

# 26th International Conference on Types for Proofs and Programs

TYPES 2020, March 2–5, 2020, University of Turin, Italy

Edited by

Ugo de'Liguoro

Stefano Berardi

Thorsten Altenkirch



*Editors*

**Ugo de'Liguoro**

University of Turin, Italy  
ugo.deliguoro@unito.it

**Stefano Berardi**

University of Turin, Italy  
stefano.berardi@unito.it

**Thorsten Altenkirch** 

University of Nottingham, UK  
Thorsten.Altenkirch@nottingham.ac.uk

*ACM Classification 2012*

Theory of computation → Proof theory; Theory of computation → Logic and verification; Software and its engineering → Formal software verification; Security and privacy → Systems security; Theory of computation → Type theory; Theory of computation → Modal and temporal logics; Theory of computation → Higher order logic; Theory of computation → Equational logic and rewriting; Theory of computation → Linear logic; Theory of computation → Constructive mathematics; Theory of computation → Complexity theory and logic

**ISBN 978-3-95977-182-5**

*Published online and open access by*

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-182-5>.

*Publication date*

June, 2021

*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 <https://portal.dnb.de>.

*License*

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): <https://creativecommons.org/licenses/by/4.0/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.2020.0

ISBN 978-3-95977-182-5

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

## LIPICs – Leibniz International Proceedings in Informatics

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

### *Editorial Board*

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

**ISSN 1868-8969**

**<https://www.dagstuhl.de/lipics>**



## ■ Contents

Preface	
<i>Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch</i> .....	0:vii
<b>Regular Papers</b>	
On Model-Theoretic Strong Normalization for Truth-Table Natural Deduction	
<i>Andreas Abel</i> .....	1:1–1:21
Extending Equational Monadic Reasoning with Monad Transformers	
<i>Reynald Affeldt and David Nowak</i> .....	2:1–2:21
Towards a Certified Reference Monitor of the Android 10 Permission System	
<i>Guido De Luca and Carlos Luna</i> .....	3:1–3:18
Coinductive Proof Search for Polarized Logic with Applications to Full Intuitionistic Propositional Logic	
<i>José Espírito Santo, Ralph Matthes, and Luís Pinto</i> .....	4:1–4:24
Synthetic Completeness for a Terminating Seligman-Style Tableau System	
<i>Asta Halkjær From</i> .....	5:1–5:17
Encoding of Predicate Subtyping with Proof Irrelevance in the $\lambda\Pi$ -Calculus Modulo Theory	
<i>Gabriel Hondet and Frédéric Blanqui</i> .....	6:1–6:18
$\Lambda$ -Symsym: An Interactive Tool for Playing with Involutions and Types	
<i>Furio Honsell, Marina Lenisa, and Ivan Scagnetto</i> .....	7:1–7:18
Why Not W?	
<i>Jasper Hugunin</i> .....	8:1–8:9
Subtype Universes	
<i>Harry Maclean and Zhaohui Luo</i> .....	9:1–9:16
Two Applications of Logic Programming to Coq	
<i>Matteo Manighetti, Dale Miller, and Alberto Momigliano</i> .....	10:1–10:19
Duality in Intuitionistic Propositional Logic	
<i>Paweł Urzyczyn</i> .....	11:1–11:10





## ■ Preface

This volume constitutes the post-proceedings of the 26th International Conference on Types for Proofs and Programs, TYPES 2020, that was planned in Turin from the 2nd to the 5th of March 2020. The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalised 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. Previous TYPES meetings were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Bergen 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), Tallinn (2015), Novi Sad (2016), Budapest (2017), Braga (2018) and Oslo (2019).

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 of open and informal character. Selection of contributed talks 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 volume must represent unpublished work and are subjected to a full peer-review process.

Due to the COVID-19 outbreak, in 2020 the conference did not take place; the abstracts of contributed talks are available from: <https://types2020.di.unito.it/>. Nonetheless the steering committee decided to have TYPES 2020 post-proceedings published. After a new call and a thorough peer-review procedure, 11 submissions could be accepted for publication. We thank all authors and reviewers for their contribution to this volume.

Ugo de'Liguoro, Stefano Berardi and Thorsten Altenkirch, June 2021







# On Model-Theoretic Strong Normalization for Truth-Table Natural Deduction

Andreas Abel   

Department of Computer Science and Engineering,  
Chalmers University of Technology, Göteborg, Sweden  
Gothenburg University, Göteborg, Sweden

---

## Abstract

Intuitionistic truth table natural deduction (ITTND) by Geuvers and Hurkens (2017), which is inherently non-confluent, has been shown strongly normalizing (SN) using continuation-passing-style translations to parallel lambda calculus by Geuvers, van der Giessen, and Hurkens (2019). We investigate the applicability of standard model-theoretic proof techniques and show (1) SN of detour reduction ( $\beta$ ) using Girard’s reducibility candidates, and (2) SN of detour and permutation reduction ( $\beta\pi$ ) using biorthogonals. In the appendix, we adapt Tait’s method of saturated sets to  $\beta$ , clarifying the original proof of 2017, and extend it to  $\beta\pi$ .

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Proof theory

**Keywords and phrases** Natural deduction, Permutative conversion, Reducibility, Strong normalization, Truth table

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.1

**Supplementary Material** *Software (Source Code)*: <https://github.com/andreasabel/truthtable> archived at `swh:1:dir:ad022539d7490ea9943b3f497f41ab2108acebc2`

**Funding** This work has been supported by grant no. 2019-04216 *Modal Dependent Type Theory* of the Swedish Research Council (Vetenskapsrådet).

**Acknowledgements** Thanks to Herman Geuvers for explaining me truth-table natural deduction during a 2018 visit to Nijmegen for the purpose of Henning Basold’s PhD ceremony. Thanks to Ralph Matthes, Herman Geuvers and Tonny Hurkens for some email discussions on the topics of this article. I am also grateful for the feedback of the reviewer that led to a substantial clarification of the proof using orthogonality.

## 1 Introduction

Recently, Geuvers and Hurkens [13] have observed that, departing from the truth table of a logical connective, one can in a schematic way construct introduction and elimination rules for that connective both for intuitionistic and classical natural deduction. For each line in the truth table where the connective computes to *true* one obtains an introduction rule, and for the *false* lines one obtains an elimination rule. It is shown that these *truth table natural deduction* (TTND) calculi are equivalent to Gentzen’s original calculi [12] in the sense that the same judgements can be derived. However, the schematic rules are sometimes unwieldy and unintuitive – for instance, in TTND there are three introduction rules for implication since  $A \rightarrow B$  is true for three out of four valuations of  $(A, B)$ . As a remedy, Geuvers and Hurkens show how the original TTND rules can be optimized in a systematic way. In this article, we shall confine ourselves to the schematic, unoptimized rules of intuitionistic TTND (ITTND).

When studying proof terms and proof normalization for ITTND, one can observe that  $\beta$ -reduction – the reduction of detours, i.e., introductions followed directly by eliminations<sup>1</sup> – is essentially non-deterministic and even non-confluent. Non-confluence poses some challenges

<sup>1</sup> Geuvers and Hurkens call detour redexes *direct intuitionistic cuts* [13] or *a*-redexes [14] and with van der Giessen D-redexes [16]. We follow Joachimski and Matthes [20] and call detour reductions simply  $\beta$ -reductions, as these are a generalization of the  $\beta$ -reduction of  $\lambda$ -calculus.



© Andreas Abel;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 1; pp. 1:1–1:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to the proof that reduction is terminating, the so-called strong normalization (SN) property. In the original presentation [13], the authors confine the SN proof to ITTND with a single but universal connective *if-then-else* and the optimized inference rules for if-then-else which yield confluent and standardizing  $\beta$ -reduction. The proof follows the *saturated sets* method pioneered by Tait [30] which is known to rely on standardization by using deterministic weak head reduction.<sup>2</sup>

In subsequent work [14], the authors attack SN for full ITTND with non-confluent  $\beta$ -reduction, introducing elements of Girard’s technique of *reducibility candidates* (RCs) [17, 19]. However, this innovative mix of Tait and Girard is not without pitfalls, as we shall investigate in Section 4.2. We tread on safer grounds by returning to Girard’s original definition of RCs in Section 4. Our proof in Section 4.1 relies on impredicativity and could not be formalized in a predicative metatheory such as Martin-Löf Type Theory [24]. We thus give in Section 4.3 a variant that replaces the use of impredicativity by inductive definitions.

However,  $\beta$ -reduction is not the only form of proof optimization in ITTND. The schematic elimination rules of ITTND have the flavor of disjunction elimination which does not pose any restriction on the formula on the right. Likewise, eliminations in ITTND have an arbitrary target. In such settings, one eliminates a hypothesis to directly prove the desired conclusion. Eliminating into an intermediate conclusion which is then eliminated again is thus considered a detour. Joachimski and Matthes [20] call such a detour a *permutation redex* or  $\pi$ -redex<sup>3</sup> – in the context of intuitionistic sequent calculus restricted to implication. Permutation reduction for ITTND by itself is terminating [14], and in *loc. cit.* it is shown that the free combination with  $\beta$ -reduction,  $\beta\pi$ , is weakly normalizing. Strong normalization was left open until the joint work of Geuvers and Hurkens with van der Giessen [16], where SN was established via a continuation-passing-style (CPS) translation to the parallel simply-typed lambda calculus (parallel STLC).<sup>4</sup>

The change of proof strategy begs the question whether the usual model-theoretic SN proofs could not work also for  $\beta\pi$ -reduction. While the saturated sets method applied in a similar situation by Joachimski and Matthes [20] seems not applicable due to non-confluence of  $\beta$ , Girard’s RCs do not cover  $\pi$ . However, there is a third popular method, *(bi)orthogonals*, that has been developed to prove SN for classical lambda-calculi which are essentially non-confluent.<sup>5</sup> Biorthogonals have been successfully applied by Lindley and Stark [22] to prove SN for Moggi’s “monadic metalanguage”, that is STLC with introduction, elimination, and permutation rules for the monad. We show in Section 6 that biorthogonals, putting elimination sequences at the center of attention, can show SN for  $\beta\pi$  of ITTND. Finally, in the Appendix A, we demonstrate how the the saturated sets method can also be adapted.

While we limit our presentation on the implicational fragment of ITTND for didactic purposes and convenience of exposition, our techniques scale immediately to the general case.

## Overview

In Section 2 we recapitulate Geuvers and Hurkens’ construction of intuitionistic inference rules from truth tables and the associated  $\beta$ -rules. In Section 3 we present a common structure of model-theoretic SN proofs. This structure is instantiated to RCs in Section 4 and we present the two ways of constructing the interpretation of the connectives: via the

<sup>2</sup> Weak head reduction is sometimes called *key reduction* in the context of saturated sets.

<sup>3</sup> Geuvers and Hurkens call  $\pi$ -redexes *b-redexes* [14] and, with van der Giessen, *P-redexes* [16].

<sup>4</sup> In a first approximation, one can think of parallel STLC as STLC with explicit non-determinism.

<sup>5</sup> Early applications of orthogonality can be found in the works of Parigot [27, 28] and Barbanera and Berardi [4].

elimination rules (Section 4.1) and via the introduction rules (Section 4.3). Further, we take a critical look at the proof of Geuvers and Hurkens [14] in Section 4.2. In Section 5 we turn to  $\pi$ -reduction, laying some foundation for the SN proof for  $\beta\pi$  using orthogonality (Section 6), which is the main contribution of this paper. We conclude with a short discussion in Section 7.

## 2 Intuitionistic Truth Table Natural Deduction

Geuvers and Hurkens [13] introduced a method to derive natural deduction proof rules from truth tables of logical connectives. For instance, consider the truth table for implication:

$A$	$B$	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

For each line where  $A \rightarrow B$  holds, e.g., the second line, an introduction rule is created where 0-valued (or *negative*) operands  $A$  become premises  $\Gamma.A \vdash A \rightarrow B$  and 1-valued (or *positive*) operands  $B$  become premises  $\Gamma \vdash B$ . Lines like the third where  $A \rightarrow B$  is false become elimination rules with a conclusion  $\Gamma \vdash C$  for an arbitrary formula  $C$ . The premises of this elimination rule are, besides the principal premise  $\Gamma \vdash A \rightarrow B$ , a premise  $\Gamma \vdash A$  for each 1-valued operand  $A$ , and a premise  $\Gamma.B \vdash C$  for each 0-valued operand  $B$ . This yields the following four rules of judgement  $\boxed{t : \Gamma \vdash A}$ :<sup>6</sup>

$$\frac{t : \Gamma.A \vdash A \rightarrow B \quad u : \Gamma.B \vdash A \rightarrow B}{\text{in}_{\rightarrow}^{00}(t, u) : \Gamma \vdash A \rightarrow B} \quad \frac{t : \Gamma.A \vdash A \rightarrow B \quad b : \Gamma \vdash B}{\text{in}_{\rightarrow}^{01}(t, b) : \Gamma \vdash A \rightarrow B}$$

$$\frac{f : \Gamma \vdash A \rightarrow B \quad a : \Gamma \vdash A \quad t : \Gamma.B \vdash C}{f \cdot \text{el}_{\rightarrow}^{10}(a, t) : \Gamma \vdash C} \quad \frac{a : \Gamma \vdash A \quad b : \Gamma \vdash B}{\text{in}_{\rightarrow}^{11}(a, b) : \Gamma \vdash A \rightarrow B}$$

As seen from these instances, we preferably use letters  $t, u, v$  for terms with a distinguished hypothesis and letters  $a, b, c, d, e, f$  for terms without such. Replacing the distinguished hypothesis, i.e., the 0th de Bruijn index, in term  $t$  by a term  $a$  is written  $t[a]$ . We use letter  $I$  for introduction terms, i.e., such with “in” at the root, and letter  $E$  for an elimination in term  $f \cdot E$ , i.e., the “el” part. Heads  $h$  are either variables  $x$  or introductions  $I$ , and each term can be written in spine form  $h \cdot E_1 \cdot \dots \cdot E_n$ . This may be written  $h \cdot \vec{E}$ .

*Detour* or  $\beta$  reductions can fire when an introduction is immediately eliminated, i.e., on well-typed subterms of the form  $I \cdot E$ . For the case of implication, there are three introduction rules that can be paired with the only elimination rule. There are two ways in which a  $\beta$

<sup>6</sup> Additional information for the reader unfamiliar with natural deduction and proof terms: Natural deduction asserts the truth of a proposition  $A$  under a list of assumed propositions  $\Gamma$ , a *context*, via the judgement  $\Gamma \vdash A$ . Derivations of such judgements form proof trees where nodes are labeled by the name of the applied proof rule and the ordered subtrees correspond to the premises of that rule. Leaves are either applications of a rule that has no premises or references to one of the hypotheses in  $\Gamma$ . We write  $\varepsilon$  for empty lists. The list  $\Gamma$  can be extended on the right by a proposition  $A$  using the notation  $\Gamma.A$ . Following de Bruijn [11], we number the hypotheses from the right starting with zero. A reference to a hypothesis – a so-called *de Bruijn index* – is a non-negative number  $i$  strictly smaller than the length of  $\Gamma$ . For example, de Bruijn index zero, written  $x_0$ , refers to proposition  $A$  in context  $\Gamma.A$ . We write  $x : \Gamma \vdash A$  to denote a de Bruijn index  $x$  pointing to proposition  $A$  in context  $\Gamma$ . In general, we use the notation  $t : \Gamma \vdash A$  to state that  $t$  is a valid proof tree, also called proof term, whose conclusion is the judgement  $\Gamma \vdash A$ . We will only refer to terms  $t$  that correspond to a valid proof tree, thus, we consider terms as intrinsically typed [3, 5]. This choice however affects neither presentation nor results in this article very much; they apply the same to extrinsic typing.

redex can fire: Either, a positive premise (1) of the introduction matches a negative premise (0) of the elimination. For the case of implication, the second premise of the elimination  $\text{el}_{\rightarrow}^{10}$  is negative, and it can react with the positive second premise of  $\text{in}_{\rightarrow}^{01}$  and  $\text{in}_{\rightarrow}^{11}$ :

$$\text{in}_{\rightarrow}^1(\_, b) \cdot \text{el}_{\rightarrow}^{10}(\_, t) \mapsto_{\beta} t[b]$$

The other reaction is between a negative premise of the introduction and a matching positive premise of the elimination. In this case, the elimination persists, but the introduction is replaced with an instantiation of its respective negative premise. In the case of implication, the first premise of  $\text{in}_{\rightarrow}^{00}$  and  $\text{in}_{\rightarrow}^{01}$  can be instantiated with the first premise of  $\text{el}_{\rightarrow}^{10}$ :

$$\text{in}_{\rightarrow}^0(u, \_) \cdot \text{el}_{\rightarrow}^{10}(a, t) \mapsto_{\beta} u[a] \cdot \text{el}_{\rightarrow}^{10}(a, t)$$

The case of implication already demonstrates the inherent non-confluence of  $\beta$ -reduction: the reducts of  $\text{in}_{\rightarrow}^{01}(u, b) \cdot \text{el}_{\rightarrow}^{10}(a, t)$  form the critical pair  $(t[b], u[a] \cdot \text{el}_{\rightarrow}^{10}(a, t))$  which can in general not be joined. Non-confluence excludes some techniques to show strong normalization, e.g., those that rely on deterministic weak head reduction. However, Girard's reducibility candidates accommodate non-confluent reduction, thus, his technique may be adapted to the present situation.

### 3 Model-theoretic proofs of strong normalization

In this section we explain the general format of a model-theoretic proof of strong normalization. We will instantiate this framework to two techniques later: reducibility candidates (Section 4) and orthogonality (Section 6).

#### 3.1 Preliminaries

We work with sets  $\Gamma \vdash A$  of nameless well-typed terms. De Bruijn indices are written  $x_n : \Gamma.A.\Delta \vdash A$  where  $\Delta$  has length  $n$ . Instead of full-fledged renaming, we confine to weakening under order-preserving embeddings (OPE)  $\boxed{\tau : \Delta \leq \Gamma}$ . Here,  $\tau$  witnesses that and how  $\Gamma$  is a subsequence of  $\Delta$ . Then,  $\uparrow\tau : \Delta.B \leq \Gamma.B$  be the *lifted* OPE. Further,  $\uparrow : \Gamma.B \leq \Gamma$  is the OPE for weakening by one variable, and OPEs form a category with identity  $\mathbb{1} : \Gamma \leq \Gamma$  and composition  $(\Gamma \leq \Delta) \rightarrow (\Delta \leq \Phi) \rightarrow (\Gamma \leq \Phi)$  written as juxtaposition. If  $a : \Gamma \vdash A$  then *weakening*  $a\tau : \Delta \vdash A$  is defined in the usual way. In particular,  $\uparrow$  is used to traverse under binders, for instance,  $\text{in}_{\rightarrow}^{01}(t, b)\tau = \text{in}_{\rightarrow}^{01}(t(\uparrow\tau), b\tau)$ .

Substitutions  $\boxed{\sigma : \Delta \vdash \Gamma}$  are defined as lists of terms  $\sigma = \varepsilon.b_1 \cdots .b_{|\Gamma|}$  typed by list  $\Gamma$  under context  $\Delta$ . Parallel substitution  $a\sigma : \Delta \vdash A$  for  $a : \Gamma \vdash A$  is defined as usual. OPEs  $\tau : \Delta \leq \Gamma$  are silently coerced to substitutions  $\Delta \vdash \Gamma$  consisting only of de Bruijn indices. Substitutions form a category, and we reuse  $\mathbb{1}$  for identity and juxtaposition for substitution. Like for OPEs, we have lifting  $\uparrow : (\Delta \vdash \Gamma) \rightarrow (\Delta.B \vdash \Gamma.B)$  to push substitutions under binders. Single substitution  $t[b]$  is an instance of parallel substitution  $t\sigma$  for substitution  $\sigma = \mathbb{1}.b : (\Gamma \vdash \Gamma.B)$  obtained from  $b : \Gamma \vdash B$ .

*Reduction*  $\boxed{a \rightarrow a'}$ , which is defined using single substitution, acts on same-typed terms  $a, a' : \Gamma \vdash A$  by definition. It is closed under weakening and substitution. It is even closed under *anti-weakening*, i.e., if  $a\tau \rightarrow a'\tau$  then also  $a \rightarrow a'$ . (Not so for substitution: it is not closed under anti-substitution, of course.) Further, reduction commutes with weakening: If  $a\tau \rightarrow b'$  then there is  $b$  with  $a \rightarrow b$  and  $b' = b\tau$ .

Via the parallel substitution operation, the family  $\_ \vdash A$  of terms of type  $A$  is a contravariant functor (i.e., presheaf) targeting the category **Set** of sets and functions. Its source is the category of substitutions, and thus also its subcategory OPE. We will work a lot

with presheaves of the latter kind, especially with families of predicates  $P_\Gamma \subseteq (\Gamma \vdash A)$  closed under weakening, meaning if  $a \in P_\Gamma$  and  $\tau : \Delta \leq \Gamma$  then  $a\tau \in P_\Delta$ . We call such predicates *term set families*. We may simply write  $a \in P$  instead of  $a \in P_\Gamma$  if  $\Gamma$  is fixed but arbitrary or can be determined by the context.

Our prime example of a term set family are the strongly normalizing terms  $\text{SN}$  given inductively by rule

$$\frac{(a \rightarrow \_) \subseteq \text{SN}}{a \in \text{SN}}.$$

While it is formally a family of inductive predicates on well-typed terms  $a : \Gamma \vdash A$ , we mostly write  $a \in \text{SN}$  instead of  $a \in \text{SN}(\Gamma \vdash A)$  for simplicity. The set  $\text{SN}$  is closed under weakening, i.e., if  $\tau : \Delta \leq \Gamma$  then  $a\tau \in \text{SN}$  as well. This follows easily from anti-weakening for reduction.

### 3.2 Semantic types and normalization proofs

A typical model-theoretic proof of strong normalization will interpret types  $A$  by families  $\mathcal{A} = \llbracket A \rrbracket$  of strongly normalizing terms of type  $A$ . To work smoothly for open terms, a further requirement on such semantic types  $\mathcal{A}$  is that they contain the variables, i.e., if  $x : \Gamma \vdash A$  then  $x \in \mathcal{A}_\Gamma$ .

