \vdash t_1 : T_1 \to T_2 \quad \Sigma \vdash t_2 : T_1}{\Sigma \vdash t_1\ t_2 : T_2}$$

$$\text{(T-NAME-APP)}\ \frac{A - \{\alpha\} \vdash c'\{\alpha \mapsto X\} \quad A; c; \Gamma \vdash t : \forall\alpha{:}c'.T \quad A \vdash X}{A; c; \Gamma \vdash t\ X : T\{\alpha \mapsto X\}}$$

$$\text{(T-OVER)}\ \frac{\begin{array}{c} A; c \vdash \Delta_1, \Delta_2 \ \texttt{OK} \\ A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta_1, \Delta_1' \rangle^{\nu_1} \quad A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta_2 \rangle^{\nu_2} \end{array}}{A; c; \Gamma \vdash t_1 \lhd t_2 : \langle \Delta \mid \Delta_1, \Delta_2 \rangle^{\nu_1 \sqcup \nu_2}} \quad \begin{array}{l} (\Delta_1 = \emptyset \text{ or } \nu_2 = \circ) \text{ and} \\ dom(\Delta_1') \subseteq dom(\Delta_2) \end{array}$$

$$\text{(T-RUN)}\ \frac{A; c; \Gamma \vdash t : \langle\ \mid T \rangle}{A; c; \Gamma \vdash !t : T}$$

$$\text{(T-REB-APP)}\ \frac{\begin{array}{c} A; c \vdash \Delta_1, \Delta_2 \ \texttt{OK} \\ A; c; \Gamma \vdash t_1 : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2 \rangle^{\nu} \quad A; c; \Gamma \vdash t_2 : \langle \Delta, \Delta_1 \mid T \rangle \end{array}}{A; c; \Gamma \vdash t_1 \succ t_2 : \langle \Delta', \Delta_1 \mid T \rangle} \quad \begin{array}{l} (\Delta_1 = \emptyset \text{ or } \nu = \circ) \text{ and} \\ dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \end{array}$$

$$\text{(T-RENAME)}\ \frac{A; c \vdash \sigma_1 \ \texttt{OK} \quad A; c \vdash \sigma_2 \ \texttt{OK} \quad A; c \vdash \sigma_1 \circ \Delta_1 \ \texttt{OK} \quad A; c; \Gamma \vdash t : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}}{A; c; \Gamma \vdash \sigma_1 \ltimes t \rtimes \sigma_2 : \langle \sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2 \rangle^\circ}$$

$ctx(x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m) = x_1{:}T_1, \ldots, x_m{:}T_m$

$name_ctx(x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m) = X_1{:}T_1, \ldots, X_m{:}T_m$

$$\sigma \circ \Delta = \begin{cases} \Delta' & \text{if } dom(\Delta) \subseteq dom(\sigma) \\ & \quad \text{and for all } X, T\ X{:}T \in \Delta' \text{ iff } \exists Y\ Y{:}T \in \Delta \wedge \sigma(Y) = X \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta \circ \sigma = \begin{cases} \Delta' & \text{if } rng(\sigma) \subseteq dom(\Delta) \\ & \quad \text{and for all } X, T\ X{:}T \in \Delta' \text{ iff } X \in dom(\sigma) \wedge T = \Delta(\sigma(X)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Figure 7 Typed calculus: typing rules

In rule (T-OVER), overriding $t_1 \lhd t_2$ is well-typed only if t_1 and t_2 have rebinding types; the name context of the type of t_1 is deterministically split in two parts. The part Δ_1' corresponds to names which are also defined in t_2, as expressed by the side condition $dom(\Delta_1') \subseteq dom(\Delta_2)$, hence are overridden, whereas the part Δ_1 corresponds to names which are not defined in t_2. If $\Delta_1 = \emptyset$, then t_1 is fully overridden, hence the name context of the result is that of t_2; in this particular case the type of t_2 is allowed to be open, whereas if $\Delta_1 \neq \emptyset$, then t_2 is required to have a closed type, otherwise it would not be possible to correctly identify Δ_1.

The previously defined operator $\sqcup$ combines the two annotations ν_1 and ν_2 so that the resulting type is closed if and only if both types of t_1 and t_2 are closed.

Note that, due to the presence of name variables, besides names which are necessarily overridden, there are names which *could* be overridden in some instantiation. For instance, in the term $\Lambda\alpha{:}\alpha \neq N_1.\langle \mid N_1{:}T_1 \mapsto t_1, N_2{:}T_2 \mapsto t_2\rangle \lhd \langle \mid \alpha{:}\mathtt{int} \mapsto 1\rangle$, the name N_1 is never overridden, whereas the name N_2 could be overridden for $\alpha = N_2$. The name context which is assigned to the overriding term is that corresponding to the case of no overriding, that is, $N_1{:}T_1, N_2{:}T_2, \alpha{:}\mathtt{int}$ in this case. However, since this name context must be well-formed under the constraints $\alpha \neq N_1$, the type T_2 must necessarily be `int`, so that we get a well-formed type even for the instantiation $\alpha = N_2$.

Rule (T-Run) states that a term of unbound type can be safely run only if its name context is empty, that is, all variables have been already properly bound in the code.

The typing rule (T-Reb-App) for rebinding application $t_1 \succ t_2$ is similar to the typing rule for overriding: to correctly identify the names in t_1 that are not necessarily bound, denoted by Δ_1, the rule requires an exact type for t_2, except when $\Delta_1 = \emptyset$ (that is, all names are bound) for which an open type is allowed as well. This is due to the fact that the bound names of t_1 must have the same type of the corresponding names in t_2, while additional names in t_2 not specified in the open type of t_2 might be used for binding names of t_1 with incompatible types. Note that by applying subsumption, it is always possible to bind a name with a term whose type is a subtype of the expected type.

Finally, in rule (T-Rename) for renaming, the two renamings must be well-formed under current name variables and constraints, that is, the newly introduced names must be existing, and names which could be equal are mapped in the same name. The name contexts of the resulting type are propagated from the original ones by the auxiliary operators $\sigma \circ \Delta$ and $\Delta \circ \sigma$, both partial, defined at the bottom of Figure 7. Note that if two names X and Y are mapped by σ_1 in two names which could be equal, then X and Y must have the same type, as formally expressed by requiring the well-formedness of the name context $\sigma_1 \circ \Delta_1$.

4 Examples of typing

In this section we give some examples of type derivations. At the end, we present a name abstraction term showing that: if constraint annotations are removed from name abstractions, then there is not a "more general" way to infer such constraints in order to make the term well typed.

For the typing derivations, we assume that our language supports the primitive types of integers and booleans with their standard operators, semantics and typing. Moreover, we assume to have the constructs `let` and `if then else` with the standard operational semantics and typing rules. In particular, the `let` construct is typed as follows.

$$\text{(T-Let)}\ \frac{A;c;\Gamma \vdash t' : T' \quad A;c;\Gamma[x{:}T'] \vdash t : T \quad A;c \vdash T'\ \mathtt{OK}}{A;c;\Gamma \vdash \mathtt{let}\ x{:}T' = t'\ \mathtt{in}\ t : T}$$

Let t_m be the typed version of the mixin adaptation term defined at the end of Section 2:

$$\Lambda\alpha_{op}{:}\emptyset.\Lambda\alpha_{in_b}{:}c_i.\Lambda\alpha_{n_op}{:}c_o.\lambda r{:}T_r.\mathtt{let}\ n_op{:}T_1 = !(r \succ \langle u_n|t_n\rangle)\ \mathtt{in}\ r \lhd \langle \mid \alpha_{n_op}{:}T_1 \mapsto n_op\rangle$$

where

- $T_1 = \mathtt{int} \to \mathtt{int}$
- $T_2 = \mathtt{int} \to \mathtt{bool}$
- $c_i = \alpha_{in_b} \neq \alpha_{op}$
- $c_o = \alpha_{n_op} \neq \alpha_{op}, \alpha_{n_op} \neq \alpha_{in_b}$
- $T_r = \langle \mid \alpha_{op}{:}T_1, \alpha_{in_b}{:}T_2\rangle^+$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_1 \quad A; c_i, c_o \vdash T_r\ \texttt{OK}}{A; c_i, c_o; \emptyset \vdash \lambda r{:}T_r.t_l : T_r \to T_{n_r}}\,\text{(T-Abs)} \quad \begin{array}{c}\alpha_{n_op} \Vdash c_o \\ A \vdash c_o\end{array}}{\{\alpha_{op}, \alpha_{in_b}\}; c_i; \emptyset \vdash \Lambda\{\alpha_{n_op}\}{:}c_o.\lambda r{:}T_r.t_l : \forall\{\alpha_{n_op}\}{:}c_o.T_r \to T_{n_r}}\,\text{(T-Nm-Abs)} \quad \begin{array}{c}\alpha_{in_b} \Vdash c_i \\ \{\alpha_{op}, \alpha_{in_b}\} \vdash c_i\end{array}}{\{\alpha_{op}\}; \emptyset; \emptyset \vdash \Lambda\alpha_{in_b}{:}c_i.\Lambda\alpha_{n_op}{:}c_o.\lambda r{:}T_r.t_l : \forall\alpha_{in_b}{:}c_i.\forall\alpha_{n_op}{:}c_o.T_r \to T_{n_r}}\,\text{(T-Nm-Abs)} \quad \begin{array}{c}\{\alpha_{op}\} \vdash \emptyset \quad \alpha_{op} \Vdash \emptyset\end{array}}{\emptyset; \emptyset; \emptyset \vdash \Lambda\alpha_{op}{:}\emptyset.\Lambda\alpha_{in_b}{:}c_i.\Lambda\alpha_{n_op}{:}c_o.\lambda r{:}T_r.t_l : \forall\alpha_{op}{:}\emptyset.\forall\alpha_{in_b}{:}c_i.\forall\alpha_{n_op}{:}c_o.T_r \to T_{n_r}}\,\text{(T-Nm-Abs)}$$

- $T_{n_r} = \langle\ |\ \alpha_{op}{:}T_1, \alpha_{in_b}{:}T_2, \alpha_{n_op}{:}T_1\rangle^+$
- $t_l = \texttt{let}\ n_op{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \texttt{in}\ r \lhd \langle\ |\alpha_{n_op}{:}T_1 \mapsto n_op\rangle$
- $A = \{\alpha_{op}, \alpha_{in_b}, \alpha_{n_op}\}$

Figure 8 Typing derivation for the term t

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A; c_i, c_o; r{:}T_r \vdash r : T_r}\,\text{(T-Var)}}{A; c_i, c_o; r{:}T_r \vdash r \succ \langle u_n | t_n\rangle : \langle\ |T_1\rangle}\,\text{(T-Reb-App)} \quad \mathcal{D}_3}{A; c_i, c_o; r{:}T_r \vdash !(r \succ \langle u_n | t_n\rangle) : T_1}\,\text{(T-Run)} \quad \mathcal{D}_2}{A; c_i, c_o; r{:}T_r \vdash \texttt{let}\ n_op{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \texttt{in}\ r \lhd \langle\ |\alpha_{n_op}{:}T_1 \mapsto n_op\rangle : T_{n_r}}\,\text{(T-Let)}$$

Figure 9 Typing derivation $\mathcal{D}_1$

- $t_n = \lambda x{:}\texttt{int}.\texttt{if}\ (\texttt{not}\,(in_b\,x))\ \texttt{then}\ {-1}\ \texttt{else}\ (op\,x)$
- $u_n = op{:}T_1 \mapsto \alpha_{op}, in_b{:}T_2 \mapsto \alpha_{in_b}$

In Figures 8÷11 we give the derivation of $\emptyset; \emptyset; \emptyset \vdash t_m : \forall\alpha_{op}{:}\emptyset.\forall\alpha_{in_b}{:}c_i.\forall\alpha_{n_op}{:}c_o.T_r \to T_{n_r}$

With rule (T-Name-App), since there are no constraints on α_{op}, we can derive

$$\emptyset; \emptyset; \emptyset \vdash t_m\ N_1 : \forall\alpha_{in_b}{:}c_i.(\forall\alpha_{n_op}{:}c_o.T_r \to T_{n_r})\{\alpha_{op} \mapsto N_1\}$$

for any name constants N_1. Note that we cannot substitute a name variable, since the name environment is empty. Again with rule (T-Name-App), we can derive

$$\emptyset; \emptyset; \emptyset \vdash (t_m\ N_1)\ N_2 : (\forall\alpha_{n_op}{:}c_o.T_r \to T_{n_r})\{\alpha_{op} \mapsto N_1, \alpha_{in_b} \mapsto N_2\}$$

In this case the name constant N_2 must be such that $\emptyset \vdash (\alpha_{in_b} \neq N_1)\{\alpha_{in_b} \mapsto N_2\}$. Finally, we can derive

$$\emptyset; \emptyset; \emptyset \vdash ((t_m\ N_1)\ N_2)\ N_3 : (T_r \to T_{n_r})\{\alpha_{op} \mapsto N_1, \alpha_{in_b} \mapsto N_2 \alpha_{op} \mapsto N_3\}$$

where the name constant N_3 must be such that $\emptyset \vdash (\alpha_{n_op} \neq N_1, \alpha_{n_op} \neq N_2)\{\alpha_{n_op} \mapsto N_3\}$.

The typing derivation above is valid also if the constraint $\alpha_{n_op} \neq \alpha_{op}$ is removed from c_o; indeed, rule (T-Over) can be correctly applied also when $\alpha_{n_op} \neq \alpha_{op}$ is not derivable, because α_{n_op} and α_{op} are associated with the same type. However, without the constraint $\alpha_{n_op} \neq \alpha_{op}$ the user is allowed to override α_{op} with α_{n_op}.

Note that, if we remove from the rule (T-Name-Abs) the condition that the constraint must refer to the name variable introduced, we could give type
$\forall\alpha_{op}{:}\emptyset.\forall\alpha_{in_b}{:}\emptyset.\forall\alpha_{n_op}{:}c_i, c_o.T_r \to T_{n_r}$ to the following term t'_m

$$\Lambda\alpha_{op}{:}\emptyset.\Lambda\alpha_{in_b}{:}\emptyset.\Lambda\alpha_{n_op}{:}c_i, c_o.\lambda r{:}T_r.\texttt{let}\ n_op{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \texttt{in}\ r \lhd \langle\ |\alpha_{n_op}{:}T_1 \mapsto n_op\rangle$$

From this using (T-Name-App) twice we could get

$$\emptyset; \emptyset; \emptyset \vdash (t'_m\ N)\ N : (\forall\alpha_{n_op}{:}c_i, c_o.T_r \to T_{n_r})\{\alpha_{op} \mapsto N, \alpha_{in_b} \mapsto N\}$$

which is, however, a stuck term, since its subterm $u_n\{\alpha_{op} \mapsto N, \alpha_{in_b} \mapsto N\} = op{:}T_1 \mapsto N, in_b{:}T_2 \mapsto N$ is not well-formed, and so cannot reduce with rule (Name-App) of Figure 1.

$$\dfrac{\dfrac{}{A;c;\Gamma\vdash n_op:T_1}\,(\text{T-Var}) \quad \dfrac{\dfrac{A;c\vdash\langle\ |\ \alpha_{n_op}{:}T_1\rangle^\circ\ \texttt{OK}}{A;c\vdash\langle|\alpha_{n_op}{:}T_1\mapsto n_op\rangle\ \texttt{OK}}}{A;c;\Gamma\vdash\langle|\alpha_{n_op}{:}T_1\mapsto n_op\rangle:\langle\ |\ \alpha_{n_op}{:}T_1\rangle^\circ}\,(\text{T-Reb}) \quad A;c\vdash\Delta\ \texttt{OK} \quad \dfrac{}{A;c;\Gamma\vdash r:T_r}\,(\text{T-Var})}{A;c;\Gamma\vdash r\lhd\langle\ |\alpha_{n_op}{:}T_1\mapsto n_op\rangle:T_{n_r}}\,(\text{T-Over})$$

- $c=c_i,c_o$
- $\Gamma=r{:}T_r,n_op{:}T_1$
- $\Delta=\alpha_{op}{:}T_1,\alpha_{in_b}{:}T_2,\alpha_{n_op}{:}T_1$

Figure 10 Typing derivation $\mathcal{D}_2$

$$\dfrac{\dfrac{\dfrac{\vdots}{A;c;\Gamma'\vdash \texttt{if}\ (\texttt{not}\,(in_b\,x))\ \texttt{then}\ -1\ \texttt{else}\ (op\,x):\texttt{int}}\,(\text{T-If})}{A;c;r{:}T_r,op{:}T_1,in_b{:}T_2\vdash t_n:T_1}\,(\text{T-Abs}) \quad A;c\vdash\alpha_{op}{:}T_1,\alpha_{in_b}{:}T_2\ \texttt{OK}}{A;c;r{:}T_r\vdash\langle u_n|t_n\rangle:\langle\alpha_{op}{:}T_1,\alpha_{in_b}{:}T_2\ |\ T_1\rangle}\,(\text{T-Unb})$$

- $\Gamma'=r{:}T_r,op{:}T_1,in_b{:}T_2,x{:}\texttt{int}$

Figure 11 Typing derivation $\mathcal{D}_3$

Exploring the possibility of inferring constraints on name variables, rather than explicitly annotating name abstractions, is a more challenging research topic, since the type system does not enjoy principality if constraint annotations are removed from name abstractions. To see this, let us consider the following term:

$$t=\Lambda\alpha{:}c.(\langle\ |\alpha{:}T_1\mapsto t_1\rangle\lhd\langle\ |N{:}T_2\mapsto t_2\rangle)$$

where

$$T_1=\langle\ \ |\ N_0{:}\texttt{int},N_1{:}\texttt{int}\rangle^\circ \qquad t_1=\langle\ |N_0{:}\texttt{int}\mapsto 0,N_1{:}\texttt{int}\mapsto 1\rangle$$
$$T_2=\langle\ \ |\ N_0{:}\texttt{int}\rangle^\circ \qquad t_2=\langle\ |N_0{:}\texttt{int}\mapsto 42\rangle$$

and N, N_1 and N_2 are distinct names.

One may think that the name abstraction t can be correctly typed only if c contains the constraint $\alpha{\neq}N$; indeed, if $c=\alpha{\neq}N$, then it is possible to derive the typing judgment $\emptyset;\emptyset;\emptyset\vdash t:\forall\alpha{:}\alpha{\neq}N.\langle\ \ |\ \alpha{:}T_1,N{:}T_2\rangle^\circ$.

However, the following different typing judgment can be derived if $c=\emptyset$: $\emptyset;\emptyset;\emptyset\vdash t:\forall\alpha{:}\emptyset.\langle\ \ |\ \alpha{:}T,N{:}T\rangle^\circ$, with $T=\langle\ \ |\ N_0{:}\texttt{int}\rangle^+$; this is possible thanks to the subsumption rule, and to the fact that T_1 and T_2 are both subtypes of T.

Surprisingly, neither of the typings above is "better" than the other, because the two types associated with t are not comparable; indeed, both $\vdash\langle\ \ |\ \alpha{:}T_1,N{:}T_2\rangle^\circ\leq\langle\ \ |\ \alpha{:}T,N{:}T\rangle^\circ$ and $\alpha{\neq}N\vdash\emptyset$ are derivable.

5 Results

First of all, we define consistency for a name substitution w.r.t. a set of constraints and prove that applying a consistent name substitution to a well-formed element (type, name context, rebinding, or renaming) produces a well-formed element.

▶ **Definition 1.** Let A and c be such that $A\vdash c$. A *name substitution* $\alpha\mapsto N$ *is consistent with* A *and* c if $\alpha\in A$ and $A{-}\{\alpha\}\vdash c\{\alpha\mapsto N\}$.

▶ **Lemma 2.** *Let A and c be such that $A\vdash c$, and let $\alpha\mapsto N$ be consistent with A and c. Let γ be T, Δ, r, or σ. If $A;c\vdash\gamma$ OK, then $A{-}\{\alpha\};c\{\alpha\mapsto N\}\vdash\gamma\{\alpha\mapsto N\}$ OK.*

Proof. By induction on the derivation of $A; c \vdash \gamma$ `OK` and case analysis on the last applied rule.

- If the last applied rule is (WF-NAME-ARROW-TYPE), then $\gamma = \forall\alpha{:}c'.T$,
  1. $A \cup \{\alpha'\} \vdash c'$,
  2. $\alpha' \Vdash c'$
  3. $A \cup \{\alpha'\}; c, c' \vdash T$ `OK`, and
  4. $\alpha' \notin A$.

  To apply the induction hypothesis on 3. we need to establish that $A \cup \{\alpha'\} \vdash c, c'$ and that $\alpha \mapsto N$ is consistent with $A \cup \{\alpha'\}$ and c, c'.
  From the assumption $A \vdash c$ and 1. we have $A \cup \{\alpha'\} \vdash c, c'$.
  From the assumption $\alpha \mapsto N$ consistent with A and c, we have that $A \vdash c\{\alpha \mapsto N\}$. Moreover, from 4., we get that $\alpha' \neq \alpha$. From 2., if α occurs in c' it can only be in a constraint $\alpha' \neq \alpha$ or $\alpha \neq \alpha'$. So from $N \neq \alpha'$ we have
  1'. $A \cup \{\alpha'\} \vdash c'\{\alpha \mapsto N\}$
  and $A \cup \{\alpha'\} \vdash (c, c')\{\alpha \mapsto N\}$. Therefore $\alpha \mapsto N$ is consistent with $A \cup \{\alpha'\}$ and c, c'.
  By induction hypothesis on 3. we get
  3'. $(A \cup \{\alpha'\}) - \{\alpha\}; (c, c')\{\alpha \mapsto N\} \vdash T\{\alpha \mapsto N\}$ `OK`.
  From 1'. since α does not occur in $c'\{\alpha \mapsto N\}$ we derive that $(A \cup \{\alpha'\}) - \{\alpha\} \vdash c'\{\alpha \mapsto N\}$.
  Therefore, from 2., 4. and 3'. applying rule (WF-NAME-ARROW-TYPE) we get

  $$A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash X_1{:}T_1, \ldots, X_n{:}T_m\{\alpha \mapsto N\}\ \texttt{OK}.$$

- If the last applied rule is (WF-NAME-CTX), then $\gamma = X_1{:}T_1, \ldots, X_n{:}T_m$
  1. $A; c \vdash T_k$ `OK` $(1 \leq k \leq m)$,
  2. $A \vdash X_k (1 \leq k \leq m)$, and
  3. $c \models X_i \stackrel{?}{=} X_j \Rightarrow T_i = T_j\ (1 \leq i, j \leq m)$

  From the assumptions $A \vdash c$ and $\alpha \mapsto N$ consistent with A and c, by induction hypotheses on 1., we have that
  1'. $A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash T_k\{\alpha \mapsto N\}$ `OK` $(1 \leq k \leq m)$.
  Let $\alpha = X_k$ for some k, $1 \leq k \leq m$. From $\alpha \mapsto N$ consistent with A and c we derive that $\alpha \neq N \notin c$ and therefore $c \models \alpha \stackrel{?}{=} N$. If $c \models X_j \stackrel{?}{=} N$ for some j, $1 \leq j \leq m$, then $c \models X_j \stackrel{?}{=} \alpha$, and, from 3., we have that $T_j = T_k$, which implies $T_j\{\alpha \mapsto N\} = T_k\{\alpha \mapsto N\}$. Therefore
  3'. $c\{\alpha \mapsto N\} \models X_i\{\alpha \mapsto N\} \stackrel{?}{=} X_j\{\alpha \mapsto N\} \Rightarrow T_i\{\alpha \mapsto N\} = T_j\{\alpha \mapsto N\}$ $(1 \leq i, j \leq m)$
  It is immediate to see that, 3'. holds also for $\alpha \notin \{X_1, \ldots X_m\}$. From 2. we derive that
  2'. $A - \{\alpha\} \vdash X_k\{\alpha \mapsto N\} (1 \leq k \leq m)$.
  Therefore, from 1'., 2'. and 3'., applying rule (WF-NAME-CTX) we derive

  $$A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash \forall\alpha{:}c'.T\{\alpha \mapsto N\}\ \texttt{OK}.$$

- If the last applied rule is (WF-REB-MAP) or (WF-REN) the proof is similar to the previous one.
- If the last applied rule is (WF-ARROW-TYPE), (WF-UNB-TYPE) or (WF-REB-TYPE), the result follows by induction hypotheses on the premises of the rules. ◀

The previous result may be proved also for terms.

▶ **Lemma 3.** *Let A and c be such that $A \vdash c$, and let $\alpha \mapsto N$ be consistent with A and c. If $A; c \vdash t$ `OK`, then $A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash t\{\alpha \mapsto N\}$ `OK`.*

Proof. Easy, using Lemma 2. ◀

▶ **Lemma 4** (Transitivity of $\leq$).
1. *If* $\vdash \Delta \leq \Delta'$ *and* $\Delta' \leq \Delta''$, *then* $\Delta \leq \Delta''$.
2. *If* $\vdash T \leq T'$ *and* $\vdash T' \leq T''$, *then* $\vdash T \leq T''$.

Proof. The two results are proved by simultaneous induction on derivations, considering the rules of Figure 5.

1. If $\vdash \Delta \leq \Delta'$, and $\vdash \Delta' \leq \Delta''$, then, in both cases, the last applied rule is (Sub-name-ctx). Let $\Delta = X_1{:}T_1, \ldots, X_m{:}T_m$, $\Delta' = X_1'{:}T_1', \ldots, X_n'{:}T_n'$ and $\Delta'' = X_1''{:}T_1'', \ldots, X_p''{:}T_p''$. For all X_i'', $1 \leq i \leq p$, from $\vdash \Delta' \leq \Delta''$, there is X_j', $1 \leq j \leq n$, such that $X_i'' = X_j'$ and $\vdash T_j' \leq T_i''$. Moreover, from $\vdash \Delta \leq \Delta'$, there is X_k, $1 \leq k \leq m$, such that $X_k = X_j'$ and and $\vdash T_k \leq T_j'$. Applying the inductions hypotheses 2. to $\vdash T_j' \leq T_i''$ and $\vdash T_k \leq T_j'$ we have that $\vdash T_k \leq T_i''$. Therefore, from (Sub-name-ctx) we have that $\Delta \leq \Delta''$.
2. By cases on the last applied rule in the derivation of $\vdash T \leq T'$.
   - If the rule is (Sub-Arr), then $T = T_1 \rightarrow T_2$, $T' = T_1' \rightarrow T_2'$, $\vdash T_1' \leq T_1$, and $\vdash T_2 \leq T_2'$. Since $T' = T_1' \rightarrow T_2'$, the last applied rule in the derivation of $\vdash T' \leq T''$ must be (Sub-Arr). Therefore, $T'' = T_1'' \rightarrow T_2''$, $\vdash T_1'' \leq T_1'$, and $\vdash T_2' \leq T_2''$. By induction hypotheses on $\vdash T_1' \leq T_1$ and $\vdash T_1'' \leq T_1'$, we derive that $\vdash T_1'' \leq T_1$, and by induction hypotheses on $\vdash T_2 \leq T_2'$ and $\vdash T_2' \leq T_2''$, we get $\vdash T_2 \leq T_2''$. Therefore, from rule (Sub-Arr), we have that $\vdash T \leq T''$.
   - Similarly if the rule is (Sub-unb) or (Sub-open-reb). The inductive hypotheses are on name contexts and types.
   - If the rule is (Sub-closed-reb), then $T = \langle \Delta_1 \mid \Delta_2 \rangle^{\circ}$, $T' = \langle \Delta_1' \mid \Delta_2' \rangle^{\circ}$, $dom(\Delta_2) = dom(\Delta_2')$, $\vdash \Delta_1' \leq \Delta_1$, and $\vdash \Delta_2 \leq \Delta_2'$. There are two cases: either the last applied rule in the derivation of $\vdash T' \leq T''$ is (Sub-closed-reb), or is (Sub-open-reb). In the first case, $T'' = \langle \Delta_1'' \mid \Delta_2'' \rangle^{\circ}$, and by inductive hypotheses we derive $\vdash T \leq T''$ applying rule (Sub-closed-reb). In the second case, $T'' = \langle \Delta_1'' \mid \Delta_2'' \rangle^{+}$, and by inductive hypotheses we derive $\vdash T \leq T''$ applying rule (Sub-open-reb).
   - If the rule is (Sub-name-arr), then $T = \forall\alpha{:}c_1.T_1$, $T' = \forall\alpha{:}c_2.T_2$, $\vdash T_1 \leq T_2$, and $c, c_2 \vdash c_1$. Since $T' = \forall\alpha{:}c_2.T_2$, the last applied rule in the derivation of $\vdash T' \leq T''$ must be (Sub-name-arr). Therefore, $T'' = \forall\alpha{:}c_3.T_3$, $\vdash T_2 \leq T_3$, and $c, c_3 \vdash c_2$. From $c, c_2 \vdash c_1$ and $c, c_3 \vdash c_2$ we get $c, c_3 \vdash c_1$. By induction hypotheses on $\vdash T_1 \leq T_2$ and $\vdash T_2 \leq T_3$ we get $\vdash T_1 \leq T_3$. Therefore, from rule (Sub-name-arr), we have that $\vdash T \leq T''$. ◀

Well-typed terms are also well-formed and their type is well-formed.

▶ **Lemma 5.** *Let* A, c *and* Γ *be such that:* $A \vdash c$ *and for all* $x{:}T' \in \Gamma$, *we have that* $A; c \vdash T'$ `OK`. *If* $A; c; \Gamma \vdash t : T$, *then* $A; c \vdash T$ `OK` *and* $A; c \vdash t$ `OK`.

Proof. By induction on the type derivation. ◀

Soundness of the type system w.r.t. the operational semantics states that *well-typed terms do not get stuck*. This is derived from the *subject reduction* and *progress* properties. To prove this properties we first need to introduce some lemmas.

▶ **Lemma 6** (Inversion).
1. *If* $A; c; \Gamma \vdash x : T$, *then* $A; c \models \Gamma(x) \leq T$.
2. *If* $A; c; \Gamma \vdash \lambda x{:}T_1.t : T$, *then for some* T_2 *we have that:*
   (a) $\vdash T_1 \rightarrow T_2 \leq T$, *and*
   (b) $A; c; \Gamma[x{:}T_1] \vdash t : T_2$.
3. *If* $A; c; \Gamma \vdash \Lambda\alpha'{:}c'.t : T$, *then for some* T' *we have that:*

   (a) $\vdash \forall\alpha'{:}c'.T' \leq T$,
   (b) $A \cup \{\alpha'\}; c, c'; \Gamma \vdash t : T'$, *and*
   (c) $\alpha' \Vdash c'$.
4. *If* $A; c; \Gamma \vdash t_1\ t_2 : T$, *then then for some* T_1 *and* T_2 *we have that:*
   (a) $\vdash T_2 \leq T$,
   (b) $A; c; \Gamma \vdash t_1 : T_1 \to T_2$, *and*
   (c) $A; c; \Gamma \vdash t_2 : T_1$.
5. *If* $A; c; \Gamma \vdash \langle u \mid t\rangle : T$, *then for some* T' *we have that:*
   (a) $\vdash \langle name_ctx(u) \mid T'\rangle \leq T$,
   (b) $A; c; \Gamma[ctx(u)] \vdash t : T'$, *and*
   (c) $A; c \vdash name_ctx(u)\ \mathtt{OK}$.
6. *If* $A; c; \Gamma \vdash \langle u \mid X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m\rangle : T$, *let* $\Delta_1 = name_ctx(u)$ *and* $\Delta_2 = X_1{:}T_1, \ldots, X_m{:}T_m$, *we have that:*
   (a) $\vdash \langle \Delta_1 \mid \Delta_2\rangle^\circ \leq T$,
   (b) $\Gamma[ctx(u)] \vdash t_i : T_i \quad (1 \leq i \leq m)$,
   (c) $A; c \vdash X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\ \mathtt{OK}$, *and*
   (d) $A; c \vdash \langle \Delta_1 \mid \Delta_2\rangle^\circ\ \mathtt{OK}$.
7. *If* $A; c; \Gamma \vdash t\ X : T$, *then for some* T' *and* c' *we have that:*
   (a) $\vdash T'\{\alpha \mapsto X\} \leq T$,
   (b) $A; c; \Gamma \vdash t : \forall\alpha{:}c'.T'$,
   (c) $A \vdash c'\{\alpha \mapsto X\}$ *and* $A \vdash X$.
8. *If* $A; c; \Gamma \vdash !t : T$, *then for some* T' *we have that:*
   (a) $\vdash T' \leq T$, *and*
   (b) $A; c; \Gamma \vdash t : \langle\ \mid T'\rangle$.
9. *If* $A; c; \Gamma \vdash t_1 \lhd t_2 : T$, *then for some* Δ, Δ^*, *and* ν *we have that:*
   α. $\vdash \langle \Delta \mid \Delta^*\rangle^\nu \leq T$,
   β. $A; c \vdash \Delta^*\ \mathtt{OK}$, *and*
   (a) *either for some* Δ_1', ν_1, *and* ν_2 *we have that:*
      (i) $\nu = \nu_1 \sqcup \nu_2$,
      (ii) $A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta^*\rangle^{\nu_2}$,
      (iii) $A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta_1'\rangle^{\nu_1}$, *and*
      (iv) $dom(\Delta_1') \subseteq dom(\Delta^*)$;
   (b) *or for some* Δ_1, Δ_2, Δ_1' *we have that:*
      (i) $\Delta^* = \Delta_1, \Delta_2$ $(dom(\Delta_1) \cap dom(\Delta_2) = \emptyset)$,
      (ii) $A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta_2\rangle^\circ$,
      (iii) $A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta_1, \Delta_1'\rangle^\nu$ $(dom(\Delta_1) \cap dom(\Delta_1') = \emptyset)$, *and*
      (iv) $dom(\Delta_1') \subseteq dom(\Delta_2)$.
10. *If* $A; c; \Gamma \vdash t_1 \succ t_2 : T$, *then for some* Δ_1, Δ_2, Δ', *and* T' *we have that:*
   α. $\vdash \langle \Delta', \Delta_1 \mid T'\rangle \leq T$,
   β. $A; c \vdash \Delta_1, \Delta_2\ \mathtt{OK}$, *and*
   (a) *either* $\Delta_1 = \emptyset$, *and for some* ν *we have that:*
      (i) $A; c; \Gamma \vdash t_2 : \langle \Delta \mid T'\rangle$,
      (ii) $A; c; \Gamma \vdash t_1 : \langle \Delta' \mid \Delta, \Delta_2\rangle^\nu$ $(dom(\Delta) \cap dom(\Delta_2) = \emptyset)$;
   (b) *or we have that:*
      (i) $A; c; \Gamma \vdash t_2 : \langle \Delta, \Delta_1 \mid T'\rangle$,
      (ii) $A; c; \Gamma \vdash t_1 : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2\rangle^\circ$ $(dom(\Delta) \cap dom(\Delta_2) = \emptyset)$, *and*
      (iii) $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$.

11. *If* $A; c; \Gamma \vdash \sigma_1 \ltimes t \rtimes \sigma_2 : T$*, then for some* Δ_1 *and* Δ_2 *we have that:*
  (a) $\sigma_1 \circ \Delta_1$ *and* $\Delta_2 \circ \sigma_2$ *are defined,*
  (b) $\vdash \langle \sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2 \rangle^{\circ} \leq T$*,*
  (c) $A; c; \Gamma \vdash t : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}$ *for some* ν*,*
  (d) $A; c \vdash \sigma_1$ `OK` *and* $A; c \vdash \sigma_2$ `OK`*, and*
  (e) $A; c \vdash \sigma_1 \circ \Delta_1$ `OK`*.*

Proof. By induction on typing derivations. For each case, we have that either the last applied rule in the derivation of $A; c; \Gamma \vdash t : T$ is the typing rule corresponding to the syntactic construct t, or rule (T-Sub). In the latter case, from Lemma 4, we get that, for some T', such that $\vdash T' \leq T$, $A; c; \Gamma \vdash t : T'$ is a derivation in which the last applied rule is the one corresponding to the syntactic construct t. The result then follows by case analysis on the structural rules. ◀

▶ **Lemma 7** (Substitution). *If* $A; c; \Gamma[x_1{:}T_1, \ldots, x_m{:}T_m] \vdash t : T$*, and* $A; c; \Gamma \vdash t_i : T'_i$ $(1 \leq i \leq m)$ *where* $\vdash T'_i \leq T_i$ $(1 \leq i \leq m)$*, then* $A; c; \Gamma \vdash t\{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} : T$*.*

Proof. By induction on terms. ◀

▶ **Lemma 8** (Name Substitution). *If* $A \cup \{\alpha\}; c; \Gamma \vdash t : T$*, and* $\alpha \mapsto N$ *is consistent with* $A \cup \{\alpha\}$ *and* c*, then* $A; c\{\alpha \mapsto N\}; \Gamma\{\alpha \mapsto N\} \vdash t\{\alpha \mapsto N\} : T\{\alpha \mapsto N\}$*.*

Proof. By induction on terms. Most cases are by induction hypotheses on the antecedent of the type rule using Lemma 2. We consider only the most interesting case, which is (T-Name-Abs). ◀

▶ **Lemma 9** (Context). *Let* $A; c; \Gamma \vdash \mathcal{E}[t] : T$*, then*
- $A; c; \Gamma \vdash t : T'$ *for some* T'*, and*
- *if* $A; c; \Gamma \vdash t' : T'$*, then* $\Gamma \vdash \mathcal{E}[t'] : T$*, for all* t'*.*

Proof. By induction on evaluation contexts $\mathcal{E}$. ◀

▶ **Definition 10.** Let A, c, and $\Delta = X'_1{:}T'_1, \ldots X'_m{:}T'_m$ be such that $A; c \vdash \Delta$ `OK`.
1. Define $unb(\Delta, A, c) = \{u \mid u = x_1{:}T_1 \mapsto X_1, \ldots, x_n{:}T_n \mapsto X_n \ \wedge\ \forall i\ 1 \leq i \leq m\ \exists j\ 1 \leq j \leq n\ \ X'_i = X_j \wedge\ \vdash T_j \leq T'_i\}$.
2. Let Γ be such that for all $x{:}T' \in \Gamma$, we have that $A; c \vdash T'$ `OK`.
   a. Define $reb(\Delta, A, c, \Gamma)^{+} = \{r \mid r = X_1{:}T_1 \mapsto t_1, \ldots, X_n{:}T_n \mapsto t_n \wedge \forall i\ 1 \leq i \leq m\ \exists j\ 1 \leq j \leq n\ \ X'_i = X_j \wedge\ \vdash T_j \leq T'_i\ \wedge\ A; c; \Gamma \vdash t_j : T_j\ (1 \leq j \leq n)\}$.
   b. Define $reb(\Delta, A, c, \Gamma)^{\circ} = \{r \mid r \in reb(\Delta, A, c, \Gamma)^{+}\ \wedge\ dom(r) = dom(\Delta)\}$.

From the definition it is immediate to see that: if $u \in unb(\Delta, A, c)$ then $\vdash \Delta \leq name_ctx(u)$, and if $r \in reb(\Delta, A, c, \Gamma)$ then $\vdash name_ctx(r) \leq \Delta$. Also, if for some ν, $r \in reb(\Delta, A, c, \Gamma)^{\nu}$, then $r \in reb(\Delta, A, c, \Gamma)^{\nu \sqcup \nu'}$ for all ν'.

▶ **Theorem 11** (Subject Reduction). *Let* A*,* c *and* Γ *be such that:* $A \vdash c$ *and for all* $x{:}T' \in \Gamma$*, we have that* $A; c \vdash T'$ `OK`*. Let* t *be such that, for some* T *we have* $A; c; \Gamma \vdash t : T$*. If* $t \longrightarrow t'$*, then* $A; c; \Gamma \vdash t' : T$*.*

Proof. By case analysis on the rule used for $t \longrightarrow t'$. We consider only rules (Ctx), (Reb-App), and (Name-App), which are the most interesting.
- If the applied rule is (Ctx), then $t = \mathcal{E}[t_1]$, $t_1 \longrightarrow t'_1$, and $t' = \mathcal{E}[t'_1]$. From Lemma 9 for some T', we have that $A; c; \Gamma \vdash t_1 : T'$. From induction hypothesis on t_1, we derive that $A; c; \Gamma \vdash t'_1 : T'$, and therefore, again by Lemma 9, $A; c; \Gamma \vdash \mathcal{E}[t'_1] : T$.

- If the applied rule is (Name-App), then $t = (\Lambda\alpha{:}c.t'')\ N$, $t' = t''\{\alpha \mapsto N\}$, and $\emptyset;\emptyset \vdash t'$ OK. Therefore, $\emptyset;\emptyset;\Gamma \vdash t : T$, and we can assume that α does not occur neither in Γ nor in T. From Lemma 6.7 for some T' and c' we have that:
  1. $\vdash T'\{\alpha \mapsto N\} \leq T$,
  2. $\emptyset;\emptyset;\Gamma \vdash \Lambda\alpha{:}c.t'' : \forall\alpha{:}c'.T'$,
  3. $\emptyset \vdash c'\{\alpha \mapsto N\}$.
  From 2. and Lemma 6.3, for some T'' we have that
  4. $\vdash \forall\alpha{:}c.T'' \leq \forall\alpha{:}c'.T'$, i.e., $c' \vdash c$ and $\vdash T'' \leq T'$ (rule (Sub-name-arr) of Figure 5),
  5. $\{\alpha\}; c; \Gamma \vdash t'' : T''$, and
  6. $\alpha \Vdash c$.
  From 3., and $c' \vdash c$ (in 4.) we have that $\emptyset \vdash c\{\alpha \mapsto N\}$. Therefore $\alpha \mapsto N$ is consistent with $\{\alpha\}$ and c. Applying Lemma 8 to 5. we get that $\emptyset; c\{\alpha \mapsto N\}; \Gamma\{\alpha \mapsto N\} \vdash t''\{\alpha \mapsto N\} : T''\{\alpha \mapsto N\}$. From 6. ($c$ refers to α) we have that $c\{\alpha \mapsto N\}$ is an empty set of constraints. Therefore, since Γ does not contain α, $\Gamma = \Gamma\{\alpha \mapsto N\}$ and

$$\emptyset;\emptyset;\Gamma \vdash t''\{\alpha \mapsto N\} : T''\{\alpha \mapsto N\}$$

  From $\vdash T'' \leq T'$ (in 4.) we have that $\vdash T''\{\alpha \mapsto N\} \leq T'\{\alpha \mapsto N\}$. Therefore, from 1. and Lemma 4, we get $\vdash T''\{\alpha \mapsto N\} \leq T$. From Lemma 5, and $A; c; \Gamma \vdash t : T$ we derive $\emptyset;\emptyset \vdash T$ OK. Applying typing rule (T-Sub), we derive

$$\emptyset;\emptyset;\Gamma \vdash t''\{\alpha \mapsto N\} : T$$

  which concludes the proof of this clause.
- If the applied rule is (Reb-App), then $t' = \langle u, u_2 \mid t''\{x \mapsto r(u_1(x)) \mid x \in dom(u_1)\}\rangle$, $t = \langle u \mid r\rangle \succ \langle u_1, u_2 \mid t''\rangle$, $rng(u_1) \subseteq dom(r)$, and $rng(u_2) \cap dom(r) = \emptyset$. Moreover, by definition of "u, u_2", $dom(u) \cap dom(u_2) = \emptyset$. From Lemma 6.10, for some Δ', Δ_1, Δ_2, and T' we have that:
  α. $\vdash \langle \Delta', \Delta_1 \mid T'\rangle \leq T$, and
  β. $A; c \vdash \Delta_1, \Delta_2$ OK.

  - Assume we are in the first of the two alternatives of Lemma 6.10, then $\Delta_1 = \emptyset$, therefore u_2 is empty, and for some ν we have that:
    1. $A; c; \Gamma \vdash \langle u_1 \mid t''\rangle : \langle \Delta \mid T'\rangle$,
    2. $A; c; \Gamma \vdash \langle u \mid r\rangle : \langle \Delta' \mid \Delta, \Delta_2\rangle^\nu$ $(dom(\Delta) \cap dom(\Delta_2) = \emptyset)$.
    From Lemma 6.5 and 1., $u_1 \in unb(\Delta, A, c)$, and
    3. $A; c; \Gamma[ctx(u_1)] \vdash t'' : T''$ where $\vdash T'' \leq T'$.
    From 3. and Lemma 6.5 we have that $u \in unb(\Delta', A, c)$ Moreover, from Lemma 6.6, we get $r \in reb((\Delta, \Delta_2), A, c, \Gamma[ctx(u)])^\nu$. From Definition 10.2, we can assume that $r = X_1{:}T_1 \mapsto t_1, \ldots, X_{m+n+k}{:}T_{m+n+k} \mapsto t_{m+n+k}$ where $\Delta = X_1{:}T'_1 \ldots, X_m{:}T'_m$, and $\Delta_2 = X_{m+1}{:}T'_{m+1} \ldots, X_{m+n}{:}T'_{m+n}$,
    4. $\vdash T_i \leq T'_i$ $(1 \leq i \leq m+n)$, and
    5. $A; c; \Gamma[ctx(u)] \vdash t_j : T_j$ $(1 \leq j \leq m+n+k)$.
    From $u_1 \in unb(\Delta)$, $u_1 = x_1{:}T''_{n_1} \mapsto X_{n_1}, \ldots, x_p{:}T''_{n_p} \mapsto X_{n_p}$, where $\{n_1, \ldots, n_p\} \subseteq \{1, \ldots, m\}$, and
    6. $T'_{n_i} \leq T''_{n_i}$ $(1 \leq i \leq p)$.
    From 5., and $\{n_1, \ldots, n_p\} \subseteq \{1, \ldots, m\}$, we derive that:
    7. $A; c; \Gamma[ctx(u)] \vdash t_{n_j} : T_{n_j}$ $(1 \leq j \leq p)$.
    Without loss of generality we can assume that $dom(u)$, and $dom(u_1)$ are disjoint. So from 3. and 7 we derive:

8. $A; c; \Gamma[ctx(u, u_1)] \vdash t'' : T''$.
9. $A; c; \Gamma[ctx(u, u_1)] \vdash t_{n_j} : T_{n_j}$ $(1 \le j \le p)$.
From 4., 6., and Lemma 4, $\vdash T_{n_i} \le T''_{n_i}$ $(1 \le i \le p)$. From 8., 9., and Lemma 7 we derive that:

$$A; c; \Gamma[ctx(u)] \vdash t''\{x_1 \mapsto t_{n_1}, \dots, x_p \mapsto t_{n_p}\} : T''.$$

(Note that, $t''\{x_1 \mapsto t_{n_1}, \dots, x_p \mapsto t_{n_p}\} = t''\{x \mapsto r(u_1(x)) \mid x \in dom(u_1)\} = t'$.) From 2. we have that $A; c \vdash \Delta$ OK, so applying rule (T-Unb), $\Gamma \vdash t' : \langle name_ctx(u) \mid T'' \rangle$. From the fact that $u \in unb(\Delta', A, c)$, $\vdash name_ctx(u) \le \Delta'$, and, from 3., we have $\vdash T'' \le T'$. Therefore $\vdash \langle name_ctx(u) \mid T'' \rangle \le \langle \Delta' \mid T' \rangle \le T$ and , so applying (T-Sub) we get $A; c; \Gamma \vdash t' : T$.

- Assume we are in the second of the two alternatives of Lemma 6.10, then for some Δ'_2 we have that:
1. $A; c; \Gamma \vdash \langle u_1, u_2 \mid t'' \rangle : \langle \Delta, \Delta_1 \mid T' \rangle$,
2. $A; c; \Gamma \vdash \langle u \mid r \rangle : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2 \rangle^\circ$, and
3. $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$.
From Lemma 6.5 and 1., $u_1, u_2 \in unb((\Delta, \Delta_1), A, c)$, (both $u_1 \in unb((\Delta, \Delta_1), A, c)$, and $u_2 \in unb((\Delta, \Delta_1, A), c)$), and
4. $A; c; \Gamma[ctx(u_1, u_2)] \vdash t'' : T''$ where $\vdash T'' \le T'$.
From 2. and Lemma 6.5, $u \in unb((\Delta', \Delta_1), A, c)$, and from Lemma 6.6, $r \in reb((\Delta, \Delta_2), A, c, \Gamma[ctx(u)])^\circ$. From Def. 10.2, $r = X_1{:}T_1 \mapsto t_1, \dots, X_{m+n}{:}T_{m+n+k} \mapsto t_{m+n}$.
Let $\Delta = X_1{:}T'_1 \dots, X_m{:}T'_m$, and $\Delta_2 = X_{m+1}{:}T'_{m+1} \dots, X_{m+n}{:}T'_{m+n}$,
5. $\vdash T_i \le T'_i$ $(1 \le i \le m+n)$, and
6. $A; c; \Gamma[ctx(u)] \vdash t_j : T_j$ $(1 \le j \le m+n)$.
Since $rng(u_1) \subseteq dom(r)$, and 3. we get $u_1 \in unb(\Delta, A, c)$. So $u_1 = x_1{:}T''_{n_1} \mapsto X_{n_1}, \dots, x_p{:}T''_{n_p} \mapsto X_{n_p}$, where $\{n_1, \dots, n_p\} \subseteq \{1, \dots, m\}$,
7. $\vdash T'_{n_i} \le T''_{n_i}$ $(1 \le i \le p)$, and
8. $A; c; \Gamma[ctx(u)] \vdash t_{n_j} : T_{n_j}$ $(1 \le j \le p)$.
Without loss of generality we can assume that $dom(u)$, $dom(u_1)$, and $dom(u_2)$, are pairwise disjoint. So from 4. and 8. we derive:
9. $A; c; \Gamma[ctx(u, u_1, u_2)] \vdash t'' : T''$.
10. $A; c; \Gamma[ctx(u, u_1, u_2)] \vdash t_{n_j} : T_{n_j}$ $(1 \le j \le p)$.
From 5., 7., and and Lemma 4, $\vdash T_{n_i} \le T''_{n_i}$ $(1 \le i \le p)$. From 9., 10., and Lemma 7 we derive that:

$$A; c; \Gamma[ctx(u, u_2)] \vdash t''\{x_1 \mapsto t_{n_1}, \dots, x_p \mapsto t_{n_p}\} : T''.$$

From 2. we have that $A; c \vdash \Delta', \Delta_1$ OK, so applying rule (T-Unb), $A; c; \Gamma \vdash t' : \langle name_ctx(u, u_2) \mid T'' \rangle$. From $u \in unb((\Delta', \Delta_1), A, c)$, $\vdash name_ctx(u) \le \Delta', \Delta_1$, and from 4. we have $\vdash T'' \le T'$, therefore $\vdash \langle name_ctx(u) \mid T'' \rangle \le \langle \Delta', \Delta_1 \mid T' \rangle \le T$, so applying (T-Sub) we get $\Gamma \vdash t' : T$. ◀

In the following we write $\vdash t : T$ for $\emptyset; \emptyset; \emptyset \vdash t : T$.

▶ **Lemma 12** (Canonical forms).
1. *If* $\vdash v : T_1 \to T_2$, *then* $v = \lambda x{:}T'_1.t$ *where* $\vdash T_1 \le T'_1$.
2. *If* $\vdash v : \langle \Delta \mid T \rangle$, *then* $v = \langle u \mid t \rangle$, *and* $rng(u) \subseteq dom(\Delta)$.
3. *If* $\vdash v : \langle \Delta_1 \mid \Delta_2 \rangle^\circ$, *then* $v = \langle u \mid r \rangle$, $rng(u) \subseteq dom(\Delta)$, *and* $dom(\Delta_2) = dom(r)$.

4. *If* $\vdash v : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}$, *then* $v = \langle u \mid r \rangle$, $rng(u) \subseteq dom(\Delta)$, *and* $dom(\Delta_2) \subseteq dom(r)$.
5. *If* $\vdash v : \forall \alpha{:}c.T$, *then* $v = \Lambda \alpha{:}c.t$.

Proof. By case analysis on the shape of values. ◀

▶ **Theorem 13** (Progress). *Let t be such that, for some T we have $\vdash t : T$. Then either t is a value or for some t', we have that $t \longrightarrow t'$.*

Proof. By induction on the derivation of $\vdash t : T$ with case analysis on the last typing rule used. Notice that since $\vdash t : T$, t cannot a variable.

- If the last applied rule is (T-App), then

$$\frac{\vdash t_1 : T_1 \to T_2 \quad \vdash t_2 : T_1}{\vdash t_1\ t_2 : T_2}$$

  If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1\ t_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,]\ t_2$, and by rule (Cont), $t_1\ t_2 \longrightarrow t_1'\ t_2$. If t_1 is a value v, and t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So $t_1\ t_2 = \mathcal{E}[t_2]$ with $\mathcal{E} = v\ [\,]$, and by rule (Cont), $v\ t_2 \longrightarrow v\ t_2'$.
  If both t_1 and t_2 are values, then by Lemma 12.1, $t_1 = \lambda x{:}T_1.t''$. Therefore, $t \longrightarrow t'$ with rule (App).
- If the last applied rule is (T-Name-App), then

$$\frac{\emptyset \vdash c\{\alpha \mapsto X\} \quad \vdash t_1 : \forall \alpha{:}c.T \quad \emptyset \vdash X}{\vdash t_1\ X : T\{\alpha \mapsto X\}}$$

  Therefore $X = N$ for some N. If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1\ X = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,]\ X$, and by rule (Cont), $t_1\ X \longrightarrow t_1'\ X$. If t_1 is a value v, then by Lemma 12.5, $t_1 = \Lambda \alpha{:}c.t''$.
  From $\vdash t_1 : \forall \alpha{:}c.T$ and Lemma 5, we have $\emptyset; \emptyset \vdash \Lambda \alpha{:}c.t''\ \texttt{OK}$. From the definition of Figure 4, $\{\alpha\}; c \vdash t''\ \texttt{OK}$. Since $\emptyset \vdash c\{\alpha \mapsto X\}$ we derive that $\alpha \mapsto N$ is consistent with $\{\alpha\}$ and c. Therefore, from Lemma 3, we get that $\emptyset; c\{\alpha \mapsto N\} \vdash t''\{\alpha \mapsto N\}\ \texttt{OK}$. Therefore, rule (Name-App) is applicable and $t \longrightarrow t''\{\alpha \mapsto N\}$.
- If the last applied rule is (T-Over), then

$$\frac{\vdash t_1 : \langle \Delta \mid \Delta_1, \Delta_1' \rangle^{\nu_1} \quad \vdash t_2 : \langle \Delta \mid \Delta_2 \rangle^{\nu_2} \quad \emptyset; \emptyset \vdash \Delta_1, \Delta_2\ \texttt{OK}}{\vdash t_1 \lhd t_2 : \langle \Delta \mid \Delta_1, \Delta_2 \rangle^{\nu_1 \sqcup \nu_2}}$$

  If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1 \lhd t_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,] \lhd t_2$, and by rule (Cont), $t_1 \lhd t_2 \longrightarrow t_1' \lhd t_2$. It t_1 is a value v, and t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So $t_1 \lhd t_2 = \mathcal{E}[t_2]$ with $\mathcal{E} = v \lhd [\,]$, and by rule (Cont), $v \lhd t_2 \longrightarrow v \lhd t_2'$.
  If both t_1 and t_2 are values, then from Lemma 12.5, $t_1 = \langle u_1 \mid r_1 \rangle$, and $t_2 = \langle u_2 \mid r_2 \rangle$. We can assume (renaming bound variables) that $dom(u_1) \cap dom(u_2) = \emptyset$. Therefore, $t \longrightarrow t'$ with rule (Over).
- If the last applied rule is (T-Reb-App), then

$$\frac{\vdash t_1 : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2 \rangle^{\nu} \quad \vdash t_2 : \langle \Delta, \Delta_1 \mid T \rangle \quad \emptyset; \emptyset \vdash \Delta_1, \Delta_2\ \texttt{OK}}{\vdash t_1 \succ t_2 : \langle \Delta', \Delta_1 \mid T \rangle}$$

  If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1 \succ t_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,] \succ t_2$, and by rule (Cont), $t_1 \succ t_2 \longrightarrow t_1' \succ t_2$. If t_1 is a value v, and t_2 is not a value,

then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So $t_1 \succ t_2 = \mathcal{E}[t_2]$ with $\mathcal{E} = v \succ [\,]$, and by rule (Cont), $v \succ t_2 \longrightarrow v \succ t_2'$.
If t_1 is a value, then from Lemma 12.5, $t_1 = \langle u \mid r \rangle$. Since t_2 is a value, from Lemma 12.3, $t_2 = \langle u' \mid t'' \rangle$.
Let $u' = u_1, u_2$ be such that $rng(u_1) \subseteq dom(r)$, and $rng(u_2) \cap dom(r) = \emptyset$, $t \longrightarrow t'$ with rule (RebApp).

- If the last applied rule is (T-Run), then

$$\frac{\vdash t_1 : \langle\ \mid T \rangle}{\vdash !t_1 : T}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $!t_1 = \mathcal{E}[t_1]$ with $\mathcal{E} = ![\,]$, and by rule (Cont), $!t_1 \longrightarrow !t_1'$. If t_1 is a value, from Lemma 12.3, $t_1 = \langle\ \mid t' \rangle$, so $t_1 \longrightarrow t'$ with rule (Run).

- If the last applied rule is (T-Rename), then

$$\frac{\vdash t_1 : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu} \quad \emptyset;\emptyset \vdash \sigma_1\ \texttt{OK} \quad \emptyset;\emptyset \vdash \sigma_2\ \texttt{OK} \quad \emptyset;\emptyset \vdash \sigma_1 \circ \Delta_1\ \texttt{OK}}{\vdash \sigma_1 \ltimes t_1 \rtimes \sigma_2 : \langle \sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2 \rangle^{\circ}}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $\sigma_1 \ltimes t_1 \rtimes \sigma_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = \sigma_1 \ltimes [\,] \rtimes \sigma_2$, and by rule (Cont), $\sigma_1 \ltimes t_1 \rtimes \sigma_2 \longrightarrow \sigma_1 \ltimes t_1' \rtimes \sigma_2$. If t_1 is a value, from Lemma 12.5, $t_1 = \langle u \mid r \rangle$, and $rng(u) \subseteq dom(\Delta)$, and $dom(\Delta_2) \subseteq dom(r)$. Therefore, both $\sigma_1 \circ u$ and $r \circ \sigma_2$ are defined, and $t \longrightarrow \langle \sigma_1 \circ u \mid r \circ \sigma_2 \rangle$ with rule (Rename). ◀

6 Towards a Typing Algorithm

Because of rule (T-Sub), the type system defined in Section 3 is not deterministic, and, hence, no typechecking algorithm can be directly derived from it.

In this section we show how the type system of Section 3 can be turned into a deterministic one from which a typechecking algorithm can be directly derived; more in details, if a term t can be typed in the non deterministic type system, then it can be typed in the new type system with a type which is the most specific (that is, the principal one) among all types that can be assigned to t by the non deterministic type system. Furthermore, thanks to the introduction of judgments to compute the greatest lower and least upper bound of two types, the deterministic type system is able to type more terms.

For space limitation, we only sketch the main typing rules and provide the most important definitions, and omit formal proofs.

To get a deterministic type system rule (T-Sub) has to be removed, and typing rules (T-App), (T-Over), (T-Reb-App), and (T-Rename) need to be modified.

Rule (T-App) is adapted in the standard way:

$$\text{(NT-App)}\ \frac{A; c; \Gamma \vdash t_1 : T_1 \to T_2 \quad A; c; \Gamma \vdash t_2 : T_1' \quad \vdash T_1' \leq T_1}{A; c; \Gamma \vdash t_1\ t_2 : T_2}$$

The remaining rules rely on two new judgments for computing the greatest lower and the least upper bound of two types, respectively.

The judgment $c \models glb(T_1; T_2) = T$ is derivable if types T_1 and T_2 admit the greatest lower bound T under c; by duality, the judgment $c \models lub(T_1; T_2) = T$ for least upper bounds is defined, as well. Both judgments are defined in Figure 12; we also use the operator $\sqcap$ which is the dual of $\sqcup$: $+ \sqcap \nu = \nu \sqcap + = \nu$, and $\circ \sqcap \circ = \circ$.

$$\text{(GWF-BASE)}\ \frac{c \models X_i \stackrel{?}{=} X_j \Rightarrow T_i = T_j\ (1 \leq i, j \leq m)}{c \models \mathit{gwf}(X_1{:}T_1, \ldots, X_n{:}T_m) = X_1{:}T_1, \ldots, X_n{:}T_m}$$

$$\text{(GWF-STEP)}\ \frac{c \models X_1 \stackrel{?}{=} X_2 \quad c \models \mathit{glb}(T_1; T_2) = T \quad c \models \mathit{gwf}(X_1{:}T, X_2{:}T, \Delta) = \Delta'}{c \models \mathit{gwf}(X_1{:}T_1, X_2{:}T_2, \Delta) = \Delta'}\ T_1 \neq T_2$$

$$\text{(LWF-BASE)}\ \frac{c \models X_i \stackrel{?}{=} X_j \Rightarrow T_i = T_j\ (1 \leq i, j \leq m)}{c \models \mathit{lwf}(X_1{:}T_1, \ldots, X_n{:}T_m) = X_1{:}T_1, \ldots, X_n{:}T_m}$$

$$\text{(LWF-STEP)}\ \frac{c \models X_1 \stackrel{?}{=} X_2 \quad c \models \mathit{lub}(T_1; T_2) = T \quad c \models \mathit{lwf}(X_1{:}T, X_2{:}T, \Delta) = \Delta'}{c \models \mathit{lwf}(X_1{:}T_1, X_2{:}T_2, \Delta) = \Delta'}\ T_1 \neq T_2$$

$$\text{(GLB-CONTEXT)}\ \frac{c \models \mathit{gwf}(\Delta_1, \Delta_2) = \Delta}{c \models \mathit{glb}(\Delta_1; \Delta_2) = \Delta}$$

$$\text{(LUB-CONTEXT)}\ \frac{c \models \mathit{lub}(\Delta_1(X); \Delta_2(X)) = T_X\ \forall X \in \mathit{dom}(\Delta)}{c \models \mathit{lub}(\Delta_1; \Delta_2) = \Delta}\ \begin{array}{l} \mathit{dom}(\Delta) = \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) \\ \forall X \in \mathit{dom}(\Delta)\ \Delta(X) = T_X \end{array}$$

$$\text{(GLB-ARR)}\ \frac{c \models \mathit{lub}(T_1; T_1') = T \quad c \models \mathit{glb}(T_2; T_2') = T'}{c \models \mathit{glb}(T_1 \to T_2; T_1' \to T_2') = T \to T'}$$

$$\text{(LUB-ARR)}\ \frac{c \models \mathit{glb}(T_1; T_1') = T \quad c \models \mathit{lub}(T_2; T_2') = T'}{c \models \mathit{lub}(T_1 \to T_2; T_1' \to T_2') = T \to T'}$$

$$\text{(GLB-UNB)}\ \frac{c \models \mathit{lub}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{glb}(T_1; T_2) = T}{c \models \mathit{glb}(\langle \Delta_1 \mid T_1 \rangle; \langle \Delta_2 \mid T_2 \rangle) = \langle \Delta \mid T \rangle}$$

$$\text{(LUB-UNB)}\ \frac{c \models \mathit{glb}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{lub}(T_1; T_2) = T}{c \models \mathit{lub}(\langle \Delta_1 \mid T_1 \rangle; \langle \Delta_2 \mid T_2 \rangle) = \langle \Delta \mid T \rangle}$$

$$\text{(GLB-REB)}\ \frac{c \models \mathit{lub}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{glb}(\Delta_1'; \Delta_2') = \Delta'}{c \models \mathit{glb}(\langle \Delta_1 \mid \Delta_1' \rangle^{\nu_1}; \langle \Delta_2 \mid \Delta_2' \rangle^{\nu_2}) = \langle \Delta \mid \Delta' \rangle^{\nu_1 \sqcap \nu_2}}\ \begin{array}{l} \nu_1 = \circ \Rightarrow \mathit{dom}(\Delta_2') \subseteq \mathit{dom}(\Delta_1') \\ \nu_2 = \circ \Rightarrow \mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2') \end{array}$$

$$\text{(LUB-REB)}\ \frac{c \models \mathit{glb}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{lub}(\Delta_1'; \Delta_2') = \Delta'}{c \models \mathit{lub}(\langle \Delta_1 \mid \Delta_1' \rangle^{\nu_1}; \langle \Delta_2 \mid \Delta_2' \rangle^{\nu_2}) = \langle \Delta \mid \Delta' \rangle^{\nu}}\ \nu = \begin{cases} \nu_1 \sqcup \nu_2 & \text{if } \mathit{dom}(\Delta_1') = \mathit{dom}(\Delta_2') \\ + & \text{otherwise} \end{cases}$$

Figure 12 Rules for *glb* and *lub*

Rule (GLB-CONTEXT) defines the greatest lower bound of two name contexts Δ_1 and Δ_2; it considers the union Δ_1, Δ_2 and then derives from it a well-formed name context with the auxiliary judgment $c \models \mathit{gwf}(\Delta_1, \Delta_2) = \Delta$; indeed, Δ_1, Δ_2 may not be well-formed. For instance, if $\Delta_1 = N{:}T_1$ and $\Delta_2 = N{:}T_2$, with $T_1 \neq T_2$, then Δ_1, Δ_2 is not well-formed; however, if $c \models \mathit{glb}(T_1; T_2) = T$, then $N{:}T, N{:}T$ is well-formed, and is the greatest lower bound of Δ_1 and Δ_2 under c.

The judgment $c \models \mathit{gwf}(\Delta) = \Delta'$ is defined by the two rules (GWF-BASE) and (GWF-STEP). The former rule is applied when the name context is well-formed; for instance, if N_1 and N_2 are distinct names, then $c \models \mathit{gwf}(N_1{:}T_1, N_2{:}T_2) = N_1{:}T_1, N_2{:}T_2$ is derivable for all c, even when $T_1 \neq T_2$; other examples of application of the rule are given by the derivation of judgments $X_1 \neq X_2 \models \mathit{gwf}(X_1{:}T_1, X_2{:}T_2) = X_1{:}T_1, X_2{:}T_2$, and $c \models \mathit{gwf}(N{:}T, N{:}T) = N{:}T, N{:}T$. Rule (GWF-STEP) is applied when there exist two names that might be equal, but are associated with different types T_1 and T_2; in such a case, the greatest lower bound of T_1 and T_2 has to be computed. For instance, if $T_1 \neq T_2$, and $c \models \mathit{glb}(T_1; T_2) = T$, then the judgment

$c \models \mathit{gwf}(N{:}T_1, N{:}T_2) = N{:}T, N{:}T$ can be derived.

Rules (lwf-base) and (lwf-step) defines the judgment $c \models \mathit{lwf}(\Delta) = \Delta'$, which is the dual of $c \models \mathit{gwf}(\Delta) = \Delta'$, and is directly used in the typing rules.

Rule (lub-context) defines the least upper bound Δ of two name contexts Δ_1 and Δ_2; Δ defines all names that are defined in both Δ_1 and Δ_2, and each of these names is associated in Δ with the least upper bound of the two corresponding types associated in Δ_1 and Δ_2; for instance, if $c \models \mathit{lub}(T_1; T_1') = T$ and X_1, X_2, and X_3 are distinct, then $c \models \mathit{lub}(X_1{:}T_1, X_2{:}T_2; X_1{:}T_1', X_3{:}T_3) = X_1{:}T$.

Since subtyping between arrow types is contravariant in the argument types and covariant in the return types, the definition of *glb* and *lub* for arrow types is straightforward. An analogous consideration applies also for unbound types, and rebinding types; however, for this latter kind of types, annotations $+/\circ$ make the definition a bit more involved.

In rule (glb-reb), the resulting type must be closed if at least one of the types is closed (as specified by the $\sqcap$ operator); for this reason, if one type T is closed, then the other type cannot specify a rebinding map whose domain contains names that are not defined in T. This means that the greatest lower bound of two rebinding types may be undefined; for instance, if N_1 and N_2 are distinct, then for all c, there is no type T s.t. $c \models \mathit{glb}(\langle \mid N_1{:}T_1\rangle^{\circ}; \langle \mid N_2{:}T_2\rangle^{+}) = T$ (actually, the two rebinding types do not even admit any lower bound).

In rule (lub-reb) there is no side condition that prevents the existence of the least upper bound of two rebinding types; however, the least upper bound can be closed only if both types are closed, and specify rebinding maps having the same domain.

Rules (T-Over), (T-Reb-App), and (T-Rename), can be modified as follows to get a deterministic type system:

$$\text{(NT-Over)}\ \frac{A; c; \Gamma \vdash t_1 : \langle \Delta' \mid \Delta_1, \Delta_1' \rangle^{\nu_1} \quad A; c; \Gamma \vdash t_2 : \langle \Delta'' \mid \Delta_2 \rangle^{\nu_2} \quad c \models \mathit{glb}(\Delta'; \Delta'') = \Delta \quad c \models \mathit{lwf}(\Delta_1, \Delta_2) = \Delta'''}{A; c; \Gamma \vdash t_1 \lhd t_2 : \langle \Delta \mid \Delta''' \rangle^{\nu_1 \sqcup \nu_2}} \quad \begin{array}{l}\Delta_1 = \emptyset \text{ or } \nu_2 = \circ\\ \mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2)\\ \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset\end{array}$$

If t_1 and t_2 have type $\langle \Delta' \mid \Delta_1'' \rangle^{\nu_1}$, and $\langle \Delta'' \mid \Delta_2 \rangle^{\nu_2}$, respectively, then Δ_1'' can be always uniquely split in Δ_1 and Δ_1', s.t. $\mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2)$, and $\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$. The result of the overriding has type $\langle \Delta \mid \Delta''' \rangle^{\nu_1 \sqcup \nu_2}$, where Δ (which is in contravariant position) must be subtype of both Δ', and Δ'', hence $c \models \mathit{glb}(\Delta'; \Delta'') = \Delta$, and Δ''' is the most specific well-formed name context compatible with Δ_1, Δ_2, that is, $c \models \mathit{lwf}(\Delta_1, \Delta_2) = \Delta'''$; indeed, Δ_1, Δ_2 might not be well-formed. This implies that rule (NT-Over) is more liberal than (T-Over).

$$\text{(NT-Reb-App)}\ \frac{A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta', \Delta_2 \rangle^{\nu} \quad A; c; \Gamma \vdash t_2 : \langle \Delta'', \Delta_1 \mid T \rangle \quad c \models \mathit{glb}(\Delta; \Delta_1) = \Delta''' \quad \vdash \Delta' \leq \Delta'' \quad c \models \Delta_2 \prec^{?} \Delta_1}{A; c; \Gamma \vdash t_1 \succ t_2 : \langle \Delta''' \mid T \rangle} \quad \begin{array}{l}\Delta_1 = \emptyset \text{ or } \nu = \circ\\ \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset\\ \mathit{dom}(\Delta') = \mathit{dom}(\Delta'')\end{array}$$

If t_1 and t_2 have type $\langle \Delta \mid \Delta_3 \rangle^{\nu}$, and $\langle \Delta'', \Delta_1 \mid T \rangle$, respectively, then Δ_3 can be always uniquely split in Δ' and Δ_2, s.t. $\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$, and $\mathit{dom}(\Delta') = \mathit{dom}(\Delta'')$.

The rule is applicable only if $\vdash \Delta' \leq \Delta''$ holds, to ensure that the names in t_2 are bound to type compatible values; furthermore, the judgment $c \models \Delta_2 \prec^{?} \Delta_1$ ensures compatibility in case some name X in Δ_1 is bound to a type with a name Y in Δ_2 after name application. The judgment $c \models \Delta_2 \prec^{?} \Delta_1$ holds if and only if for all $X \in \mathit{dom}(\Delta_2)$ and $Y \in \mathit{dom}(\Delta_1)$, $c \models X \stackrel{?}{=} Y$ implies $\vdash \Delta_2(X) \leq \Delta_1(Y)$.

The final type of the rebinding is $\langle \Delta''' \mid T \rangle$, where Δ''' has to be a subtype of both Δ, and Δ_1, hence $c \models \mathit{glb}(\Delta; \Delta_1) = \Delta'''$.

$$\text{(NT-Rename)}\ \frac{A; c \vdash \sigma_1\ \texttt{OK} \quad A; c \vdash \sigma_2\ \texttt{OK} \quad c \models \mathit{gwf}(\sigma_1 \circ \Delta_1) = \Delta_1' \quad A; c; \Gamma \vdash t : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}}{A; c; \Gamma \vdash \sigma_1 \ltimes t \rtimes \sigma_2 : \langle \Delta_1' \mid \Delta_2 \circ \sigma_2 \rangle^{\circ}}$$

The typing rule is almost the same as rule (T-Rename) of Figure 7; however, in this case the typing succeeds even when $\sigma_1 \circ \Delta_1$ is not well-formed, if there exists Δ'_1 s.t. $c \models gwf(\sigma_1 \circ \Delta_1) = \Delta'_1$ is derivable (that is, Δ'_1 is the greatest well-formed subtype of $\sigma_1 \circ \Delta_1$).

7 Conclusion

We have proposed a calculus which integrates standard static binding with incremental rebinding of code based on a *parametric* nominal interface. That is, names, which can be either constants or variables, are used as interface of fragments of code with free variables, which can be passed around and rebound. The type system is based on *constrained name-polymorphic types*, where simple inequality constraints prevent conflicts when instantiating name variables. The calculus can express type-safe dynamic adaptation of code, as illustrated by the example of mixins. Similar results can be achieved in dynamically typed languages, such as JavaScript or through the use of reflection. However, in these settings we loose the possibility of expressing type constraints that can be statically checked. In C++ with multiple inheritance and templates we can define mixins, but we have to know the names of the methods that will be mixed in.

This work continues a stream of research on foundations of binding mechanisms, started with [9, 8]. The goal was to provide a unifying foundation for dynamic scoping, rebinding of marshalled computations, meta-programming features, and operators present in calculi for modules. Classical (ad-hoc) models for dynamic scoping are [11] and [7], whereas the λ_{marsh} calculus of [6] supports rebinding w.r.t. named contexts (not individual variables). The meta-programming features of our calculus are orthogonal to the one of MetaML [16], since, on one side, we do not have an analog of the *escape* annotation of MetaML forcing evaluation inside boxed code, but on the other, our rebinding construct avoids the problem of unwanted variable capturing. Module calculi are described, e.g., in [3].

In future work we will formalize and prove the relation between the non deterministic and algorithmic variants of the type system. Exploring the possibility of inferring constraints on name variables, rather than explicitly annotating name abstractions, is another possible direction of work. However, as the example at the end of Section 4 shows, this is difficult due to the lack of a suitable notion of "principal typing" with respect to name constraints.

Another possible direction is to add polymorphic types, so that name polymorphism can be more effectively used. Finally, we plan to study the relations between our name abstraction and the one provided by languages of the family of FreshML [14, 13], where it is possible to compute with syntactical data structures involving names and name binding in a statically typed setting.

Acknowledgements We thank the referees for their helpful comments, the paper improved due to their suggestions.

References

1 D. Ancona, P. Giannini, and E. Zucca. Reconciling positional and nominal binding. In S. Graham-Lengrand and L. Paolini, editors, *Proc. of 6th Wksh. on Intersection Types and Related Systems, ITRS 2012*, volume 121 of *Electron. Proc. in Theor. Comput. Sci.*, pages 81–93. Open Publishing Assoc., 2013. doi:10.4204/eptcs.121.6.

2 D. Ancona, P. Giannini, and E. Zucca. Type safe incremental rebinding. *Math. Struct. Comput. Sci.*, 27(2):94–122, 2017. doi:10.1017/s0960129515000109.

3 D. Ancona and E. Zucca. A calculus of module systems. *J. Funct. Program.*, 12(2):91–132, 2002. doi:10.1017/s0956796801004257.

4 Davide Ancona, Paola Giannini, and Elena Zucca. Incremental rebinding with name polymorphism. *Electr. Notes Theor. Comput. Sci.*, 322:19–34, 2016. doi:10.1016/j.entcs.2016.03.003.

5 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - designing a java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003. doi:10.1145/937563.937567.

6 Gavin M. Bierman, Michael W. Hicks, Peter Sewell, Gareth Paul Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time? In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 99–110. ACM, 2003. doi:10.1145/944705.944715.

7 L. Dami. A lambda-calculus for dynamic binding. *Theor. Comput. Sci.*, 192(2):201–231, 1997. doi:10.1016/s0304-3975(97)00150-3.

8 M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. Intersection types for unbind and rebind. In E. Pimentel, B. Venneri, and J. Wells, editors, *Proc. of 5th Wksh. on Intersection Types and Related Systems, ITRS 2010*, volume 45 of *Electron. Proc. in Theor. Comput. Sci.*, pages 45–58. Open Publishing Assoc., 2011. doi:10.4204/eptcs.45.4.

9 Mariangiola Dezani-Ciancaglini, Paola Giannini, and Elena Zucca. Extending the lambda-calculus with unbind and rebind. *RAIRO - Theor. Inf. and Applic.*, 45(1):143–162, 2011. doi:10.1051/ita/2011008.

10 D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 4th edition, 2011.

11 L. Moreau. A syntactic theory of dynamic binding. *High. Order Symb. Comput.*, 11(3):233–279, 1998. doi:10.1023/a:1010087314987.

12 Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 207–218. ACM, 2003. doi:10.1145/888251.888271.

13 A. M. Pitts and M. R. Shinwell. Generative unbinding of names. *Log. Methods Comput. Sci.*, 4(1):article 4, 2008. doi:10.2168/lmcs-4(1:4)2008.

14 Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. Freshml: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003. doi:10.1145/944705.944729.

15 B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 4th edition, 2013.

16 W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1–2):211–242, 2000. doi:10.1016/s0304-3975(00)00053-0.

Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*

Cyril Cohen[1], Thierry Coquand[2], Simon Huber[3], and Anders Mörtberg[4]

1 **Inria Sophia Antipolis – Méditerranée, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex, France**
`cyril.cohen@inria.fr`
2 **Department of Computer Science and Engineering, University of Gothenburg, 412 96 Gothenburg, Sweden**
`thierry.coquand@cse.gu.se`
3 **Department of Computer Science and Engineering, University of Gothenburg, 412 96 Gothenburg, Sweden**
`simon.huber@cse.gu.se`
4 **School of Mathematics, Institute for Advanced Study, 1 Einstein Drive, Princeton, NJ 08540, USA**
`amortberg@math.ias.edu`

Abstract

This paper presents a type theory in which it is possible to directly manipulate n-dimensional cubes (points, lines, squares, cubes, etc.) based on an interpretation of dependent type theory in a cubical set model. This enables new ways to reason about identity types, for instance, function extensionality is directly provable in the system. Further, Voevodsky's univalence axiom is provable in this system. We also explain an extension with some higher inductive types like the circle and propositional truncation. Finally we provide semantics for this cubical type theory in a constructive meta-theory.

1998 ACM Subject Classification F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages, F.4.1 Mathematical Logic and Formal Languages: Mathematical Logic

Keywords and phrases univalence axiom, dependent type theory, cubical sets

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.5

1 Introduction

This work is a continuation of the program started in [6, 13] to provide a constructive justification of Voevodsky's univalence axiom [27]. This axiom allows many improvements for the formalization of mathematics in type theory: function extensionality, identification of isomorphic structures, etc. In order to preserve the good computational properties of type theory it is crucial that postulated constants have a computational interpretation. Like in [6, 13, 22] our work is based on a nominal extension of λ-calculus, using *names* to represent formally elements of the unit interval $[0, 1]$. This paper presents two main contributions.

* This material is based upon work supported by the National Science Foundation under agreement No. DMS-1128155. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

© Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg;
licensed under Creative Commons License CC-BY

The first one is a refinement of the semantics presented in [6, 13]. We add new operations on names corresponding to the fact that the interval $[0, 1]$ is canonically a de Morgan algebra [3]. This allows us to significantly simplify our semantical justifications. In the previous work, we noticed that it is crucial for the semantics of higher inductive types [26] to have a "diagonal" operation. By adding this operation we can provide a semantical justification of some higher inductive types and we give two examples (the spheres and propositional truncation). Another shortcoming of the previous work was that using path types as equality types did not provide a justification of the computation rule of the Martin-Löf identity type [19] as a judgmental equality. This problem has been solved by Andrew Swan [25], in the framework of [6, 13, 22], who showed that we can define a new type, *equivalent to*, but not judgmentally equal to the path type. This has a simple definition in the present framework.

The second contribution is the design of a type system[1] inspired by this semantics which extends Martin-Löf type theory [20, 19]. We add two new operations on contexts: addition of new names representing dimensions and a restriction operation. Using these we can define a notion of extensibility which generalizes the notion of being connected by a path, and then a Kan composition operation that expresses that being extensible is preserved along paths. We also define a new operation on types which expresses that this notion of extensibility is preserved by equivalences. The axiom of univalence, and composition for the universe, are then both expressible using this new operation.

The paper is organized as follows. The first part, Sections 2 to 7, presents the type system. The second part, Section 8, provides its semantics in cubical sets. Finally, in Section 9, we present two possible extensions: the addition of an identity type, and two examples of higher inductive types.

2 Basic Type Theory

In this section we introduce the version of dependent type theory on which the rest of the paper is based. This presentation is standard, but included for completeness. The type theory that we consider has a type of natural numbers, but no universes (we consider the addition of universes in Section 7). It also has β and η-conversion for dependent functions and surjective pairing for dependent pairs.

The syntax of contexts, terms and types is specified by:

$$
\begin{array}{lcll}
\Gamma, \Delta & ::= & () \mid \Gamma, x : A & \text{Contexts} \\[1em]
t, u, A, B & ::= & x \mid \lambda x : A.\, t \mid t\; u \mid (x : A) \to B & \Pi\text{-types} \\
 & \mid & (t, u) \mid t.1 \mid t.2 \mid (x : A) \times B & \Sigma\text{-types} \\
 & \mid & 0 \mid \mathsf{s}\; u \mid \mathsf{natrec}\; t\; u \mid \mathsf{N} & \text{Natural numbers}
\end{array}
$$

We write $A \to B$ for the non-dependent function space and $A \times B$ for the type of non-dependent pairs. Terms and types are considered up to α-equivalence of bound variables. Substitutions, written $\sigma = (x_1/u_1, \ldots, x_n/u_n)$, are defined to act on expressions as usual, i.e., simultaneously replacing x_i by u_i, renaming bound variables whenever necessary. The inference rules of this system are presented in Figure 1 where in the η-rule for Π- and Σ-types we omitted the premises that t and u should have the respective type.

---

[1] We have implemented a type-checker for this system in Haskell, which is available at: `https://github.com/mortberg/cubicaltt`

Well-formed contexts, $\Gamma \vdash$ (The condition $x \notin \mathrm{dom}(\Gamma)$ means that x is not declared in Γ)

$$\frac{}{() \vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash} \; (x \notin \mathrm{dom}(\Gamma))$$

Well-formed types, $\Gamma \vdash A$

$$\frac{\Gamma, x : A \vdash B}{\Gamma \vdash (x : A) \to B} \qquad \frac{\Gamma, x : A \vdash B}{\Gamma \vdash (x : A) \times B} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{N}}$$

Well-typed terms, $\Gamma \vdash t : A$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : (x : A) \to B} \qquad \frac{\Gamma \vdash}{\Gamma \vdash x : A} \; (x : A \in \Gamma)$$

$$\frac{\Gamma \vdash t : (x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B(x/u)} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash t.1 : A} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash t.2 : B(x/t.1)}$$

$$\frac{\Gamma, x : A \vdash B \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B(x/t)}{\Gamma \vdash (t, u) : (x : A) \times B} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathsf{N}} \qquad \frac{\Gamma \vdash n : \mathsf{N}}{\Gamma \vdash \mathsf{s}\, n : \mathsf{N}}$$

$$\frac{\Gamma, x : \mathsf{N} \vdash P \qquad \Gamma \vdash a : P(x/0) \qquad \Gamma \vdash b : (n : \mathsf{N}) \to P(x/n) \to P(x/\mathsf{s}\, n)}{\Gamma \vdash \mathsf{natrec}\; a\; b : (x : \mathsf{N}) \to P}$$

Type equality, $\Gamma \vdash A = B$ (Congruence and equivalence rules which are omitted)

Term equality, $\Gamma \vdash a = b : A$ (Congruence and equivalence rules are omitted)

$$\frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t = u : B} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x : A.\, t)\; u = t(x/u) : B(x/u)}$$

$$\frac{\Gamma, x : A \vdash t\; x = u\; x : B}{\Gamma \vdash t = u : (x : A) \to B} \qquad \frac{\Gamma, x : A \vdash B \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B(x/t)}{\Gamma \vdash (t, u).1 = t : A}$$

$$\frac{\Gamma, x : A \vdash B \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B(x/t)}{\Gamma \vdash (t, u).2 = u : B(x/t)}$$

$$\frac{\Gamma, x : A \vdash B \qquad \Gamma \vdash t.1 = u.1 : A \qquad \Gamma \vdash t.2 = u.2 : B(x/t.1)}{\Gamma \vdash t = u : (x : A) \times B}$$

$$\frac{\Gamma, x : \mathsf{N} \vdash P \qquad \Gamma \vdash a : P(x/0) \qquad \Gamma \vdash b : (n : \mathsf{N}) \to P(x/n) \to P(x/\mathsf{s}\, n)}{\Gamma \vdash \mathsf{natrec}\; a\; b\; 0 = a : P(x/0)}$$

$$\frac{\Gamma, x : \mathsf{N} \vdash P \qquad \Gamma \vdash a : P(x/0) \qquad \Gamma \vdash b : (n : \mathsf{N}) \to P(x/n) \to P(x/\mathsf{s}\, n) \qquad \Gamma \vdash n : \mathsf{N}}{\Gamma \vdash \mathsf{natrec}\; a\; b\; (\mathsf{s}\; n) = b\; n\; (\mathsf{natrec}\; a\; b\; n) : P(x/\mathsf{s}\, n)}$$

Figure 1 Inference rules of the basic type theory.

We define $\Delta \vdash \sigma : \Gamma$ by induction on Γ. Given $\Delta \vdash$ we have $\Delta \vdash () : ()$ (the empty substitution), and $\Delta \vdash (\sigma, x/u) : \Gamma, x : A$ if $\Delta \vdash \sigma : \Gamma$ and $\Delta \vdash u : A\sigma$.

We write J for an arbitrary judgment and, as usual, we consider also *hypothetical* judgments $\Gamma \vdash \mathsf{J}$ in a *context* Γ.

The following lemma will be valid for all extensions of type theory we consider below.

▶ **Lemma 1.** *Substitution is admissible:*

$$\frac{\Gamma \vdash \mathsf{J} \qquad \Delta \vdash \sigma : \Gamma}{\Delta \vdash \mathsf{J}\sigma}$$

In particular, weakening is admissible, i.e., a judgment valid in a context stays valid in any extension of this context.

3 Path Types

As in [6, 22] we assume that we are given a discrete infinite set of names (representing directions) $i, j, k, \ldots$ We define $\mathbb{I}$ to be the free de Morgan algebra [3] on this set of names. This means that $\mathbb{I}$ is a bounded distributive lattice with top element 1 and bottom element 0 with an involution $1 - r$ satisfying:

$$1 - 0 = 11 - 1 = 01 - (r \vee s) = (1 - r) \wedge (1 - s)1 - (r \wedge s) = (1 - r) \vee (1 - s)$$

The elements of $\mathbb{I}$ can hence be described by the following grammar:

$$r, s \quad ::= \quad 0 \mid 1 \mid i \mid 1 - r \mid r \wedge s \mid r \vee s$$

The set $\mathbb{I}$ also has decidable equality, and as a distributive lattice, it can be described as the free distributive lattice generated by symbols i and $1 - i$ [3]. As in [6], the elements in $\mathbb{I}$ can be thought as formal representations of elements in $[0, 1]$, with $r \wedge s$ representing $min(r, s)$ and $r \vee s$ representing $max(r, s)$. With this in mind it is clear that $(1 - r) \wedge r \neq 0$ and $(1 - r) \vee r \neq 1$ in general.

▶ Remark. We could instead also use a so-called Kleene algebra [15], i.e., a de Morgan algebra satisfying in addition $r \wedge (1 - r) \leqslant s \vee (1 - s)$. The free Kleene algebra on the set of names can be described as above but by additionally imposing the equations $i \wedge (1 - i) \leqslant j \vee (1 - j)$ on the generators; this still has a decidable equality. Note that $[0, 1]$ with the operations described above is a Kleene algebra. With this added condition, $r = s$ if, and only if, their interpretations in $[0, 1]$ are equal. A consequence of using a Kleene algebra instead would be that more terms would be judgmentally equal in the type theory.

3.1 Syntax and Inference Rules

Contexts can now be extended with name declarations:

$$\Gamma, \Delta \quad ::= \quad \ldots \mid \Gamma, i : \mathbb{I}$$

together with the context rule:

$$\frac{\Gamma \vdash}{\Gamma, i : \mathbb{I} \vdash} \; (i \notin \mathrm{dom}(\Gamma))$$

A judgment of the form $\Gamma \vdash r : \mathbb{I}$ means that $\Gamma \vdash$ and r in $\mathbb{I}$ depends only on the names declared in Γ. The judgment $\Gamma \vdash r = s : \mathbb{I}$ means that r and s are equal as elements of $\mathbb{I}$,

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Path}\ A\ t\ u} \qquad \frac{\Gamma \vdash A \qquad \Gamma, i : \mathbb{I} \vdash t : A}{\Gamma \vdash \langle i \rangle\ t : \mathsf{Path}\ A\ t(i0)\ t(i1)}$$

$$\frac{\Gamma \vdash t : \mathsf{Path}\ A\ u_0\ u_1 \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash t\ r : A} \qquad \frac{\Gamma \vdash A \qquad \Gamma, i : \mathbb{I} \vdash t : A \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash (\langle i \rangle\ t)\ r = t(i/r) : A}$$

$$\frac{\Gamma, i : \mathbb{I} \vdash t\ i = u\ i : A}{\Gamma \vdash t = u : \mathsf{Path}\ A\ u_0\ u_1} \qquad \frac{\Gamma \vdash t : \mathsf{Path}\ A\ u_0\ u_1}{\Gamma \vdash t\ 0 = u_0 : A} \qquad \frac{\Gamma \vdash t : \mathsf{Path}\ A\ u_0\ u_1}{\Gamma \vdash t\ 1 = u_1 : A}$$

Figure 2 Inference rules for path types.

$\Gamma \vdash r : \mathbb{I}$, and $\Gamma \vdash s : \mathbb{I}$. Note, that judgmental equality for $\mathbb{I}$ will be re-defined once we introduce restricted contexts in Section 4.

The extension to the syntax of basic dependent type theory is:

$$\begin{array}{lcll} t, u, A, B & ::= & \dots & \\ & \mid & \mathsf{Path}\ A\ t\ u \mid \langle i \rangle\ t \mid t\ r & \text{Path types} \end{array}$$

Path abstraction, $\langle i \rangle\ t$, binds the name i in t, and path application, $t\ r$, applies a term t to an element $r : \mathbb{I}$. This is similar to the notion of name-abstraction in nominal sets [21].

The substitution operation now has to be extended to substitutions of the form (i/r). There are special substitutions of the form $(i/0)$ and $(i/1)$ corresponding to taking faces of an n-dimensional cube, we write these simply as $(i0)$ and $(i1)$.

The inference rules for path types are presented in Figure 2 where again in the η-rule we omitted that t and u should be appropriately typed.

We define $1_a : \mathsf{Path}\ A\ a\ a$ as $1_a = \langle i \rangle\ a$, which corresponds to a proof of reflexivity.

The intuition is that a type in a context with n names corresponds to an n-dimensional cube:

$() \vdash A$	$\bullet\ A$
$i : \mathbb{I} \vdash A$	$A(i0) \xrightarrow{A} A(i1)$
$i : \mathbb{I}, j : \mathbb{I} \vdash A$	$\begin{array}{ccc} A(i0)(j1) & \xrightarrow{A(j1)} & A(i1)(j1) \\ {\scriptstyle A(i0)} \uparrow & & \uparrow {\scriptstyle A(i1)} \\ A(i0)(j0) & \xrightarrow[A(j0)]{} & A(i1)(j0) \end{array}$
$\vdots$	$\vdots$

Note that $A(i0)(j0) = A(j0)(i0)$. The substitution (i/j) corresponds to renaming a dimension, while $(i/1-i)$ corresponds to the inversion of a path. If we have $i : \mathbb{I} \vdash p$ with $p(i0) = a$ and $p(i1) = b$ then it can be seen as a line

$$a \xrightarrow{p} b$$

in direction i, then:

$$b \xrightarrow{p(i/1-i)} a$$

The substitutions $(i/i \wedge j)$ and $(i/i \vee j)$ correspond to special kinds of degeneracies called *connections* [7]. The connections $p(i/i \wedge j)$ and $p(i/i \vee j)$ can be drawn as the squares:

$$
\begin{array}{ccc}
a & \xrightarrow{p} & b \\
{\scriptstyle p(i0)}\uparrow & p(i/i \wedge j) & \uparrow{\scriptstyle p(i/j)} \\
a & \xrightarrow[p(i0)]{} & a
\end{array}
\qquad
\begin{array}{ccc}
b & \xrightarrow{p(i1)} & b \\
{\scriptstyle p(i/j)}\uparrow & p(i/i \vee j) & \uparrow{\scriptstyle p(i1)} \\
a & \xrightarrow[p]{} & b
\end{array}
\qquad
\begin{array}{l}
j \uparrow \\
\quad \xrightarrow[i]{}
\end{array}
$$

where, for instance, the right-hand side of the left square is computed as

$$p(i/i \wedge j)(i1) \;=\; p(i/1 \wedge j) \;=\; p(i/j)$$

and the bottom and left-hand sides are degenerate.

3.2 Examples

Representing equalities using path types allows novel definitions of many standard operations on identity types that are usually proved by identity elimination. For instance, the fact that the images of two equal elements are equal can be defined as:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash f : A \to B \qquad \Gamma \vdash p : \mathsf{Path}\ A\ a\ b}{\Gamma \vdash \langle i \rangle\ f\ (p\ i) : \mathsf{Path}\ B\ (f\ a)\ (f\ b)}$$

This operation satisfies some judgmental equalities that do not hold judgmentally when the identity type is defined as an inductive family (see Section 7.2 of [6] for details).

We can also define new operations, for instance, function extensionality for path types can be proved as:

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash g : (x : A) \to B \qquad \Gamma \vdash p : (x : A) \to \mathsf{Path}\ B\ (f\ x)\ (g\ x)}{\Gamma \vdash \langle i \rangle\ \lambda x : A.\ p\ x\ i : \mathsf{Path}\ ((x : A) \to B)\ f\ g}$$

To see that this is correct we check that the term has the correct faces, for instance:

$$(\langle i \rangle\ \lambda x : A.\ p\ x\ i)\ 0 \;=\; \lambda x : A.\ p\ x\ 0 \;=\; \lambda x : A.\ f\ x \;=\; f$$

We can also justify the fact that singletons are contractible, that is, that any element in $(x : A) \times (\mathsf{Path}\ A\ a\ x)$ is equal to $(a, 1_a)$:

$$\frac{\Gamma \vdash p : \mathsf{Path}\ A\ a\ b}{\Gamma \vdash \langle i \rangle\ (p\ i, \langle j \rangle\ p\ (i \wedge j)) : \mathsf{Path}\ ((x : A) \times (\mathsf{Path}\ A\ a\ x))\ (a, 1_a)\ (b, p)}$$

As in the previous work [6, 13] we need to add *composition operations*, defined by induction on the type, in order to justify the elimination principle for paths.

4 Systems, Composition, and Transport

In this section we define the operation of context *restriction* which will allow us to describe new geometrical shapes corresponding to “sub-polyhedra” of a cube. Using this we can define the composition operation. From this operation we will also be able to define the transport operation and the elimination principle for Path types.

4.1 The Face Lattice

The *face lattice*, $\mathbb{F}$, is the distributive lattice generated by symbols $(i = 0)$ and $(i = 1)$ with the relation $(i = 0) \wedge (i = 1) = 0_\mathbb{F}$. The elements of the face lattice, called *face formulas*, can be described by the grammar

$$\varphi, \psi \quad ::= \quad 0_\mathbb{F} \mid 1_\mathbb{F} \mid (i = 0) \mid (i = 1) \mid \varphi \wedge \psi \mid \varphi \vee \psi$$

There is a canonical lattice map $\mathbb{I} \to \mathbb{F}$ sending i to $(i = 1)$ and $1 - i$ to $(i = 0)$. We write $(r = 1)$ for the image of $r : \mathbb{I}$ in $\mathbb{F}$ and we write $(r = 0)$ for $(1 - r = 1)$. We have $(r = 1) \wedge (r = 0) = 0_\mathbb{F}$ and we define the lattice map $\mathbb{F} \to \mathbb{F}$, $\psi \longmapsto \psi(i/r)$ sending $(i = 1)$ to $(r = 1)$ and $(i = 0)$ to $(r = 0)$.

Any element of $\mathbb{F}$ is the join of the irreducible elements below it. An irreducible element of this lattice is a *face*, i.e., a conjunction of elements of the form $(i = 0)$ and $(j = 1)$. This provides a disjunctive normal form for face formulas, and it follows from this that the equality on $\mathbb{F}$ is decidable.

Geometrically, the face formulas describe "sub-polyhedra" of a cube. For instance, the element $(i = 0) \vee (j = 1)$ can be seen as the union of two faces of the square in directions j and i. If I is a finite set of names, we define the *boundary* of I as the element ∂_I of $\mathbb{F}$ which is the disjunction of all $(i = 0) \vee (i = 1)$ for i in I. It is the greatest element depending at most on elements in I which is $< 1_\mathbb{F}$.

We write $\Gamma \vdash \psi : \mathbb{F}$ to mean that ψ is a face formula using only the names declared in Γ. We introduce then the new *restriction* operation on contexts:

$$\Gamma, \Delta \quad ::= \quad \ldots \mid \Gamma, \varphi$$

together with the rule:

$$\frac{\Gamma \vdash \varphi : \mathbb{F}}{\Gamma, \varphi \vdash}$$

This allows us to describe new geometrical shapes: as we have seen above, a type in a context $\Gamma = i : \mathbb{I}, j : \mathbb{I}$ can be thought of as a square, and a type in the restricted context Γ, φ will then represent a compatible union of faces of this square. This can be illustrated by:

$i : \mathbb{I}, (i = 0) \vee (i = 1) \vdash A$	$A(i0) \bullet \qquad A(i1) \bullet$
$i : \mathbb{I}, j : \mathbb{I}, (i = 0) \vee (j = 1) \vdash A$	$A(i0)(j0) \xrightarrow{A(i0)} A(i0)(j1) \xrightarrow{A(j1)} A(i1)(j1)$
$i : \mathbb{I}, j : \mathbb{I}, (i = 0) \vee (i = 1) \vee (j = 0) \vdash A$	$A(i0)(j1) \xleftarrow{A(i0)} A(i0)(j0) \xrightarrow{A(j0)} A(i1)(j0) \xrightarrow{A(i1)} A(i1)(j1)$

There is a canonical map from the lattice $\mathbb{F}$ to the congruence lattice of $\mathbb{I}$, which is distributive [3], sending $(i = 1)$ to the congruence identifying i with 1 (and $1 - i$ with 0) and sending

$(i = 0)$ to the congruence identifying i with 0 (and $1 - i$ with 1). In this way, any element ψ of $\mathbb{F}$ defines a congruence $r = s$ $(\mathsf{mod}.\ \psi)$ on $\mathbb{I}$.

This congruence can be described as a substitution if ψ is irreducible; for instance, if ψ is $(i = 0) \wedge (j = 1)$ then $r = s$ $(\mathsf{mod}.\ \psi)$ is equivalent to $r(i0)(j1) = s(i0)(j1)$. The congruence associated to $\psi = \varphi_0 \vee \varphi_1$ is the meet of the congruences associated to φ_0 and φ_1 respectively, so that we have, e.g., $i = 1 - j$ $(\mathsf{mod}.\ \psi)$ if $\varphi_0 = (i = 0) \wedge (j = 1)$ and $\varphi_1 = (i = 1) \wedge (j = 0)$.

To any context Γ we can associate recursively a congruence on $\mathbb{I}$, the congruence on Γ, ψ being the join of the congruence defined by Γ and the congruence defined by ψ. The congruence defined by $()$ is equality in $\mathbb{I}$, and an extension $x : A$ or $i : \mathbb{I}$ does not change the congruence. The judgment $\Gamma \vdash r = s : \mathbb{I}$ then means that $r = s$ $(\mathsf{mod}.\ \Gamma)$, $\Gamma \vdash r : \mathbb{I}$, and $\Gamma \vdash s : \mathbb{I}$.

In the case where Γ does not use the restriction operation, this judgment means $r = s$ in $\mathbb{I}$. If i is declared in Γ, then $\Gamma, (i = 0) \vdash r = s : \mathbb{I}$ is equivalent to $\Gamma \vdash r(i0) = s(i0) : \mathbb{I}$. Similarly any context Γ defines a congruence on $\mathbb{F}$ with $\Gamma, \psi \vdash \varphi_0 = \varphi_1 : \mathbb{F}$ being equivalent to $\Gamma \vdash \psi \wedge \varphi_0 = \psi \wedge \varphi_1 : \mathbb{F}$.

As explained above, the elements of $\mathbb{I}$ can be seen as formal representations of elements in the interval $[0, 1]$. The elements of $\mathbb{F}$ can then be seen as formulas on elements of $[0, 1]$. We have a simple form of *quantifier elimination* on $\mathbb{F}$: given a name i, we define $\forall i \colon \mathbb{F} \to \mathbb{F}$ as the lattice morphism sending $(i = 0)$ and $(i = 1)$ to $0_{\mathbb{F}}$, and being the identity on all the other generators. If ψ is independent of i, we have $\psi \leqslant \varphi$ if, and only if, $\psi \leqslant \forall i.\varphi$. For example, if φ is $(i = 0) \vee ((i = 1) \wedge (j = 0)) \vee (j = 1)$, then $\forall i.\varphi$ is $(j = 1)$. This operation will play a crucial role in Section 6.2 for the definition of composition of glueing.

Since $\mathbb{F}$ is not a Boolean algebra, we don't have in general $\varphi = (\varphi \wedge (i = 0)) \vee (\varphi \wedge (i = 1))$, but we always have the following decomposition:

▶ **Lemma 2.** *For any element φ of $\mathbb{F}$ and any name i we have*

$$\varphi = (\forall i.\varphi) \vee (\varphi \wedge (i = 0)) \vee (\varphi \wedge (i = 1))$$

We also have $\varphi \wedge (i = 0) \leqslant \varphi(i0)$ and $\varphi \wedge (i = 1) \leqslant \varphi(i1)$.

4.2 Syntax and Inference Rules for Systems

Systems allow to introduce "sub-polyhedra" as compatible unions of cubes. The extension to the syntax of dependent type theory with path types is:

$$\begin{array}{rcll} t, u, A, B & ::= & \ldots & \\ & \mid & [\, \varphi_1\ t_1, \ldots, \varphi_n\ t_n \,] & \text{Systems} \end{array}$$

We allow $n = 0$ and get the empty system $[\,]$. As explained above, a context now corresponds in general to the union of sub-faces of a cube. In Figure 3 we provide operations for combining compatible systems of types and elements, the side condition for these rules is that $\Gamma \vdash \varphi_1 \vee \cdots \vee \varphi_n = 1_{\mathbb{F}} : \mathbb{F}$. This condition requires Γ to be sufficiently restricted: for example $\Delta, (i = 0) \vee (i = 1) \vdash (i = 0) \vee (i = 1) = 1_{\mathbb{F}}$. The first rule introduces systems of types, each defined on one φ_l and requiring the types to agree whenever they overlap; the second rule is the analogous rule for terms. The last two rules make sure that systems have the correct faces. The third inference rule says that any judgment which is valid locally at each φ_l is valid; note that in particular $n = 0$ is allowed (then the side condition becomes $\Gamma \vdash 0_{\mathbb{F}} = 1_{\mathbb{F}} : \mathbb{F}$).

$$\frac{\Gamma, \varphi_1 \vdash A_1 \quad \cdots \quad \Gamma, \varphi_n \vdash A_n \qquad \Gamma, \varphi_i \wedge \varphi_j \vdash A_i = A_j \quad (1 \leqslant i, j \leqslant n)}{\Gamma \vdash [\, \varphi_1 \; A_1, \ldots, \varphi_n \; A_n \,]}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, \varphi_1 \vdash t_1 : A \quad \cdots \quad \Gamma, \varphi_n \vdash t_n : A \qquad \Gamma, \varphi_i \wedge \varphi_j \vdash t_i = t_j : A \quad (1 \leqslant i, j \leqslant n)}{\Gamma \vdash [\, \varphi_1 \; t_1, \ldots, \varphi_n \; t_n \,] : A}$$

$$\frac{\Gamma, \varphi_1 \vdash \mathsf{J} \quad \cdots \quad \Gamma, \varphi_n \vdash \mathsf{J}}{\Gamma \vdash \mathsf{J}} \qquad \frac{\Gamma \vdash [\, \varphi_1 \; A_1, \ldots, \varphi_n \; A_n \,] \qquad \Gamma \vdash \varphi_i = 1_{\mathbb{F}} : \mathbb{F}}{\Gamma \vdash [\, \varphi_1 \; A_1, \ldots, \varphi_n \; A_n \,] = A_i}$$

$$\frac{\Gamma \vdash [\, \varphi_1 \; t_1, \ldots, \varphi_n \; t_n \,] : A \qquad \Gamma \vdash \varphi_i = 1_{\mathbb{F}} : \mathbb{F}}{\Gamma \vdash [\, \varphi_1 \; t_1, \ldots, \varphi_n \; t_n \,] = t_i : A}$$

Figure 3 Inference rules for systems with side condition $\Gamma \vdash \varphi_1 \vee \cdots \vee \varphi_n = 1_{\mathbb{F}} : \mathbb{F}$.

Note that when $n = 0$ the second of the above rules should be read as: if $\Gamma \vdash 0_{\mathbb{F}} = 1_{\mathbb{F}} : \mathbb{F}$ and $\Gamma \vdash A$, then $\Gamma \vdash [\,] : A$.

We extend the definition of the substitution judgment by $\Delta \vdash \sigma : \Gamma, \varphi$ if $\Delta \vdash \sigma : \Gamma$, $\Gamma \vdash \varphi : \mathbb{F}$, and $\Delta \vdash \varphi\sigma = 1_{\mathbb{F}} : \mathbb{F}$.

If $\Gamma, \varphi \vdash u : A$, then $\Gamma \vdash a : A[\varphi \mapsto u]$ is an abbreviation for $\Gamma \vdash a : A$ and $\Gamma, \varphi \vdash a = u : A$. In this case, we see this element a as a witness that the partial element u, defined on the "extent" φ (using the terminology from [10]), is *extensible*. More generally, we write $\Gamma \vdash a : A[\varphi_1 \mapsto u_1, \ldots, \varphi_k \mapsto u_k]$ for $\Gamma \vdash a : A$ and $\Gamma, \varphi_l \vdash a = u_l : A$ for $l = 1, \ldots, k$.

For instance, if $\Gamma, i : \mathbb{I} \vdash A$ and $\Gamma, i : \mathbb{I}, \varphi \vdash u : A$ where $\varphi = (i = 0) \vee (i = 1)$ then the element u is determined by two elements $\Gamma \vdash a_0 : A(i0)$ and $\Gamma \vdash a_1 : A(i1)$ and an element $\Gamma, i : \mathbb{I} \vdash a : A[(i = 0) \mapsto a_0, (i = 1) \mapsto a_1]$ gives a path connecting a_0 and a_1.

▶ **Lemma 3.** *The following rules are admissible:*[2]

$$\frac{\Gamma \vdash \varphi \leqslant \psi : \mathbb{F} \qquad \Gamma, \psi \vdash \mathsf{J}}{\Gamma, \varphi \vdash \mathsf{J}} \qquad \frac{\Gamma, 1_{\mathbb{F}} \vdash \mathsf{J}}{\Gamma \vdash \mathsf{J}} \qquad \frac{\Gamma, \varphi, \psi \vdash \mathsf{J}}{\Gamma, \varphi \wedge \psi \vdash \mathsf{J}}$$

Furthermore, if φ is independent of i, the following rules are admissible

$$\frac{\Gamma, i : \mathbb{I}, \varphi \vdash \mathsf{J}}{\Gamma, \varphi, i : \mathbb{I} \vdash \mathsf{J}}$$

and it follows that we have in general:

$$\frac{\Gamma, i : \mathbb{I}, \varphi \vdash \mathsf{J}}{\Gamma, \forall i.\varphi, i : \mathbb{I} \vdash \mathsf{J}}$$

4.3 Composition Operation

The syntax of compositions is given by:

$$\begin{array}{lcll} t, u, A, B & ::= & \ldots & \\ & \mid & \mathsf{comp}^i \; A \; [\varphi \mapsto u] \; a_0 & \text{Compositions} \end{array}$$

[2] The inference rules with double line are each a pair of rules, because they can be used in both directions.

where u is a system on the extent φ.

The composition operation expresses that being extensible is preserved along paths: if a partial path is extensible at 0, then it is extensible at 1.

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A(i0)[\varphi \mapsto u(i0)]}{\Gamma \vdash \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0 : A(i1)[\varphi \mapsto u(i1)]}$$

Note that comp^i binds i in A and u and that we have in particular the following equality judgments for systems:

$$\Gamma \vdash \mathsf{comp}^i\ A\ [1_{\mathbb{F}} \mapsto u]\ a_0 = u(i1) : A(i1)$$

If we have a substitution $\Delta \vdash \sigma : \Gamma$, then

$$(\mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0)\sigma = \mathsf{comp}^j\ A(\sigma, i/j)\ [\varphi\sigma \mapsto u(\sigma, i/j)]\ a_0\sigma$$

where j is fresh for Δ, which corresponds semantically to the *uniformity* [6, 13] of the composition operation.

We use the abbreviation $[\varphi_1 \mapsto u_1, \ldots, \varphi_n \mapsto u_n]$ for $[\bigvee_l \varphi_l \mapsto [\varphi_1\ u_1, \ldots, \varphi_n\ u_n]]$ and in particular we write $[]$ for $[0_{\mathbb{F}} \mapsto [\ ]]$.

▶ **Example 4.** With composition we can justify transitivity of path types:

$$\frac{\Gamma \vdash p : \mathsf{Path}\ A\ a\ b \qquad \Gamma \vdash q : \mathsf{Path}\ A\ b\ c}{\Gamma \vdash \langle i \rangle\ \mathsf{comp}^j\ A\ [(i=0) \mapsto a, (i=1) \mapsto q\ j]\ (p\ i) : \mathsf{Path}\ A\ a\ c}$$

This composition can be visualized as the dashed arrow in the square:

$$\begin{array}{ccc} a & \dashrightarrow & c \\ {\scriptstyle a}\uparrow & & \uparrow{\scriptstyle q\ j} \\ a & \xrightarrow[p\ i]{} & b \end{array} \qquad \begin{array}{l} j \uparrow \\ \quad \longrightarrow i \end{array}$$

4.4 Kan Filling Operation

As we have connections we also get Kan filling operations from compositions:

$$\Gamma, i : \mathbb{I} \vdash \mathsf{fill}^i\ A\ [\varphi \mapsto u]\ a_0 = \mathsf{comp}^j\ A(i/i \wedge j)\ [\varphi \mapsto u(i/i \wedge j), (i=0) \mapsto a_0]\ a_0 : A$$

where j is fresh for Γ. The element $\Gamma, i : \mathbb{I} \vdash v = \mathsf{fill}^i\ A\ [\varphi \mapsto u]\ a_0 : A$ satisfies:

$$\Gamma \vdash v(i0) = a_0 : A(i0) \quad \Gamma \vdash v(i1) = \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0 : A(i1) \quad \Gamma, \varphi, i : \mathbb{I} \vdash v = u : A$$

This means that we can not only compute the lid of an open box but also its filling. If φ is the boundary formula on the names declared in Γ, we recover the Kan operation for cubical sets [16].

4.5 Equality Judgments for Composition

The equality judgments for $\mathsf{comp}^i\ C\ [\varphi \mapsto u]\ a_0$ are defined by cases on the type C which depends on i, i.e., $\Gamma, i : \mathbb{I} \vdash C$. The right hand side of the definitions are all equal to $u(i1)$ on the extent φ by the typing rule for compositions. There are four cases to consider:

Product types, $C = (x : A) \to B$

Given $\Gamma, \varphi, i : \mathbb{I} \vdash \mu : C$ and $\Gamma \vdash \lambda_0 : C(i0)[\varphi \mapsto \mu(i0)]$ the composition will be of type $C(i1)$. For $\Gamma \vdash u_1 : A(i1)$, we first let:

$$w = \mathsf{fill}^i \ A(i/1-i) \ [] \ u_1 \qquad \text{(in context } \Gamma, i : \mathbb{I} \text{ and of type } A(i/1-i))$$
$$v = w(i/1-i) \qquad \text{(in context } \Gamma, i : \mathbb{I} \text{ and of type } A)$$

Using this we define the equality judgment:

$$\Gamma \vdash (\mathsf{comp}^i \ C \ [\varphi \mapsto \mu] \ \lambda_0) \ u_1 = \mathsf{comp}^i \ B(x/v) \ [\varphi \mapsto \mu \ v] \ (\lambda_0 \ v(i0)) : B(x/v)(i1)$$

Sum types, $C = (x : A) \times B$

Given $\Gamma, \varphi, i : \mathbb{I} \vdash w : C$ and $\Gamma \vdash w_0 : C(i0)[\varphi \mapsto w(i0)]$ we let:

$$a = \mathsf{fill}^i \ A \ [\varphi \mapsto w.1] \ w_0.1 \qquad \text{(in context } \Gamma, i : \mathbb{I} \text{ and of type } A)$$
$$c_1 = \mathsf{comp}^i \ A \ [\varphi \mapsto w.1] \ w_0.1 \qquad \text{(in context } \Gamma \text{ and of type } A(i1))$$
$$c_2 = \mathsf{comp}^i \ B(x/a) \ [\varphi \mapsto w.2] \ w_0.2 \qquad \text{(in context } \Gamma \text{ and of type } B(x/a)(i1))$$

From which we define:

$$\Gamma \vdash \mathsf{comp}^i \ C \ [\varphi \mapsto w] \ w_0 = (c_1, c_2) : C(i1)$$

Natural numbers, $C = \mathsf{N}$

In this we define $\mathsf{comp}^i \ C \ [\varphi \mapsto n] \ n_0$ by recursion:

$$\Gamma \vdash \mathsf{comp}^i \ C \ [\varphi \mapsto \mathsf{0}] \ \mathsf{0} = \mathsf{0} : C$$
$$\Gamma \vdash \mathsf{comp}^i \ C \ [\varphi \mapsto \mathsf{s} \ n] \ (\mathsf{s} \ n_0) = \mathsf{s} \ (\mathsf{comp}^i \ C \ [\varphi \mapsto n] \ n_0) : C$$

Path types, $C = \mathsf{Path} \ A \ u \ v$

Given $\Gamma, \varphi, i : \mathbb{I} \vdash p : C$ and $\Gamma \vdash p_0 : C(i0)[\varphi \mapsto p(i0)]$ we define:

$$\Gamma \vdash \mathsf{comp}^i \ C \ [\varphi \mapsto p] \ p_0 = \langle j \rangle \ \mathsf{comp}^i \ A \ [\varphi \mapsto p \ j, (j=0) \mapsto u, (j=1) \mapsto v] \ (p_0 \ j) : C(i1)$$

4.6 Transport

Composition for $\varphi = 0_{\mathbb{F}}$ corresponds to transport:

$$\Gamma \vdash \mathsf{transp}^i \ A \ a = \mathsf{comp}^i \ A \ [] \ a : A(i1)$$

Together with the fact that singletons are contractible, from Section 3.2, we get the elimination principle for Path types in the same manner as explained for identity types in Section 7.2 of [6].

5 Derived Notions and Operations

This section defines various notions and operations that will be used for defining compositions for the glue operation in the next section. This operation will then be used to define the composition operation for the universe and to prove the univalence axiom.

5.1 Contractible Types

We define $\mathsf{isContr}\ A = (x : A) \times ((y : A) \to \mathsf{Path}\ A\ x\ y)$. A proof of $\mathsf{isContr}\ A$ witnesses the fact that A is *contractible*.

Given $\Gamma \vdash p : \mathsf{isContr}\ A$ and $\Gamma, \varphi \vdash u : A$ we define the operation[3]

$$\Gamma \vdash \mathsf{contr}\ p\ [\varphi \mapsto u] = \mathsf{comp}^i\ A\ [\varphi \mapsto p.2\ u\ i]\ p.1 : A[\varphi \mapsto u]$$

Conversely, we can state the following characterization of contractible types:

▶ **Lemma 5.** *Let $\Gamma \vdash A$ and assume that we have one operation*

$$\frac{\Gamma, \varphi \vdash u : A}{\Gamma \vdash \mathsf{contr}\ [\varphi \mapsto u] : A[\varphi \mapsto u]}$$

then we can find an element in $\mathsf{isContr}\ A$.

Proof. We define $x = \mathsf{contr}\ [] : A$ and prove that any element $y : A$ is path equal to x. For this, we introduce a fresh name $i : \mathbb{I}$ and define $\varphi = (i = 0) \vee (i = 1)$ and $u = [(i = 0) \mapsto x, (i = 1) \mapsto y]$. Using this we obtain $\Gamma, i : \mathbb{I} \vdash v = \mathsf{contr}\ [\varphi \mapsto u] : A[\varphi \mapsto u]$. In this way, we get a path $\langle i \rangle\ \mathsf{contr}\ [\varphi \mapsto u]$ connecting x and y. ◀

5.2 The pres Operation

The pres operation states that the image of a composition is path equal to the composition of the respective images, so that any function *preserves* composition, up to path equality.

▶ **Lemma 6.** *We have an operation:*

$$\frac{\Gamma, i : \mathbb{I} \vdash f : T \to A \qquad \Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, \varphi, i : \mathbb{I} \vdash t : T \qquad \Gamma \vdash t_0 : T(i0)[\varphi \mapsto t(i0)]}{\Gamma \vdash \mathsf{pres}^i\ f\ [\varphi \mapsto t]\ t_0 : (\mathsf{Path}\ A(i1)\ c_1\ c_2)[\varphi \mapsto \langle j \rangle\ (f\ t)(i1)]}$$

where $c_1 = \mathsf{comp}^i\ A\ [\varphi \mapsto f\ t]\ (f(i0)\ t_0)$ *and* $c_2 = f(i1)\ (\mathsf{comp}^i\ T\ [\varphi \mapsto t]\ t_0)$.

Proof. Let $\Gamma \vdash a_0 = f(i0)\ t_0 : A(i0)$ and $\Gamma, i : \mathbb{I} \vdash v = \mathsf{fill}^i\ T\ [\varphi \mapsto t]\ t_0 : T$. We take $\mathsf{pres}^i\ f\ [\varphi \mapsto t]\ t_0 = \langle j \rangle\ \mathsf{comp}^i\ A\ [\varphi \vee (j = 1) \mapsto f\ v]\ a_0$. ◀

Note that pres^i binds i in f and t.

5.3 The equiv Operation

We define $\mathsf{isEquiv}\ T\ A\ f = (y : A) \to \mathsf{isContr}\ ((x : T) \times \mathsf{Path}\ A\ y\ (f\ x))$ and $\mathsf{Equiv}\ T\ A = (f : T \to A) \times \mathsf{isEquiv}\ T\ A\ f$. If $f : \mathsf{Equiv}\ T\ A$ and $t : T$, we may write $f\ t$ for $f.1\ t$.

▶ **Lemma 7.** *If $\Gamma \vdash f : \mathsf{Equiv}\ T\ A$, we have an operation*

$$\frac{\Gamma, \varphi \vdash t : T \qquad \Gamma \vdash a : A \qquad \Gamma, \varphi \vdash p : \mathsf{Path}\ A\ a\ (f\ t)}{\Gamma \vdash \mathsf{equiv}\ f\ [\varphi \mapsto (t, p)]\ a : ((x : T) \times \mathsf{Path}\ A\ a\ (f\ x))[\varphi \mapsto (t, p)]}$$

Conversely, if $\Gamma \vdash f : T \to A$ and we have such an operation, then we can build a proof that f is an equivalence.

Proof. We define $\mathsf{equiv}\ f\ [\varphi \mapsto (t, p)]\ a = \mathsf{contr}\ (f.2\ a)\ [\varphi \mapsto (t, p)]$ using the contr operation defined above. The second statement follows from Lemma 5. ◀

---

[3] This expresses that the restriction map $\Gamma, \varphi \to \Gamma$ has the left lifting property w.r.t. any "trivial fibration", i.e., contractible extensions $\Gamma, x : A \to \Gamma$. The restriction maps $\Gamma, \varphi \to \Gamma$ thus represent "cofibrations" while the maps $\Gamma, x : A \to \Gamma$ represent "fibrations".

$$\frac{\Gamma \vdash A \qquad \Gamma, \varphi \vdash T \qquad \Gamma, \varphi \vdash f : \mathsf{Equiv}\ T\ A}{\Gamma \vdash \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A} \qquad \frac{\Gamma \vdash b : \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A}{\Gamma \vdash \mathsf{unglue}\ b : A[\varphi \mapsto f\ b]}$$

$$\frac{\Gamma, \varphi \vdash f : \mathsf{Equiv}\ T\ A \qquad \Gamma, \varphi \vdash t : T \qquad \Gamma \vdash a : A[\varphi \mapsto f\ t]}{\Gamma \vdash \mathsf{glue}\ [\varphi \mapsto t]\ a : \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A}$$

$$\frac{\Gamma \vdash T \qquad \Gamma \vdash f : \mathsf{Equiv}\ T\ A}{\Gamma \vdash \mathsf{Glue}\ [1_{\mathbb{F}} \mapsto (T, f)]\ A = T} \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash f : \mathsf{Equiv}\ T\ A}{\Gamma \vdash \mathsf{glue}\ [1_{\mathbb{F}} \mapsto t]\ (f\ t) = t : T}$$

$$\frac{\Gamma \vdash b : \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A}{\Gamma \vdash b = \mathsf{glue}\ [\varphi \mapsto b]\ (\mathsf{unglue}\ b) : \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A}$$

$$\frac{\Gamma, \varphi \vdash f : \mathsf{Equiv}\ T\ A \qquad \Gamma, \varphi \vdash t : T \qquad \Gamma \vdash a : A[\varphi \mapsto f\ t]}{\Gamma \vdash \mathsf{unglue}\ (\mathsf{glue}\ [\varphi \mapsto t]\ a) = a : A}$$

Figure 4 Inference rules for glueing.

6 Glueing

In this section, we introduce the glueing operation. This operation expresses that to be "extensible" is invariant by equivalence. From this operation, we can define a composition operation for universes, and prove the univalence axiom.

6.1 Syntax and Inference Rules for Glueing

We introduce the *glueing* construction at type and term level by:

$$\begin{array}{rcll} t, u, A, B & ::= & \dots & \\ & | & \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A & \text{Glue type} \\ & | & \mathsf{glue}\ [\varphi \mapsto t]\ u & \text{Glue term} \\ & | & \mathsf{unglue}\ [\varphi \mapsto f]\ u & \text{Unglue term} \end{array}$$

We may write simply $\mathsf{unglue}\ b$ for $\mathsf{unglue}\ [\varphi \mapsto f]\ b$. The inference rules for these are presented in Figure 4.

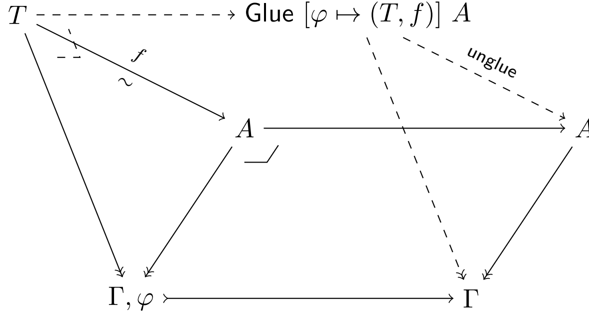
It follows from these rules that if $\Gamma \vdash b : \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A$, then $\Gamma, \varphi \vdash b : T$.

In the case $\varphi = (i = 0) \vee (i = 1)$ the glueing operation can be illustrated as the dashed line in:

$$\begin{array}{ccc} T_0 & \dashrightarrow & T_1 \\ {\scriptstyle f(i0)}\downarrow\wr & & \wr\downarrow{\scriptstyle f(i1)} \\ A(i0) & \xrightarrow[A]{} & A(i1) \end{array}$$

This illustrates why the operation is called glue: it *glues* together along a partial equivalence the partial type T and the total type A to a total type that extends T.

▶ Remark. In general $\mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A$ can be illustrated as:



This diagram suggests that a construction similar to Glue also appears in the simplicial set model. Indeed, the proof of Theorem 3.4.1 in [17] contains a similar diagram where $\overline{E}_1$ corresponds to $\mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A$.

▶ **Example 8.** Using glueing we can construct a path from an equivalence $\Gamma \vdash f : \mathsf{Equiv}\ A\ B$ by defining

$$\Gamma, i : \mathbb{I} \vdash E = \mathsf{Glue}\ [(i = 0) \mapsto (A, f), (i = 1) \mapsto (B, \mathsf{id}_B)]\ B$$

so that $E(i0) = A$ and $E(i1) = B$, where $\mathsf{id}_B : \mathsf{Equiv}\ B\ B$ is defined as:

$$\mathsf{id}_B = (\lambda x : B.\ x, \lambda x : B.\ ((x, 1_x), \lambda u : (y : B) \times \mathsf{Path}\ B\ x\ y.\ \langle i \rangle\ (u.2\ i, \langle j \rangle\ u.2\ (i \wedge j))))$$

In Section 7 we introduce a universe of types U and we will be able to define a function of type $(A\ B : \mathsf{U}) \to \mathsf{Equiv}\ A\ B \to \mathsf{Path}\ \mathsf{U}\ A\ B$ by:

$$\lambda A\ B : \mathsf{U}.\ \lambda f : \mathsf{Equiv}\ A\ B.\ \langle i \rangle\ \mathsf{Glue}\ [(i = 0) \mapsto (A, f), (i = 1) \mapsto (B, \mathsf{id}_B)]\ B$$

6.2 Composition for Glueing

We assume $\Gamma, i : \mathbb{I} \vdash B = \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A$, and define the composition in B. In order to do so, assume

$$\Gamma, \psi, i : \mathbb{I} \vdash b : B \qquad\qquad \Gamma \vdash b_0 : B(i0)[\psi \mapsto b(i0)]$$

and define:

$$a = \mathsf{unglue}\ b \qquad \text{(in context } \Gamma, \psi, i : \mathbb{I} \text{ and of type } A[\varphi \mapsto f\ b])$$
$$a_0 = \mathsf{unglue}\ b_0 \qquad \text{(in context } \Gamma \text{ and of type } A(i0)[\varphi(i0) \mapsto f(i0)\ b_0, \psi \mapsto a(i0)])$$

The following provides the algorithm for composition $\mathsf{comp}^i\ B\ [\psi \mapsto b]\ b_0 = b_1$ of type $B(i1)[\psi \mapsto b(i1)]$.

δ	$=$	$\forall i.\varphi$	Γ
a'_1	$=$	$\mathsf{comp}^i\ A\ [\psi \mapsto a]\ a_0$	Γ
t'_1	$=$	$\mathsf{comp}^i\ T\ [\psi \mapsto b]\ b_0$	Γ, δ
ω	$=$	$\mathsf{pres}^i\ f\ [\psi \mapsto b]\ b_0$	Γ, δ
(t_1, α)	$=$	$\mathsf{equiv}\ f(i1)\ [\delta \mapsto (t'_1, \omega), \psi \mapsto (b(i1), \langle j \rangle a'_1)]\ a'_1$	$\Gamma, \varphi(i1)$
a_1	$=$	$\mathsf{comp}^j\ A(i1)\ [\varphi(i1) \mapsto \alpha\ j, \psi \mapsto a(i1)]\ a'_1$	Γ
b_1	$=$	$\mathsf{glue}\ [\varphi(i1) \mapsto t_1]\ a_1$	Γ

We can check that whenever $\Gamma, i : \mathbb{I} \vdash \varphi = 1_{\mathbb{F}} : \mathbb{F}$ the definition of b_1 coincides with $\mathsf{comp}^i\ T\ [\psi \mapsto b]\ b_0$, which is consistent with the fact that $B = T$ in this case.

In the next section we will use the glue operation to define the composition for the universe and to prove the univalence axiom.

7 Universe and the Univalence Axiom

As in [20], we now introduce a universe U à la Russell by reflecting all typing rules and

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A}$$

In particular, we have $\Gamma \vdash \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A : \mathsf{U}$ whenever $\Gamma \vdash A : \mathsf{U}$, $\Gamma, \varphi \vdash T : \mathsf{U}$, and $\Gamma, \varphi \vdash f : \mathsf{Equiv}\ T\ A$.

7.1 Composition for the Universe

In order to describe the composition operation for the universe we first have to explain how to construct an equivalence from a line in the universe. Given $\Gamma \vdash A$, $\Gamma \vdash B$, and $\Gamma, i : \mathbb{I} \vdash E$, such that $E(i0) = A$ and $E(i1) = B$, we will construct $\mathsf{equiv}^i\ E : \mathsf{Equiv}\ A\ B$. In order to do this we first define

$$\begin{aligned}
f &= \lambda x : A.\ \mathsf{transp}^i\ E\ x && \text{(in context } \Gamma \text{ and of type } A \to B)\\
g &= \lambda y : B.\ (\mathsf{transp}^i\ E(i/1-i)\ y)(i/1-i) && \text{(in context } \Gamma \text{ and of type } B \to A)\\
u &= \lambda x : A.\ \mathsf{fill}^i\ E\ []\ x && \text{(in context } \Gamma, i : \mathbb{I} \text{ and of type } A \to E)\\
v &= \lambda y : B.\ (\mathsf{fill}^i\ E(i/1-i)\ []\ y)(i/1-i) && \text{(in context } \Gamma, i : \mathbb{I} \text{ and of type } B \to E)
\end{aligned}$$

such that:

$$u(i0) = \lambda x : A.x \qquad u(i1) = f \qquad v(i0) = g \qquad v(i1) = \lambda y : B.y$$

We will now prove that f is an equivalence. Given $y : B$ we see that $(x : A) \times \mathsf{Path}\ B\ y\ (f\ x)$ is inhabited as it contains the element $(g\ y, \langle j \rangle\ \theta_0(i1))$ where

$$\theta_0 = \mathsf{fill}^i\ E\ [(j=0) \mapsto v\ y, (j=1) \mapsto u\ (g\ y)]\ (g\ y).$$

Next, given an element (x, β) of $(x : A) \times \mathsf{Path}\ B\ y\ (f\ x)$ we will construct a path from $(g\ y, \langle j \rangle\ \theta_0(i1))$ to (x, β). Let

$$\theta_1 = \big(\mathsf{fill}^i\ E(i/1-i)\ [(j=0) \mapsto (v\ y)(i/1-i), (j=1) \mapsto (u\ x)(i/1-i)]\ (\beta\ j)\big)\ (i/1-i)$$

and $\omega = \theta_1(i0)$ so $\Gamma, i : \mathbb{I}, j : \mathbb{I} \vdash \theta_1 : E$, $\omega(j0) = g\ y$, and $\omega(j1) = x$. And further with

$$\delta = \mathsf{comp}^i\ E\ [(k=0) \mapsto \theta_0, (k=1) \mapsto \theta_1, (j=0) \mapsto v\ y, (j=1) \mapsto u\ \omega(j/k)]\ \omega(j/j \wedge k)$$

we obtain

$$\langle k \rangle\ (\omega(j/k), \langle j \rangle\ \delta) : \mathsf{Path}\ ((x : A) \times \mathsf{Path}\ B\ y\ (f\ x))\ (g\ y, \langle j \rangle\ \theta_0(i1))\ (x, \beta)$$

as desired. This concludes the proof that f is an equivalence and thus also the construction of $\mathsf{equiv}^i\ E : \mathsf{Equiv}\ A\ B$.