To obtain a compositional interpretation of types, each type constructor such as implication  $A \rightarrow B$  is interpreted by a suitable operation  $\mathcal{A} \rightarrow \mathcal{B}$  on semantic types. For pure implicational truth table natural deduction, types are formed from uninterpreted base types  $o$  (propositional variables) and function space:  $A, B ::= o \mid A \rightarrow B$ . Types are interpreted as the following semantic types:

$$\begin{aligned} \llbracket o \rrbracket_\Gamma &= \text{SN}(\Gamma \vdash o) \\ \llbracket A \rightarrow B \rrbracket_\Gamma &= (\llbracket A \rrbracket \rightarrow \llbracket B \rrbracket)_\Gamma \end{aligned}$$

The main structure of the normalization proof then proceeds as follows: Contexts  $\Gamma$  are interpreted as families of sets of substitutions.

$$\begin{aligned} \llbracket \varepsilon \rrbracket_\Delta &= \Delta \vdash \varepsilon \quad (= \{\sigma \mid \sigma : \Delta \vdash \varepsilon\}) \\ \llbracket \Gamma.A \rrbracket_\Delta &= \{\sigma.a \mid \sigma \in \llbracket \Gamma \rrbracket_\Delta \text{ and } a \in \llbracket A \rrbracket_\Delta\} \end{aligned}$$

Thanks to the requirement that the variables inhabit the semantic types, each context can be valuated by the identity substitution:

► **Lemma 1** (Identity substitution).  $\mathbb{1} \in \llbracket \Gamma \rrbracket_\Gamma$ .

**Proof.** By induction on  $\Gamma$ . In case  $\Gamma.A$ , we have  $\mathbb{1} \in \llbracket \Gamma \rrbracket_\Gamma$  by induction hypothesis, thus, by weakening,  $\uparrow \in \llbracket \Gamma \rrbracket_{\Gamma.A}$ . Further, the 0th de Bruijn index  $x_0 \in \llbracket A \rrbracket_{\Gamma.A}$ . Thus  $(\uparrow.x_0) = \mathbb{1} \in \llbracket \Gamma.A \rrbracket_{\Gamma.A}$ . ◀

The main theorem shows that each well-typed term inhabits the corresponding semantic type:

► **Theorem 2** (Fundamental theorem of logical predicates). *If  $a : \Gamma \vdash A$  and  $\sigma \in \llbracket \Gamma \rrbracket_\Delta$  then  $a\sigma \in \llbracket A \rrbracket_\Delta$ .*

Normalization is then a direct consequence:

► **Corollary 3** (Strong normalization). *If  $a : \Gamma \vdash A$  then  $a \in \text{SN}$ .*

**Proof.** By Theorem 2 with Lemma 1,  $a \mathbb{1} = a \in \llbracket A \rrbracket_\Gamma$ , thus,  $a \in \text{SN}$  since each semantic type contains only strongly normalizing terms.  $\blacktriangleleft$

The definition of the semantic types such as  $\mathcal{A} \rightarrow \mathcal{B}$  needs be crafted such as to allow us to prove Theorem 2. In the next section we identify the necessary properties.

### 3.3 Modelling the inference rules

To formulate the properties that allow us to prove Theorem 2 we introduce an auxiliary construction  $\boxed{\mathcal{C}[\mathcal{A}]}$ , “*abstraction*”, given semantic types  $\mathcal{A}$  and  $\mathcal{C}$ , where  $\mathcal{A}$  classifies terms of type  $A$  and  $\mathcal{C}$  terms of type  $C$ .

$$\mathcal{C}[\mathcal{A}]_\Gamma = \{t \in \Gamma.A \vdash C \mid t(\tau.a) \in \mathcal{C}_\Delta \text{ for all } \tau : \Delta \leq \Gamma \text{ and } a \in \mathcal{A}_\Delta\}.$$

The abstraction<sup>7</sup>  $\mathcal{C}[\mathcal{A}]$  is a presheaf via the weakening with the lifted OPE:

► **Lemma 4.** *If  $\tau : \Delta \leq \Gamma$  and  $t \in \mathcal{C}[\mathcal{A}]_\Gamma$  then  $t(\uparrow\tau) : \mathcal{C}[\mathcal{A}]_\Delta$ .*

**Proof.** Assume  $\tau' : \Phi \leq \Delta$  and  $a \in \mathcal{A}_\Phi$  and show  $t(\uparrow\tau)(\tau'.a) \in \mathcal{C}_\Phi$ . Since  $(\uparrow\tau)(\tau'.a) = \tau\tau'.a$  this follows by definition of  $t \in \mathcal{C}[\mathcal{A}]_\Gamma$ .  $\blacktriangleleft$

Using abstraction, the properties of the semantic connective can be mechanically obtained from the inference rules for the syntactic connective. In the formulation of these properties, a judgement  $a : \Gamma \vdash A$  turns into statement  $a \in \mathcal{A}_\Gamma$  and a judgement  $t : \Gamma.A \vdash C$  into  $t \in \mathcal{C}[\mathcal{A}]_\Gamma$ . In case of semantic implication  $\mathcal{A} \rightarrow \mathcal{B}$ , we obtain the following four requirements, one for each rule:

- ( $\text{in}_{\rightarrow}^{00}$ ) If  $t \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]$  and  $u \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{B}]$  then  $\text{in}_{\rightarrow}^{00}(t, u) \in \mathcal{A} \rightarrow \mathcal{B}$ .
- ( $\text{in}_{\rightarrow}^{01}$ ) If  $t \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]$  and  $b \in \mathcal{B}$  then  $\text{in}_{\rightarrow}^{01}(t, b) \in \mathcal{A} \rightarrow \mathcal{B}$ .
- ( $\text{in}_{\rightarrow}^{11}$ ) If  $a \in \mathcal{A}$  and  $b \in \mathcal{B}$  then  $\text{in}_{\rightarrow}^{11}(a, b) \in \mathcal{A} \rightarrow \mathcal{B}$ .
- ( $\text{el}_{\rightarrow}^{10}$ ) If  $f \in \mathcal{A} \rightarrow \mathcal{B}$  and  $a \in \mathcal{A}$  and  $t \in \mathcal{C}[\mathcal{B}]$  then  $f \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}$ .

Given these properties of semantic implication, we can show that semantic types model the inference rules:

**Proof of Theorem 2.** By induction on  $t : \Gamma \vdash C$ , prove  $t\sigma \in \llbracket C \rrbracket_\Delta$  for all  $\sigma \in \llbracket \Gamma \rrbracket_\Delta$ . In case of a variable  $t = x$ , we have  $x\sigma \in \llbracket \Gamma(x) \rrbracket_\Delta$  by assumption on  $\sigma$ .

The other cases are covered by the assumptions on semantic implication. For instance, consider:

$$\frac{t : \Gamma.A \vdash A \rightarrow B \quad b : \Gamma \vdash B}{\text{in}_{\rightarrow}^{01}(t, b) : \Gamma \vdash A \rightarrow B}$$

By induction hypothesis (2)  $b\sigma \in \llbracket B \rrbracket_\Delta$  and (1)  $t(\sigma\tau.a) \in \llbracket A \rightarrow B \rrbracket_\Phi$  for arbitrary  $\tau : \Phi \leq \Delta$  and  $a \in \llbracket A \rrbracket_\Phi$ , since then  $\sigma\tau \in \llbracket \Gamma \rrbracket_\Phi$ . Hence,  $t(\uparrow\sigma) \in (\llbracket A \rightarrow B \rrbracket)(\llbracket A \rrbracket)_\Delta$  by definition of abstraction. By property ( $\text{in}_{\rightarrow}^{01}$ ), it follows that  $\text{in}_{\rightarrow}^{01}(t, b)\sigma = \text{in}_{\rightarrow}^{01}(t(\uparrow\sigma), b\sigma) \in \llbracket A \rightarrow B \rrbracket_\Delta$ .  $\blacktriangleleft$

This completes the description of our framework for strong normalization proofs. We now turn our attention to ways how to instantiate this framework.

<sup>7</sup> Matthes [25, Sec. 6.2] uses the notation  $S_x(\mathcal{A}, \mathcal{C})$  for abstraction (in a setting with named variables  $x$ ).

### 3.4 Flavors of semantic types

We are familiar with three principal methods how to construct semantic types for strong normalization proofs.

1. *Saturated sets* following Tait [30], see e.g. the exposition by Luo [23]. This technique requires semantic types to be closed under weak head expansion and is only known to work for deterministic weak head reduction. While it has been applied [13] to the *if-then-else* instance of ITTND with optimized rules, it does not cover the general case of TTND with non-deterministic and even non-confluent weak head reduction.
2. *Reducibility candidates* following Girard [17, 19]. We apply this method in Section 4. It covers  $\beta$ -reduction but not  $\beta\pi$ .
3. *Biorthogonals* that have been used in SN proofs for  $\lambda$ -calculi for classical logic, e.g. by Parigot [27], and in SN proofs for the monadic meta-language by Lindley and Stark [22]. These cover even  $\beta\pi$ , and we shall turn to these in Section 6.

## 4 Reducibility Candidates

Girard's reducibility candidates are a flavor of semantic types that can show strong normalization also for non-confluent rewrite relations such as reduction in truth-table natural deduction.

When defining the semantic versions of the logical connectives such as  $\mathcal{A} \rightarrow \mathcal{B}$ , we have the choice to base the definition either on the introduction rules or the elimination rules.<sup>8</sup> We will study both approaches, but first, we recapitulate the definition of reducibility candidates.

Let **Intro** be the term set of introductions, i.e., the terms of the form  $\text{in}_c^{\vec{b}}(\vec{t})$ . This set is clearly closed under weakening and anti-weakening.

A *reducibility candidate*  $\mathcal{A}$  for a type  $A$  is a term set family with the following properties:

CR1  $\mathcal{A}_\Gamma \subseteq \text{SN}$ .

CR2 If  $a \in \mathcal{A}_\Gamma$  and  $a \rightarrow a'$  then  $a' \in \mathcal{A}_\Gamma$ .

CR3 For  $a : \Gamma \vdash A$ , if  $a \notin \text{Intro}$  and  $(a \rightarrow \_) \subseteq \mathcal{A}_\Gamma$ , then  $a \in \mathcal{A}_\Gamma$ .

We write  $\boxed{\mathcal{A} \in \text{CR}}$  if  $\mathcal{A}$  is a term set family satisfying CR1-3. It is easy to see that  $\text{SN} \in \text{CR}$ . If  $\mathcal{A}$  satisfies only CR1/2, it shall be called a *precandidate*.

Term set abstraction operates on precandidates:

► **Lemma 5 (Abstraction).** *Let  $\mathcal{A}_\Gamma$  be inhabited for any  $\Gamma$ . If  $\mathcal{C}$  is a precandidate, so is  $\mathcal{C}[\mathcal{A}]$ .*

**Proof.** CR1 holds by non-emptiness of  $\mathcal{A}$ : Given  $t \in \mathcal{C}[\mathcal{A}]_\Gamma$  and arbitrary  $a \in \mathcal{A}_\Gamma$  we have  $t[a] \in \mathcal{C}_\Gamma$ . In particular,  $t[a] \in \text{SN}$ , and thus,  $t \in \text{SN}$ .

CR2 relies on the closure of reduction under substitution: Assume  $\mathcal{C}[\mathcal{A}]_\Gamma \ni t \rightarrow t'$  and  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_\Delta$ . To show  $t'(\tau.a) \in \mathcal{C}_\Delta$  observe that  $t(\tau.a) \in \mathcal{C}_\Delta$  and that CR2 holds for  $\mathcal{C}$ . ◀

► **Remark 6 (On emptiness of RCs).** In untyped presentations of RCs, CR3 guarantees non-emptiness of any  $\mathcal{A} \in \text{CR}$ , since automatically all variables will inhabit  $\mathcal{A}$  by virtue of CR3. In our case,  $\mathcal{A}_\Gamma$  may be empty since there may be no variables  $x : \Gamma \vdash A$  of the correct type  $A$ . We thus have to be a bit careful when carrying over the textbook proofs [19] to our intrinsically-typed setting.

<sup>8</sup> See Matthes' [25, Section 6.2] systematic exposition of introduction-based vs. elimination-based definition of semantic types (in the context of the saturated sets method).



### 4.1 Elimination-based approach

Geuvers and Hurkens [14] base the semantic definition of the logical connective on the elimination rules. A term inhabits a semantic type if it can be soundly eliminated by all possible eliminations for that type. In case of implication,

$$f \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma \iff \forall \mathcal{C} \in \text{CR}, \tau : \Delta \leq \Gamma, a \in \mathcal{A}_\Delta, t \in \mathcal{C}[\mathcal{B}]_\Delta. f\tau \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}_\Delta.$$

Due to our intrinsic typing, in contrast to Geuvers and Hurkens [14], we need *Kripke-style function space*, i.e., quantify over all extensions  $\Delta$  of  $\Gamma$  with their respective embeddings  $\tau : \Delta \leq \Gamma$ . Still, this definition can be mechanically derived from the elimination rules of implication, which is the single rule:

$$\frac{f : \Gamma \vdash A \rightarrow B \quad a : \Gamma \vdash A \quad t : \Gamma.B \vdash C}{f \cdot \text{el}_{\rightarrow}^{10}(a, t) : \Gamma \vdash C}$$

In case of several elimination rules, the definition of the semantic type has to require the closure under all rules [14].

Note the impredicative quantification over all reducibility candidates  $\mathcal{C}$ , which requires an impredicative meta-theory to formalize this definition. Such an impredicative quantification is not required in the introduction-based approach that we study in Section 4.3.

The elimination-based approach gives us the soundness of the elimination rules for free.

► **Lemma 7** (Elimination). *If  $f \in \mathcal{A} \rightarrow \mathcal{B}$  and  $a \in \mathcal{A}$  and  $t \in \mathcal{C}[\mathcal{B}]$  then  $f \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}$ . (Property  $(\text{el}_{\rightarrow}^{10})$ .)*

**Proof.** By definition of  $\mathcal{A} \rightarrow \mathcal{B}$  using  $\tau = \mathbb{1}$ . ◀

Soundness of the introduction rules requires some work.

► **Lemma 8** (Introduction). *Properties  $(\text{in}_{\rightarrow}^{00})$ ,  $(\text{in}_{\rightarrow}^{01})$  and  $(\text{in}_{\rightarrow}^{11})$  hold for  $\mathcal{A} \rightarrow \mathcal{B}$ .*

**Proof.** We show property  $(\text{in}_{\rightarrow}^{01})$ , the others are analogous. Assume  $t \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]$  and  $b \in \mathcal{B}$  and show  $\text{in}_{\rightarrow}^{01}(t, b) \in \mathcal{A} \rightarrow \mathcal{B}$ . To this end, assume  $\mathcal{C} \in \text{CR}$  and  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_\Delta$  and  $u \in \mathcal{C}[\mathcal{B}]_\Delta$  and show  $v := \text{in}_{\rightarrow}^{01}(t, b)\tau \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \mathcal{C}_\Delta$  by induction on  $t(\uparrow\tau)$ ,  $b\tau$ ,  $a$ ,  $u \in \text{SN}$  (obtained by CR1).

Since  $v$  is not an introduction we shall utilize CR3 for  $\mathcal{C}$ . Therefore, we have to show that all reducts of  $v$  are already in  $\mathcal{C}_\Delta$ .

If reduction happens in subterm  $b\tau$ , so  $b\tau \rightarrow b'$ , we can apply the induction hypothesis on  $b' \in \text{SN}$ , since  $b' \in \mathcal{B}_\Delta$  by CR2. Reduction in one of the other subterms  $t, a, u$  of  $v$  is treated analogously.

It remains to cover the  $\beta$ -reductions at the root, which are  $v \rightarrow u[b\tau]$  and  $v \rightarrow t(\tau.a) \cdot \text{el}_{\rightarrow}^{10}(a, u)$ . We have  $u[b\tau] \in \mathcal{C}_\Delta$  by assumptions on  $u$  and  $b$ . Further, since  $t(\tau.a) \in (\mathcal{A} \rightarrow \mathcal{B})_\Delta$ , by definition  $t(\tau.a) \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \mathcal{C}_\Delta$ . ◀

Let us not forget to verify that  $\mathcal{A} \rightarrow \mathcal{B}$  is indeed a reducibility candidate.

► **Lemma 9** (Function space). *If  $\mathcal{A}, \mathcal{B} \in \text{CR}$  then  $(\mathcal{A} \rightarrow \mathcal{B}) \in \text{CR}$ .*

**Proof.** First,  $\mathcal{A} \rightarrow \mathcal{B}$  needs to be a term set family. This is facilitated by the Kripke-style definition of the function space: Assume  $f \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$  and  $\tau : \Delta \leq \Gamma$  and show  $f\tau \in (\mathcal{A} \rightarrow \mathcal{B})_\Delta$ . To this end assume  $\mathcal{C} \in \text{CR}$  and  $\tau' \in \Phi \leq \Delta$  and  $a \in \mathcal{A}_\Phi$  and  $t \in \mathcal{C}[\mathcal{B}]_\Phi$  and show  $f\tau\tau' \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}_\Phi$ . This follows from the assumption on  $f$  with OPE  $\tau\tau' : \Phi \leq \Gamma$ .



For CR1, assume  $f \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$  and show  $f \in \text{SN}$ . Let  $\mathcal{C} = \mathcal{A}$  (this choice is simplest, but any RC would do) and  $\Delta = \Gamma.A$ . Clearly  $a := (x_0 : \Delta \vdash A) \in \mathcal{A}_\Delta$  and  $t := (x_1 : \Delta.B \vdash A) \in \mathcal{C}[\mathcal{B}]_\Delta$ . Thus  $f\tau \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}_\Delta \subseteq \text{SN}$ . This implies  $f \in \text{SN}$ .

Closure under reduction (CR2) follows because reduction is closed under weakening and elimination.

For CR3, assume  $f : \Gamma \vdash A \rightarrow B$  that is not an introduction and whose reducts are in  $(\mathcal{A} \rightarrow \mathcal{B})_\Gamma$ . To show  $f \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$ , assume  $\mathcal{C} \in \text{CR}$  and  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_\Delta$  and  $t \in \mathcal{C}[\mathcal{B}]_\Delta$  and show  $f\tau \cdot E \in \mathcal{C}_\Delta$  where  $E = \text{el}_{\rightarrow}^{10}(a, t)$ . We proceed by CR3 for  $\mathcal{C}$ , exploiting that  $f\tau \cdot E$  is not an introduction either. It is sufficient to show that all reducts of  $f\tau \cdot E$  are in  $\mathcal{C}_\Delta$ . We proceed by induction on  $a, t \in \text{SN}$ . Since  $f$  is not an introduction, we can only reduce in  $f$  or in  $E$ . Reductions in  $f$  are covered by the assumption on  $f$ . Reductions in  $E$  are either  $a \rightarrow a'$  or  $t \rightarrow t'$  and covered by the respective induction hypothesis, since  $a'$  and  $t'$  stay in their respective RCs by virtue of CR2. ◀

Strong normalization now follows according to Section 3.

## 4.2 A gap in the original proof by Geuvers and Hurkens, and its fix

In their elimination-based SN proof, Geuvers and Hurkens [14, Section 6.1] use for semantic types saturated sets with the expansion closure modified to liken CR3. To explain their approach, let us first introduce weak head reduction<sup>9</sup>  $I \cdot E \cdot \vec{E} \triangleright_\beta v \cdot \vec{E}$  where  $\beta$ -redex  $I \cdot E$  contracts to  $v$  and the elimination sequence  $\vec{E}$  is arbitrary (can be empty). Any SN term that is neither an introduction nor a  $\triangleright_\beta$ -redex is called neutral (set  $\text{Neut}$ ).

In Def. 57.3 [14] a set of terms  $\mathcal{X}$  is defined to be saturated ( $\mathcal{X} \in \text{SAT}$ ) if

- a. (SAT1)  $\mathcal{X} \subseteq \text{SN}$ ,
- b. (SAT2)  $\text{Neut} \subseteq \mathcal{X}$ , and
- c. (SAT3')  $\mathcal{X}$  is closed under  $\triangleright_\beta$ -expansion, namely if  $t \in \text{SN}$  and  $(t \triangleright_\beta \_) \subseteq \mathcal{X}$  (\*) then  $t \in \mathcal{X}$ .

In the original formulation (SAT3) of the saturated sets method,<sup>10</sup> the requirement (\*) is that  $(t \triangleright_\beta \_) \cap \mathcal{X}$  is inhabited, meaning that  $t$  is the weak-head expansion of some term that is already in  $\mathcal{X}$ . In the new formulation the requirement is that all weak-head reducts of  $t$  are in  $\mathcal{X}$ . It is easy to see that now SAT2 is subsumed under SAT3', since neutrals have no weak-head reducts, and the condition (\*) is trivially satisfied. The modification of SAT3 towards CR3-style SAT3' was perhaps undertaken to account for the non-determinism of  $\triangleright_\beta$  in ITTND.

Unfortunately, with SAT3' it is not clear how to show the equivalent of Lemma 9,  $(\mathcal{A} \rightarrow \mathcal{B}) \in \text{SAT}$  [14, Lemma 58]. In the formulation based on untyped terms,  $\mathcal{A} \rightarrow \mathcal{B}$  is defined by

$$f \in (\mathcal{A} \rightarrow \mathcal{B}) \iff \forall \mathcal{C} \in \text{SAT}, a \in \mathcal{A}, t \in \mathcal{C}[\mathcal{B}]. f \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}.$$

To attempt to show SAT3' for  $\mathcal{A} \rightarrow \mathcal{B}$ , assume  $f \in \text{SN}$  with  $(f \triangleright_\beta \_) \subseteq \mathcal{A} \rightarrow \mathcal{B}$  and derive  $f \in \mathcal{A} \rightarrow \mathcal{B}$ . To this end, assume  $\mathcal{C} \in \text{SAT}$  and  $a \in \mathcal{A}$  and  $t \in \mathcal{C}[\mathcal{B}]$  and show  $f \cdot E \in \mathcal{C}$  with  $E = \text{el}_{\rightarrow}^{10}(a, t)$ . Since  $\mathcal{C}$  is arbitrary, we have to rely on SAT3' to introduce elements into  $\mathcal{C}$ . Thus, we need to show (1)  $f \cdot E \in \text{SN}$  and (2)  $t' \in \mathcal{C}$  whenever  $f \cdot E \triangleright_\beta t'$ . For both goals we need to analyze the reducts of  $f \cdot E$ . The problem is that  $f$  could be an introduction and,

<sup>9</sup> Weak head reduction is called *key reduction* in *loc. cit.*.

<sup>10</sup> See for instance the exposition by Luo [23].

hence,  $f \cdot E$  a  $\beta$ -redex reducing to some  $v$ . We lack assumptions to show  $v \in \mathcal{C}$  and even  $v \in \text{SN}$ , since  $v$  is not of the same form as  $f \cdot E$ . Were it either  $f' \cdot E$  (with  $f \longrightarrow f'$ ) or  $f \cdot E'$  (with  $E \longrightarrow E'$ ) there would be some hope to use the assumptions, in particular  $f, E \in \text{SN}$ .

Note that with the original SAT3 the relevant part of the proof goes in the other direction, we can exploit the closure of weak head reduction under elimination, namely if  $f \triangleright_\beta f'$  then  $f \cdot E \triangleright_\beta f' \cdot E$ . It seems that this direction is employed in the proof sketch [14, Lemma 58.c], not matching the new requirement SAT3'.

Pointed to the gaps in their argument Geuvers and Hurkens published a revision [15] with two amendments to the definitions:

1. Closure condition SAT3' now applies only to *weak head redexes*  $t$ . Only strongly normalizing weak head redexes  $t$  whose weak head reducts are in saturated set  $\mathcal{X}$  are forced into  $\mathcal{X}$ . The thus relativized SAT3' no longer subsumes SAT2 which forces *neutrals* into  $\mathcal{X}$ .
2. The semantic connectives are relativized to SN terms. E.g.,  $f \in (\mathcal{A} \rightarrow \mathcal{B})$  stipulates also  $f \in \text{SN}$ .

The second amendment fixes a problem with connectives that have no eliminations, like *truth*, but does not add anything for connectives with at least one elimination, like  $\mathcal{A} \rightarrow \mathcal{B}$ .

Yet the first amendment allows us now to analyse the reducts of  $f \cdot E$  in the proof of SAT3' for  $\mathcal{A} \rightarrow \mathcal{B}$ . Since  $f$  is not an introduction, the only weak head redexes of  $f \cdot E$  are of the form  $f' \cdot E$  with  $f \triangleright_\beta f'$ . To show  $(f \cdot E \triangleright_\beta \_) \subseteq \mathcal{C}$ , we can thus utilize the assumption  $(f \triangleright_\beta \_) \subseteq \mathcal{A} \rightarrow \mathcal{B}$ . This repairs the proof; in Appendix A.1 we will see a variant of the amended proof be spelled out in detail.

In the following section, we can get rid of the impredicative definition of  $\mathcal{A} \rightarrow \mathcal{B}$  and use an inductive definition instead. We study this introduction-based approach to type interpretation in the context of Girard's method, but conjecture that it could be utilized in the arguably more structured method of Geuvers and Hurkens as well.

### 4.3 Introduction-based approach

Instead of the impredicative *elimination*-based definition of semantic types like  $\mathcal{A} \rightarrow \mathcal{B}$ , we can base their definition on the *introduction* rules. The rough idea is that elements of  $\mathcal{A} \rightarrow \mathcal{B}$  can be introduced by any of  $\text{in}_\rightarrow^{00}$ ,  $\text{in}_\rightarrow^{01}$ , and  $\text{in}_\rightarrow^{11}$  – this is a union of reducibility candidates. However, since the first two of these need already the implication they introduce, the construction of a least fixed-point is required.

Note that the union  $\mathcal{A} \cup \mathcal{B}$  of two reducibility candidates  $\mathcal{A}$  and  $\mathcal{B}$  preserves CR1/2, but not CR3. However, property CR3 can be forced by the following closure operation  $\overline{\mathcal{A}}$  on a term set  $\mathcal{A} \subseteq (\Gamma \vdash A)$ .

$$\text{EMB} \frac{a \in \mathcal{A}}{a \in \overline{\mathcal{A}}} \quad \text{EXP} \frac{a : \Gamma \vdash A \quad a \notin \text{Intro} \quad (a \longrightarrow \_) \subseteq \overline{\mathcal{A}}}{a \in \overline{\mathcal{A}}}$$

The closure operation lifts pointwise to families  $\mathcal{A}_\Gamma \subseteq \Gamma \vdash A$  of term sets.

► **Lemma 10.** *If  $a \in \overline{\mathcal{A}}_\Gamma$  and  $\tau : \Delta \leq \Gamma$  then  $a\tau \in \overline{\mathcal{A}}_\Delta$ .*

**Proof.** By induction on  $a \in \overline{\mathcal{A}}_\Gamma$ . In case  $a \in \mathcal{A}_\Gamma$  (EMB) use the functoriality of  $\mathcal{A}$  and EMB. In case EXP, i.e.,  $a \in \text{SN}(\Gamma \vdash A) \setminus \text{Intro}$  and  $(a \longrightarrow \_) \subseteq \overline{\mathcal{A}}_\Delta$  we first have  $a\tau \in \text{SN}(\Delta \vdash A) \setminus \text{Intro}$ . If  $a\tau \longrightarrow b'$  then there is  $b$  with  $a \longrightarrow b$  and  $b' = b\tau$ , and by induction hypothesis  $b\tau \in \overline{\mathcal{A}}_\Delta$ . Thus  $a\tau \in \overline{\mathcal{A}}_\Delta$  by EXP. ◀

► **Lemma 11 (Saturation).**  *$\overline{\mathcal{A}}$  is a reducibility candidate for any precandidate  $\mathcal{A}$ .*

**Proof.** CR3 is forced by the closure operation. Closure under reduction (CR2) and preservation of SN (CR1) are proven by induction on  $a \in \overline{\mathcal{A}}$ , the latter using that  $a \in \text{SN}$  when all of  $a$ 's reducts are. ◀

We may now define a notion of function space on reducibility candidates based on the introduction rules for implication. Since introduction rules are “recursive” in general, i.e., may mention the principal formula in the subsequent of a premise, we need to employ the least fixed-point operation  $\mu$  for monotone operators on the lattice of reducibility candidates. We define  $\mathcal{A} \rightarrow \mathcal{B} = \mu\mathcal{F}$  where

$$\mathcal{F}(\mathcal{X})_\Gamma = \overline{\{\text{in}_{\rightarrow}^{00}(t, u), \text{in}_{\rightarrow}^{01}(t, b), \text{in}_{\rightarrow}^{11}(a, b) \mid a \in \mathcal{A}_\Gamma, b \in \mathcal{B}_\Gamma, t \in \mathcal{X}[\mathcal{A}]_\Gamma, u \in \mathcal{X}[\mathcal{B}]_\Gamma\}}$$

This operation acts on reducibility candidates:

► **Lemma 12** (Function space). *If  $\mathcal{A}$  and  $\mathcal{B}$  are reducibility candidates, so is  $\mathcal{A} \rightarrow \mathcal{B}$ .*

**Proof.** It is sufficient to show that  $\mathcal{F}$  acts on reducibility candidates. Since CR3 is forced, it is sufficient that  $\mathcal{F}(\mathcal{X})$  is a precandidate for any candidate  $\mathcal{X}$ , and this follows mostly from Lemma 5 and the candidacy of  $\mathcal{A}$  and  $\mathcal{B}$ . CR1 follows since any reduction of an introduction needs to happen in one of the arguments of in, which are SN. CR2 follows by the same observation. ◀

By definition,  $\mathcal{A} \rightarrow \mathcal{B}$  models the introduction rules for implication: properties  $(\text{in}_{\rightarrow}^{00})$ ,  $(\text{in}_{\rightarrow}^{01})$  and  $(\text{in}_{\rightarrow}^{11})$ . For the elimination rule, property  $(\text{el}_{\rightarrow}^{10})$ , we have to do a bit of work.

► **Lemma 13** (Function elimination). *Let  $\mathcal{A}, \mathcal{B}, \mathcal{C}$  be candidates. If  $f \in \mathcal{A} \rightarrow \mathcal{B}$  and  $a \in \mathcal{A}$  and  $u \in \mathcal{C}[\mathcal{B}]$  then  $f \cdot E \in \mathcal{C}$  where  $E = \text{el}_{\rightarrow}^{10}(a, u)$ .*

**Proof.** By main induction on  $f \in \mathcal{A} \rightarrow \mathcal{B}$ .

**Case (exp)  $f \notin \text{Intro}$  and  $f \rightarrow f'$  implies  $f' \in \mathcal{A} \rightarrow \mathcal{B}$ .** We show  $f \cdot E \in \mathcal{C}$  by side induction on  $E \in \text{SN}$  via CR3. First,  $f \cdot E \notin \text{Intro}$ . Assume  $f \cdot E \rightarrow c$ . Since  $f$  is not an introduction, we have either  $f \rightarrow f'$  or  $E \rightarrow E'$ . In the first case, by main induction hypothesis,  $f' \cdot E \in \mathcal{C}$ . In the second case,  $f \cdot E' \in \mathcal{C}$  by side induction hypothesis. In any case,  $c \in \mathcal{C}$ . Since  $c$  was arbitrary,  $f \cdot E \in \mathcal{C}$  by CR3.

**Case  $f = \text{in}_{\rightarrow}^{00}(t_1, t_2)$  where  $t_1 \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]$  and  $t_2 \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{B}]$ .** We show  $f \cdot E \in \mathcal{C}$  by side induction on  $t_1, t_2, E \in \text{SN}$  via CR3. Given  $f \cdot E \rightarrow c$ , there are three cases. Either  $c = f' \cdot E$  with  $f \rightarrow f'$  or  $c = f \cdot E'$  with  $E \rightarrow E'$  or  $c = t_1[a] \cdot E$ . The first two cases are handled by the side induction hypotheses, the last case by main induction hypothesis on  $t_1[a] \in \mathcal{A} \rightarrow \mathcal{B}$ .

**Case  $f = \text{in}_{\rightarrow}^{01}(t_1, b)$  where  $t_1 \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]$  and  $b \in \mathcal{B}$ .** We show  $f \cdot E \in \mathcal{C}$  by side induction on  $t, b, E \in \text{SN}$  via CR3. Given  $f \cdot E \rightarrow c$ , there are four cases. Either  $c = f' \cdot E$  with  $f \rightarrow f'$  or  $c = f \cdot E'$  with  $E \rightarrow E'$  or  $c = t[a] \cdot E$  or  $c = u[b]$ . The first two cases are handled by the side induction hypotheses, the but-last case by main induction hypothesis on  $t[a] \in \mathcal{A} \rightarrow \mathcal{B}$ , and the last case by assumption  $u \in \mathcal{C}[\mathcal{B}]$ .

**Case  $f = \text{in}_{\rightarrow}^{11}(a', b)$  where  $a' \in \mathcal{A}$  and  $b \in \mathcal{B}$ .** We show  $f \cdot E \in \mathcal{C}$  by side induction on  $a', b, E \in \text{SN}$  via CR3.

Given  $f \cdot E \rightarrow c$ , there are three cases. Either  $c = f' \cdot E$  with  $f \rightarrow f'$  or  $c = f \cdot E'$  with  $E \rightarrow E'$  or  $c = u[b]$ . The first two cases are handled by the side induction hypotheses and the last case by assumption  $u \in \mathcal{C}[\mathcal{B}]$ . ◀

The pattern outlined here for implication generalizes to arbitrary connectives given by truth tables. Each connective is interpreted as an operation on candidates, using the least fixed-point of the closure of the term set generated by the introductions. Each elimination then has to be proven sound in a lemma similar to Lemma 13.

This concludes our study of reducibility candidates to show SN for ITTND. In the remaining technical sections, we study the extension of the normalization argument to permutations.

## 5 Permutation Reductions

In previous sections, we have studied the reduction  $\beta$  of detours  $I \cdot E$  stemming from an elimination  $E$  of a via  $I$  just introduced connective. In ITTND, even an elimination  $E$  followed by another elimination  $E'$ , thus, a term of the form  $f \cdot E \cdot E'$ , constitutes a detour and can be  $\pi$ -reduced.

For the sake of defining and studying  $\pi$ -reduction, let us introduce eliminations  $E$  and evaluation contexts  $\vec{E}$ , aka *spines*, as syntactic classes separate from terms. Eliminations  $E$  from type  $A$  into type  $C$  are typed by judgement  $\boxed{E : \Gamma \mid A \vdash C}$ . In the case of implication, we have:

$$\frac{a : \Gamma \vdash A \quad u : \Gamma.B \vdash C}{\text{el}_{\rightarrow}^{10}(a, u) : \Gamma \mid A \rightarrow B \vdash C}$$

Sequences of eliminations form spines  $\vec{E}$ , where we denote the empty spine as  $\text{id}$  and spine construction by a centered dot.

$$\frac{}{\text{id} : \Gamma \mid A \vdash A} \quad \frac{E : \Gamma \mid A \vdash B \quad \vec{E} : \Gamma \mid B \vdash C}{E \cdot \vec{E} : \Gamma \mid A \vdash C}$$

Spine construction straightforwardly extends to spine concatenation  $\vec{E} \cdot \vec{E}'$ . Weakening  $\vec{E}\tau$  and substitution  $\vec{E}\sigma$  are defined in the obvious way.

Since the target type  $C$  of an elimination can be freely chosen, one can structure a proof to always eliminate a hypothesis  $x : A$  directly into the goal  $C$ . Thus, a sequence  $x \cdot E \cdot E'$  of two eliminations  $E : \Gamma \mid A \vdash B$  and  $E' : \Gamma \mid B \vdash C$ , going via an intermediate formula  $B$ , can be considered a detour.

This detour is removed by a permutation contraction  $\boxed{E \cdot E' \mapsto_{\pi} E\{E'\}}$  that shifts (“permutes”) the outer elimination  $E'$  into the negative branches of the inner elimination  $E$ . The composition<sup>11</sup>  $\boxed{E\{E'\}}$  of eliminations moves a weakened version of  $E'$  to the negative branches of  $E$ . In the case of implication, we have

$$\text{el}_{\rightarrow}^{10}(a, u)\{E'\} = \text{el}_{\rightarrow}^{10}(a, u \cdot E'\uparrow) \tag{1}$$

where  $E'\uparrow$  shall denote the weakening of elimination  $E'$  by  $\uparrow : \Gamma.B \leq \Gamma$ . In particular,  $\text{el}_{\rightarrow}^{10}(a', u')\uparrow = \text{el}_{\rightarrow}^{10}(a'\uparrow, u'(\uparrow\uparrow))$ .

► **Remark 14.** If in Equation (1) term  $u$  is an introduction, it may  $\beta$ -react with  $E'$  to eliminate further detours. Thus,  $\pi$ -reductions can lead to significant further normalization.

<sup>11</sup>The notation  $E\{E'\}$  is due to Joachimski and Matthes [20].

Now a one-step  $\pi$ -reduction  $\boxed{t \longrightarrow_{\pi} t'}$  shall be a  $\pi$ -contraction in some spine within term  $t$ . Let us further define *spine reduction*  $\boxed{\vec{E} \triangleright_{\pi} \vec{E}'}$  as  $\pi$ -contraction within a spine at the root, i.e., inductively by the axiom

$$\vec{E} \cdot E_1 \cdot E_2 \cdot \vec{E}' \triangleright_{\pi} \vec{E} \cdot E_1 \{E_2\} \cdot \vec{E}'.$$

Since a spine reduction shortens the length of the spine by 1, spine reduction is SN. For  $\pi$ -reduction, the situation is slightly more complicated since a  $\pi$ -reduction can create new  $\pi$ -redexes: for instance, if in Equation (1) the term  $u$  is an elimination. However, these  $\pi$ -redexes have moved deeper into the term, thus, by ranking  $\pi$ -redexes by their depth we can easily construct a termination order. Consequently,  $\pi$ -reduction alone is also SN [14, Thm. 55]. Since elimination composition is associative, i.e.,  $(E_1 \{E_2\}) \{E_3\} = E_1 \{E_2 \{E_3\}\}$ , spine and  $\pi$ -reduction are confluent.

## 5.1 Permutations are harmless

For  $\beta$ -reduction alone, we have the following closure property of SN: If all proper sub-terms and all  $\triangleright_{\beta}$ -reducts of a term are  $\beta$ -SN, so is the term itself. This is Lemma 2.3. of Geuvers and Hurkens' addendum [15]. We reprove it here for  $\beta\pi$ -SN. Note that the requirements are not extended to include the  $\triangleright_{\pi}$ -reducts! So, the addition of permutation reduction is actually "harmless".

From now, let "reduction" be  $\beta\pi$ -reduction and SN be understood w.r.t. this reduction relation.

► **Lemma 15** (Weak head expansion). *Assume  $a, b, t, u, a', u' \in \text{SN}$ , where mentioned. Let  $E = \text{el}_{\rightarrow}^{10}(a', u')$ .*

1. *If  $t[a'] \cdot E \cdot \vec{E} \in \text{SN}$  then  $\text{in}_{\rightarrow}^{00}(t, u) \cdot E \cdot \vec{E} \in \text{SN}$ .*
2. *If  $t[a'] \cdot E \cdot \vec{E} \in \text{SN}$  and  $u'[b] \cdot \vec{E} \in \text{SN}$  then  $\text{in}_{\rightarrow}^{01}(t, b) \cdot E \cdot \vec{E} \in \text{SN}$ .*
3. *If  $u'[b] \cdot \vec{E} \in \text{SN}$  then  $\text{in}_{\rightarrow}^{11}(a, b) \cdot E \cdot \vec{E} \in \text{SN}$ .*

**Proof.** We demonstrate statement 2 in detail, the others are similar. For  $\text{in}_{\rightarrow}^{01}(t, b) \cdot \text{el}_{\rightarrow}^{10}(a', u') \cdot \vec{E} \in \text{SN}$ , we show that all its one-step reducts are SN. To this end, we induct on our two main hypotheses (i) and (ii). The induction on (i)  $t[a'] \cdot E \cdot \vec{E} \in \text{SN}$  immediately covers reductions in  $t$ ,  $E$ , and  $\vec{E}$ , and the induction on (ii)  $u'[b] \cdot \vec{E} \in \text{SN}$  covers the remaining inner reductions, namely in  $b$ .

Besides inner reductions, we have two  $\triangleright_{\beta}$ -reductions, yet they are directly implied by our two main hypotheses. It remains to show that the  $\pi$ -contraction of  $E \cdot \vec{E}$  is also benign, meaning  $I \cdot E' \cdot \vec{E}' \in \text{SN}$ , where  $I = \text{in}_{\rightarrow}^{01}(t, b)$  and  $E' = \text{el}_{\rightarrow}^{10}(a', u' \cdot E_1 \uparrow)$  and  $\vec{E}' = E_1 \cdot \vec{E}'$ . To tackle this by induction hypothesis, we need to show the two new main hypotheses, which are now (i')  $t[a'] \cdot E' \cdot \vec{E}' \in \text{SN}$  and (ii')  $(u' \cdot E_1 \uparrow)[b] \cdot \vec{E}' \in \text{SN}$ . But (i') is just a  $\pi$ -reduct of (i), and (ii') is identical to (ii), once we distribute the substitution  $[b]$ . The inductive step is thus justified by the first induction hypothesis.

Statement 1 is very similar, only that the second induction is on  $(u, u') \in \text{SN}$ , to cover reductions in  $u$  and  $u'$ .

Statement 3 needs a main induction on the length of  $\vec{E}$  to cover the case of  $\triangleright_{\pi}$ -reduction. Further side inductions are needed on  $a, a' \in \text{SN}$ . ◀

Similar arguments to Lemma 15 can be found in the work of Joachimski and Matthes [21, Sect. 5 and 6]. I have also formalized that argument in Agda, albeit for a simpler case: simply-typed combinatory algebra with conditionals.<sup>12</sup>

<sup>12</sup><https://github.com/andreasabel/truthtable/blob/1a7a01fd28ffb327e9c91a3722e49b467d05a79d/agda/SK-Bool-ortho.agda>

## 5.2 Failure of the CR method for $\beta\pi$

Our goal is now a model-theoretic proof of the SN of  $\beta\pi$ -reduction. Unfortunately, just throwing permutation reductions into the mix and replaying the CR proof for SN- $\beta$  does not work, despite the “harmless” character of permutations. The proof of Lemma 13 relies on the fact that if  $f \cdot E \rightarrow c$  and  $f \notin \text{Intro}$  then either  $f \rightarrow f'$  or  $E \rightarrow E'$ , and the structure of the elimination  $f \cdot E$  is preserved. However, with permutations, in case  $f = f_0 \cdot E_0$  it could be that  $c = f_0 \cdot E_0\{E\}$ , changing the structure of the elimination. Such reductions are not covered by any of the induction hypotheses.

We cannot arbitrarily tighten the restriction  $\_ \notin \text{Intro}$  in the formulation of CR3, since CR3 is used in Lemma 13 to introduce terms of the shape  $f \cdot E$  into a reducibility candidate  $\mathcal{C}$ . Such terms need to satisfy the restriction, therefore we cannot exclude  $\pi$ -redexes in general: a priori,  $f \cdot E$  could be a  $\pi$ -redex.

## 6 Orthogonality

Since the reducibility candidate method does not immediately extend to permutations, we turn to a more powerful technique: (bi)orthogonals [6, 29, 8, 18, 32, 1]. Lindley and Stark [22] have observed that biorthogonals (“ $\top\top$ -lifting”) deal well with the permutation reduction for the monadic bind in a strong normalization proof for the monadic meta-language. We shall thus adapt this technique, although it is more demanding on our meta-theory, requiring greatest fixed-points of non-strictly positive operators. This is covered by Knaster and Tarski’s fixed-point theorem [31], but not readily available in type-theoretic proof assistants like Coq [7] and Agda [2].

In the following, when we speak of context-indexed families, we implicitly assume that the family is closed under weakening.