Using this we can now define the composition for the universe:

$$\Gamma \vdash \mathsf{comp}^i\ \mathsf{U}\ [\varphi \mapsto E]\ A_0 = \mathsf{Glue}\ [\varphi \mapsto (E(i1), \mathsf{equiv}^i\ E(i/1-i))]\ A_0 : \mathsf{U}$$

▶ Remark. Given $\Gamma, i : \mathbb{I} \vdash E$ we can also get an equivalence in $\mathsf{Equiv}\ A\ B$ (where $A = E(i0)$ and $B = E(i1)$) with a less direct description by

$$\Gamma \vdash \mathsf{transp}^i\ (\mathsf{Equiv}\ A\ E)\ \mathsf{id}_A : \mathsf{Equiv}\ A\ B$$

where id_A is the identity equivalence as given in Example 8.

7.2 The Univalence Axiom

Given $B = \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A$ the map $\mathsf{unglue} : B \to A$ extends f, in the sense that $\Gamma, \varphi \vdash \mathsf{unglue}\ b = f\ b : A$ if $\Gamma \vdash b : B$.

▶ **Theorem 9.** *The map* $\mathsf{unglue} : B \to A$ *is an equivalence.*

Proof. By Lemma 7 it suffices to construct

$$\tilde{b} : B[\psi \mapsto b] \qquad\qquad \tilde{\alpha} : \mathsf{Path}\ A\ u\ (\mathsf{unglue}\ \tilde{b})[\psi \mapsto \alpha]$$

given $\Gamma, \psi \vdash b : B$ and $\Gamma \vdash u : A$ and $\Gamma, \psi \vdash \alpha : \mathsf{Path}\ A\ u\ (\mathsf{unglue}\ b)$.

Since $\Gamma, \varphi \vdash f : T \to A$ is an equivalence and

$$\Gamma, \varphi, \psi \vdash b : T \qquad\qquad \Gamma, \varphi, \psi \vdash \alpha : \mathsf{Path}\ A\ u\ (f\ b)$$

we get, using Lemma 7

$$\Gamma, \varphi \vdash t : T[\psi \mapsto b] \qquad\qquad \Gamma, \varphi \vdash \beta : \mathsf{Path}\ A\ u\ (f\ t)\ [\psi \mapsto \alpha]$$

We then define $\tilde{a} = \mathsf{comp}^i\ A\ [\varphi \mapsto \beta\ i, \psi \mapsto \alpha\ i]\ u$, and using this we conclude by letting $\tilde{b} = \mathsf{glue}\ [\varphi \mapsto t]\ \tilde{a}$ and $\tilde{\alpha} = \mathsf{fill}^i\ A\ [\varphi \mapsto \beta\ i, \psi \mapsto \alpha\ i]\ u$. ◀

▶ **Corollary 10.** *For any type* $A : \mathsf{U}$ *the type* $C = (X : \mathsf{U}) \times \mathsf{Equiv}\ X\ A$ *is contractible.*[4]

Proof. It is enough by Lemma 5 to show that any partial element $\varphi \vdash (T, f) : C$ is path equal to the restriction of a total element. The map unglue extends f and is an equivalence by the previous theorem. Since any two elements of the type $\mathsf{isEquiv}\ X\ A\ f.1$ are path equal, this shows that any partial element of type C is path equal to the restriction of a total element. We can then conclude by Theorem 9. ◀

▶ **Corollary 11** (Univalence axiom)**.** *For any term*

$$t : (A\ B : \mathsf{U}) \to \mathsf{Path}\ \mathsf{U}\ A\ B \to \mathsf{Equiv}\ A\ B$$

the map $t\ A\ B : \mathsf{Path}\ \mathsf{U}\ A\ B \to \mathsf{Equiv}\ A\ B$ *is an equivalence.*

Proof. Both $(X : \mathsf{U}) \times \mathsf{Path}\ \mathsf{U}\ A\ X$ and $(X : \mathsf{U}) \times \mathsf{Equiv}\ A\ X$ are contractible. Hence the result follows from Theorem 4.7.7 in [26]. ◀

Two alternative proofs of univalence can be found in Appendix B.

---

[4] This formulation of the univalence axiom can be found in the message of Martín Escardó in: `https://groups.google.com/forum/#!msg/homotopytypetheory/HfCB_b-PNEU/Ibb48LvUMeUJ`
This is also used in the (classical) proofs of the univalence axiom, see Theorem 3.4.1 of [17] and Proposition 2.18 of [8], where an operation similar to the glueing operation appears implicitly.

8 Semantics

In this section we will explain the semantics of the type theory under consideration in cubical sets. We will first review how cubical sets, as a presheaf category, yield a model of basic type theory, and then explain the additional so-called composition structure we have to require to interpret the full cubical type theory.

8.1 The Category of Cubes and Cubical Sets

Consider the monad dM on the category of sets associating to each set the free de Morgan algebra on that set. The *category of cubes* $\mathcal{C}$ is the small category whose objects are finite subsets $I, J, K, \dots$ of a fixed, discrete, and countably infinite set, called *names*, and a morphism $\mathrm{Hom}(J, I)$ is a map $I \to \mathsf{dM}(J)$. Identities and compositions are inherited from the Kleisli category of dM, i.e., the identity on I is given by the unit $I \to \mathsf{dM}(I)$, and composition $fg \in \mathrm{Hom}(K, I)$ of $g \in \mathrm{Hom}(K, J)$ and $f \in \mathrm{Hom}(J, I)$ is given by $\mu_K \circ \mathsf{dM}(g) \circ f$ where $\mu_K \colon \mathsf{dM}(\mathsf{dM}(K)) \to \mathsf{dM}(K)$ denotes multiplication of dM. We will use f, g, h for morphisms in $\mathcal{C}$ and simply write $f \colon J \to I$ for $f \in \mathrm{Hom}(J, I)$. We will often write unions with commas and omit curly braces around finite sets of names, e.g., writing I, i, j for $I \cup \{i, j\}$ and $I - i$ for $I - \{i\}$ etc.

If i is in I and b is $0_{\mathbb{I}}$ or $1_{\mathbb{I}}$, we have maps (ib) in $\mathrm{Hom}(I - i, I)$ whose underlying map sends $j \neq i$ to itself and i to b. A *face map* is a composition of such maps. A *strict map* $\mathrm{Hom}(J, I)$ is a map $I \to \mathsf{dM}(J)$ which never takes the value $0_{\mathbb{I}}$ or $1_{\mathbb{I}}$. Any f can be uniquely written as a composition $f = gh$ where g is a face map and h is strict.

▶ **Definition 12.** A *cubical set* is a presheaf on $\mathcal{C}$.

Thus, a cubical set Γ is given by sets $\Gamma(I)$ for each $I \in \mathcal{C}$ and maps (called restrictions) $\Gamma(f) \colon \Gamma(I) \to \Gamma(J)$ for each $f \colon J \to I$. If we write $\Gamma(f)(\rho) = \rho f$ for $\rho \in \Gamma(I)$ (leaving the Γ implicit), these maps should satisfy $\rho\,\mathrm{id}_I = \rho$ and $(\rho f)g = \rho(fg)$ for $f \colon J \to I$ and $g \colon K \to J$.

Let us discuss some important examples of cubical sets. Using the canonical de Morgan algebra structure of the unit interval, $[0, 1]$, we can define a functor

$$\mathcal{C} \to \mathbf{Top}, \quad I \mapsto [0,1]^I. \tag{1}$$

If u is in $[0, 1]^I$ we can think of u as an environment giving values in $[0, 1]$ to each $i \in I$, so that iu is in $[0, 1]$ if $i \in I$. Since $[0, 1]$ is a de Morgan algebra, this extends uniquely to ru for $r \in \mathsf{dM}(I)$. So any $f \colon J \to I$ in $\mathcal{C}$ induces $f \colon [0, 1]^J \to [0, 1]^I$ by $i(fu) = (if)u$.

To any topological space X we can associate its *singular cubical set* $\mathrm{S}(X)$ by taking $\mathrm{S}(X)(I)$ to be the set of continuous functions $[0, 1]^I \to X$.

For a finite set of names I we get the formal cube $\mathbf{y}\,I$ where $\mathbf{y} \colon \mathcal{C} \to [\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ denotes the Yoneda embedding. Note that since $\mathbf{Top}$ is cocomplete the functor in (1) extends to a cocontinuous functor assigning to each cubical set its *geometric realization* as a topological space, in such a way that $\mathbf{y}\,I$ has $[0, 1]^I$ as its geometric realization.

The formal interval $\mathbb{I}$ induces a cubical set given by $\mathbb{I}(I) = \mathsf{dM}(I)$. The face lattice $\mathbb{F}$ induces a cubical set by taking as $\mathbb{F}(I)$ to be those $\varphi \in \mathbb{F}$ which only use symbols in I. The restrictions along $f \colon J \to I$ are in both cases simply *substituting* the symbols $i \in I$ by $f(i) \in \mathsf{dM}(J)$.

As any presheaf category, cubical sets have a subobject classifier Ω where $\Omega(I)$ is the set of sieves on I (i.e., subfunctors of $\mathbf{y}\,I$). Consider the natural transformation $(\cdot = 1) \colon \mathbb{I} \to \Omega$ where for $r \in \mathbb{I}(I)$, $(r = 1)$ is the sieve on I of all $f \colon J \to I$ such that $rf = 1_{\mathbb{I}}$. The image of $(\cdot = 1)$ is $\mathbb{F} \to \Omega$, assigning to each φ the sieve of all f with $\varphi f = 1_{\mathbb{F}}$.

8.2 Presheaf Semantics

The category of cubical sets (with morphisms being natural transformations) induce—as does any presheaf category—a category with families (CwF) [9] where the category of contexts and substitutions is the category of cubical sets. We will review the basic constructions but omit verification of the required equations (see, e.g., [12, 13, 6] for more details).

Basic Presheaf Semantics

As already mentioned the category of (semantic) contexts and substitutions is given by cubical sets and their maps. In this section we will use Γ, Δ to denote cubical sets and (semantic) substitutions by $\sigma \colon \Delta \to \Gamma$, overloading previous use of the corresponding meta-variables to emphasize their intended role.

Given a cubical set Γ, the types A in context Γ, written $A \in \mathrm{Ty}(\Gamma)$, are given by sets $A\rho$ for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ together with restriction maps $A\rho \to A(\rho f)$, $u \mapsto uf$ for $f \colon J \to I$ satisfying $u\,\mathrm{id}_I = u$ and $(uf)g = u(fg) \in A(\rho f g)$ if $g \colon K \to J$. Equivalently, $A \in \mathrm{Ty}(\Gamma)$ are the presheaves on the category of elements of Γ. For a type $A \in \mathrm{Ty}(\Gamma)$ its terms $a \in \mathrm{Ter}(\Gamma; A)$ are given by families of elements $a\rho \in A\rho$ for each $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ such that $(a\rho)f = a(\rho f)$ for $f \colon J \to I$. Note that our notation leaves a lot implicit; e.g., we should have written $A(I, \rho)$ for $A\rho$; $A(I, \rho, f)$ for the restriction map $A\rho \to A(\rho f)$; and $a(I, \rho)$ for $a\rho$.

For $A \in \mathrm{Ty}(\Gamma)$ and $\sigma \colon \Delta \to \Gamma$ we define $A\sigma \in \mathrm{Ty}(\Delta)$ by $(A\sigma)\rho = A(\sigma\rho)$ and the induced restrictions. If we also have $a \in \mathrm{Ter}(\Gamma; A)$, we define $a\sigma \in \mathrm{Ter}(\Delta; A\sigma)$ by $(a\sigma)\rho = a(\sigma\rho)$. For a type $A \in \mathrm{Ty}(\Gamma)$ we define the cubical set $\Gamma.A$ by $(\Gamma.A)(I)$ being the set of all (ρ, u) with $\rho \in \Gamma(I)$ and $u \in A\rho$; restrictions are given by $(\rho, u)f = (\rho f, uf)$. The first projection yields a map $\mathsf{p} \colon \Gamma.A \to \Gamma$ and the second projection a term $\mathsf{q} \in \mathrm{Ter}(\Gamma.A; A\mathsf{p})$. Given $\sigma \colon \Delta \to \Gamma$, $A \in \mathrm{Ty}(\Gamma)$, and $a \in \mathrm{Ter}(\Delta; A\sigma)$ we define $(\sigma, a) \colon \Delta \to \Gamma.A$ by $(\sigma, a)\rho = (\sigma\rho, a\rho)$. For $u \in \mathrm{Ter}(\Gamma; A)$ we define $[u] = (\mathrm{id}_\Gamma, u) \colon \Gamma \to \Gamma.A$.

The basic type formers are interpreted as follows. For $A \in \mathrm{Ty}(\Gamma)$ and $B \in \mathrm{Ty}(\Gamma.A)$ define $\Sigma_\Gamma(A, B) \in \mathrm{Ty}(\Gamma)$ by letting $\Sigma_\Gamma(A, B)\rho$ contain all pairs (u, v) where $u \in A\rho$ and $v \in B(\rho, v)$; restrictions are defined as $(u, v)f = (uf, vf)$. Given $w \in \mathrm{Ter}(\Gamma; \Sigma(A, B))$ we get $w.1 \in \mathrm{Ter}(\Gamma; A)$ and $w.2 \in \mathrm{Ter}(\Gamma; B[w.1])$ by $(w.1)\rho = \mathsf{p}(w\rho)$ and $(w.2)\rho = \mathsf{q}(w\rho)$ where $\mathsf{p}(u, v) = u$ and $\mathsf{q}(u, v) = v$ are the set-theoretic projections.

Given $A \in \mathrm{Ty}(\Gamma)$ and $B \in \mathrm{Ty}(\Gamma.A)$ the dependent function space $\Pi_\Gamma(A, B) \in \mathrm{Ty}(\Gamma)$ is defined by letting $\Pi_\Gamma(A, B)\rho$ for $\rho \in \Gamma(I)$ contain all families $w = (w_f \mid J \in \mathcal{C}, f \colon J \to I)$ where

$$w_f \in \prod_{u \in A(\rho f)} B(\rho f, u) \quad \text{such that} \quad (w_f\, u)g = w_{fg}(ug) \quad \text{for } u \in A(\rho f),\ g \colon K \to J.$$

The restriction by $f \colon J \to I$ of such a w is defined by $(wf)_g = w_{fg}$. Given $v \in \mathrm{Ter}(\Gamma.A; B)$ we have $\lambda_{\Gamma;A} v \in \mathrm{Ter}(\Gamma; \Pi(A, B))$ given by $((\lambda v)\rho)_f\, u = v(\rho f, u)$. Application $\mathtt{app}(w, u) \in \mathrm{Ter}(\Gamma; B[u])$ of $w \in \mathrm{Ter}(\Gamma; \Pi(A, B))$ to $u \in \mathrm{Ter}(\Gamma; A)$ is defined by

$$\mathtt{app}(w, u)\rho = (w\rho)_{\mathrm{id}_I}(u\rho) \in (B[u])\rho. \tag{2}$$

Basic data types like the natural numbers can be interpreted as discrete presheaves, i.e., $\mathtt{N} \in \mathrm{Ty}(\Gamma)$ is given by $\mathtt{N}\rho = \mathbb{N}$; the constants are interpreted by the lifts of the corresponding set-theoretic operations on $\mathbb{N}$. This concludes the outline of the basic CwF structure on cubical sets.

▶ Remark. Following Aczel [1] we will make use of that our semantic entities are actual sets in the ambient set theory. This will allow us to interpret syntax in Section 8.3 with fewer type annotations than are usually needed for general categorical semantics of type theory (see [24]). E.g., the definition of application $\mathtt{app}(w,u)\rho$ as defined in (2) is independent of Γ, A and B, since set-theoretic application is a (class) operation on all sets. Likewise, we don't need annotations for first and second projections. But note that we will need the type A for λ-abstraction for $(\lambda_{\Gamma;A} v)\rho$ to be a set by the replacement axiom.

Semantic path types

Note that we can consider any cubical set X as $X' \in \mathrm{Ty}(\Gamma)$ by setting $X'\rho = X(I)$ for $\rho \in \Gamma(I)$. We will usually simply write X for X'. In particular, for a cubical set Γ we can form the cubical set $\Gamma.\mathbb{I}$.

For $A \in \mathrm{Ty}(\Gamma)$ and $u, v \in \mathrm{Ter}(\Gamma; A)$ the semantic path type $\mathtt{Path}^{\Gamma}_A(u,v) \in \mathrm{Ty}(\Gamma)$ is given by: for $\rho \in \Gamma(I)$, $\mathtt{Path}_A(u,v)\rho$ consists of equivalence classes $\langle i\rangle\ w$ where $i \notin I$, $w \in A(\rho \mathrm{s}_i)$ such that $w(i0) = u\rho$ and $w(i1) = v\rho$; two such elements $\langle i\rangle\ w$ and $\langle j\rangle\ w'$ are equal iff $w(i/j) = w'$. Here $\mathrm{s}_i \colon I, i \to I$ is induced by the inclusion $I \subseteq I, i$ and (i/j) setting i to j. We define $(\langle i\rangle\ w)f = \langle j\rangle\ w(f, i/j)$ for $f \colon J \to I$ and $j \notin J$. For $r \in \mathbb{I}(I)$ we set $(\langle i\rangle\ w)\,r = w(i/r)$. Both operations, name abstraction and application, lift to terms, i.e., if $w \in \mathrm{Ter}(\Gamma.\mathbb{I}; A)$, then $\langle\rangle w \in \mathrm{Ter}(\Gamma; \mathtt{Path}_A(w[0], w[1]))$ given by $(\langle\rangle w)\rho = \langle i\rangle\ w(\rho \mathrm{s}_i)$ for a fresh i; also if $u \in \mathrm{Ter}(\Gamma; \mathtt{Path}_A(a,b))$ and $r \in \mathrm{Ter}(\Gamma; \mathbb{I})$, then $u\ r \in \mathrm{Ter}(\Gamma; A)$ defined as $(u\ r)\rho = (u\rho)\ (r\rho)$.

Composition Structure

For $\varphi \in \mathrm{Ter}(\Gamma; \mathbb{F})$ we define the cubical set Γ, φ by taking $\rho \in (\Gamma,\varphi)(I)$ iff $\rho \in \Gamma(I)$ and $\varphi\rho = 1_{\mathbb{F}} \in \mathbb{F}$; the restrictions are those induced by Γ. In particular, we have $\Gamma, 1 = \Gamma$ and $\Gamma, 0$ is the empty cubical set. (Here, $0 \in \mathrm{Ter}(\Gamma; \mathbb{F})$ is $0\rho = 0_{\mathbb{F}}$ and similarly for $1_{\mathbb{F}}$.) Any $\sigma \colon \Delta \to \Gamma$ gives rise to a morphism $\Delta, \varphi\sigma \to \Gamma, \varphi$ which we also will denote by σ.

If $A \in \mathrm{Ty}(\Gamma)$ and $\varphi \in \mathrm{Ter}(\Gamma; \mathbb{F})$, we define a *partial element of* $A \in \mathrm{Ty}(\Gamma)$ *of extent* φ to be an element of $\mathrm{Ter}(\Gamma,\varphi; A\iota_\varphi)$ where $\iota_\varphi \colon \Gamma,\varphi \hookrightarrow \Gamma$ is the inclusion. So, such a partial element u is given by a family of elements $u\rho \in A\rho$ for each $\rho \in \Gamma(I)$ such that $\varphi\rho = 1$, satisfying $(u\rho)f = u(\rho f)$ whenever $f \colon J \to I$. Each $u \in \mathrm{Ter}(\Gamma; A)$ gives rise to the partial element $u\iota \in \mathrm{Ter}(\Gamma,\varphi; A\iota)$; a partial element is *extensible* if it is induced by such an element of $\mathrm{Ter}(\Gamma; A)$.

For the next definition note that if $A \in \mathrm{Ty}(\Gamma)$, then $\rho \in \Gamma(I)$ corresponds to $\rho \colon \mathbf{y}\, I \to \Gamma$ and thus $A\rho \in \mathrm{Ty}(\mathbf{y}\, I)$; also, any $\varphi \in \mathbb{F}(I)$ corresponds to $\varphi \in \mathrm{Ter}(\mathbf{y}\, I; \mathbb{F})$.

▶ **Definition 13.** A *composition structure* for $A \in \mathrm{Ty}(\Gamma)$ is given by the following operations. For each I, $i \notin I$, $\rho \in \Gamma(I,i)$, $\varphi \in \mathbb{F}(I)$, u a partial element of $A\rho$ of extent φ, and $a_0 \in A\rho(i0)$ with $a_0 f = u_{(i0)f}$ for all $f \colon J \to I$ with $\varphi f = 1_{\mathbb{F}}$ (i.e., $a_0 \iota_\varphi = u(i0)$ if a_0 is considered as element of $\mathrm{Ter}(\mathbf{y}\, I; A\rho(i0))$), we require

$$\mathtt{comp}(I, i, \rho, \varphi, u, a_0) \in A\rho(i1)$$

such that for any $f \colon J \to I$ and $j \notin J$,

$$(\mathtt{comp}(I, i, \rho, \varphi, u, a_0))f = \mathtt{comp}(J, j, \rho(f, i=j), \varphi f, u(f, i=j), a_0 f),$$

and $\mathtt{comp}(I, i, \rho, 1_{\mathbb{F}}, u, a_0) = u_{(i1)}$.

A type $A \in \mathrm{Ty}(\Gamma)$ together with a composition structure $\mathtt{comp}$ on A is called a *fibrant type*, written $(A, \mathtt{comp}) \in \mathrm{FTy}(\Gamma)$. We will usually simply write $A \in \mathrm{FTy}(\Gamma)$ and $\mathtt{comp}_A$ for its composition structure. But observe that $A \in \mathrm{Ty}(\Gamma)$ can have different composition structures. Call a cubical set Γ *fibrant* if it is a fibrant type when Γ considered as type $\Gamma \in \mathrm{Ty}(\top)$ is fibrant where $\top$ is a terminal cubical set. A prime example of a fibrant cubical set is the singular cubical set of a topological space (see Appendix C).

▶ **Theorem 14.** *The CwF on cubical sets supporting dependent products, dependent sums, and natural numbers described above can be extended to fibrant types.*

Proof. For example, if $A \in \mathrm{FTy}(\Gamma)$ and $\sigma \colon \Delta \to \Gamma$, we set

$$\mathtt{comp}_{A\sigma}(I, i, \rho, \varphi, u, a_0) = \mathtt{comp}_A(I, i, \sigma\rho, \varphi, u, a_0)$$

as the composition structure on $A\sigma$ in $\mathrm{FTy}(\Delta)$. Type formers are treated analogously to their syntactic counterpart given in Section 4. Note that one also has to check that all equations between types are also preserved by their associated composition structures. ◀

Note that we can also, like in the syntax, define a composition structure on $\mathtt{Path}_A(u, v)$ given that A has one.

Semantic Glueing

Next we will give a semantic counterpart to the $\mathtt{Glue}$ construction. To define the semantic glueing as an element of $\mathrm{Ty}(\Gamma)$ it is not necessary that the given types have composition structures or that the functions are equivalences; this is only needed later to give the composition structure. Assume $\varphi \in \mathrm{Ter}(\Gamma; \mathbb{F})$, $T \in \mathrm{Ty}(\Gamma, \varphi)$, $A \in \mathrm{Ty}(\Gamma)$, and $w \in \mathrm{Ter}(\Gamma, \varphi; T \to A\iota)$ (where $A \to B$ is $\Pi(A, B\mathtt{p})$).

▶ **Definition 15.** The *semantic glueing* $\mathtt{Glue}_\Gamma(\varphi, T, A, w) \in \mathrm{Ty}(\Gamma)$ is defined as follows. For $\rho \in \Gamma(I)$, we let $u \in \mathtt{Glue}(\varphi, T, A, w)\rho$ iff either
- $u \in T\rho$ and $\varphi\rho = 1_\mathbb{F}$; or
- $u = \mathtt{glue}(\varphi\rho, t, a)$ and $\varphi\rho \neq 1_\mathbb{F}$, where $t \in \mathrm{Ter}(\mathbf{y}\,I, \varphi\rho; T\rho)$ and $a \in \mathrm{Ter}(\mathbf{y}\,I; A\rho)$ such that $\mathtt{app}(w\rho, t) = a\iota \in \mathrm{Ter}(\mathbf{y}\,I, \varphi\rho; A\rho\iota)$.

For $f \colon J \to I$ we define the restriction uf of $u \in \mathtt{Glue}(\varphi, T, A, w)$ to be given by the restriction of $T\rho$ in the first case; in the second case, i.e., if $\varphi\rho \neq 1_\mathbb{F}$, we let $uf = \mathtt{glue}(\varphi\rho, t, a)f = t_f \in T\rho f$ in case $\varphi\rho f = 1_\mathbb{F}$, and otherwise $uf = \mathtt{glue}(\varphi\rho f, tf, af)$.

Here $\mathtt{glue}$ was defined as a constructor; we extend $\mathtt{glue}$ to any $t \in \mathrm{Ter}(\mathbf{y}\,I; T\rho)$, $a \in \mathrm{Ter}(\mathbf{y}\,I; A\rho)$ such that $\mathtt{app}(w\rho, t) = a$ (so if $\varphi\rho = 1_\mathbb{F}$) by $\mathtt{glue}(1_\mathbb{F}, t, a) = t_{\mathrm{id}_I}$. This way any element of $\mathtt{Glue}(\varphi, T, A, w)\rho$ is of the form $\mathtt{glue}(\varphi\rho, t, a)$ for suitable t and a, and restriction is given by $(\mathtt{glue}(\varphi\rho, t, a))f = \mathtt{glue}(\varphi\rho f, tf, af)$. Note that we get

$$\mathtt{Glue}_\Gamma(1_\mathbb{F}, T, A, w) = T \text{ and } (\mathtt{Glue}_\Gamma(\varphi, T, A, w))\sigma = \mathtt{Glue}_\Delta(\varphi\sigma, T\sigma, A\sigma, w\sigma) \tag{3}$$

for $\sigma \colon \Delta \to \Gamma$. We define $\mathtt{unglue}(\varphi, w) \in \mathrm{Ter}(\Gamma.\,\mathtt{Glue}(\varphi, T, A, w); A\mathtt{p})$ by

$$\begin{aligned} \mathtt{unglue}(\varphi, w)(\rho, t) &= \mathtt{app}(w\rho, t)_{\mathrm{id}_I} \in A\rho && \text{whenever } \varphi\rho = 1_\mathbb{F}, \text{ and} \\ \mathtt{unglue}(\varphi, w)(\rho, \mathtt{glue}(\varphi, t, a)) &= a && \text{otherwise,} \end{aligned}$$

where $\rho \in \Gamma(I)$.

▶ **Definition 16.** For $A, B \in \mathrm{Ty}(\Gamma)$ and $w \in \mathrm{Ter}(\Gamma; A \to B)$ an *equivalence structure* for w is given by the following operations such that for each

- $\rho \in \Gamma(I)$,
- $\varphi \in \mathbb{F}(I)$,
- $b \in B\rho$, and
- partial elements a of $A\rho$ and ω of $\mathtt{Path}_B(\mathtt{app}(w\rho, a), b\iota)\rho$ with extent φ,

we are given

$$\mathsf{e}_0(\rho, \varphi, b, a, \omega) \in A\rho, \text{ and a path } \mathsf{e}_1(\rho, \varphi, b, a, \omega) \text{ between } \mathtt{app}(w\rho, \mathsf{e}_0(\rho, \varphi, b, a, \omega)) \text{ and } b$$

such that $\mathsf{e}_0(\rho, \varphi, b, a, \omega)\iota = a$, $\mathsf{e}_1(\rho, \varphi, b, a, \omega)\iota = \omega$ (where $\iota \colon \mathbf{y}\, I, \varphi \to \mathbf{y}\, I$) and for any $f \colon J \to I$ and $\nu = 0, 1$:

$$(\mathsf{e}_\nu(\rho, \varphi, b, a, \omega))f = \mathsf{e}_\nu(\rho f, \varphi f, bf, af, \omega f).$$

Following the argument in the syntax we can use the equivalence structure to explain a composition for `Glue`.

▶ **Theorem 17.** *If $A \in \mathrm{FTy}(\Gamma)$, $T \in \mathrm{FTy}(\Gamma, \varphi)$, and we have an equivalence structure for w, then we have a composition structure for $\mathtt{Glue}(\varphi, T, A, w)$ such that the equations* (3) *also hold for the respective composition structures.*

Semantic Universes

Assuming a Grothendieck universe of small sets in our ambient set theory, we can define $A \in \mathrm{Ty}_0(\Gamma)$ iff all $A\rho$ are small for $\rho \in \Gamma(I)$; and $A \in \mathrm{FTy}_0(\Gamma)$ iff $A \in \mathrm{Ty}_0(\Gamma)$ when forgetting the composition structure of A.

▶ **Definition 18.** The semantic universe $\mathtt{U}$ is the cubical set defined by $\mathtt{U}(I) = \mathrm{FTy}_0(\mathbf{y}\, I)$; restriction along $f \colon J \to I$ is simply substitution along $\mathbf{y}\, f$.

We can consider $\mathtt{U}$ as an element of $\mathrm{Ty}(\Gamma)$. As such we can, as in the syntactic counterpart, define a composition structure on $\mathtt{U}$ using semantic glueing, so that $\mathtt{U} \in \mathrm{FTy}(\Gamma)$. Note that semantic glueing preserves smallness.

For $T \in \mathrm{Ter}(\Gamma; \mathtt{U})$ we can define decoding $\mathsf{El}\, T \in \mathrm{FTy}_0(\Gamma)$ by $(\mathsf{El}\, T)\rho = (T\rho)\, \mathrm{id}_I$ and likewise for the composition structure. For $A \in \mathrm{FTy}_0(\Gamma)$ we get its code $\ulcorner A \urcorner \in \mathrm{Ter}(\Gamma; \mathtt{U})$ by setting $\ulcorner A \urcorner \rho \in \mathrm{FTy}_0(\mathbf{y}\, I)$ to be given by the sets $(\ulcorner A \urcorner \rho) f = A(\rho f)$ and likewise for restrictions and composition structure. These operations satisfy $\mathsf{El} \ulcorner A \urcorner = A$ and $\ulcorner \mathsf{El}\, T \urcorner = T$.

8.3 Interpretation of the syntax

Following [24] we define a partial interpretation function from raw syntax to the CwF with fibrant types given in the previous section.

To interpret the universe rules à la Russell we assume two Grothendieck universes in the underlying set theory, say *tiny* and *small* sets. So that any tiny set is small, and the set of tiny sets is small. For a cubical set X we define $\mathrm{FTy}_0(X)$ and $\mathrm{FTy}_1(X)$ as in the previous section, now referring to tiny and small sets, respectively. We get semantic universes $\mathtt{U}_i(I) = \mathrm{FTy}_i(\mathbf{y}\, I)$ for $i = 0, 1$; we identify those with their lifts to types. As noted above, these lifts carry a composition structure, and thus are fibrant. We also have $\mathtt{U}_0 \subseteq \mathtt{U}_1$ and thus $\mathrm{Ter}(X; \mathtt{U}_0) \subseteq \mathrm{Ter}(X; \mathtt{U}_1)$. Note that coding and decoding are, as set-theoretic operations, the same for both universes. We get that $\ulcorner \mathtt{U}_0 \urcorner \in \mathrm{Ter}(X; \mathtt{U}_1)$ which will serve as the interpretation of $\mathtt{U}$.

In what follows, we define a partial interpretation function of raw syntax: $[\![\Gamma]\!]$, $[\![\Gamma; t]\!]$, and $[\![\Delta; \sigma]\!]$ by recursion on the raw syntax. Since we want to interpret a universe à la Russell we

cannot assume terms and types to have different syntactic categories. The definition is given below and should be read such that the interpretation is defined whenever all interpretations on the right-hand sides are defined *and* make sense; so, e.g., for $[\![\Gamma]\!].\,\mathsf{El}\,[\![\Gamma; A]\!]$ below, we require that $[\![\Gamma]\!]$ is defined and a cubical set, $[\![\Gamma; A]\!]$ is defined, and $\mathsf{El}\,[\![\Gamma; A]\!] \in \mathrm{FTy}([\![\Gamma]\!])$. The interpretation for raw contexts is given by:

$$[\![()]\!] = \top \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!].\mathsf{El}\,[\![\Gamma; A]\!] \qquad \text{if } x \notin \mathrm{dom}(\Gamma)$$
$$[\![\Gamma, \varphi]\!] = [\![\Gamma]\!], [\![\Gamma; \varphi]\!] \qquad [\![\Gamma, i : \mathbb{I}]\!] = [\![\Gamma]\!].\mathbb{I} \qquad \text{if } i \notin \mathrm{dom}(\Gamma)$$

where $\top$ is a terminal cubical set and in the last equation $\mathbb{I}$ is considered as an element of $\mathrm{Ty}([\![\Gamma]\!])$. When defining $[\![\Gamma; t]\!]$ we require that $[\![\Gamma]\!]$ is defined and a cubical set; then $[\![\Gamma; t]\!]$ is a (partial) family of sets $[\![\Gamma; t]\!](I, \rho)$ for $I \in \mathcal{C}$ and $\rho \in [\![\Gamma]\!](I)$ (leaving I implicit in the definition). We define:

$$\begin{aligned}
[\![\Gamma; \mathsf{U}]\!] &= \ulcorner \mathtt{U}_0 \urcorner \in \mathrm{Ter}([\![\Gamma]\!]; \mathtt{U}_1) \\
[\![\Gamma; \mathsf{N}]\!] &= \ulcorner \mathtt{N} \urcorner \in \mathrm{Ter}([\![\Gamma]\!]; \mathtt{U}_0) \\
[\![\Gamma; (x : A) \to B]\!] &= \ulcorner \Pi_{[\![\Gamma]\!]}(\mathsf{El}\,[\![\Gamma; A]\!], \mathsf{El}\,[\![\Gamma, x : A; B]\!]) \urcorner \\
[\![\Gamma; (x : A) \times B]\!] &= \ulcorner \Sigma_{[\![\Gamma]\!]}(\mathsf{El}\,[\![\Gamma; A]\!], \mathsf{El}\,[\![\Gamma, x : A; B]\!]) \urcorner \\
[\![\Gamma; \mathsf{Path}\ A\ a\ b]\!] &= \ulcorner \mathtt{Path}^{[\![\Gamma]\!]}_{\mathsf{El}\,[\![\Gamma; A]\!]}([\![\Gamma; a]\!], [\![\Gamma; b]\!]) \urcorner \\
[\![\Gamma; \mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A]\!] &= \ulcorner \mathtt{Glue}_{[\![\Gamma]\!]}([\![\Gamma; \varphi]\!], \mathsf{El}\,[\![\Gamma, \varphi; T]\!], \mathsf{El}\,[\![\Gamma; A]\!], [\![\Gamma, \varphi; f]\!]) \urcorner \\
[\![\Gamma; \lambda x : A.t]\!] &= \lambda_{[\![\Gamma]\!]; \mathsf{El}\,[\![\Gamma; A]\!]}([\![\Gamma, x : A; t]\!]) \\
[\![\Gamma; t\ u]\!] &= \mathsf{app}([\![\Gamma; t]\!], [\![\Gamma; u]\!]) \\
[\![\Gamma; \langle i \rangle\ t]\!] &= \langle\,\rangle_{[\![\Gamma]\!]} [\![\Gamma, i\colon \mathbb{I}; t]\!] \\
[\![\Gamma; t\ r]\!] &= [\![\Gamma; t]\!][\![\Gamma; r]\!]
\end{aligned}$$

where for path application, juxtaposition on the right-hand side is semantic path application. In the case of a bound variable, we assume that x (respectively i) is a *chosen* variable fresh for Γ; if this is not possible the expression is undefined. Moreover, all type formers should be read as those on fibrant types, i.e., also defining the composition structure. In the case of `Glue`, it is understood that the function part, i.e., the fourth argument of `Glue` in Definition 15 is $\mathsf{p} \circ [\![\Gamma, \varphi; f]\!]$ and the required (by Theorem 17) equivalence structure is to be extracted from $\mathsf{q} \circ [\![\Gamma, \varphi; f]\!]$ as in Section 5.3. In virtue of the remark in Section 8.2 we don't need type annotations to interpret applications. Note that coding and decoding tacitly refer to $[\![\Gamma]\!]$ as well. For the rest of the raw terms we also assume we are given $\rho \in [\![\Gamma]\!](I)$. Variables are interpreted by:

$$[\![\Gamma, x : A; x]\!]\rho = \mathsf{q}(\rho) \qquad [\![\Gamma, x : A; y]\!]\rho = [\![\Gamma; y]\!](\mathsf{p}(\rho)) \qquad [\![\Gamma, \varphi; y]\!]\rho = [\![\Gamma; y]\!]\rho$$

These should also be read to include the case when x or y are name variables; if x is a name variable, we require A to be $\mathbb{I}$. The interpretations of $[\![\Gamma; r]\!]\rho$ where r is not a name and $[\![\Gamma; \varphi]\!]\rho$ follow inductively as elements of $\mathbb{I}$ and $\mathbb{F}$, respectively.

Constants for dependent sums are interpreted by:

$$[\![\Gamma; (t, u)]\!]\rho = ([\![\Gamma; t]\!]\rho, [\![\Gamma; u]\!]\rho) \qquad [\![\Gamma; t.1]\!]\rho = \mathsf{p}([\![\Gamma; t]\!]\rho) \qquad [\![\Gamma; t.2]\!]\rho = \mathsf{q}([\![\Gamma; t]\!]\rho)$$

Likewise, constants for N will be interpreted by their semantic analogues (omitted). The interpretations for the constants related to glueing are

$$\begin{aligned}
[\![\Gamma; \mathsf{glue}\,[\varphi \mapsto t]\,u]\!]\rho &= \mathtt{glue}([\![\Gamma; \varphi]\!]\rho, [\![\Gamma, \varphi; t]\!]\hat{\rho}, [\![\Gamma; u]\!]\rho) \\
[\![\Gamma; \mathsf{unglue}\,[\varphi \mapsto f]\,u]\!]\rho &= \mathtt{unglue}([\![\Gamma; \varphi]\!], \mathsf{p} \circ [\![\Gamma; f]\!])(\rho, [\![\Gamma; u]\!]\rho)
\end{aligned}$$

where $[\![\Gamma, \varphi; t]\!]\hat{\rho}$ is the family assigning $[\![\Gamma, \varphi; t]\!](\rho f)$ to $J \in \mathcal{C}$ and $f \colon J \to I$ (and ρf refers to the restriction given by $[\![\Gamma]\!]$ which is assumed to be a cubical set). Partial elements are interpreted by

$$[\![\Gamma; [\ \varphi_1\ u_1, \dots, \varphi_n\ u_n\ ]]\!]\rho = [\![\Gamma, \varphi_i; u_i]\!]\rho \qquad \text{if } [\![\Gamma; \varphi_i]\!]\rho = 1_{\mathbb{F}},$$

where for this to be defined we additionally assume that all $[\![\Gamma, \varphi_i; u_i]\!]$ are defined and $[\![\Gamma, \varphi_i; u_i]\!]\rho' = [\![\Gamma, \varphi_j; u_j]\!]\rho'$ for each $\rho' \in [\![\Gamma]\!](I)$ with $[\![\Gamma; \varphi_i \wedge \varphi_j]\!]\rho' = 1_{\mathbb{F}}$.

Finally, the interpretation of composition is given by

$$[\![\Gamma; \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0]\!]\rho = \mathsf{comp}_{\mathsf{El}\,[\![\Gamma, i:\mathbb{I}; A]\!]}(I, j, \rho', [\![\Gamma; \varphi]\!]\rho, [\![\Gamma, \varphi, i\colon \mathbb{I}; u]\!]\rho', [\![\Gamma; a_0]\!]\rho)$$

if $i \notin \mathrm{dom}(\Gamma)$, and where j is fresh and $\rho' = (\rho \mathrm{s}_j, i = j)$ with $\mathrm{s}_j \colon I, j \to I$ induced from the inclusion $I \subseteq I, j$.

The interpretation of raw substitutions $[\![\Delta; \sigma]\!]$ is a (partial) family of sets $[\![\Delta; \sigma]\!](I, \rho)$ for $I \in \mathcal{C}$ and $\rho \in [\![\Delta]\!](I)$. We set

$$[\![\Delta; ()]\!]\rho = *, \qquad\qquad [\![\Delta; (\sigma, x/t)]\!]\rho = ([\![\Delta; \sigma]\!]\rho, [\![\Delta; t]\!]\rho) \quad \text{if } x \notin \mathrm{dom}(\sigma),$$

where $*$ is the unique element of $\top(I)$. This concludes the definition of the interpretation of syntax.

In the following α stands for either a raw term or raw substitution. In the latter case, $\alpha\sigma$ denotes composition of substitutions.

▶ **Lemma 19.** *Let Γ' be like Γ but with some φ's inserted, and assume both $[\![\Gamma]\!]$ and $[\![\Gamma']\!]$ are defined; then:*

1. *$[\![\Gamma']\!]$ is a sub-cubical set of $[\![\Gamma]\!]$;*
2. *if $[\![\Gamma; \alpha]\!]$ is defined, then so is $[\![\Gamma'; \alpha]\!]$ and they agree on $[\![\Gamma']\!]$.*

▶ **Lemma 20** (Weakening)**.** *Let $[\![\Gamma]\!]$ be defined.*

1. *If $[\![\Gamma, x : A, \Delta]\!]$ is defined, then so is $[\![\Gamma, x : A, \Delta; x]\!]$ which is moreover the projection to the x-part.*[5]
2. *If $[\![\Gamma, \Delta]\!]$ is defined, then so is $[\![\Gamma, \Delta; \mathrm{id}_\Gamma]\!]$ which is moreover the projection to the Γ-part.*
3. *If $[\![\Gamma, \Delta]\!]$, $[\![\Gamma; \alpha]\!]$ are defined and the variables in Δ are fresh for α, then $[\![\Gamma, \Delta; \alpha]\!]$ is defined and for $\rho \in [\![\Gamma, \Delta]\!](I)$:*

$$[\![\Gamma; \alpha]\!]([\![\Gamma, \Delta; \mathrm{id}_\Gamma]\!]\rho) = [\![\Gamma, \Delta; \alpha]\!]\rho$$

▶ **Lemma 21** (Substitution)**.** *For $[\![\Gamma]\!]$,$[\![\Delta]\!]$, $[\![\Delta; \sigma]\!]$, and $[\![\Gamma; \alpha]\!]$ defined with $\mathrm{dom}(\Gamma) = \mathrm{dom}(\sigma)$ (as lists), also $[\![\Delta; \alpha\sigma]\!]$ is defined and for $\rho \in [\![\Delta]\!](I)$:*

$$[\![\Gamma; \alpha]\!]([\![\Delta; \sigma]\!]\rho) = [\![\Delta; \alpha\sigma]\!]\rho$$

▶ **Lemma 22.** *If $[\![\Gamma]\!]$ is defined and a cubical set, and $[\![\Gamma; \alpha]\!]$ is defined, then $([\![\Gamma; \alpha]\!]\rho)f = [\![\Gamma; \alpha]\!](\rho f)$.*

To state the next theorem let us set $[\![\Gamma; \mathbb{I}]\!] = \ulcorner \mathbb{I} \urcorner$ and $[\![\Gamma; \mathbb{F}]\!] = \ulcorner \mathbb{F} \urcorner$ as elements of $\mathrm{Ty}_0([\![\Gamma]\!])$.

▶ **Theorem 23** (Soundness)**.** *We have the following implications, and all occurrences of $[\![-]\!]$ in the conclusions are defined. In (3) and (5) we allow A to be $\mathbb{I}$ or $\mathbb{F}$.*

[5] E.g., if Γ is $y : B, z : C$, the projection to the x-part maps $(b, (c, (a, \delta)))$ to a, and the projection to the Γ-part maps $(b, (c, \delta))$ to (b, c).

1. *if* $\Gamma \vdash$ *, then* $[\![\Gamma]\!]$ *is a cubical set;*
2. *if* $\Gamma \vdash A$*, then* $[\![\Gamma; A]\!] \in \mathrm{Ter}([\![\Gamma]\!]; \mathsf{U}_1)$*;*
3. *if* $\Gamma \vdash t : A$*, then* $[\![\Gamma; t]\!] \in \mathrm{Ter}([\![\Gamma]\!]; \mathsf{El}\,[\![\Gamma; A]\!])$*;*
4. *if* $\Gamma \vdash A = B$*, then* $[\![\Gamma; A]\!] = [\![\Gamma; B]\!]$*;*
5. *if* $\Gamma \vdash a = b : A$*, then* $[\![\Gamma; a]\!] = [\![\Gamma; b]\!]$*;*
6. *if* $\Gamma \vdash \sigma : \Delta$*, then* $[\![\Gamma; \sigma]\!]$ *restricts to a natural transformation* $[\![\Gamma]\!] \to [\![\Delta]\!]$*.*

9 Extensions: Identity Types and Higher Inductive Types

In this section we consider possible extensions to cubical type theory. The first is an identity type defined using path types whose elimination principle holds as a judgmental equality. The second are two examples of higher inductive types.

9.1 Identity Types

We can use the path type to represent equalities. Using the composition operation, we can indeed build a substitution function $P(a) \to P(b)$ from any path between a and b. However, since we don't have in general the judgmental equality $\mathsf{transp}^i\ A\ a_0 = a_0$ if A is independent of i (which is an equality that we cannot expect geometrically in general, as shown in Appendix C), this substitution function does not need to be the constant function when the path is constant. This means that, as in the previous model [6, 13], we don't get an interpretation of Martin-Löf identity type [19] with the standard judgmental equalities.

However, we can define another type which *does* give an interpretation of this identity type following an idea of Andrew Swan.

Identity Types

The basic idea of $\mathsf{Id}\ A\ a_0\ a_1$ is to define it in terms of $\mathsf{Path}\ A\ a_0\ a_1$ but also mark the paths where they are known to be constant. Formally, the formation and introduction rules are

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash a_0 : A \qquad \Gamma \vdash a_1 : A}{\Gamma \vdash \mathsf{Id}\ A\ a_0\ a_1} \qquad \frac{\Gamma \vdash \omega : \mathsf{Path}\ A\ a_0\ a_1[\varphi \mapsto \langle i \rangle\ a_0]}{\Gamma \vdash (\omega, \varphi) : \mathsf{Id}\ A\ a_0\ a_1}$$

and we can define $\mathsf{r}\,a = (1_a, 1_{\mathbb{F}}) : \mathsf{Id}\ A\ a\ a$ for $a : A$. The elimination rule, given $\Gamma \vdash a : A$, is

$$\frac{\Gamma, x : A, \alpha : \mathsf{Id}\ A\ a\ x \vdash C \qquad \Gamma \vdash d : C(x/a, \alpha/\mathsf{r}\,a) \qquad \Gamma \vdash b : A \qquad \Gamma \vdash \beta : \mathsf{Id}\ A\ a\ b}{\Gamma \vdash \mathsf{J}_{x,\alpha.C}\ d\ b\ \beta : C(x/b, \alpha/\beta)}$$

together with the following judgmental equality in case β is of the form (ω, φ)

$$\mathsf{J}\ d\ b\ \beta = \mathsf{comp}^i\ C(x/\omega\ i, \alpha/\beta^*(i))\ [\varphi \mapsto d]\ d$$

where $\Gamma, i : \mathbb{I} \vdash \beta^*(i) : \mathsf{Id}\ A\ a\ (\omega\ i)$ is given by

$$\beta^*(i) = (\langle j \rangle\ \omega\ (i \wedge j), \varphi \vee (i = 0)).$$

Note that with this definition we get $\mathsf{J}\ d\ a\ (\mathsf{r}\,a) = d$ as desired.

The composition operation for Id is explained as follows. Given $\Gamma, i : \mathbb{I} \vdash \mathsf{Id}\ A\ a_0\ a_1$, $\Gamma, \varphi, i : \mathbb{I} \vdash (\omega, \psi) : \mathsf{Id}\ A\ a_0\ a_1$, and $\Gamma \vdash (\omega_0, \psi_0) : (\mathsf{Id}\ A\ a_0\ a_1)(i0)[\varphi \mapsto (\omega(i0), \psi(i0))]$ we have the judgmental equality

$$\mathsf{comp}^i\ (\mathsf{Id}\ A\ a_0\ a_1)\ [\varphi \mapsto (\omega, \psi)]\ (\omega_0, \psi_0) = (\mathsf{comp}^i\ (\mathsf{Path}\ A\ a_0\ a_1)\ [\varphi \mapsto \omega]\ \omega_0, \varphi \wedge \psi(i1)).$$

It can then be shown that the types $\mathsf{Id}\ A\ a\ b$ and $\mathsf{Path}\ A\ a\ b$ are (Path)-equivalent. In particular, a type is (Path)-contractible if, and only if, it is (Id)-contractible. The univalence axiom, proved in Section 7.2 for the Path-type, hence holds as well for the Id-type.[6]

Cofibration-Trivial Fibration Factorization

The same idea can be used to factorize an arbitrary map of (not necessary fibrant) cubical sets $f : A \to B$ into a cofibration followed by a trivial fibration. We define a "trivial fibration" to be a first projection from a total space of a contractible family of types and a "cofibration" to be a map that has the left lifting property against any trivial fibration. For this we define, for $b : B$, the type $T_f(b)$ to be the type of elements $[\varphi \mapsto a]$ with $\varphi \vdash a : A$ and $\varphi \vdash f\ a = b : B$.

▶ **Theorem 24.** *The type $T_f(b)$ is contractible and the map*

$$A \to (b : B) \times T_f(b), \qquad a \longmapsto (f\ a, [1_{\mathbb{F}} \mapsto a])$$

is a cofibration.

The definition of the identity type can be seen as a special case of this, if we take the B the type of paths in A and for f the constant path function.

9.2 Higher Inductive Types

In this section we consider the extension of cubical type theory with two different higher inductive types: spheres and propositional truncation. The presentation in this section is syntactical, but it can be directly translated into semantic definitions.

Extension to Dependent Path Types

In order to formulate the elimination rules for higher inductive types, we need to extend the path type to *dependent path type*, which is described by the following rules. If $i : \mathbb{I} \vdash A$ and $\vdash a_0 : A(i0),\ a_1 : A(i1)$, then $\vdash \mathsf{Path}^i\ A\ a_0\ a_1$. The introduction rule is that $\vdash \langle i \rangle\ t : \mathsf{Path}^i\ A\ t(i0)\ t(i1)$ if $i : \mathbb{I} \vdash t : A$. The elimination rule is $\vdash p\ r : A(i/r)$ if $\vdash p : \mathsf{Path}^i\ A\ a_0\ a_1$ with equalities $p\ 0 = a_0 : A(i0)$ and $p\ 1 = a_1 : A(i1)$.

Spheres

We define the circle, S^1, by the rules:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{S}^1} \quad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{base} : \mathsf{S}^1} \quad \frac{\Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \mathsf{loop}(r) : \mathsf{S}^1}$$

with the equalities $\mathsf{loop}(0) = \mathsf{loop}(1) = \mathsf{base}$.

Since we want to represent the *free* type with one base point and a loop, we add composition as a *constructor* operation hcomp^i:

$$\frac{\Gamma, \varphi, i : \mathbb{I} \vdash u : \mathsf{S}^1 \qquad \Gamma \vdash u_0 : \mathsf{S}^1[\varphi \mapsto u(i0)]}{\Gamma \vdash \mathsf{hcomp}^i\ [\varphi \mapsto u]\ u_0 : \mathsf{S}^1}$$

with the equality $\mathsf{hcomp}^i\ [1_{\mathbb{F}} \mapsto u]\ u_0 = u(i1)$.

[6] This has been formally verified using the Haskell implementation:
`https://github.com/mortberg/cubicaltt/blob/v1.0/examples/idtypes.ctt`

Given a dependent type $x : \mathsf{S}^1 \vdash A$ and $a : A(x/\mathsf{base})$ and $l : \mathsf{Path}^i\ A(x/\mathsf{loop}(i))\ a\ a$ we can define a function $g : (x : \mathsf{S}^1) \to A$ by the equations[7] $g\ \mathsf{base} = a$ and $g\ \mathsf{loop}(r) = l\ r$ and

$$g\ (\mathsf{hcomp}^i\ [\varphi \mapsto u]\ u_0) = \mathsf{comp}^i\ A(x/v)\ [\varphi \mapsto g\ u]\ (g\ u_0)$$

where $v = \mathsf{fill}^i\ \mathsf{S}^1\ [\varphi \mapsto u]\ u_0 = \mathsf{hcomp}^j\ [\varphi \mapsto u(i/i \wedge j), (i = 0) \mapsto u_0]\ u_0$.

This definition is non ambiguous since $l\ 0 = l\ 1 = a$.

We have a similar definition for S^n taking as constructors base and $\mathsf{loop}(r_1, \ldots, r_n)$.

Propositional Truncation

We define the propositional truncation, $\mathsf{inh}\ A$, of a type A by the rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{inh}\ A} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inc}\ a : \mathsf{inh}\ A} \qquad \frac{\Gamma \vdash u_0 : \mathsf{inh}\ A \qquad \Gamma \vdash u_1 : \mathsf{inh}\ A \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \mathsf{squash}(u_0, u_1, r) : \mathsf{inh}\ A}$$

with the equalities $\mathsf{squash}(u_0, u_1, 0) = u_0$ and $\mathsf{squash}(u_0, u_1, 1) = u_1$.

As before, we add composition as a constructor, but only in the form[8]

$$\frac{\Gamma, \varphi, i : \mathbb{I} \vdash u : \mathsf{inh}\ A \qquad \Gamma \vdash u_0 : \mathsf{inh}\ A[\varphi \mapsto u(i0)]}{\Gamma \vdash \mathsf{hcomp}^i\ [\varphi \mapsto u]\ u_0 : \mathsf{inh}\ A}$$

with the equality $\mathsf{hcomp}^i\ [1_{\mathbb{F}} \mapsto u]\ u_0 = u(i1)$.

This provides only a definition of $\mathsf{comp}^i\ (\mathsf{inh}\ A)\ [\varphi \mapsto u]\ u_0$ in the case where A is independent of i, and we have to explain how to define the general case.

In order to do this, we define first two operations

$$\frac{\Gamma, i : \mathbb{I} \vdash A \qquad \Gamma \vdash u_0 : \mathsf{inh}\ A(i0)}{\Gamma \vdash \mathsf{transp}\ u_0 : \mathsf{inh}\ A(i1)} \qquad \frac{\Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, i : \mathbb{I} \vdash u : \mathsf{inh}\ A}{\Gamma \vdash \mathsf{squeeze}^i\ u : \mathsf{Path}\ (\mathsf{inh}\ A(i1))\ (\mathsf{transp}\ u(i0))\ u(i1)}$$

by the equations

$$\begin{array}{lcl}
\mathsf{transp}\ (\mathsf{inc}\ a) & = & \mathsf{inc}\ (\mathsf{comp}^i\ A\ []\ a) \\
\mathsf{transp}\ (\mathsf{squash}(u_0, u_1, r)) & = & \mathsf{squash}(\mathsf{transp}\ u_0, \mathsf{transp}\ u_1, r) \\
\mathsf{transp}\ (\mathsf{hcomp}^j\ [\varphi \mapsto u]\ u_0) & = & \mathsf{hcomp}^j\ [\varphi \mapsto \mathsf{transp}\ u]\ (\mathsf{transp}\ u_0) \\
 & & \\
\mathsf{squeeze}^i\ (\mathsf{inc}\ a) & = & \langle i \rangle\ \mathsf{inc}\ (\mathsf{comp}^j\ A(i \vee j)\ [(i = 1) \mapsto a(i1)]\ a) \\
\mathsf{squeeze}^i\ (\mathsf{squash}(u_0, u_1, r)) & = & \langle k \rangle\ \mathsf{squash}(\mathsf{squeeze}^i\ u_0\ k, \mathsf{squeeze}^i\ u_1\ k, r(i/k)) \\
\mathsf{squeeze}^i\ (\mathsf{hcomp}^j\ [\varphi \mapsto u]\ v) & = & \langle k \rangle\ \mathsf{hcomp}^j\ S\ (\mathsf{squeeze}^i\ v\ k)
\end{array}$$

where S is the system

$$[\delta \mapsto \mathsf{squeeze}^i\ u\ k,\ \ \varphi(i/k) \wedge (k = 0) \mapsto \mathsf{transp}\ u(i0),\ \ \varphi(i/k) \wedge (k = 1) \mapsto u(i1)]$$

and $\delta = \forall i.\varphi$, using Lemma 2.

Using these operations, we can define the general composition

$$\frac{\Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : \mathsf{inh}\ A \qquad \Gamma \vdash u_0 : \mathsf{inh}\ A(i0)[\varphi \mapsto u(i0)]}{\Gamma \vdash \mathsf{comp}^i\ (\mathsf{inh}\ A)\ [\varphi \mapsto u]\ u_0 : \mathsf{inh}\ A(i1)[\varphi \mapsto u(i1)]}$$

---

[7] For the equation $g\ \mathsf{loop}(r) = l\ r$, it may be that l and r are dependent on the same name i, and we could not have followed this definition in the framework of [6].

[8] This restriction on the constructor is essential for the justification of the elimination rule below.

by $\Gamma \vdash \mathsf{comp}^i\ (\mathsf{inh}\ A)\ [\varphi \mapsto u]\ u_0 = \mathsf{hcomp}^j\ [\varphi \mapsto \mathsf{squeeze}^i\ u\ j]\ (\mathsf{transp}\ u_0) : \mathsf{inh}\ A(i1)$.

Given $\Gamma \vdash B$ and $\Gamma \vdash q : (x\ y : B) \to \mathsf{Path}\ B\ x\ y$ and $f : A \to B$ we define $g : \mathsf{inh}\ A \to B$ by the equations

$$
\begin{array}{lcl}
g\ (\mathsf{inc}\ a) & = & f\ a \\
g\ (\mathsf{squash}(u_0, u_1, r)) & = & q\ (g\ u_0)\ (g\ u_1)\ r \\
g\ (\mathsf{hcomp}^j\ [\varphi \mapsto u]\ u_0) & = & \mathsf{comp}^j\ B\ [\varphi \mapsto g\ u]\ (g\ u_0)
\end{array}
$$

10 Related and Future Work

Cubical ideas have proved useful to reason about equality in homotopy type theory [18]. In cubical type theory these techniques could be simplified as there are new judgmental equalities and better notations for manipulating higher dimensional cubes. Indeed some simple experiments using the Haskell implementation have shown that we can simplify some constructions in synthetic homotopy theory.[9]

Other approaches to extending intensional type theory with extensionality principles can be found in [2, 23]. These approaches have close connections to techniques for internalizing parametricity in type theory [5]. Further, nominal extensions to λ-calculus and semantical ideas related to the ones presented in this paper have recently also proved useful for justifying type theory with internalized parametricity [4].

The paper [11] provides a general framework for analyzing the uniformity condition, which applies to simplicial and cubical sets.

Large parts of the semantics presented in this paper have been formally verified in NuPrl by Mark Bickford[10], in particular, the definition of Kan filling in terms of composition as in Section 4.4 and composition for glueing as given in Section 6.2.

Following the usual reducibility method, we expect it to be possible to adapt our presheaf semantics to a proof of normalization and decidability of type checking. A first step in this direction is the proof of canonicity in [14]. We end the paper with a list of open problems and conjectures:

1. Extend the semantics of identity types to the semantics of inductive families.
2. Give a general syntax and semantics of higher inductive types.
3. Extend the system with resizing rules and show normalization.
4. Is there a model where Path and Id coincide?

Acknowledgements. This work originates from discussions between the authors around an implementation of a type system corresponding to the model described in [6]. This implementation indicated a problem with the representation of higher inductive types, e.g., the elimination rule for the circle, and suggested the need of extending this cubical model with a diagonal operation. The general framework (uniformity condition, connections, semantics of spheres and propositional truncation) is due to the second author. In particular, the glueing operation with its composition was introduced as a generalization of the operation described in [6] transforming an equivalence into a path, and with the condition $A = \mathsf{Glue}\ []\ A$. In a first attempt, we tried to force "regularity", i.e., the equation $\mathsf{transp}\ i\ A\ a_0 = a_0$ if A is independent of i (which seemed to be necessary in order to get filling from compositions, and which implies $\mathsf{Path} = \mathsf{Id}$). There was a problem however for getting regularity for the

[9] For details see: `https://github.com/mortberg/cubicaltt/tree/master/examples/`
[10] For details see: `http://www.nuprl.org/wip/Mathematics/cubical!type!theory/`

universe, that was discovered by Dan Licata (from discussions with Carlo Angiuli and Bob Harper). Thanks to this discovery, it was realized that regularity is actually not needed for the model to work. In particular, the second author adapted the definition of filling from composition as in Section 4.4, the third author noticed that we can remove the condition $A = \mathsf{Glue}\ []\ A$, and together with the last author, they derived the univalence axiom from the glueing operation as presented in the appendix. This was surprising since glueing was introduced a priori only as a way to transform equivalences into paths, but was later explained by a remark of Dan Licata (also presented in the appendix: we get univalence as soon as the transport map associated to this path is path equal to the given equivalence). The second author introduced then the restriction operation Γ, φ on contexts, which, as noticed by Christian Sattler, can be seen as an explicit syntax for the notion of cofibration, and designed the other proof of univalence in Section 7.2 from discussions between Nicola Gambino, Peter LeFanu Lumsdaine and the third author. Not having regularity, the type of paths is not the same as the Id type but, as explained in Section 9.1, we can recover the usual identity type from the path type, following an idea of Andrew Swan.

The authors would like to thank the referees and Martín Escardó, Georges Gonthier, Dan Grayson, Peter Hancock, Dan Licata, Peter LeFanu Lumsdaine, Christian Sattler, Andrew Swan, Vladimir Voevodsky for many interesting discussions and remarks.

References

1. Peter Aczel. On relating type theories and set theories. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *Types for Proofs and Programs, International Workshop TYPES '98, Kloster Irsee, Germany, March 27-31, 1998, Selected Papers*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1998. doi:10.1007/3-540-48167-2_1.
2. T. Altenkirch. Extensional equality in intensional type theory. In *Proc. of 14th Ann. IEEE Symp. on Logic in Computer Science, LICS '99*, pages 412–420. IEEE, 1999. doi:10.1109/lics.1999.782636.
3. R. Balbes and P. Dwinger. *Distributive Lattices*. University of Missouri Press, 1974. Reprinted by Abstract Space Publishing in 2011.
4. Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electr. Notes Theor. Comput. Sci.*, 319:67–82, 2015. doi:10.1016/j.entcs.2015.12.006.
5. Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 61–72. ACM, 2013. doi:10.1145/2500365.2500577.
6. M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In R. Matthes and A. Schubert, editors, *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES 2013*, volume 26 of *Leibniz Int. Proc. in Inform.*, pages 107–128. Dagstuhl Publishing, 2014. doi:10.4230/lipics.types.2013.107.
7. R. Brown, P. J. Higgins, and R. Sivera. *Nonabelian Algebraic Topology: Filtered Spaces, Crossed Complexes, Cubical Homotopy Groupoids*, volume 15 of *EMS Tracts in Mathematics*. Europ. Math. Soc., 2011. doi:10.4171/083.
8. D.-C. Cisinski. Univalent universes for elegant models of homotopy types, 2014. arXiv preprint 1406.0058. URL: https://arxiv.org/abs/1406.0058.
9. Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995. doi:10.1007/3-540-61780-9_66.

10 M. Fourman and D. Scott. Sheaves and logic. In M. Fourman, C. Mulvey, and D. Scott, editors, *Applications of Sheaves*, volume 753 of *Lect. Notes in Math.*, pages 302–401. Springer, 1979. `doi:10.1007/bfb0061824`.

11 N. Gambino and C. Sattler. Uniform fibrations and the Frobenius condition, 2015. arXiv preprint 1510.00669. URL: `https://arxiv.org/abs/1510.00669`.

12 M. Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14 of *Publications of the Newton Institute*, pages 79–130. Cambridge University Press, 1997. `doi:10.1017/cbo9780511526619.004`.

13 S. Huber. *A Model of Type Theory in Cubical Sets*. Licentiate thesis, University of Gothenburg, 2015. URL: `http://www.cse.chalmers.se/~simonhu/misc/lic.pdf`.

14 S. Huber. Canonicity for cubical type theory, 2016. arXiv preprint 1607.04156. URL: `https://arxiv.org/abs/1607.04156`.

15 J. A. Kalman. Lattices with involution. *Trans. Amer. Math. Soc.*, 87:485–491, 1958. `doi:10.1090/s0002-9947-1958-0095135-x`.

16 D. M. Kan. Abstract homotopy I. *Proc. Nat. Acad. Sci. USA*, 41(12):1092–1096, 1955. URL: `http://www.pnas.org/content/41/12/1092.full.pdf`.

17 C. Kapulkin and P. LeFanu Lumsdaine. The simplicial model of univalent foundations (after Voevodsky), 2012. arXiv preprint 1211.2851. URL: `https://arxiv.org/abs/1211.2851`.

18 D. R. Licata and G. Brunerie. A cubical approach to synthetic homotopy theory. In *Proc. of 30th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS '15*, pages 92–103. IEEE, 2015. `doi:10.1109/lics.2015.19`.

19 P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North Holland, 1975. `doi:10.1016/s0049-237x(08)71945-1`.

20 P. Martin-Löf. An intuitionistic theory of types. In G. Sambin and J. M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Clarendon Press, 1998.

21 A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013. `doi:10.1017/cbo9781139084673`.

22 A. M. Pitts. Nominal presentation of cubical sets models of type theory. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *Proc. of 20th Int. Conf. on Types for Proofs and Programs, TYPES 2014*, volume 39 of *Leibniz Int. Proc. in Inform.*, pages 202–220. Dagstuhl Publishing, 2015. `doi:10.4230/lipics.types.2014.202`.

23 A. Polonsky. Extensionality of $\lambda *$. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *Proc. of 20th Int. Conf. on Types for Proofs and Programs, TYPES 2014*, volume 39 of *Leibniz Int. Proc. in Inform.*, pages 221–250. Dagstuhl Publishing, 2015. `doi:10.4230/lipics.types.2014.221`.