Semantic types  $\mathcal{A}$  shall now be context-indexed families of sets of spines  $\vec{E}$ , and we write  $a \perp \mathcal{A}_\Gamma$  to characterize a term  $a : \Gamma \vdash A$  as classified by semantic type  $\mathcal{A}$ . The orthogonality relation  $\perp$  is defined as

$$a \perp \mathcal{A}_\Gamma : \iff a \in \mathcal{A}_\Gamma^\perp : \iff a \cdot \vec{E} \in \text{SN for all } \vec{E} \in \mathcal{A}_\Gamma.$$

We demand of semantic types that they contain the empty spine  $\text{id}$  and only contain strongly normalizing spines. Reductions  $\vec{E} \rightarrow \vec{E}'$  in spines  $\vec{E}$  can either be  $\beta\pi$ -reductions in the subterms of the eliminations or can be  $\pi$ -contractions along the spine.

More formally, a semantic type  $\mathcal{A}_\Gamma$  for syntactic type  $A$  at context  $\Gamma$  is a set of *pairs*  $(C, (\vec{E} : \Gamma \mid A \vdash C))$ . Then  $a \perp \mathcal{A}_\Gamma$  is defined as  $a \cdot \vec{E} \in \text{SN}(\Gamma \vdash C)$  for all  $(C, \vec{E} : \Gamma \mid A \vdash C) \in \mathcal{A}_\Gamma$ . However, we typically suppress the type component  $C$  which is implicitly determined by  $\vec{E}$ .

► **Lemma 16** (Semantic types). *Let  $\mathcal{A}$  be a semantic type for  $A$ .*

1. *If  $x : \Gamma \vdash A$  is a variable, then  $x \perp \mathcal{A}_\Gamma$ .*
2.  $\mathcal{A}^\perp \subseteq \text{SN}$ .
3.  $\mathcal{A}^\perp$  is closed under reduction.

**Proof.**

1. Given  $(C, \vec{E}) \in \mathcal{A}_\Gamma$  show  $x \cdot \vec{E} \in \text{SN}$ . This holds since the only reductions are in  $\vec{E}$ , which is required to be SN by definition of semantic types.
2. Given  $t \perp \mathcal{A}_\Gamma$  show  $t \in \text{SN}$ . Since  $\text{id} \in \mathcal{A}_\Gamma$ , we have  $t \cdot \text{id} = t \in \text{SN}$ .
3. Given  $t \perp \mathcal{A}_\Gamma$  and  $t \rightarrow t'$  and  $\vec{E} \in \mathcal{A}_\Gamma$  we have  $t' \cdot \vec{E} \in \text{SN}$  since  $t \cdot \vec{E} \in \text{SN}$  and  $t \cdot \vec{E} \rightarrow t' \cdot \vec{E}$ . ◀

Symmetrically to  $\mathcal{A}^\perp$ , given a set of terms  $\mathcal{T}_\Gamma \subseteq (\Gamma \vdash A)$  we define

$$\mathcal{T}_\Gamma^\perp = \{(C, (\vec{E} : \Gamma \mid A \vdash C)) \mid a \cdot \vec{E} \in \text{SN}(\Gamma \vdash C) \text{ for all } a \in \mathcal{T}_\Gamma\}.$$

Taking the orthogonal  $\mathcal{T}^\perp$  of a non-empty SN term set  $\mathcal{T}$  is one way to construct a semantic type:

► **Lemma 17** (Orthogonals are semantic types). *If  $\mathcal{T}$  is a family of non-empty sets of strongly normalizing terms of type  $A$ , then  $\mathcal{T}^\perp$  is a semantic type for type  $A$ .*

**Proof.** First,  $\text{id} \in \mathcal{T}^\perp$  since  $\mathcal{T} \subseteq \text{SN}$ . Then  $\mathcal{T}^\perp \subseteq \text{SN}$  since  $\mathcal{T}$  is non-empty. ◀

By definition, orthogonality gives rise to the Galois connection

$$\mathcal{T}^\perp \supseteq \mathcal{A} \iff \mathcal{T} \subseteq \mathcal{A}^\perp$$

(both sides of  $\iff$  expand to the same statement  $\forall t \in \mathcal{T}, \vec{E} \in \mathcal{A}. t \cdot \vec{E} \in \text{SN}$ ). As a consequence, biorthogonality  $\_ \perp^\perp$  is a closure operator both on sets of terms,  $\mathcal{T} \subseteq \mathcal{T}^{\perp\perp}$ , and evaluation contexts,  $\mathcal{A} \subseteq \mathcal{A}^{\perp\perp}$ .

The abstraction type  $\mathcal{X}[\mathcal{A}]$  is now defined by

$$\mathcal{X}[\mathcal{A}]_\Gamma = \{(C, (\vec{E} : \Gamma.A \mid X \vdash C)) \mid \vec{E}(\tau.a) \in \mathcal{X}_\Delta \text{ for all } \tau : \Delta \leq \Gamma \text{ and } a \perp \mathcal{A}_\Delta\}.$$

Abstraction operates on semantic types:

► **Lemma 18** (Abstraction, revisited). *If  $\mathcal{A}$  and  $\mathcal{X}$  are semantic types for  $A$  and  $X$ , then  $\mathcal{X}[\mathcal{A}]$  is a semantic type for  $X$ .*

**Proof.** We first show that  $(X, (\text{id} : \Gamma.A \mid X \vdash X)) \in \mathcal{X}[\mathcal{A}]_\Gamma$ . To this end, assume  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_\Delta$  and show  $\text{id}(\tau.a) \in \mathcal{X}_\Delta$ . This is trivial, since  $\text{id}(\tau.a) = \text{id}$  and  $\mathcal{X}$  is a semantic type.

Then, assume  $(C, (\vec{E} : \Gamma.A \mid X \vdash C)) \in \mathcal{X}[\mathcal{A}]_\Gamma$  and show  $\vec{E} \in \text{SN}$ . Choose  $\tau = \uparrow : \Gamma.A \leq \Gamma$  and  $a = x_0 \in \mathcal{A}_{\Gamma.A}$  the 0th de Bruijn index, then  $\vec{E}(\uparrow, x_0) = \vec{E} \in \mathcal{X}_{\Gamma.A}$  and hence SN. ◀

Given two semantic types  $\mathcal{A}$  and  $\mathcal{B}$ , the function space  $\mathcal{A} \rightarrow \mathcal{B}$  is defined as the *greatest fixpoint*  $\nu \mathcal{F}^\perp$  of the pointwise orthogonal  $\mathcal{F}^\perp$  of the operator

$$\mathcal{F}(\mathcal{X})_\Gamma = \{\text{in}_{\rightarrow}^{00}(t, u), \text{in}_{\rightarrow}^{01}(t, b), \text{in}_{\rightarrow}^{11}(a, b) \mid a \perp \mathcal{A}_\Gamma, b \perp \mathcal{B}_\Gamma, t \perp \mathcal{X}[\mathcal{A}]_\Gamma, u \perp \mathcal{X}[\mathcal{B}]_\Gamma\}.$$

In comparison with the reducibility candidate version in Section 4, the closure operation has been replaced by biorthogonalization, and we converted  $\mu(\mathcal{F}^{\perp\perp})$  to  $(\nu(\mathcal{F}^\perp))^\perp$ . We dropped the outer orthogonalization since we now compute sets of evaluation contexts, but note that  $\mathcal{F}$  applies orthogonalization on  $\mathcal{X}$ . Due to the double “negation”,  $\mathcal{F}^\perp$  is a non-strictly positive operator which has a (greatest) fixpoint thanks to its monotonicity, yet, this fixpoint is not directly obtainable in meta-theories that only accept *strictly* positive coinductive definitions, such as the type theories of Agda [2] and Coq [7].

► **Lemma 19** (Function space, revisited). *If  $\mathcal{A}$  is a semantic type for  $A$  and  $\mathcal{B}$  one for  $B$ , then  $\mathcal{A} \rightarrow \mathcal{B}$  is a semantic type for  $A \rightarrow B$ .*

**Proof.** Applying Lemma 17, it is sufficient to show that  $\mathcal{F}(\mathcal{X})$  is a family of non-empty sets of SN terms for semantic types  $\mathcal{X}$ . This is the case by assumptions on  $\mathcal{A}$ ,  $\mathcal{B}$ , and  $\mathcal{X}$ . ◀

► **Lemma 20** (Function introduction). *Given  $a \perp \mathcal{A}_\Gamma$  and  $b \perp \mathcal{B}_\Gamma$  and  $t \perp (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]_\Gamma$  and  $u \perp (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{B}]_\Gamma$ , we have  $\text{in}_{\rightarrow}^{00}(t, u), \text{in}_{\rightarrow}^{01}(t, b), \text{in}_{\rightarrow}^{11}(a, b) \perp (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$ .*



**Proof.** For any of the mentioned introductions  $I$  we have  $I \in \mathcal{F}(\mathcal{A} \rightarrow \mathcal{B})_\Gamma$  by definition of  $\mathcal{F}$ . Since biorthogonalization is a closure operator, we have  $I \in \mathcal{F}(\mathcal{A} \rightarrow \mathcal{B})_\Gamma^\perp$  and thus  $I \perp \mathcal{F}(\mathcal{A} \rightarrow \mathcal{B})_\Gamma^\perp = (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$ , since  $\mathcal{A} \rightarrow \mathcal{B}$  is a fixed point of  $\mathcal{F}^\perp$ . ◀

It seems now logical to prove the following soundness statement for eliminations:

► **Lemma 21** (Function elimination, preliminary). *Let  $\mathcal{A}, \mathcal{B}, \mathcal{C}$  be semantic types for  $A, B, C$ , resp. If  $a \perp \mathcal{A}_\Gamma$  and  $u \perp \mathcal{C}[\mathcal{B}]_\Gamma$  then  $E = \text{el}_{\rightarrow}^{10}(a, u) \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$ .*

However, such a lemma is not strong enough to justify the implication elimination rule, as from  $f \perp (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$  and  $E \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$  we only get  $f \cdot E \in \text{SN}$ , but we need the stronger  $f \cdot E \in \mathcal{C}_\Gamma$ . Thus, we prove the following stronger lemma.

► **Lemma 22** (Function elimination, revisited). *Let  $\mathcal{A}, \mathcal{B}, \mathcal{C}$  be semantic types for  $A, B, C$ , resp. If  $a \perp \mathcal{A}_\Gamma$  and  $u \perp \mathcal{C}[\mathcal{B}]_\Gamma$  and  $E = \text{el}_{\rightarrow}^{10}(a, u)$  and  $\vec{E} \in \mathcal{C}_\Gamma$  then  $E \cdot \vec{E} \in (\mathcal{A} \rightarrow \mathcal{B})_\Gamma$ .*

**Proof.** Let  $\mathcal{X}_\Gamma = \{E \cdot \vec{E} \mid E = \text{el}_{\rightarrow}^{10}(a, u) \text{ for some } a \perp \mathcal{A}_\Gamma \text{ and } u \perp \mathcal{C}[\mathcal{B}]_\Gamma, \text{ and } \vec{E} \in \mathcal{C}_\Gamma\}$ . To show  $\mathcal{X} \subseteq \mathcal{A} \rightarrow \mathcal{B}$ , by coinduction it is sufficient that  $\mathcal{X}$  is a post-fixpoint of  $\mathcal{F}^\perp$ . So assume  $E \cdot \vec{E} \in \mathcal{X}$  and  $I \in \mathcal{F}(\mathcal{X})$  and show  $v := I \cdot E \cdot \vec{E} \in \text{SN}$  by Lemma 15. To this end, we have to show that all  $\triangleright_\beta$ -redexes of  $v$  are SN. We distinguish the different introduction forms  $I$ .

**Case  $I = \text{in}_{\rightarrow}^{00}(t, u')$  with  $t \perp \mathcal{X}[\mathcal{A}]_\Gamma$  and  $u' \perp \mathcal{X}[\mathcal{B}]_\Gamma$ .** We have  $t[a] \perp \mathcal{X}_\Gamma$  by assumption on  $t$  and  $E \cdot \vec{E} \in \mathcal{X}_\Gamma$ , thus,  $t[a] \cdot E \cdot \vec{E} \in \text{SN}$ .

**Case  $I = \text{in}_{\rightarrow}^{11}(a, b)$  with  $a \perp \mathcal{A}_\Gamma$  and  $b \perp \mathcal{B}_\Gamma$ .** We have  $u[b] \perp \mathcal{C}_\Gamma$  and  $\vec{E} \in \mathcal{C}_\Gamma$ , thus  $u[b] \cdot \vec{E} \in \text{SN}$ .

**Case  $I = \text{in}_{\rightarrow}^{01}(t, b)$ .** In this case we have two weak head  $\beta$ -redexes which we handle as in the previous cases. ◀

Plugging these lemmata into the framework of Section 3, we obtain a new proof of  $\beta\pi$ -SN for ITTND.

## 7 Conclusion

We have successfully applied Girard's method, in its original form, to prove  $\beta$ -SN of ITTND, and the orthogonality method to prove  $\beta\pi$ -SN. The applicability of established methods is reassuring that ITTND does not offer a new form of computation asking for new theoretical justifications.

Our proof using orthogonality places rather high demands on the meta-theory: non-strictly positive coinductive definitions. Neither Coq nor Agda directly support those; in Coq, though, we can always fall back to impredicativity to construct the necessary fixed-point in the lattice of term or spine sets ordered by inclusion. In Martin-Löf Type Theory (MLTT) [24], the basis of Agda, such backups do not exist. This begs the question whether non-strictly positive (co)inductive types could be added in some form to MLTT without jeopardizing its soundness.

In the appendix (Appendix A), we investigate how the SN-method of Joachimski and Matthes [21, 26] can be applied to ITTND to prove  $\beta\pi$ -SN without the need for impredicativity nor non-strict positivity nor CPS-translation. Whether even an arithmetical proof à la David and Nour [9, 10] works for unoptimized ITTND is unclear, since already the introduction rules for implication are recursive and thus make implication semantically an inductive type.



A further question is the computational content of the normalization arguments presented here. The double negation on the meta level employed in the biorthogonals superficially resembles the CPS translation by Geuvers, van der Giessen, and Hurkens [16], and perhaps the latter can be extracted from our normalization proof.

Finally, the classical version of TTND has been little explored so far. It is unclear whether it has a computational interpretation that enjoys the strong normalization property.

---

## References

- 1 Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
- 2 Agda developers. *Agda 2.6.1 documentation*, 2020. URL: <http://agda.readthedocs.io/en/v2.6.1/>.
- 3 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 Int. Wksh., CSL '99, 8th Annual Conf. of the EACSL*, volume 1683 of *Lect. Notes in Comput. Sci.*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0\_32.
- 4 Franco Barbanera and Stefano Berardi. A symmetric lambda calculus for classical program extraction. *Inf. Comput.*, 125(2):103–117, 1996. doi:10.1006/inco.1996.0025.
- 5 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. of Autom. Reasoning*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.
- 6 Garrett Birkhoff. *Lattice Theory*. Amer. Math. Soc., Providence, RI, USA, 3rd edition, 1967.
- 7 Coq developers. The Coq proof assistant, version 8.12.0, 2019. doi:10.5281/zenodo.2554024.
- 8 Vincent Danos and Jean-Louis Krivine. Disjunctive tautologies as synchronisation schemes. In Peter Clote and Helmut Schwichtenberg, editors, *Computer Science Logic, 14th Int. Wksh., CSL 2000, 9th Annual Conf. of the EACSL*, volume 1862 of *Lect. Notes in Comput. Sci.*, pages 292–301. Springer, 2000. doi:10.1007/3-540-44622-2\_19.
- 9 René David. Normalization without reducibility. *Ann. Pure Appl. Logic*, 107(1–3):121–130, 2001. doi:10.1016/S0168-0072(00)00030-0.
- 10 René David and Karim Nour. Arithmetical proofs of strong normalization results for the symmetric lambda-mu-calculus. In Pawel Urzyczyn, editor, *Proc. of the 7th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2005*, volume 3461 of *Lect. Notes in Comput. Sci.*, pages 162–178. Springer, 2005. doi:10.1007/11417170\_13.
- 11 N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.
- 12 Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. URL: <http://gdz.sub.uni-goettingen.de/>.
- 13 Herman Geuvers and Tonny Hurkens. Deriving natural deduction rules from truth tables. In Sujata Ghosh and Sanjiva Prasad, editors, *Proc. of the 7th Indian Conference on Logic and Its Applications*, volume 10119 of *Lect. Notes in Comput. Sci.*, pages 123–138. Springer, 2017. doi:10.1007/978-3-662-54069-5\_10.
- 14 Herman Geuvers and Tonny Hurkens. Proof terms for generalized natural deduction. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd Int. Conf. on Types for Proofs and Programs, TYPES 2017*, volume 104 of *Leibniz Int. Proc. in Informatics*, pages 3:1–3:39. Schloss Dagstuhl, 2017. doi:10.4230/LIPIcs.TYPES.2017.3.
- 15 Herman Geuvers and Tonny Hurkens. Addendum to “Proof terms for generalized natural deduction”, 2020. URL: [http://www.cs.ru.nl/~herman/PUBS/addendum\\_to\\_TYPES.pdf](http://www.cs.ru.nl/~herman/PUBS/addendum_to_TYPES.pdf).
- 16 Herman Geuvers, Iris van der Giessen, and Tonny Hurkens. Strong normalization for truth table natural deduction. *Fundam. Inform.*, 170(1-3):139–176, 2019. doi:10.3233/FI-2019-1858.
- 17 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.

- 18 Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Math. Struct. in Comput. Sci.*, 11(3):301–506, 2001. doi:10.1017/S096012950100336X.
- 19 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoret. Comput. Sci.* Cambridge University Press, 1989.
- 20 Felix Joachimski and Ralph Matthes. Standardization and confluence for a lambda calculus with generalized applications. In Leo Bachmair, editor, *Rewriting Techniques and Applications, RTA 2000, Norwich, UK*, volume 1833 of *Lect. Notes in Comput. Sci.*, pages 141–155. Springer, 2000. doi:10.1007/10721975\_10.
- 21 Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel’s T. *Archive of Mathematical Logic*, 42(1):59–87, 2003. doi:10.1007/s00153-002-0156-9.
- 22 Sam Lindley and Ian Stark. Reducibility and  $\top\top$ -lifting for computation types. In Pawel Urzyczyn, editor, *Proc. of the 7th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2005*, volume 3461 of *Lect. Notes in Comput. Sci.*, pages 262–277. Springer, 2005. doi:10.1007/11417170\_20.
- 23 Zhaohui Luo. *ECC: An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. URL: <https://www.cs.rhul.ac.uk/home/zhaohui/ECS-LFCS-90-118.pdf>.
- 24 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
- 25 Ralph Matthes. Characterizing strongly normalizing terms of a calculus with generalized applications via intersection types. In José D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and J. B. Wells, editors, *Intersect. Types and Related Sys. (ITRS 2000), ICALP Satellite Wksh.*, pages 339–354. Carleton Scientific, Waterloo, ON, Canada, 2000.
- 26 Ralph Matthes. Non-strictly positive fixed-points for classical natural deduction. *Ann. Pure Appl. Logic*, 133(1–3):205–230, 2005. doi:10.1016/j.apal.2004.10.009.
- 27 Michel Parigot. Proofs of strong normalization for second order classical natural deduction. *J. Symb. Logic*, 62(4):1461–1479, 1997. doi:10.2307/2275652.
- 28 Michel Parigot. Strong normalization of second order symmetric  $\lambda$ -calculus. In Sanjiv Kapoor and Sanjiva Prasad, editors, *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lect. Notes in Comput. Sci.*, pages 442–453. Springer, 2000. doi:10.1007/3-540-44450-5\_36.
- 29 Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. in Comput. Sci.*, 10(3):321–359, 2000. URL: <http://journals.cambridge.org/action/displayAbstract?aid=44651>.
- 30 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Logic*, 32(2):198–212, 1967. doi:10.2307/2271658.
- 31 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- 32 Jérôme Vouillon and Paul-André Melliès. Semantic types: A fresh look at the ideal model for types. In Neil D. Jones and Xavier Leroy, editors, *Proc. of the 31st ACM Symp. on Principles of Programming Languages, POPL 2004*, pages 52–63. ACM Press, 2004. doi:10.1145/964001.964006.

## **A** Saturated Sets

In this appendix, we show how to adapt the original *saturated sets* method to IITND, first just for  $\beta$ -SN, then including  $\pi$ -reductions.

### **A.1** Saturated Sets for Computation Reductions

In the following, we adapt Tait’s method of saturated sets to show  $\beta$ -SN for ITTND. This is a variation of the proof by Geuvers and Hurkens [14].

We first observe that the set SN contains a weak-head redex already when (1) all of its reducts are SN and (2) its *lost terms* are SN, where a lost term is a subterm that could get dropped by all of the weak-head reductions. This fact is made precise by the following lemma:

► **Lemma 23.** *The following implications, written as rules, are valid closure properties of SN:*

$$\frac{t_1[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \text{SN} \quad t_2 \in \text{SN}}{\text{in}_{\rightarrow}^{00}(t_1, t_2) \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \text{SN}}$$

$$\frac{t[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \text{SN} \quad u[b] \cdot \vec{E} \in \text{SN} \quad b \in \text{SN}}{\text{in}_{\rightarrow}^{01}(t, b) \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \text{SN}} \quad \frac{u[b] \cdot \vec{E} \in \text{SN} \quad a_1, a_2, b \in \text{SN}}{\text{in}_{\rightarrow}^{11}(a_1, b) \cdot \text{el}_{\rightarrow}^{10}(a_2, u) \cdot \vec{E} \in \text{SN}}$$

(Spine  $\vec{E}$  may be empty in all cases.)

**Proof.** Each of these implications is proven by induction on the premises, establishing that the possible reducts of the term in the conclusion are SN. The weak-head reduct(s) are covered by the premises in each case. Reductions in lost terms are covered by the extra SN hypotheses. Reductions in preserved terms are covered by the main SN hypotheses. (This includes reductions in the spine  $\vec{E}$ .)

For example, consider the case for  $\text{in}_{\rightarrow}^{00}$ : By induction on  $t_1[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \text{SN}$  and  $t_2 \in \text{SN}$  show  $t' \in \text{SN}$  given  $\text{in}_{\rightarrow}^{00}(t_1, t_2) \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \rightarrow t'$ .