24 T. Streicher. *Semantics of Type Theory: Correctness, Completeness and Independence Results*. Progress in Theoretical Computer Science. Birkhäuser, 1991. `doi:10.1007/978-1-4612-0433-6`.

25 A. Swan. An algebraic weak factorisation system on 01-substitution sets: A constructive proof, 2014. arXiv preprint 1409.1829. URL: `https://arxiv.org/abs/1409.1829`.

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

27 V. Voevodsky. The equivalence axiom and univalent models of type theory (talk at CMU on feb. 4, 2010), 2014. arXiv preprint 1402.5556. URL: `https://arxiv.org/abs/1402.5556`.

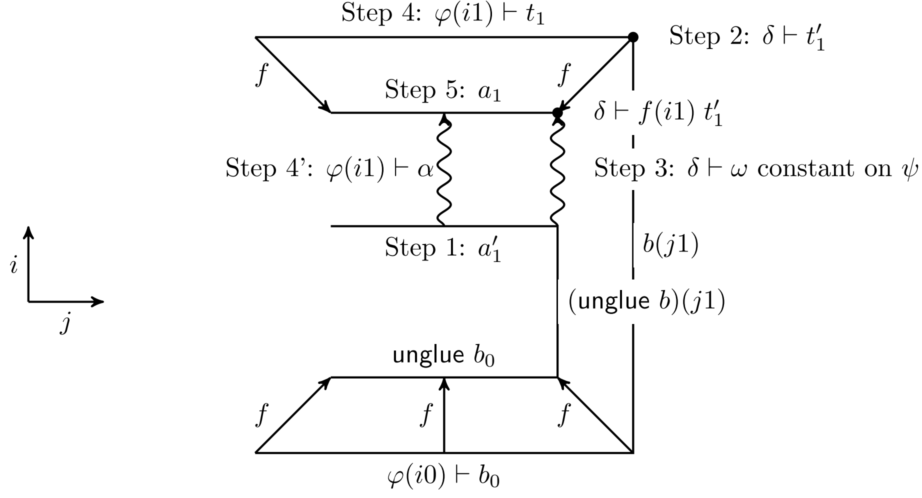


Figure 5 Composition for glueing.

A Details of Composition for Glueing

We build the element $\Gamma \vdash b_1 = \mathsf{comp}^i\ B\ [\psi \mapsto b]\ b_0 : (\mathsf{Glue}\ [\varphi \mapsto (T, f)]\ A)(i1)$ as the element $\mathsf{glue}\ [\varphi(i1) \mapsto t_1]\ a_1$ where

$$\Gamma, \varphi(i1) \vdash t_1 : T(i1)[\psi \mapsto b(i1)]$$
$$\Gamma \vdash a_1 : A(i1)[\varphi(i1) \mapsto f(i1)\ t_1, \psi \mapsto (\mathsf{unglue}\ b)(i1)]$$

As intermediate steps, we gradually build elements that satisfy more and more of the equations that the final elements t_1 and a_1 should satisfy. The construction of these is given in five steps.

Before explaining how we can define them and why they are well defined, we illustrate the construction in Figure 5, with $\psi = (j = 1)$ and $\varphi = (i = 0) \vee (j = 1) \vee (i = 1)$.

We pose $\delta = \forall i.\varphi$ (cf. Section 3), so that we have that δ is independent from i, and in our example $\delta = (j = 1)$ and it represents the right-hand side of the picture.

1. The element $a'_1 : A(i1)$ is a first approximation of a_1, but a'_1 is not necessarily in the image of $f(i1)$ in $\Gamma, \varphi(i1)$;
2. the partial element $\delta \vdash t'_1 : T(i1)$, which is a partial final result for $\varphi(i1) \vdash t_1$;
3. the partial path $\delta \vdash \omega$, between a'_1 and the image of t'_1;
4. both the final element $\varphi(i1) \vdash t_1$ and a path $\varphi(i1) \vdash \alpha$ between a'_1 and $f(i1)\ t_1$;
5. finally, we build a_1 from a'_1 and α.

We define:

$$\Gamma, \psi, i : \mathbb{I} \vdash a = \mathsf{unglue}\ b : A[\varphi \mapsto f\ b]$$
$$\Gamma \vdash a_0 = \mathsf{unglue}\ b_0 : A(i0)[\varphi(i0) \mapsto f(i0)\ b_0, \psi \mapsto a(i0)]$$

Step 1 We define a'_1 as the composition of a and $\mathsf{unglue}\ b_0$, in the direction i, which is well defined since $\mathsf{unglue}\ b_0 = (\mathsf{unglue}\ b)(i0)$ over the extent ψ

$$\Gamma \vdash a'_1 = \mathsf{comp}^i\ A\ [\psi \mapsto\ a]\ a_0 : A(i1)[\psi \mapsto\ a(i1)] \quad (4)$$

Step 2 We also define t'_1 as the composition of b and b_0, in the direction i:

$$\Gamma, \delta \vdash t'_1 = \mathsf{comp}^i\ T\ [\psi \mapsto\ b]\ b_0 : T(i1)[\psi \mapsto\ b(i1)] \tag{5}$$

$$\text{which is well defined because} \quad \begin{cases} \Gamma, \delta, i : \mathbb{I}, \psi \vdash b : T & \text{by Lemma 3} \\ \Gamma, \delta \vdash b_0 : T(i0)[\psi \mapsto b(i0)] & \text{as}\quad \delta \leqslant \varphi(i0) \end{cases}$$

Moreover, since

$$\begin{cases} \Gamma, \delta, \psi, i : \mathbb{I} \vdash a = f\ b & \text{by}\quad \delta \leqslant \varphi \\ \Gamma, \delta \vdash\ a_0 = f(i0)\ b_0 & \text{by}\quad \delta \leqslant \varphi(i0) \end{cases}$$

we can re-express a'_1 on the extent δ

$$\Gamma, \delta \vdash a'_1 = \mathsf{comp}^i\ A\ [\psi \mapsto f\ b]\ \ (f(i0)\ b_0)$$

Step 3 We can hence find a path ω connecting a'_1 and $f(i1)\ t'_1$ in Γ, δ using Lemma 6:

$$\Gamma, \delta \vdash \omega = \mathsf{pres}^i\ f\ [\psi \mapsto b]\ b_0 : (\mathsf{Path}\ A(i1)\ a'_1\ \ (f(i1)\ t'_1))\ [\psi \mapsto \langle j \rangle\ a(i1)]$$

Picking a fresh name j, we have

$$\Gamma, \delta, j : \mathbb{I} \vdash \omega\ j : A(i1)[(j = 0) \mapsto a'_1, (j = 1) \mapsto f(i1)\ t'_1, \psi \mapsto a(i1)] \tag{6}$$

Step 4 Now we define the final element t_1 as the inverse image of a'_1 by $f(i1)$, together with the path α between a'_1 and $f(i1)\ t_1$, in $\Gamma, \varphi(i1) \vdash$, using Lemma 7:

$$\Gamma, \varphi(i1) \vdash (t_1, \alpha) = \mathsf{equiv}\ f(i1)\ [\delta \mapsto\ (t'_1, \omega), \psi \mapsto\ (b(i1), \langle j \rangle\ a'_1)]\ a'_1$$

$$\text{with} \quad \begin{cases} \Gamma, \varphi(i1) \vdash t_1 : T(i1)[\delta \mapsto\ t'_1, \psi \mapsto\ b(i1)] \\ \Gamma, \varphi(i1) \vdash \alpha : (\mathsf{Path}\ A(i1)\ a'_1\ \ (f(i1)\ t_1))\ [\delta \mapsto \omega, \psi \mapsto\ \langle j \rangle\ a'_1)] \end{cases}$$

These are well defined because the two systems in δ and ψ are compatible:

$$\begin{cases} \Gamma, \delta, \psi \vdash t'_1 = b(i1) & \text{by (5)} \\ \Gamma, \delta, \psi \vdash \omega = \langle j \rangle\ a'_1 & \text{by (6) and (4)} \end{cases}$$

Picking a fresh name j, we have

$$\Gamma, \varphi(i1), j : \mathbb{I} \vdash \alpha\ j : A(i1)[(j = 0) \mapsto a'_1,\ (j = 1) \mapsto f(i1)\ t_1,\ \delta \mapsto\ a'_1,\ \psi \mapsto a(i1)] \tag{7}$$

Step 5 Finally, we define a_1 by composition of α and a'_1:

$$\Gamma \vdash\ a_1 := \mathsf{comp}^j\ A(i1)\ [\varphi(i1) \mapsto \alpha\ j, \psi \mapsto\ a(i1)]\ a'_1 : A(i1)[\varphi(i1) \mapsto \alpha\ 1, \psi \mapsto\ a(i1)]$$

$$\text{which is well defined because} \quad \begin{cases} \Gamma, j : \mathbb{I}, \varphi(i1), \psi \vdash \alpha\ j = a(i1) & \text{by (7)} \\ \Gamma, \varphi(i1) \vdash \alpha\ 0 = a'_1 & \text{by (7)} \\ \Gamma, \psi \vdash a(i1) = a'_1 & \text{by (4)} \end{cases}$$

and since $\Gamma, \varphi(i1) \vdash \alpha\ 1 = f(i1)\ t_1$, we can re-express the type of a_1 in the following way:

$$\Gamma \vdash a_1 : A(i1)[\varphi(i1) \mapsto f(i1)\ t_1,\ \psi \mapsto a(i1)]$$

Which is exactly what we needed to build $\Gamma \vdash b_1 := \mathsf{glue}\ [\varphi(i1) \mapsto t_1]\ a_1 : B(i1)[\psi \mapsto b(i1)]$.

Finally we check that $b_1 = \mathsf{comp}^i\ T\ [\psi \mapsto\ b]\ b_0$ on δ:

$$
\begin{aligned}
b_1 &= \mathsf{glue}\ [\varphi(i1) \mapsto t_1]\ a_1 && \text{by def.}\\
&= t_1 : T(i1)[\delta \mapsto\ t_1', \psi \mapsto\ b(i1)] && \text{as } \varphi(i1) = 1_{\mathbb{F}}\\
&= t_1' && \text{as } \delta = 1_{\mathbb{F}}\\
&= \mathsf{comp}^i\ T\ [\psi \mapsto\ b]\ b_0 && \text{by def.}
\end{aligned}
$$

B Univalence from Glueing

We also give two alternative proofs of the univalence axiom for Path only involving the glue construction.[11] The first is a direct proof of the standard formulation of the univalence axiom while the second goes through an alternative formulation as in Corollary 10.[12]

▶ **Lemma 25.** *For $\Gamma \vdash A : \mathsf{U}$, $\Gamma \vdash B : \mathsf{U}$, and an equivalence $\Gamma \vdash f : \mathsf{Equiv}\ A\ B$ we have the following constructions:*

1. $\Gamma \vdash \mathsf{eqToPath}\ f : \mathsf{Path}\,\mathsf{U}\,A\,B$;
2. $\Gamma \vdash \mathsf{Path}\,(A \to B)\,(\mathsf{transp}^i(\mathsf{eqToPath}\,f\,i))\,f.1$ *is inhabited; and*
3. *if $f = \mathsf{equiv}^i(P\,i)$ for $\Gamma \vdash P : \mathsf{Path}\,\mathsf{U}\,A\,B$, then the following type is inhabited:*

$$\Gamma \vdash \mathsf{Path}\,(\mathsf{Path}\,\mathsf{U}\,A\,B)\,(\mathsf{eqToPath}\,(\mathsf{equiv}^i(P\,i)))\,P$$

Proof. For (1) we define

$$\mathsf{eqToPath}\,f = \langle i \rangle\ \mathsf{Glue}\,[(i = 0) \mapsto (A, f), (i = 1) \mapsto (B, \mathsf{equiv}^k B)]\,B. \tag{8}$$

Note that here $\mathsf{equiv}^k B$ is an equivalence between B and B (see Section 7.1). For (2) we have to closely look at how the composition was defined for Glue. By unfolding the definition, we see that the left-hand side of the equality is equal $f.1$ composed with multiple transports in a constant type; using filling and functional extensionality, these transports can be shown to be equal to the identity; for details see the formal proof.

The term for (3) is given by:

$$
\begin{aligned}
\langle j \rangle\ \langle i \rangle\ \mathsf{Glue}\,[&(i = 0) \mapsto (A, \mathsf{equiv}^k(P\,k)),\\
&(i = 1) \mapsto (B, \mathsf{equiv}^k B),\\
&(j = 1) \mapsto (P\,i, \mathsf{equiv}^k(P(i \vee k)))]\\
&B
\end{aligned}
$$
◀

▶ **Corollary 26** (Univalence axiom)**.** *For the canonical map*

$$\mathsf{pathToEq} : (A\,B : \mathsf{U}) \to \mathsf{Path}\,\mathsf{U}\,A\,B \to \mathsf{Equiv}\ A\ B$$

we have that $\mathsf{pathToEq}\,A\,B$ is an equivalence for all $A : \mathsf{U}$ and $B : \mathsf{U}$.

---

[11] The proofs of the univalence axiom have all been formally verified inside the system using the Haskell implementation. We note that the proof of Theorem 9 can be given such that it extends $f.2$ and hence in Corollary 10 we do not need the fact that $\mathsf{isEquiv}\ X\ A\ f.1$ is a proposition. For details see: `https://github.com/mortberg/cubicaltt/blob/v1.0/examples/univalence.ctt`

[12] The second of these proofs is inspired by a proof by Dan Licata from: `https://groups.google.com/d/msg/homotopytypetheory/j2KBIvDw53s/YTDK4D0NFQAJ`

Proof 1. Let us first show that the canonical map $\mathsf{pathToEq}$ is path equal to:

$$\mathsf{equiv} = \lambda A\, B : \mathsf{U}.\, \lambda P : \mathsf{Path}\,\mathsf{U}\,A\,B.\, \mathsf{equiv}^i (P\,i)$$

By function extensionality, it suffices to check this pointwise. Using path-induction, we may assume that P is reflexivity. In this case $\mathsf{pathToEq}\,A\,A\,1_A$ is the identity equivalence by definition. Because being an equivalence is a proposition, it thus suffices that the first component of $\mathsf{equiv}^i A$ is propositionally equal to the identity. By definition, this first component is given by transport (now in the constant type A) which is easily seen to be the identity using filling (see Section 4.4).

Thus it suffices to prove that $\mathsf{equiv}\,A\,B$ is an equivalence. To do so it is enough to give an inverse (see Theorems 4.2.3 and 4.2.6 of [26]). But $\mathsf{eqToPath}$ is a left inverse by Lemma 25 (3), and a right inverse by Lemma 25 (2) using that being an equivalence is a proposition. ◀

Proof 2. Points (1) and (2) of Lemma 25 imply that $\mathsf{Equiv}\ A\ B$ is a retract of $\mathsf{Path}\,\mathsf{U}\,A\,B$. Hence $(X : \mathsf{U}) \times \mathsf{Equiv}\ A\ X$ is a retract of $(X : \mathsf{U}) \times \mathsf{Path}\,\mathsf{U}\,A\,X$. But $(X : \mathsf{U}) \times \mathsf{Path}\,\mathsf{U}\,A\,X$ is contractible, so $(X : \mathsf{U}) \times \mathsf{Equiv}\ A\ X$ is also contractible as a retract of a contractible type. As discussed in Section 7.2 this is an alternative formulation of the univalence axiom and the rest of this proof follows as there. ◀

Note that the first proof uses all three of the points of Lemma 25 while the second proof only uses the first two. As the second proof only uses the first two points it is possible to prove it if point (1) is defined as in Example 8 leading to a slightly simpler proof of point (2).

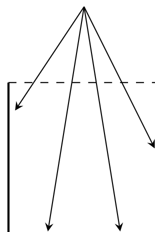
C Singular Cubical Sets

Recall the functor $\mathcal{C} \to \mathbf{Top}, I \mapsto [0,1]^I$ given at (1) in Section 8.1. In particular, the face maps $(ib)\colon I - i \to I$ (for $b = 0_{\mathbb{I}}$ or $1_{\mathbb{I}}$) induce the maps $(ib)\colon [0,1]^{I-i} \to [0,1]^I$ by $i(ib)u = b$ and $j(ib)u = ju$ if $j \neq i$ is in I. If ψ is in $\mathbb{F}(I)$ and u in $[0,1]^I$, then ψu is a truth value.

We assume given a family of idempotent functions $r_I : [0,1]^I \times [0,1] \to [0,1]^I \times [0,1]$ such that

1. $r_I(u, z) = (u, z)$ iff $\partial_I u = 1$ or $z = 0$, and
2. for any *strict* f in $\mathsf{Hom}(I, J)$ we have $r_J(f \times \mathrm{id}) r_I = r_J(f \times \mathrm{id})$.

Such a family can for instance be defined as in the following picture ("retraction from above center"). If the center has coordinate $(1/2, 2)$, then $r_I(u, z) = r_I(u', z')$ is equivalent to $(2 - z')(-1 + 2u) = (2 - z)(-1 + 2u')$.



Property (1) holds for the retraction defined by this picture. The property (2) can be reformulated as $r_I(u, z) = r_I(u', z') \to r_J(fu, z) = r_J(fu', z')$. It also holds in this case,

since $r_I(u,z) = r_I(u',z')$ is then equivalent to $(2-z')(-1+2u) = (2-z)(-1+2u')$, which implies $(2-z')(-1+2fu) = (2-z)(-1+2fu')$ if f is strict.

Using this family, we can define for each ψ in $\mathbb{F}(I)$ an idempotent function

$$r_\psi : [0,1]^I \times [0,1] \to [0,1]^I \times [0,1]$$

having for fixed-points the element (u,z) such that $\psi u = 1$ or $z = 0$. This function r_ψ is completely characterized by the following properties:

1. $r_\psi = \mathrm{id}$ if $\psi = 1$
2. $r_\psi = r_\psi r_I$ if $\psi \neq 1$
3. $r_\psi(u,z) = (u,z)$ if $z = 0$
4. $r_\psi((ib) \times \mathrm{id}) = ((ib) \times \mathrm{id}) r_{\psi(ib)}$

These properties imply for instance $r_{\partial_I}(u,z) = (u,z)$ if $\partial_I u = 1$ or $z = 0$ and so they imply $r_{\partial_I} = r_I$. They also imply that $r_\psi(u,z) = (u,z)$ if $\psi u = 1$.

From these properties, we can prove the uniformity of the family of functions r_ψ.

▶ **Theorem 27.** *If f is in* $\mathsf{Hom}(I,J)$ *and ψ is in $\mathbb{F}(J)$, then $r_\psi(f \times \mathrm{id}) = (f \times \mathrm{id}) r_{\psi f}$.*

This is proved by induction on the number of element of I (the result being clear if I is empty).

A particular case is $r_J(f \times \mathrm{id}) = (f \times \mathrm{id}) r_{\partial_J f}$. Note that, in general, $\partial_J f$ is not ∂_I.

A direct consequence of the previous theorem is the following.

▶ **Corollary 28.** *The singular cubical set associated to a topological space has a composition structure.*

Efficient Type Checking for Path Polymorphism*

Juan Edi[1], Andrés Viso[2], and Eduardo Bonelli[3]

1 **Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Pabellón I, Ciudad Universitaria, Buenos Aires C1428EGA, Argentina**
jedi@dc.uba.ar

2 **Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina and**
Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Pabellón I, Ciudad Universitaria, Buenos Aires C1428EGA, Argentina
aeviso@gmail.com

3 **Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina and**
Departamento de Ciencia y Tecnología, Universidad Nacional de Quilmes, Roque Sáenz Peña 352, Bernal B1876BXD, Argentina
eabonelli@gmail.com

Abstract

A type system combining type application, constants as types, union types (associative, commutative and idempotent) and recursive types has recently been proposed for statically typing *path polymorphism*, the ability to define functions that can operate uniformly over recursively specified applicative data structures. A typical pattern such functions resort to is $x\,y$ which decomposes a compound, in other words any applicative tree structure, into its parts. We study type-checking for this type system in two stages. First we propose algorithms for checking type equivalence and subtyping based on coinductive characterizations of those relations. We then formulate a syntax-directed presentation and prove its equivalence with the original one. This yields a type-checking algorithm which unfortunately has exponential time complexity in the worst case. A second algorithm is then proposed, based on automata techniques, which yields a polynomial-time type-checking algorithm.

1998 ACM Subject Classification F.4.1 Mathematical Logic and Formal Languages: Lambda Calculus and Related Systems; F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages; D.3.3 Programming Languages: Language Constructs and Features

Keywords and phrases λ-calculus, pattern matching, path polymorphism, type checking

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.6

1 Introduction

The *lambda-calculus* plays an important role in the study of programming languages (PLs). Programs are represented as syntactic terms and execution by repeated simplification of these terms using a reduction rule called *β-reduction*. The study of the lambda-calculus has produced deep results in both the theory and the implementation of PLs. Many variants of the lambda-calculus have been introduced with the purpose of studying specific PL features.

* A full version of the paper is available at [9], https://arxiv.org/abs/1704.09026.

© Juan Edi, Andrés Viso, and Eduardo Bonelli;
licensed under Creative Commons License CC-BY
21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 6; pp. 6:1–6:24

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

One such feature of interest is *pattern-matching*. Pattern-matching is used extensively in PLs as a means for writing more succinct and, at the same time, elegant programs. This is particularly so in the functional programming community, but by no means restricted to that community.

In the standard lambda-calculus, functions are represented as expressions of the form $\lambda x.t$, x being the formal parameter and t the body. Such a function may be applied to any term, regardless of its form. This is expressed by the above mentioned β-reduction rule: $(\lambda x.t)\,s \to_\beta \{s/x\}\,t$, where $\{s/x\}\,t$ stands for the result of replacing all free occurrences of x in t with s. Note that, in this rule, no requirement on the form of s is placed. *Pattern calculi* are generalizations of the β-reduction rule in which abstractions $\lambda x.t$ are replaced by $\lambda p.t$ where p is called a *pattern*. An example is $\lambda\langle x, y\rangle.x$ for projecting the first component of a pair, the pattern p being $\langle x, y\rangle$. An expression such as $(\lambda\langle x, y\rangle.x)\,s$ will only be able to reduce if s indeed is of the form $\langle s_1, s_2\rangle$; it will otherwise be blocked.

Patterns may be catalogued in at least two dimensions. One is their *structure* and another their *time of creation*. The structure of patterns may be very general. Such is the case of variables: any term can match a variable, as in the standard lambda-calculus. The structure of a pattern may also be very specific. Such is the case when arbitrary terms are allowed to be patterns [13, 17]. Regarding the time of creation, patterns may either be *static* or *dynamic*. Static patterns are those that are created at compile time, such as the pattern $\langle x, y\rangle$ mentioned above. Dynamic patterns are those that may be generated at run-time [10, 11]. For example, consider the term $\lambda x.(\lambda(x\,y).y)$; note that it has an occurrence of a pattern $x\,y$ with a free variable, namely the x in $x\,y$, that is bound to the outermost lambda. If this term is applied to a constant $\mathtt{c}$, then one obtains $\lambda\mathtt{c}\,y.y$. However, if we apply it to the constant $\mathtt{d}$, then we obtain $\lambda\mathtt{d}\,y.y$. Both patterns $\mathtt{c}\,y$ and $\mathtt{d}\,y$ are created during execution. Note that one could also replace the x in the pattern $x\,y$ with an abstraction. This leads to computations that evaluate to patterns.

Expressive pattern features may easily break desired properties, such as confluence, and are not easy to endow with type systems. This work is an attempt at devising type systems for such expressive pattern calculi. We originally set out to type-check the *Pure Pattern Calculus* (PPC) [10, 11]. PPC is a lambda-calculus that embodies the essence of dynamic patterns by stripping away everything inessential to the reduction and matching process of dynamic patterns. It admits terms such as $\lambda x.(\lambda(x\,y).y)$. We soon realized that typing PPC was too challenging and noticed that the static fragment of PPC, which we dub *Calculus of Applicative Patterns* (CAP), was already challenging in itself. CAP also admits patterns such as $x\,y$ however all variables in this pattern are considered *bound*. Thus, in a term such as $\lambda(x\,y).s$ both occurrences of x and y are bound in s, disallowing reduction inside patterns. Such patterns, however, allow arguments that are applications to be decomposed, as long as these applications encode *data structures*. They are therefore useful for writing functions that operate on *semi-structured data*.

The main obstacle for typing CAP is dealing in the type system with a form of polymorphism called *path polymorphism* [10, 11], that arises from these kinds of patterns. We next briefly describe path polymorphism and the requirements it places on typing considerations.

Path Polymorphism. In CAP data structures are trees. These trees are built using application and variable arity constants or constructors. Examples of two such trees follow, where the first one represents a list and the second a binary tree:

$$\mathtt{cons}\,(\mathtt{vl}\,1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$$
$$\mathtt{node}\,(\mathtt{vl}\,3)\,(\mathtt{node}\,(\mathtt{vl}\,4)\,\mathtt{nil}\,\mathtt{nil})\,(\mathtt{node}\,(\mathtt{vl}\,5)\,\mathtt{nil}\,\mathtt{nil})$$

The constructor vl is used to tag values (1 and 2 in the first case, and 3, 4 and 5 in the

second). A "map" function for updating the values of any of these two structures by applying some user-supplied function f follows, where type annotations are omitted for clarity:

$$\begin{aligned} \mathsf{upd} = f \to \ (\ & \mathtt{vl}\, z \to \mathtt{vl}\,(f\, z) \\ |\ & x\, y \ \to (\mathsf{upd}\, f\, x)\,(\mathsf{upd}\, f\, y) \\ |\ & w \quad \to w) \end{aligned} \tag{1}$$

The expression $\mathsf{upd}\,(+1)$ may thus be applied to any of the two data structures illustrated above. For example, we can evaluate $\mathsf{upd}\,(+1)\,\mathtt{cons}\,(\mathtt{vl}\,1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$ and also $\mathsf{upd}\,(+1)\,\mathtt{node}\,(\mathtt{vl}\,3)\,(\mathtt{node}\,(\mathtt{vl}\,4)\,\mathtt{nil}\,\mathtt{nil})\,(\mathtt{node}\,(\mathtt{vl}\,5)\,\mathtt{nil}\,\mathtt{nil})$. The expression to the right of "=" is called an *abstraction* (or *case*) and consists of a unique *branch*; this branch in turn is formed from a pattern (f), and a body (in this case the body is itself another abstraction that consists of three branches). An argument to an abstraction is matched against the patterns, in the order in which they are written, and the appropriate body is selected.

Notice the pattern $x\,y$. During evaluation of $\mathsf{upd}\,(+1)\,\mathtt{cons}\,(\mathtt{vl}\,1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$ the variables x and y may be instantiated with different applicative terms in each recursive call to upd. For example:

	x	y
$\mathsf{upd}\,(+1)\,s$	$\mathtt{cons}\,(\mathtt{vl}\,1)$	$\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil}$
$\mathsf{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{vl}\,1))$	$\mathtt{cons}$	$\mathtt{vl}\,1$
$\mathsf{upd}\,(+1)\,(\mathtt{cons}\,(\mathtt{vl}\,2)\,\mathtt{nil})$	$\mathtt{cons}\,(\mathtt{vl}\,2)$	$\mathtt{nil}$

The type assigned to x and y should encompass all terms in its respective column.

Singleton Types and Type Application. A further consideration in typing CAP is that terms such as the ones depicted below should clearly not be typable.

$$(\mathtt{nil} \to 0)\,\mathtt{cons} \qquad\qquad (\mathtt{vl}\,x \to_{\{x:\mathsf{Nat}\}} x + 1)\,(\mathtt{vl}\,\mathtt{true}) \tag{2}$$

In the first case, $\mathtt{cons}$ will never match $\mathtt{nil}$. The type system will resort to singleton types in order to determine this: $\mathtt{cons}$ will be assigned a type of the form $\mathbb{cons}$ which will fail to match $\mathbb{nil}$. The second expression in (2) breaks *Subject Reduction* (SR): reduction will produce $\mathtt{true} + 1$. Applicative types of the form $\mathbb{vl} @ \mathbb{true}$ will allow us to check for these situations, @ being a new type constructor that applies datatypes to arbitrary types. Also, note the use of typing environments (the expression $\{x : \mathsf{Nat}\}$) to declare the types of the variables of patterns in branches. These are supplied by the programmer.

Union and Recursive Types. On the assumption that the programmer has provided an exhaustive coverage, the type assigned by CAP to the variable x in the pattern $x\,y$ in upd is:

$$\mu\alpha.(\mathbb{vl} @ A) \oplus (\alpha @ \alpha) \oplus (\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil})$$

Here μ is the recursive type constructor and $\oplus$ the union type constructor. $\mathbb{vl}$ is the singleton type used for typing the constant $\mathtt{vl}$ and @ denotes type application, as mentioned above. The union type constructor is used to collect the types of all the branches. The variable y in the pattern $x\,y$ will also be assigned the same type as x. Thus variables in applicative patterns are assigned union types. upd itself is assigned type $(A \supset B) \supset (F_A \supset F_B)$, where F_X is $\mu\alpha.(\mathbb{vl} @ X) \oplus (\alpha @ \alpha) \oplus (\mathbb{cons} \oplus \mathbb{node} \oplus \mathbb{nil})$.

Type-Checking for CAP. Based on these, and other similar considerations, we proposed *typed CAP* [18], referred to simply as CAP in the sequel. The system consists of typing rules that combine singleton types, type application, union types, recursive types and subtyping. Also it enjoys several properties, the salient one being safety (subject reduction and progress). Safety relies on a notion of *typed pattern compatibility* based on subtyping that guarantees that examples such as (2–right) and the following one do not break safety:

$$((\mathtt{vl}\, x \rightarrow_{\{x:\mathsf{Bool}\}} \mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \mid (\mathtt{vl}\, y \rightarrow_{\{y:\mathsf{Nat}\}} y+1))\,(\mathtt{vl}\, 4) \tag{3}$$

Assumptions on associativity and commutativity of typing operators in [18], make it non-trivial to deduce a type-checking algorithm from the typing rules. The proposed type system is, moreover, not syntax-directed. Also, it relies on coinductive notions of type equivalence and subtyping which in the presence of recursive subtypes are not obviously decidable. A practical implementation of CAP is instrumental since a robust theoretical analysis without such an implementation is of little use.

Goal and Summary of Contributions. This paper addresses this implementation. It does so in two stages:

- The first stage presents a naïve but correct, high-level description of a type-checking algorithm, the principal aim being clarity. We propose an invertible presentation of the coinductive notions of type-equivalence and subtyping of [18] and also a syntax-directed variant of the presentation in [18]. This leads to algorithms for checking subtyping membership and equivalence modulo associative, commutative and idempotent (ACI) unions, both based on an invertible presentation of the functional generating the associated coinductive notions.
- The second stage builds on ideas from the first algorithm with the aim of improving efficiency. μ-types are interpreted as infinite n-ary trees and represented using automata, avoiding having to explicitly handle unfoldings of recursive types, and leading to a significant improvement in the complexity of the key steps of the type-checking process, namely equality and subtype checking.

Related work. For literature on (typed) pattern calculi the reader is referred to [18]. The algorithms for checking equality of recursive types or subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli [1]; Kozen, Palsberg, and Schwartzbach [14]; Brandt and Henglein [3]; Jim and Palsberg [12] among others. Additionally, Zhao and Palsberg [15] studied the possibilities of incorporating associative and commutative (AC) products to the equality check, on an automata-based approach that the authors themselves claimed was not extensible to subtyping [20]. Later on Di Cosmo, Pottier, and Rémy [6] presented another automata-based algorithm for subtyping that properly handles AC products with a complexity cost of $\mathcal{O}(n^2 n'^2 d^{5/2})$, where n and n' are the sizes of the analyzed types, and d is a bound on the arity of the involved products.

Structure of the paper. Sec. 2 reviews the syntax and operational semantics of CAP, its type system and the main properties. Further details may be consulted in [18]. Sec. 3 proposes invertible generating functions for coinductive notions of type-equivalence and subtyping that lead to inefficient but elegant algorithms for checking these relations. Sec. 4 proposes a syntax-directed type system for CAP. Sec. 5 studies a more efficient type-checking algorithm based on automaton. Finally, we conclude in Sec. 6. Full details of all omitted proofs may be found in an extended report [9]. An implementation of the algorithms described here is available online [8].

2 Review of CAP

2.1 Syntax and Operational Semantics

We assume given an infinite set of term variables $\mathbb{V}$ and constants $\mathbb{C}$. CAP has four syntactic categories, namely **patterns** $(p, q, \ldots)$, **terms** $(s, t, \ldots)$, **data structures** $(d, e, \ldots)$ and **matchable forms** $(m, n, \ldots)$:

$$
\begin{array}{rcll}
p & ::= & x & \text{(matchable)} \\
 & | & \mathsf{c} & \text{(constant)} \\
 & | & p\,p & \text{(compound)}
\end{array}
\qquad
\begin{array}{rcll}
t & ::= & x & \text{(variable)} \\
 & | & \mathsf{c} & \text{(constant)} \\
 & | & t\,t & \text{(application)} \\
 & | & p \to_\theta t \mid \ldots \mid p \to_\theta t & \text{(abstraction)}
\end{array}
$$

$$
\begin{array}{rcll}
d & ::= & \mathsf{c} & \text{(constant)} \\
 & | & d\,t & \text{(compound)}
\end{array}
\qquad
\begin{array}{rcll}
m & ::= & d & \text{(data structure)} \\
 & | & p \to_\theta t \mid \ldots \mid p \to_\theta t & \text{(abstraction)}
\end{array}
$$

The set of patterns, terms, data structures and matchable forms are denoted $\mathbb{P}$, $\mathbb{T}$, $\mathbb{D}$ and $\mathbb{M}$, resp. Variables occurring in patterns are called **matchables**. We often abbreviate $p_1 \to_{\theta_1} s_1 \mid \ldots \mid p_n \to_{\theta_n} s_n$ with $(p_i \to_{\theta_i} s_i)_{i \in 1..n}$. The θ_i are typing contexts annotating the type assignments for the variables in p_i (*cf.* Sec. 2.3). The **free variables** of a term t (notation $\mathsf{fv}(t)$) are defined as expected; in a pattern p we call them **free matchables** ($\mathsf{fm}(p)$). All free matchables in each p_i are assumed to be bound in their respective bodies s_i. Positions in patterns and terms are defined as expected and denoted $\pi, \pi', \ldots$ (ϵ denotes the root position). We write $\mathsf{pos}(s)$ for the set of positions of s and $s|_\pi$ for the subterm of s occurring at position π.

A **substitution** $(\sigma, \sigma_i, \ldots)$ is a partial function from term variables to terms. If it assigns u_i to x_i, $i \in 1..n$, then we write $\{u_1/x_1, \ldots, u_n/x_n\}$. Its domain $(\mathsf{dom}\,(\sigma))$ is $\{x_1, \ldots, x_n\}$. Also, $\{\}$ is the identity substitution. We write σs for the result of applying σ to term s. We say a pattern p **subsumes** a pattern q, written $p \lhd q$ if there exists σ such that $\sigma p = q$. **Matchable forms** are required for defining the **matching operation**, described next.

Given a pattern p and a term s, the matching operation $\{\!\{s/p\}\!\}$ determines whether s matches p. It may have one of three outcomes: success, fail (in which case it returns the special symbol `fail`) or undetermined (in which case it returns the special symbol `wait`). We say $\{\!\{s/p\}\!\}$ is **decided** if it is either successful or it fails. In the former it yields a substitution σ; in this case we write $\{\!\{s/p\}\!\} = \sigma$. The disjoint union of matching outcomes is given as follows ("$\triangleq$" is used for definitional equality):

$$
\begin{array}{rcl}
\mathtt{fail} \uplus o & \triangleq & \mathtt{fail} \\
o \uplus \mathtt{fail} & \triangleq & \mathtt{fail} \\
\sigma_1 \uplus \sigma_2 & \triangleq & \sigma
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{wait} \uplus \sigma & \triangleq & \mathtt{wait} \\
\sigma \uplus \mathtt{wait} & \triangleq & \mathtt{wait} \\
\mathtt{wait} \uplus \mathtt{wait} & \triangleq & \mathtt{wait}
\end{array}
$$

where o denotes any possible output and $\sigma_1 \uplus \sigma_2 \triangleq \sigma$ if the domains of σ_1 and σ_2 are disjoint. This always holds given that patterns are assumed to be linear (at most one occurrence of any matchable). The matching operation is defined as follows, where the defining clauses below are evaluated from top to bottom[1]:

[1] This is simplification to the static patterns case of the matching operation introduced in [10].

$$
\begin{array}{llll}
\{\!\{u/x\}\!\} & \triangleq & \{u/x\} & \\
\{\!\{\mathtt{c}/\mathtt{c}\}\!\} & \triangleq & \{\} & \\
\{\!\{u\,v/p\,q\}\!\} & \triangleq & \{\!\{u/p\}\!\} \uplus \{\!\{v/q\}\!\} & \text{if } u\,v \text{ is a } \textit{matchable form} \\
\{\!\{u/p\}\!\} & \triangleq & \mathtt{fail} & \text{if } u \text{ is a } \textit{matchable form} \\
\{\!\{u/p\}\!\} & \triangleq & \mathtt{wait} &
\end{array}
$$

For example: $\{\!\{x \to s/\mathtt{c}\}\!\} = \mathtt{fail}$; $\{\!\{\mathtt{d}/\mathtt{c}\}\!\} = \mathtt{fail}$; $\{\!\{x/\mathtt{c}\}\!\} = \mathtt{wait}$ and $\{\!\{x\,\mathtt{d}/\mathtt{c}\,\mathtt{c}\}\!\} = \mathtt{fail}$. We now turn to the only reduction axiom of CAP:

$$
\frac{\{\!\{u/p_i\}\!\} = \mathtt{fail} \text{ for all } i < j \quad \{\!\{u/p_j\}\!\} = \sigma_j \quad j \in 1..n}{(p_i \to_{\theta_i} s_i)_{i \in 1..n}\, u \to \sigma_j s_j}\ (\beta)
$$

It may be applied under any context and states that if the argument u to an abstraction $(p_i \to_{\theta_i} s_i)_{i\in 1..n}$ fails to match all patterns p_i with $i < j$ and successfully matches pattern p_j (producing a substitution σ_j), then the term $(p_i \to_{\theta_i} s_i)_{i\in 1..n}\, u$ reduces to $\sigma_j s_j$.

For instance, consider the function

$$
\mathsf{head} = ((\mathtt{nil} \to_{\{\}} \mathtt{nothing}) \mid (\mathtt{cons}\, x\, xs \to_{\{x:\mathsf{Nat}, xs:\mu\alpha.\mathbb{nil}\oplus\mathbb{cons}@\mathsf{Nat}@\alpha\}} \mathtt{just}\, x))
$$

Then, $\mathsf{head}\,\mathtt{nil} \to \mathtt{nothing}$ with $\{\!\{\mathtt{nil}/\mathtt{nil}\}\!\} = \{\}$, while $\mathsf{head}\,(\mathtt{cons}\,4\,\mathtt{nil}) \to \mathtt{just}\,4$ since $\{\!\{\mathtt{cons}\,x\,\mathtt{nil}/\mathtt{nil}\}\!\} = \mathtt{fail}$ and $\{\!\{\mathtt{cons}\,4\,\mathtt{nil}/\mathtt{cons}\,x\,xs\}\!\} = \{4/x, \mathtt{nil}/xs\}$.

▶ **Proposition 1.** *Reduction in CAP is confluent [18].*

2.2 Types

In order to ensure that patterns such as $x\,y$ decompose only data structures rather than arbitrary terms, we shall introduce two sorts of typing expressions: *types* and *datatypes*, the latter being strictly included in the former. We assume given countably infinite sets $\mathcal{V}_D$ of **datatype variables** $(\alpha, \beta, \ldots)$, $\mathcal{V}_A$ of **type variables** $(X, Y, \ldots)$ and $\mathcal{C}$ of **type constants** $(\mathbb{c}, \mathbb{d}, \ldots)$. We define $\mathcal{V} \triangleq \mathcal{V}_A \cup \mathcal{V}_D$ and use meta-variables $V, W, \ldots$ to denote an arbitrary element in it. Likewise, we write $a, b, \ldots$ for elements in $\mathcal{V} \cup \mathcal{C}$. The sets $\mathcal{T}_D$ of **μ-datatypes** and $\mathcal{T}$ of **μ-types**, resp., are inductively defined as follows:

$$
\begin{array}{llll}
D & ::= & \alpha & \text{(datatype variable)} \\
 & \mid & \mathbb{c} & \text{(atom)} \\
 & \mid & D\ @\ A & \text{(compound)} \\
 & \mid & D \oplus D & \text{(union)} \\
 & \mid & \mu\alpha.D & \text{(recursion)}
\end{array}
\qquad
\begin{array}{llll}
A & ::= & X & \text{(type variable)} \\
 & \mid & D & \text{(datatype)} \\
 & \mid & A \supset A & \text{(type abstraction)} \\
 & \mid & A \oplus A & \text{(union)} \\
 & \mid & \mu X.A & \text{(recursion)}
\end{array}
$$

▶ Remark. A type of the form $\mu\alpha.A$ is not valid in general since it may produce invalid unfoldings. For example, $\mu\alpha.\alpha \supset \alpha = (\mu\alpha.\alpha \supset \alpha) \supset (\mu\alpha.\alpha \supset \alpha)$, which fails to preserve sorting. On the other hand, types of the form $\mu X.D$ are not necessary since they denote the solution to the equation $X = D$, hence X is a variable representing a datatype, a role already fulfilled by α.

We consider $\oplus$ to bind tighter than $\supset$, while @ binds tighter than $\oplus$. *E.g.* $D\ @\ A \oplus A' \supset B$ means $((D\ @\ A) \oplus A') \supset B$. We write $A \neq \oplus$ to mean that the root symbol of A is different from $\oplus$; and similarly with the other type constructors. Expressions such as $A_1 \oplus \ldots \oplus A_n$ will be abbreviated $\oplus_{i\in 1..n} A_i$; this is sound since μ-types will be considered modulo associativity of $\oplus$. A type of the form $\oplus_{i\in 1..n} A_i$ where each $A_i \neq \oplus$, $i \in 1..n$, is called a **maximal union**.

$$\frac{}{\vdash A \simeq_\mu A}\ (\text{E-REFL}) \qquad \frac{\vdash A \simeq_\mu B \quad \vdash B \simeq_\mu C}{\vdash A \simeq_\mu C}\ (\text{E-TRANS}) \qquad \frac{\vdash A \simeq_\mu B}{\vdash B \simeq_\mu A}\ (\text{E-SYMM})$$

$$\frac{\vdash D \simeq_\mu D' \quad \vdash A \simeq_\mu A'}{\vdash D @ A \simeq_\mu D' @ A'}\ (\text{E-COMP}) \qquad \frac{\vdash A \simeq_\mu A' \quad \vdash B \simeq_\mu B'}{\vdash A \supset B \simeq_\mu A' \supset B'}\ (\text{E-FUNC})$$

$$\frac{}{\vdash A \oplus A \simeq_\mu A}\ (\text{E-UNION-IDEM}) \qquad \frac{}{\vdash A \oplus B \simeq_\mu B \oplus A}\ (\text{E-UNION-COMM})$$

$$\frac{}{\vdash A \oplus (B \oplus C) \simeq_\mu (A \oplus B) \oplus C}\ (\text{E-UNION-ASSOC})$$

$$\frac{\vdash A \simeq_\mu A' \quad \vdash B \simeq_\mu B'}{\vdash A \oplus B \simeq_\mu A' \oplus B'}\ (\text{E-UNION}) \qquad \frac{\vdash A \simeq_\mu B}{\vdash \mu V.A \simeq_\mu \mu V.B}\ (\text{E-REC})$$

$$\frac{}{\vdash \mu V.A \simeq_\mu \{\mu V.A/V\}\, A}\ (\text{E-FOLD}) \qquad \frac{\vdash A \simeq_\mu \{A/V\}\, B \quad \mu V.B \text{ contractive}}{\vdash A \simeq_\mu \mu V.B}\ (\text{E-CONTR})$$

Figure 1 Type equivalence for μ-types.

We often write $\mu V.A$ to mean either $\mu\alpha.D$ or $\mu X.A$. A **non-union μ-type** A is a μ-type of one of the following forms: α, $\mathbb{c}$, $D @ A'$, X, $A' \supset A''$ or $\mu V.B$ with B a non-union μ-type. We assume μ-types are **contractive**: $\mu V.A$ is contractive if V occurs in A only under a type constructor $\supset$ or $@$, if at all. For instance, $\mu X.X \supset \mathbb{c}$, $\mu X.X \supset X$ and $\mu X.\mathbb{c} @ X \oplus X$ are contractive while $\mu X.X$ and $\mu X.X \oplus X$ are not. We henceforth redefine $\mathcal{T}$ to be the set of **contractive μ-types**.

μ-types come equipped with a notion of **type equivalence** $\simeq_\mu$ (Fig. 1) and **subtyping** $\preceq_\mu$ (Fig. 2). In Fig. 2 a subtyping context Σ is a set of assumptions over type variables of the form $V \preceq_\mu W$ with $V, W \in \mathcal{V}$. (E-REC) actually encodes two rules, one for datatypes ($\mu\alpha.D$) and one for arbitrary types ($\mu X.A$). Likewise for (E-FOLD) and (E-CONTR). Regarding the subtyping rules, we adopt those for union of [19]. It should be noted that the naïve variant of (S-REC) in which $\Sigma \vdash \mu V.A \preceq_\mu \mu V.B$ is deduced from $\Sigma \vdash A \preceq_\mu B$, is known to be unsound [1]. We often abbreviate $\vdash A \preceq_\mu B$ as $A \preceq_\mu B$.

2.3 Typing and Safety

A **typing context** Γ (or θ) is a partial function from term variables to μ-types; $\Gamma(x) = A$ means that Γ maps x to A. We have two typing judgments, one for patterns $\theta \vdash_{\mathsf{p}} p : A$ and one for terms $\Gamma \vdash s : A$. Accordingly, we have two sets of typing rules: Fig. 3, top and bottom. We write $\rhd\, \theta \vdash_{\mathsf{p}} p : A$ to indicate that the typing judgment $\theta \vdash_{\mathsf{p}} p : A$ is derivable (likewise for $\rhd\, \Gamma \vdash s : A$). The typing schemes speak for themselves except for two of them which we now comment. The first is (T-APP). Note that we do not impose any additional restrictions on A_i, in particular it may be a union-type itself. This implies that the argument u can have a union type too. Regarding (T-ABS) it requests a number of conditions. First of all, each of the patterns p_i must be typable under the typing context θ_i, $i \in 1..n$. Also, the set of free matchables in each p_i must be exactly the domain of θ_i. Another condition, indicated by $(\Gamma, \theta_i \vdash s_i : B)_{i \in 1..n}$, is that the bodies of each of the branches s_i, $i \in 1..n$, must

$$\frac{}{\Sigma \vdash A \preceq_\mu A}\ (\text{S-REFL}) \qquad \frac{}{\Sigma, V \preceq_\mu W \vdash V \preceq_\mu W}\ (\text{S-HYP}) \qquad \frac{\vdash A \simeq_\mu B}{\Sigma \vdash A \preceq_\mu B}\ (\text{S-EQ})$$

$$\frac{\Sigma \vdash A \preceq_\mu B \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \preceq_\mu C}\ (\text{S-TRANS}) \qquad \frac{\Sigma \vdash D \preceq_\mu D' \quad \Sigma \vdash A \preceq_\mu A'}{\Sigma \vdash D \mathbin{@} A \preceq_\mu D' \mathbin{@} A'}\ (\text{S-COMP})$$

$$\frac{\Sigma \vdash A \preceq_\mu A' \quad \Sigma \vdash B \preceq_\mu B'}{\Sigma \vdash A' \supset B \preceq_\mu A \supset B'}\ (\text{S-FUNC}) \qquad \frac{\Sigma \vdash A \preceq_\mu C \quad \Sigma \vdash B \preceq_\mu C}{\Sigma \vdash A \oplus B \preceq_\mu C}\ (\text{S-UNION-L})$$

$$\frac{\Sigma \vdash A \preceq_\mu B}{\Sigma \vdash A \preceq_\mu B \oplus C}\ (\text{S-UNION-R1}) \qquad \frac{\Sigma \vdash A \preceq_\mu C}{\Sigma \vdash A \preceq_\mu B \oplus C}\ (\text{S-UNION-R2})$$

$$\frac{\Sigma, V \preceq_\mu W \vdash A \preceq_\mu B \quad W \notin \mathsf{fv}(A) \quad V \notin \mathsf{fv}(B)}{\Sigma \vdash \mu V.A \preceq_\mu \mu W.B}\ (\text{S-REC})$$

Figure 2 Strong subtyping for μ-types.

be typable under the context extended with the corresponding θ_i. More noteworthy is the condition that the list $[p_i : A_i]_{i \in 1..n}$ be *compatible*.

Compatibility is a condition that ensures that Subject Reduction is not violated. We briefly recall it; see [18] for further details and examples. As already mentioned in example (3) of the introduction, if p_i subsumes p_j (*i.e.* $p_i \lhd p_j$) with $i < j$, then the branch $p_j \to_{\theta_j} s_j$ will never be evaluated since the argument will already match p_i. Thus, in this case, in order to ensure SR we demand that $A_j \preceq_\mu A_i$. If p_i does not subsume p_j (*i.e.* $p_i \not\lhd p_j$) with $i < j$ we analyze the cause of failure of subsumption in order to determine whether requirements on A_i and A_j must be put forward, focusing on those cases where $\pi \in \mathsf{pos}(p_i) \cap \mathsf{pos}(p_j)$ is an offending position in both patterns. The following table exhaustively lists them:

	$p_i\|_\pi$	$p_j\|_\pi$	
(a)		y	restriction required
(b)	c	d	no overlapping ($p_j \not\lhd p_i$)
(c)		$q_1\, q_2$	no overlapping
(d)		y	restriction required
(e)	$q_1\, q_2$	d	no overlapping

In cases (b), (c) and (e), no extra condition on the types of p_i and p_j is necessary, since their respective sets of possible arguments are disjoint. The cases where A_i and A_j must be related are (a) and (d): for those we require $A_j \preceq_\mu A_i$. In summary, the cases requiring conditions on their types are: 1) $p_i \lhd p_j$; and 2) $p_i \not\lhd p_j$ and $p_j \lhd p_i$.

▶ **Definition 2.** Given a pattern $\theta \vdash_{\mathsf{p}} p : A$ and $\pi \in \mathsf{pos}(p)$, we say A *admits a symbol* $\odot$ *(with* $\odot \in \mathcal{V} \cup \mathcal{C} \cup \{\supset, @\}$*) at position* π iff $\odot \in A\|_\pi$, where:

$$\begin{array}{rcll} a\|_\epsilon & \triangleq & \{a\} & \\ (A_1 \star A_2)\|_\epsilon & \triangleq & \{\star\}\,, & \star \in \{\supset, @\} \\ (A_1 \star A_2)\|_{i\pi} & \triangleq & A_i\|_\pi, & \star \in \{\supset, @\}\,, i \in \{1,2\} \\ (A_1 \oplus A_2)\|_\pi & \triangleq & A_1\|_\pi \cup A_2\|_\pi & \\ (\mu V.A')\|_\pi & \triangleq & (\{\mu V.A'/V\}\, A')\|_\pi & \end{array}$$

Patterns

$$\frac{\theta(x) = A}{\theta \vdash_{\mathsf{p}} x : A}\ (\text{P-MATCH}) \qquad \frac{}{\theta \vdash_{\mathsf{p}} \mathsf{c} : \mathbb{c}}\ (\text{P-CONST}) \qquad \frac{\theta \vdash_{\mathsf{p}} p : D \quad \theta \vdash_{\mathsf{p}} q : A}{\theta \vdash_{\mathsf{p}} p\, q : D \,@\, A}\ (\text{P-COMP})$$

Terms

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{T-VAR}) \qquad \frac{}{\Gamma \vdash \mathsf{c} : \mathbb{c}}\ (\text{T-CONST}) \qquad \frac{\Gamma \vdash r : D \quad \Gamma \vdash u : A}{\Gamma \vdash r\, u : D \,@\, A}\ (\text{T-COMP})$$

$$\frac{\begin{array}{c}[p_i : A_i]_{i \in 1..n} \text{ compatible} \\ (\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i \in 1..n} \quad (\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i))_{i \in 1..n} \quad (\Gamma, \theta_i \vdash s_i : B)_{i \in 1..n}\end{array}}{\Gamma \vdash (p_i \to_{\theta_i} s_i)_{i \in 1..n} : (\bigoplus_{i \in 1..n} A_i) \supset B}\ (\text{T-ABS})$$

$$\frac{\Gamma \vdash r : \bigoplus_{i \in 1..n} A_i \supset B \quad \Gamma \vdash u : A_k \quad k \in 1..n}{\Gamma \vdash r\, u : B}\ (\text{T-APP}) \qquad \frac{\Gamma \vdash s : A \quad \vdash A \preceq_\mu A'}{\Gamma \vdash s : A'}\ (\text{T-SUBS})$$

Figure 3 Typing rules for patterns and terms.

Note that $\rhd\, \theta \vdash_{\mathsf{p}} p : A$ and contractiveness of A imply $A\|_\pi$ is well-defined for $\pi \in \mathsf{pos}(p)$.

▶ **Definition 3.** The *maximal positions* in a set of positions P are:

$$\mathsf{maxpos}(P) \triangleq \{\pi \in P \mid \nexists \pi' \neq \epsilon . \pi\pi' \in P\}$$

The *mismatching positions* between two patterns are defined below where, recall from the introduction, $p|_\pi$ stands for the sub-pattern at position π of p:

$$\mathsf{mmpos}(p, q) \triangleq \{\pi \mid \pi \in \mathsf{maxpos}(\mathsf{pos}(p) \cap \mathsf{pos}(q)) \wedge p|_\pi \not\lhd q|_\pi\}$$

For instance, given patterns $\mathtt{nil}$ and $\mathtt{cons}\, x\, xs$ with set of positions $\{\epsilon\}$ and $\{\epsilon, 1, 2, 11, 12\}$ respectively, we have $\mathsf{maxpos}(\mathtt{nil}) = \{\epsilon\}$ and $\mathsf{maxpos}(\mathtt{cons}\, x\, xs) = \{11, 12\}$, while the only mismatching position among them is the root, *i.e.* $\mathsf{mmpos}(\mathtt{nil}, \mathtt{cons}\, x\, xs) = \{\epsilon\}$.

▶ **Definition 4.** Define the *compatibility predicate* as

$$\mathcal{P}_{\mathsf{comp}}(p : A, q : B) \triangleq \forall \pi \in \mathsf{mmpos}(p, q)\,.A\|_\pi \cap B\|_\pi \neq \varnothing$$

We say $p : A$ is *compatible* with $q : B$, notation $p : A \lll q : B$, iff

$$\mathcal{P}_{\mathsf{comp}}(p : A, q : B) \implies B \preceq_\mu A$$

A list of patterns $[p_i : A_i]_{i \in 1..n}$ is compatible if $\forall i, j \in 1..n . i < j \implies p_i : A_i \lll p_j : A_j$.

Following the example, consider types $\mathbb{nil}$ and $\mathbb{cons} \,@\, \mathsf{Nat} \,@\, (\mu\alpha.\mathbb{nil} \oplus \mathbb{cons} \,@\, \mathsf{Nat} \,@\, \alpha)$ for patterns $\mathtt{nil}$ and $\mathtt{cons}\, x\, xs$ respectively. Compatibility requires no further restriction in this case since $\mathsf{mmpos}(\mathtt{nil}, \mathtt{cons}\, x\, xs) = \{\epsilon\}$ and

$$\mathbb{nil}\|_\epsilon = \{\mathbb{nil}\} \qquad (\mathbb{cons} \,@\, \mathsf{Nat} \,@\, (\mu\alpha.\mathbb{nil} \oplus \mathbb{cons} \,@\, \mathsf{Nat} \,@\, \alpha))\|_\epsilon = \{@\}$$

hence $\mathcal{P}_{\mathsf{comp}}$ is false and the property gets validated trivially.

On the contrary, recall example (3) on Sec. 1. $\mathtt{vl}\,x : \mathsf{vl} \mathbin{@} \mathsf{Bool} \lll \mathtt{vl}\,y : \mathsf{vl} \mathbin{@} \mathsf{Nat}$ requires $\mathsf{vl} \mathbin{@} \mathsf{Nat} \preceq_{\mu} \mathsf{vl} \mathbin{@} \mathsf{Bool}$ since $\mathsf{mmpos}(\mathtt{vl}\,x, \mathtt{vl}\,y) = \varnothing$ (*i.e.* $\mathcal{P}_{\mathsf{comp}}$ is trivially true). This actually fails because $\mathsf{Nat} \not\preceq_{\mu} \mathsf{Bool}$. Thus, this pattern combination is rejected by rule (T-ABS).

Types are preserved along reduction. The proof relies crucially on compatibility.

▶ **Proposition 5** (Subject Reduction). *If $\triangleright\, \Gamma \vdash s : A$ and $s \to s'$, then $\triangleright\, \Gamma \vdash s' : A$.*

Let the set of **values** be defined as $v ::= x\, v_1 \dots v_n \mid \mathsf{c}\, v_1 \dots v_n \mid (p_i \to_{\theta_i} s_i)_{i \in 1..n}$. The following property guarantees that no functional application gets stuck. Essentially this means that, in a well-typed closed term, a function which is applied to an argument has at least one branch that is capable of handling it.

▶ **Proposition 6** (Progress). *If $\triangleright\; \vdash s : A$ and s is not a value, then $\exists s'$ such that $s \to s'$.*

3 Checking Equivalence and Subtyping

As mentioned in the related work, there are roughly two approaches to implementing equivalence and subtype checking in the presence of recursive types, one based on automata theory and another based on coinductive characterizations of the associated relations. The former leads to efficient algorithms [15] while the latter is more abstract in nature and hence closer to the formalism itself although they may not be as efficient. In the particular case of subtyping for recursive types in the presence of ACI operators, the automata approach of [15] is known not to be applicable [20] while the coinductive approach, developed in this section, yields a correct algorithm. In Sec. 5 we explore an alternative approach for subtyping based on automata inspired from [6]. We next further describe the reasoning behind the coinductive approach.

Preliminaries. Coinductive characterizations of subsets of $\mathcal{T} \times \mathcal{T}$ whose generating function Φ is *invertible* admit a simple (although not necessarily efficient) algorithm for subtype membership checking and consists in "running Φ backwards" [16, Sec. 21.5]. This strategy is supported by the fact that contractiveness of μ-types guarantees a finite state space to explore (*i.e.* unfolding these types results in regular trees); invertibility further guarantees that there is at most one way in which a member of $\nu\Phi$, the greatest fixed-point of Φ, can be generated. Invertibility of $\Phi : \wp(\mathcal{T} \times \mathcal{T}) \to \wp(\mathcal{T} \times \mathcal{T})$ means that for any $\langle A, B\rangle \in \mathcal{T}$, the set $\{\mathcal{X} \in \wp(\mathcal{T} \times \mathcal{T}) \mid \langle A, B\rangle \in \Phi(\mathcal{X})\}$ is either empty or contains a unique member.

3.1 Equivalence Checking

Fig. 4 presents a coinductive definition of type equality over μ-types. This relation $\simeq_{\vec{\mu}}$ is defined by means of rules that are interpreted coinductively (indicated by the double lines). The rule (E-UNION-AL) makes use of functions f and g to encode the ACI nature of $\oplus$. Letters $\mathcal{C}, \mathcal{D}$, used in rules (E-REC-L-AL) and (E-REC-R-AL), denote contexts of the form:

$$A_1 \oplus \dots A_{i-1} \oplus \Box \oplus A_{i+1} \oplus \dots \oplus A_n$$

where $\Box$ denotes the hole of the context, $A_j \neq \oplus$ for all $j \in 1..n \setminus i$ and $A_l \neq \mu$ for all $l \in 1..i-1$. Note that, in particular, $\mathcal{C}$ may take the form $\Box$. These contexts help identify the first occurrence of a μ constructor within a maximal union. In turn, this allows us to guarantee the invertibility of the generating function associated to these rules.

$$\frac{}{a \simeq_{\vec{\mu}} a} \text{ (E-REFL-AL)}$$

$$\frac{D \simeq_{\vec{\mu}} D' \quad A \simeq_{\vec{\mu}} A'}{D \mathbin{@} A \simeq_{\vec{\mu}} D' \mathbin{@} A'} \text{ (E-COMP-AL)} \qquad \frac{A \simeq_{\vec{\mu}} A' \quad B \simeq_{\vec{\mu}} B'}{A \supset B \simeq_{\vec{\mu}} A' \supset B'} \text{ (E-FUNC-AL)}$$

$$\frac{\mathcal{C}[\{\mu V.A/V\}\, A] \simeq_{\vec{\mu}} B}{\mathcal{C}[\mu V.A] \simeq_{\vec{\mu}} B} \text{ (E-REC-L-AL)}$$

$$\frac{A \simeq_{\vec{\mu}} \mathcal{D}[\{\mu W.B/W\}\, B] \quad A \neq \mathcal{C}[\mu V.C]}{A \simeq_{\vec{\mu}} \mathcal{D}[\mu W.B]} \text{ (E-REC-R-AL)}$$

$$\frac{\begin{array}{ll} A_i \simeq_{\vec{\mu}} B_{f(i)} & f : 1..n \to 1..m \\ A_{g(j)} \simeq_{\vec{\mu}} B_j & g : 1..m \to 1..n \end{array} \quad A_i, B_j \neq \mu, \oplus \quad n+m > 2}{\bigoplus_{i \in 1..n} A_i \simeq_{\vec{\mu}} \bigoplus_{j \in 1..m} B_j} \text{ (E-UNION-AL)}$$

Figure 4 Coinductive axiomatization of type equality for contractive μ-types.

▶ **Proposition 7.** *The generating function associated with the rules of Fig. 4 is invertible.*

Moreover, $\simeq_{\vec{\mu}}$ coincides with $\simeq_{\mu}$:

▶ **Proposition 8.** $A \simeq_{\vec{\mu}} B$ *iff* $A \simeq_{\mu} B$*.*

The proof of Prop. 8 relies on an intermediate relation $\simeq_{\mathfrak{T}}$ over the possibly infinite trees resulting from the complete unfolding of μ-types. This relation is defined using the same rules as in Fig. 4 except for two important differences: 1) the relation is defined over regular trees in $\mathfrak{T}$, and 2) rules (E-REC-L-AL) and (E-REC-R-AL) are dropped.

Thus we can resort to invertibility of the generating function to check for $\simeq_{\vec{\mu}}$. Fig. 5 presents the algorithm. It uses `seq` $e_1 \ldots e_n$ which sequentially evaluates each of its arguments, returning the value of the first of these that does not fail. Evaluation of $\mathsf{eqtype}(\emptyset, A, B)$ can have one of two outcomes: `fail`, meaning that $A \not\simeq_{\vec{\mu}} B$, or a set $S \in \wp(\mathcal{T} \times \mathcal{T})$ that is Φ-dense with $(A, B) \in S$, proving that $A \simeq_{\vec{\mu}} B$.

3.2 Subtype Checking

The approach to subtype checking is similar to that of type equivalence. First consider the relation $\preceq_{\vec{\mu}}$ over μ-types defined in Fig. 6. It captures $\preceq_{\mu}$:

▶ **Proposition 9.** $A \preceq_{\vec{\mu}} B$ *iff* $A \preceq_{\mu} B$*.*

The proof strategy is similar to that of Prop. 8. In this case we resort to a proper subtyping relation for infinite trees that essentially results from dropping rules (S-REC-L-AL) and (S-REC-R-AL) in Fig. 6.

Unfortunately, the generating function determined by the rules in Fig. 6, let us call it $\Phi_{\preceq_{\vec{\mu}}}$, is not invertible. Notice that (S-UNION-R-AL) overlaps with itself. For example, $\mathbb{c} \preceq_{\vec{\mu}} (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})$ belongs to two $\Phi_{\preceq_{\vec{\mu}}}$-saturated sets (*i.e.* sets $\mathcal{X}$ such that $\mathcal{X} \subseteq \Phi_{\preceq_{\vec{\mu}}}(\mathcal{X})$):

$$\begin{array}{lcl} \mathcal{X}_1 & = & \{\langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d})\rangle, \langle \mathbb{c}, \mathbb{c}\rangle\} \\ \mathcal{X}_2 & = & \{\langle \mathbb{c}, (\mathbb{c} \oplus \mathbb{d}) \oplus (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, (\mathbb{e} \oplus \mathbb{c})\rangle, \langle \mathbb{c}, \mathbb{c}\rangle\} \end{array}$$

```
eqtype(S, A, B) ≜
    if ⟨A, B⟩ ∈ S
      then S
      else let S₀ = S ∪ {⟨A, B⟩} in
        case ⟨A, B⟩ of
          ⟨a, a⟩ →
            S₀
          ⟨A' @ A'', B' @ B''⟩ →
            if A', B' are datatypes
              then let S₁ = eqtype(S₀, A', B') in
                eqtype(S₁, A'', B'')
              else fail
          ⟨A' ⊃ A'', B' ⊃ B''⟩ →
            let S₁ = eqtype(S₀, A', B') in
              eqtype(S₁, A'', B'')
          ⟨C[μV.A'], B⟩ →
            eqtype(S₀, C[{μV.A'/V} A'], B)
          ⟨A, D[μW.B']⟩ →
            eqtype(S₀, A, D[{μW.B'/W} B'])
          ⟨⊕_{i∈1..n} A_i, ⊕_{j∈1..m} B_j⟩ →
            let S₁ = (seq eqtype(S₀, A₁, B₁), ..., eqtype(S₀, A₁, B_m)) in
            ...
            let S_n = (seq eqtype(S_{n−1}, A_n, B₁), ..., eqtype(S_{n−1}, A_n, B_m)) in
            let S_{n+1} = (seq eqtype(S_n, A₁, B₁), ..., eqtype(S_n, A_n, B₁)) in
            ...
            let S_{n+m−1} = (seq eqtype(S_{n+m−2}, A₁, B_{m−1}), ..., eqtype(S_{n+m−2}, A_n, B_{m−1}))
              in seq eqtype(S_{n+m−1}, A₁, B_m), ..., eqtype(S_{n+m−1}, A_n, B_m)
          otherwise →
            fail
```

Figure 5 Equivalence checking algorithm.

$$\frac{}{a \preceq_{\vec{\mu}} a}\ (\text{S-REFL-AL})$$

$$\frac{D \preceq_{\vec{\mu}} D' \quad A \preceq_{\vec{\mu}} A'}{D \mathbin{@} A \preceq_{\vec{\mu}} D' \mathbin{@} A'}\ (\text{S-COMP-AL}) \qquad \frac{A' \preceq_{\vec{\mu}} A \quad B \preceq_{\vec{\mu}} B'}{A \supset B \preceq_{\vec{\mu}} A' \supset B'}\ (\text{S-FUNC-AL})$$

$$\frac{\{\mu V.A/V\}\, A \preceq_{\vec{\mu}} B}{\mu V.A \preceq_{\vec{\mu}} B}\ (\text{S-REC-L-AL}) \qquad \frac{A \preceq_{\vec{\mu}} \{\mu W.B/W\}\, B \quad A \neq \mu}{A \preceq_{\vec{\mu}} \mu W.B}\ (\text{S-REC-R-AL})$$

$$\frac{A_i \preceq_{\vec{\mu}} B \text{ for all } i \in 1..n \quad n > 1 \quad B \neq \mu \quad A_i \neq \oplus}{\bigoplus_{i\in 1..n} A_i \preceq_{\vec{\mu}} B}\ (\text{S-UNION-L-AL})$$

$$\frac{A \preceq_{\vec{\mu}} B_k \text{ for some } k \in 1..m \quad m > 1 \quad A \neq \mu, \oplus \quad B_j \neq \oplus}{A \preceq_{\vec{\mu}} \bigoplus_{j\in 1..m} B_j}\ (\text{S-UNION-R-AL})$$

Figure 6 Coinductive axiomatization of subtyping for contractive μ-types.

```
subtype(S, A, B) ≜
    if ⟨A, B⟩ ∈ S
      then S
      else let S_0 = S ∪ {⟨A, B⟩} in
        case ⟨A, B⟩ of
          ⟨a, a⟩ →
            S_0
          ⟨A′ @ A″, B′ @ B″⟩ →
            if A′, B′ are datatypes
              then let S_1 = subtype(S_0, A′, A″) in
                subtype(S_1, B′, B″)
              else fail
          ⟨A′ ⊃ A″, B′ ⊃ B″⟩ →
            let S_1 = subtype(S_0, B′, A′) in
              subtype(S_1, A″, B″)
          ⟨μV.A′, B⟩ →
            subtype(S_0, {μV.A′/V} A′, B)
          ⟨A, μW.B′⟩ →
            subtype(S_0, A, {μW.B′/W} B′)
          ⟨⊕_{i∈1..n} A_i, B⟩ →
            let S_1 = subtype(S_0, A_1, B) in
            let S_2 = subtype(S_1, A_2, B) in
            ...
            let S_{n−1} = subtype(S_{n−2}, A_{n−1}, B) in
              subtype(S_{n−1}, A_n, B)
          ⟨A, ⊕_{j∈1..m} B_j⟩ →
            seq subtype(S_0, A, B_1), ..., subtype(S_0, A, B_m)
          otherwise →
            fail
```

Figure 7 Subtype checking algorithm.

However, since this is the only source of non-invertibility we easily derive a subtype membership checking function $\mathsf{subtype}(\bullet, \bullet, \bullet)$ that, in the case of (S-UNION-R-AL), simply checks all cases (Fig. 7).

4 Type Checking

A syntax-directed presentation for typing in CAP, inferring judgments of the form $\Gamma \mapsto s : A$, may be obtained from the rules of Fig. 3 by dropping subsumption. This requires "hard-wiring" it back in into (T-APP). Unfortunately, the naïve syntax-directed variant:

$$\frac{\Gamma \mapsto r : (\bigoplus_{i\in 1..n} A_i) \supset B \quad \Gamma \mapsto u : A' \quad A' \preceq_\mu A_k, \text{ for some } k \in 1..n}{\Gamma \mapsto r\, u : B} \ (\text{T-APP-AL})'$$

fails to capture all the required terms. In other words, there are Γ, s and A such that $\Gamma \vdash s : A$ but no $A' \preceq_\mu A$ such that $\Gamma \mapsto s : A'$. For example, take $\Gamma(x) \triangleq (\mathbb{c} \oplus \mathbb{e} \supset \mathbb{d}) \oplus (\mathbb{c} \oplus \mathbb{f} \supset \mathbb{d})$, $s \triangleq x\, \mathsf{c}$ and $A \triangleq \mathbb{d}$. More generally, from $\Gamma \mapsto r : A$ and $A \preceq_\mu \bigoplus_{i\in 1..n} A_i \supset B$ we cannot infer

$$\frac{\Gamma(x) = A}{\Gamma \mapsto x : A}\ (\text{T-VAR-AL}) \qquad \frac{}{\Gamma \mapsto \mathsf{c} : \mathbb{c}}\ (\text{T-CONST-AL}) \qquad \frac{\Gamma \mapsto r : D \quad \Gamma \mapsto u : A}{\Gamma \mapsto r\,u : D \,@\, A}\ (\text{T-COMP-AL})$$

$$\frac{\begin{array}{c}[p_i : A_i]_{i\in 1..n} \text{ compatible} \\ (\theta_i \vdash_{\mathsf{p}} p_i : A_i)_{i\in 1..n} \quad (\mathsf{dom}\,(\theta_i) = \mathsf{fm}(p_i))_{i\in 1..n} \quad (\Gamma, \theta_i \mapsto s_i : B_i)_{i\in 1..n}\end{array}}{\Gamma \mapsto (p_i \to_{\theta_i} s_i)_{i\in 1..n} : \bigoplus_{i\in 1..n} A_i \supset \bigoplus_{i\in 1..n} B_i}\ (\text{T-ABS-AL})$$

$$\frac{\begin{array}{l}\Gamma \mapsto r : A \quad A \simeq_\mu \bigoplus_{i\in 1..n}(A_i \supset B_i) \quad A_i \neq \oplus \\ \Gamma \mapsto u : C \quad (\vdash C \preceq_\mu A_i)_{i\in 1..n}\end{array}}{\Gamma \mapsto r\,u : \bigoplus_{i\in 1..n} B_i}\ (\text{T-APP-AL})$$

Figure 8 Syntax-directed typing rules for terms.

that A is a functional type due to the presence of union types. A complete (Prop. 10) syntax directed presentation is obtained by dropping (T-SUBS) from Fig. 3 and replacing (T-ABS) and (T-APP) by (T-ABS-AL) and (T-APP-AL), resp., of Fig. 8.

▶ **Proposition 10.**
1. *If* $\Gamma \mapsto s : A$*, then* $\Gamma \vdash s : A$*.*
2. *If* $\Gamma \vdash s : A$*, then* $\exists A'$ *such that* $A' \preceq_\mu A$ *and* $\Gamma \mapsto s : A'$*.*

From this we may obtain a simple type-checking function $\mathsf{tc}(\Gamma, s)$ (Fig. 9-top) such that $\mathsf{tc}(\Gamma, s) = A$ iff $\Gamma \mapsto s : A'$, for some $A' \simeq_\mu A$. The interesting clause is that of application, where the decision of whether (T-COMP-AL) or (T-APP-AL) may be applied depends on the result of the recursive call. If the term r is assigned a datatype, then a new compound datatype is built; if its type can be rewritten as a union of functional types, then a proper type is constructed with each of the co-domains of the latter, as established in rule (T-APP-AL). The expression $\mathsf{unfold}(A)$, in the clause defining $\mathsf{tc}(\Gamma, r\,u)$, is the result of unfolding type A using rules (E-REC-L-AL) and (E-REC-R-AL) until the result is an equivalent type $A' = \oplus_{i\in 1..n} A'_i$ with $A'_i \neq \mu, \oplus$, and then simply verifying that $A'_i = \supset$ for all $i \in 1..n$.

$$\begin{array}{lll}\mathsf{unfold}(A) & \triangleq & \texttt{if } A = A' \supset A'' \texttt{ then } A \\ & & \texttt{else if } A = \oplus_{i\in 1..n} A_i \texttt{ and } n > 1 \texttt{ and } A_i \neq \oplus \texttt{ then} \\ & & \quad \texttt{let } \oplus_{j\in 1..m_i}(A_{ij} \supset B_{ij}) = \mathsf{unfold}(A_i) \texttt{ foreach } i \in 1..n \texttt{ in} \\ & & \qquad \bigoplus_{\substack{i\in 1..n \\ j\in 1..m_i}} (A_{ij} \supset B_{ij}) \\ & & \texttt{else if } A = \mu V.A' \texttt{ then } \mathsf{unfold}(\{\mu V.A/V\}\,A) \\ & & \texttt{else fail}\end{array}$$

Termination is guaranteed by contractiveness of μ-types. In the worst case it requires exponential time due to the need to unfold types until the desired equivalent form is obtained (*e.g.* $\mu X_1 . \ldots \mu X_n . X_1 \supset \ldots X_n \supset \mathbb{c}$).

Compatibility between branches is verified by checking if $\mathcal{P}_{\mathsf{comp}}(p : A, q : B)$ holds:

$$\mathsf{compatible}(p : A, q : B) \quad \triangleq \quad (\texttt{not } \mathsf{pcomp}(p : A, q : B)) \texttt{ or } \mathsf{subtype}(\varnothing, B, A)$$

In pcomp we may assume that it has already been checked that p has type A and q has type B. Therefore, if these are compound patterns they can only be assigned application types, and union types may only appear at leaf positions of a pattern. We use this correspondence

$$
\begin{array}{rcl}
\mathsf{tc}(\Gamma, x) & \triangleq & \Gamma(x) \\
\mathsf{tc}(\Gamma, \mathsf{c}) & \triangleq & \mathbb{c} \\
\mathsf{tc}(\Gamma, (p_i \to_{\theta_i} s_i)_{i \in 1..n}) & \triangleq & \texttt{let } A_i = \mathsf{tcp}(\theta_i, p_i), B_i = \mathsf{tc}(\Gamma, \theta_i, s_i) \texttt{ in} \\
& & \quad \texttt{if } \forall i \in 1..n. \forall j \in i+1..n. \mathsf{compatible}(p_i : A_i, p_j : A_j) \\
& & \quad\quad \texttt{then } \bigoplus_{i \in 1..n} A_i \supset \bigoplus_{i \in 1..n} B_i \\
& & \quad\quad \texttt{else fail} \\
\mathsf{tc}(\Gamma, r\, u) & \triangleq & \texttt{let } A = \mathsf{tc}(\Gamma, r), C = \mathsf{tc}(\Gamma, u) \texttt{ in} \\
& & \quad \texttt{if } A \text{ is a datatype} \\
& & \quad\quad \texttt{then } A @ C \\
& & \quad\quad \texttt{else let } \bigoplus_{i \in 1..n} (A_i \supset B_i) = \mathsf{unfold}(A) \texttt{ in} \\
& & \quad\quad\quad \texttt{if } \forall i \in 1..n. \mathsf{subtype}(\varnothing, C, A_i) \\
& & \quad\quad\quad \texttt{then } \bigoplus_{i \in 1..n} B_i \\
& & \quad\quad\quad \texttt{else fail}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{tcp}(\Gamma, x) & \triangleq & \Gamma(x) \\
\mathsf{tcp}(\Gamma, \mathsf{c}) & \triangleq & \mathbb{c} \\
\mathsf{tcp}(\Gamma, p\, q) & \triangleq & \texttt{let } A = \mathsf{tcp}(\Gamma, p), B = \mathsf{tcp}(\Gamma, q) \texttt{ in} \\
& & \quad \texttt{if } A \text{ is a datatype} \\
& & \quad\quad \texttt{then } A @ B \\
& & \quad\quad \texttt{else fail}
\end{array}
$$

Figure 9 Type-checking CAP.

to traverse both pattern and type simultaneously in linear time, which means the worst-case execution time of the compatibility check is governed by the complexity of subtyping.

$$
\begin{array}{rcl}
\mathsf{pcomp}(p : A, q : B) & \triangleq & \texttt{if } p = p_1\, p_2 \textsf{ and } q = q_1\, q_2 \texttt{ then} \\
& & \quad \texttt{let } A = A_1 @ A_2, B = B_1 @ B_2 \texttt{ in} \\
& & \quad\quad \mathsf{pcomp}(p_1 : A_1, q_1 : B_1) \textsf{ and } \mathsf{pcomp}(p_2 : A_2, q_2 : B_2) \\
& & \texttt{else} \\
& & \quad\quad (p = x) \textsf{ or } (p = q = \mathsf{c}) \textsf{ or } (A\|_\epsilon \cap B\|_\epsilon \neq \varnothing)
\end{array}
$$

5 Towards Efficient Type-Checking

The algorithms presented so far are clear but inefficient. The number of recursive calls in eqtype and subtype is not bounded (it depends on the size of the type) and unfolding recursive types may increment their size exponentially. This section draws from ideas in [6, 12, 15] and adopts a dag-representation of recursive types which are encoded as *term automata* (described below). Associativity is handled by resorting to n-ary unions, commutativity and idempotence of $\oplus$ is handled by how types are decomposed in their automaton representation (*cf.* check in Fig. 13). The algorithm itself is tc of Fig. 9 except that:

1. The representation of μ-types are now term automata. This renders unfold linear.
2. The subtyping algorithm is optimized, based on the new representation and following ideas from [6, 15].

The end product is an algorithm with complexity $\mathcal{O}(n^7 d)$ where n is the size of the input (*i.e.* that of Γ plus t) and d is the maximum arity of the n-unions occurring in Γ and t. Note that all the information needed to type t is either in the context or annotated within the

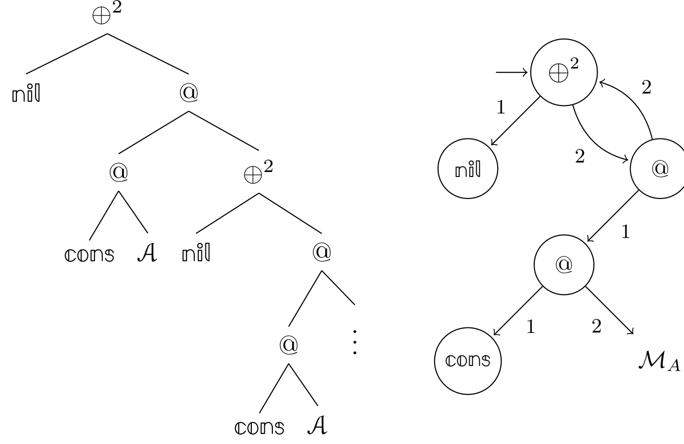


Figure 10 The type List_A represented as an infinite tree and as a term automaton.

term itself. Thus, a linear relation can be established between the size of the input and the size of the resulting type; and we can think of n as the size of the latter.

5.1 Term Automata

μ-types may be understood as finite dags since their infinite unfolding produce a regular (infinite) trees. We further simplify the types whose dags we consider by flattening the union type constructor and switching to an alphabet where unions are n-ary: $\mathfrak{L}^{\mathfrak{n}} \triangleq \{a^0 \mid a \in \mathcal{V} \cup \mathcal{C}\} \cup \{@^2, \supset^2\} \cup \{\oplus^n \mid n > 1\}$ and we let $\mathfrak{T}^{\mathfrak{n}}$ stand for possibly infinite trees whose nodes are symbols in $\mathfrak{L}^{\mathfrak{n}}$. μ-types may be interpreted in $\mathfrak{T}^{\mathfrak{n}}$ simply by unfolding and then considering maximal union types as their underlying n-ary union types. We write $[\![\bullet]\!]^{\mathfrak{n}}$ for this function and use meta-variables $\mathcal{A}, \mathcal{B}, \ldots$ when referring to elements of $\mathfrak{T}^{\mathfrak{n}}$. Types in $\mathfrak{T}^{\mathfrak{n}}$ may be represented as *term automata* [1].

▶ **Definition 11.** A **term automaton** is a tuple $\mathcal{M} = \langle Q, \Sigma, q_0, \delta, \ell \rangle$ where:

1. Q is a finite set of states.
2. Σ is an alphabet where each symbol has an associated arity.
3. q_0 is the initial state.
4. $\delta : Q \times \mathbb{N} \to Q$ is a partial transition function between states, defined over $1..k$, where k is the arity of the symbol associated by ℓ to the origin state.
5. $\ell : Q \to \Sigma$ is a total function that associates a symbol in Σ to each state.

We write $\mathcal{M}_A$ for the automaton associated to type A. $\mathcal{M}_A$ recognizes all paths from the root of A to any of its sub-expressions. Fig. 10 illustrates an example type, namely $\mathsf{List}_A = \mu\alpha.\mathsf{nil} \oplus (\mathsf{cons} \mathbin{@} A \mathbin{@} \alpha)$, represented as an infinite tree and as a term automaton $\mathcal{M}_{\mathsf{List}_A}$. If q_0 is the initial state of $\mathcal{M}_{\mathsf{List}_A}$ and $\widehat{\delta}$ denotes the natural extension of δ to sequences of symbols, then $\ell(\widehat{\delta}(q_0, 211)) = \mathsf{cons}$. As mentioned, the regular structure of trees arising from types yields automata with a finite number of states.

5.2 Subtyping and Subtype Checking

We next present a coinductive notion of subtyping over $\mathfrak{T}^{\mathfrak{n}}$. It is a binary relation $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ *up-to* a set of hypothesis $\mathcal{R}$ (Fig. 11). For $\mathcal{R} = \varnothing$, $\preceq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$ coincides with $\preceq_\mu$, modulo application of our translation $[\![\bullet]\!]^{\mathfrak{n}}$.

$$\frac{}{a \preceq^{\mathcal{R}}_{\mathfrak{T}^n} a} \text{(S-REFL-UP)}$$

$$\frac{\mathcal{D} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{D}' \quad \mathcal{A} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{A}'}{\mathcal{D} \ @ \ \mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{D}' \ @ \ \mathcal{A}'} \text{(S-COMP-UP)}$$

$$\frac{\mathcal{A}' \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{A} \quad \mathcal{B} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \preceq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{A}' \supset \mathcal{B}'} \text{(S-FUNC-UP)}$$

$$\frac{\mathcal{A}_i \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{B}_{f(i)} \quad f : 1..n \to 1..m \quad \mathcal{A}_i, \mathcal{B}_j \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \preceq^{\mathcal{R}}_{\mathfrak{T}^n} \bigoplus_j^m \mathcal{B}_j} \text{(S-UNION-UP)}$$

$$\frac{\mathcal{A}_i \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{B} \text{ for all } i \in 1..n \quad \mathcal{A}_i \neq \oplus \quad \mathcal{B} \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \preceq^{\mathcal{R}}_{\mathfrak{T}^n} \mathcal{B}} \text{(S-UNION-L-UP)}$$

$$\frac{\mathcal{A} \ (\preceq^{\mathcal{R}}_{\mathfrak{T}^n} \cup \mathcal{R}) \ \mathcal{B}_k \text{ for some } k \in 1..m \quad \mathcal{A} \neq \oplus \quad \mathcal{B}_j \neq \oplus}{\mathcal{A} \preceq^{\mathcal{R}}_{\mathfrak{T}^n} \bigoplus_j^m \mathcal{B}_j} \text{(S-UNION-R-UP)}$$

Figure 11 Subtyping relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^n$.

▶ **Proposition 12.** *$A \preceq_\mu B$ iff $[\![A]\!]^n \preceq^{\varnothing}_{\mathfrak{T}^n} [\![B]\!]^n$.*

So we can use $\preceq^{\varnothing}_{\mathfrak{T}^n}$ to determine whether types are related via $\preceq_\mu$: take two types, construct their automaton representation and check whether these are related via $\preceq^{\varnothing}_{\mathfrak{T}^n}$. Moreover, our formulation of $\preceq^{\mathcal{R}}_{\mathfrak{T}^n}$ will prove convenient for proving correctness of our subtyping algorithm.

5.2.1 Algorithm Description

The algorithm that checks whether types are related by the new subtyping relation builds on ideas from [6]. Our presentation is more general than required for subtyping; this general scheme will also be applicable to type equivalence, as we shall later see. Call $p \in \mathfrak{T}^n \times \mathfrak{T}^n$ *valid* if $p \in \preceq^{\varnothing}_{\mathfrak{T}^n}$. The algorithm consists of two phases. The aim of the first one is to construct a set $U \subseteq \mathfrak{T}^n \times \mathfrak{T}^n$ that delimits the universe of pairs of types that will later be refined to obtain a set of only valid pairs. It starts off with an initial pair (*cf.* Fig. 12, buildUniverse) and then explores pairs of sub-terms of both types in this pair by decomposing the type constructors (*cf.* Fig. 12, children). Note that, given p, the algorithm may add invalid pairs in order to prove the validity of p. The second phase shall be in charge of eliminating these invalid pairs. Note that the first phase can easily be adapted to other relations by simply redefining function children.

U may be interpreted as a directed graph where an edge from pair p to q means that q might belong to the support set of p in the final relation $\preceq^{\varnothing}_{\mathfrak{T}^n}$. In this case we say that p is a **parent** of q. Since types could have cycles, a pair may be added to U more than once and hence have more than one parent. Set $u(p)$ to be the **incoming degree** of p, *i.e.* the number of parents.

During the second phase (Fig. 13, gfp) the following sets are maintained, all of which conform a partition of U:

```
buildUniverse(p_0) :
    U = ∅
    W = {p_0}
    while W ≠ ∅ :
      p := takeOne(W)
      if p ∉ U
        insert(p, U)
        foreach q ∈ children(p)
          insert(q, W)
    return U
```

```
children(p) :
  case p of
    ⟨D @ A, D' @ B⟩ →
        {⟨D, D'⟩, ⟨A, B⟩}
    ⟨A' ⊃ A'', B' ⊃ B''⟩ →
        {⟨B', A'⟩, ⟨A'', B''⟩}
    ⟨⊕_i^n A_i, ⊕_j^m B_j⟩ →
        {⟨A_i, B_j⟩ | i ∈ 1..n, j ∈ 1..m}
    ⟨⊕_i^n A_i, B⟩, B ≠ ⊕ →
        {⟨A_i, B⟩ | i ∈ 1..n}
    ⟨A, ⊕_j^m B_j⟩, A ≠ ⊕ →
        {⟨A, B_j⟩ | j ∈ 1..m}
    otherwise →
        ∅
```

Figure 12 Pseudo-code of the first phase of the algorithm (construction of the universe U).

- W: pairs whose validity has yet to be determined
- S: pairs considered conditionally valid
- F: invalid pairs

The algorithm repeatedly takes elements in W and, in each iteration, transfers to S the selected pair p if its validity can be proved assuming valid only those pairs which have not been discarded up until now (*i.e.* those in $W \cup S$). Otherwise, p is transferred to F and all of its parents in S need to be reconsidered (their validity up-to W may have changed). Thus these parents are moved back to W (*cf.* Fig. 13, invalidate). Intuitively, S contains elements in $\preceq^{W}_{\mathfrak{T}^n}$. The process ends when W is empty. The only aspect of this second phase specific to $\preceq^{W}_{\mathfrak{T}^n}$ is function check, which may be redefined to be other suitable *up-to* relations.

5.2.2 Correctness

It is based on the fact that S may be considered a set of valid pairs *assuming the validity of those in* W. More generally, the following holds:

▶ **Proposition 13.** *The algorithm preserves the following invariant:*
- $\langle W, S, F\rangle$ *is a partition of* U
- F *is composed solely of invalid pairs*
- $S \subseteq \Phi_{\preceq^{W}_{\mathfrak{T}^n}}(S)$

When the main cycle ends we know that W is empty, and therefore that $S \subseteq \Phi_{\preceq^{\emptyset}_{\mathfrak{T}^n}}(S)$. The coinduction principle then yields $S \subseteq \preceq^{\emptyset}_{\mathfrak{T}^n}$ (*i.e.* every pair in S is valid) and therefore we are left to verify whether the original pair of types is in S or F.

5.2.3 Complexity

The first phase consists of identifying relevant pairs of sub-terms in both types being evaluated. If we call N and N' the size of such types (considering nodes and edges in their representations) we have that the size and cost of building the universe U can be bounded

```
gfp(𝒜, ℬ) :
    W = buildUniverse(⟨𝒜, ℬ⟩)
    S = ∅
    F = ∅
    while W ≠ ∅ :
      p := takeOne(W)
      if check(p, F)
        then insert(p,  S)
        else invalidate(p, S, F, W)
    return  p ∈ S

invalidate(p, S, F, W) :
    insert(p,  F)
    foreach q ∈ parents(p) ∩ S
      move(q,  S,  W)

check(p, F) :
    case  p  of
      ⟨a, a⟩ →
          true
      ⟨𝒟 @ 𝒜, 𝒟′ @ ℬ⟩ →
          ⟨𝒟, 𝒟′⟩ ∉ F and ⟨𝒜, ℬ⟩ ∉ F
      ⟨𝒜′ ⊃ 𝒜″, ℬ′ ⊃ ℬ″⟩ →
          ⟨ℬ′, 𝒜′⟩ ∉ F and ⟨𝒜″, ℬ″⟩ ∉ F
      ⟨⊕_i^n 𝒜_i, ⊕_j^m ℬ_j⟩ →
          ∀i.∃j. ⟨𝒜_i, ℬ_j⟩ ∉ F
      ⟨⊕_i^n 𝒜_i, ℬ⟩,  ℬ ≠ ⊕ →
          ∀i. ⟨𝒜_i, ℬ⟩ ∉ F
      ⟨𝒜, ⊕_j^m ℬ_j⟩,  𝒜 ≠ ⊕ →
          ∃m. ⟨𝒜, ℬ_m⟩ ∉ F
```

Figure 13 Pseudo-code of the second phase (relation refinement).

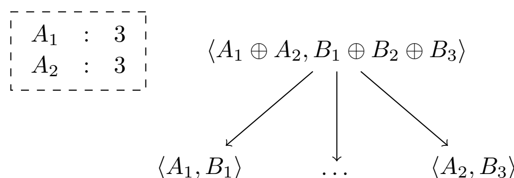


Figure 14 Verification of invalidated descendants.

by $\mathcal{O}(NN')$. As we shall see, the total cost of the algorithm is governed by the amount of operations in the second phase.

As stated in [6], since any pair can be invalidated at most once (in which case $u(p)$ nodes are transferred back to W for reconsideration) the amount of iterations in the refinement stage can be bounded by

$$\sum_{p \in U} 1 + \sum_{p \in U} u(p) \quad = \quad \sum_{p \in U}(1 + u(p)) \quad = \quad size(U)$$

Assuming that set operations can be performed in constant time, the cost of evaluating each node in the main loop is that of deciding whether to suspend or invalidate the pair and, in the later case, the cost of the invalidation process. The decision of where to transfer the node is computed in the function check, which always performs a constant amount of operations for pairs of non-union types. The worst case involves checking pairs of the form $\langle \bigoplus_i^n A_i, \bigoplus_j^m B_i \rangle$, which can be resolved by maintaining in each node a table indicating, for every A_i, the amount of pairs $\langle A_i, B_j \rangle$ that have not yet been invalidated. Using this approach, this check can be performed in $\mathcal{O}(d)$ operations, where d is a bound on the size of both unions. Whenever a pair is invalidated, all tables present in its immediate parents are updated as necessary.

Finally we resort to an argument introduced in [6] to argue that the cost of invalidating an element can be seen as $\mathcal{O}(1)$: a new iteration will be performed for each of the $u(p)$ pairs added to W when invalidating p. Because of this, a more precise bound for the cost of the

$$\dfrac{}{a \simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} a}\ (\text{E-REFL-UP})$$

$$\dfrac{\mathcal{D}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{D}' \quad \mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{A}'}{\mathcal{D} \mathbin{@} \mathcal{A} \simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{D}' \mathbin{@} \mathcal{A}'}\ (\text{E-COMP-UP})$$

$$\dfrac{\mathcal{A}'\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{A} \quad \mathcal{B}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{B}'}{\mathcal{A} \supset \mathcal{B} \simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{A}' \supset \mathcal{B}'}\ (\text{E-FUNC-UP})$$

$$\dfrac{\begin{array}{lll} \mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{B}_{f(i)} & f : 1..n \to 1..m & \\ \mathcal{A}_{g(j)}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{B}_j & g : 1..m \to 1..n & \mathcal{A}_i, \mathcal{B}_j \neq \oplus \end{array}}{\bigoplus_i^n \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \bigoplus_j^m \mathcal{B}_j}\ (\text{E-UNION-UP})$$

$$\dfrac{\mathcal{A}_i\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{B} \text{ for all } i \in 1..n \quad \mathcal{A}_i \neq \oplus \quad \mathcal{B} \neq \oplus}{\bigoplus_i^n \mathcal{A}_i \simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \mathcal{B}}\ (\text{E-UNION-L-UP})$$

$$\dfrac{\mathcal{A}\ (\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \cup \mathcal{R})\ \mathcal{B}_j \text{ for all } j \in 1..m \quad \mathcal{A} \neq \oplus \quad \mathcal{B}_j \neq \oplus}{\mathcal{A} \simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}} \bigoplus_j^m \mathcal{B}_j}\ (\text{E-UNION-R-UP})$$

Figure 15 Equivalence relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^{\mathfrak{n}}$.

complete execution of the algorithm can be obtained if we consider the cost of adding each of these elements to W as part of the iteration itself, yielding an amortized cost of $\mathcal{O}(d)$ operations for each iteration. This leaves a total cost of $\mathcal{O}(size(U)d)$ for the subtyping check, expressed as $\mathcal{O}(NN'd)$ in terms of the size of the input automata.

Let us call n and n' the amount of constructors in types A and B, respectively. N and N' are the sizes of automata representing these types, and can consequently be bounded by $\mathcal{O}(n^2)$ and $\mathcal{O}(n'^2)$. Therefore, the complexity of the algorithm can be expressed as $\mathcal{O}(n^2 n'^2 d)$.

5.3 Equivalence Checking

In this section we adapt the previous algorithm to obtain one proper of equivalence checking with the same complexity cost. Fig. 15 introduces an equivalence relation *up-to* $\mathcal{R}$ over $\mathfrak{T}^{\mathfrak{n}}$ which can be used to compute $\simeq_\mu$ via the translation $[\![\bullet]\!]^{\mathfrak{n}}$.

▶ **Lemma 14.** $A \simeq_\mu B$ *iff* $[\![A]\!]^{\mathfrak{n}} \simeq^{\varnothing}_{\mathfrak{T}^{\mathfrak{n}}} [\![B]\!]^{\mathfrak{n}}$.

The algorithm is the result of adapting the scheme presented for subtyping to the new relation $\simeq^{\mathcal{R}}_{\mathfrak{T}^{\mathfrak{n}}}$. This is done by redefining functions children and check from the first and second phase respectively (*cf.* Fig. 16). For the former the only difference is on rule (E-FUNC-UP), where we need to add pair $\langle \mathcal{A}', \mathcal{B}' \rangle$ instead of $\langle \mathcal{B}', \mathcal{A}' \rangle$, added for subtyping. We could have omitted this by using the same rule for functional types as before and resorting to the symmetry of the resulting relation (which does not depend on this rule), but we wanted to emphasize the fact that phase one can easily be adapted if needed. For the refinement phase we need to properly check the premises of rules (E-UNION-UP) and (E-UNION-R-UP), while the others remain the same.

```
children(p) :
    case p of
      ⟨𝒟 @ 𝒜, 𝒟' @ ℬ⟩ →
          {⟨𝒟, 𝒟'⟩, ⟨𝒜, ℬ⟩}
      ⟨𝒜' ⊃ 𝒜'', ℬ' ⊃ ℬ''⟩ →
          {⟨𝒜', ℬ'⟩, ⟨𝒜'', ℬ''⟩}
      ⟨⊕_i^n 𝒜_i, ⊕_j^m ℬ_j⟩ →
          {⟨𝒜_i, ℬ_j⟩ | i ∈ 1..n, j ∈ 1..m}
      ⟨⊕_i^n 𝒜_i, ℬ⟩, ℬ ≠ ⊕ →
          {⟨𝒜_i, ℬ⟩ | i ∈ 1..n}
      ⟨𝒜, ⊕_j^m ℬ_j⟩, 𝒜 ≠ ⊕ →
          {⟨𝒜, ℬ_j⟩ | j ∈ 1..m}
      otherwise →
          ∅
```

```
check(p, F) :
    case p of
      ⟨a, a⟩ →
          true
      ⟨𝒟 @ 𝒜, 𝒟' @ ℬ⟩ →
          ⟨𝒟, 𝒟'⟩ ∉ F and ⟨𝒜, ℬ⟩ ∉ F
      ⟨𝒜' ⊃ 𝒜'', ℬ' ⊃ ℬ''⟩ →
          ⟨𝒜', ℬ'⟩ ∉ F and ⟨𝒜'', ℬ''⟩ ∉ F
      ⟨⊕_i^n 𝒜_i, ⊕_j^m ℬ_j⟩ →
          ∀i.∃j. ⟨𝒜_i, ℬ_j⟩ ∉ F and ∀j.∃i. ⟨𝒜_i, ℬ_j⟩ ∉ F
      ⟨⊕_i^n 𝒜_i, ℬ⟩, ℬ ≠ ⊕ →
          ∀i. ⟨𝒜_i, ℬ⟩ ∉ F
      ⟨𝒜, ⊕_j^m ℬ_j⟩, 𝒜 ≠ ⊕ →
          ∀j. ⟨𝒜, ℬ_j⟩ ∉ F
```

Figure 16 Pseudo-code of first (left) and second (right) phase for equivalence checking.

With these considerations is easy to see that, in each iteration, S consists of pairs in the relation $\simeq^{W}_{\mathfrak{T}^n}$, getting $S \subseteq \simeq^{\varnothing}_{\mathfrak{T}^n}$ at the end of the process.

▶ **Proposition 15.** *The algorithm preserves the following invariant:*
- $\langle W, S, F\rangle$ *is a partition of* U
- F *is composed solely of invalid pairs*
- $S \subseteq \Phi_{\simeq^{W}_{\mathfrak{T}^n}}(S)$

For the complexity analysis, notice that the size of the built universe is the same as before and phase one is governed by phase two, which has at most $\mathcal{O}(NN')$ iterations. For the cost of each iteration it is enough to analyze the complexity of check, since the rest of the scheme remains the same. As we remarked before, the only difference in check between subtyping and equality is in the cases involving unions. Here the worst case is when checking rule (E-UNION-UP) that requires the existence of two functions f and g relating elements of each type. This can be done in linear time by maintaining tables with the count of non-invalidated pairs of descendants, as indicated in Sec. 5.2.3. Thus, the cost of an iteration is $\mathcal{O}(d)$, resulting in an overall cost of $\mathcal{O}(NN'd)$ as before.

5.4 Type Checking

Let us revisit type-checking (tc). As already discussed, it linearly traverses the input term, the most costly operations being those that deal with application and abstraction. These cases involve calling subtype. Notice that these calls do not depend directly on the input to tc. However, a linear correspondence can be established between the size of the types being considered in subtype and the input to the algorithm, since such expressions are built from elements of Γ (the input context) or from annotations in the input term itself. Consider for instance subtype$(\varnothing, A, B)$ with a and b the size of each type resp. This has complexity $\mathcal{O}(a^2b^2d)$ and, from the discussion above, we can refer to it as $\mathcal{O}(n^4d)$, where n is the size of the input to tc (*i.e.* that of Γ plus t). Similarly, we may say that unfold is linear in n.

We now analyze the application and abstraction cases of the algorithm in detail:

Application First it performs a linear check on the type to verify if it is a datatype. If so it returns. Otherwise, a second linear check is required (unfold) in order to then perform as

many calls to `subtype` as elements there are in the union of the functional types. This yields a local complexity of $\mathcal{O}(n^4 d^2)$.

Abstraction First there are as many calls to `tcp` (the algorithm for type-checking patterns) as branches the abstraction has. Note that `tcp` has linear complexity in the size of its input and this call is instantiated with arguments p_i and θ_i which occur in the original term. All these calls, taken together, may thus be considered to have linear time complexity with respect to the input of `tcp`. Then it is necessary to perform a quadratic number (in the number of branches) of checks on compatibility. We have already analyzed that compatibility in the worst case involves checking subtyping. If we assume a linear number of branches w.r.t. the input, we obtain a total complexity of $\mathcal{O}(n^6 d)$ for this case.

Finally, the total complexity of `tc` is governed by the case of the abstraction, and is therefore $\mathcal{O}(n^7 d)$.

5.5 Prototype implementation

A prototype in Scala is available [8]. It implements `tc` but resorts to the efficient algorithm for subtyping and type equivalence described above. It also includes further optimizations. For example, following a suggestion in [6], the order in which elements in W are selected for evaluation relies on detecting strongly connected components, using Tarjan's [7] algorithm of linear cost and topologically sorting them in reverse order. In the absence of cycles this results in evaluating every pair only after all its children have already been considered. For cyclic types pairs for which no order can be determined are encapsulated within the same strongly connected component.

6 Conclusions

We address efficient type-checking for path polymorphism. We start off with the type system of [18] which includes singleton types, union types, type application and recursive types. The union type constructor is assumed associative, commutative and idempotent. First we formulate a syntax-directed presentation. Then we devise invertible coinductive presentations of type-equivalence and subtyping. This yields a naïve but correct type-checking algorithm. However, it proves to be inefficient (exponential in the size of the type). This prompts us to change the representation of type expressions and use automata techniques to considerably improve the efficiency. Indeed, the final algorithm has complexity $\mathcal{O}(n^7 d)$ where n is the size of the input and d is the maximum arity of the unions occurring in it.

Regarding future work an outline of possible avenues follows. These are aimed at enhancing the expressiveness of CAP itself and then adapting the techniques presented here to obtain efficient type checking algorithms.

- Addition of parametric polymorphism (presumably in the style of $F_{<:}$ [4, 5, 16]). We believe this should not present major difficulties.
- Strong normalization requires devising a notion of positive/negative occurrence in the presence of strong μ-type equality, which is known not to be obvious [2, page 515].
- A more ambitious extension is that of *dynamic patterns*, namely patterns that may be computed at run-time, PPC being the prime example of a calculus supporting this feature.

References

1 Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993. doi:10.1145/155183.155231.

2. H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. doi:10.1017/cbo9781139032636.
3. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998. doi:10.3233/fi-1998-33401.
4. Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings*, volume 526 of *Lecture Notes in Computer Science*, pages 750–770. Springer, 1991. doi:10.1007/3-540-54415-1_73.
5. Dario Colazzo and Giorgio Ghelli. Subtyping recursion and parametric polymorphism in kernel Fun. *Inf. Comput.*, 198(2):71–147, 2005. doi:10.1016/j.ic.2004.11.003.
6. Roberto Di Cosmo, François Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2005. doi:10.1007/11417170_14.
7. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980. doi:10.1145/322217.322228.
8. J. Edi and A. Viso. Prototype implementation of efficient type-checker in Scala. URL: https://github.com/juanedi/cap-typechecking.
9. J. Edi, A. Viso, and E. Bonelli. Efficient type checking for path polymorphism, 2017. arXiv preprint 1704.09026. URL: http://arxiv.org/abs/1704.09026.
10. B. Jay and D. Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, 2009. doi:10.1017/s0956796808007144.
11. Barry Jay. *Pattern Calculus - Computing with Functions and Structures*. Springer, 2009. doi:10.1007/978-3-540-89185-7.
12. T. Jim and J. Palsberg. Type inference in systems of recursive types with subtyping, 1999. Draft. URL: http://web.cs.ucla.edu/~palsberg/draft/jim-palsberg99.pdf.
13. Jan Willem Klop, Vincent van Oostrom, and Roel C. de Vrijer. Lambda calculus with patterns. *Theor. Comput. Sci.*, 398(1-3):16–31, 2008. doi:10.1016/j.tcs.2008.01.019.
14. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. *Math. Struct. Comput. Sci.*, 5(1):113–125, 1995. doi:10.1017/s0960129500000657.
15. Jens Palsberg and Tian Zhao. Efficient and flexible matching of recursive types. *Inf. Comput.*, 171(2):364–387, 2001. doi:10.1006/inco.2001.3090.
16. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
17. V. van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Vrije Universiteit Amsterdam, 1990. URL: http://www.phil.uu.nl/~oostrom/publication/pdf/IR-228.pdf.
18. Andrés Viso, Eduardo Bonelli, and Mauricio Ayala-Rincón. Type soundness for path polymorphism. *Electr. Notes Theor. Comput. Sci.*, 323:235–251, 2016. doi:10.1016/j.entcs.2016.06.015.
19. Jerome Vouillon. Subtyping union types. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 415–429. Springer, 2004. doi:10.1007/978-3-540-30124-0_32.
20. T. Zhao. *Type Matching and Type Inference for Object-Oriented Systems*. PhD thesis, Purdue University, 2002. URL: http://docs.lib.purdue.edu/dissertations/AAI3099873/.

Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of TYPES 2015 (LIPIcs, volume 69, http://www.dagstuhl.de/dagpub/978-3-95977-030-9 published in March, 2019). This version fixes a character set problem that caused some symbols to be displayed incorrectly.

Dagstuhl Publishing – March 17, 2019.

A Certified Study of a Reversible Programming Language*

Luca Paolini[1], Mauro Piccolo[2], and Luca Roversi[3]

1 **Dipartimento di Informatica, Università degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy**
lpaolini@unito.it
1 **Dipartimento di Informatica, Università degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy**
mrpiccol@gmail.com
1 **Dipartimento di Informatica, Università degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy**
lroversi@unito.it

Abstract

We advance in the study of the semantics of Janus, a C-like reversible programming language. Our study makes utterly explicit some backward and forward evaluation symmetries. We want to deepen mathematical knowledge about the foundations and design principles of reversible computing and programming languages. We formalize a big-step operational semantics and a denotational semantics of Janus. We show a full abstraction result between the operational and denotational semantics. Last, we certify our results by means of the proof assistant Matita.

1998 ACM Subject Classification F.1.2 Computation by Abstract Devices: Modes of Computation, F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages

Keywords and phrases reversible computing, Janus, operational semantics, bi-deterministic evaluation, categorical semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.7

1 Introduction

Reversible computing is an alternative form of computing: the isentropic core of classical computing [14] and quantum computing [21]. A classic computation is (forward-)deterministic, i.e. each state is followed by a unique state. The reversible computation is a classic computation which is also *backward*-deterministic: every state has a unique predecessor state. Research issues on this subject have emerged in a plethora of situations: isentropic digital circuits, conservative logic, computability and computational complexity, program transformation and software verification, view-update problem, unconventional computing models (bio, quantum, etc.), parallel computing and synchronization, processor architecture, debugging systems and other general backtrack-based settings. Perumalla [15] has recently surveyed many reversible computing facets.

Our focus is on the semantics of reversible programming languages.

These languages fully preserve information inherent in the input of their programs and they allow some form of built-in program inversion for free. Program inversion is the concrete counterpart that forward and backward deterministic computations let available

* Partially supported by the LINTEL project.

© Luca Paolini, Mauro Piccolo and Luca Roversi;
licensed under Creative Commons License CC-BY
21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 7; pp. 7:1–7:21

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

to the programmer. Each computation step can move forth/back between a unique pair of predecessor/successor states. Therefore, in these languages the backward execution of a program can result in an efficient search/backtrack-free process. We remark that, despite such a restricted-looking form of computation, reversible programming languages exist which can simulate every function which is computable in classical sense.

The subject of this work is Janus which Lutz and Derby conceived as a student project in 1982 at Caltech [10]. Janus is the first imperative structured programming language, explicitly supporting reversible computing. Yokoyama Glück present and study the language in [20]. Extensions, mainly addressed to introduce built-in programming facilities, are in [19, 18] by Yokoyama, Axelsen, and Glück who also formalize an operational semantics in [20] and improve it in [19, 18].

The operational semantics in [19, 18] does not naturally incorporate an efficient implementation of the reversible aspects which are specific to Janus. This is why we provide an all-round certified treatment of denotational and operational semantics for it.

Concerning the operational side, we define a fully self-contained *big-step* operational evaluation that formalizes a relation on terms which is injective. Our big-step operational semantics explicitly completes the one in [20] by formalizing the *"un-call of a procedure"* as a fully embedded deterministic process. The reason to make the "un-call" explicit is to put the backward and the forward computational directions at the same level, which sounds coherent with the spirit of the reversible computation.

Concerning the denotational side, we interpret the statements and the programs of Janus as functional injective relations. The interpretation composes suitable reversible categorical combinators which belong to **Pinj**, the category of sets and functional injective relations. The advantage of this approach is twofold. The correctness of the interpretation follows directly from composing combinators which we already know they are reversible. By the way, this addresses the possibility of synthesizing a combinatorial reversible language along the lines of James and Sabry [9] that we could use as a target language for compiling Janus. Furthermore, we prove that our denotational interpretation is fully abstract with respect to the operational semantics, entailing that operational and denotational equivalences coincide.

Finally, we certify our results by means of Matita [1]. Matita is an interactive theorem prover based on the Calculus of (Co)Inductive Constructions (CIC) — a dependent type theory. At the proof term level, Matita's proofs are compatible with those ones the theorem prover Coq [7] is based on. The certification is obtained by defining an abstract framework for imperative reversible languages. This abstract structure provides sufficient conditions to model classes of denotational and operational semantics for imperative languages which are reversible. It turns out that both the denotational and the operational semantics of Janus are possible instances of the framework. Getting both the operational and the denotational semantics as a instances of a general framework naturally allows us to fill some of the gaps that [20, 19, 18] leave open about the semantics of Janus. The formalization is available on-line [13].

2 Janus, a Reversible Programming Language

We introduce a minor variant of Janus, starting from [20], while we neglect some extensions which [19] provides. Specifically, we extend ground constants to all natural numbers unlike in [20] which limits them to 32-bit non-negative integer ranging from 0 to $2^{32} - 1$.

▶ **Definition 1** (The syntax of Janus). The grammar which generates *expressions*, *programs* and *statements* of the dialect of Janus we focus on is:

$$
\begin{array}{lll}
e & ::= & \mathtt{n} \,|\, x \,|\, e+e \,|\, e-e \,|\, e*e \,|\, e/e \,|\, e\%e \,|\, e <= e \,|\, e\,!= e \,|\, e == e \\
\mathtt{P} & ::= & (\mathtt{procedure}\; id\; s)(\mathtt{procedure}\; id\; s)^* \\
s & ::= & x \mathrel{+}= e \,|\, x \mathrel{-}= e \,|\, \mathtt{call}\; id \,|\, \mathtt{uncall}\; id \,|\, \mathtt{skip} \,|\, s\, s \\
 & & |\; \mathtt{if}\; e\; \mathtt{then}\; s\; \mathtt{else}\; s\; \mathtt{fi}\; e \;|\; \mathtt{from}\; e\; \mathtt{do}\; s\; \mathtt{loop}\; s\; \mathtt{until}\; e \quad .
\end{array}
$$

An expression e is either a numeral $\mathtt{n}$ (tacitly confused with a natural number in $\mathbb{N}$) a variable x or the application of a binary operator to sub-expressions. Arithmetic operators are $+, -, *, /, \%$. Relational ones are $<=, != , ==$. Under a standard convention, they return zero meaning `false`, and a number different from zero which stands for `true`. Binary operators are not necessarily reversible.

A program `P` consists of a list of *procedure declaration.* The keyword `procedure` starts a procedure declaration. A procedure identifier *id*, or procedure name, follows it. The declaration completes by means of a statement s, i.e. the procedure body.

A statement s is one among a *reversible assignment*, a procedure call, a procedure un-call, a skip, a statement sequence, a reversible conditional, or a reversible loop. In both reversible assignments $x \mathrel{+}= e$ and $x \mathrel{-}= e$ we ask that there are no free occurrences of the variable x in the expression e. All variables are global to the whole program as in [19] (we just omit their declaration at the beginning of programs).

We provide a formalization of the programming language Janus in the proof assistant Matita [1]. The formalization is available at [13]. There, we first provide a parametric syntax of Janus, where the choice of values, unary and binary operator used in expression and assignments and their behaviour is not fixed (file `janus.ma`). Then we show that all the properties of the operational semantics (like reversibility) hold for all possible instances of the parameters. Finally we provide the concrete instance of Janus being introduced by the following definition (file `concrjanus.ma`).

We now describe the semantics of Janus, informally. We informally recall the semantics of Janus following [20, 19, 18]. We remark that in informal discussion we confuse booleans and numbers used as truth-values (0 represents false while other numbers represent true). The leftmost diagram in Figure 1 helps understanding the work-flow of a reversible conditional $\mathtt{if}\; e\; \mathtt{then}\; s\; \mathtt{else}\; s\; \mathtt{fi}\; e$. Two branches are available both in forward direction, from left to right, and in the backward one, from right to left. Let us assume we are interpreting it forwardly, entering e_1 from its left. First we evaluate e_1. If it is "`true`", i.e. if e_1 yields a non zero value, we execute s_1. Otherwise, we execute s_2. No matter which between s_1 and s_2 we have executed, the interpretation proceeds as expected, i.e. it exits from e_2, only if e_2 yields the correct value. Such a value *must be* $\neq 0$ if we just came from s_1 and *must be* 0 if we just came from s_2. Every remaining combination results in a "system abort". The interpretation of Figure 1(a) from right to left does not abort only if, conversely, either both e_2 and e_1 yield "`true`" with s_1 executed as intermediate step, or either both e_2 and e_1 yield "`false`" with s_2 executed in-between them.

The rightmost diagram in Figure 1 helps understanding the work-flow of a reversible loop $\mathtt{from}\; e_1\; \mathtt{do}\; s_1\; \mathtt{loop}\; s_2\; \mathtt{until}\; e_2$. Let us assume we are interpreting it forwardly, trying to enter e_1 from its left. If e_1 gives "`false`", then we stress that the computation aborts. Otherwise, entering the loop means executing s_1 and evaluating e_2. As soon as e_2 results in a value $\neq 0$, the interpreter exits the loop. Otherwise the interpreter executes s_2. However, it keeps looping only if the value of e_1 is "`false`" just after the execution of s_2. On the

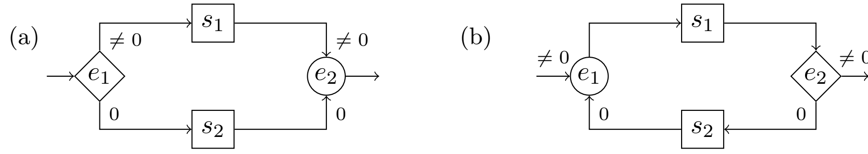


Figure 1 Work-flows of Reversible Conditional (a) and Reversible Loop (b).

contrary, getting a value $\neq 0$ from e_1, would result in a "system abort". The interpretation of Figure 1(b) from right to left is perfectly symmetric. The correct flow is: e_2 evaluates to "`true`", s_1 operates on the state of the program, e_1 evaluates to "`false`", s_2 operates on the state of the program, e_2 evaluates to "`false`" ... and so on until either we exit the loop, or we abort. Exiting is a consequence of evaluating e_1 to "`true`".

Calling and *un-calling* procedures execute the procedures in the right direction and will be the subject of the operational semantics in the coming sections.

We now formally certify the semantics of Janus. A main contribution is the formal certification of the study presented in this paper in Matita. The above definition has been formalized in Matita as an instance of an abstract syntax of Janus presented in the file named `janus.ma`.

In order to define a concrete instance of the abstract language we are referring to, we need to provide a set of constants representing the data being manipulated by the program (const_type), a special constant being the initial value of all variables appearing in the program (init_val), a sort for unary and binary operators (op1_type and op2_type), a sort for operators being used in reversible assignments (rev_type) together with a self-dual function on them used to reverse the variable assignment (rev). We discuss our Matita representation of Janus, in order to drive the reader in the certified formalization reading and understanding.

```
record params : Type[1] :=
{ const_type : DeqSet
; init_val  : const_type
; op1_type : Type[0]
; op2_type : Type[0]
; rev_type : Type[0]
; rev  :  rev_type → rev_type
; idem_prop : ∀x.rev (rev x) = x
}.
```

Both variables and procedure identifiers are implemented as inductive types with one constructor storing a natural number. This number represents the index in which the value of that variable or the body of that procedure can be found in the state.

```
inductive Variable : Type[0] :=
| var  :  ℕ → Variable.

inductive FunctionName : Type[0] :=
| a_function_id : ℕ→ FunctionName.
```

The set of expressions in Janus are defined in the following way. Expressions that could be variables, values, the application of an operator of arity 1 to an expression, or the application of an operator of arity 2 to two expressions.

```
inductive Expression (p : params) : Type[0] :=
| VAR : Variable → Expression p
| CONST : const_type p → Expression p
| OP_1 : op1_type p → Expression p → Expression p
| OP_2 : op2_type p → Expression p →
Expression p → Expression p.
```

Thus, the set of statements of Janus is implemented as an inductive type with the expected constructors, one for each syntactic construct.

```
inductive stm (p : params) : Type[0] :=
| ASSIGN : rev_type ...p → ∀ x : Variable.∀ e : Expression p.
(x ∈ (expr_fv ...e)) = false → stm p
| CALL : FunctionName → stm p
| UNCALL : FunctionName → stm p
| SKIP : stm p
| COMP : stm p → stm p → stm p
| IF : Expression p → stm p → stm p →
Expression p → stm p
| LOOP : Expression p → stm p → stm p →
Expression p → stm p.
```

In the previous definition and in some other following ones, we use the Matita syntactical construt .... We remind that the dots stand here for an arbitrary number of implicit arguments to be guessed by the system.

It turns out that a program is just a list of statements that constitute the bodies corresponding to each procedure identifier.

```
record program (p : params) : Type[0] :=
{ procs :> list (stm p)
}.
```

The procedure identifier carries a natural number. Thus it provides the index being the position in the list of statement being the body of the considered procedure.

3 Big-Step Operational Semantics

The operational semantics in Figure 2 comes directly from [19, 18]. It defines two relations $\Downarrow_p$ and $\Downarrow_e$ (program and expression evaluations) whose meaning we shall introduce once recalled some notations.

Let $\mathtt{P}$ be a program as in Definition 1. With $\sigma_{\mathtt{P}}$ we denote a state-function from the variables of $\mathtt{P}$ to $\mathbb{N}$. The state-function represents a statically allocated fragment of memory in a hypothetical real implementation. Thus, $\sigma_{\mathtt{P}}(x)$ is the value of x in $\sigma_{\mathtt{P}}$ whenever x is a variable of $\mathtt{P}$. The allocation of the value $\mathtt{n}$ to the variable x in $\sigma_{\mathtt{P}}$ is $\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]$. Conversely, the restriction of $\sigma_{\mathtt{P}}$ to all its names but x is $\sigma_{\mathtt{P}}\lceil_x$. With $\mathtt{P}(id)$ we identify the body of $\mathtt{P}$ with name id.

Let $\odot$ range over the operators $+, -, *, /, \%$. By $[\![\odot]\!]$ we denote its obvious interpretation. Every clause $\sigma_{\mathtt{P}}, e \Downarrow_e n$ says that the evaluation of the expression e in state $\sigma_{\mathtt{P}}$ yields the result $n \in \mathbb{N}$ using the rules:

$$\frac{}{\sigma_{\mathtt{P}}, \mathtt{n} \Downarrow_e \mathtt{n}}\ \textsc{Con} \qquad \frac{}{\sigma_{\mathtt{P}}, x \Downarrow_e \sigma_{\mathtt{P}}(x)}\ \textsc{Var} \qquad \frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}_1 \quad \sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}_2 \quad \mathtt{n}_1 [\![\odot]\!] \mathtt{n}_2 = \mathtt{n}}{\sigma_{\mathtt{P}}, e_1 \odot e_2 \Downarrow_e \mathtt{n}}\ \textsc{BinOp} \quad . \tag{1}$$

$$\frac{\sigma_{\mathtt{P}}\lceil_x, e \Downarrow_e \mathtt{n}_1 \quad \mathtt{n} = \mathtt{n}_0[\![\odot]\!]\mathtt{n}_1 \quad \odot \in \{+,-\}}{\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}_0], x \odot\!= e \Downarrow_p \sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]}\ \textsc{AssVar} \qquad \frac{}{\sigma_{\mathtt{P}}, \mathtt{skip} \Downarrow_p \sigma_{\mathtt{P}}}\ \textsc{Skip}$$

$$\frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{IfTrue} \qquad \frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_2 \Downarrow_e 0}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{IfFalse}$$

$$\frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2 \Downarrow_p \sigma''_{\mathtt{P}}}\ \textsc{LoopMain} \qquad \frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma_{\mathtt{P}}}\ \textsc{LoopBase}$$

$$\frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_1 \Downarrow_e 0 \quad \sigma'_{\mathtt{P}}, s_1 \Downarrow_p \sigma''_{\mathtt{P}} \quad \sigma''_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'''_{\mathtt{P}}}\ \textsc{LoopRec}$$

$$\frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{call}\ id \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{Call} \qquad \frac{\sigma'_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p \sigma_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{uncall}\ id \Downarrow_p \sigma'_{\mathtt{P}}}\ \textsc{Uncall} \qquad \frac{\sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, s_2 \Downarrow_p \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, s_1\ s_2 \Downarrow_p \sigma''_{\mathtt{P}}}\ \textsc{Seq}$$

Figure 2 Original operational semantics of Janus.

$$\frac{\sigma_{\mathtt{P}}\lceil_x, e \Downarrow_e \mathtt{n}_1 \quad \mathtt{n} = \mathtt{n}_0[\![+]\!]n_1}{\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}_0], x -\!= e \Downarrow^{®}_p \sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]}\ +^{®} \qquad \frac{\sigma_{\mathtt{P}}\lceil_x, e \Downarrow_e \mathtt{n}_1 \quad \mathtt{n} = \mathtt{n}_0[\![-]\!]\mathtt{n}_1}{\sigma_{\mathtt{P}}[x \mapsto \mathtt{n}_0], x +\!= e \Downarrow^{®}_p \sigma_{\mathtt{P}}[x \mapsto \mathtt{n}]}\ -^{®} \qquad \frac{}{\sigma_{\mathtt{P}}, \mathtt{skip} \Downarrow^{®}_p \sigma_{\mathtt{P}}}\ \textsc{Skip}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow^{®}_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow^{®}_p \sigma'_{\mathtt{P}}}\ \textsc{IfTrue}^{®} \qquad \frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow^{®}_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_1 \Downarrow_e 0}{\sigma_{\mathtt{P}}, \mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2 \Downarrow^{®}_p \sigma'_{\mathtt{P}}}\ \textsc{IfFalse}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n}+1 \quad \sigma_{\mathtt{P}}, s_1 \Downarrow^{®}_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow^{®}_p \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2 \Downarrow^{®}_p \sigma''_{\mathtt{P}}}\ \textsc{LoopMain}^{®} \qquad \frac{\sigma_{\mathtt{P}}, e_1 \Downarrow^{®}_e \mathtt{n}+1}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma_{\mathtt{P}}}\ \textsc{LoopBase}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, e_1 \Downarrow_e 0 \quad \sigma_{\mathtt{P}}, s_2 \Downarrow^{®}_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, e_2 \Downarrow_e 0 \quad \sigma'_{\mathtt{P}}, s_1 \Downarrow^{®}_p \sigma''_{\mathtt{P}} \quad \sigma''_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow^{®}_p \sigma'''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow^{®}_p \sigma'''_{\mathtt{P}}}\ \textsc{LoopRec}^{®}$$

$$\frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow^{®}_p \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{call}\ id \Downarrow^{®}_p \sigma'_{\mathtt{P}}}\ \textsc{Call}^{®} \qquad \frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{uncall}\ id \Downarrow^{®}_p \sigma'_{\mathtt{P}}}\ \textsc{coCall}^{®} \qquad \frac{\sigma_{\mathtt{P}}, s_2 \Downarrow^{®}_p \sigma'_{\mathtt{P}} \quad \sigma'_{\mathtt{P}}, s_1 \Downarrow^{®}_p \sigma''_{\mathtt{P}}}{\sigma_{\mathtt{P}}, s_1\ s_2 \Downarrow^{®}_p \sigma''_{\mathtt{P}}}\ \textsc{Seq}^{®}$$

Figure 3 Rules that formalize how the backward interpretation of Janus works.

We remark that no rule here above have side effects on $\sigma_{\mathtt{P}}$.

Concerning $\Downarrow_p$, every clause $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'_{\mathtt{P}}$ says that the evaluation of the statement s in state $\sigma_{\mathtt{P}}$ yields the state $\sigma'_{\mathtt{P}}$ possibly affected by the side effects of s.

It is usual (albeit not mandatory) to assume that variables are initialized to zero starting the evaluation of a program, i.e. executing the body of the first procedure. The evaluation of $\mathtt{P} = \mathtt{procedure}\ id_1; s_1 \ldots \mathtt{procedure}\ id_n; s_n$ starting from $\sigma_{\mathtt{P}}$ stops whenever $\sigma_{\mathtt{P}}, \mathtt{call}\ id_1 \Downarrow_p \sigma'_{\mathtt{P}}$ holds, for some $\sigma'_{\mathtt{P}}$.

Some remarks on the rules in Figure 2 are worth making and help moving towards our first contribution.

The rules LoopMain, LoopRec and LoopBase introduce the auxiliary syntax (e_1, s_1, s_2, e_2). It separates the interpretation phases of $\mathtt{from}\ e_1\ \mathtt{do}\ s_1\ \mathtt{loop}\ s_2\ \mathtt{until}\ e_2$. LoopMain introduces (e_1, s_1, s_2, e_2) if e_1 evaluates to "true". LoopRec uses (e_1, s_1, s_2, e_2) to keep unfolding the loop only if e_1 and e_2 evaluate to "false". LoopBase concludes the interpretation of the loop.

Last, the rule Uncall in Figure 2 serves to unroll the interpretation of the procedure with name id. The rule is effectively computable but it is non-deterministic and inefficient. Uncall must pick up in the whole set of state-functions, a $\sigma'_{\mathtt{P}}$ such that when evaluating the statement $\mathtt{P}(id)$ we obtain the "input" state $\sigma_{\mathtt{P}}$. Uncall does not provide any explicit guideline on how explicitly and efficiently finding that such a state $\sigma'_{\mathtt{P}}$ exists at the level of big-step semantics.

Our first contribution is the definition of a big-step operational semantics which makes the process of reversing the interpretation of a program both explicit and efficient. The operational semantics we propose contains the union between the set of new rules in Figure 3 and the set of rules in Figure 2 with the proviso of replacing

$$\frac{\sigma_{\mathtt{P}}, \mathtt{P}(id) \Downarrow_p^{®} \sigma'_{\mathtt{P}}}{\sigma_{\mathtt{P}}, \mathtt{uncall}\ id \Downarrow_p \sigma'_{\mathtt{P}}}\ \text{coCall} \tag{2}$$

for Uncall.

An other possible strategy to recover efficiency could be to define a compiler that translates the "un-call" of $\mathtt{P}(id)$ into the equivalent program of Janus that we can interpret with the operational semantics in Figure 2. This is proposed in [20]. The rules in Figure 3 somewhat embed the self-interpreter in [20] directly into the evaluation process. Our completion of the operational semantics greatly improves the understanding of Janus and the possibility to formally reason on it, as we do in Matita.

▶ **Definition 2.** Given a program $\mathtt{P}$, the two mutually defined relations $\Downarrow_p$ and $\Downarrow_p^{®}$ map a pair with a state $\sigma_{\mathtt{P}}$ and a statement s into a state $\sigma'_{\mathtt{P}}$. The relations hold whenever there is a finite derivation of $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'_{\mathtt{P}}$ or $\sigma_{\mathtt{P}}, s \Downarrow_p^{®} \sigma'_{\mathtt{P}}$, using the rules of Figure 2 with coCall in place of Uncall and the rules of Figure 3. The evaluation of $\mathtt{P}$ exists if $\sigma_{\mathtt{P}}, \mathtt{P} \Downarrow_p \sigma'_{\mathtt{P}}$, for some $\sigma_{\mathtt{P}}$ and $\sigma'_{\mathtt{P}}$.

Some comments on the rules in Figure 3 are worth making before stating the main properties of the operational semantics we propose. Roughly, those rules interpret programs backwardly.

For instance, Seq® evaluates $s_1 s_2$ by starting from s_2. The rule LoopMain® mirrors the behavior of LoopMain. It checks whether e_2 evaluates to a value $\neq 0$ and introduces (e_1, s_1, s_2, e_2) to signal that the interpretation of a reversible loop has just started. The choice between LoopRec® or LoopBase® follows from the value that e_1 — not e_2 — produces. Analogously, the interpretation of $\mathtt{if}\ e_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi}\ e_2$ under $\Downarrow_p^{®}$ starts from evaluating e_2. Finally, let us focus on Call® and coCall®. The first one works like Call but on the "reversed" procedure. We mean that Call® executes the procedure from its conclusion, replacing every instruction by means of its "opposite". Instead, coCall® re-reverses the evaluation "direction" moving again the evaluation control flow to $\Downarrow_p$.

Figure 4 shows the details about what "undoing a procedure-call" means directly inside the new, full blown, big-step operational semantics.

The following Lemma is preliminary to the main properties. If the control-flow enters a loop (cf. Figure 1(b)), then s_1 is executed once and the control moves to the exit-conditional: either the control exits the loop or, s_1, s_2 are both executed (forwardly or backwardly), before to re-check the exit-conditional.