**Case  $t' = t_1[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E}$ .** Then  $t' \in \text{SN}$  by assumption.

**Case  $t' = \text{in}_{\rightarrow}^{00}(t_1, t'_2) \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E}$  where  $t_2 \rightarrow t'_2$ .** Then  $t' \in \text{SN}$  by induction hypothesis  $t'_2 \in \text{SN}$ .

**Case  $t' = \text{in}_{\rightarrow}^{00}(t'_1, t_2) \cdot \text{el}_{\rightarrow}^{10}(a', u') \cdot \vec{E}'$  where  $(t_1, a, u, \vec{E}) \rightarrow (t'_1, a', u', \vec{E}')$  (a single reduction in one of these subterms).** Then  $t_1[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \rightarrow^+ t'_1[a'] \cdot \text{el}_{\rightarrow}^{10}(a', u') \cdot \vec{E}'$  (several steps possible, e.g., if reduction was in  $a$  and  $t_1$  mentions the 0th de Bruijn index). Thus,  $t' \in \text{SN}$  by induction hypothesis on  $t'_1[a'] \cdot \text{el}_{\rightarrow}^{10}(a', u') \cdot \vec{E}' \in \text{SN}$ . ◀

Mimicking Lemma 23, the *saturation*  $\overline{\mathcal{A}}$  of a term set is – in the case of the implicational fragment of ITTND – defined inductively as follows:

$$\frac{t \in \mathcal{A}_{\Gamma} \quad t_1[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma} \quad t_2 \in \text{SN}}{t \in \overline{\mathcal{A}}_{\Gamma} \quad \text{in}_{\rightarrow}^{00}(t_1, t_2) \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma}}$$

$$\frac{t[a] \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma} \quad u[b] \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma} \quad b \in \text{SN}}{\text{in}_{\rightarrow}^{01}(t, b) \cdot \text{el}_{\rightarrow}^{10}(a, u) \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma}} \quad \frac{u[b] \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma} \quad a_1, a_2, b \in \text{SN}}{\text{in}_{\rightarrow}^{11}(a_1, b) \cdot \text{el}_{\rightarrow}^{10}(a_2, u) \cdot \vec{E} \in \overline{\mathcal{A}}_{\Gamma}}$$

► **Lemma 24.**  $\overline{\text{SN}} \subseteq \text{SN}$ .

**Proof.** We show  $t \in \text{SN}$  by induction on  $t \in \overline{\text{SN}}$ , using Lemma 23. ◀

► **Corollary 25.** *If  $\mathcal{A} \subseteq \text{SN}$  then  $\overline{\mathcal{A}} \subseteq \text{SN}$ .*

**Proof.** Since closure is a monotone operator, we have  $\overline{\mathcal{A}} \subseteq \overline{\overline{\text{SN}}} \subseteq \text{SN}$  by Lemma 24. ◀

A saturated set  $\mathcal{A} \in \text{SAT}$  must fulfill the following three properties:

- SAT1  $\mathcal{A} \subseteq \text{SN}$  (contains only SN terms).
- SAT2 If  $\vec{E} \in \text{SN}$  then  $x \cdot \vec{E} \in \mathcal{A}$  (contains SN neutrals).
- SAT3  $\overline{\mathcal{A}} \subseteq \mathcal{A}$  (closed under SN weak-head expansion).

Semantic implication can now be defined as:

$$f \in (\mathcal{A} \rightarrow \mathcal{B})_{\Gamma} \iff f \in \text{SN} \text{ and } \forall \mathcal{C} \in \text{SAT}, \tau : \Delta \leq \Gamma, a \in \mathcal{A}_{\Delta}, t \in \mathcal{C}[\mathcal{B}]_{\Delta}. f\tau \cdot \text{el}_{\rightarrow}^{10}(a, t) \in \mathcal{C}_{\Delta}.$$

► **Lemma 26** (Function space on SAT). *If  $\mathcal{A} \subseteq \text{SN}$  and  $\mathcal{B} \in \text{SAT}$ , then  $\mathcal{A} \rightarrow \mathcal{B} \in \text{SAT}$ .*

**Proof.** SAT1 holds by definition. SAT2 holds by SAT2 of  $\mathcal{B}$ . SAT3 holds by SAT3 of  $\mathcal{B}$ . ◀

The introductions rules for implication are indeed modeled for the SAT variant of semantic function space. For instance,  $\text{in}_{\rightarrow}^{01}$ :

► **Lemma 27** (Introduction ( $\text{in}_{\rightarrow}^{01}$ )). *If  $t \in (\mathcal{A} \rightarrow \mathcal{B})[\mathcal{A}]_{\Gamma}$  and  $b \in \mathcal{B}_{\Gamma}$  then  $\text{in}_{\rightarrow}^{01}(t, b) \in (\mathcal{A} \rightarrow \mathcal{B})_{\Gamma}$ .*

**Proof.** Assume  $\mathcal{C} \in \text{SAT}$  and  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_{\Delta}$  and  $u \in \mathcal{C}[\mathcal{B}]_{\Delta}$  and show  $\text{in}_{\rightarrow}^{01}(t, b)\tau \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \mathcal{C}_{\Delta}$ . Using SAT3 on  $\mathcal{C}$ , it is sufficient to show that (1)  $t(\tau.a) \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \mathcal{C}_{\Delta}$  and (2)  $u[b\tau] \in \mathcal{C}_{\Delta}$  and (3)  $b\tau \in \text{SN}$ . Subgoals (2) and (3) follow since  $b\tau \in \mathcal{B}_{\Delta}$ , and (1) holds since  $t(\tau.a) \in (\mathcal{A} \rightarrow \mathcal{B})_{\Delta}$ . ◀

## A.2 On Permutation Reductions

Ralph Matthes' [26] formulation of saturated sets in the context of  $\pi$ -reductions can also be adapted to ITTND.

First, we observe that Lemma 23 still holds if  $\pi$ -reductions are taken into account. This is because any reduction in the spine of a conclusion can be simulated in the spine of at least one of the premises.

Thus, SAT3 can remain in place, only SAT2 needs to be reformulated, since a neutral  $x \cdot \vec{E}$  can be subject to a  $\beta$ -reduction after a  $\pi$ -reduction in  $\vec{E}$  has created a new  $\beta$ -redex. Towards a reformulation of SAT2, we observe the following closure properties of SN by neutral terms:

► **Lemma 28** (Neutral closure of SN). *The following implications, written as rules, are valid closure properties of SN:*

$$\frac{}{x \in \text{SN}} \quad \frac{a \in \text{SN} \quad u \in \text{SN}}{x \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \text{SN}} \quad \frac{x \cdot E_1\{E_2\} \cdot \vec{E} \in \text{SN} \quad E_2 \cdot \vec{E} \in \text{SN}}{x \cdot E_1 \cdot E_2 \cdot \vec{E} \in \text{SN}}$$

The extra assumption  $E_2 \cdot \vec{E} \in \text{SN}$  in the third implication is equivalent to  $y \cdot E_2 \cdot \vec{E} \in \text{SN}$  for some variable  $y$ . In the implicational fragment, this assumption is redundant since the composition  $\text{el}_{\rightarrow}^{10}(a, u)\{E_2\} = \text{el}_{\rightarrow}^{10}(a, u \cdot E_2\uparrow)$  does not lose  $E_2$ . In particular, any reduction in  $E_2 \cdot \vec{E}$  can be replayed in  $x \cdot E_1\{E_2\} \cdot \vec{E}$ . However, in general there can be eliminations with only positive premises, such as  $\text{el}_{\rightarrow}^1(a)$  for negation, where composition  $\text{el}_{\rightarrow}^1(a)\{E_2\} = \text{el}_{\rightarrow}^1(a)$

simply drops  $E_2$ . This means that reductions in part  $E_2 \cdot \vec{E}$  of  $x \cdot E_1 \cdot E_2 \cdot \vec{E}$  cannot necessarily be simulated in  $x \cdot E_1 \{E_2\} \cdot \vec{E}$ . In particular, a reduction  $E_2 \cdot E_3 \rightarrow_{\pi} E_2 \{E_3\}$  could lead to new  $\beta$ -redexes which have no correspondence in  $E_1 \{E_2\} \cdot E_3$ .

Mimicking Lemma 28, we *extend* the definition of saturation  $\bar{\mathcal{C}}$  of a semantic type  $\mathcal{C}$  for  $C$  by the following three clauses:

$$\begin{array}{c} \text{VAR } \frac{x : \Gamma \vdash C}{x \in \bar{\mathcal{C}}_{\Gamma}} \quad \text{EL } \frac{x : \Gamma \vdash A \rightarrow B \quad a \in \text{SN}(\Gamma \vdash A) \quad u \in \bar{\mathcal{C}}_{\Gamma, B}}{x \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \bar{\mathcal{C}}_{\Gamma}} \\ \\ \text{PI } \frac{\begin{array}{c} x : \Gamma \vdash A \quad E_1 : \Gamma \mid A \vdash B \quad x \cdot E_1 \{E_2\} \cdot \vec{E} \in \bar{\mathcal{C}}_{\Gamma} \\ \tau : \Delta \leq \Gamma \quad y : \Delta \vdash B \quad y \cdot (E_2 \cdot \vec{E})\tau \in \bar{\mathcal{C}}_{\Delta} \end{array}}{x \cdot E_1 \cdot E_2 \cdot \vec{E} \in \bar{\mathcal{C}}_{\Gamma}} \end{array}$$

Note that a premise such as  $y \cdot (E_2 \cdot \vec{E})\tau \in \text{SN}$  would be too weak to show that semantic function space is saturated.

We revise the definition of SAT such that SAT3 uses the extended definition of closure, obsoleting SAT2.

► **Lemma 29** (Function space on SAT). *If  $\mathcal{A} \subseteq \text{SN}$  and  $\mathcal{B} \in \text{SAT}$ , then  $\mathcal{A} \rightarrow \mathcal{B} \in \text{SAT}$ .*

**Proof.** We shall focus on the new closure conditions for SAT:

- VAR: Show  $x \in (\mathcal{A} \rightarrow \mathcal{B})_{\Gamma}$ . Clearly  $x \in \text{SN}$ . Now assume  $\mathcal{C} \in \text{SAT}$  and  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_{\Delta}$  and  $u \in \mathcal{C}[\mathcal{B}]_{\Delta}$  and show  $x\tau \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \mathcal{C}_{\Delta}$ . By EL, it is sufficient that  $a \in \text{SN}$  and  $u \in \mathcal{C}_{\Delta, B}$ . By VAR we have  $x_0 \in \mathcal{B}_{\Delta, B}$ , thus  $u(\uparrow x_0) = u \in \mathcal{C}_{\Delta, B}$ .
- EL: Assume  $x : \Gamma \vdash A_0 \rightarrow B_0$  and  $a_0 \in \text{SN}(\Gamma \vdash A_0)$  and  $u_0 \in \overline{\mathcal{A} \rightarrow \mathcal{B}}_{\Gamma, B_0}$  and show  $x \cdot \text{el}_{\rightarrow}^{10}(a_0, u_0) \in \overline{\mathcal{A} \rightarrow \mathcal{B}}_{\Gamma}$ . First,  $x \cdot \text{el}_{\rightarrow}^{10}(a_0, u_0) \in \text{SN}$ . Further, assume  $\mathcal{C} \in \text{SAT}$  and  $\tau : \Delta \leq \Gamma$  and  $a \in \mathcal{A}_{\Delta}$  and  $u \in \mathcal{C}[\mathcal{B}]_{\Delta}$  and show  $(x \cdot \text{el}_{\rightarrow}^{10}(a_0, u_0))\tau \cdot \text{el}_{\rightarrow}^{10}(a, u) \in \mathcal{C}_{\Delta}$ . Using PI, we first discharge the last subgoal  $x_0 \cdot \text{el}_{\rightarrow}^{10}(a, u)\uparrow \in \mathcal{C}_{\Delta, (A \rightarrow B)}$  by EL for  $\mathcal{C}$  with  $a\uparrow \in \mathcal{A}_{\Delta, (A \rightarrow B)}$  and  $u(\uparrow\uparrow) \in \mathcal{C}_{\Delta, (A \rightarrow B), B}$ . It remains to show that  $x\tau \cdot \text{el}_{\rightarrow}^{10}(a_0\tau, u_0(\uparrow\tau)) \cdot \text{el}_{\rightarrow}^{10}(a, u)\uparrow \in \mathcal{C}_{\Delta}$ . Again, we use EL for  $\mathcal{C}$ . Clearly  $a_0\tau \in \text{SN}$ , so it remains to show that  $u_0(\uparrow\tau) \cdot \text{el}_{\rightarrow}^{10}(a\uparrow, u(\uparrow\uparrow)) \in \mathcal{C}_{\Delta, B_0}$ . Since  $u_0(\uparrow\tau) \in (\mathcal{A} \rightarrow \mathcal{B})_{\Delta, B_0}$  and  $a\uparrow \in \mathcal{A}_{\Delta, B_0}$  and  $u(\uparrow\uparrow) \in \mathcal{C}[\mathcal{B}]_{\Delta, B_0}$ , this is the case by definition of  $\mathcal{A} \rightarrow \mathcal{B}$ .
- PI: The case PI for  $\mathcal{A} \rightarrow \mathcal{B}$  is shown by PI for  $\mathcal{C}$  (what  $\mathcal{C}$  refers to, see the previous cases). This part is a bit tedious to spell out, but completely uninteresting, since just  $\vec{E}$  is extended by another  $\text{el}_{\rightarrow}^{10}$ -elimination at the end. ◀

The soundness of the introductions carries over from the previous section (Lemma 27) since the saturated sets are still closed by weak head expansion.

This concludes the  $\beta\pi$ -SN proof for ITTND using saturated sets.



# Extending Equational Monadic Reasoning with Monad Transformers

Reynald Affeldt 

National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

David Nowak

Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France

---

## Abstract

---

There is a recent interest for the verification of monadic programs using proof assistants. This line of research raises the question of the integration of monad transformers, a standard technique to combine monads. In this paper, we extend *Monae*, a Coq library for monadic equational reasoning, with monad transformers and we explain the benefits of this extension. Our starting point is the existing theory of modular monad transformers, which provides a uniform treatment of operations. Using this theory, we simplify the formalization of models in *Monae* and we propose an approach to support monadic equational reasoning in the presence of monad transformers. We also use *Monae* to revisit the lifting theorems of modular monad transformers by providing equational proofs and explaining how to patch a known bug using a non-standard use of Coq that combines impredicative polymorphism and parametricity.

**2012 ACM Subject Classification** Theory of computation → Logic and verification; Software and its engineering → Formal software verification

**Keywords and phrases** monads, monad transformers, Coq, impredicativity, parametricity

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.2

**Related Version** *Previous Version*: <https://arxiv.org/abs/2011.03463>

**Supplementary Material** *Software (Proof Scripts)*: <https://github.com/affeldt-aist/monae/>  
archived at `swh:1:dir:2d68878d365fe72744f8b085fa29df385567f6c9`

**Funding** We acknowledge the support of the JSPS KAKENHI Grant Number 18H03204.

**Acknowledgements** We thank all the participants of the JSPS-CNRS bilateral program “FoRmal tools for IoT sEcurity” (PRC2199) for fruitful discussions. We also thank Takafumi Saikawa for his comments. This work is based on joint work with Célestine Sauvage [29].

## 1 Introduction

There is a recent interest for the formal verification of monadic programs stemming from *monadic equational reasoning*: an approach to the verification of monadic programs that emphasizes equational reasoning [8, 9, 25–27]. In this approach, an effect is represented by an operator belonging to an interface together with equational laws. The interfaces all inherit from the type class of monads and the interfaces are organized in a hierarchy where they are extended and composed. There are several efforts to bring monadic equational reasoning to proof assistants [1, 2, 28].

In monadic equational reasoning, the user cannot rely on the *model* of the interfaces because the implementation of the corresponding monads is kept hidden. The construction of models is nevertheless important to avoid mistakes when adding equational laws [1]. This means that a formalization of monadic equational reasoning needs to provide tools to formalize models.

In this paper, we extend an existing formalization of monadic equational reasoning (called *MONAE* [2]) with *monad transformers*. Monad transformers is a well-known approach to combine monads that is both modular and practical [20]. It is also commonly used to write Haskell programs. The interest in extending monadic equational reasoning with monad



© Reynald Affeldt and David Nowak;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 2; pp. 2:1–2:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

transformers is therefore twofold: (1) it enriches the toolbox to build formal models of monad interfaces, and (2) it makes programs written with monad transformers amenable to equational reasoning.

In fact, the interest for a formal theory of monad transformers goes beyond its application to monadic equational reasoning. Past research advances about monad transformers could have benefited from formalization. For example, a decade ago, Jaskelioff identified a lack of uniformity in the definitions of the liftings of operations through monad transformers [15]. He proposed *modular monad transformers* which come with a uniform definition of lifting for operations that qualify as *sigma-operations* or their sub-class of *algebraic operations*. Unfortunately, the original proposal in terms of System  $F\omega$  was soon ruled out as faulty [17, Sect. 6] [14, p. 7] and its fix gave rise to a more involved presentation in terms of (non-trivial) category theory [17]. More recently, this is the comparison between monad transformers and algebraic effects that attracts attention, and it connects back to reasoning using equational laws (e.g., [30, Sect. 7]). This is why in this paper, not only do we provide examples of monad transformers and applications of monadic equational reasoning, but also formalize a theory of monad transformers.

**Contributions.** In this paper, we propose a formalization in the COQ proof assistant [32] of monad transformers. This formalization comes as an extension of MONAE, an existing library that provides a hierarchy of monad interfaces [2]. The benefits of this extension are as follows.

- The addition of sigma-operations and of monad transformers to MONAE improves the implementation of models of monads. These models are often well-known and it is tempting to define them in an ad hoc way. Sigma-operations help us discipline proof scripts and naming, which are important aspects of proof-engineering.
- We illustrate with an example how to extend MONAE to verify a program written with a monad transformer. Verification is performed by equational reasoning using equational laws from a monad interface whose model is built using a monad transformer.
- We use our formalization of monad transformers to formalize the theory of lifting of modular monad transformers. Thanks to MONAE, the main theorems of modular monad transformers can be given short formal proofs in terms of equational reasoning.

Regarding the theory of lifting of modular monad transformers, our theory fixes the original presentation [15]. This fix consists in a non-standard use of COQ combining impredicativity and parametricity (as implemented by PARAMCOQ [18]) that allows for an encoding using the language of the proof assistant and thus avoids the hassle of going through a technical formalization of category theory (which is how Jaskelioff fixed his original proposal). It must be said that this was not possible at the time of the original paper on modular monad transformers because parametric models of dependent type theory were not known [4] (but were “expected” [16]). We are therefore in the situation where formalization using a proof assistant allowed for a fruitful revisit of pencil-and-paper proofs.

Regarding the benefit of extending MONAE with sigma-operations and monad transformers, we would like to stress that this is also a step towards more modularity in our formalization of monadic equational reasoning. Indeed, one important issue that we have been facing is the quality of our proof scripts. Proof scripts that reproduce monadic equational reasoning must be as concise as they are on paper. Proof scripts that build models (and prove lemmas) should be maintainable (to be improved or fixed easily in case of changes in the hypotheses) and understandable (this means having a good balance between the length of the proof script and its readability). This manifests as mundane but important tasks such as factorization



of proof scripts, generalization of lemmas, abstraction of data structures, etc. From the viewpoint of proof-engineering, striving for modularity is always a good investment because it helps in breaking the formalization task into well-identified, loosely-coupled pieces.

**Outline.** In Sect. 2, we recall the main constructs of MONAE. In Sect. 3, we formalize the basics of modular monad transformers: sigma-operations, monad transformers, and their variants (algebraic operations and functorial monad transformers). Section 4 is our first application: we show with an example how to extend MONAE to verify a program written using monad transformers. In Sect. 5, we use our formalization of monad transformers to prove a first theorem about modular monad transformers (namely, the lifting of algebraic operations) using equational reasoning. In Sect. 6, we formalize (and fix) the main theorem of modular monad transformers (namely, the lifting of sigma-operations that are not necessarily algebraic along functorial monad transformers). We review related work in Sect. 7 and conclude in Sect. 8.

## 2 Overview of the Monae Library

MONAE [2] is a formal library implemented in the COQ proof assistant [32] to support monadic equational reasoning [9]. It takes advantage of the rewriting capabilities of the tactic language called SSREFLECT [10] to achieve formal proofs by rewriting that are very close to their pencil-and-paper counterparts. MONAE provides a hierarchy of monad interfaces formalized using the methodology of *packed classes* [6]. Effects are declared as operations in interfaces together with equational laws, and some effects extend others by (simple or multiple) inheritance. This modularity is important to achieve natural support for monadic equational reasoning.

Let us briefly explain some types and notations provided by MONAE that we will use in the rest of this paper. MONAE provides basic category-theoretic definitions such as functors, natural transformations, and monads. By default, they are specialized to  $\mathbb{U}0$ , the lowest universe in the hierarchy of COQ types, understood as a category<sup>1</sup>. The type of functors is `functor`. The application of a functor `F` to a function `f` is denoted by `F # f`. The composition of functors is denoted by the infix notation `\o`. The identity functor is denoted by `FId`. Natural transformations from the functor `F` to the functor `G` are denoted by `F ~> G`. Natural transformations are formalized by their components (represented by the type `forall A, F A -> G A`, denoted by `F ~-> G`) together with the proof that they are natural, i.e., the proof that they satisfy the following predicate:

```
Definition naturality (M N : functor) (m : M ~-> N) :=
  forall (A B : UU0) (h : A -> B), (N # h) \o m A = m B \o (M # h).
```

(The infix notation `\o` is for function composition.) Vertical composition of natural transformations is denoted by the infix notation `\v`. The application of a functor `F` to a natural transformation `n` is denoted by `F ## n`.

The type of monads is `monad`, which inherits from the type `functor`. Let `M` be of type `monad`. Then `Ret` is a natural transformation `FId ~> M` and `Join` is a natural transformation `M \o M -> M`. Using `Ret` and `Join`, we define the standard bind operator with the notation `>>=`.