▶ **Lemma 3** (Length of loops and number of states)**.**

1. $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'_{\mathtt{P}}$ *if and only if there are* $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ *where* $m \in \mathbb{N}$ *(*$2m$ *is the double of* m*),* $\sigma^0_{\mathtt{P}} = \sigma_{\mathtt{P}}$*,* $\sigma^{2m}_{\mathtt{P}} = \sigma'_{\mathtt{P}}$*, such that:*
   - $\sigma^{2m}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n+1}$ *and* $\sigma^{2i}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{0}$*,* $\sigma^{2i}_{\mathtt{P}}, s_2 \Downarrow_p \sigma^{2i+1}_{\mathtt{P}}$ *for* $i < m$*;*
   - $\sigma^{2i+1}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{0}$ *for* $i < m$*, and* $\sigma^{2i+1}_{\mathtt{P}}, s_1 \Downarrow_p \sigma^{2i+2}_{\mathtt{P}}$ *for* $i \leq m-2$*.*
2. $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{®} \sigma'_{\mathtt{P}}$ *if and only if there are* $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ *where* $m \in \mathbb{N}$*,* $\sigma^0_{\mathtt{P}} = \sigma_{\mathtt{P}}$*,* $\sigma^{2m}_{\mathtt{P}} = \sigma'_{\mathtt{P}}$*, such that:*
   - $\sigma^{2m}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n+1}$ *and* $\sigma^{2i}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{0}$*,* $\sigma^{2i}_{\mathtt{P}}, s_1 \Downarrow_p^{®} \sigma^{2i+1}_{\mathtt{P}}$ *for* $i < m$*;*
   - $\sigma^{2i+1}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{0}$ *for* $i < m$*, and* $\sigma^{2i+1}_{\mathtt{P}}, s_2 \Downarrow_p^{®} \sigma^{2i+2}_{\mathtt{P}}$ *for* $i \leq m-2$*.*

Proof. ($\Rightarrow$). By induction on the derivation proving $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'_{\mathtt{P}}$.
($\Leftarrow$). By induction on the derivation proving $\sigma_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'_{\mathtt{P}}$. ◀

Forward and backward computation are related in the correct and expected way, as stated by the following Theorem.

▶ **Theorem 4** ($\Downarrow_p$ and $\Downarrow_p^{®}$ annihilate each other)**.** *$\sigma_{\mathtt{P}}, s \Downarrow_p \sigma^*_{\mathtt{P}}$ if and only if $\sigma^*_{\mathtt{P}}, s \Downarrow_p^{®} \sigma_{\mathtt{P}}$.*

Proof. ($\Rightarrow$). The proof is given by induction on s. All cases are straightforward except for loop. Suppose $\sigma_{\mathtt{P}}, \mathtt{from}\, e_1\, \mathtt{do}\, s_1\, \mathtt{loop}\, s_2\, \mathtt{until}\, e_2 \Downarrow_p \sigma^*_{\mathtt{P}}$. Thus $\sigma_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{n+1}$, $\sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}}$ and $\sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma^*_{\mathtt{P}}$. By Lemma 3.1 there are $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ where $m \in \mathbb{N}$, $\sigma^0_{\mathtt{P}} = \sigma'_{\mathtt{P}}$, $\sigma^{2m}_{\mathtt{P}} = \sigma^*_{\mathtt{P}}$, such that: $\sigma^{2m}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{n+1}$ and $\sigma^{2i}_{\mathtt{P}}, e_2 \Downarrow_e \mathtt{0}$, $\sigma^{2i}_{\mathtt{P}}, s_2 \Downarrow_p \sigma^{2i+1}_{\mathtt{P}}$ for $i < m$; $\sigma^{2i+1}_{\mathtt{P}}, e_1 \Downarrow_e \mathtt{0}$ for $i < m$, and $\sigma^{2i+1}_{\mathtt{P}}, s_1 \Downarrow_p \sigma^{2i+2}_{\mathtt{P}}$ for $i \leq m-2$. Re-organizing (in reverse order the list of state), we have that the list $\sigma^{2m-1}_{\mathtt{P}}, \ldots, \sigma^0_{\mathtt{P}}, \sigma_{\mathtt{P}}$ satisfies the right-hand side of the statement Lemma 3.2. Thus, $\sigma^{2m-1}_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p^{®} \sigma_{\mathtt{P}}$. It is easy to conclude $\sigma^*_{\mathtt{P}}, \mathtt{from}\, e_1\, \mathtt{do}\, s_1\, \mathtt{loop}\, s_2\, \mathtt{until}\, e_2 \Downarrow_p^{®} \sigma_{\mathtt{P}}$. ($\Leftarrow$). Dual to the above proof. ◀

The operational semantics we introduce is deterministically syntax driven. Given a statement and chosen which evaluation between $\Downarrow_p$ or $\Downarrow_p^{®}$ to use, we can apply a single rule at every step. The consequence is the following Corollary, whose existence we anticipated in the introduction

▶ **Corollary 5** ($\Downarrow_p$ and $\Downarrow_p^{®}$ are functional and injective)**.** *The relations $\Downarrow_p$ and $\Downarrow_p^{®}$ are functional, i.e.:*
1. *If $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma^1_{\mathtt{P}}$ and $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma^2_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*
2. *If $\sigma_{\mathtt{P}}, s \Downarrow_p^{®} \sigma^1_{\mathtt{P}}$ and $\sigma_{\mathtt{P}}, s \Downarrow_p^{®} \sigma^2_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*
The relations $\Downarrow_p$ and $\Downarrow_p^{®}$ are injective, i.e.:
1. *If $\sigma^1_{\mathtt{P}}, s \Downarrow_p \sigma_{\mathtt{P}}$ and $\sigma^2_{\mathtt{P}}, s \Downarrow_p \sigma_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*
2. *If $\sigma^1_{\mathtt{P}}, s \Downarrow_p^{®} \sigma_{\mathtt{P}}$ and $\sigma^2_{\mathtt{P}}, s \Downarrow_p^{®} \sigma_{\mathtt{P}}$, then $\sigma^1_{\mathtt{P}} = \sigma^2_{\mathtt{P}}$.*

The proofs of Lemma 3 and Theorem 4 are certified in the file `janus.ma`, while the proof of Corollary 5 is formalized in the file `completeness.ma` in [13].

We now give some comments on how we represent and certify the properties of the whole operational semantics in Figure 2 and 3. To this aim we need to comment about how the abstract syntax and its instances work. We start from the evaluation of unary and binary operators. This formalization is in the file `janus.ma` in [13].

```
record sem_params (p : params) : Type[0] :=
{ evaluate_op1 : op1_type ...p → const_type ...p → option (const_type ...p)
; evaluate_op2 : op2_type ...p → const_type ...p → const_type ...p
→ option (const_type ...p)
; evaluate_rev : rev_type ...p → const_type ...p → const_type ...p
→ option (const_type ...p)
; const_to_bool : const_type ...p → bool
; reverse_eval_rev : ∀ r,a,b,c.evaluate_rev r a b = return c
→ evaluate_rev (rev ...r) c b = return a
}.
```

The behaviour of unary operators is specified by the field evaluate_op1. The behaviour of binary operators is specified by the field evaluate_op2. The evaluation of the binary operators used in reversible assignments is specified by the field evaluate_rev. We require also to specify a canonical projection const_to_bool from constant types to boolean ones, that will be used in the evaluation of the expression guards in conditional and loop statements. Furthermore

we require that the evaluation of a binary operator used in reversible assignment can be reversed by using the specific reverse operator specified by the function specified in the field rev of record params.

All the procedures implementing the evaluation of the above mentioned operators may fail. In order to keep track of this possibility, we use the option monad, which is specified in the standard library of Matita.

```
inductive option (A:Type[0]) : Type[0] :=
None : option A
| Some : A → option A.
```

In the file `concrjanus.ma` of [13] we provide an instance of the above record which correspond to the concrete operational semantics in Figures 2 and 3. Let us see how to instantiate the record sem_params. Since there is no unary operator, we do not provide any implementation. The procedures implementing the behaviour of remaining operators are straightforward by cases. The projection from integers to boolean values is the standard one: zero is `false` and every non-zero value correspond to `true`. Finally a proof of correctness for the specific choice of operators used in reversible assignments is provided.

The specification of the behaviour of all operators involved in the syntax of expressions makes it possible to define how the expressions are evaluated in a given program state.

```
definition syn_state :=λ p : params.list (const_type ...p).

let rec evaluate_expression (p : params) (p' : sem_params p)
(e : Expression p) (st : syn_state p) on e : option (const_type ...p) :=...
```

A state (syn_state) is just a list of constant values. The Matita representation of variables assumes that each variable carries an index identifying uniquely the position where its value is stored. After having provided suitable instances of both the records params and sem_params as specified above the evaluation of an expression e on a given state st is described by the procedure evaluate_expression. Notice that the evaluation of an expression on a given state may fail. When e is a variable, then it is evaluated to its value in the state st (if the index carried by the variable is not beyond the length of the state). The value of a constant is the constant itself. The value of an expression obtained by applying an unary (resp. binary) operator o to a sub-expression(s) e1 (and e2) is equal to the application of the behaviour of that operator to the value of the expression e1 (and the value of e2).

```
let rec fwd_operational_semantics (p : params) (p' : sem_params p)
(env : list (stm p)) (q : stm p) (s1 : syn_state p)
(n : ℕ) on n : option (syn_state p) :=
match n with
[ O ⇒None ?
| S m ⇒match q with [....]
]
and
bwd_operational_semantics (p : params) (p' : sem_params p)
(env : list (stm p)) (q : stm p) (s1 : syn_state p)
(n : ℕ) on n : option (syn_state p) :=
[ O ⇒None ?
| S m ⇒match q with [....]
]
```

$$\left\{\begin{array}{c} x == m \\ y == n \\ z == 0 \\ w == 0 \end{array}\right\} \texttt{call}\ incr; \left\{\begin{array}{c} x == m \\ y == n+m \\ z == m \\ w == 0 \end{array}\right\} \texttt{call}\ copy; \left\{\begin{array}{c} x == m \\ y == n+m \\ z == m \\ w == n+m \end{array}\right\} \texttt{uncall}\ incr; \left\{\begin{array}{c} x == m \\ y == n \\ z == 0 \\ w == n+m \end{array}\right\} \texttt{call}\ exchange; \left\{\begin{array}{c} x == m \\ y == n \\ z == n+m \\ w == 0 \end{array}\right\}$$

Figure 4 The sequence of transformations that `call` *sum* in (3) operates on the initial state.

Since Janus allows both forward and backward execution, we have to provide a proper way to unroll the computation. We defined a fully fledged operational semantics of Janus statements in a given program state. It is realized by two mutually recursive procedures named respectively fwd_operational_semantics and bwd_operational_semantics. The former defines how a statement is executed in forward way while the latter defines how a statement is executed backward way.

Both procedures take in input also a natural number n used as a threshold limit to the number of steps needed by both the forward and backward executors to reach a final state. It follows that the evaluation predicate $\sigma, s \Downarrow_p \sigma'$ can be expressed in Matita by postulating the existence of a number n such that the final state σ' is reached from σ when evaluating s after having settled n as threshold.

We have proven a monotonicity result of both forward and backward execution i.e. if in the evaluation of a statement a final state is reached in at least n steps threshold, then it can again be reached even if the threshold is augmented. Monotonicity entails that the operational semantics is forward deterministic. Backward determinism is guaranteed by the fact that if the forward evaluation of a statement on a given state s1 evaluates to the state s2 then the backward evaluation of the same statement from s2 evaluates to s1. These results hold for every choice of syntactical and semantic parameters, thus they hold also for the concrete language formalized in `concrjanus.ma`, providing in this way the certification of Theorem 4 and Corollary 5.

```
theorem op_sem_reversibility :
∀p : params.∀p' : sem_params p.∀env : list (stm p).
∀q : stm p.∀s1,s2 : syn_state p.∀n : ℕ.
(fwd_operational_semantics p p' env q s1 n = return s2 →
bwd_operational_semantics p p' env q s2 n = return s1) ∧
(bwd_operational_semantics p p' env q s1 n = return s2 →
fwd_operational_semantics p p' env q s2 n = return s1).

theorem operational_semantics_monotone : ∀p : params.∀p' : sem_params p.
∀env : list (stm p).∀q : stm p.∀s1,s2 : syn_state p.∀n,m : ℕ.n≤m →
(fwd_operational_semantics p p' env q s1 n = return s2 →
fwd_operational_semantics p p' env q s1 m = return s2).

theorem bwd_operational_semantics_monotone : ∀p : params.∀p' : sem_params p.
∀env : list (stm p).∀q : stm p.∀s1,s2 : syn_state p.∀n,m : ℕ.n≤m →
(bwd_operational_semantics p p' env q s1 n = return s2 →
bwd_operational_semantics p p' env q s1 m = return s2).
```

3.1 A Running Example

The sum of two natural numbers is computable, but certainly not injective. More precisely, if we say that the result of a sum is 8, we do not know if it results from summing 5 and 3 or 2 and 6. Following Bennett [4], we can program the sum of x and y in Janus as follows:

$$Rsum(x, y, z) = \begin{cases} (x, y, x + y) & \text{if } z = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function $Rsum : \mathbb{N}^3 \to \mathbb{N}^3$ is reversible because it is defined to preserve the input values. Using the very rich language of expressions of Janus, $RSum$ simply can be
`from` $z = 0$ `do` $z+ = (x + y)$ `loop skip until` 1. As a curiosity the same function exists in a restriction of the syntax of expression that only contains successor, predecessor and an equality test. Inside the restriction $RSum$ is given by the following Janus-program (where we used semicolons and a natural indentation to make explicit the language parsing).

```
procedure main
    call sum;
procedure sum
    call incr; call copy; uncall incr; call exchange;
procedure incr
    from z = 0 do skip loop z += 1; y += 1 until z = x;        (3)
procedure copy
    from w = 0 do skip loop w += 1 until w = y;
procedure exchange
    from z = 0 do skip loop z += 1; w-= 1 until w = 0;
```

The execution of (3) would yield $\sigma_{\mathtt{P}}, \mathtt{P}(sum) \Downarrow \sigma_{\mathtt{P}}[\mathtt{z} \mapsto \sigma_{\mathtt{P}}(\mathtt{x}) + \sigma_{\mathtt{P}}(\mathtt{y})]$ for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{z}) = 0$ and $\sigma_{\mathtt{P}}(\mathtt{w}) = 0$. We have that the following statements hold:

- $\sigma_{\mathtt{P}}, \mathtt{P}(incr) \Downarrow \sigma_{\mathtt{P}}[\mathtt{y} \mapsto \sigma_{\mathtt{P}}(\mathtt{x}) + \sigma_{\mathtt{P}}(\mathtt{y}), \mathtt{z} = \sigma(\mathtt{x})]$, for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{z}) = 0$,
- $\sigma_{\mathtt{P}}, \mathtt{P}(copy) \Downarrow \sigma_{\mathtt{P}}[\mathtt{w} \mapsto \sigma_{\mathtt{P}}(\mathtt{y})]$, for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{w}) = 0$, and
- $\sigma_{\mathtt{P}}, \mathtt{P}(exchange) \Downarrow \sigma_{\mathtt{P}}[\mathtt{z} \mapsto \sigma_{\mathtt{P}}(\mathtt{w}), \mathtt{w} = 0]$, for all $\sigma_{\mathtt{P}}$ such that $\sigma_{\mathtt{P}}(\mathtt{z}) = 0$.

Compactly, the sequence of predicates that describes the states change along the execution of the procedure are depicted in Figure 4.

Notice that the variable w is a variable for which we can predict its final value at the end of computation of the sum, so it could be safely deleted. For this purpose it might be useful to extend the syntax of Janus with the reversible allocation of local variables as in [19]. Let $decr$ be defined as `from` $z = x$ `do` $z -= 1; y -= 1$ `loop skip until` $z = 0$. A call $decr$ can be used to replace `uncall` $incr$. Symmetrically, a uncall to $decr$ be used to replace `call` $incr$.

4 The Model

In this section, we introduce step by step the components that we will need in the next section to define the model of Janus. We are going to interpret Janus statements and programs in terms of relations between states. These relations are both functional and injective. We show a full abstraction between the equality on programs induced by the denotational semantics and the equality induced by the operational semantics. Both the denotational semantics and the full abstraction theorems have been formalized in the proof assistant Matita. The formalization provided in the files `rel.ma`, `pinj.ma`, `rel_interpretation.ma`. The correspondence results are formalized in `correctness.ma`, `completeness.ma` and `compl_thm.ma` in [13].

4.1 A Category of Partial Injective Functions

We denote with **Rel** the category of sets and relations. A relation $r : A \to B$ is *functional* if $(a, b), (a, b') \in r$ implies $b = b'$ for all a, b, b'. A relation $r : A \to B$ is *injective* if $(a, b), (a', b) \in r$ implies $a = a'$ for all a, a', b. **Pinj** is a subcategory of **Rel** whose objects are

sets and whose morphisms are functional injective relations (partial injective function, in other words). We will denote with **Pfn** the subcategory of **Rel** whose objects are sets and whose morphisms are functional relations.

We will denote with $id_A : A \to A$ the identity relation on A and with $\circ$ the relational composition. When $f : A \to B$ is a functional relation and $a \in A$, we sometimes write $f(a)$ to denote the unique $b \in B$ (if it exists) such that $(a, b) \in f$. When $\odot : A \times B \to C$ is a functional relation, we could also use the infix notation $a \odot b$ to denote the unique $c \in C$ (if it exists) such that $((a, b), c) \in \odot$.

Given $r : A \to B$ in **Pinj** we define the *inverse* of r denoted with $r^\dagger$ as $\{(b, a) \mid (a, b) \in r\}$. It is again a morphism in **Pinj**.

Both **Pinj** and **Pfn** admit two symmetric tensor products. They are the *cartesian product* $\times$ (a.k.a. product) and the *disjoint union* $+$ (a.k.a. sum or coproduct). We will denote with $\mathbf{1}$ the singleton set i.e. the unit of the product, and with $\mathbf{0}$ the empty set i.e. the unit of the sum. We will denote with

$$\begin{array}{ll} \alpha^\times_{A,B,C} : A \times (B \times C) \to (A \times B) \times C & \alpha^+_{A,B,C} : A + (B + C) \to (A + B) + C \\ \sigma^\times_{A,B} : A \times B \to B \times A & \sigma^+_{A,B} : A + B \to B + A \\ \lambda^\times_A : A \to \mathbf{1} \times A & \lambda^+_A : A \to \mathbf{0} + A \end{array}$$

respectively the associative laws of product and sum, the symmetric laws of product and sum and the left neutral element laws of product and sum: they are the same morphisms in both **Pinj** and **Pfn**. Sometimes, the apexes and the subscripts will be omitted when clear from the context or uninteresting. These morphisms satisfy the standard properties that make both the sum and the product two symmetric tensor products in both **Pinj** and **Pfn**.

The two tensor products are related by a distributive law that establishes a natural isomorphism $\delta : (B + C) \times A \to (B \times A) + (C \times A)$. It is defined in the following way

$$\delta = \{((inl(b), a), inl(b, a)) \mid a \in A, b \in B\} \cup \{((inr(c), a), inr(c, a)) \mid a \in A, c \in C\}$$

while its inverse $\delta^\dagger_{A,B,C}$ is defined as

$$\delta^\dagger = \{(inl(b, a), (inl(b), a)) \mid a \in A, b \in B\} \cup \{(inr(c, a), (inr(c), a)) \mid a \in A, c \in C\}$$

Suppose $r : X + V \to Y + U$ is a relation. The coproduct injections induce four restricted relations $r_{ll}, r_{rl}, r_{lr}, r_{rr}$ defined as $r_{ll} = \{(x, y) \in X \times Y \mid (inl(x), inl(y)) \in r\}$, $r_{rl} = \{(v, y) \in V \times Y \mid (inr(v), inl(y)) \in r\}$, $r_{lr} = \{(x, u) \in X \times U \mid (inl(x), inr(u)) \in r\}$ and $r_{rr} = \{(v, u) \in V \times U \mid (inr(v), inr(u)) \in r\}$. We denote with $\cdot^\star$ the reflexive transitive closure of a relation. Given $r : X + U \to Y + U$, its trace can be defined as follows:

$$\mathrm{Tr}^U_{X,Y} r = r_{ll} \cup r_{rl} \circ r^\star_{rr} \circ r_{lr} : X \to Y$$

where $r^\star_{rr}$ is the reflexive transitive closure of r_{rr}. This operator is sometimes known as particle-style trace operator. The trace operation preserves injectivity and functionality of relations, namely if $r : X + U \to Y + U$ is functional (resp. injective) then $\mathrm{Tr}^U_{X,Y} r$ is functional (resp. injective). For details see Blute and Scott [6].

▶ **Corollary 6.** **Rel**, **Pfn** *and* **Pinj** *are symmetric traced monoidal categories.*

The proof of the above corollary has been certified in the file `pinj.ma` of [13].

The standard notion of assignment cannot be coherent with the idea of reversible computation. Just observe that the value v in a given variable is lost forever once we overwrite v by $u \neq v$. The way out is a restricted version of assignment, called *reversible update* by Axelsen, and Yokoyama [3], which we recall in the following.

▶ **Definition 7.** A functional relation $\odot : (A \times B) \to C$ is *first argument injective operator* if and only if $((a, b), c) \in \odot$ and $((a', b), c) \in \odot$ then $a = a'$.

Equivalently, if a exists such that $a \odot b = c$, for some b, c, then a is unique. Then we can define an operator $\oslash : (C \times B) \to A$ as $\oslash = \{((c, b), a) \mid a \odot b = c\}$. Notice that $\oslash$ is again a first argument injective operator and it is such that $\forall a \in A.\forall b \in B.((a \odot b) \oslash b) = a$ and $((a \oslash b) \odot b) = a$. Sometimes we identify $\oslash$ as *the reverse of* $\odot$.

▶ **Definition 8.** Given a functional (possibly non-injective) relation $f : D \to B$ and a first argument injective operator $\odot : (A \times B) \to C$, a partial function $g : (A \times D) \to (C \times D)$ is a *reversible update wrt to its first argument* if it is functionally equivalent to

$$g(x, y) = (x \odot f(y), y).$$

There always exists a (left) inverse for a reversible update:

$$g^{\dagger}(x, y) = (x \oslash f(y), y)$$

where $\oslash$ is the reverse of $\odot$. Thus a reversible update g is necessarily injective.

Given an operator $\odot : A \times B \to C$ being injective in his first argument and given a partial function $f : D \to B$, we denote with $ru(\odot, f) : A \times D \to C \times D$ the partial injective function $ru(\odot, f)(x, y) = (x \odot f(y), y)$ which is a reversible update.

Reversible updates are particularly well suited to model changes of a computation state where one part of the state is updated using the remaining part of the state that is not changed. As an example consider $g(x, y) = (x+f(y), y)$ and its inverse $g^{\dagger}(x, y) = (x-f(y), y)$. A reversible update with the XOR (^) is self-inverse for any f: $g(x, y) = g^{\dagger}(x, y) = (x\hat{\ }f(y), y)$. See [3] for more details on reversible updates.

4.2 Formalization of the Model

The formalization of **Pinj** in Matita relies on a domain-theoretic substrate whose purpose is to certify a whole set of standard properties which allows us to prove the categorical/denotational correctness of a model.

We assume some familiarity with the notion and the formalization of *chain complete partial orders* (CPOs), CPO-enriched categories and monoidal categories. The formalization of domain theory is based on Benton, Kennedy, and Varming [5] and it is in `domain.ma`. We remind that the notion of chain on a CPO D is defined as any monotonic function $c : \mathbb{N} \to D$ where $\mathbb{N}$ is ordered in the natural way. Formalization of CPO-enriched category and monoidal categories are in the files `category.ma` and `monoidal_category.ma`. Our formalization of categories adapts a fragment of the COQ library [11]. As usual the homsets are CPOs and the composition of morphisms is a continuous function. Observe that given a CPO, it is always possible to extract a setoid from it by taking the symmetric closure of the preorder in the CPO as the equivalence relation of reference. This is a standard trick useful to tackle the certification of properties of extensional equivalences. The formalization of this concept is in the file `cpo_to_setoid.ma`.

Given A,B : Type[0] in Matita (roughly representing two sets), a *relation* is implemented as an inhabitant of A → B → Prop, where Prop represents the set of logic propositions. *Relational composition* of two relations r : A → B → Prop and s : B → C → Prop is $r \cdot s$ is the relation λa : A.λc : C.∃b : B.r a b ∧ s b c. This approach allows us to stay in a constructive set theory (as Matita-competent readers know). The setoid construction allows us to define the extensional collapse of these intensional definitions of sets. As expected, relations can

be structured in a CPO by taking the inclusion as its underlying pre-order and the generalized union as its lub operator (i.e. $\lambda c : (\mathbb{N} \to (A \to B \to Prop)).\lambda a : A.\lambda b : B.\exists n : \mathbb{N}.c\ n\ a\ b$). Relational composition is certified to be a monotonic and continuous function. It is also associative and admits a neutral element being the identity relation $\lambda a : A.\lambda a' : A.a = a'$.

We aim at certifying that the category **Pinj** of sets and relations, being injective and functional, provides a correct model for Janus. Our approach is very general. First we identify sufficient constraints on a property P on relations about our formal construction in Matita that are sufficient to provide a correct model. Second, we prove that **Pinj** can be obtained from a suitable instance of P. More precisely, if the property $P : (A \to B \to Prop) \to Prop$ on relations between A and B satisfies our sufficient constraints then our formal construction is:

1. a CPO-enriched category of relations satisfying the property P;
2. a symmetric monoidal category built on top of the category mentioned on point (1.), where the tensor product is the cartesian product;
3. a symmetric monoidal category built on top of the category mentioned on point (1.), where the tensor product is the disjoint union;
4. a dagger category built on top of the category mentioned on point (1.);
5. a traced monoidal category built on top of the category mentioned on point (3.), where the trace operator is particle-style;
6. is a category admitting a distributive law of cartesian product over disjoint union built on top of the categories mentioned on point (2.) and (3.).

In order to get (1.) we ask that the identity relation satisfies the property P (condition named id_ok), P is closed under logical equivalence (condition named cong_ok) and composition (condition named comp_preserve), the empty relation satisfies the property P (condition named good_bot) and given any chain $c : (\mathbb{N} \to (A \to B \to Prop))$ such that c n satisfies P for all $n : \mathbb{N}$ we have that the lub of c satisfies P (condition named good_lub). Each mathematical structure arising from relations satisfying the property P enjoying these constraints is actually a CPO-enriched category. In the formalization, all these conditions are given in the record good_rel_category in the file `rel.ma` of [13].

In order to get (2.) we ask that the property P is closed under the product of relations (condition named prod_ok) and that, both the isomorphisms describing the associativity law of the product, the left and right identity law of the product (the neutral element is the singleton set unit) and commutativity law of the product (i.e. the isomorphisms between $(A \times (B \times C))$ and $((A \times B) \times C)$ and between A and $(unit \times A)$ and between A and $(A \times unit)$ and between $(A \times B)$ and $(B \times A)$) satisfy the property P. In the formalization, all these conditions are given in the record good_rel_prod, in the file `rel_prod.ma` of [13].

In order to get (3.), we ask *mutatis mutandis* the same conditions asked for (2.). In the formalization, all these conditions are given in the record good_rel_sum, in the file `rel_sum.ma` of [13].

In order to get (4.), we ask that the property P is closed under the operation of inversion of relations. In the formalization, this condition is given in the record good_rel_dagger, in the file `rel_dagger.ma` of [13].

In order to get (5.), we ask that P is closed under application of the particle style trace operator. In the formalization, this condition is given in the record good_rel_trace, in the file `rel_trace.ma` of [13].

In order to get (6.) we ask that the isomorphisms describing the distributive law of product over disjoint union (i.e. the isomorphisms between $A \times (B+C)$ and $(A \times B) + (A \times C)$) satisfy the property P. In the formalization this condition is given in the record good_rel_distr, in the file `rel_distr.ma` of [13].

A relation r : A → B → Prop is *functional* when if r a b and r a b' hold then b = b'. A relation r : A → B → Prop is *injective* when if r a b and r a' b hold then a = a'. If P is "all relations are functional" or "all relations are injective" then, it turns out that P satisfies all above conditions, except the one expressed by good_rel_dagger. If P is "all relations are both functional and injective" then P satisfies all above conditions, including the one expressed by good_rel_dagger. So it is possible to build the CPO-enriched category of functional injective relations which is named Pinj. The formalization of the results here mentioned can be found in the file `pinj.ma`.

4.3 Graphical Language

After Selinger [16], we use the graphical notation of Figure 5 for monoidal categories to illustrate the semantics of Janus . We do not intend neither to formalize it in the proof assistant nor to introduce it formally, we introduce it only to give a geometrical intuition of the objects being the denotation of our Janus-programs to readers. The general idea of the graphical notation is that combinators are modeled by "wiring diagrams" or "circuits" and that values are modeled as "particles" that flow along the wires. Every wire of the graph is labeled with an object which corresponds to the type of the value (denoted with a particle) flowing along that wire. Evaluation is modeled by the flow of particles along the wires. In this paper we will use graphical conventions introduced by James and Sabry [9].

Every circuit is built up from basic atomic components that are connected together. The identity is a wire. Sum and products are parallel wires: in order to distinguish them graphically, we put a + symbol between wires labeled with objects that are summed. On two summed wires a value can reside in only one of the two, while on two non-summed wires, a values have to stay on both. Commutativity is represented by crisscrossing wires. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : A$ depending on what value inhabits the type $B + C$ as shown below. Factoring is the corresponding inverse operation. The trace operation is a looped circuit where the traced type U is shown as flowing backwards. Sum injections and quasi projections are represented as parallel summed wires in which one of them is isolated in order to denote the absence of values on that wire. Moreover an combinatorial representation of reversible update is given.

5 Interpretation

In this section we provide a denotational semantics of Janus statements in terms of injective functional relations, i.e. partial injective functions. Every Janus language construct is interpreted as a suitable composition of some categorical combinators introduced in the previous section. The goal is twofold: we aim at enforcing reversibility (since the interpretation is obtained by composition of some basic reversible functions) and we aim at making evident a connection between Janus and a framework of categorical reversible languages like those introduced in [9].

The interpretation of numerals and states is straightforward and it is given by the identity function. We denote with Σ the set of all states. From sake of simplicity, we stop to annotate states with the involved program (it is implicit in the context), but we still assume that the state is a function from all involved variables to natural numbers. An expression e of the language is interpreted into a functional relation $[\![e]\!]$ from states to ℕ in the following way.

$$
\begin{array}{lcl}
[\![\mathtt{n}]\!] & = & \{(\sigma, n) \mid \sigma \in \Sigma\} \\
[\![y]\!] & = & \{(\sigma, \sigma(y)) \mid \sigma \in \Sigma\} \\
[\![e_1 \circledast e_2]\!] & = & \{(\sigma, n) \mid (\sigma, n_1) \in [\![e_1]\!], (\sigma, n_2) \in [\![e_2]\!], n = n_1 [\![\circledast]\!] n_2\}
\end{array}
$$

Combinator	Graphical convention	Evaluation
identity relation on A		
Cartesian product. Tokens flow in parallel along the two wires		
Disjoint union. Only one token can flow in one of the two wires.		
Isomorphism between $A \times B$ and $B \times A$		
Isomorphism between $A \times unit$ and A		
Isomorphism between $A \times (B + C)$ and $(A \times B) + (A \times C)$		
Particle-style trace on the relation $f : A + U \to B + U$		
Reversible update built from a first argument injective operator $\odot$ and a functional relation f		
Left and right injection, left and right quasi-projections (i.e. the inverses of injections)		

Figure 5 Graphical convention for circuits.

Notice that the relation being the interpretation of an expression may not be injective.

▶ **Lemma 9.** *$\sigma, e \Downarrow_e n$ if and only if $(\sigma, n) \in [\![e]\!]$.*

Lemma 9 is certified in `rel_interpretation.ma`.

To interpret guards of the conditional and loop constructs, we give a representation of truth-values. The set of boolean values is the set of functional (injective) relations between $\mathbf{1}$ and $\mathbf{1} + \mathbf{1}$ which contains only the injections $\{inl, inr\}$ as elements. True is identified by inr while false is identified by inl. The idea is to use them in conjunction with the distributive law of product over sum in order to redirect the flow in the correct branch of the conditional.

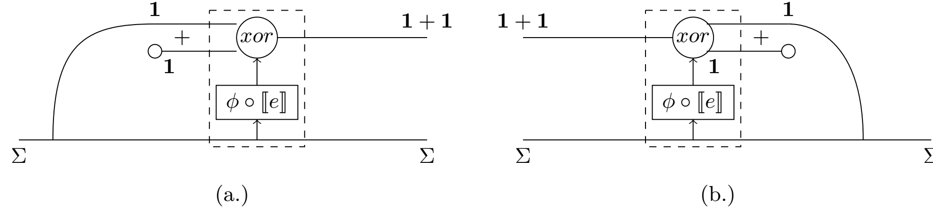


Figure 6 (a.) The testing morphism $test(e)_{\vec{x}}$. (b.) The assertion morphism $ass(e)_{\vec{x}}$.

Let $\phi : \mathbb{N} \to (\mathbf{1}+\mathbf{1})$ be the functional (non-injective) relation performing the canonical embedding of natural numbers into booleans, i.e. the functional relation mapping the natural number 0 into $inl(it)$ and all other non-zero values into $inr(it)$, where it is the unique element of the set $\mathbf{1}$. Let e be an expression. The *testing morphism* $test(e) : \Sigma \to ((\mathbf{1}+\mathbf{1}) \times \Sigma)$ generates a state annotated with the evaluation of the expression e. More formally

$$test(e) = \{(\sigma, (b, \sigma)) \mid \sigma \in \Sigma, (\sigma, n) \in [\![e]\!], (n, b) \in \phi\}$$

which is obviously an injective functional relation.

Its inverse is the *assertion morphism* $ass(e) = test(e)^\dagger : ((\mathbf{1}+\mathbf{1}) \times \Sigma) \to \Sigma$. It asserts whether in the annotated state the value of the additional variable coincides with the value of e (seen as a boolean). More formally:

$$ass(e) = \{((b, \sigma), \sigma) \mid \sigma \in \Sigma, (\sigma, n) \in [\![e]\!], (n, b) \in \phi\}$$

In Figure 6 we depicted the circuits realizing both tests and assertions.

Each statement is interpreted as a functional injective relation on states. Suppose that $\vec{id} = (id_1, \ldots, id_k)$ are the identifiers of the procedures defined. A functional environment is a k-pla $\varphi = (\varphi_1, \ldots, \varphi_k)$ where each $\varphi_i : \Sigma \to \Sigma$ is a morphism of **Pinj**, i.e. a partial injective function. We denote with $\mathbf{FEnv}_{\vec{id}}$ the set of all functional environments.

▶ **Definition 10** (Interpretation of statements)**.** Let $\varphi = (\varphi_1, \ldots, \varphi_k)$ be a functional environment, let s be a statement. $[\![s]\!]\varphi$ is a partial injective function on Σ i.e. $[\![s]\!]\varphi : \Sigma \to \Sigma$ and it is defined by structural induction on s as follows.

- $[\![z \odot\!= e]\!]\varphi = \{(\sigma[z \mapsto n], \sigma[z \mapsto m]) \mid \sigma \in \Sigma, (\sigma\restriction_z, n') \in [\![e]\!], m = n[\![\odot]\!]n'\}$
- $[\![\texttt{call}\ id_i]\!]\varphi = \varphi_i$
- $[\![\texttt{uncall}\ id_i]\!]\varphi = \varphi_i^\dagger$
- $[\![\texttt{skip}]\!]\varphi = id_\Sigma$
- $[\![s_1\ s_2]\!]\varphi = [\![s_2]\!]\varphi \circ [\![s_1]\!]\varphi$
- $[\![\texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2]\!]\varphi = ass(e_2) \circ \delta^\dagger \circ ((id_{\mathbf{1}} \times [\![s_2]\!]\varphi) + (id_{\mathbf{1}} \times [\![s_1]\!]\varphi)) \circ \delta \circ test(e_1)$
- $[\![\texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2]\!]\varphi = \mathrm{Tr}^{\Sigma}_{\Sigma,\Sigma}(f)$ where $f = ((\lambda^\times)^\dagger + ((\lambda^\times)^\dagger \circ [\![s_2]\!]\varphi)) \circ \delta \circ test(e_2) \circ [\![s_1]\!]\varphi \circ ass(e_1) \circ \delta^\dagger \circ (\lambda^\times + \lambda^\times)$

The interpretation of the assignment, the conditional and the loop deserves some explanations. The morphism denoting $z \odot\!= e$ is a reversible update that allows the embedding in Janus of possibly irreversible evaluation of expressions. Clearly $\odot$ is a first argument injective operator since addition and subtraction are. We recall that z cannot occur in e because of an explicit proviso in Definition 1. In the formalization the restriction of state σ to all its names but z is implemented by setting the value of z to 0. This operation is again inane since z does not appear in e. The morphisms denoting $\texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2$ evaluates the guard e_1 to annotate the state with a corresponding boolean. Then, according to the value of this

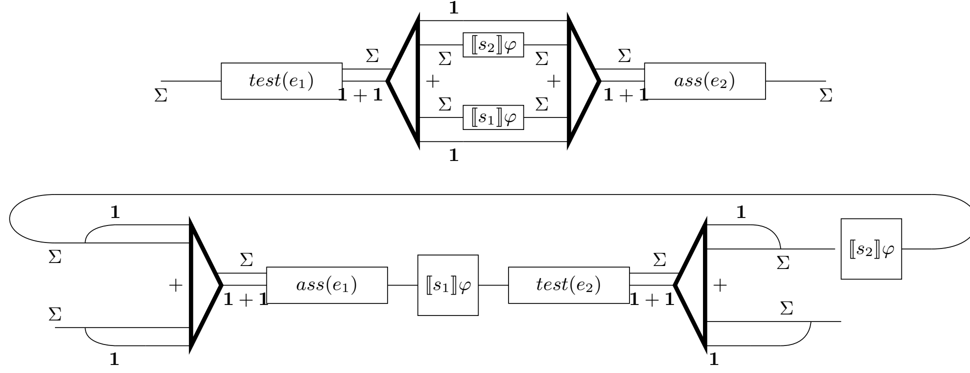


Figure 7 Graphical representation of the interpretation of conditional and loop.

annotation, the flow is redirected either to the circuit denoting s_1 or to the circuit denoting s_2 giving back the transformed state. Finally the annotation is tested to be the same as the value of the expression e_2: if the test is satisfied then the annotation is removed otherwise the operation is undefined. The morphism denoting `from` e_1 `do` s_1 `loop` s_2 `until` e_2 is obtained applying the trace operator on a function $f : (\Sigma + \Sigma) \to (\Sigma + \Sigma)$. This function is the composition of many mappings that we discuss step-by-step. First of all, f annotates the state with a boolean recording which side of the input sum was filled. Then it "asserts" whether the value of annotation corresponds to the value of the expression e_1 and if so, the annotation is removed (otherwise the function is undefined). Afterwards it evaluates the circuit denoting s_1. Then it evaluates the guard e_2 to annotate the state. Then the transformed state is redirected either on the left or on the right side of a sum according to the state annotation. In right case, the state is again transformed according to the evaluation of the circuit denoting s_2. A graphical representation of these circuits is given in Figure 7.

The set of morphisms of **Pinj** between two sets can be CPO-enriched with the set-theoretical order. For the same reason, also the set $\mathbf{FEnv}_{\vec{id}}$ endowed with the pointwise order is still a CPO. Following Winskell [17, p. 162, Lemma 9.3] one can prove that $[\![s]\!]$ is a continuous function. Thus, we can apply the fixpoint theorem to define the denotation of a Janus program containing procedure calls in terms of functional environments as explained in the following definition.

▶ **Definition 11** (Interpretation of programs). If $\mathtt{P} = \mathtt{procedure}\ id_1\ s_1, \ldots, \mathtt{procedure}\ id_k\ s_k$ is a program then, its interpretation is

$$[\![\mathtt{P}]\!] = \mathrm{fix}(\lambda\varphi.([\![s_1]\!]\varphi, \ldots, [\![s_k]\!]\varphi)) \in \mathbf{FEnv}_{\vec{id}}.$$

The following statements assert that the interpretation of statements is correct w.r.t. the operational semantics of statements.

▶ **Theorem 12.** *Let* P *be a program. Then* $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'$ *implies* $(\sigma_{\mathtt{P}}, \sigma') \in [\![s]\!]([\![\mathtt{P}]\!])$.

Proof. By induction on the derivation of the evaluation predicates. We develop only the case of loop of point (1.), the other cases being easy. Notice that in the case of `call` we need to use the definition of fixpoint together with the inductive hypothesis, while in the case of `uncall` we can use inductive hypothesis, Theorem 4 and definition of fixpoint. If $s =$`from` e_1 `do` s_1 `loop` s_2 `until` e_2 then we have that $\sigma_{\mathtt{P}}, e_1 \Downarrow_e n+1$, $\sigma_{\mathtt{P}}, s_1 \Downarrow_p \sigma'_{\mathtt{P}}$ and $\sigma'_{\mathtt{P}}, (e_1, s_1, s_2, e_2) \Downarrow_p \sigma'$. By applying Lemma 3 point (i) we know that there are $\sigma^0_{\mathtt{P}}, \ldots, \sigma^{2m}_{\mathtt{P}}$ where $m \in \mathbb{N}$ ($2m$ is the double of m), $\sigma^0_{\mathtt{P}} = \sigma'_{\mathtt{P}}$, $\sigma^{2m}_{\mathtt{P}} = \sigma'$, such that:

- $\sigma_{\mathtt{P}}^{2m}, e_2 \Downarrow_e \mathtt{n+1}$ and $\sigma_{\mathtt{P}}^{2i}, e_2 \Downarrow_e \mathtt{0}$, $\sigma_{\mathtt{P}}^{2i}, s_2 \Downarrow_p \sigma_{\mathtt{P}}^{2i+1}$ for $i < m$;
- $\sigma_{\mathtt{P}}^{2i+1}, e_1 \Downarrow_e \mathtt{0}$ for $i < m$, and $\sigma_{\mathtt{P}}^{2i+1}, s_1 \Downarrow_p \sigma_{\mathtt{P}}^{2i+2}$ for $i \leq m-2$.

It is not difficult to prove by induction that if $m = 0$ then $(\sigma_{\mathtt{P}}, \sigma') \in f_{ll}$ while $(\sigma_{\mathtt{P}}, \sigma') \in f_{rl} \circ f_{rr}^* \circ f_{lr}$ if $m \geq 0$ by observing that m is the number of iterations in the trace operator. This allows us to conclude. ◀

Let P be a program. Notice that by Theorem 12 in combination with Theorem 4, it follows that $\sigma_{\mathtt{P}}, s \Downarrow_p^{®} \sigma'$ implies $(\sigma_{\mathtt{P}}, \sigma') \in (\llbracket s \rrbracket(\llbracket \mathtt{P} \rrbracket))^{\dagger}$.

▶ **Theorem 13.** *Let* P *be a program. If* $(\sigma_{\mathtt{P}}, \sigma') \in \llbracket s \rrbracket(\llbracket \mathtt{P} \rrbracket)$ *then* $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'$.

Proof. We follow [17, Lemma 9.7 p. 167]. Let $\mathtt{P} = (\mathtt{procedure}\ id_1\ s_1, \ldots, \mathtt{procedure}\ id_k\ s_k)$ and define the functional environment φ as $(\sigma, \sigma') \in \varphi_i$ if $\sigma, s_i \Downarrow_p \sigma'$. Then we can prove by structural induction on s that if $(\sigma_{\mathtt{P}}, \sigma') \in \llbracket s \rrbracket(\varphi)$ then $\sigma_{\mathtt{P}}, s \Downarrow_p \sigma'$. Then we can prove easily that φ is a pre-fixpoint of the function defined in Definition 11, allowing us to conclude. ◀

The formalization of denotational semantics is given in the file `rel_interpretation.tex`. The interpretation is provided for all suitable instances of the abstract syntax.

```
let rec den_eval_stm (p : params) (p' : sem_params p)
(prog : stm p) (n : ℕ) on prog :
ℒ^{n}_{(Mor Pinj (list (const_type ...p)) (list (const_type ...p)))} →
(Mor Pinj (list (const_type ...p)) (list (const_type ...p))) :=...
```

The interpretation is defined by induction on the statement prog. It gives back a function from ℒ^{n}_{(Mor Pinj (list (const_type ...p)) (list (const_type ...p)))} (which is the implementation of $\mathbf{FEnv}_{\vec{id}}$) to an injective functional relation on states. We proved that the obtained function is both monotonic and continuous. So it is possible to define the interpretation using the fixpoint operator. The analogous of theorems 12 and 13 are certified in `correctness.ma` and `compl_thm.ma`.

6 Conclusions

This paper focuses on various aspects of the semantics of the reversible and imperative basic programming language Janus.

- On one side we focus on the operational semantics of Janus. The reason is standard: constructing efficient compilers and interpreters rely on well formalized and unambiguous operational semantics. We provide a syntax-driven operational semantics which is deterministic. This property does not hold for the first contributions to the operational semantics of Janus [20, 19, 18, 12]. The proposals in [20, 19, 18, 12] rest on a non-deterministic rule which describes a mechanism of "procedure un-call." For sake of completeness, however, we must recall that the authors of [20, 12] explain how to let their evaluation be deterministic by means of an external program transformation.
  A noteworthy by-product of our approach is that the well-known properties expressed by Corollary 5 can be formally expressed and proven in a friendly operational settings (indeed, the functionality and the injectivity of the evaluation are usually proved for Reversible Turing-Machines, for which we refer to Axelsen and Glück [2].)
- On the other side, we focus on a suitable categorical domain where correctly interpreting Janus. Obtaining full-abstraction is somewhat expected because Janus is not a higher-order language. The formalization of the interpretation, however, is not trivial because the semantic analysis of reversible languages is quite a new trend. In particular, we see

our analysis and its relationships with the work by James and Sabry [9, 8] as a starting point to deepen the knowledge about the reversible programming. So we are in the position to exploit a standard by-product of semantics, i.e. simplified tools to investigate equivalences among programs.

- Finally, our (meta)-proofs are certified by Matita.

References

1. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011. `doi:10.1007/978-3-642-22438-6_7`.
2. Holger Bock Axelsen and Robert Glück. What do reversible programs compute? In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6604 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 2011. `doi:10.1007/978-3-642-19805-2_4`.
3. Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible machine code and its abstract processor architecture. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*, volume 4649 of *Lecture Notes in Computer Science*, pages 56–69. Springer, 2007. `doi:10.1007/978-3-540-74510-5_9`.
4. C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Develop.*, 17(6):525–532, 1973. `doi:10.1147/rd.176.0525`.
5. Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2009. `doi:10.1007/978-3-642-03359-9_10`.
6. R. Blute and P. Scott. Category theory for linear logicians. In T. Ehrhard, J.-Y. Girard, P. Ruet, and P. Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Math. Soc. Lecture Note Series*, pages 3–64. Cambridge University Press, 2004. `doi:10.1017/cbo9780511550850.002`.
7. The Coq Development Team. The Coq proof assistant development manual, 2005. URL: `http://coq.inria.fr/doc/main.html`.
8. R. P. James and A. Sabry. Theseus: A high level language for reversible computing, 2014. URL: `http://www.cs.indiana.edu/~sabry/research.html`.
9. Roshan P. James and Amr Sabry. Isomorphic interpreters from logically reversible abstract machines. In Robert Glück and Tetsuo Yokoyama, editors, *Reversible Computation, 4th International Workshop, RC 2012, Copenhagen, Denmark, July 2-3, 2012. Revised Papers*, volume 7581 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2012. `doi:10.1007/978-3-642-36315-3_5`.
10. C. Lutz. Janus, a time reversible language. Letter to R. Landauer, 1986. URL: `http://www.peterhines.net/downloads/others/JANUS.html`.
11. A. Megacz. Category theory library for Coq. URL: `http://www.cs.berkeley.edu/~megacz/coq-categories/`.

12 Torben Æ. Mogensen. Partial evaluation of the reversible language janus. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, pages 23–32. ACM, 2011. doi:10.1145/1929501.1929506.

13 L. Paolini, M. Piccolo, and L. Roversi. Janus formalization in Matita. URL: http://www.di.unito.it/~piccolo/janus.zip.

14 Luca Paolini, Mauro Piccolo, and Luca Roversi. A class of reversible primitive recursive functions. *Electr. Notes Theor. Comput. Sci.*, 322:227–242, 2016. doi:10.1016/j.entcs.2016.03.016.

15 K. S. Perumalla. *Introduction to Reversible Computing*. Chapman & Hall/CRC Computational Science. CRC Press, 2013. doi:10.1201/b15719.

16 P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, Lect. Notes in Physics, pages 289–355. Springer, 2011. doi:10.1007/978-3-642-12821-9_4.

17 G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

18 Tetsuo Yokoyama. Reversible computation and reversible programming languages. *Electr. Notes Theor. Comput. Sci.*, 253(6):71–81, 2010. doi:10.1016/j.entcs.2010.02.007.

19 Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In Alex Ramírez, Gianfranco Bilardi, and Michael Gschwind, editors, *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*, pages 43–54. ACM, 2008. doi:10.1145/1366230.1366239.

20 Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 144–153. ACM, 2007. doi:10.1145/1244381.1244404.

21 M. Zorzi. On quantum lambda calculi: A foundational perspective. *Math. Struct. Comput. Sci.*, 26(7):1107–1195, 2016. doi:10.1017/s0960129514000425.

Functional Kan Simplicial Sets: Non-Constructivity of Exponentiation

Erik Parmann

Institutt for Informatikk, Universitetet i Bergen,
Postboks 7803, 5020 Bergen, Norway
eparmann@gmail.com

Abstract

Functional Kan simplicial sets are simplicial sets in which the horn-fillers required by the Kan extension condition are given explicitly by functions. We show the non-constructivity of the following basic result: if B and A are functional Kan simplicial sets, then A^B is a Kan simplicial set. This strengthens a similar result for the case of non-functional Kan simplicial sets shown by Bezem, Coquand and Parmann [TLCA 2015, v. 38 of LIPIcs]. Our result shows that – from a constructive point of view – functional Kan simplicial sets are, as it stands, unsatisfactory as a model of even simply typed lambda calculus. Our proof is based on a rather involved Kripke countermodel which has been encoded and verified in the Coq proof assistant.

1998 ACM Subject Classification F.4.1 Mathematical Logic and Formal Languages: Mathematical Logic

Keywords and phrases constructive logic, simplicial sets, semantics of simple types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.8

1 Introduction

In this paper, we show that the following theorem cannot be constructively proven in Intuitionistic Zermelo-Fraenkel (IZF) set theory.

▶ **Theorem 1** (classical). *If B and A are* functional *Kan simplicial sets, then A^B is a Kan simplicial set.*

We showed a similar result in [2], but for non-functional Kan simplicial sets. We will introduce (functional Kan) simplicial sets properly in the next section; for now, we will explain what is needed to characterize the crucial difference between functional and non-functional Kan simplicial sets.

A simplicial set consist of a family of sets $A[i], i \in \mathbb{N}$ with certain functions going between them, such that these functions satisfy the so-called *simplicial identities*. A Kan simplicial set is a simplicial set which is, in some sense, "full": it satisfies that, for every compatible n-tuple of elements in $A[n-1]$, there exists a compatible element in $A[n]$, using the meaning of "compatible" given in Definition 4.

Functional and non-functional Kan simplicial sets differ only in that the expression "for every...there exists..." is given a constructive interpretation. Although classical mathematics easily passes – by applying the axiom of choice – from elements existing to functions giving those elements, constructive mathematics does not take this so lightly. Constructively, all functional Kan simplicial sets are Kan simplicial sets, but the converse does not hold unless we adopt the axiom of choice, which – depending on the context – makes the logic classical [5].

© Erik Parmann;
licensed under Creative Commons License CC-BY

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 8; pp. 8:1–8:25

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

Theorem 1 is true classically, even without requiring that B is Kan (cf. [11, Appendix A, Theorem 3] or [10, Theorem 6.9] for a more modern approach), and plays an important role when using Kan simplicial sets as a model of type theory. The way we prove that Theorem 1 cannot be constructively proven is to show that the following constructive consequence of it cannot be constructively proven.

▶ **Theorem 2** (classical). *If B and A are functional Kan simplicial sets, then any edge in A^B can be reversed.*

In [2] we gave a Kripke counterexample to the constructive provability of Theorem 1 for *non-functional* Kan simplicial sets, showing that the appeal to classical logic in the proofs is essential. We did this by showing that certain graph-like, first-order structures can be constructively extended to Kan simplicial sets; and by using the corresponding version of Theorem 2 on the resulting simplicial set, we got that the graphs have a particular feature we can call *function-space edge reversal*. We then showed that the same class of structures does not have function-space edge reversal constructively.

Unfortunately, the countermodel only yields a *non-functional* Kan simplicial set, and we showed that if we assume explicit filler functions, then the simplicial sets induced by graphs always have function-space edge reversal. This shows that a simple tweak of the model is not sufficient; we need structures other than simple graphs. More precisely, we conjectured that a countermodel must be a hypergraph containing at least three dimensions of a simplicial set – not only points and edges, but also triangles – and this might significantly increase the complexity.

The present paper provides such a Kripke countermodel. In addition to the extra complexity of the new dimension, it also contains explicit filler functions respecting equality (the equality relation must be a congruence). Since this Kripke model equates elements (as the one in [2]), ensuring congruence turns out to be quite involved. To validate correctness, we have encoded and verified the model in the Coq [4] proof assistant.

The simplicial set A^B in Theorem 1 is not claimed to be functional Kan. This makes Theorem 1 weaker than if we had required A^B to be functional Kan, strengthening the non-provability result in this paper. It also means that the present paper properly generalizes [2].

In [3] it was shown that the homotopy equivalence of the fibers of a functional Kan fibration over a connected base cannot be proved constructively. The techniques used in the present paper are strongly inspired by [3].

The first section of [2] provides a introduction as to why Kan simplicial sets are interesting from a type-theoretical perspective. In short, Kan simplicial sets can be used to build a model of Martin-Löf Type Theory(MLTT) [8] with the homotopy theoretic interpretation of equality, and in this construction, Theorem 1 is important for the interpretation of function types. The results in [3] show that this construction, with equality interpreted as homotopy equivalences, is fundamentally non-constructive. This result closes one of the possible paths to finding a computational interpretation of the Univalence Axiom.

In [2] we showed that an even more fundamental part of the Kan simplicial set model of Type Theory – the interpretation of function types – is fundamentally non-constructive for *non-functional* Kan simplicial sets. The present paper shows the same for *functional* Kan simplicial sets. As a result the Kan simplicial set model is shown to presently be, from a constructive perspective, unsatisfactory as a model of even simply typed lambda calculus.

An alternative to interpreting type theory in Kan simplicial sets is to use cubical sets with a uniform Kan condition, as in [1]. The results of the present paper suggests that using

cubical sets with the uniform Kan condition is more promising than using Kan simplicial sets.

In addition to its type-theoretical implications, we think the result in this paper is valuable in its own right: we prove that a basic result in homotopy theory is not constructively provable.

The rest of the paper is organized as follows. In Section 2, we introduce simplicial sets and provide several examples of simplicial sets which will be used later. In Section 3, we define hypergraphs which we can constructively interpret as simplicial sets. In Section 4, we use that interpretation, in combination with Theorem 1, to formulate a theorem about graphs. In Section 5, we provide a Kripke model rejecting the constructive provability of this theorem. In Section 6, we explain how we used the proof assistant Coq to verify the Kripke model, before concluding in Section 7.

2 Simplicial Sets

We start by recalling the formal definition of simplicial sets and Kan simplicial sets from [2]. We also introduce functional Kan simplicial sets, before we provide a more intuitive explanation intended for those new to simplicial sets.

▶ **Definition 3** (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:

$$d_i d_j = d_{j-1} d_i \qquad \text{if } i < j \tag{1}$$

$$d_i s_j = s_{j-i} d_i \qquad \text{if } i < j \tag{2}$$

$$d_i s_j = id \qquad \text{for } i = j, j+1 \tag{3}$$

$$d_i s_j = s_j d_{i-1} \qquad \text{if } i > j+1 \tag{4}$$

$$s_i s_j = s_j s_{i-1} \qquad \text{if } i > j \tag{5}$$

An element of $A[i]$ is called an *i-simplex*. A *degenerate* element is any element $a \in A[i+1]$ in the image of a degeneracy map.

Note that a simplicial identity, such as, $d^n_i d^{n+1}_j = d^n_{j-1} d^{n+1}_i$, actually means

$$\forall x \in A[n+1].\ d^n_i(d^{n+1}_j(x)) = d^n_{j-1}(d^{n+1}_i(x)).$$

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, see [10, 7, 6].

▶ **Definition 4** (Functional 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.

If we have *functions* $\text{fill}^{n-1}_k : Y[n-1] \times \ldots Y[n-1] \to Y[n]$ giving the required n-simplex, then we say that Y is a *functional Kan simplicial set.*

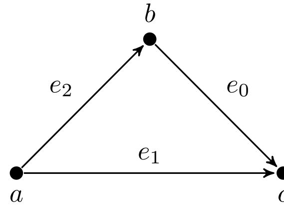


Figure 1 A single triangle.

A similar notion to functional Kan simplicial sets has been introduced by Thomas Nikolaus [12] as algebraic Kan complexes (AlgKan). The difference between AlgKan and functional Kan simplicial sets lies in the notion of morphisms (maps) in the corresponding category. For functional Kan simplicial sets, the morphisms are the same as between simplicial sets – they are natural transformations commuting with face and degeneracy maps – while for AlgKan, the morphisms must also send fillings to fillings. While the category of simplicial sets have a well-behaved exponential object, there is, to the author's knowledge, no good notion of exponentiation for AlgKan. Exponentiation is used to interpret the function type.

We will now provide some intuition of Kan simplicial sets by inspecting how they work in the lower dimensions. Readers who are already familiar with simplicial sets can skip ahead to Notation 5.

A simplicial set is an algebraic model of a topological space. It can also be seen as generalizations of reflexive directed multigraphs with countably infinite many dimensions. The first four dimensions of a simplicial set A can be viewed as points, edges, triangles and tetrahedrons. There are two functions going from edges to points – $d_0^1 : A[1] \to A[0]$ and $d_1^1 : A[1] \to A[0]$ – and we say that d_1^1 gives the startpoint and d_0^1 gives the endpoint of an edge. Likewise, there are three functions from triangles to edges, $d_0^2, d_1^2, d_2^2 : A[2] \to A[1]$ (giving the three edges a triangle consists of); and there are four similarly-named functions from tetrahedrons to triangles (giving the four triangles of the tetrahedron.)

It is important to note that elements are *not* necessarily equal when their components are; there can be several different edges going from one point to another, there can be several triangles having the exact same edges components, and so on.

Figure 1 shows a triangle t consisting of three edges, e_0, e_1 and e_2, with $d_i^2(t) = e_i$. Since the triangle is built up by three edges, we expect certain relations between the endpoints of those edges; for example, that the endpoint of e_2 matches the startpoint of e_0. This is precisely what is enforced by simplicial identity 1.

In addition to the face maps d_j^n, there are the degeneracy maps $s_j^n : A[n] \to A[n+1]$. These give degenerate elements: elements which are, so to say, constructed solely from a lower-dimensional object. For points, s_0^0 gives a reflexive edge on that point, and this edge is called the degenerate self-loop of the point. For edges, s_0^1 gives a triangle where two of the sides are the original edge and the third side is the degenerate self-loop of the startpoint of the edge, as shown in Figure 2. The function s_1^1 gives a triangle with the degenerate edge built on the endpoint. These properties are enforced by the simplicial identities 2-4, while simplicial identity 5 enforces the natural constraint that certain different ways of degenerating lead to the same degenerate element. The latter is most easily exemplified by first taking the degenerate edge on a point, and then either s_1^1 or s_0^1 of this edge, both provide the same degenerate triangle (with all three faces degenerate.)

Kan simplicial sets are special insofar as they are guaranteed to contain certain elements. One intuition is that they are simplicial sets satisfying the following condition: Whenever we

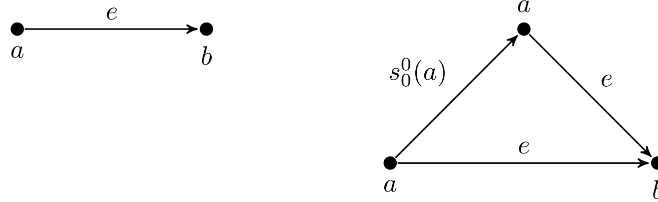


Figure 2 An edge e and the degenerate triangle $s_0^1(e)$.

Figure 3 An example of two compatible edges getting filled.

have $n+1$ elements in $A[n]$ such that we lack exactly one element in $A[n]$ to have all the faces of an element in $A[n+1]$, then we have this extra element in $A[n]$, and we have the element in $A[n+1]$ containing them all. For example, if we have two edges as in the left of Figure 3 where we lack only one edge to have all the edges of a triangle, then that edge exists (the edge g in the figure), *and* there is a triangle such that its faces are exactly e, f and g. For triangles, the Kan condition ensures that if we have three triangles such that we only lack a fourth to form a tetrahedron, then we have both that triangle and the tetrahedron. The relatively unwieldy condition on the sequence of elements $y_0, \ldots, y_{k-1}, y_{k+1}, \ldots, y_n$ given in Definition 4 express precisely that we lack exactly one element in $A[n]$ to have all the faces of an element in $A[n+1]$.

▶ **Notation 5.** *We introduce some notation for describing elements in the lower dimensions of a simplicial set A. We write $e : a \to b$ if $e \in A[1]$, $d_1^1(e) = a$, and $d_0^1(e) = b$ (note the direction). We write*

$$t : \begin{array}{c} e_2 \nearrow \;\; \searrow e_0 \\ \xrightarrow[e_1]{} \end{array}$$

for a triangle $t \in A[2]$ with $d_i^2(t) = e_i$. We say that a triangle t contains an edge e if $d_i^2 t = e$ for some $0 \leq i \leq 2$. The simplicial identities enforce that all triangles $t \in A[2]$ satisfy that, if $t : \begin{array}{c} e_2 \nearrow \;\; \searrow e_0 \\ \xrightarrow[e_1]{} \end{array}$, then $d_0^1 e_2 = d_1^1 e_0$, $d_1^1 e_2 = d_1^1 e_1$ and $d_0^1 e_0 = d_0^1 e_1$. This justifies writing

$$t : \begin{array}{ccc} & b & \\ e_2 \nearrow & & \searrow e_0 \\ a & \xrightarrow[e_1]{} & c \end{array}$$

for a triangle t with $d_i^1 t = e_i$ and $e_2 : a \to b$, $e_0 : b \to c$, and $e_1 : a \to c$.

Before moving on to some examples of simplicial sets, we show a property of Kan simplicial sets which will be important later: they have edge reversal.

▶ **Definition 6** (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 $d^1_1(f) = d^1_0(e)$ and $d^1_0(f) = d^1_1(e)$. We say that a simplicial set has *functional edge reversal* when we have a *function* giving, for every edge $e \in Y[1]$, the edge $f \in Y[1]$ as above.

▶ **Lemma 7.** *Functional Kan simplicial sets have functional edge reversal, and Kan simplicial sets have edge reversal.*

Proof of functional edge reversal. For all $e \in Y[1]$ where Y is a functional Kan fill-graph, let

$$f = d^2_0(\text{fill}^1_0(s^0_0(d^1_1(e)), e)).$$

If $e : a \to b$, then $s^0_0(d^1_1(e)) : a \to a$, and

$$\text{fill}^1_0(s^0_0(d^1_1(e)), e) : \quad \begin{array}{ccc} & b & \\ {}^{e}\nearrow & & \searrow \\ a & \xrightarrow[s^0_0(d^1_1(e))]{} & a \end{array} ,$$

so $d^2_0(\text{fill}^1_0(s^0_0(d^1_1(e)), e)) : b \to a$. ◀

Proof of non-functional version. As above, but instead of using the fill function we can only claim that the edge exists. ◀

2.1 Examples of Simplicial Sets

In this section we give some examples of simplicial sets which will be useful later. This section contains standard definitions, and is taken from [2].

2.1.1 Standard Simplicial k-Simplex Δ^k

Δ^k is the simplicial set with $\Delta^k[j]$ consisting of all non-decreasing sequences of numbers $0, \dots, k$ of length $j+1$. Equivalently, $\Delta^k[j]$ is the set of order-preserving functions $[j] \to [k]$, where $[i]$ denotes $0, \dots, i$ 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, $s^j_k(x_0 \dots x_k \dots x_{j+1}) = x_0 \dots x_k x_k \dots x_{j+1}$. The face map $d^j_k : \Delta^i[j] \to \Delta^i[j-1]$ deletes the k-th element. So, $d^j_k(x_0 \dots x_j) = x_0 \dots x_{k-1} x_{k+1} \dots x_j$.

2.1.2 The k-Horns Λ^k_j

Λ^k_j is the j'th horn of the standard k-simplex Δ^k, and defined by $\Lambda^k_j[n] = \{f \in \Delta^k[n] \mid [k] - \{j\} \nsubseteq Im(f)\}$. 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 functional Kan extension condition from Definition 4 for a simplicial set Y can also be formulated as: we have a dependent function $\text{fill}(k, j, F)$ such that $\text{fill}(k, j, F) : \Delta^k \to Y$ extends $F : \Lambda^k_j \to Y$, for any k, j, F.

2.1.3 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.)

2.1.4 Function Spaces

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 map $d_k[i] : Y^X[i] \to Y^X[i-1]$ 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 a non-decreasing 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 $(d_k F)[j](a_0 \dots a_j, x) = F[j](d_k^* a_0 \dots d_k^* a_j, x)$. 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 we define $(s_k F)[i](a, x) = F[i](s_k^* a, x)$.

3 Hypergraphs as Simplicial Sets

We now define graph classes corresponding to (functional Kan) simplicial sets. The meaning of "corresponding" is made precise in Lemma 11; these are graphs which can be constructively interpreted as (functional Kan) simplicial sets.

▶ **Definition 8** (Reflexive hypergraph). A *reflexive hypergraph* consists of C_2, C_1, C_0, d_0^1, d_1^1, d_0^2, d_1^2, d_2^2, s, s_0, s_1 where C_0 is a set of points, C_1 a set of edges and C_2 a set of triangles. For $d_i^1 : C_1 \to C_0$, d_1^1 is the *source* and d_0^1 the *target* function, and $s(c)$ is a degenerate self-loop. Each $d_i^2 : C_2 \to C_1$ is an *edge* function, giving an edge of a triangle; and each $s_i : C_1 \to C_2$ is a function mapping an edge to a degenerate triangle. These are all subject to different restrictions, which are given below.