In this paper, we show COQ proof scripts verbatim when it is reasonable to do so. When we write mathematical formulas, we keep the same typewriter font, but, for clarity and to ease reading, we make explicit some information that would otherwise be implicitly inferred

<sup>1</sup> MONAE also provides a more generic setting [24, file `category.v`] but we do not use it in this paper.

by COQ. For example, one simply writes `Ret` or `Join` in proof scripts written using MONAE because it has been implemented in such a way that COQ infers from the context which monad they refer to and which type they apply to. In mathematical formulas, we sometimes make the monad explicit by writing it as a superscript of `Ret` or `Join`, and we sometimes write the argument of a function application as a subscript. This leads to terms such as  $\text{Ret}_A^M$ : the unit of the monad  $M$  applied to some type  $A$ .

See the online development for technical details (in particular, [24, file `hierarchy.v`]).

### 3 Sigma-operations and Monad Transformers in Monae

The first step is to formalize sigma-operations (Sect. 3.1) and monad transformers (Sect. 3.3). We illustrate sigma-operations with the example of the model of the state monad (Sect. 3.1.1) and its get operation (Sect. 3.1.2).

#### 3.1 Extending Monae with Sigma-operations

Given a functor  $E$ , an  $E$ -operation for a monad  $M$  (sigma-operation for short) is a natural transformation from  $E \circ M$  to  $M$ . The fact that sigma-operations are defined in terms of natural transformations is helpful to build models because it involves structured objects (functors and natural transformations) already instrumented with lemmas. In other words, we consider sigma-operations as a disciplined way to formalize effects. For illustration, we explain how the get operation of the state monad is formalized.

##### 3.1.1 Example: Model of the State Monad

First we define a model `State.t` for the state monad (without `get` and `put` for the time being). We assume a type `S` (line 2) and define the action on objects `actO` (line 3), abbreviated as  $M$  (line 4). We define the action on morphisms `map` (line 5) and prove the functor laws (omitted here, see [24, file `monad_model.v`] for details). This provides us with a functor `functor` (line 8, `Functor.Pack` and `Functor.Mixin` are constructors from MONAE and are named after the packed classes methodology [6]). We define the unit of the monad by first providing its components `ret_component` (line 9), and prove naturality (line 10, proof script omitted). We then package this proof to form a genuine natural transformation at line 12 (`Natural.Pack` and `Natural.Mixin` are constructors from MONAE). We furthermore define `bind` (line 14), prove the properties of the unit and `bind` (omitted). Finally, we call the function `Monad_of_ret_bind` from MONAE to build the monad (line 16):

```

1  (* in Module State *)
2  Variable S : UU0.
3  Definition actO := fun A => S -> A * S.
4  Local Notation M := actO.
5  Definition map A B (f : A -> B) (m : M A) : M B :=
6    fun (s : S) => let (x1, x2) := m s in (f x1, x2).
7  (* functor laws map_id and map_comp omitted *)
8  Definition functor := Functor.Pack (Functor.Mixin map_id map_comp).
9  Definition ret_component : FId ~-> M := fun A a => fun s => (a, s).
10 Lemma naturality_ret : naturality FId functor ret_component.
11 (* proof script of naturality omitted *)
12 Definition ret : FId ~-> functor :=
13   Natural.Pack (Natural.Mixin naturality_ret).
14 Definition bind := fun A B (m : M A) (f : A -> M B) => uncurry f \o m.
15 (* proofs of neutrality of ret and of associativity of bind omitted *)
16 Definition t := Monad_of_ret_bind left_neutral right_neutral associative.
```

### 3.1.2 Example: The Get Operation as a Sigma-operation

By definition, for each sigma-operation we need a functor. The functor corresponding to the get operation is defined below as `Get.func` (line 5): `acto` is the action on the objects, `actm` is the action on the morphisms (the prefix `@` disables implicit arguments in `COQ`):

```

1  (* in Module Get *)
2  Variable S : UU0.
3  Definition acto X := S -> X.
4  Definition actm (X Y : UU0) (f : X -> Y) (t : acto X) : acto Y := f \o t.
5  Program Definition func := Functor.Pack (@Functor.Mixin _ actm _ _).
6  (* proofs of the functors law omitted *)

```

We then define the sigma-operation itself (`StateOps.get_op` at line 7), which is a natural transformation from `Get.func S \0 M` to `M`, where `M` is the state monad `State.t S` built in Sect. 3.1.1. Note that this get operation ( $\lambda s. k s s$ , line 4) is *not* the usual operation [15, Example 13].

```

1  (* in Module StateOps *)
2  Variable S : UU0.
3  Local Notation M := (State.t S).
4  Definition get A (k : S -> M A) : M A := fun s => k s s.
5  Lemma naturality_get : naturality (Get.func S \0 M) M get.
6  (* proof script of naturality omitted *)
7  Definition get_op : (Get.func S).-operation M :=
8    Natural.Pack (Natural.Mixin naturality_get).

```

### 3.1.3 Example: Model of the Interface of the State Monad

MONAE originally comes with an interface `stateMonad` for the state monad (*with* the get and put operations). It implements the interface as presented by Gibbons and Hinze [9, Sect. 6]; it therefore expects the operations to be the usual ones. We show how to instantiate it using the definition of sigma-operations. First, we need to define the usual get from `StateOps.get_op` (line 4 below):

```

1  (* in Module ModelState *)
2  Variable S : UU0.
3  Local Notation M := (ModelMonad.State.t S).
4  Definition get : M S := StateOps.get_op _ Ret.

```

We do the same for the put operation (omitted). We then build the model of interface of the state monad (with its operations) using the appropriate constructors from MONAE:

```

Program Definition state : stateMonad S := MonadState.Pack (MonadState.Class
  (@MonadState.Mixin _ _ get put _ _ _ _)).
(* proofs of the laws of get and put automatically discharged *)

```

Similarly, using sigma-operations, we have formalized the operations of the list, the output, the state, the environment, and the continuation monads, which are the monads discussed along with modular monad transformers [15, Fig. 1] (see [24, file `monad_model.v`] for their formalization).

## 3.2 The Sub-class of Algebraic Operations

An E-operation `op` for `M` is *algebraic* [15, Def. 15] when it satisfies the predicate `algebraicity` defined as follows in `COQ` (observe the position of the continuation `>>= f`):

## 2:6 Extending Equational Monadic Reasoning with Monad Transformers

```
forall A B (f : A -> M B) (t : E (M A)),
  op A t >>= f = op B ((E # (fun m => m >>= f)) t).
```

Algebraic operations are worth distinguishing because they lend themselves more easily to lifting, and this result can be used to define lifting for the whole class of sigma-operations (this is the purpose of Sections 5 and 6). We can check using COQ that, as expected, all the operations discussed along with modular monad transformers [15, Fig. 1] are algebraic except for flush, local, and handle<sup>2</sup>.

### Example: the Get operation is Algebraic

For example, the get operation of the state monad is algebraic:

```
Lemma algebraic_get S : algebraicity (@StateOps.get_op S). Proof. by []. Qed.
```

In the COQ formalization, we furthermore provide the type `E.-aoperation M` (note the prefix “a”) of an `E.-operation M` that is actually algebraic. For example, here is how we define the algebraic version of the get operation:

```
Definition get_aop S : (StateOps.Get.func S).-aoperation (ModelMonad.State.t S) :=
  AOperation.Pack (AOperation.Class (AOperation.Mixin (@algebraic_get S))).
```

## 3.3 Extending Monae with Monad Transformers

Given two monads `M` and `N`, a *monad morphism* `e` is a function of type `M -> N` such that for all types `A, B` the following laws hold:

- `e A \o Ret = Ret.` (\* MonadMLaws.ret \*)
- `forall (m : M A) (f : A -> M B),` (\* MonadMLaws.bind \*)  
`e B (m >>= f) = e A m >>= (e B \o f).`

In COQ, we define the type of monad morphisms `monadM` that implement the two laws above. Monad morphisms are also natural transformations (this can be proved easily using the laws of monad morphisms). We therefore equip monad morphisms `e` with a canonical structure of natural transformation. Since it is made canonical, COQ is able to infer it in proof scripts but we need to make it explicit in statements; we provide the notation `monadM_nt e` for that purpose.

A *monad transformer* `t` is a function of type `monad -> monad` with an operator `Lift` such that for any monad `M`, `Lift t M` is a monad morphism from `M` to `t M`. Let `monadT` be the type of monad transformers in MONAE. We reproduced all the examples of modular monad transformers (state, exception, environment, output, continuation monad transformers, resp. `stateT`, `exceptT`, `envT`, `outputT`, and `contT` in [24, file `monad_transformer.v`]).

### Example: The Exception Monad Transformer

Let us assume given some type `Z : UU0` for exceptions and some monad `M`. First, we define the action on objects of the monad transformed by the exception monad transformer (the type `Z + X` represents the sum type of the types `Z` and `X`):

<sup>2</sup> In fact, we had to fix the output operation of the output monad. Indeed, it is defined as follows in [15, Example 32]: `output((w, m) : W × OX) : OX ≐ let (x, w') = m in (x, append(w', w))`. We changed `append(w', w)` to `append(w, w')` to be able to prove algebraicity.

**Definition** `MX` := `fun X : UUO => M (Z + X)`.

We also define the unit and the bind operator of the transformed monad (the constructors `inl/inr` inject a type into the left/right of a sum type):

**Definition** `retX X x : MX X := Ret (inr x)`.

**Definition** `bindX X Y (t : MX X) (f : X -> MX Y) : MX Y :=`  
`t >>= fun c => match c with inl z => Ret (inl z) | inr x => f x end.`

Second, we define the monad morphism that will be returned by the lift operator of the monad transformer. In COQ, we can formalize the corresponding function by constructing the desired monad assuming `M`. This is similar to the construction of the state monad we saw in Sect. 3.1. We start by defining the underlying functor `MX_map`, prove the two functor laws (let us call `MX_map_i` and `MX_map_o` these proofs), and package them as a functor:

**Definition** `MX_functor := Functor.Pack (Functor.Mixin MX_map_i MX_map_o)`.

We then provide the natural transformation `retX_natural` corresponding to `retX` and call the MONAE constructor `Monad_of_ret_bind` (like we did in Sect. 3.1):

**Program Definition** `exceptTmonad : monad :=`  
`@Monad_of_ret_bind MX_functor retX_natural bindX _ _ _.`  
*(\* proofs of monad laws omitted \*)*

Then we define the lift operation as a function that given a computation `m` in the monad `M X` returns a computation in the monad `exceptTmonad X`:

**Definition** `liftX X (m : M X) : exceptTmonad X := m >>= (@RET exceptTmonad _)`.

(The function `RET` is a variant of `Ret` better suited for type inference here.) We can finally package the definition of `liftX` to form a monad morphism:

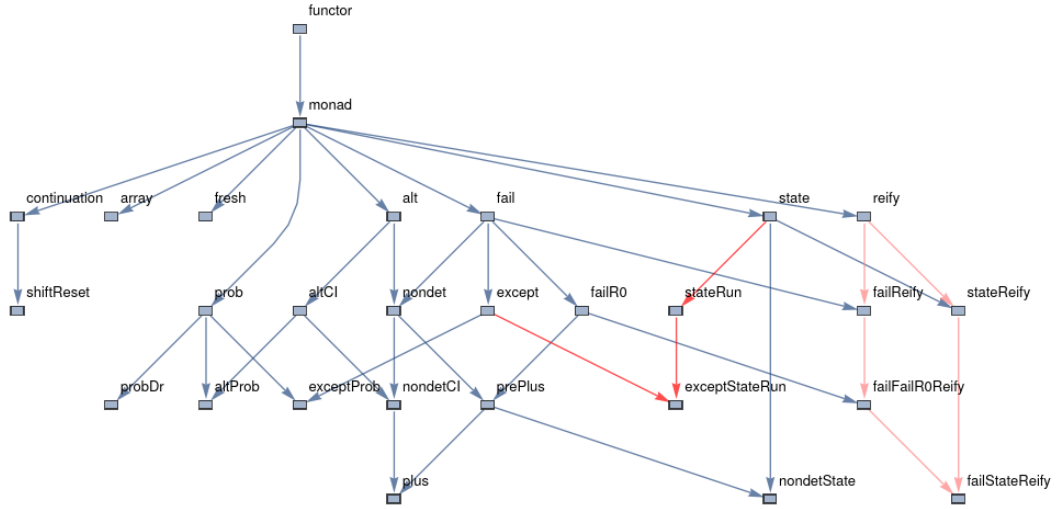
**Program Definition** `exceptTmonadM : monadM M exceptTmonad :=`  
`monadM.Pack (@monadM.Mixin _ _ liftX _ _).`  
*(\* proof of monad morphism laws omitted \*)*

The exception monad transformer merely packages the monad morphism we have just defined to give it the type `monadT`:

**Definition** `exceptT Z := MonadT.Pack (MonadT.Mixin (exceptTmonadM Z))`.

One might wonder what is the relation between the monads that can be built with these monad transformers and the monads already present in MONAE. For example, in Sect. 3.1, we already mentioned the `stateMonad` interface and we built a model for it (namely, `ModelState.state`). On the other hand, we can now, say, build a model for the identity monad (let us call it `identity`) and build a model for that state monad as `stateT S identity` (we have not provided the details of `stateT`, see [2]). We can actually prove in COQ that `stateT S identity` and `State.t` are *equal*<sup>3</sup>, so that no confusion has been introduced by extending MONAE with monad transformers.

<sup>3</sup> [24, Section `instantiations_with_the_identity_monad`, file `monad_model.v`]



■ **Figure 1** Hierarchy of Monad Interfaces Provided by MONAE.

### 3.4 Functorial Monad Transformers

A *functorial monad transformer* [15, Def. 20] is a monad transformer  $\mathfrak{t}$  with a function  $h$  (hereafter denoted by  $\mathsf{Hmap} \ \mathfrak{t}$ ) of type

```
forall (M N : monad), (M ~> N) -> (t M ~> t N)
```

such that (1)  $h$  preserves monad morphisms (the laws `MonadMLaws.ret` and `MonadMLaws.bind` seen in Sect. 3.3), (2)  $h$  preserves identities and composition of natural transformations, and (3)  $\mathsf{Lift} \ \mathfrak{t}$  is natural, i.e.,

```
forall (M N : monad) (n : M ~> N) X, h M N n X \o Lift t M X = Lift t N X \o n X.
```

Note that we cannot define the naturality of  $\mathsf{Lift} \ \mathfrak{t}$  using the predicate `naturality` we saw in Sect. 2 because it is restricted to endofunctors on  $\mathcal{U}\mathcal{O}$ . Also note that Jaskelioff distinguishes monad transformers from functorial monad transformers while Maillard defines monad transformers as functorial by default [21, Def. 4.1.1].

## 4 Application 1: Monadic Equational Reasoning in the Presence of Monad Transformers

We apply our formalization of monad transformers to the verification of a recursive program combining the effects of state and exception. We argue that this program is similar in style to what an Haskell programmer would typically write with monad transformers. Despite this programming style and the effects, the correctness proof is by equational reasoning.

### 4.1 Extending the Hierarchy

The first thing to do is to extend the hierarchy of interfaces with `stateRunMonad` and `exceptStateRunMonad` (Fig. 1).

The interface `stateRunMonad` is a parameterized interface that extends `stateMonad` with the primitive `RunStateT` and its equations. Concretely, let  $N$  be a monad and  $S$  be the type of states. When  $m$  is a computation in the monad `stateRunMonad S N`, `RunStateT m s` runs  $m$  in a state  $s$  and returns a computation in the monad  $N$ . There is one equation for each combination of `RunStateT` with operations below in the hierarchy:

```

RunStateT (Ret a)    s = Ret (a, s)
RunStateT (m >>= f) s = RunStateT m s >>= fun x => RunStateT (f x.1) x.2
RunStateT Get       s = Ret (s, s)
RunStateT (Put s')  s = Ret (tt, s')

```

This is the methodology of packed classes that allows for the overloading of the notations `Ret` and `>>=` here. The notation `.1` (resp. `.2`) is for the first (resp. second) projection of a pair. The unique value of type `unit` is `tt`. The operations `Get` and `Put` are the standard operations of the state monad. Intuitively, given a monad `M` that inherits from the state monad, `Get` is a computation of type `M S` that returns the state and `Put` has type `S -> M unit` and updates the state (see Sect. 3 for a model of these operations).

The interface `exceptStateRunMonad` is the combination of the operations and equations of `stateRunMonad` and `exceptMonad` [9, Sect. 5] [24, file `hierarchy.v`] plus two additional equations on the combination of `RunStateT` with the operations of `exceptMonad`. Recall that the operations of the exception monad are the computations `Fail` of type `M A` and `Catch` of type `M A -> M A -> M A` for some type `A` (which happens to be the type of the state in this example); intuitively, `Fail` raises an exception while `Catch` handles it.

```

RunStateT Fail      s = Fail
RunStateT (Catch m1 m2) s = Catch (RunStateT m1 s) (RunStateT m2 s)

```

Using our formalization of monad transformers presented in this paper, it is then easy to build a model that validates those equations, whereas in previous work we had to build a model from scratch each time we were introducing a new combination of effects.

## 4.2 Example: The Fast Product

Now let us write a program and reason on it equationally. First, we write a recursive function that traverses a list of natural numbers to compute their product, but fails in case a 0 is met. Intermediate results are stored in the state:

```

Variables (N : exceptMonad) (M : exceptStateRunMonad nat N).
Fixpoint fastProductRec l : M unit :=
  match l with
  | [::] => Ret tt
  | 0 :: _ => Fail
  | n.+1 :: l' => Get >>= fun m => Put (m * n.+1) >> fastProductRec l'
  end.

```

Then, the main function will catch an eventual failure. If there is a failure, then the result is 0, else the result is the value stored in the state:

```

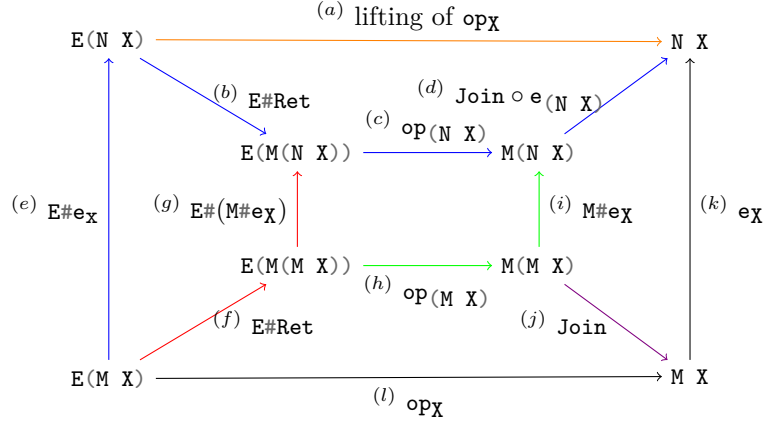
Variables (N : exceptMonad) (M : exceptStateRunMonad nat N).
Definition fastProduct l : M _ :=
  Catch (Put 1 >> fastProductRec l >> Get) (Ret 0 : M _).

```

To implement this algorithm in Haskell, we would use the state monad transformer applied to the exception monad. It would then be necessary to prefix each primitive of the exception monad with a lifting operation (`lift` or `mapStateT2` in Haskell). Here, we avoid this by using the hierarchy of interfaces, and the use of monad transformers is restricted to the construction of models for the interfaces.

The correctness states that the result of the fast product is always the same as a purely functional version:





■ **Figure 2** Proof of Uniform Algebraic Lifting (Theorem 1).

**Lemma** `fastProductCorrect 1 n : evalStateT (fastProduct 1) n = Ret (product 1)`.

where `evalStateT m s` is defined as `RunStateT m s >>= fun x => Ret x`. This proposition is proved easily with a 10 lines proof script that consists of an induction on `1`, rewriting with the equations in `exceptStateRunMonad` and application of standard arithmetic (see Appendix A.1).

Note that in this section we are dealing with the state monad transformer applied to the exception monad, and that the last equation in Sect. 4.1 specifies that the state is “backtracked”, i.e., if the state is modified in `m1` before an exception occurs, then this change is forgotten before `m2` is executed. This is usual in Haskell. The alternative semantics without backtracking would be closer to, say, OCaml, where the state is not be backtracked in case of an exception. Our program would behave the same way because it happens that the exception handler ignores the state. However, we would need to devise new equations to deal with `Fail` and `Catch`.

## 5 Application 2: Formalization of the Lifting of an Algebraic Operation

This section is an application of MONAE extended with the formalization of sigma-operations and of monad transformers of Sect. 3. We prove using equational reasoning a theorem about the lifting of algebraic operations along monad morphisms. This corresponds to the theorem that concludes the first of part of the original paper on modular monad transformers [15, Sect.2–4].

In the following `M` and `N` are two monads. Given an `E`-operation `op` for `M` and a monad morphism `e` from `M` to `N`, a *lifting* of `op` (to `N`) along `e` is an `E`-operation `op'` for `N` such that for all `X`:

$$e_X \circ op_X = op'_X \circ (E\#e_X).$$

► **Theorem 1** (Uniform Algebraic Lifting [15, Thm. 19]). *Given an algebraic `E`-operation `op` for `M` and a monad morphism `e` from `M` to `N`, let `op'` be*

$$X \mapsto Join_X^N \circ e_{(N X)} \circ op_{(N X)} \circ (E\#Ret_{(N X)}^M).$$

*Then `op'` is an algebraic `E`-operation for `N` and a lifting of `op` along `e`.*

**Proof.** The proof that `op'` is a lifting is depicted by the diagram of Fig. 2.

The first step is to show that the path (a) ( $\rightarrow$ ) and the path (b)-(c)-(d) ( $\rightarrow$ ) are equal, which is by definition of a lifting. The resulting goal is rendered in COQ as follows (for any `Y`):



$e \ X \ (\text{op } X \ Y) = \text{Join } (e \ (N \ X) \ (\text{op } (N \ X) \ ((E \ \# \ \text{Ret}) \ ((E \ \# \ e \ X) \ Y))))$

The second step of the proof is to show that the path (e)-(b) ( $\rightarrow$ ) and the path (f)-(g) ( $\rightarrow$ ) are equal, which is achieved by appealing to the functor laws and the naturality of `Ret`. More precisely, to prove

$(E \ \# \ \text{Ret}) \ ((E \ \# \ e \ X) \ Y) = (E \ \# \ (M \ \# \ e \ X)) \ ((E \ \# \ \text{Ret}) \ Y),$

it suffices to execute the following sequence of rewritings:

```
rewrite -[in LHS]compE -functor_o. (* functor composition law in the lhs *)
rewrite -[in RHS]compE -functor_o. (* functor composition law in the rhs *)
(* the goal is now: (E # (Ret \o e X)) Y = (E # (M # e X \o Ret)) Y *)
rewrite (natural RET).           (* naturality of ret *)
(* the goal is now: (E # (Ret \o e X)) Y = (E # (Ret \o FId # e X)) Y *)
by rewrite FIdf.                 (* property of the identity functor *)
```

The next step is to show that the path (g)-(c) and the path (h)-(i) ( $\rightarrow$ ) are equal; this is by naturality of `op`.