We will use the notation introduced in Notation 5 for reflexive hypergraphs as well. We require that all triangles $t \in C_2$ satisfy that, if $t : \begin{smallmatrix} & e_2 \nearrow \; \searrow e_0 & \\ & \xrightarrow[e_1]{} & \end{smallmatrix}$, then $d_0^2 e_2 = d_1^2 e_0$, $d_1^2 e_2 = d_1^2 e_1$ and $d_0^2 e_0 = d_0^2 e_1$, justifying writing

$$t: \quad \begin{array}{ccc} & b & \\ {}^{e_2}\nearrow & & \searrow^{e_0} \\ a & \xrightarrow[e_1]{} & c \end{array}$$

for a triangle t with $d_i^2 t = e_i$, $e_2 : a \to b$, $e_0 : b \to c$, and $e_1 : a \to c$.

We require $d_i^1(s(c)) = c$ for all $c \in C_0$, and we require that the functions s_0 and s_1 satisfy the following: for all $e : a \to b$, $s_0(e) : \begin{array}{ccc} & a & \\ {}^{s(a)}\nearrow & & \searrow^{e} \\ a & \xrightarrow[e]{} & b \end{array}$ and $s_1(e) : \begin{array}{ccc} & b & \\ {}^{e}\nearrow & & \searrow^{s(b)} \\ a & \xrightarrow[e]{} & b \end{array}$, and that for all v, $s_0(s(v)) = s_1(s(v)) : \begin{array}{ccc} & v & \\ {}^{s(v)}\nearrow & & \searrow^{s(v)} \\ v & \xrightarrow[s(v)]{} & v \end{array}$.

The definition above is not much more than a specialization of Definition 3 to the first three dimensions, so it is not particularly surprising that we can extend any reflexive hypergraph to a simplicial set. The method is a natural extension of the one used in [2] and [3], but extended in such a way that triangles are not necessarily equal when they have equal faces.

▶ **Definition 9** ($S(C)$)**.** Given a reflexive hypergraph C, we can construct a simplicial set $S(C)$ in the following way:

$S(C)[0] = C_0$, $S(C)[1] = C_1$, $S(C)[2] = C_2$ and $S(C)[n]$, for $n \geq 3$, consisting of all tuples of the form $(u_0, \ldots, u_n; \ldots, e_{ij}, \ldots; \ldots, t_{ijl}, \ldots)$ such that

$e_{ij} : u_i \to u_j$ in C_1 for all $0 \leq i < j \leq n$, and

$$t_{ijl} : \begin{array}{ccc} & u_j & \\ {}^{e_{ij}}\nearrow & & \searrow^{e_{jl}} \\ u_i & \xrightarrow[e_{il}]{} & u_l \end{array} \quad \text{in } C_2 \text{ for all } 0 \leq i < j < l \leq n.$$

The maps d^n_k in $S(C)$ are defined for $n > 3$ by removing from the input tuple

$(u_0, \ldots, u_n; \ldots, e_{ij}, \ldots; \ldots, t_{ijl}, \ldots)$

the point u_k, all edges e_{ik} and e_{kj}, and all triangles containing either of those edges. For $n = 3$, if we do this on an element q in $S(C)[3]$, the result is a tuple $(u_0, u_1, u_2; e_{01}, e_{02}, e_{12}; t_{012})$ containing only one triangle t_{012}, and we let $d^3_k(q) = t_{012}$.

The maps s^n_k in $S(C)$ for $n > 3$ are defined by duplicating the point u_k in the tuple $(u_0, \ldots, u_n; \ldots, e_{ij}, \ldots; \ldots, t_{ijl}, \ldots)$, adding an edge $e_{k(k+1)} = s(u_k)$, and duplicating edges and incrementing indices of edges as appropriate. In addition, we add $t_{k(k+1)j} = s_0(e_{kj})$ for every e_{kj} and $t_{ik(k+1)} = s_1(e_{ik})$ for every e_{ik}, and duplicating triangles and incrementing indices as needed. For $n = 3$, we are given a triangle $t : \begin{array}{ccc} & b & \\ {}^{e_2}\nearrow & & \searrow^{e_0} \\ a & \xrightarrow[e_1]{} & c \end{array}$, and we perform the above construction on the tuple $(a, b, c; e_2, e_1, e_0; t)$.

This completes the construction of the simplicial set $S(C)$, and it is fairly easy to see that it satisfies the simplicial identities.

Similarly, we can specialize Definition 4 to the first three dimensions, giving us a first-order structure we can extend to a functional Kan simplicial set.

▶ **Definition 10** (Kan fill-hypergraph)**.** A *Kan fill-hypergraph* is a reflexive hypergraph where we have functions $\text{fill}_{1,i} : C_1 \times C_1 \to C_2$ for $0 \leq i \leq 2$ and $\text{fill}_{2,i} : C_2 \times C_2 \times C_2 \to C_2$ for $0 \leq i \leq 3$, satisfying the following requirements.

For all $e_1, e_2 \in C_1$, $\text{fill}_{1,i} : C_1 \times C_1 \to C_2$ must satisfy:

If $e_1 : a \to c$ and $e_2 : a \to b$, then $\text{fill}_{1,0}(e_1, e_2) : \begin{array}{ccc} & b & \\ {}^{e_2}\nearrow & & \searrow \\ a & \xrightarrow[e_1]{} & c \end{array}$.

If $e_0 : b \to c$ and $e_2 : a \to b$, then $\text{fill}_{1,1}(e_0, e_2) : \begin{array}{ccc} & b & \\ {}^{e_2}\nearrow & & \searrow^{e_0} \\ a & \longrightarrow & c \end{array}$.

If $e_0 : b \to c$ and $e_1 : a \to c$, then $\text{fill}_{1,2}(e_0, e_1) : \begin{array}{ccc} & b & \\ \nearrow & & \searrow^{e_0} \\ a & \xrightarrow[e_1]{} & c \end{array}$.

For all $t_1, t_2, t_3 \in C_2$, $\mathrm{fill}_{2,i} : C_2 \times C_2 \times C_2 \to C_2$ must satisfy:

If t_1: $\overset{e_1 \nearrow\, c\, \searrow e_5}{a \xrightarrow[e_3]{} d}$, t_2: $\overset{e_2 \nearrow\, b\, \searrow e_4}{a \xrightarrow[e_3]{} d}$, and t_3: $\overset{e_2 \nearrow\, b\, \searrow e_0}{a \xrightarrow[e_1]{} c}$, then $\mathrm{fill}_{2,0}(t_1, t_2, t_3)$: $\overset{e_0 \nearrow\, c\, \searrow e_5}{b \xrightarrow[e_4]{} d}$.

If t_1: $\overset{e_0 \nearrow\, c\, \searrow e_5}{b \xrightarrow[e_4]{} d}$, t_2: $\overset{e_2 \nearrow\, b\, \searrow e_4}{a \xrightarrow[e_3]{} d}$, and t_3: $\overset{e_2 \nearrow\, b\, \searrow e_0}{a \xrightarrow[e_1]{} c}$, then $\mathrm{fill}_{2,1}(t_1, t_2, t_3)$: $\overset{e_1 \nearrow\, c\, \searrow e_5}{a \xrightarrow[e_3]{} d}$.

If t_1: $\overset{e_0 \nearrow\, c\, \searrow e_5}{b \xrightarrow[e_4]{} d}$, t_2: $\overset{e_1 \nearrow\, c\, \searrow e_5}{a \xrightarrow[e_3]{} d}$, and t_3: $\overset{e_2 \nearrow\, b\, \searrow e_0}{a \xrightarrow[e_1]{} c}$, then $\mathrm{fill}_{2,2}(t_1, t_2, t_3)$: $\overset{e_2 \nearrow\, b\, \searrow e_4}{a \xrightarrow[e_3]{} d}$.

If t_1: $\overset{e_0 \nearrow\, c\, \searrow e_5}{b \xrightarrow[e_4]{} d}$, t_2: $\overset{e_1 \nearrow\, c\, \searrow e_5}{a \xrightarrow[e_3]{} d}$, and t_3: $\overset{e_2 \nearrow\, b\, \searrow e_4}{a \xrightarrow[e_3]{} d}$, then $\mathrm{fill}_{2,3}(t_1, t_2, t_3)$: $\overset{e_2 \nearrow\, b\, \searrow e_0}{a \xrightarrow[e_1]{} c}$.

Note that the above requirements can be translated from the visual description above into first-order logic, and this is the intended reading of the above definition. For example, the requirement for $\mathrm{fill}_{1,0}$ is:

$$\forall e_1, e_2 \in C_1, d_1^1(e_1) = d_1^1(e_2) \to d_1^2(\mathrm{fill}_{1,0}(e_1, e_2)) = e_1 \wedge d_2^2(\mathrm{fill}_{1,0}(e_1, e_2)) = e_2.$$

▶ **Lemma 11.** *If C is a Kan fill-hypergraph, we can extend $S(C)$ to a functional Kan simplicial set.*

Proof. We have to define the functions $\mathrm{fill}_{n,k}$ in $S(C)$. We write $\mathrm{fill}^S_{n,k}$ for the functions in $S(C)$ and $\mathrm{fill}^C_{n,k}$ for the functions in C.

If $n = 0$, we simply let $\mathrm{fill}^S_{0,i} : C_0 \to C_1$ be s. If $n = 1$ we put $\mathrm{fill}^S_{1,i} = \mathrm{fill}^C_{1,i}$, and it is easy to see that $\mathrm{fill}^C_{1,i}$ satisfies the requirements given in Definition 4.

If $n = 2$, we use $\mathrm{fill}^C_{2,k}$; but we cannot use it directly, as it provides an element of $S(C)[2]$, not an element in $S(C)[3]$ as needed. Instead, we use it to construct an element of $S(C)[3]$. We show the procedure for $k = 1$; the other cases proceed analogously. We are given $t_0, t_2, t_3 \in S(C)[2] = C_2$, such that $d_i t_j = d_{j-1} t_i$ for any $i < j \leq 3$ with $i \neq 1$ and $j \neq 1$, and we proceed to construct an $r \in S(C)[3]$ such that $d_i^2 r = t_i$ for $i = 0, 2, 3$. Expanding this and naming the resulting edges gives the equations

$$\begin{aligned} d_0 t_2 &= d_1 t_0 = e_4 \\ d_0 t_3 &= d_2 t_0 = e_0 \\ d_2 t_3 &= d_2 t_2 = e_2 \end{aligned}$$

Naming the three remaining edges e_1, e_3 and e_5 we get that $t_0 :$ $\overset{e_0 \nearrow\, c\, \searrow e_5}{b \xrightarrow[e_4]{} d}$, $t_2 :$ $\overset{e_2 \nearrow\, b\, \searrow e_4}{a \xrightarrow[e_3]{} d}$,

and $t_3 :$ $\overset{e_2 \nearrow\, b\, \searrow e_0}{a \xrightarrow[e_1]{} c}$, giving $\mathrm{fill}^C_{2,1}(t_1, t_2, t_3) :$ $\overset{e_1 \nearrow\, c\, \searrow e_5}{a \xrightarrow[e_3]{} d}$.

Observe that the tuple

$$r = (a, b, c, d\ ;\ e_2, e_1, e_3, e_0, e_4, e_5\ ;\ t_3, t_2, \mathrm{fill}^C_{2,1}(t_0, t_2, t_3), t_0)$$

satisfies the form given in Definition 9 for elements in $S(C)[3]$, so $r \in S(C)[3]$, while also satisfying $d_i r = t_i$ for $i = 0, 2, 3$, giving us the value for $\mathrm{fill}^C_{2,1}(t_0, t_2, t_3)$.

For higher values of n, we observe that any sequence of tuples applicable to $\mathrm{fill}^S_{n,k}$ contains all the components of a satisfying element; it is just a matter of extracting the right triangles, edges and points from the arguments. ◀

4 Function Spaces between Hypergraphs

In this section, we identify a class of functions between reflexive hypergraphs which we can extend to edges in the function space between the corresponding simplicial sets. This enables us to formulate a constructive consequence of Theorem 1 which we will give a countermodel to in the next section.

Remember that the function space between two simplicial sets A and B has as the ith dimension $Hom_S(\Delta^i \times A, B)$, so its edges are elements of $Hom_S(\Delta^1 \times A, B)$.

▶ **Definition 12** ($\Delta^i|_m$)**.** We define $\Delta^i|_m$ to be the family of m sets given by removing from Δ^i every dimension larger than m. The functions s^j and d^{j+1} for $0 \le j < m$ are kept as is, while they are discarded for $j > m$.

▶ **Definition 13.** For reflexive hypergraphs X and Y, we say that an $F : \Delta^i|_2 \times X \to Y$ is *commuting* when it commutes with d^n and s^m for $0 \le m \le 1 \le n \le 2$, where both work on the product $\Delta^i|_2[n] \times X[n]$ component-wise, similar to Cartesian products of simplicial sets as described in Section 2.1.3.

▶ **Lemma 14.** *For reflexive hypergraphs X and Y, any commuting $F : \Delta^1|_2 \times X \to Y$ can be extended to an edge in $S(Y)^{S(X)}$.*

Proof. We need to extend F to an $F' \in Hom_S(\Delta^1 \times S(X), S(Y))$; that is, a family of functions $F'[n] : (\Delta^1 \times S(X))[n] \to S(Y)[n]$ for $n \in \mathbb{N}$ commuting with s^i_j and d^i_j. For $0 \le n \le 2$, we let $F'[n] = F$. For $n > 2$, any input to $F'[n]$ will have the form

$$(0^a 1^b, (x_0, \ldots, x_n; \ldots e_{ij}, \ldots; \ldots, t_{ijl}, \ldots))$$

such that $a + b = n + 1$. We define the function $gt_a : \mathbb{N} \to \{0, 1\}$ as $gt_a(x) = 1$ if $x \ge a$ and $gt_a(x) = 0$ otherwise, and we then let $F'[n]$ map the input to the tuple

$$(F(0, x_0), \ldots, F(0, x_{a-1}), F(1, x_a) \ldots, F(1, x_{a+b-1}); \ldots e'_{ij}, \ldots; \ldots t'_{ijl}, \ldots),$$

where e'_{ij} and t'_{ijl} are given by $e'_{ij} = F(gt_a(i)gt_a(j), e_{ij})$ and $t'_{ijl} = F(gt_a(i)gt_a(j)gt_a(l), t_{ijl})$.

It should be clear that this map does indeed commute with d_i and s_j. It holds in the lower dimensions since we assume F to be commuting. It also holds in the higher dimensions since we apply F uniformly to every element in the tuple; if we remove a particular element and then apply F, we get the same result as if we first apply F to all elements in the tuple and then remove the particular element. ◀

Now we are ready to formulate the constructive consequence of Theorem 1 – which is essentially Theorem 2, reformulated as a property of Kan fill-hypergraphs.

▶ **Theorem 15** (Classical)**.** *For any Kan fill-hypergraphs X and Y, for any commuting $F : \Delta^1|_2 \times X \to Y$ we can find a commuting $F^- : \Delta^1|_2 \times X \to Y$ such that for all $p \in X[0]$, $l \in X[1]$ and $t \in X[2]$ we have*

$$F^-(0,p) = F(1,p) \qquad F^-(1,p) = F(0,p)$$
$$F^-(00,l) = F(11,l) \qquad F^-(11,l) = F(00,l)$$
$$F^-(000,t) = F(111,t) \qquad F^-(111,t) = F(000,t)$$

Proof. We first extend F to an edge in $S(Y)^{S(X)}$ by Lemma 14 and note that, by Lemma 11, $S(Y)$ is a functional Kan simplicial set. We then apply the classical Theorem 2, giving that $S(Y)^{S(X)}$ is a Kan simplicial set, enabling us to reverse the edge by Lemma 7, giving an F^- in $S(Y)^{S(X)}[1]$ satisfying $d_0(F) = d_1(F^-)$ and $d_1(F) = d_0(F^-)$. We discard every dimension of F^- above 2. Being an element of $Hom_S(\Delta^1 \times S(X), S(Y))$ means that F^- commutes, and expanding the definition of d_i from Section 2.1.4 we calculate:

$$F^-(0,p) = F^-(d_1^*(0),p) = d_1(F^-)(0,p) = d_0(F)(0,p) = F(d_0^*(0),p) = F(1,p),$$

$$F^-(1,p) = F^-(d_0^*(0),p) = d_0(F^-)(0,p) = d_1(F)(0,p) = F(d_1^*(0),p) = F(0,p)$$

The other dimensions go the same way, showing that F^- is as desired. ◀

Observe that the only non-constructive step in the proof of Theorem 15 was the application of Theorem 2.

The reasoning in our constructive proofs can be formalized in IZF (Intuitionistic Zermelo-Fraenkel set theory), so IZF proves that Theorem 1 implies Theorem 2, and that Theorem 2 implies Theorem 15. We also have that if IZF proves Theorem 15, then Theorem 15 holds in any Kripke model. This result has been further elaborated in Section 6 of [2]. For this, it is vital that Theorem 15 can be expressed in first-order logic. So finally, by giving a Kripke model falsifying Theorem 15, we show that IZF cannot prove Theorem 1, and we will provide exactly such a model in the next section.

5 Kripke Countermodel

In this section we present a Kripke model falsifying the first-order sentence representing Theorem 15. Recall that a Kripke model is a partially ordered set of classical models – often called states or days – where the domains and relations are monotone, and a formula holds in a state if it holds (classically) at that state and all of its successors. For further elaboration, see [9].

We have two classical models in our Kripke model, and we will call them "day 1" and "day 2", with day 1 before day 2. Each consists of an X-part and a Y-part, both satisfying the requirements on Kan fill-hypergraphs. In addition, our Kripke model contains a commuting family of functions $F : \Delta^1|_2[n] \times X[n] \to Y[n]$ for $0 \le n \le 2$.

The only change from day 1 to day 2 is the interpretation of equivalence; we equate more elements in day 2. As we equate elements, we have to ensure that all functions respect equivalence; that is, that they send equal elements to equal elements. In addition, by equating elements, more elements may satisfy the antecedents in Definition 10, and we need to ensure that these formulae remain satisfied for our X and Y-parts to be valid Kan fill-hypergraphs.

We first present X and Y in day 1. We then present the family of functions $F : \Delta^1|_2[n] \times X[n] \to Y[n]$ for $0 \le n \le 2$, before we present X and Y day 2.

5.1 Day 1

The two first dimensions of X and Y are both shown below, with triangles and fill functions further described below. Different edges in the figure are non-equated, and an edge f in the figure from a to b represents an edge f in the model with $d_0^1(f) = b$ and $d_1^1(f) = a$.

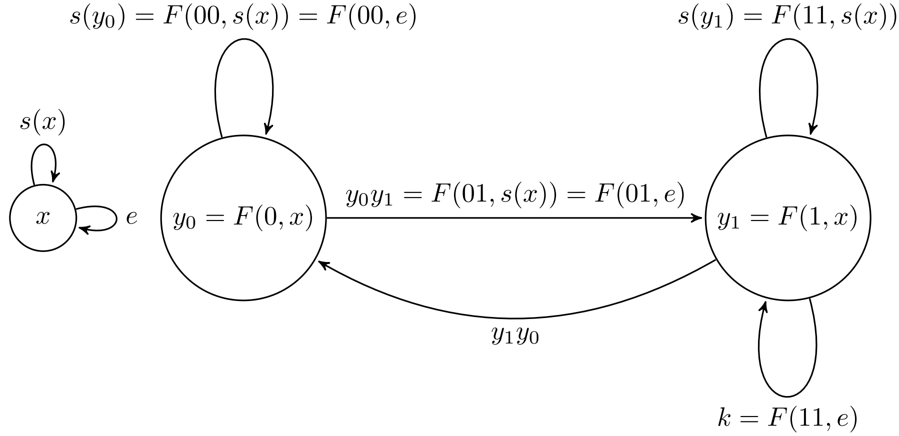


Figure 4 Kripke (counter)model for edge reversal, day 1.

5.1.1 Triangles in X, Day 1

$X[2]$ consists of exactly all eight combinations of $s(x)$ and e as faces. So we have the following triangles, where the names are given as the concatenation of d_i^2 of the triangle for $0 \leq i \leq 2$.

$sss :$ $s(x)$ $s(x)$ $s(x)$ $\qquad$ $sse :$ e $s(x)$ $s(x)$

$\vdots$ $\qquad$ $\vdots$

$ees :$ $s(x)$ e e $\qquad$ $eee :$ e e e

The functions s_i^1 are forced by the simplicial identities to be:

$$s_0^1(s(x)) = s_1^1(s(x)) = sss$$

$$s_0^1(e) = ees : \quad s(x) \quad e \quad e$$

$$s_1^1(e) = see : \quad e \quad s(x) \quad e$$

Finally, we define the functions $\text{fill}_{1,i}^X : X[1] \times X[1] \to X[2]$ and $\text{fill}_{2,i}^X : X[2] \times X[2] \times X[2] \to X[2]$. For the former, we have a choice; the two arguments determine two of the edges in the resulting triangle, but the third edge can be either $s(x)$ or e. We chose, rather arbitrary, for it to always be $s(x)$, resulting in only one possible triangle. For $\text{fill}_{2,i}^X$, there are no options; its three arguments determine the three edges of the resulting triangle, describing it completely.

5.1.2 Triangles in Y, Day 1

We construct $Y[2]$ in two stages. First, it consists of all compatible triples of edges from $Y[1]$. That is, for all edges e_0, e_2, e_1 such that $e_0 : b \to c$, $e_1 : a \to c$, and $e_2 : a \to b$, we add

exactly one triangle $e_0_e_1_e_2 : \begin{array}{ccc} & b & \\ e_2 \nearrow & & \searrow e_0 \\ a & \xrightarrow[e_1]{} & c \end{array}$ to $Y[2]$. This result in 18 triangles, and as with $X[2]$ we name then according to their faces. This means that we have a triangle

$$y_1y_0_y_1y_0_s(y_1) : (y_1, y_1, y_0; s(y_1), y_1y_0, y_1y_0) \in Y[1],$$

and we will call it T_{de}. The second stage of the construction is simply to add an additional triangle T_{fi} of the form

$$T_{\text{fi}} : (y_1, y_1, y_0; s(y_1), y_1y_0, y_1y_0)$$

to $Y[2]$. It is no coincidence that T_{fi} has identical faces to T_{de}; this enables us to use T_{fi} as a substitute of T_{de} in certain situations. All triangles of $Y[2]$ at day 1 are listed in Table 1 in Appendix B.

We define $s_i^1 : Y[1] \to Y[2]$ before concluding with the fill functions. In most cases, there is only one compatible triangle to which s_i^1 can map, forcing

$$s_0^1(y_0y_1) = y_0y_1_y_0y_1_s(y_0) : \begin{array}{ccc} & y_0 & \\ s(y_0) \nearrow & & \searrow y_0y_1 \\ y_0 & \xrightarrow[y_0y_1]{} & y_1 \end{array}$$

$$s_1^1(y_0y_1) = s(y_1)_y_0y_1_y_0y_1 : \begin{array}{ccc} & y_1 & \\ y_0y_1 \nearrow & & \searrow s(y_1) \\ y_0 & \xrightarrow[y_0y_1]{} & y_1 \end{array}$$

and similar for the other edges. The exception is $s_0^1(y_1y_0)$, which we can map to both T_{de} and T_{fi}. We set

$$s_0^1(y_1y_0) = T_{\text{de}},$$

and this concludes the definition of s_i^1. The complete listing of s_i^1 can be found in Table 2 in Appendix B.

Finally, we define the functions $\text{fill}_{1,i} : Y[1] \times Y[1] \to Y[2]$ and $\text{fill}_{2,i} : Y[2] \times Y[2] \times Y[2] \to Y[2]$. They are, in the same way as s_i^1, in most cases determined by the fact that there is exactly one compatible triangle. There are some exceptions. For $\text{fill}_{1,i}$, we have certain inputs where we can choose if the third edge in the resulting triangle is $s(y_1)$ or k, and in those cases we choose for it to be $s(y_1)$; and when there is a choice between mapping to T_{fi} and T_{de}, we map to T_{fi}. If we want $\text{fill}_{1,i}$ to be total, we map every non-compatible pair of edges to $y_0y_0_y_0y_0_y_0y_0$.

For $\text{fill}_{2,i}$, there are inputs where the output can be either T_{de} or T_{fi}, and in these cases we map to T_{fi}. In addition, we have the choice of how to map the triangles which do *not* satisfy the requirements in Definition 10. Equating elements in day 2 will have the effect of making more triangles satisfy the requirements for $\text{fill}_{2,i}$, so the choices we make now must be compatible with this. We show this for $\text{fill}_{2,1}$; the other cases are similar. Recall that the requirement for $\text{fill}_{2,1}$ is:

If t_1: $\begin{array}{ccc} & c & \\ e_0 \nearrow & & \searrow e_5 \\ b & \xrightarrow[e_4]{} & d \end{array}$, t_2: $\begin{array}{ccc} & b & \\ e_2 \nearrow & & \searrow e_4 \\ a & \xrightarrow[e_3]{} & d \end{array}$, and t_3: $\begin{array}{ccc} & b & \\ e_2 \nearrow & & \searrow e_0 \\ a & \xrightarrow[e_1]{} & c \end{array}$, then $\text{fill}_{2,1}(t_1, t_2, t_3)$: $\begin{array}{ccc} & c & \\ e_1 \nearrow & & \searrow e_5 \\ a & \xrightarrow[e_3]{} & d \end{array}$.

Given three triangles t_1, t_2, t_3, when there is a triangle with the signature $\begin{array}{ccc} & c & \\ e_1 \nearrow & & \searrow e_5 \\ a & \xrightarrow[e_3]{} & d \end{array}$ –

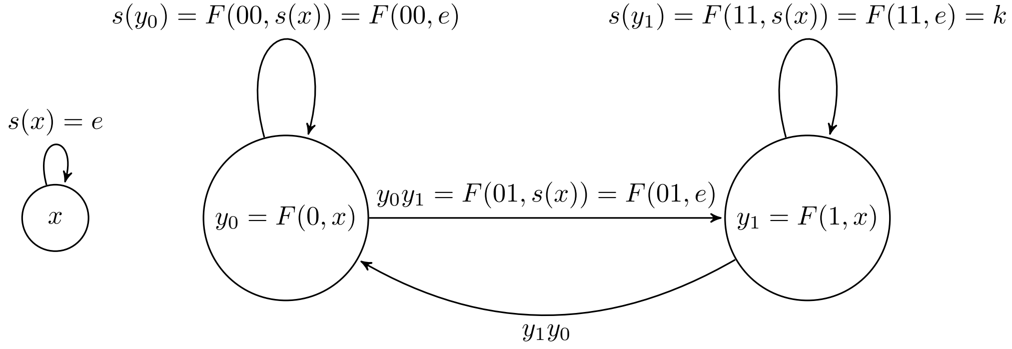


Figure 5 Kripke (counter)model for edge reversal, day 2.

where $e_5 = d_0^2(t_1)$, $e_1 = d_1^2(t_3)$ and $e_3 = d_1^2(t_2)$ – we map $\text{fill}_{2,1}(t_1, t_2, t_3)$ to it (and to T_{fi} if there is a choice between T_{fi} and T_{de}.) If there is no such triangle, we map $\text{fill}_{2,1}(t_1, t_2, t_3)$ to $y_0y_0_y_0y_0_y_0y_0$.

5.1.3 *F*

We define the commuting family $F : \Delta^1|_2[n] \times X[n] \to Y[n]$ for $0 \leq n \leq 2$. Both $F_0 : \Delta^1|_2[0] \times X[0] \to Y[0]$ and $F_1 : \Delta^1|_2[1] \times X[1] \to Y[1]$ are completely given in Figure 4, only $F_2 : \Delta^1|_2[2] \times X[2] \to Y[2]$ is in need of further description. But it is completely locked by the requirement that it should commute with d_i (since, besides T_{fi} and T_{de}, we have exactly one triangle per compatible triple of edges). Note that F_1 does not map anything to the edge y_1y_0, since this goes from $F(1, x)$ to $F(0, x)$; similarly, F_2 maps to neither T_{de} nor T_{fi}, relieving us from having to choose which of these to map to.

5.2 Day 2

Moving from day 1 to day 2, we equate a number of elements, but make no changes otherwise. This means that we only need to ensure that the defined functions still respect equality, and verify that the filling-conditions in Definition 10 remain satisfied.

First, we present the equating for the first two levels of both X and Y; following this we equate triangles. In $X[1]$, we set $s(x) = e$; and in $Y[1]$, we set $s(y_1) = y_1y_1 = k$. Other edges are as they were in day 1. The first two dimensions are shown in Figure 5. In $X[2]$, we equate every triangle, leaving us with only one degenerate triangle. Having only one point, one edge and one triangle means that all functions with both domain and co-domain inside X trivially respect equality.

In $Y[2]$, we equate exactly those triangles which have identical faces after the equation of elements in $Y[1]$, except that we keep T_{de} distinct from any other triangle. Since the only edge-equation in $Y[1]$ was $y_1y_1 = k$, we only equate triangles containing this edge. The complete list of equated triangles can be found in Table 3; the list of the remaining, non-equated triangles can be found in Table 4. Note that we can keep T_{de} distinct from every other triangle, since T_{de} is only in the image of s_0^1 (it is, crucially, *not* in the image of $\text{fill}_{1/2,i}$), and then as the value of $s_0^1(y_1y_0)$. The edge y_1y_0 is not equated with any other edge, thus we are not forced through s_0^1 to equate T_{de} with any other triangle.

This clearly does not enforce any further equations in $Y[1]$. We also claim that this keeps all functions consistent. It should be clear from Figure 5 that both $F_0 : \Delta^1|_2[0] \times X[0] \to Y[0]$ and $F_1 : \Delta^1|_2[1] \times X[1] \to Y[1]$ remain consistent. $F_2 : \Delta^1|_2[2] \times X[2] \to Y[2]$ is consistent

since F commutes with d_i and as $F_1 : \Delta^1|_2[1] \times X[1] \to Y[1]$ is consistent, so any two triangles mapped to in $Y[2]$ (with the same argument from $\Delta^1|_2[2]$) now must have identical faces, giving that they are also equal.

It is important for $\text{fill}_{1/2,i}$ that T_{de} is not in their image, and that all other triangles are equal exactly when they have equal faces. So if $e_1 = e_1'$ and $e_2 = e_2'$, then $\text{fill}_{1,i}(e_1, e_2)$ and $\text{fill}_{1,i}(e_1', e_2')$ have identical faces, giving $\text{fill}_{1,i}(e_1, e_2) = \text{fill}_{1,i}(e_1', e_2')$.

For $\text{fill}_{2,i}$, observe that the output is a triangle containing one edge from each of its inputs. So if we have $t_1 = t_1'$, $t_2 = t_2'$, and $t_3 = t_3'$ then $\text{fill}_{2,i}(t_1, t_2, t_3)$ will have identical faces to $\text{fill}_{2,i}(t_1', t_2', t_3')$, so they are also equal. Also observe that all requirements in Definition 10 remain satisfied. Three triangles which now satisfy one of the antecedents are already sent to a satisfying triangle by the way we defined $\text{fill}_{2,i}$.

Finally, we observe that all the simplicial identities are still satisfied, since we have not changed s_i/d_i, and the equivalence relation is monotone.

5.3 Non-existence of F^-

We will now see that we cannot consistently define the commuting reverse function $F^- : \Delta^1|_2 \times X \to Y$, prescribed by Theorem 15, such that for all $p \in X[0]$, $l \in X[1]$ and $t \in X[2]$ we have:

$$
\begin{aligned}
F^-(0, p) &= F(1, p) & F^-(1, p) &= F(0, p) \\
F^-(00, l) &= F(11, l) & F^-(11, l) &= F(00, l) \\
F^-(000, t) &= F(111, t) & F^-(111, t) &= F(000, t).
\end{aligned}
$$

Assume towards a contradiction that there is such an F^-. We will expand on the values of $F^-(001, eee)$ and $F^-(001, sss)$. Applying commutativity of the face maps in combination with the above requirements, we see that F^- would have to satisfy the following two requirements:

$$d_2^2(F^-(001, eee)) = F^-(d_2^2(001), d_2^2(eee)) = F^-(00, e) = F(11, e) = k$$

$$d_0^2 d_0^2(F^-(001, eee)) = F^-(d_0^2 d_0^2(001), d_0^2 d_0^2(eee)) = F^-(1, x) = F(0, x) = y_0.$$

This forces $F^-(001, eee) = y_1 y_0_y_1 y_0_k$, since this is the only triple in $Y[2]$ satisfying the above requirements.

Since $sss = s_0^1(s(x))$ and $001 = s_0(01)$, commutativity with s_0 forces $F^-(001, sss) = s_0^1(F^-(01, s(x)))$. The only compatible edge for $F^-(01, s(x))$ is $y_1 y_0$. Since $s_0^1(y_1 y_0) = T_{\text{de}}$ we get $F^-(001, sss) = T_{\text{de}}$.

In day 2, we have that $eee = sss$; but we also have that $T_{\text{de}} \neq y_1 y_0_y_1 y_0_k$, showing that there can be no consistent F^- satisfying the desired requirements.

6 Formal Verification of the Kripke Model

The Kripke model from Section 5 is quite complex. Verifying that it has the properties claimed is not a trivial task; it is for this reason that we have formally verified it using the Coq proof assistant.[1] Using Coq in this way – essentially, as a model checker – is not very common, but it worked quite well. The reason is the nature of our model checking problem;

[1] See `https://github.com/epa095/funKanPowCounterModel-coq` for the Coq script.

we have a model with relatively few states and we want to prove many properties. Encoding the model in Coq makes it easy to read the statements of the theorems, verifying that they indeed prove what we need to prove.

The model checking is divided into two separate parts. The first part is the encoding of the Kripke model and the second part consists of the proofs that the encoding satisfies the desired properties.

Appendix A contains the complete list of Theorems (sans proofs) and definitions.

6.1 Encoding the Kripke Model

The encoding of the Kripke model from Section 5 is quite direct. First, we define that there are two days:

```
Inductive Days := d1 | d2.
```

We then define each of the sorts in the Kripke model – points, edges and triangles – for both X and Y, as finite inductive types. Here we show it for Y; the definitions are similar for X.

```
Inductive VY := y0 | y1.
Inductive EdgeY := y0y0 | y0y1 | y1y0 | y1y1| k.

Inductive TriangleY :=
| y0y0_y0y0_y0y0
| y0y1_y0y1_y0y0
⋮
| fi
| de.
```

The names of the triangles are given by $d_0_d_1_d_2$ of the triangle, so as an example $d_2(y0y1_y0y1_y0y0) = y0y0$. We then encode the functions $d^1_0, d^1_1, d^2_0, d^2_1, d^2_2, s, s_0, s_1$, which we name sY, d0Y, d1Y, se0Y, se1Y, dp0Y, dp1Y, and dp2Y. They are all defined explicitly for all possible inputs, as shown in the following example.

```
Function sY (v1 :VY) := match v1 with
                        | y0 ⇒ y0y0
                        | y1 ⇒ y1y1
                        end.
```

We are using an explicitly defined equivalence relation for each sort. In day 1, two elements are equal exactly when they have the same constructors. In day 2, the edge $y1y1$ is equated with k; otherwise, edges are equal when they have the same constructor. Two triangles are equal when their faces are equal, except de, which is only equal to itself:

```
Function eqTriangleY (day : Days)(t1 t2 :TriangleY) :=
 match day with
   | d1 ⇒ sameConstructorTriangleY t1 t2
   | d2 ⇒
     match t1,t2 with
       | de,de ⇒ true
       | de,_ | _,de ⇒ false
       | _,_ ⇒ andb (eqEdgeY d2 (dp0Y t1) (dp0Y t2))
               (andb (eqEdgeY d2 (dp1Y t1) (dp1Y t2))
                     (eqEdgeY d2 (dp2Y t1) (dp2Y t2)))
     end
 end.
```

Finally, we define the filler functions, $\text{fill}_{1,i}$: `EdgeY` $\times$ `EdgeY` $\to$ `TriangleY` and $\text{fill}_{2,i}$: `TriangleY` $\times$ `TriangleY` $\times$ `TriangleY` $\to$ `TriangleY`. We implement them according to Section 5.1.2, making sure that e.g $\text{fill}_{1,i}(y1y0, y1y1) = fi$ instead of de, and similarly for the other inputs. For $\text{fill}_{2,i}$, we make use of a function `edgesToTriangle` which maps three edges $d0, d1, d2$ to a triangle t, such that $dp0Y(t) = d0$, $dp1Y(t) = d1$ and $dp2Y(t) = d2$. There are two things to notice. First, `edgesToTriangle` maps to fi and not de when it has a choice; and since it needs to be total, it also maps every non-compatible triple of edges to the triangle $y0y0_y0y0_y0y0$. The implementation of $\text{fill}_{2,i}$ consists of just picking out the right edges from its argument, as exemplified by $\text{fill}_{2,0}$:

```
Function fill20Y (t1 t2 t3 :TriangleY) := (edgesToTriangle (dp0Y t1)
                                                           (dp0Y t2)
                                                           (dp0Y t3)).
```

This concludes the definition of Y and its associated functions. The encoding of X is slightly simpler, since it has exactly one triangle per compatible triple of edges, eliminating the fi vs de distinction. In addition, we encode $\Delta^1|_2[n]$, $0 \leq n \leq 2$ with face and degeneracy maps in the same explicit way, with `Delta10`, `Delta11` and `Delta12` being points, edges and triangles respectively. This lets us define the function $F : \Delta^1|_2[n] \times X[n] \to Y[n]$ for $0 \leq n \leq 2$ from Section 5.1.3 explicitly, ending the definition of the Kripke model.

```
Function Fv (delt:Delta10) (v:VX) := ...
Function Fe (delt:Delta11) (e:EdgeX) :=...
Function Ft (delt:Delta12) (t:TriangleX) :=
```

6.2 Verifying the Encoded Model

The encoding above is straightforward, but tedious to verify. In this section, we will explain how we used Coq to show that the encoded model has the properties desired.

A useful feature for doing this is Coq's type classes. These enables us to define a collection of properties – parameterized on types and functions – and give several instantiations of those types and functions, ensuring that each instance satisfies the properties specified in the type class.

We define two type classes: one for reflexive hypergraphs, and another for Kan fill-hypergraphs. We show that X, Y and Delta are instances of the first class, and that X and Y are instances of the second. In what follows, we will describe the properties encoded by these type classes.

We begin by defining what it means for a function `eqFun: Days → A → A → bool` to be an equivalence, before going on to define what it means for a unary, binary and ternary function to respect equality on its domain and co-domain.

```
Definition EquivalenceFun {A:Type}(eqFun: Days → A → A → bool):=
ReflexiveFun eqFun ∧ SymetricFun eqFun ∧ TransitiveFun eqFun.

Definition binaryFunctionRespectsEquality {Domain CoDomain:Type}
        (eqDomain: Days → Domain → Domain → bool)
        (eqCoDomain: Days → CoDomain → CoDomain → bool)
        (function: Domain→  Domain → CoDomain):= ...
```

We now define the class giving the basic properties of X, Y and Delta. It expresses that all of the face and degeneracy maps respect equality, that the equality functions are monotone equalities, and that the simplicial identities hold between the face and degeneracy maps.

The whole class can be found in Appendix A. Giving Y, X and Delta as instances leaves the properties that need to be shown as obligations, which are all pretty straightforward to close.

The next step consists of verifying the filling functions. We construct a type class once more, this time parameterized on a member of the previously defined type class and all seven fill functions. The type class specifies that all of the fill functions must respect equality, in addition to encoding each of the properties the fill functions must respect, as the following example for $\text{fill}_{1,0}$:

```
fill10Prop: forall (d:Days)(e1 e2:Edges),
                  ((eqV d (d1E e1) (d1E e2))=true)→
                      (eqEdges d (d1T (fill10 e1 e2)) e1)=true ∧
                      (eqEdges d (d2T (fill10 e1 e2)) e2)=true
```

Finally, we verify that our encoding of the family of functions F is correct, and that the argumentation from Section 5.3 holds. We start by formalizing the notion of a family of functions $F : \Delta^1|_2[n] \times X[n] \to Y[n]$ commuting with both the face and degeneracy maps in X, Delta and Y according to Definition 13, before showing that F, as encoded above, commutes.

We then encode that an inverse of F is a family $F^- : \Delta^1|_2 \times X \to Y$ such that, for all $p \in X[0]$, $l \in X[1]$ and $t \in X[2]$, we have

$$
\begin{aligned}
F^-(0,p) &= F(1,p) & \qquad F^-(1,p) &= F(0,p)\\
F^-(00,l) &= F(11,l) & \qquad F^-(11,l) &= F(00,l)\\
F^-(000,t) &= F(111,t) & \qquad F^-(111,t) &= F(000,t).
\end{aligned}
$$

We finish by showing that all commuting reverses of F must satisfy $F^-(001, eee) = y1y0_y1y0_k$ and $F^-(001, sx_sx_sx) = de$, and that these two images remain distinct in both days.

7 Conclusion

In this paper, we provided a model showing that we cannot constructively prove that the Kan property is preserved under exponentiation. This means, from a constructive perspective, that Kan simplicial sets are currently unsatisfactory as models of simply typed lambda calculus. The result was shown for a strong interpretation of Kan simplicial sets requiring explicit functions for the horn fillings, closing the gaps from previous work on a similar problem for Kan simplicial sets *without* explicit filler functions. The model has been encoded and verified using the Coq proof assistant.

References

1. M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In R. Matthes and A. Schubert, editors, *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES 2013*, volume 26 of *Leibniz Int. Proc. in Inf.*, pages 107–128. Dagstuhl Publishing, 2014. doi:10.4230/lipics.types.2013.107.
2. M. Bezem, T. Coquand, and E. Parmann. Non-constructivity in Kan simplicial sets. In T. Altenkirch, editor, *Proc. of 13th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2015*, volume 38 of *Leibniz Int. Proc. in Inf.*, pages 92–106. Dagstuhl Publishing, 2015. doi:10.4230/lipics.tlca.2015.92.
3. Marc Bezem and Thierry Coquand. A Kripke model for simplicial sets. *Theor. Comput. Sci.*, 574:86–91, 2015. doi:10.1016/j.tcs.2015.01.035.

4 The Coq Development Team. The Coq reference manual, version 8.4, 2012. URL: `https://coq.inria.fr/distrib/8.4/refman/`.

5 R. Diaconescu. Axiom of choice and complementation. *Proc. Amer. Math. Soc.*, 51(1):176–178, 1975. `doi:10.1090/s0002-9939-1975-0373893-x`.

6 G. Friedman. An elementary illustrated introduction to simplicial sets. arXiv preprint 0809.4221, 2008. URL: `https://arxiv.org/abs/0809.4221`.

7 P. Goerss and J. F. Jardine. *Simplicial Homotopy Theory*, volume 174 of *Progress in Mathematics*. Birkhäuser, 1999. Reprinted in Modern Birkhäuser Classics, 2009. `doi:10.1007/978-3-0348-8707-6`.

8 C. Kapulkin, P. LeFanu Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. arXiv preprint 1211.2851, 2012. URL: `https://arxiv.org/abs/1211.2851`.

9 S. Kripke. Semantical analysis of intuitionistic logic I. In M. Dummett and J.N. Crossley, editors, *Formal Systems and Recursive Functions*, volume 40 of *Studies in Logic and the Foundations of Mathematics*, pages 92–130. North Holland, 1965. `doi:10.1016/s0049-237x(08)71685-9`.

10 J. P. May. *Simplicial Objects in Algebraic Topology*. Chicago Lectures in Mathematics. University of Chicago Press, 2nd edition, 1993.

11 J. C. Moore. Algebraic homotopy theory. Lectures at Princeton, 1956. URL: `http://faculty.tcu.edu/gfriedman/notes/aht1.pdf`.

12 T. Nikolaus. Algebraic models for higher categories. *Indag. Math.*, 21(1–2):52–75, 2011. `doi:10.1016/j.indag.2010.12.004`.

A Theorems Proved in Coq

```
Definition ReflexiveFun {A:Type}(eqFun: Days → A → A → bool):=
  forall (d:Days)(el:A), (eqFun d el el) = true.

Definition SymmetricFun {A:Type}(eqFun: Days → A → A → bool):=
  forall (d:Days)(elem1 elem2:A),
    ((eqFun d elem1 elem2)=true)→ (eqFun d elem1 elem2)=true.

Definition TransitiveFun {A:Type}(eqFun: Days → A → A → bool):=
  forall (d:Days)(elem1 elem2 elem3:A),
    ((eqFun d elem1 elem2)=true ∧ (eqFun d elem2 elem3)=true) →
    (eqFun d elem1 elem3)=true.

Definition EquivalenceFun {A:Type}(eqFun: Days → A → A → bool):=
  ReflexiveFun eqFun ∧ SymetricFun eqFun ∧ TransitiveFun eqFun.


Definition unaryFunctionRespectsEquality {Domain CoDomain:Type}
           (eqDomain: Days → Domain → Domain → bool)
           (eqCoDomain: Days → CoDomain → CoDomain → bool)
           (function: Domain→ CoDomain):=
  forall d:Days, forall (elem1 elem2: Domain),
    (eqDomain d elem1 elem2)=true → (eqCoDomain d (function elem1) (function elem2))=true.

Definition binaryFunctionRespectsEquality {Domain CoDomain:Type}
           (eqDomain: Days → Domain → Domain → bool)
           (eqCoDomain: Days → CoDomain → CoDomain → bool)
           (function: Domain→  Domain → CoDomain):=
  forall d:Days, forall (elem1 elem1p elem2 elem2p: Domain),
    (eqDomain d elem1 elem1p)=true ∧ (eqDomain d elem2 elem2p)=true
    → (eqCoDomain d (function elem1 elem2) (function elem1p elem2p))=true.
```

```
Definition tertaryFunctionRespectsEquality {Domain CoDomain:Type}
           (eqDomain: Days → Domain → Domain → bool)
           (eqCoDomain: Days → CoDomain → CoDomain → bool)
           (function: Domain→  Domain→  Domain → CoDomain):=
  forall d:Days, forall (elem1 elem2 elem3 elem1p elem2p elem3p: Domain),
    ((eqDomain d elem1 elem1p)=true ∧
     (eqDomain d elem2 elem2p)=true ∧
     (eqDomain d elem3 elem3p)=true)
    → (eqCoDomain d (function elem1 elem2 elem3) (function elem1p elem2p elem3p))=true.
```

```
Definition EqFunctionMonotone {Domain:Type}
       (eq: Days → Domain → Domain → bool):=
       forall (elem1 elem2: Domain),
         (eq d1 elem1 elem2)=true → (eq d2 elem1 elem2)=true.
```

```
Class twoDayKripke (Points Edges Triangles :Type)
:=  {
    sP :  Points → Edges;
    d0E :  Edges → Points;
    d1E :  Edges → Points;
    s0E :  Edges → Triangles;
    s1E :  Edges → Triangles;
    d0T :  Triangles → Edges;
    d1T :  Triangles → Edges;
    d2T :  Triangles → Edges;
    eqV :  Days → Points → Points → bool;
    eqEdges : Days → Edges → Edges → bool;
    eqTriangles : Days → Triangles → Triangles → bool;
    eqVisEq : EquivalenceFun eqV;
    eqEdgessisEq : EquivalenceFun eqEdges;
    eqTrianglesisEq : EquivalenceFun eqTriangles;
    _ :  EqFunctionMonotone eqV;
    _ :  EqFunctionMonotone eqEdges;
    _ :  EqFunctionMonotone eqTriangles;
    sPRespectsEq : unaryFunctionRespectsEquality eqV eqEdges sP;
    d0ERespectsEq : unaryFunctionRespectsEquality eqEdges eqV d0E;
    d1ERespectsEq : unaryFunctionRespectsEquality eqEdges eqV d1E;
    s0ERespectsEq : unaryFunctionRespectsEquality eqEdges eqTriangles s0E;
    s1ERespectsEq : unaryFunctionRespectsEquality eqEdges eqTriangles s1E;
    d0TRespectsEq : unaryFunctionRespectsEquality eqTriangles eqEdges d0T;
    d1TRespectsEq : unaryFunctionRespectsEquality eqTriangles eqEdges d1T;
    d2TRespectsEq : unaryFunctionRespectsEquality eqTriangles eqEdges d2T;
    (* Simplicial identity 1 *)
    _: forall t: Triangles, d0E(d1T(t)) = d0E(d0T(t));
    _: forall t: Triangles, d0E(d2T(t)) = d1E(d0T(t));
    _: forall t: Triangles, d1E(d2T(t)) = d1E(d1T(t));
    (* Simplicial identity 2 *)
    _: forall e: Edges, d0T(s1E(e)) = sP(d0E(e)) ;
    (* Simplicial identity 3 *)
    _: forall p: Points, d0E(sP(p)) = p;
    _: forall p: Points, d1E(sP(p)) = p;
    _: forall e: Edges,  d0T(s0E(e)) = e;
    _: forall e: Edges,  d1T(s0E(e)) = e;
    _: forall e: Edges,  d1T(s1E(e)) = e;
    _: forall e: Edges,  d2T(s1E(e)) = e;
    (* Simplicial identity 4 *)
    _ :  forall e:Edges, d2T(s0E(e)) = sP(d1E(e));
    (* Simplicial identity 5 *)
    _: forall p: Points, s1E(sP(p)) = s0E(sP(p))
 }.
```

```
Class fillableModel {Points Edges Triangles:Type} {m: twoDayKripke Points Edges Triangles}
:= {
   fill10: Edges → Edges → Triangles;
   fill11: Edges → Edges → Triangles;
   fill12: Edges → Edges → Triangles;
   fill20: Triangles → Triangles → Triangles → Triangles;
   fill21: Triangles → Triangles → Triangles → Triangles;
   fill22: Triangles → Triangles → Triangles → Triangles;
   fill23: Triangles → Triangles → Triangles → Triangles;

   fill10RespectEquality: binaryFunctionRespectsEquality eqEdges eqTriangles fill10;
   fill11RespectEquality: binaryFunctionRespectsEquality eqEdges eqTriangles fill11;
   fill12RespectEquality: binaryFunctionRespectsEquality eqEdges eqTriangles fill12;
   fill20RespectsEqualty: tertaryFunctionRespectsEquality eqTriangles eqTriangles fill20;
   fill21RespectsEqualty: tertaryFunctionRespectsEquality eqTriangles eqTriangles fill21;
   fill22RespectsEqualty: tertaryFunctionRespectsEquality eqTriangles eqTriangles fill22;
   fill23RespectsEqualty: tertaryFunctionRespectsEquality eqTriangles eqTriangles fill23;

   fill10Prop: forall (d:Days)(e1 e2:Edges), ((eqV d (d1E e1) (d1E e2))=true)→
                                               (eqEdges d (d1T (fill10 e1 e2)) e1)=true ∧
                                               (eqEdges d (d2T (fill10 e1 e2)) e2)=true;

   fill11Prop: forall (d:Days)(e1 e2:Edges), ((eqV d (d1E e1) (d0E e2))=true)→
                                               (eqEdges d (d0T (fill11 e1 e2)) e1)=true ∧
                                               (eqEdges d (d2T (fill11 e1 e2)) e2)=true;

   fill12Prop: forall (d:Days)(e0 e1:Edges), ((eqV d (d0E e0) (d0E e1))=true)→
                                               (eqEdges d (d0T (fill12 e0 e1)) e0)=true ∧
                                               (eqEdges d (d1T (fill12 e0 e1)) e1)=true;

   fill20Prop: forall (d:Days)(t1 t2 t3:Triangles), (eqEdges d (d1T t1) (d1T t2))=true ∧
                                                     (eqEdges d (d2T t1) (d1T t3))=true ∧
                                                     (eqEdges d (d2T t2) (d2T t3))=true →
                                              ((eqEdges d (d0T t1) (d0T (fill20 t1 t2 t3)))=true ∧
                                               (eqEdges d (d0T t2) (d1T (fill20 t1 t2 t3)))=true ∧
                                               (eqEdges d (d0T t3) (d2T (fill20 t1 t2 t3)))=true);
   fill21Prop: forall (d:Days)(t1 t2 t3:Triangles),
                 (eqEdges d (d1T t1) (d0T t2))=true ∧
                 (eqEdges d (d2T t1) (d0T t3))=true ∧
                 (eqEdges d (d2T t2) (d2T t3))=true →
                 ((eqEdges d (d0T t1) (d0T (fill21 t1 t2 t3)))=true ∧
                  (eqEdges d (d1T t2) (d1T (fill21 t1 t2 t3)))=true ∧
                  (eqEdges d (d1T t3) (d2T (fill21 t1 t2 t3)))=true);

   fill22Prop: forall (d:Days)(t1 t2 t3:Triangles),
                 (eqEdges d (d0T t1) (d0T t2))=true ∧
                 (eqEdges d (d2T t1) (d0T t3))=true ∧
                 (eqEdges d (d2T t2) (d1T t3))=true →
                 ((eqEdges d (d1T t1) (d0T (fill22 t1 t2 t3)))=true ∧
                  (eqEdges d (d1T t2) (d1T (fill22 t1 t2 t3)))=true ∧
                  (eqEdges d (d2T t3) (d2T (fill22 t1 t2 t3)))=true);

   fill23Prop: forall (d:Days)(t1 t2 t3:Triangles),
                 (eqEdges d (d0T t1) (d0T t2))=true ∧
                 (eqEdges d (d1T t1) (d0T t3))=true ∧
                 (eqEdges d (d1T t2) (d1T t3))=true →
                 ((eqEdges d (d2T t1) (d0T (fill23 t1 t2 t3)))=true ∧
                  (eqEdges d (d2T t2) (d1T (fill23 t1 t2 t3)))=true ∧
                  (eqEdges d (d2T t3) (d2T (fill23 t1 t2 t3)))=true)

  }.
```

```
Definition FCommutesS (Fpoint: Delta10 → VX → VY)
          (FEdge: Delta11 → EdgeX → EdgeY)
          (FTriangle: Delta12 → TriangleX → TriangleY):=
  (forall (deltp:Delta10) (p:VX), FEdge (s deltp) (sX p) = sY (Fpoint deltp p))
  ∧ (forall (d:Days) (delt:Delta11) (e:EdgeX),
       eqTriangleY d (FTriangle (s0 delt) (se0X e))( se0Y (FEdge delt e)) = true)
  ∧ (forall (d:Days) (delt:Delta11) (e:EdgeX),
       eqTriangleY d (FTriangle (s1 delt) (se1X e))( se1Y (FEdge delt e)) = true).

Definition FCommutesD (Fpoint: Delta10 → VX → VY)
          (FEdge: Delta11 → EdgeX → EdgeY)
          (FTriangle: Delta12 → TriangleX → TriangleY):=
  (forall (delt:Delta11) (e:EdgeX), Fpoint (dv0 delt) (d0X e) = d0Y (FEdge delt e)) ∧
  (forall (delt:Delta11) (e:EdgeX), Fpoint (dv1 delt) (d1X e) = d1Y (FEdge delt e)) ∧
  (forall (delt:Delta12) (e:TriangleX), FEdge (dp2 delt) (dp2X e) = dp2Y (FTriangle delt e)) ∧
  (forall (delt:Delta12) (e:TriangleX), FEdge (dp1 delt) (dp1X e) = dp1Y (FTriangle delt e)) ∧
  forall (delt:Delta12) (e:TriangleX), FEdge (dp0 delt) (dp0X e) = dp0Y (FTriangle delt e).

Definition FCommutes (Fpoint: Delta10 → VX → VY)
          (FEdge: Delta11 → EdgeX → EdgeY)
          (FTriangle: Delta12 → TriangleX → TriangleY):=
  (FCommutesS Fpoint FEdge FTriangle) ∧ (FCommutesD Fpoint FEdge FTriangle).
```

```
Theorem FOrdCommutesS: FCommutesS Fv Fe Ft.
Theorem FOrdCommutesD: FCommutesD Fv Fe Ft.
```

```
Theorem FVRespectsEq:
  forall (delt:Delta10), unaryFunctionRespectsEquality eqVX eqVY (Fv delt).
Theorem FERespectsEq:
  forall (delt:Delta11), unaryFunctionRespectsEquality eqEdgeX eqEdgeY (Fe delt).
Theorem FTRespectsEq:
  forall (delt:Delta12), unaryFunctionRespectsEquality eqTriangleX eqTriangleY (Ft delt).
```

```
Theorem allFInverseInconsistentEEE: forall (FIpoint: Delta10 → VX → VY)
                                           (FIEdge: Delta11 → EdgeX → EdgeY)
                                           (FITriangle: Delta12 → TriangleX → TriangleY),
                                   Finverse FIpoint FIEdge FITriangle →
                                   FCommutesS FIpoint FIEdge FITriangle →
                                   FCommutesD FIpoint FIEdge FITriangle →
                                   FITriangle delta001 e_e_e = y1y0_y1y0_k.
```

```
Theorem allFInverseInconsistentsss: forall (FIpoint: Delta10 → VX → VY)
                                           (FIEdge: Delta11 → EdgeX → EdgeY)
                                           (FITriangle: Delta12 → TriangleX → TriangleY),
                                  Finverse FIpoint FIEdge FITriangle →
                                  FCommutesS FIpoint FIEdge FITriangle →
                                  FCommutesD FIpoint FIEdge FITriangle →
                                  FITriangle delta001 sx_sx_sx = de.
```

```
Theorem deNotEqualToThatOtherEdge: forall (d:Days), (eqTriangleY d y1y0_y1y0_k de)=false.
```

B Triangles in Y

Table 1 Triangles in Y.

y0y0_y0y0_y0y0: $y0 \xrightarrow{y0y0} y0 \xrightarrow{y0y0} y0$, $y0 \xrightarrow{y0y0} y0$	y1y1_k_y1y1: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{k} y1$
y0y1_y0y1_y0y0: $y0 \xrightarrow{y0y0} y0 \xrightarrow{y0y1} y1$, $y0 \xrightarrow{y0y1} y1$	k_y1y1_y1y1: $y1 \xrightarrow{y1y1} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{y1y1} y1$
y1y0_y0y0_y0y1: $y0 \xrightarrow{y0y1} y1 \xrightarrow{y1y0} y0$, $y0 \xrightarrow{y0y0} y0$	k_k_y1y1: $y1 \xrightarrow{y1y1} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{k} y1$
y1y1_y0y1_y0y1: $y0 \xrightarrow{y0y1} y1 \xrightarrow{y1y1} y1$, $y0 \xrightarrow{y0y1} y1$	y1y0_y1y0_k: $y1 \xrightarrow{k} y1 \xrightarrow{y1y0} y0$, $y1 \xrightarrow{y1y0} y0$
k_y0y1_y0y1: $y0 \xrightarrow{y0y1} y1 \xrightarrow{k} y1$, $y0 \xrightarrow{y0y1} y1$	y1y1_y1y1_k: $y1 \xrightarrow{k} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{y1y1} y1$
y0y0_y1y0_y1y0: $y1 \xrightarrow{y1y0} y0 \xrightarrow{y0y0} y0$, $y1 \xrightarrow{y1y0} y0$	y1y1_k_k: $y1 \xrightarrow{k} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{k} y1$
y0y1_y1y1_y1y0: $y1 \xrightarrow{y1y0} y0 \xrightarrow{y0y1} y1$, $y1 \xrightarrow{y1y1} y1$	k_y1y1_k: $y1 \xrightarrow{k} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{y1y1} y1$
y0y1_k_y1y0: $y1 \xrightarrow{y1y0} y0 \xrightarrow{y0y1} y1$, $y1 \xrightarrow{k} y1$	k_k_k: $y1 \xrightarrow{k} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{k} y1$
T_{de}: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y0} y0$, $y1 \xrightarrow{y1y0} y0$	T_{fi}: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y0} y0$, $y1 \xrightarrow{y1y0} y0$
y1y1_y1y1_y1y1: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{y1y1} y1$	

Table 2 Degenerate triangles in Y.

Degeneracy	Left vertex	Top-left edge	Top vertex	Top-right edge	Right vertex	Bottom edge
$s_0(\text{y0y0}) = y0y0_y0y0_y0y0$:	$y0$	$y0y0$	$y0$	$y0y0$	$y0$	$y0y0$
$s_1(\text{y0y0}) = y0y0_y0y0_y0y0$:	$y0$	$y0y0$	$y0$	$y0y0$	$y0$	$y0y0$
$s_0(\text{y0y1}) = y0y1_y0y1_y0y0$:	$y0$	$y0y0$	$y0$	$y0y1$	$y1$	$y0y1$
$s_1(\text{y0y1}) = y1y1_y0y1_y0y1$:	$y0$	$y0y1$	$y1$	$y1y1$	$y1$	$y0y1$
$s_0(\text{y1y0}) = T_{\text{de}}$:	$y1$	$y1y1$	$y1$	$y1y0$	$y0$	$y1y0$
$s_1(\text{y1y0}) = y0y0_y1y0_y1y0$:	$y1$	$y1y0$	$y0$	$y0y0$	$y0$	$y1y0$
$s_0(\text{y1y1}) = y1y1_y1y1_y1y1$:	$y1$	$y1y1$	$y1$	$y1y1$	$y1$	$y1y1$
$s_1(\text{y1y1}) = y1y1_y1y1_y1y1$:	$y1$	$y1y1$	$y1$	$y1y1$	$y1$	$y1y1$
$s_0(\text{k}) = k_k_y1y1$:	$y1$	$y1y1$	$y1$	k	$y1$	k
$s_1(\text{k}) = y1y1_k_k$:	$y1$	k	$y1$	$y1y1$	$y1$	k

Table 3 Equated triangles in Y, day 2.

$y1y1_y1y1_y1y1$: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{y1y1} y1$	$y1y1_y0y1_y0y1$: $y0 \xrightarrow{y0y1} y1 \xrightarrow{y1y1} y1$, $y0 \xrightarrow{y0y1} y1$
$y1y1_k_y1y1$: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{k} y1$	k_y0y1_y0y1: $y0 \xrightarrow{y0y1} y1 \xrightarrow{k} y1$, $y0 \xrightarrow{y0y1} y1$
k_y1y1_y1y1: $y1 \xrightarrow{y1y1} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{y1y1} y1$	$y0y1_y1y1_y1y0$: $y1 \xrightarrow{y1y0} y0 \xrightarrow{y0y1} y1$, $y1 \xrightarrow{y1y1} y1$
k_k_y1y1: $y1 \xrightarrow{y1y1} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{k} y1$	$y0y1_k_y1y0$: $y1 \xrightarrow{y1y0} y0 \xrightarrow{y0y1} y1$, $y1 \xrightarrow{k} y1$
$y1y1_y1y1_k$: $y1 \xrightarrow{k} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{y1y1} y1$	$y1y0_y1y0_k$: $y1 \xrightarrow{k} y1 \xrightarrow{y1y0} y0$, $y1 \xrightarrow{y1y0} y0$
$y1y1_k_k$: $y1 \xrightarrow{k} y1 \xrightarrow{y1y1} y1$, $y1 \xrightarrow{k} y1$	T_{fi}: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y0} y0$, $y1 \xrightarrow{y1y0} y0$
k_y1y1_k: $y1 \xrightarrow{k} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{y1y1} y1$	
k_k_k: $y1 \xrightarrow{k} y1 \xrightarrow{k} y1$, $y1 \xrightarrow{k} y1$	

Table 4 Non-equated triangles in Y, day 2.

$y0y0_y0y0_y0y0$: $y0 \xrightarrow{y0y0} y0 \xrightarrow{y0y0} y0$, $y0 \xrightarrow{y0y0} y0$	$y0y1_y0y1_y0y0$: $y0 \xrightarrow{y0y0} y0 \xrightarrow{y0y1} y1$, $y0 \xrightarrow{y0y1} y1$
$y1y0_y0y0_y0y1$: $y0 \xrightarrow{y0y1} y1 \xrightarrow{y1y0} y0$, $y0 \xrightarrow{y0y0} y0$	$y0y0_y1y0_y1y0$: $y1 \xrightarrow{y1y0} y0 \xrightarrow{y0y0} y0$, $y1 \xrightarrow{y1y0} y0$
T_{de}: $y1 \xrightarrow{y1y1} y1 \xrightarrow{y1y0} y0$, $y1 \xrightarrow{y1y0} y0$	

Π-Ware: Hardware Description and Verification in Agda*

João Paulo Pizani Flor[1], Wouter Swierstra[2], and Yorick Sijsling[3]

1 **Department of Information and Computing Sciences, Utrecht University, PO Box 80089, 3508 TB Utrecht, The Netherlands**
J.P.PizaniFlor@uu.nl

2 **Department of Information and Computing Sciences, Utrecht University, PO Box 80089, 3508 TB Utrecht, The Netherlands**
W.S.Swierstra@uu.nl

3 **Department of Information and Computing Sciences, Utrecht University, PO Box 80089, 3508 TB Utrecht, The Netherlands**
Y.Sijsling@uu.nl

Abstract

There is a long tradition of modelling digital circuits using functional programming languages. This paper demonstrates that by employing *dependently typed programming languages*, it becomes possible to define circuit descriptions that may be simulated, tested, verified and synthesized using a single language. The resulting domain specific embedded language, Π-Ware, makes it possible to define and verify entire families of circuits at once. We demonstrate this by defining an algebra of parallel prefix circuits, proving their correctness and further algebraic properties.

1998 ACM Subject Classification B.7.2 Integrated Circuits: Design Aids – Verification, D.3.2 Programming Languages: Language Classifications – Functional Languages, F.3.1 Logics and Meanings of Programs: Specifying and Verifying and Reasoning about Programs

Keywords and phrases dependently typed programming, Agda, EDSL, hardware description languages, functional programming

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.9

1 Introduction

There is a long tradition of using functional programming to model hardware circuits. Dating as far back as the 1980's, there have been many different proposed Domain-Specific Languages (DSLs) for the design and verification of circuits [20, 2, 19, 5]. Initially these languages were mostly standalone, but later *embedded* DSLs for hardware were developed, hosted in functional languages such as Haskell or ML.

All of these *functional hardware* DSLs have some limitations with respects to *(type) safety* or aspects of the design process that they support. Most of them allow for simulation, some support synthesis to Register-Transfer Level (RTL) netlists; others are focused on verification (using a combination of SMT solvers, automated theorem provers, or interactive proof assistants). We would argue, however, that none of these are *unified* typed languages for the design, simulation, verification, and synthesis of hardware circuits.

* This work was supported by the Netherlands Organization for Scientific Research (NWO) project on *A Dependently-Typed Language for Verified Hardware*.

© João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling;
licensed under Creative Commons License CC-BY
21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 9; pp. 9:1–9:27

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

Yet the need for better tools for the design of custom hardware accelerators is greater than ever. The performance of general purpose processors is becoming increasingly harder to improve, as we have already long ago faced the power wall and Instruction-Level Parallelism (ILP) shows diminishing returns [9]. There is a growing demand for hardware acceleration that is both easy to maintain and accurate.