The next step is to show that the paths (i)-(d) and (j)-(k) are equal, which is by the bind law of monad morphisms and naturality of monad morphisms.

The last step (equality of the paths (f)-(h)-(j) and (l)) amounts to proving:

$\text{op } X \ Y = \text{Join } (\text{op } (M \ X) \ ((E \ \# \ \text{Ret}) \ Y)).$

This step depends of an intermediate lemma [15, Prop. 17]. Let us explain it because it introduces functions and we will use one of them again later in this paper. Given a natural transformation  $n : E \rightsquigarrow M$ , `psi` is an `E`-operation for `M` defined by the function  $X \mapsto \text{Join}_X \circ n$ . Given an `E`-operation for `M`, `phi` is a natural transformation  $E \rightsquigarrow M$  defined by the function  $X \mapsto \text{op}_X \circ (E\#\text{Ret})$ . It turns out that `psi` is algebraic and that `psi` cancels `phi` for algebraic operations (proofs omitted here, see [24]), which proves the last goal.

The second part of the proof is to prove that `op'` is algebraic. This is a direct consequence of the fact that `psi` is algebraic.

It should be noted that, even though the statement of the theorem defines the lifting as the composition of the functions `Join`, `e`, etc., it is actually much more practical from the view point of formal proof to define it as `psi (monadM_nt e \v phi op)`, i.e., the application of the function `psi` to the vertical composition of `e` and `phi op`, because this object (let us call it `alifting`) is endowed with the properties of algebraic operations, whose immediate availability facilitates the formal proof. ◀

The reader can observe in Appendix A.2 that the complete proof script for Theorem 1 essentially amounts to a small number of rewritings, as has been partially illustrated in the proof just above.

### Example: Lifting the get Operation along the Exception Monad Transformer

Let us assume the availability of a type `S` for states and of a type `Z` for exceptions. We consider `M` to be the state monad. To define the lifting of the `get` operation of `M` (more precisely its algebraic version seen in Sect. 3.2) along `exceptT` (Sect. 3.3) it suffices to call the `alifting` function with the right arguments:

```
Let M S : monad := ModelState.state S.
```

```
Definition aLGet {Z S} : (StateOps.Get.func S).-aoperation (exceptT Z (M S)) :=
  aLifting (get_aop S) (Lift (exceptT Z) (M S)).
```

By the typing, we see that the result `aLGet` is also an algebraic operation.

For example, we can check that the resulting sigma-operation is indeed the get operation of the transformed monad:

```
Goal forall Z (S : UU0) X (k : S -> exceptT Z (M S) X),
  aLGet _ k = StateOps.get_op _ k. by [].
```

## 6 Application 3: Formalization of the Lifting of Sigma-Operations

This section is an application of our formalization of sigma-operations and (functorial) monad transformers of Sect. 3 and also of Theorem 2. Using `MONAE`, we give an equational proof for a theorem that generalizes the lifting of Sect. 5 which was restricted to algebraic operations. This corresponds to the second part of the original paper on modular monad transformers [15, Sect. 5].

This application requires us to use a non-standard setting of `COQ`. Section 6.1 introduces a monad transformer whose formalization requires impredicativity. Section 6.2 focuses on the main technical difficulty that we identified when going from the pencil-and-paper proofs to a formalization using `COQ`: an innocuous-looking proof that actually calls for an argument based on parametricity. We conclude this section with the formal statement of [15, Thm. 27] and its formal proof (Sect. 6.3).

### 6.1 Impredicativity Setting for the Codensity Monad Transformer

To implement the lifting of an operation along a functorial monad transformer, Jaskelioff introduces a monad transformer `codensityT` related to the construction of the codensity monad for an endofunctor [15, Def. 23]. Its formalization requires impredicativity and if nothing is done, the standard setting of `COQ` would lead to *universe inconsistencies*.

Let us give a bit of background on impredicativity with `COQ`. The type theory of `COQ` is constrained by a hierarchy of universes `Set`, `Type1`, `Type2`, etc. The `COQ` language only provides the keywords `Set` and `Type`, the `COQ` system figures out the right indices for `Types`. Universes are not impredicative by default; yet, `COQ` has an option (`-impredicative-set`) that changes the logical theory by declaring the universe `Set` as impredicative. This option is useful in `COQ` to formalize System  $F/F\omega$ , their impredicative encodings of data types, and for extraction of programs in CPS style. It is known to be inconsistent with some standard axioms of classical mathematics [7, 31] but we do not rely on them here<sup>4</sup>. To keep a firm grip on the universes involved, we fix a few universes at the beginning of the formal development [24, file `ihierarchy.v`]:

```
Definition UU2 : Type := Type.
```

```
Definition UU1 : UU2 := Type.
```

```
Definition UU0 : UU1 := Set.
```

and only use them instead of `Set` or `Type` (so far we have been using `UU0` but it is really another name for the native `Set` universe).

<sup>4</sup> More precisely, the development we discuss in this paper [24, directory `impredicative_set`] uses together with impredicative `Set` only the standard axioms of functional extensionality and proof irrelevance, which are compatible.

Now that we have set `COQ` appropriately, we define the codensity monad transformer. Given a monad `M`, a computation of a value of type `A` in the monad `codensityT M` has type `forall (B : UU0), (A -> M B) -> M B` of type `UU0`: here, impredicativity comes into play. We abbreviate this type expression as `MK M A` in the following. We do not detail the formalization of `codensityT` because it follows the model of the exception monad transformer that we explained in Sect. 3.3. Let us just display its main ingredients, i.e., the unit, bind, and lift operations [15, Def. 23]:

```

Definition retK (A : UU0) (a : A) : MK M A :=
  fun (B : UU0) (k : A -> M B) => k a.
Definition bindK (A B : UU0) (m : MK M A) f : MK M B :=
  fun (C : UU0) (k : B -> M C) => m C (fun a : A => (f a) C k).
(* definition of codensityTmonadM omitted *)
Definition liftK (A : UU0) (m : M A) : codensityTmonadM A :=
  fun (B : UU0) (k : A -> M B) => m >>= k.

```

We can check in `COQ` that they indeed give rise to a monad transformer in the sense of Sect. 3.3, so that `codensityT` does have the type `monadT` (Sect. 3.3) of monad transformers.

## 6.2 Parametricity to Prove Naturality

The monad transformer `codensityT` is needed to state the theorem about the lifting of sigma-operations and in particular to define a natural transformation called `from` [15, Prop. 26]. Formally, we can define `from`'s components as follows (`M` is a monad):

```

Definition from_component : codensityT M ~-> M :=
  fun (A : UU0) (c : codensityT M A) => c A Ret.

```

At first sight, the naturality of `from_component` seems obvious and indeed no proof is given in the original paper on modular monad transformers (see the first of the two statements of [15, Prop. 26]). It is however a bit more subtle than it appears and, as a matter of fact, it is shown in a later paper that this claim is wrong: `fromM` cannot be a natural transformation in the setting of  $F\omega$  [17, p. 4452]. We explain how we save the day in `COQ` by relying on parametricity.

We state the naturality of `fromM` as `naturality (codensityT M) M from_component`. This goal reduces<sup>5</sup> to:

```
forall (m : codensityT M A) (h : A -> B), (M # h \o m A) Ret = m B (M # h \o Ret).
```

This last goal is an instance of a more general statement (recall from Sect. 6.1 that `MK M` is the action on the objects of the monad `codensityT M`):

```
forall (M : monad) (A : UU0) (m : MK M A) (A1 B : UU0) (h : A1 -> B),
  M # h \o m A1 = m B \o (fun f : A -> M A1 => (M # h) \o f).
```

This is actually a special case of naturality as one can observe by rewriting the type of `m` with the appropriate functors: `exponential_F A \0 M` and `M`, where `exponential_F` is the functor whose action on objects is `forall X : UU0, A -> X`:

```
forall (M : monad) (A : UU0) (m : MK M A), naturality (exponential_F A \0 M) M m
```

Unfortunately, we are not able to prove it in plain `COQ` (with or without impredicative `Set`), even if we consider particular functors `M` such as the identity functor.

<sup>5</sup> By functional extensionality, by naturality of `Ret`, and by definition of `from_component`.

## 2:14 Extending Equational Monadic Reasoning with Monad Transformers

The solution consists in assuming an axiom of parametricity for each functor  $M$  and derive naturality from it. That is, we follow the approach advocated by Wadler [34]. It has been shown to be sound in COQ [4, 5, 18, 19] and it is implemented by the PARAMCOQ plugin [18]. For instance, let us describe what happens when  $M$  is the list monad. First, we rewrite the naturality statement above in the case of the list functor (`map` is the map function of lists):

```
forall (X Y : UU0) (f : X -> Y) (g : A -> seq X),
  (map f \o m X) g = (m Y \o (exponential_F A \O M) # f) g.
```

The proof proceeds by induction on a proof-term of type

```
list_R X Y (fun x y => f x = y) (m X g) ((m Y \o (exponential_F A \O M) # f) g)
```

where `list_R X Y X_R 11 12` means that the elements of lists `11` and `12` are pairwise related by the relation `X_R`. The role of PARAMCOQ is to generate definitions (including `list_R`) for us to be able to produce this proof. Concretely, starting from `MK`, PARAMCOQ generates the logical relation `T_R` of type (it is obtained by induction on types [11]):

```
(forall X : UU0, (A -> list X) -> list X) ->
  (forall X : UU0, (A -> list X) -> list X) -> UU0
```

Here, `T_R m1 m2` expands to:

```
forall (X1 X2 : UU0) (RX : X1 -> X2 -> UU0)
  (f1 : A -> list X1) (f2 : A -> list X2),
  (forall a1 a2 : A, a1 = a2 -> list_R X1 X2 RX (f1 a1) (f2 a2)) ->
  list_R X1 X2 RX (m1 X1 f1) (m2 X2 f2)
```

It is then safe to assume the following parametricity axiom:

```
Axiom param : forall m : MK M A, T_R m m.
```

The application of `param` is the first step to produce the proof required for the induction:

```
have : list_R X Y (fun x y = f x = y) (m X g) ((m Y \o (exponential_F A \O M) # f) g).
  apply: param.
  (* ∀ a a', a = a' ->
    list_R X Y (fun x y = f x = y) (g a) (((exponential_F A \O M) # f) g a') *)
```

The goal generated is proved by induction on `g a` which is a list.

The same approach is applied to other monads (identity, exception, option, state) [24, file `iparametricity_codensity.v`].

### 6.3 Lifting of Sigma-operations: Formal Statement

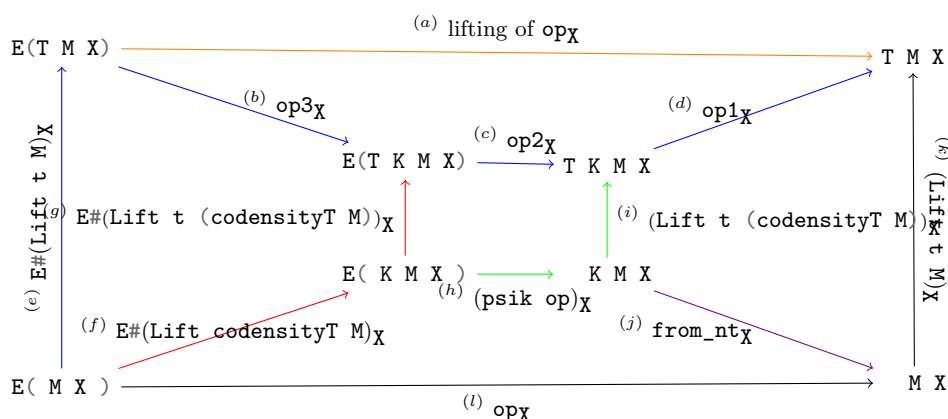
Before stating and proving the main theorem about lifting of sigma-operations, we formally define a special algebraic operation [15, Def. 25]. Let  $E$  be a functor,  $M$  be a monad, and `op` be an  $E$ -operation for  $M$ . The natural transformation `kappa` from  $E$  to `codensityT M` is defined by the components

$$A, (s : E A), B, (k : A \rightarrow M B) \mapsto \text{op } B ((E\#k) s)$$

and `psik` is the algebraic  $E$ -operation for the monad `codensityT M` defined by:

```
Definition psik : E.-aoperation (codensityT M) := psi (kappa op).
```

Recall that the function `psi` has been defined in the proof of Theorem 1.



■ **Figure 3** Proof of Uniform Lifting (Theorem 2).

► **Theorem 2** (Uniform Lifting [15, Thm. 27]). *Let  $M$  be a monad such that any computation  $m : MK M A$  is natural in the sense of Sect. 6.2 (hypothesis *naturality<sub>MK</sub>*). Let  $op$  be an  $E$ -operation for  $M$  and  $t$  be a functorial monad transformer. We denote:*

- by  $op1$  the term  $Hmap\ t$  (from *naturality<sub>MK</sub>*) (see Sect. 6.2 for *from*, *Hmap* was defined in Sect. 3.4),
- by  $op2$  the algebraic lifting along  $Lift\ t$  of  $(psik\ op)$  (see just above for *psik*), and
- by  $op3$  the term  $E\ \#\# \ Hmap\ t$  (monad <sub>$M$</sub> *\_nt* ( $Lift\ codensityT\ M$ )) (see Sect. 6.1 for *codensityT*).

Then the operation  $op1 \ \vee \ op2 \ \vee \ op3$  (where  $\vee$  is the vertical composition seen in Sect. 2) is a lifting of  $op$  along  $t$ .

**Proof.** The proof is depicted by the diagram in Fig. 3.

The first step of the proof is to unfold the definition of lifting (which amounts to showing that the paths (a) ( $\rightarrow$ ) and (b)-(c)-(d) are equal). Consequently, the proof goal is rendered in COQ as follows (for all  $X : \mathbb{U}0$ ):

$$Lift\ t\ M\ X \ \backslash o \ op\ X = (op1 \ \vee \ op2 \ \vee \ op3)\ X \ \backslash o \ E\ \#\ Lift\ t\ M\ X$$

The second step of the proof is to show that the path (e)-(b) and the path (f)-(g) ( $\rightarrow$ ) are equal, which is achieved by appealing to the law of functor composition and the naturality of *Hmap*.

The next step is to show that the path (g)-(c) and the path (h)-(i) ( $\rightarrow$ ) are equal; this is by applying Theorem 1.

At this point, the goal becomes:

$$Lift\ t\ M\ X \ \backslash o \ op\ X = (op1\ X \ \backslash o \ (Lift\ t\ (codensityT\ M)\ X \ \backslash o \ psik\ op\ X)) \ \backslash o \ E\ \#\ Lift\ codensityT\ M\ X$$

It happens that we can use the naturality of *Hmap* to make the *from* function appear in the right-hand side of the goal:

$$Lift\ t\ M\ X \ \backslash o \ op\ X = ((Lift\ t\ M\ X \ \backslash o \ from\ naturality\_{MK}\ X) \ \backslash o \ psik\ op\ X) \ \backslash o \ E\ \#\ Lift\ codensityT\ M\ X$$

The last step is to identify  $op$  with the composition of the *from* function, *psik op*, and  $E\ \#\ List\ codensityT\ M$ , which is the purpose of a lemma [15, Prop. 26] (see [24, file *ifmt\_lifting.v*, lemma *psikE*]). ◀

The proof script corresponding to the proof above is reproduced in Appendix A.3.

Finally, we show that, for all the monad transformers considered in this paper, the lifting of an algebraic operation provided by Theorem 2 coincides with the one provided by Theorem 1. This corresponds to the last results about modular monad transformers [15, Prop. 28].

## 7 Related Work

The example we detail in Sect. 4 adds to several examples of monadic equational reasoning [8, 9, 25–28]. Its originality is to use a parameterized interface and the `RunStateT` command, which are typical of programs written using monad transformers.

Huffman formalizes three monad transformers in the Isabelle/HOL proof assistant [12]. This experiment is part of a larger effort to overcome the limitations of Isabelle/HOL type classes to reason about Haskell programs that use (Haskell) type classes. Compared to Isabelle/HOL, the type system of COQ is more expressive so that we could formalize a much larger theory, even relying on extra features of COQ such as impredicativity and parametricity to do so.

Maillard proposes a meta language to define monad transformers in the COQ proof assistant [21, Chapter 4]. It is an instance implementation of one element of a larger framework to verify programs with monadic effects using Dijkstra monads [22]. The lifting of operations is one topic of this framework but it does not go as far as the deep analysis of Jaskelioff [14, 15, 17].

There are also formalizations of monads and their morphisms that focus on the mathematical aspects, e.g., UniMath [33]. However, the link to the monad transformers of functional programming is not done.

Monad transformers is one approach to combine effects. Algebraic effects is a recent alternative. It turns out that the two are related [30] and we have started to extend MONAE to clarify formally this relation.

## 8 Conclusions and Future Work

In this paper, we extended MONAE, a formalization of monadic equational reasoning, with monad transformers. We explained how it helps us to better organize the models of monads, thanks to sigma-operations in particular. We also explained how to extend the hierarchy of monad interfaces to handle programs written with monad transformers in mind. We also used our formalization of monad transformers to formalize the theory of liftings of modular monad transformers [15] using equational reasoning. For that purpose, we needed to fix the original presentation by using COQ’s impredicativity and parametricity.

The main result of this paper is a robust, formal theory of monad transformers. We plan to extend the hierarchy of monad interfaces of MONAE similarly to how we proceeded for `exceptStateRunMonad`. Such an extension will call for more models to be formalized and we expect our formalized theory of liftings to be useful on this occasion.

Results up to Sect. 5 hold whether or not `Set` is impredicative. In contrast, the setting of Sect. 6 conflicts with MONAE programs relying on some data structures from the `MATHCOMP` library [23] (such as fixed-size lists) or from the `INFOTHEO` library [13] (such as probability distributions) because these data structures are in `Type` and cannot be computed with monads in `Set`. One could think about reimplementing them but this is a substantial amount of work. A cheap way to preserve these data structures together with the theorem on lifting of sigma-operations is to disable universe checking as soon as this theorem is used; this way,

monads can stay in **Type**. Disabling universe checking is not ideal because it is unsound in general<sup>6</sup>; note however that this is sometimes used for the formalization of category-theoretic notions (e.g., [3, Sect. 6]). How to improve this situation is another direction for future work.

---

## References

- 1 Reynald Affeldt, Jacques Garrigue, David Nowak, and Takafumi Saikawa. A trustful monad for axiomatic reasoning with probability and nondeterminism. arXiv cs.LO 2003.09993, 2020. [arXiv:2003.09993](https://arxiv.org/abs/2003.09993).
- 2 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019*, volume 11825 of *Lecture Notes in Computer Science*, pages 226–254. Springer, 2019.
- 3 Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. Categorical structures for type theory in univalent foundations. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017), August 20–24, 2017, Stockholm, Sweden*, volume 82 of *LIPICs*, pages 8:1–8:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- 4 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014), San Diego, CA, USA, January 20–21, 2014*, pages 503–516. ACM, 2014.
- 5 Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012), Dubrovnik, Croatia, June 25–28, 2012*, pages 135–144. IEEE Computer Society, 2012.
- 6 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.
- 7 Herman Geuvers. Inconsistency of classical logic in type theory. Short note, December 2001. URL: <http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz>.
- 8 Jeremy Gibbons. Unifying theories of programming with monads. In *4th International Symposium on Unifying Theories of Programming (UTP 2012), Paris, France, August 27–28, 2012*, volume 7681 of *Lecture Notes in Computer Science*, pages 23–67. Springer, 2012.
- 9 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM, 2011.
- 10 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008. Version 17 (Nov 2016). Now part of [32].
- 11 Jean Goubault-Larrecq, Slawomir Lasota, and David Nowak. Logical relations for monadic types. *Math. Struct. Comput. Sci.*, 18(6):1169–1217, 2008.
- 12 Brian Huffman. Formal verification of monad transformers. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’12, Copenhagen, Denmark, September 9–15, 2012*, pages 15–16. ACM, 2012.
- 13 Infotheo. A Coq formalization of information theory and linear error-correcting codes. Coq scripts. Last stable release: 0.3, 2021. URL: <https://github.com/affeldt-aist/infotheo/>.
- 14 Mauro Jaskieloff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

---

<sup>6</sup> One can derive **False** by applying a variant of Hurkens paradox (see <https://coq.inria.fr/library/Coq.Logic.Hurkens.html>).