This paper presents Π-Ware, a Hardware Description Language (HDL) embedded in *Agda* [18, 17], a dependently-typed programming language. Π-Ware provides a single, strongly-typed language which supports the definition, synthesis, simulation, testing and formal verification of complex circuits. The project is hosted at `http://piware.alvb.in`.

After giving a high-level overview of the language and its features (Section 2), this paper makes the following contributions:

- Π-Ware has been designed to be a safe and strongly-typed language. Our embedding (Section 3) makes use of dependent types to provide safety guarantees beyond the ones offered by HDLs embedded in simply-typed functional languages. The embedding is parameterized by the type of fundamental data over the wires and the library of fundamental gates being used. For example, instead of using only logic gates, one could add binary arithmetic operators to the fundamental library. This raises the level of abstraction of the description and simplifies verification.
- Unlike other embeddings in proof assistants (such as Coquet [5]), Π-Ware circuits are executable. We defined functional semantics for both combinational and sequential circuits (Section 4). This may seem trivial, but providing a total semantics of circuits that run forever requires some care. Specifically, we define a *causal stream-based* semantics to model the simulation of sequential circuits.
- As the circuit semantics is executable, we can use it to test our designs. Furthermore, for finite domains, we can automatically derive a proof by exhaustive testing (Section 5). More generally, the full power of Agda as an interactive theorem prover can be used to prove properties of *circuit generators*. These generators are usually defined using some sort of recursive pattern (*connection pattern*), and proofs about them rely on induction following the same pattern (*proof combinator*).
- Finally, we show how all these features may be combined in a case study: the verification of parallel prefix circuits (Section 6). We describe this family of circuits in terms of Π-Ware primitives and verify its behaviour. In particular, we formulate and prove algebraic laws involving operators and transformations over parallel prefix circuits, providing machine-verified versions of proofs previously developed on paper [12].

2 Overview

We can illustrate circuit models in Π-Ware by analyzing a simple example. Let us model a 2-way multiplexer (mux). For convenience, booleans are being carried over the wires, and we have the usual set of gates at our disposal: `NOT`, `AND`, `OR`.

A first step when designing a circuit is to think of its *specification*. For such a small circuit as mux, it's *truth table* (Table 1) serves well as specification.

Our mux has two data inputs (`A`, `B`) and a selection input (`S`). It behaves in such a way that, whenever ($S = 0$), the output is equal to input `A`, otherwise it is equal to input `B`. From the truth table we can (straightforwardly) derive a boolean formula:

$$Z = (A \land \neg S) \lor (B \land S) \qquad (1)$$

From this logical formula, a designer could then *implement* a circuit with the structure shown in Figure 1. This kind of graphical model is often known as *block diagram*.

Table 1 Truth table specification of mux.

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

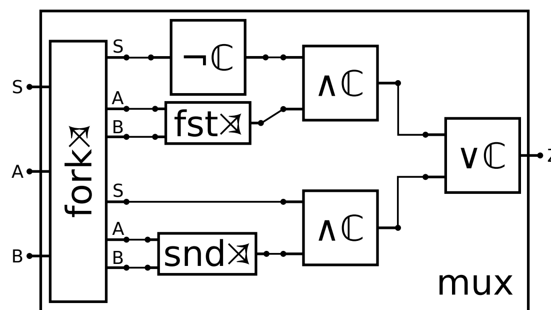


Figure 1 Block diagram of mux.

We can also view such a diagram in a different way, by considering the fundamental gates present, and grouping them using *sequential* (_⟫_) and *parallel* (_||_) composition. This corresponds exactly to the definition of mux in Π-Ware, shown in Listing 1.

In this description, we use three kinds of fundamental gates: AND (∧ℂ), OR (∨ℂ) and NOT (¬ℂ). Notice how the circuit type is *indexed* by *sizes* of the input and output. The *total* size of all inputs of mux amounts to 3 and total size of all outputs amounts to 1. Besides fundamental gates and composition, we also use some blocks to do *rewiring*. The circuit called fork×⤫ outputs two exact copies of its input bus side-by-side, while fst⤫$_1$ and snd⤫$_1$ select respectively the first and second wire from an input of size 2.

The mux example shows how Π-Ware circuits are described in a low level of abstraction. Circuits are combined in an *architectural* way, and there is no way *in the DSL* to refer to intermediary results (no variable binding). We discuss how this low-level description relates to other levels of abstraction in Section 7.

In mux, the parameters to the ℂ type constructor are constants, but they need not be in general. Using dependent types, we can precisely define and reason about *circuit generators*. For example, here is the type and definition of the aforementioned fork×⤫:

```
fork×⤫ : ∀ {n} → ℂ n (n + n)
fork×⤫ {n} = Plug (tabulate {n} id ++ tabulate {n} id)
```

Even though circuit semantics is only given in Section 4, we can already take a look at what would be possible given a functional (simulation) semantics for our examples. For now, let's assume the semantic function for circuits has the following type:

$$⟦_⟧ : ℂ\ i\ o → (\mathsf{Vec\ Bool}\ i → \mathsf{Vec\ Bool}\ o)$$

```
mux : ℂ 3 1
mux =  fork×⤨
   ⟫   (¬ℂ ∥ fst⤨₁ ⟫ ∧ℂ) ∥ (id⤨₁ ∥ snd⤨₁ ⟫ ∧ℂ)
   ⟫   ∨ℂ
```

Listing 1 Π-Ware model of mux.

That is, it takes a circuit with inputs of *size* i and outputs of *size* o, and returns its semantics: a function between appropriately-sized binary words. A first possibility to "reason" about circuit behaviour is to just *test* a circuit with some inputs and observe the produced outputs to gain confidence in its correctness. In the following snippet we give some test cases for mux. As using dependent types implies evaluation during type checking, we can formulate our tests as a type checking problem, requiring that our circuit will compute the required type by definition:

```
test1 : ⟦ mux ⟧ (false  :: (true :: false :: [])) ≡ (true  :: [])
test2 : ⟦ mux ⟧ (true   :: (true :: false :: [])) ≡ (false :: [])
```

This approach works fine to check the correctness of the simple mux: just write one test for each line of the truth table. However, one cannot simply *test* circuit generators (such as fork×⤨), as that would entail simulating a possibly infinite number of circuits. To definitely convince ourselves of the correctness of fork×⤨ we will want to *prove* a more general statement, such as:

```
fork×⤨⊑++ : ∀ n (w : Vec Bool n) → ⟦ fork×⤨ {n} ⟧ w ≡ w ++ w
```

Here we can regard the concatenation function (_++_) as a formal *specification* of fork×⤨, and fork×⤨⊑++ as a *proof* that fork×⤨ complies with its specification. What the statement of fork×⤨⊑++ intuitively means is that the circuit has the *effect* of duplicating its inputs into its outputs. The proof of this statement is written by induction on w, and relies on auxiliary lemmas involving vector functions such as _++_ and tabulate.

We discuss proofs of circuit (generator) properties more thoroughly in Section 5. In that section we also talk about a notion of *equivalence* between circuits and several algebraic properties of circuit constructors and *combinators*. As a prerequisite for verification, however, we first must define the syntax and semantics of circuits, in Sections 3 and 4.

3 Circuit Structure

The *syntax* of Π-Ware models gives a low-level description of a circuit's *architecture*, indicating how fundamental *gates* are connected to each other. This style approximates *block diagrams* drawn by hardware designers, but with a key distinction: in Π-Ware, components are connected in a *nameless* fashion, without explicitly mentioning ports or wires.

As Π-Ware is a *deeply-embedded* DSL, the syntax of the language is defined by a datatype (called ℂ). Our DSL distinguishes between *combinational* and *sequential* circuits. In summary, sequential circuits can have *internal state*, while combinational ones do not. We denote this distinction by *indexing* the ℂ type with an element of IsComb:

```
data IsComb : Set where σ ω : IsComb
```

```
infixl 4 _⟫_; infixr 5 _||_
data ℂ : {s : IsComb} → ℕ → ℕ → Set where
  Gate  : ∀ (g : Gate) {s}      → ℂ {s} (#in g) (#out g)
  Plug  : ∀ {i o s} → i ⤨ o  → ℂ {s} i o

  _⟫_  : ∀ {i m o s}        → ℂ {s} i m   → ℂ {s} m o   → ℂ {s} i o
  _||_  : ∀ {i₁ o₁ i₂ o₂ s} → ℂ {s} i₁ o₁ → ℂ {s} i₂ o₂ → ℂ {s} (i₁ + i₂) (o₁ + o₂)

  DelayLoop : ∀ {i o l} → ℂ {σ} (i + l) (o + l) → ℂ {ω} i o
```

Listing 2 The circuit datatype (ℂ).

We consider circuits indexed with the ω value to be sequential, and those with σ to be combinational. The choice of name for these (one-letter) constructors is only partially arbitrary: we were motivated by the usage of Σ^ω in mathematics to represent the set of all infinite sequences over a given alphabet S. This distinction between combinational and sequential circuits results in some convenience: with the knowledge that a circuit has no state, its simulation semantics can be simpler – just a function between appropriately-sized vectors. And with a simpler semantics, we can also get simpler proofs.

Listing 2 shows the circuit type (ℂ). The handling of the IsComb tag in each of the constructors tells us which sort of semantics (stateful or not) we need to have from the subparts in order to get the semantics of the whole circuit.

The constructors for sequential (_⟫_) and parallel (_||_) composition, for example, *preserve* the s tag. This means that, to evaluate a circuit (c_1 ⟫ c_2) in a stateless way, *both* c_1 and c_2 need to be stateless (combinational). Equivalently, if any of the parts is stateful, only a stateful evaluation of the whole is allowed.

Besides IsComb, the circuit datatype (ℂ) is also indexed by two natural numbers. These correspond, respectively, to the *total* number of input wires into the circuit and total number of output wires from the circuit. We were strongly influenced in our circuit syntax design choices by Coquet [5], especially in the usage of dependent types in the (_⟫_) and (_||_) constructors to enforce sizing constraints.

In order to facilitate discussion of the constructors of ℂ, we categorize them as either *primitive* or *composite*: composite constructors take arguments of type ℂ, while primitive ones do not. First, we look at the primitive constructors:

- Circuits constructed with Gate are the smallest possible *with computational content*. The whole PiWare.Circuit module is parameterized by a *gate library* (detailed in Section 3.2), and by calling Gate we simply pick one of those gates to use as building block.
- The other primitive constructor is Plug, which is necessary due to the *nameless* fashion in which we compose circuits. Since it is impossible to refer to any specific circuit port we cannot, for example, map the “first” output of a circuit to the “second” input of another. Plugs are required to do *rewiring*, but they perform *no computation*.

The argument to the Plug constructor has type i ⤨ o, and this is a synonym for a mapping from output wires (indices) to input wires (indices).

```
_⤨_ : ℕ → ℕ → Set
i ⤨ o = Vec (Fin i) o
```

Using such a mapping, no Plug can ever be built containing any information other than the origin of each output wire. An intuitive definition for (i ⤨ o) would be (Fin o → Fin i), but we opted for the (first-order) Vec representation to get easier combination of plugs and easier proofs. Also, the first-order representation will make synthesis more straightforward.

The composite constructors in Π-Ware represent ways in which smaller circuits can be connected to form a larger one. First, let us focus on the most interesting of them: DelayLoop. Both other composite constructors (_⟫_ and _||_) *preserve* the IsComb index. The DelayLoop constructor, however, is an exception: it is the only way to build a sequential circuit given a combinational one as argument.

This single possible way to *introduce state* makes the definition of circuit semantics simpler and, as the name hints, we make sure to always introduce a *clocked delay* at each occurrence of DelayLoop. Thus we avoid *combinatorial loops* in the circuit, which can make circuit analysis much more complex [10]. The remaining composite constructors of ℂ are:

- Sequential composition (_⟫_), which connects the output of one circuit to the input of another. The indices ensure that the sizes match up correctly.
- Parallel composition (_||_), that creates a combined circuit which has as inputs (resp. outputs) the inputs (resp. outputs) of *both* constituent subcircuits.

Careful indexing of sequential and parallel composition, together with the type of _⤨_, ensure that some design mistakes are *prevented by construction.* Floating wires are forbidden by _⟫_: in a term "c_1 ⟫ c_2", the output size of c_1 needs to equal the input size of c_2. Also, because Plug takes a *function* from outputs to inputs, only one source can be assigned to each load (no short-circuits). Lastly, the *totality* of the argument to Plug ensures that no plug output can be left unassigned.

As already mentioned, our circuit syntax is strongly influenced by Coquet [5]. Some differences are the partitioning of circuits by the IsComb tag into *combinational* or *sequential*, the first-order Plugs, and the type of the DelayLoop constructor, which in our case does not allow nesting of state.

In Π-Ware, circuits are parameterized both by the type of data travelling in the wires (an Atomic type) and by a set of fundamental Gates upon which all circuits are built. The first design choice taken when describing circuits in Π-Ware is which Atomic type to use, so let's start with that.

3.1 Atomic Types

Hardware descriptions in VHDL or Verilog, often model the information carried on the wires as bits. This stays close to a physical implementation and thus remains popular, however, sometimes it is useful to think of other types being carried in the wires. For example, an enumeration type better describes the possible states of a state machine. In Π-Ware, all circuit descriptions are *parameterized* by the type of element carried over the wires.

Types that can be carried over ports and wires are called *atomic* types (Listing 3). Elements of such types are considered to have *no parts* and cannot be *inspected* by Π-Ware.

Some simple examples of atomic types are: Bool, (named) enumerations and Fin n (for some n). In order to be used as an atomic type, a given type must be *finite* and *inhabited.* These requirements are packed up in the FiniteInhabited record (Listing 4).

We witness the finiteness of Atom by a bijection with Fin n, and the default field shows that the type in question has at least one inhabitant. The reason to forbid empty types from being used as Atom lies in the semantics of DelayLoop.

```
record Atomic : Set₁ where
  field Atom : Set
        enum : FiniteInhabited Atom

  open FiniteInhabited enum public
  W = Vec Atom
```

Listing 3 The Atomic record.

```
record FiniteInhabited {l} (A : Set l) : Set l where
  field finite   : Finite A
        default  : A

  open Finite finite public
```

Listing 4 The FiniteInhabited record.

We will cover circuit semantics with more detail in Section 4 but, in summary, each occurrence of DelayLoop prepends one extra element to the circuit's output stream. This extra element will have type Vec Atom n (for arbitrary n), and thus we need to have *at least one arbitrary value* of type Atom at our disposal (default). As a last remark, we make W n a synonym for Vec Atom n. Thus in any context parameterized by an instance of the Atomic record, we can refer to words of atoms in a more convenient way.

A last detail to note is that the bijection between the Atom type and Fin n implies the existence of a decidable equality (and decidable setoid structure) for Atom. There is a function in the Agda standard library (called eq?^{n}) that gives a decidable equality for any type A, provided there is an injection from A into Fin n. In our case, we pass to eq?^{n} the injection obtained as a consequence of the bijection between Atom and Fin n. The relevant definitions are shown in Listing 5.

3.2 Fundamental Gates

The mux example from Listing 1 was built with the usual boolean gates (AND, OR, NOT). Instead of hardwiring this choice in the definition of $\mathbb{C}$, Π-Ware allows users to choose their own collection of *fundamental gates*. These could be the boolean gates mentioned above, but also more complex circuits, such as muxes, registers, or arithmetic circuits, depending on the particular design.

To build a library of fundamental gates, users must define a suitable Agda record specifying the interface and semantics of each gate. This record (Gates) is shown in Listing 6.

First of all, the whole Gates module is parameterized by an instance of Atomic, thus fixing W and defining the type of elements that appear as inputs and outputs of gates.

The type of gate identifiers is stored in the Gate field, and there are functions that assign to each gate identifier a corresponding number of inputs (#in), number of outputs (#out), and specification function (spec). Notice the highly dependent type of spec and of Gates as a whole: the return type of spec depends on it's g parameter and on the #in and #out fields. The type of #in and #out depends, in turn, on Gate.

The choice of fundamental gates strongly influences circuit correctness proofs: the correctness of each gate defined by the Gates record is *assumed* rather than proved.

```
_≟ⁿ_ : ∀ {A : Set} ⦃ fA : Finite A ⦄ → Decidable {A = A} _≡_
_≟ⁿ_ ⦃ fA ⦄ = let open Finite fA in eq?ⁿ injection

_≟A_ : Decidable {A = Atom} _≡_
_≟A_ = _≟ⁿ_ ⦃ FiniteInhabited.finite enum ⦄

decSetoidA : DecSetoid _ _
decSetoidA = decSetoid _≟A_
```

Listing 5 Decidable equality for Atom derived via injection to Fin n.

```
record Gates : Set₁ where
  field Gate        : Set
        #in #out    : Gate → ℕ
        spec        : ∀ g → (W (#in g) → W (#out g))
```

Listing 6 The Gates record.

To perform boolean logic with our circuits, we will want to use any *functionally complete* set of boolean gates. A particularly simple such set is {NAND}, which contains only the negated AND gate. First, we must define how many input and output ports does each gate in the library have:

```
#in #out : NandGate → ℕ
#in   ⊼ℂ' = 2
#out  ⊼ℂ' = 1
```

Notice that the parameter of the #in and #out functions is of type NandGate. This is the type of *gate identifiers* in the library. We impose no requirements on a type to satisfy this role, but here NandGate is a simple enumeration type. Having defined the interface of each gate in our library (there is only one), we then define the specification function:

```
spec-⊼ℂ : W 2 → W 1
spec-⊼ℂ (x ∷ y ∷ []) = [ not (x ∧ y) ]
```

There are no restrictions imposed by Π-Ware on which kind of gate should or should not be present in a library, and higher-level Atomic and Gates instances can make designs much simpler. For example, with an Atomic instance defined to represent 8-bit signed integers, there can be a useful Gates library containing some set of modular arithmetic operators over these integers.

As another example of gate library, Π-Ware also includes BoolTrio, a gate library operating over booleans with three boolean operations (NOT, AND, OR) and two constant gates (FALSE and TRUE). We specify the behaviour of the gates using the boolean functions from Agda's standard library (Data.Bool).

When *simulating* a Π-Ware circuit, we will use the specification functions of the gate library used in that circuit. Likewise, in proofs of circuit correctness, the fundamental gates are assumed to be correct. Therefore the elements in a Gates library can be seen as fundamental in two ways:

- Fundamental *behaviour*, as they have no subparts.
- Fundamental *functional correctness*, as it is assumed.

3.3 Abstraction Levels

Throughout this paper, we will deal with circuit models and circuit semantics only in terms of their *size* (the ℂ datatype is indexed by two natural numbers, representing the sizes of a circuit's input and output). However, Π-Ware offers a thin *data abstraction* layer, allowing Agda types in circuit's inputs/outputs (instead of Vec Atom).

A typed circuit is defined as just a wrapper record around a sized circuit (ℂ). Therefore, the computation still is performed over words, but the description of a typed circuit contains information on how to *convert* between elements of the involved Agda types and the correspondingly-sized words.

This thin layer makes mainly *simulation* and testing more convenient and less verbose (no need to always build vectors to compare with in testing, for example). However, as the computation is still performed over words, proving a circuit's correctness will still rely on lemmas involving the *sized* level (vectors, atoms).

We discuss with more detail in Section 7 how this layer of data abstraction influences modelling and verification, and what could be other possible ways of raising the level of abstraction in circuit description.

4 Circuit Semantics

Due to the choice of using deep embedding to implement our DSL, it is possible to write several different semantics for circuit models.

When talking about deeply embedded languages, a semantic function is just a function mapping the Abstract Syntax Tree (AST) of our DSL to a desired *carrier* type. All of the circuit semantics currently implemented in Π-Ware are *compositional*, which means that they can be defined by *folding* the ℂ type with an algebra.

The module PiWare.Circuit.Algebra defines the *algebra type* for ℂ (as a record), along with the associated *catamorphism* (fold). There are two algebra types: one for combinational circuits (ℂσA) and one for (possibly) sequential ones (ℂA). The only difference between them is that a case for DelayLoop is absent from ℂσA. Here we show the algebra type for combinational circuits (ℂσA):

```
record ℂσA : Set where
  field GateA  : ∀ g#              → T (#in g#) (#out g#)
        PlugA  : ∀ {i o} → i ⤨ o  → T i o
        _⟫A_   : ∀ {i m o}         → T i m   → T m o   → T i o
        _||A_  : ∀ {i₁ o₁ i₂ o₂}   → T i₁ o₁ → T i₂ o₂ → T (i₁ + i₂) (o₁ + o₂)
```

We use Agda's feature of *sections* (parameterized anonymous modules) to avoid repetition of parameters used in several definitions. Firstly, the algebra record type (ℂσA) is parameterized by the *carrier* type (called T). Also, the catamorphism for combinational circuits (called cataℂσ) is parameterized by an instance of the algebra record.

Listing 7 shows the catamorphism for combinational circuits (cataℂσ). Notice how this definition, *by itself*, takes only the circuit to be interpreted as parameter. However, cataℂσ is part of a section, so outside of the section it takes an extra parameter (an element of the record type ℂσA). Notice also how (due to the σ index) there is no need to define a clause for DelayLoop.

```
cataℂσ : ∀ {i o} → ℂ {σ} i o → T i o
cataℂσ (Gate g)    = GateA g
cataℂσ (Plug f)    = PlugA f
cataℂσ (c₁ ⟫ c₂)  = cataℂσ c₁ ⟫A cataℂσ c₂
cataℂσ (c₁ ∥ c₂)  = cataℂσ c₁ ∥A cataℂσ c₂
```

Listing 7 Catamorphism for combinational circuits.

```
simulationσ : ℂσA
simulationσ = record { GateA  = spec
                     ; PlugA  = λ p ins → tabulate (flip lookup ins ∘ flip lookup p)
                     ; _⟫A_   = flip _∘′_
                     ; _∥A_   = λ f₁ f₂ → uncurry′ _++_ ∘ map× f₁ f₂ ∘ splitAt′ _ }
```

Listing 8 Simulation semantics algebra for combinational circuits.

4.1 Combinational Simulation

As a particular example of such a compositional semantics, we defined *executable* simulation for Π-Ware circuit models, which maps circuits to the domain of Agda functions. This semantics is *executable* in the sense that, by applying the function obtained using the semantics to an input, the same output should be calculated as if the circuit had been implemented in hardware and run.

When getting the simulation semantics of a combinational circuit, we want to obtain a function between appropriately-sized words, that is, a circuit of type "ℂ i o" should result in a function of type "W i → W o". Thus the carrier type for the combinational simulation algebra is:

```
W⟶W : ℕ → ℕ → Set
W⟶W m n = W m → W n
```

With the appropriate carrier defined, we get a very simple type and definition for the combinational simulation function:

```
⟦_⟧ : ∀ {i o} → ℂ {σ} i o → W⟶W i o
⟦_⟧ = cataℂσ simulationσ
```

Notice how the type of the semantic function *requires* the interpreted circuit to be combinational (it must be indexed by σ). In this way, the algebra used (simulationσ) does not have a field for the DelayLoop case. We show on Listing 8 the definitions for each of the fields in the algebra (simulationσ)[1].

The cases for sequential (_⟫A_) and parallel composition (_∥A_) rely, respectively, on function composition and map× over products. Sequential circuit composition is evaluated simply as flipped composition of the functions obtained from evaluating the subcircuits. For parallel composition, the simulation behaviour is to split the input at the appropriate index, pass each of the parts to the functions obtained from evaluating the subcircuits, and

[1] It is useful to note Agda's convention of adding primes to the names of non-dependent functions (uncurry′, _∘′_)

concatenate the results. In the case of fundamental gates, we simply rely on that gate's specification function. This leaves the most interesting definition to be explained: PlugA.

In the case of a Plug, we build the output word *pointwise* by using tabulate. The tabulate function from Agda's standard library "fills" a (Vec A n) by evaluating a given function of type (Fin $n \to A$) on all points of its domain. In our case, each of these points is an output index (element of Fin o). First, we lookup the output index in the plug mapping p, obtaining the corresponding input index. Then we use this index to lookup the input word and place the correct Atom on the output.

Let's now consider a simple example circuit, its simulation semantics and the involved types, to better understand how all these definitions fit into place. We consider a two-input `NAND` gate (⊼ℂ), with the following type and definition:

```
⊼ℂ : ∀ {s} → ℂ {s} 2 1
⊼ℂ = ∧ℂ ⟫ ¬ℂ
```

The gate is described using two-input conjunction (∧ℂ) and one-input negation (¬ℂ) as building blocks, and these pieces come from the library of fundamental gates that we are using (BoolTrio). By evaluating ⊼ℂ we then obtain the following function:

```
⟦ ⊼ℂ ⟧ : W⟶W 2 1
⟦ ⊼ℂ ⟧ = spec ¬ℂ' ∘ spec ∧ℂ'
```

To simulate ⊼ℂ we rely on the specification functions of both gates and on function composition. The names ¬ℂ' and ∧ℂ' are just the gate *identifiers*. If we reduce the expression above further (expanding spec and W⟶W), we obtain the following:

```
⟦ ⊼ℂ ⟧ : W 2 → W 1
⟦ ⊼ℂ ⟧ = λ { (x :: y :: []) → [ not (x ∧ y) ] }
```

The verification of circuits and circuit generators will be discussed in detail in Section 5. But it is already clear that it will rely heavily on laws involving vectors, as well as algebraic properties of the fundamental gates and plugs used in the design.

4.2 Sequential Simulation

Sequential circuits are those in which their output at any given instant may depend not only on a combination of the input at the same instant, but on the *sequence* of previous inputs.

In Π-Ware, we model only the *discrete time domain*, and therefore a circuit's input *signal*[2] is *piecewise constant* (as well as its output signal). Because of this characteristic, we can model both input and output signals as Streams[3], in which each element of the Stream is a word.

Perhaps the simplest example of a circuit with internal state is shift. This circuit will output at any clock cycle t the value present on its input at the preceding cycle. The architecture of shift consists simply of one DelayLoop and one Plug:

```
shift : ℂ {ω} 1 1
shift = DelayLoop swap⤨₁
```

[2] By signal we mean a function over the time domain.
[3] A stream is an infinite list, i.e., a list without the Nil constructor.

The expected behaviour of shift exemplifies why the type of Atoms (values that can be carried over Π-Ware wires) needs to be not only finite but also *inhabited*. The shift circuit will put out the value present at its input on the *previous* clock cycle. On the zeroth clock cycle, however, there is no *previous* cycle, and thus there must be some default value to be put out.

To discuss the semantics of shift, we first note that the type of shift is tagged by ω (omega), and for circuits with ω in their type, we must use the *sequential simulation semantics* (⟦_⟧ω). In the specific case of shift, the function obtained via the sequential simulation semantics will have the following type:

```
⟦ shift ⟧ω : Stream (W 1) → Stream (W 1)
```

The function obtained via ⟦_⟧ω consumes and produces a Stream of adequately-sized words. To explain in detail how the sequential semantics is actually defined, however, we have to mention a key distinction between digital circuits and stream functions in general: An unconstrained stream function (that is, an arbitrary element of type Stream A → Stream B) can (possibly) *look into the future*. Considering Streams over the discrete time domain, one simple example of stream function that "looks into the future" is tail.

```
tail : ∀ {A} → Stream A → Stream A
tail (in₀ :: in₁₊) = ♭ in₁₊
```

The element at position 0 in the output of tail depends on the input at position 1, and so forth. Sequential circuits clearly cannot show this behaviour (at least not if we want to physically implement them). As we want our sequential circuits to be synthesizable to actual hardware, we should ensure that our semantics will only ever produce *causal stream functions*.

One way to define a causal stream function is as the unfolding of a function producing only the *next* output, given the current and past inputs. We call these functions *causal step functions*, and they are defined as follows:

```
_⇒c_ : ∀ {l₁ l₂} (A : Set l₁) (B : Set l₂) → Set (l₁ ⊔ l₂)
A ⇒c B = Γc A → B
```

The symbol Γc means *causal context*, and it is defined simply as a non-empty list (List⁺), that is, a pair of the head (current value) with a possibly-empty tail (past values).

```
Γc : ∀ {l} (A : Set l) → Set l
Γc = List⁺
```

Coming back to the definition of simulation semantics for sequential circuits, we can now establish the *carrier* type for the algebra of sequential circuits as being a *causal step function* between words of the appropriate length. Then, the *causal* simulation of a circuit is defined as a catamorphism over ℂ with the simulationc algebra:

```
W⇒cW : ∀ i o → Set
W⇒cW i o = W i ⇒c W o
```

```
⟦_⟧c : ∀ {i o} → ℂ i o → W⇒cW i o
⟦_⟧c = cataℂ {aσ = simulationσ} simulationc
```

```
simulationc : ℂA {W⟶W} {W⇒cW}
simulationc = record { GateA        = λ g → GateAσ g ∘ head
                     ; PlugA        = λ f → PlugAσ f ∘ head
                     ; _⟫A_         = λ f₁ f₂ → f₂ ∘ map⁺ f₁ ∘ pasts
                     ; _||A_        = λ f₁ f₂ → uncurry′ _++_ ∘ map× f₁ f₂ ∘ unzip⁺ ∘ splitAt⁺ _
                     ; DelayLoopA   = λ {_} {o} f → take_v o ∘ delay o f }
  where open ℂσA simulationσ using () renaming (GateA to GateAσ; PlugA to PlugAσ)
```

Listing 9 Simulation semantics algebra for sequential circuits.

Listing 9 shows the simulationc record. Note how the the definitions packed inside of simulationσ are made available for use by open-ing the corresponding module in the where block.

In the case of GateA and PlugA we simply take the *present* value from the causal context and pass it to the corresponding combinational field (GateAσ and PlugAσ), while the sequence (_⟫A_) and parallel (_||A_) cases are a bit more involved.

To understand the _⟫A_ case, first notice that each of the parameters f_1 and f_2 is a causal step function. The pasts function takes the causal context and produces a (non-empty) list of all of its tails: essentially, each element from this list is a causal context *viewed from a given moment*. We then map⁺ the f_1 function over each of these pasts, producing a list in which each element is the *next output considering that given past*. Finally, the result of mapping is fed as the causal context of f_2.

The definition for the parallel case (_||A_) is somewhat similar to the combinational one. We also split the input at the appropriate point and use map× to apply f_1 and f_2 to each part of the product. However, splitAt⁺ works pointwise over the causal context, thus the need to use unzip⁺ in order to make the list of pairs into a pair of lists.

Perhaps the most important case in the sequential simulation algebra is DelayLoopA. The delay function transforms the regular word function (which takes l extra wires) into a causal step function, which depends on the history of inputs instead of the current state. According to this semantics, a circuit built with DelayLoop corresponds to a *Mealy machine*, where the state has size l, and the combinational circuit inside of it calculates *both* the next output and the next state.

By calling our causal semantic function (⟦_⟧c) over a circuit we obtain a causal *step function*. Then, by just unfolding this step function we obtain the *causal stream function* which is actually the user-facing type for the simulation of sequential circuits:

```
⟦_⟧ω : ∀ {i o} → ℂ i o → (Stream (W i) → Stream (W o))
⟦_⟧ω = runc ∘ ⟦_⟧c
```

In contrast with the type of ⟦_⟧ (the combinational semantics), the type of ⟦_⟧ω makes no requirement on how the circuit parameter should be *indexed*. This means that the sequential semantics can be used to obtain a stream function from both sequential and combinational circuits. In the case of evaluating a combinational circuit using ⟦_⟧ω, the obtained stream function just applies the calculation performed by the circuit *pointwise* on the stream (ignoring the past).

5 Verification

Π-Ware also allows for the *verification* of circuit models. The kind of properties that can be stated and verified depends on the semantics being used. With Π-Ware in its current form,

we can express mainly *functional* specifications, that is, those related to the input/output characteristics of the circuit. Furthermore, due to the embedding in a dependently-typed language, Π-Ware allows for both *testing* of any specific circuit, as well as *proofs* of correctness for *circuit generators*.

Tests and proofs can be written which check constraints on the outputs or witness arbitrary relations between the inputs and outputs of a circuit. In particular, the Design Under Test (DUT) can be verified to have the same input/output behaviour as an Agda function *assumed to be correct*. Also, we have defined a notion of *extensional equivalence* between circuits, allowing us to prove algebraic properties of circuit constructors and combinators, as well as to define provably-correct semantics-preserving circuit transformations.

5.1 Testing

Testing can be used by a designer to *gain confidence* in the functional correctness of a model early in the design process, before attempting to write proofs in their full generality. Writing test cases can also be a useful way to capture requirements from whoever commissioned the circuit, thus aiding in *validation*.

Using only the simulation functions (⟦_⟧ and ⟦_⟧c), *manual* test cases can already be written: this method is usually called *unit testing*. These are some examples of unit tests for a two-input mux (the leftmost boolean in the input vector being the *selection* bit):

```
test-mux₁ : ⟦ mux ⟧ (false ∷ (true ∷ false ∷ [])) ≡ [ true ]
test-mux₁ = refl

test-mux₂ : ⟦ mux ⟧ (true ∷ (true ∷ false ∷ [])) ≡ [ false ]
test-mux₂ = refl
```

Unit testing is useful, but we can do better, thus the focus of this subsection is on Π-Ware's facilities to help *test automation*. For circuits with inputs and outputs of small size, verification via *exhaustive checking* is feasible, and our ultimate goal is to make this as automatic and concise as possible.

The first step of abstraction from manually-written test cases is to have an Agda function serving as specification of the circuit behaviour. This means that, for any possible input to the circuit, evaluating the function with this input will produce an output assumed to be correct.

Continuing with our mux example, let's check its correctness by comparing it with a specification function. First of all, the simulation semantics of mux has the following type:

```
⟦ mux ⟧ : W 3 → W 1
```

Looking at this type, and thinking about the expected behaviour of mux (*selecting* one of two inputs), a reasonable candidate for specification is as follows:

```
ite : W 3 → W 1
ite (s ∷ a ∷ b ∷ []) = [ if s then b else a ]
```

The first required task for automatic exhaustive checking is to *generate* all possible values of the circuit's input type. For this, we need the input type to have an instance of the Finite record, which embodies a *bijection* between the type in question and Fin n. The definition of Finite is shown in Listing 10.

```
record Finite {l} (A : Set l) : Set l where
  field #A      : ℕ
        mapping : A ↔′ Fin #A

  open Inverse′ mapping public
```

Listing 10 The Finite record.

There are some instances of Finite defined in the Π-Ware library for primitive types (amongst which Bool), along with products, sums and vectors. Specifically, the input type of ⟦ mux ⟧ is W 3 (equal to Vec Bool 3), so the necessary Finite instance relies on the pre-defined instances for vectors and booleans.

Having a way to generate all values of a type, we can create a vector containing all of them. More interestingly, we can create a (heterogeneous) vector containing all of the *proofs* that each value satisfies a certain predicate: this vector will contain proofs $\{P(x_1), P(x_2), \ldots, P(x_n)\}$ for all the elements x_n of a type A given a certain predicate (P : $A \to$ Set).

By using its bijection with Fin n, we can have a ∀-introduction rule for any Finite type.

```
∀-Finite : ∀ {l} {A : Set l} {P : A → Set} ⦃ fin : Finite A ⦄
             {ps : vec↑ (tabulate (P ∘ from ⦃ fin ⦄))} → (∀ (x : A) → P x)
```

The ∀-Finite function takes an *implicit* parameter (*ps*) containing the aforementioned heterogeneous vector of proofs. This parameter can be implicit because each of the vector's elements reduces to (tt : ⊤) (if the predicate holds), and the whole "vector" reduces to a nested pair of units. Agda's definition of both pairs and the unit type as a record, combined with the η-rule for records, ensure that the value of *ps* can be guessed.

Now, our final goal is to have exhaustive checking of circuit behaviour, so we need a ∀-introduction rule for the type of circuit inputs and outputs – Vec Atom n, abbreviated as W n. We know that Vec A n is Finite whenever A also is (proof omitted here), and combining this knowledge with the previous definition of ∀-Finite we arrive at the desired ∀-W rule (∀-introduction for *words*):

```
∀-W : ∀ {n} {P : W n → Set} {ps : vec↑ (tabulate (P ∘ fromFinite ⦃ Finite-W ⦄))}
      → (∀ (w : W n) → P w)
```

In particular, we can then simply use ∀-W to prove properties involving all possible inputs of a circuit. The property in which we are interested is whether a circuit and a given specification function *agree* on a certain input.

```
_⊑?_at_ : ∀ {i o} (c : ℂ {σ} i o) (f : W i → W o) → (W i → Set)
c ⊑? f at w = T ⌊ (⟦ c ⟧ w) ≟W (f w) ⌋
```

The statement (c ⊑? f at w) can be read as "*c complies with f at input w*". This relation relies on a decidable equality over output words of the checked circuit (_≟W_), and uses it to compare the results obtained by running the circuit simulation and the specification function. In the definition of _⊑?_at_, the call to ⌊_⌋ serves only to transform the value returned by _≟W_ into a Bool, which is then further transformed by T into a Set (either ⊤ or ⊥).

To put the last pieces of the puzzle together we call ∀-W, passing the relation just defined (partially applied) as the predicate to be exhaustively checked. In this way we obtain the function we ultimately wanted: check⊑.

```
check⊑ : ∀ {i o} (c : ℂ i o) (f : W i → W o) {ps : vec↑ (tabulate (c ⊑? f at_ ∘ fromW i))} → c ⊑? f
check⊑ c f {ps} = ∀-W {P = c ⊑? f at_} {ps}
```

This function performs automatic exhaustive checking to verify that a circuit complies with a given specification. It is feasible to use check⊑ for verification of small circuits such as mux, or for small parts of bigger designs (parts with few ports). For bigger designs, and in particular to verify circuit *generators*, we need to resort to manually-written proofs.

5.2 Proofs

The key advantage brought to verification by using a dependently-typed language is that properties can be proven not only of any *specific* circuit, but of *circuit generators*. Circuit generators are *parameterized* definitions from which for each value of the parameter, a different circuit can be derived.

The term "circuit generator" itself comes from the Lava [2] EDSL, but the idea of parameterized definitions is *at least* as old as VHDL's *generics* [13]. The parameters of these circuit generators are usually structural properties of the circuit, such as the *sizes* or amount of inputs and outputs of a circuit. Another example would be configuring how many clock cycles does the input get delayed in a shift register.

Usually, these definitions will be *recursive*, and thus the proofs of statements involving these generators will then be performed by induction. A circuit generator muxN, that selects between *two* inputs of size n each, has the following type and definition:

```
muxN : ∀ n {s} → ℂ {s} (1 + (n + n)) n
muxN zero     = nil⤫
muxN (suc n)  = adapt⤫ n ⟫ (mux ∥ muxN n)
```

For a given value of the parameter n, this definition produces a circuit with input size $1 + (n + n)$ (1 selection bit, plus n bits for each input) and output size n. The base case is a circuit with one input and zero outputs, and that matches the size of the empty plug (nil⤫). In the recursive case, we connect the 2-input mux and the recursive call (muxN n) in parallel, and we need a Plug (called adapt⤫) to make the right wires meet the right ports.

```
adapt⤫ : ∀ n {s} → ℂ {s} (1 + ((1 + n) + (1 + n))) ((1 + 1 + 1) + (1 + (n + n)))
```

The diagram on Figure 2 shows exactly how adapt⤫ forks the selection bit and rearranges the remaining wires appropriately.

As already mentioned before, the choice of specification function has a significant impact on the proof of correctness for a circuit. In the case of muxN, the specification is iteN:

```
iteN : ∀ n → W (1 + (n + n)) → W n
iteN zero     _                 = []
iteN (suc n)  (_ ∷ ab)          with splitAt (suc n) ab
iteN (suc n)  (s ∷ .(a ++ b))   | a , b , refl = if s then b else a
```

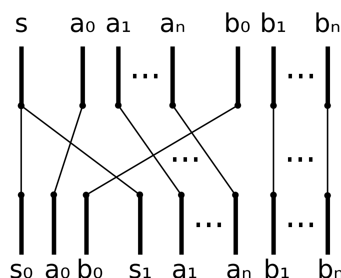


Figure 2 Architecture of the adapt⤩ plug.

In iteN, the tail of the input is split into two equal parts and we use if_then_else_ to choose (based on the selection bit), which of the two parts will be the output.

The functional correctness property that we are interested in is the *pointwise equality* of the specification function for a circuit and the function obtained via the simulation semantics, that is, both functions have to *agree on all inputs*. This relation is abbreviated with the name _⊑_, where a statement $(c \sqsubseteq f)$ can be read as "*c complies with f*".

In the case of our muxN example, the proof of compliance will need to follow the same induction pattern used to define the specification (iteN) itself. Namely, we need to pattern match on n and do case analysis on the result of splitting the tail of the input.

```
muxN⊑iteN : ∀ n → muxN n ⊑ iteN n
muxN⊑iteN zero      (_ :: []) = refl
muxN⊑iteN (suc n)  (_ :: ab)            with splitAt (suc n) ab
muxN⊑iteN (suc n)  (s :: .(a ++ b))  | a , b , refl = muxN⊑iteN′
```

Unfortunately, the full proof of muxN⊑iteN is a bit too long to be completely analyzed here (we abbreviate at muxN⊑iteN′). The proof relies of course on the proof of correctness for mux (the basic circuit with type ℂ 3 1). It also relies on properties of the adapt⤩ plug, ensuring that it's semantics essentially rearranges the input word in the appropriate way.

5.3 Connection Patterns

We are interested not only in proving properties of circuits in isolation, but also about the behaviour of so-called *connection patterns*[4]. Connection patterns are just functions taking circuits as inputs and producing circuits as outputs. Typically they use the constructors of ℂ to *connect* their arguments in a certain fashion, thus the name.

A (very simple) example of connection pattern is parsN, which connects n copies of a given circuit in parallel. The type and definition of parsN are:

```
parsN : ∀ {k i o s} → ℂ {s} i o → ℂ {s} (k * i) (k * o)
parsN {k} {i} {o} c = subst₂ ℂ  (*-sum-replicate k i) (*-sum-replicate k o)
                                (pars (replicate₂ {n = k} c))
```

Notice how the input and output sizes of the combined circuit are *statically guaranteed* to be correct, as they are calculated from the input/output sizes of the circuit passed as parameter. This definition (parsN) is a special case of a more general pattern: instead of

[4] This name comes from Lava as well.

replicating the same circuit n times, we can connect a whole vector of (different) circuits in parallel. This is achieved by pars:

```
pars : ∀ {n s} {is os : Vec ℕ n} (cs : VecI₂ (ℂ {s}) is os) → ℂ {s} (sum is) (sum os)
pars {is = []}        {[]}       []I₂         = nil⤫
pars {is = _ :: _}  {_ :: _}  (c ::I₂ cs)  = c ∥ pars cs
```

As the parameter of pars we need a special kind of vector, ensuring that only elements of types built with a given type constructor (ℂ) can be present in the vector. This special kind of vector is VecI_2, what we call an *index-heterogeneous vector* [5]. It is heterogeneous in the sense that its elements have different types, but only the indices vary, and the type constructor is fixed for all elements.

Another case of basic connection pattern is seqsN, taking a circuit and connecting n copies of it in sequence:

```
seqsN : ∀ k {s io} → ℂ {s} io io → ℂ {s} io io
seqsN k = seqs ∘ replicate {n = k}
```

Notice how the input and output sizes of the argument circuit are the same (io). This is because the _⟫_ constructor forces the output size of a circuit in this sequence to match the input size of the next.

Also seqsN is a special case of a general pattern: connecting a vector with n circuits in sequence. For this connection to be even *possible*, we need the input/output sizes of the circuits in the vector to be *pairwise compatible*. This means that for each circuit, its output size must be equal to the input size of the next. To this end, we adapt the work done on *type-aligned sequences* [22] to a dependently-typed setting.

We are currently working on establishing lemmas about the behaviour of these connection patterns in order to make proofs involving their usage simpler. For example, the simulation behaviour of seqsN can be shown to be that of the iterate function, that is:

```
∀ w → ⟦ seqsN k c ⟧ w ≡ (iterate k ⟦ c ⟧) w
```

Furthermore, we are also working on expressing connection patterns as folds over the underlying (indexed heterogeneous) vectors, as that would allow for more general and powerful laws.

5.4 Circuit Equivalence

Until now we have talked about relations between a circuit and a function — such as the *complies with* relation (i.e. "$c \sqsubseteq f$"). However, it is also very important to have an *equivalence relation* between circuits themselves. Given a properly-defined such relation, we can then have at our disposal laws like "c ⟫ id⤫ ≈ c", allowing for *provably safe* circuit optimizations.

We have defined such a notion of circuit equivalence *up-to-simulation* for *combinational* circuits, and a similar notion (and laws) for sequential circuits is left for future work. In this section we explain the several iterations we have gone through until achieving the current definition of circuit equivalence, correcting a small issue at each step. In the most naïve and first attempt, we just require identical inputs and compare the outputs of simulating both circuits using (propositional) equality.

---

[5] More specifically, VecI_2 only handles type constructors with *two* indices.

```
_≡c_ : ∀ {i o} (c₁ c₂ : ℂ i o) → Set
c₁ ≡c c₂ = ∀ w → ⟦ c₁ ⟧ w ≡ ⟦ c₂ ⟧ w
```

This definition is very unsatisfactory though, because it can only be used to compare circuits with *definitionally equal* indices, i.e, we cannot compare (c_1 : ℂ 1 n) with (c_2 : ℂ 1 ($n + 0$)). The first improvement over this definition is to use *vector equality* to compare the outputs. The notion of *semi-heterogeneous* vector equality (_≈_) is defined in Agda's standard library and it considers two vectors equal whenever the elements are *pointwise* propositionally equal. The new definition of circuit equivalence looks as follows:

```
_≅_ : ∀ {i o₁ o₂} → ℂ i o₁ → ℂ i o₂ → Set
c₁ ≅ c₂ = ∀ w → ⟦ c₁ ⟧ w ≈ ⟦ c₂ ⟧ w
```

While now the problem of word size ($n + 0$ vs. n) has been solved for the outputs, the issue remains for the input: we cannot yet compare (c_1 : ℂ n 1) with (c_2 : ℂ ($n + 0$) 1). Ultimately, what we want for circuit equivalence is to ensure that, when given "vector equal" inputs, both circuits will generate "vector equal" outputs:

```
_≊_ : ∀ {i₁ o₁ i₂ o₂} → ℂ i₁ o₁ → ℂ i₂ o₂ → Set
_≊_ {i₁} {_} {i₂} {_} c₁ c₂ =
  ∀ {w₁ : W i₁} {w₂ : W i₂}
  → w₁ ≈ w₂ → ⟦ c₁ ⟧ w₁ ≈ ⟦ c₂ ⟧ w₂
```

This is the definition that *almost* gets us there. It has a big problem, though: it's unsound. Very easily we can construct a term of type (c_1 ≊ c_2) simply by making sure the hypothesis is false. A simple example of a term that should be banned by _≊_ but is allowed is the following:

```
≊-unsound : (c₁ : ℂ 0 0) (c₂ : ℂ 1 1) → c₁ ≊ c₂
≊-unsound c₁ c₂ ()
```

To solve this issue, we make an extra requirement for two circuits to be considered equal. Now, not only vector equal inputs must lead to vector equal outputs, but also there must be a proof that the sizes of the input words are propositionally equal.

```
data _≋_ {i₁ o₁ i₂ o₂} : ℂ i₁ o₁ → ℂ i₂ o₂ → Set where
  refl≋ : {c₁ : ℂ i₁ o₁} {c₂ : ℂ i₂ o₂} (i≡ : i₁ ≡ i₂)
        → c₁ ≊ c₂ → c₁ ≋ c₂
```

Besides the requirement over sizes of inputs, we also implement this relation as a datatype (instead of a Set-valued function). This allows us to pattern-match on the arguments of the refl≋ constructor, which is needed when proving lemmas about _≋_ (for example, symmetry and transitivity).

With _≋_, we arrive at the definition of circuit equivalence used to state algebraic properties of circuit constructors and combinators, and also in the case study discussed in Section 6. For extra convenience, we packed up (ℂ, _≋_) into an indexed setoid structure, and we added to Agda's standard library some facilities for *equational reasoning* with indexed setoids. All in all, this allows proofs about circuit equivalence to be written in a very nice-looking style. A good example of such a proof can be seen in Listing 12 of Section 6.

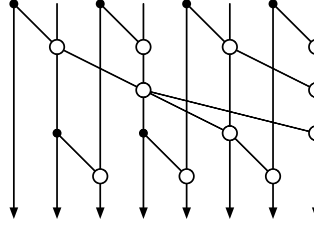


Figure 3 Example of an 8-input parallel prefix circuit.

6 Case Study: Parallel Prefix Circuits

In order to put in practice the definitions of Π-Ware we decided to perform a case study involving *parallel prefix circuits*. Parallel prefix circuits are a wide family of circuit architectures that compute *scans*, that is, given a binary operator ⊕ and a vector of inputs $[x_0, x_1, x_2, ..., x_n]$, it calculates the output vector $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), ..., (x_0 \oplus ... \oplus x_n)]$.

When talking about parallel prefix circuits, we always assume that the binary operator ⊕ is *associative*, thus allowing different parts of the output to be calculated *in parallel*. In Figure 3 we show an example of a circuit utilizing maximal parallelism to calculate a scan with 8 inputs.

In this style of diagram, the data flows from top to bottom, each black dot is a forking point for wires and each white circle is an occurrence of the binary operator. Our case study was heavily influenced by the paper “An Algebra of Scans” [12]. In this paper, the author defines a set of primitives and combinators from which any scan circuit can be built, then proves their algebraic properties.

Our work consisted of formalizing the same primitives and combinators using Π-Ware, and proving the same basic algebraic facts about these combinators. Also, we formalized what exactly means to be a scan circuit, and proved that applying *scan combinators* to scan circuits will result in a scan.

Several of the primitives and combinators defined in the original paper [12] match exactly those present in Π-Ware, amongst them sequential (_⟫_) and parallel composition (_||_) along with the identity plug (id⤨). This coincidence makes the case study especially fruitful, as several of the basic algebraic properties assumed in the original paper could be proved in Π-Ware. For example, sequential combination (_⟫_) forms a monoid of circuits, with id⤨ as identity:

```
⟫-left-identity : ∀ {i o} (c : ℂ i o) → id⤨ ⟫ c ≈ c
⟫-left-identity c = ≅⇒≈ (from-≡ ∘ cong ⟦ c ⟧ ∘ id⤨-id)
```

```
⟫-assoc : ∀ {i m n o} (c₁ : ℂ i m) (c₂ : ℂ m n) (c₃ : ℂ n o) → (c₁ ⟫ c₂) ⟫ c₃ ≈ c₁ ⟫ (c₂ ⟫ c₃)
⟫-assoc c₁ c₂ c₃ = ≅⇒≈ (from-≡ ∘ λ _ → refl)
```

In fact, the need to state and prove these basic algebraic laws was what led us to develop the notion of circuit equivalence (Section 5.4) in the first place. We predict that such algebraic structures over circuits will be important when reasoning about circuit transformations and synthesis. For example, the identity laws for id⤨ allow us to *remove* such plugs from any circuit, while being *certain* that the functional behaviour will not change.

```
serial : ∀ n → ℂ n n
serial zero            = id⤨ 0
serial (suc zero)      = id⤨ 1
serial (suc (suc n))   = serial (suc n) ▯ id⤨ 1
```

Listing 11 Π-Ware description of a serial scan.

The concept of scan circuit itself was formalized by defining a "prototype" scan, which was assumed to be correct. This definition is very inefficient (in terms of gate usage and also in depth), but has a very simple inductive definition:

```
scan : ∀ n → ℂ n n
scan zero =      id⤨₀
scan (suc n) =   id⤨₁ ∥ scan n ⟫ fan (suc n)
```

Besides the parts already mentioned (_⟫_, _∥_, id⤨), here we also use the fan primitive. A term "fan n" has type ℂ n n, and calculates $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_2), ..., (x_0 \oplus x_n)]$. The diagram in Figure 4 illustrates the structure of "scan 3".

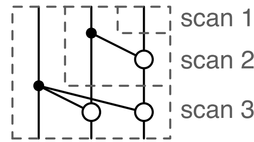


Figure 4 Structure of the prototype scan of size 3.

Using this specification, we could prove that several different architectures all indeed compute a scan. The proofs rely on the fact that all of these architectures are built by combining smaller scans into bigger ones.

Namely, we defined in Π-Ware the *sequential scan combinator* (called _▱_) and the *parallel scan combinator* (called _▯_). The sequential scan combinator connects the last output of its first argument into the first input of the second argument. For the parallel combinator both scan circuits are put side-by-side, and an extra fan connects the last output of the first argument to all inputs of the second.

Having defined those combinators, we then proved that their definitions indeed satisfy their namesake property: whenever given scans as arguments, they produce a scan as output:

```
▱-law  : scan (suc m) ▱ scan (suc n) ≈ scan (m + suc n)
▯-law  : scan (suc m) ▯ scan n       ≈ scan (suc m + n)
```

As an example of a proof involving lots of these algebraic properties, we show the correctness of a *serial scan*. A serial scan is a parallel prefix circuit of maximal depth, as it makes no use of parallelism at all, and it has the structure shown in Figure 5.

The Π-Ware description for serial is pretty simple and makes essential use of the parallel scan combinator (_▯_). The code for serial is shown in Listing 11

Finally, we prove that serial does indeed compute a scan. The proof (Listing 12) relies on the key fact that _▯_ preserves scans. Furthermore, it relies on the fact that _▯_ is a congruence with regards to circuit equivalence, and that two calls of scan with equal arguments will be equivalent (scan-cong).

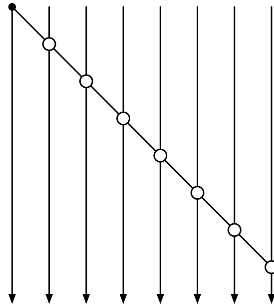


Figure 5 Structure of a serial scan.

```
serial-is-scan : ∀ n → serial n ≈ scan n
serial-is-scan zero            = ≈-refl
serial-is-scan (suc zero)      = id⤫₁≈scan₁
serial-is-scan (suc (suc n))   = begin
    serial (suc (suc n))
  ≈⟨⟩ - definition of serial (suc (suc n))
    serial (suc n) □ id⤫ 1
  ≈⟨ serial-is-scan (suc n) □-cong id⤫₁≈scan₁ ⟩
    scan (suc n) □ scan 1
  ≈⟨ □-law n 1 ⟩
    scan (suc n + 1)
  ≈⟨ scan-cong (cong suc (+-comm n 1)) ⟩
    scan (suc (suc n))
  ∎
```

.

Listing 12 Proof that serial computes a scan.

7 Discussion

7.1 Related Work

There are numerous languages for hardware description; there is a wide variety of techniques that may be used for hardware verification, including the usage of automatic theorem provers, SAT solvers, model checking, and interactive theorem provers, notably HOL [7].

Systems such as ACL2 have been used to prove correctness of entire microprocessors [14], and the maturity of the ACL2 and HOL ecosystems is clearly visible in the highly optimized engines and large scale of some of the formalization efforts done using these languages. One of the key differences with our approach is the use of a typed higher-order host language, with which we can also have *higher-order specifications* for *connection patterns*. For example, the behaviour of the parsN pattern is equivalent to a functorial *map* over vectors.

The field of formal methods and functional programming applied to hardware design is indeed a crowded one, thus rather than attempt to survey these fields here, we will restrict ourselves to the most closely related work. There has been a great deal of work in the last thirty years marrying functional programming and hardware design, leading to languages such as Lava [2], Hawk [16], Wired [1] and ForSyDe [19]; Sheeran [21] gives an excellent

overview. When comparing Π-Ware to these other DSLs, the overarching theme is the use by Π-Ware of a host language in which stronger static guarantees are provided and in which circuits and proofs live side-by-side.

Specifically when comparing Π-Ware and Lava, we can say:

- Lava uses observational sharing as a binding technique, while Π-Ware is combinator-based.
- Lava verification uses external tools, while Π-ware circuits and proofs share a language.
- In Lava, only specific instances of a circuit generator can be verified, while Π-ware exploits the inductive structure of generators for verifying the generators themselves.

ForSyDe is still closely related to Π-Ware in terms of its goals, but much less closely than Lava. It offers similar verification facilities to those of Lava (using external model checking tools), so the same comparison applies. However, ForSyDe offers a different "front-end" to the user, by using reflection features of Haskell (namely quasi-quoters). Another considerable restriction on ForSyDe is that the set of types supported as inputs and outputs of circuits is very restricted (booleans, sequences, tuples), and cannot easily be extended, while Π-Ware is designed to be as general as possible regarding these choices.

With regards to Π-Ware and Wired, both languages describe the *architecture* of circuits, but while Wired definitions specify the exact geometry and placement of components, Π-Ware only talks about *topology*. We recognize the usefulness of geometric descriptions, but we have chosen to focus on topology as our initial goal is to prove properties involving (the preservation of) functional correctness and circuit transformations with a space vs. time trade-off.

When looking at the literature for hardware verification methods, we note that the idea of using dependent types for circuit description is not new and can be traced back as far as Hanna [11]. The paper "Constructing Correct Circuits" [4] gives a clear example of how dependent types can *tie together* specification and implementation. In this paper, the authors give a mapping between Peano naturals and binary numbers, then used to build a (ripple-carry) binary adder which is *correct by construction*. The approach taken in Π-Ware is significantly different. Rather than carry the functional specification of a circuit *in its type*, we clearly separate the construction, testing, and verification of circuits. This means that, for example, a designer can first simulate some instances of a design and get confidence in its correctness before trying to *prove* it. This greater degree of freedom may be particularly useful when exploring the design space, deferring the testing and verification effort until a satisfactory candidate design has been found.

Some of the most complete EDSLs for hardware (Coquet [5] and Fe-Si [6]) are hosted in the Coq theorem prover. Our design and implementation has been particularly inspired by Coquet. Both Coquet and Π-Ware use a similar *structural* and *nameless* description of circuits, parameterized by the type of gates. The most important difference between Π-Ware and Coquet, however, is that Π-Ware defines a *functional* semantics for circuits, while Coquet uses a *relational* semantics, i.e., the semantics are specified by defining a suitably indexed inductive data type. The choice of semantics style is crucial: Π-Ware circuits can be tested, simulated, and verified as any other Agda function.

Where Coq's richer language for proof tactics may provide a great deal of automation, the functional semantics presented here reduces *for free*, without relying on the invocation of tactics or proof search. We expect to reap the benefits of a functional semantics while combining them with some proof-by-reflection techniques [23, 15]. Furthermore, we can use Agda features such as goal and context reflection, as well as solvers for algebraic structures (monoids, semirings, etc.) [3].

7.2 Future Work

Equality Plugs

When explaining the behaviour of Plugs in Section 3, we said that they perform no computation. But further than that, some plugs in fact have also *no structural effect*. By this we mean plugs whose mapping is the identity function. They usually are an expression of arithmetic equalities over circuit indices, such as associativity:

$$\text{assoc}⤫ : \forall \{a\ b\ c\} \rightarrow ((a + b) + c)\ ⤫\ (a + (b + c))$$
$$\text{assoc}⤫\ \{a\}\ \{b\}\ \{c\} = \text{eq}⤫\ (+\text{-assoc}\ a\ b\ c)$$

The need to place such a Plug between two circuits is essentially an artifact of Intensional Type Theory (ITT). In the sequence constructor (_⟫_), the output index of the first parameter and input index of the second must be *definitionally* equal, that is, they must have the same normal form. If Agda had the *equality reflection rule*, then equalities involving indices could be used during type checking, and we would not need to insert *equality plugs*.

Right now we are investigating two approaches to make this issue less inconvenient. Firstly, we can use the *ring solver* from Agda's standard library (coupled with reflection) to automatically solve index equalities and introduce the corresponding plugs whenever needed. Secondly, there was a recent addition to Agda of a language pragma called `REWRITE`, which allows for user-defined equalities to be added to Agda's typing rules, essentially turning Agda into an Extensional Type Theory (ETT). We will investigate how the use of this pragma affects our library and examples.

Functional Language

Even though we are using a functional language to model our circuits, the circuit description themselves are very low-level. In particular, we need to rewire intermediate results explicitly using our Plug constructors. While our style closely resembles the netlist representation of circuits, we would like to provide circuit designers with a more high-level, applicative interface.

One problem that we must address to do so, however, is that of observable sharing [8]. Any domain specific language for hardware description embedded in a general purpose functional language must, at some point, ensure that the sharing and recursion of the circuit definitions are not lost. Although various solutions do exist, these typically place a higher burden on the programmer through the necessity of explicit fixed-point and sharing combinators or rely on specific compiler support. We hope to find a satisfactory solution to this problem in the context of dependently typed programming languages such as Agda, and use this to define a more "functional" layer on top of the definitions presented here.

Typed Circuits

While Π-Ware rules out certain errors, such as short-circuits, we would like to investigate how to provide stronger static guarantees. So far, we have parameterized the type of circuits by the *size* of their inputs and outputs; we have started investigating how to parameterize circuits by their *type*.

For example, the type of a 2-input multiplexer would then become "ℂ^ (Bool × (Bool × Bool)) Bool", rather than the less informative "ℂ 3 1". To add extra type information to our circuits, we define a record wrapper for typed circuits (Listing 13).

```
record ℂ^ {s : IsComb} (A B : Set) {i j : ℕ} : Set where
  constructor Mkℂ^
  field ⦃ α^ ⦄ : ⇓W⇑ A {i}
        ⦃ β^ ⦄ : ⇓W⇑ B {j}
        base  : ℂ {s} i j
```

Listing 13 Typed Circuit type.

Here we require that the input and output types of our circuits are synthesizable – that is, they can indeed be represented in our simulation semantics (as vectors of atoms). By adding a series of *smart constructors* that produce and combine such typed circuits, we can provide a more convenient and type-safe interface to our library. We are currently extending our library with such type-safe definitions, including use of reflection to generate the required serializer/deserializer and proof that they are inverses.

8 Conclusion

With Π-Ware we have only started to explore the benefits that dependent types offer to digital circuit design. Π-Ware and the wider Agda ecosystem may not be mature enough yet to compete with some of the existing commercial tools and more mature prover technology; nonetheless we believe that the combination of the executable circuits, static types, and compositional proofs that Π-Ware offers form a novel point in the design space.

All the examples we have developed up to now, especially the case study on scan circuits, lead us to believe that this is indeed a fruitful avenue of study. By treating circuits as first-class objects in a dependently-typed language, we can reason about their behaviour and prove algebraic properties of relations, operators *over* circuits, and circuit generators. At the same time, we can simulate our designs and synthesize netlist descriptions. It should come as no surprise that type theory, a language of both computation and proof, provides a perfect setting for hardware verification and simulation.

Acknowledgements We would like to thank the helpful comments and suggestions of the attendants of the TYPES 2015 conference in Tallinn where we presented our initial results on Π-Ware. The participation in other venues such as for instance the Midlands Graduate School 2015 in Sheffield was also very fruitful in allowing discussions about the type-theoretical underpinnings of this work.

References

1 Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2005. doi:10.1007/11560548_4.

2 Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 174–184. ACM, 1998. doi:10.1145/289423.289440.

3 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9_6.

4 E. Brady, J. McKinna, and K. Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In M. T. Morazán, editor, *Trends in Functional Programming 8*, pages 159–176. Intellect, 2007.

5 Thomas Braibant. Coquet: A Coq library for verifying hardware. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2011. doi:10.1007/978-3-642-25379-9_24.

6 Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013. doi:10.1007/978-3-642-39799-8_14.

7 A. Camilleri, M. Gordon, and T. F. Melham. Hardware verification using higher-order logic. Technical Report 91, University of Cambridge Computer Laboratory, 1986. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-91.pdf.

8 Koen Claessen and David Sands. Observable sharing for functional circuit description. In P. S. Thiagarajan and Roland H. C. Yap, editors, *Advances in Computing Science - ASIAN'99, 5th Asian Computing Science Conference, Phuket, Thailand, December 10-12, 1999, Proceedings*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999. doi:10.1007/3-540-46674-6_7.

9 Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In Ravi Iyer, Qing Yang, and Antonio González, editors, *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pages 365–376. ACM, 2011. doi:10.1145/2000064.2000108.

10 Morteza Fayyazi and Laurent Kirsch. Efficient simulation of oscillatory combinational loops. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 777–780. ACM, 2010. doi:10.1145/1837274.1837470.

11 F. K. Hanna and N. Daeche. Dependent types and formal synthesis. *Philos. Trans. Phys. Sci. Eng.*, 339(1652):121–135, 1992. URL: http://www.jstor.org/stable/54016.

12 Ralf Hinze. An algebra of scans. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004. doi:10.1007/978-3-540-27764-4_11.

13 IEEE standard VHDL language reference manual: IEEE standard 1076-1987, 1988. doi:10.1109/ieeestd.1988.122645.

14 Warren A. Hunt Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Computer Science*. Springer, 1994. doi:10.1007/3-540-57960-5.

15 Pepijn Kokke and Wouter Swierstra. Auto in Agda - programming proof search using reflection. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29*

- *July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 276–301. Springer, 2015. `doi:10.1007/978-3-319-19797-5_14`.

16 John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In Didier Rémy and Peter Lee, editors, *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, pages 60–69. ACM, 1999. `doi:10.1145/317636.317784`.

17 U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007. URL: `http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf`.

18 Nicolas Oury and Wouter Swierstra. The power of pi. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50. ACM, 2008. `doi:10.1145/1411204.1411213`.

19 I. Sander and A Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. Comput. Aided Des. Integ. Circ. Syst.*, 23(1):17–32, 2004. `doi:10.1109/tcad.2003.819898`.

20 M. Sheeran. muFP, a language for VLSI design. In *Proc. 1984 ACM Symp. on LISP and Functional Programming, LFP '84*, pages 104–112. ACM, 1984. `doi:10.1145/800055.802026`.

21 Mary Sheeran. Hardware design and functional programming: a perfect match. *J. UCS*, 11(7):1135–1158, 2005. `doi:10.3217/jucs-011-07-1135`.

22 Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 133–144. ACM, 2014. `doi:10.1145/2633357.2633360`.

23 Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2012. `doi:10.1007/978-3-642-41582-1_10`.