- 15 Mauro Jaskelioff. Modular monad transformers. In *18th European Symposium on Programming (ESOP 2009), York, UK, March 22–29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2009.
- 16 Mauro Jaskelioff. Private communication, May 2020.
- 17 Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, 2010.
- 18 Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. In *21st Annual Conference of the EACSL on Computer Science Logic (CSL 2012), September 3–6, 2012, Fontainebleau, France*, volume 16 of *LIPICs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
- 19 Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic 2013 (CSL 2013), September 2–5, 2013, Torino, Italy*, volume 23 of *LIPICs*, pages 432–451. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.
- 20 Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995), San Francisco, California, USA, January 23–25, 1995*, pages 333–343. ACM Press, 1995.
- 21 Kenji Maillard. *Principes de la Vérification de Programmes à Effets Monadiques Arbitraires*. PhD thesis, Université PSL, November 2019.
- 22 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP):104:1–104:29, 2019.
- 23 Mathematical Components Team. Mathematical Components library, 2007. Last stable version 1.11 (2020). URL: <https://github.com/math-comp/math-comp>.
- 24 Monae. Monadic effects and equational reasoning in Coq. Coq scripts. Last stable release: 0.3, 2021. URL: <https://github.com/affeldt-aist/monae/>.
- 25 Shin-Cheng Mu. Calculating a backtracking algorithm: An exercise in monadic program derivation. Technical Report TR-IIS-19-003, Institute of Information Science, Academia Sinica, June 2019.
- 26 Shin-Cheng Mu. Equational reasoning for non-deterministic monad: A case study of Spark aggregation. Technical Report TR-IIS-19-002, Institute of Information Science, Academia Sinica, June 2019.
- 27 Shin-Cheng Mu and Tsung-Ju Chiang. Declarative pearl: Deriving monadic quicksort. In Keisuke Nakano and Konstantinos Sagonas, editors, *15th International Symposium on Functional and Logic Programming (FLOPS 2020), Akita, Japan, September 14–16, 2020*, volume 12073 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2020.
- 28 Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. Handling local state with global state. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019*, volume 11825 of *Lecture Notes in Computer Science*, pages 18–44. Springer, 2019.
- 29 Célestine Sauvage, Reynald Affeldt, and David Nowak. Vers la formalisation en Coq des transformateurs de monades modulaires. In *Trente-et-unièmes Journées Francophones des Langages Applicatifs (JFLA 2020), Janvier 2020, Gruissan, France*, pages 23–30, 2020. In French. URL: <https://hal.archives-ouvertes.fr/hal-02434736v2/document>.
- 30 Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In *12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), Berlin, Germany, August 18–23, 2019*, pages 98–113. ACM, 2019.
- 31 The Coq Development Team. Impredicative set, 2019. Last revision: 2019-07-12. URL: <https://github.com/coq/coq/wiki/Impredicative-Set>.
- 32 The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2021. Version 8.13.0. URL: <https://coq.inria.fr/distrib/current/refman/>.



- 33 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath – a computer-checked library of univalent mathematics. Available at <https://github.com/UniMath/UniMath>.
- 34 Philip Wadler. Theorems for free! In *4th international conference on Functional programming languages and computer architecture (FPCA 1989)*, London, UK, September 11–13, 1989, pages 347–359. ACM, 1989.

## A Proof Scripts for the Three Applications of This Paper

The following proof scripts have been copied verbatim from MONAE [24] for the reader’s convenience. We claim that these proof scripts are readable and short in the sense that each line corresponds to a genuine proof step and that there are few administrative tactics hampering reading. The terseness of the SSREFLECT tactic language could actually make these proof scripts much shorter but that is not our point here. In particular, we make explicit the proof steps of Theorems 1 and 2 (in Appendices A.2 and A.3) with `transitivity` steps or explicit `rewrite` steps followed by indented (sub-)proof scripts.

### A.1 Proof Script for the Correctness of `fastProduct`

The following proof script can be found in [24, file `example_transformer.v`].

```
Lemma fastProductCorrect l n :
  evalStateT (fastProduct l) n = Ret (product l).
Proof.
rewrite /fastProduct -(mul1n (product _)); move: 1.
elim: l => [ | [ | x] l ih] m.
- rewrite muln1 bindA bindretf putget.
  rewrite /evalStateT RunStateTCatch RunStateTBind RunStateTPut bindretf.
  by rewrite RunStateTRet RunStateTRet catchret bindretf.
- rewrite muln0.
  rewrite /evalStateT RunStateTCatch RunStateTBind RunStateTBind RunStateTPut.
  by rewrite bindretf RunStateTFail bindfailf catchfailm RunStateTRet bindretf.
- rewrite [fastProductRec _]/=.
  by rewrite -bindA putget bindA bindA bindretf -bindA -bindA putput ih mulnA.
Qed.
```

### A.2 Proof Script for Theorem 1 [15, Thm. 19]

The following proof script can be found in [24, file `monad_transformer.v`].

```
Section uniform_algebraic_lifting.
Variables (E : functor) (M : monad) (op : E.-aoperation M).
Variables (N : monad) (e : monadM M N).

Definition alifting : E.-aoperation N := psi (monadM_nt e \v phi op).

Lemma aliftingE :
  alifting = (fun X => Join \o e (N X) \o phi op (N X)) :> (_ ~~> _).
Proof. by []. Qed.

Theorem uniform_algebraic_lifting : lifting op e alifting.
Proof.
move=> X.
apply fun_ext => Y.
```

```

rewrite /alifting !compE psiE vcompE phiE !compE.
rewrite (_ : (E # Ret) ((E # e X) Y) =
      (E # (M # e X)) ((E # Ret) Y)); last first.
  rewrite -[in LHS]compE -functor_o.
  rewrite -[in RHS]compE -functor_o.
  rewrite (natural RET).
  by rewrite FIdf.
rewrite (_ : op (N X) ((E # (M # e X)) ((E # Ret) Y)) =
      (M # e X) (op (M X) ((E # Ret) Y))); last first.
  rewrite -(compE (M # e X)).
  by rewrite (natural op).
transitivity (e X (Join (op (M X) ((E # Ret) Y)))); last first.
  rewrite joinE monadMbind.
  rewrite bindE -(compE _ (M # e X)).
  by rewrite -natural.
by rewrite -[in LHS](phiK op).
Qed.
End uniform_algebraic_lifting.

```

### A.3 Proof Script for Theorem 2 [15, Thm. 27]

The following proof script can be found in [24, file ifmt\_lifting.v].

```

Section uniform_sigma_lifting.
Variables (E : functor) (M : monad) (op : E.-operation M) (t : FMT).
Hypothesis naturality_MK : forall (A : UU0) (m : MK M A),
  naturality_MK m.

Let op1 : t (codensityT M) ~> t M := Hmap t (from naturality_MK).
Let op2 := alifting (psik op) (Lift t _).
Let op3 : E \0 t M ~> E \0 t (codensityT M) :=
  E ## Hmap t (monadM_nt (Lift codensityT M)).

Definition slifting : E.-operation (t M) := op1 \v op2 \v op3.

Theorem uniform_sigma_lifting : lifting_monadT op slifting.
Proof.
rewrite /lifting_monadT /slifting => X.
apply/esym.
transitivity ((op1 \v op2) X \o op3 X \o E # Lift t M X).
  by rewrite (vassoc op1).
rewrite -compA.
transitivity ((op1 \v op2) X \o
  ((E # Lift t (codensityT M) X) \o (E # Lift codensityT M X))).
  congr (_ \o _); rewrite /op3.
  by rewrite -functor_o -natural_hmap functor_o functor_app_naturalE.
transitivity (op1 X \o
  (op2 X \o E # Lift t (codensityT M) X) \o E # Lift codensityT M X).
  by rewrite vcompE -compA.
rewrite -uniform_algebraic_lifting.
transitivity (Lift t M X \o from naturality_MK X \o (psik op) X \o
  E # Lift codensityT M X).
  congr (_ \o _).
  by rewrite compA natural_hmap.
rewrite -2!compA.

```

```
congr (_ \o _).  
by rewrite compA -psikE.  
Qed.  
End uniform_sigma_lifting.
```



# Towards a Certified Reference Monitor of the Android 10 Permission System

Guido De Luca ✉

Universidad Nacional de Rosario, Argentina

Carlos Luna ✉🏠

InCo, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay

---

## Abstract

---

Android is a platform for mobile devices that captures more than 85% of the total market share [14]. Currently, mobile devices allow people to develop multiple tasks in different areas. Regrettably, the benefits of using mobile devices are counteracted by increasing security risks. The important and critical role of these systems makes them a prime target for formal verification. In our previous work [10], we exhibited a formal specification of an idealized formulation of the permission model of version 6 of Android. In this paper we present an enhanced version of the model in the proof assistant Coq, including the most relevant changes concerning the permission system introduced in versions Nougat, Oreo, Pie and 10. The properties that we had proved earlier for the security model have been either revalidated or refuted, and new ones have been formulated and proved. Additionally, we make observations on the security of the most recent versions of Android. Using the programming language of Coq we have developed a functional implementation of a reference validation mechanism and certified its correctness. The formal development is about 23k LOC of Coq, including proofs.

**2012 ACM Subject Classification** Security and privacy → Systems security; Theory of computation → Proof theory

**Keywords and phrases** Android, Permission model, Formal idealized model, Reference monitor, Formal proofs, Certified implementation, Coq

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.3

**Related Version** *Preliminary Version*: <https://arxiv.org/abs/2011.00720> [20]

**Supplementary Material** *Software (Source Code)*:

<https://github.com/g-deluca/android-coq-model>

archived at `swh:1:dir:a190369ad1ad079f2c9acc1470b0de19762520ae`

## 1 Introduction

Android [24] is the most used mobile OS in the world, capturing approximately 85% of the total market-share [14]. It offers a huge variety of applications in its official store that aim to help people in their daily activities, many of them critical in terms of privacy. In order to guarantee their users the security they expect, Android relies on a *multi-party* consensus system where user, OS and application must be all in favour of performing a task. This security framework is built upon a system of permissions, which are basically tags that developers declare on their applications to gain access to sensitive resources. Whenever an action that requires some of this permissions is executed for the first time, the user will be asked for authorization and if provided, the OS will ensure that only the required access is granted. The important and critical role of this security mechanism makes it a prime target for (formal) verification.

Security models play an important role in the design and evaluation of security mechanisms of systems. Earlier, their importance was already pointed in the Anderson report [1], where the concept of *reference monitor* was first introduced. This concept defines the design requirements for implementing what is called a *reference validation mechanism*, which shall



© Guido De Luca and Carlos Luna;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 3; pp. 3:1–3:18

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be responsible for enforcing the access control policy of a system. For ensuring the correct working of this mechanism three design requirements are specified: i) the reference validation mechanism (RVM) must always be invoked (*complete mediation*); ii) the RVM must always be tamper-proof (*tamper-proof*); and iii) the RVM must be small enough to be subject to analysis and tests, the completeness of which can be assured (*verifiable*).

The work presented here is concerned with the verifiability requirement. In particular we put forward an approach where formal analysis and verification of properties is performed on an idealized model that abstracts away the specifics of any particular implementation, and yet provides a realistic setting in which to explore the issues that pertain to the realm of (critical) security mechanisms of Android. The formal specification of the reference monitor shall be used to establish and prove that the security properties that constitute the intended access control policy are satisfied by the modeled behavior of the validation mechanisms.

**Contributions.** In our previous work [10] we presented a formal specification of an idealized formulation of the permission model of version 6 of Android. We also developed, using the programming language of Coq [27], an executable (functional) specification of the reference validation mechanism and we proved its correctness conforming to the specified model. Lastly, we used the program extraction mechanism provided by Coq [18] to derive a certified Haskell implementation of the reference validation mechanism. Here we present an enhanced version of the model, including the most relevant changes concerning the permission system introduced in versions Nougat, Oreo, Pie and 10. Some of these changes don't have a direct impact on our abstract model. In those cases, an informal analysis is included. The executable specification was also updated, and with that, the derived implementation as well. The properties that we had proved for the security model have been either revalidated or refuted, and new ones have been formulated and proved. The formal development is about 23k LOC of Coq, including various lemmas and their proofs.

**Organization of the paper.** Section 2 reviews the security mechanisms of Android and briefly describes the changes introduced in the later versions. Sections 3 and 4 present the formal axiomatic specification and the semantics of the certified implementation, respectively. Both sections discuss relevant properties concerning the new features. Section 5 considers related work and finally, Section 6 concludes with a summary of our contributions and directions for future work. The full formalization is available at <https://github.com/g-deluca/android-coq-model> [19] and can be verified using the Coq proof assistant. A preliminary version of this paper is accessible on arXiv [20].

## 2 Android's security model

### 2.1 Basic security mechanisms

The Android security model is primarily based on a sandbox and permission mechanism. Each application runs in a private virtual machine with a unique ID assigned to it, which means that one application's code is isolated from the code of the rest. This isolation means that, by default, applications can not interact with each other and have limited access to the OS. For example, if an application tries to do something malicious, like reading the user's contacts without permission, the action will fail due to the lack of privileges. However, these actions could also be started by trusted applications, and therefore, need to be done. Android's permission system is the mechanism in charge of deciding which of these actions should occur and which ones should not, depending on the permissions that each application has.

Every permission is identified by a unique name/text, has a protection level and may belong to a permission group. Furthermore, permissions can be classified into two groups: the ones defined by an application, for the sake of self-protection; and those predefined by Android, which are required to gain access to certain system features, like internet or location. Depending on the protection level of the permission, the system defines the rules to grant that permission. There are three classes of permission levels [4]: i) *normal*, these permissions can be automatically granted since they cover data or resources where there's very little risk to the user's privacy or the operation of other apps; ii) *dangerous*, permissions of this level provide access to data or resources that may be sensitive or could potentially affect the operation of other applications, and explicit user approval is needed to be granted; and iii) *signature*, a permission of this level is granted only if the application that requires it and the application that defined it are both signed with the same certificate. An application must declare—in an XML file called `AndroidManifest`—the set of permissions it needs to acquire further capacities than the default ones. From version 6 of Android, *dangerous* permissions are granted at runtime whereas both *normal* and *signature* are given when the application is installed.

Permissions may belong to groups that reunite a device's capabilities. The main purpose of grouping permissions in this way is to handle permission requests at the group level, in order to avoid overwhelming the user with too many questions. For example, the SMS group includes the permission needed to read text messages as well as the one needed to receive them (both considered to be *dangerous*). Whenever an application needs one of those for the first time, the user will be asked to authorize the whole group. In Section 2.2, we explain what *authorizing a group* means depending on the platform version.

An Android application is built up from *components*. A component is a basic unit that provides a particular functionality and that can be run by any other application with the right permissions. There exist four types of components [2]: i) *activity*, which is essentially a user interface of the application; ii) *service*, a component that executes in the background without providing an interface to the user; iii) *content provider*, a component intended to share information among applications; and iv) *broadcast receiver*, a component whose objective is to receive messages, sent either by the system or an application, and trigger the corresponding actions. The communication between components is achieved with the exchange of special messages called *intents*, which can be either i) *explicit*, meaning that the target application is specified; or ii) *implicit*, where only the action to be performed is declared and the system determines which application will run the task (if there is more than one capable application, the user is allowed to choose). In order to be candidates for the resolution of implicit intents, an application must declare on their manifest an *intent filter* that indicates the types of intents it can respond to.

Android provides two mechanisms by which an application can delegate its own permissions to another one. These mechanisms are called *pending intents* and *URI permissions*. An intent may be defined by a developer to perform a particular action. A `PendingIntent` specifies a reference to an action, which might be used by another application to perform the operation with the same permissions and identity of the one that created the intent. The *URI permissions* mechanism can be used by an application that has read/write access to a *content provider* to temporarily delegate those permissions to another application. These permissions are revoked once the receiver activity or service becomes inactive.

## 2.2 A brief review on the changelog

As we described in our previous work [10], the sixth version of Android introduced an important change to the system, allowing the users to handle permissions at runtime. In this section, we give a short account of the changes introduced between Android Nougat (7) and Android 10, that had a significant impact on the permission system.

### Filesystem

In order to improve security, the private directory of applications targeting<sup>1</sup> Android 7.0 or higher has restricted access: only the owner is capable of reading, writing or executing files stored in it. This configuration prevents leakage of metadata of private files, like the size or existence. With this change, applications are no longer able to share files simply by changing the file permissions and sharing their private URI; a content provider must be used in order to generate a reference to the file. With this approach, a new kind of URI is generated, which grants a temporary permission that will be available for the receiver activity or service only while they are active/running.

Our previous model already allowed granting temporary permissions to content providers URIs, so no change was required to formalize this new feature.

### Grouped permissions

Prior to Android 8, if an application requested a grouped permission at runtime and the user authorized it, the system also granted the rest of the permissions from the same group that were declared on the manifest. This behaviour was incorrect since it violated the intended least privilege security policy claimed by the designers of the platform. For applications targeting Android 8 or higher, this action was corrected and only the requested permission is granted. However, once the user authorized a group, all subsequent requests for permissions in that group are automatically granted. This change was added to the model.

**Normal grouped permissions.** According to Android’s official documentation, *any permission can belong to a permission group regardless of protection level* [3]. However, it is not specified if normal and dangerous permissions can share a group or, in case that it is possible, how the system should handle this situation. A few questions we have raised are the following: i) Is the authorization to automatically concede permissions from that group granted at installation time together with the normal permissions?; ii) Is the user warned about that decision?; iii) If that is the case, then there’s a contradiction with the documentation, since it claims that *a permission’s group only affects the user experience if the permission is dangerous*; and iv) If it’s not, how does the system avoid that dangerous permissions from the same group are not automatically granted later by the system?

In this work we formalized a worst-case scenario (that still suits the informal specification given by the authors of the platform), where a normal permission enables the automatic granting of dangerous permissions belonging to the same group. We formally discuss this situation in Section 3.4.

---

<sup>1</sup> Applications can *target* a particular version of the system. Android uses this setting to determine whether to enable any compatibility behaviors or features.



### Privacy changes

Android Pie (9) introduced several changes aiming to enhance users' privacy, such as limiting background apps' access to device sensors, restricting information retrieved from Wi-Fi scans, and adding new permission groups and rules to reorganize phone calls and phone state related permissions. Later, the tenth version of the platform continued adding limitations to services: a new permission for accessing the location in the background was added. Furthermore, Android 10 placed restrictions on when a service can start an activity, in order to minimize interruptions for the user and keep the user more in control of what is shown on their screen.

These changes are specific to the implementation, meaning that they have no impact on an abstract representation like ours.

### Permission check on legacy apps

Applications that target Android 5.1 or lower are considered to be old<sup>2</sup>. If an *old* application runs on an Android 10 system for the first time, a prompt appears on the screen, giving the user an opportunity to revoke access to permissions that the system previously granted at install time. This feature has been added to our model.

## 3 Formalization of Android's permission system

In this section we describe the axiomatic semantics of our model of the system, focusing on the features introduced in the later versions. We also discuss some of the verified properties.

**Formal language used.** Coq is an interactive theorem prover based on higher order logic that allows to write formal specifications and interactively generate machine-checked proofs of theorems. It also provides a (dependently typed) functional programming language that can be used to write executable algorithms. The Coq environment also provides program extraction towards languages like Ocaml and Haskell for execution of (certified) algorithms [17, 18]. In this work, enumerated types and sum types are defined using Haskell-like notation; for example,  $option\ T \stackrel{\text{def}}{=} None \mid Some\ (t : T)$ . Record types are of the form  $\{l_1 : T_1, \dots, l_n : T_n\}$ , whereas their elements are of the form  $\{t_1, \dots, t_n\}$ . Field selection is written as  $r.l_i$ . We also use  $\{T\}$  to denote the set of elements of type  $T$ . Finally, the symbol  $\times$  defines tuples, and  $nat$  is the datatype of natural numbers. We omit Coq code for reasons of clarity; this code is available in [19].

### 3.1 Model states

The Android security model we have developed has been formalized as an abstract state machine. In this model, states (AndroidST) are modelled as 13-tuples that store dynamic data about the system such as the installed applications and their current permissions, as well as static data like the declared manifest of each installed app. A complete formal definition is given in Figure 1.

The type `PerMId` represents the set of permissions identifiers; `PermGroup`, the set of permission groups identifiers; `Comp`, the application components whose code will run on the system; `Appld` represents the set of application identifiers; `iComp` is the set of identifiers of running instances of application components; `ContProv` is a subset of `Comp`, a special type

<sup>2</sup> We can also refer to them as *legacy* applications.

<b>Auxiliary definitions</b>	
OpTy	::= <i>read</i>   <i>write</i>   <i>rw</i>
PermLvl	::= <i>Dangerous</i>   <i>Normal</i>   <i>Signature</i>   <i>Signature/System</i>
Perm	::= PermId × <i>option</i> PermGroup × PermLvl
Complnstance	::= iComp × Comp
Manifest	::= {Comp} × <i>option nat</i> × <i>option nat</i> × {PermId} × {Perm} × <i>option</i> PermId
<b>State components</b>	
InstApps	::= {Appld}
VerifiedApps	::= {Appld}
AppPS	::= {Appld × PermId}
PermsGr	::= {Appld × PermGroup}
ComplnsRun	::= {Complnstance}
DelPPerms	::= {Appld × ContProv × Uri × OpTy}
DelTPerms	::= {iComp × ContProv × Uri × OpTy}
ARVS	::= {Appld × Res × Val}
Intents	::= {iComp × Intent}
Manifests	::= {Appld × Manifest}
Certs	::= {Appld × Cert}
AppDefPS	::= {Appld × Perm}
SysImage	::= {SysImgApp}
AndroidST	::= InstApps × VerifiedApps × PermsGr × AppPS × ComplnsRun × DelPPerms × DelTPerms × ARVS × Intents × Manifests × Certs × AppDefPS × SysImage

■ **Figure 1** Android state.

of component that allows sharing resources among different applications; a member of the type *Uri* is a particular URI (uniform resources identifier); the type *Res* represents the set of resources an application can have (through its *content providers*, members of *ContProv*); the type *Val* is the set of possible values that can be written on resources; an intent –i.e. a member of type *Intent*– represents the intention of a running component instance to start or communicate with other applications; a member of *SysImgApp* is a special kind of application which is deployed along with the OS itself and has certain privileges, like being impossible to uninstall.

The first component of the state records the identifiers (*Appld*) of the applications installed by the user. The second component is a subset of the first one, and represents those applications that are considered to be old but have already been verified, also by the user. The third component keeps track of the permissions granted to every application present in the system, including the ones preinstalled on the system. Similarly, the next component holds the information about what permission groups have already been authorized by the user on each app. The fifth component of the state stores the set of running component instances (*Complnstance*), while the components *DelPPerms* and *DelTPerms* store the information concerning permanent and temporary permissions delegations, respectively<sup>3</sup>. The eighth and ninth components of the state store respectively the values (*Val*) of resources (*Res*) of applications and the set of intents (*Intent*) sent by running instances of components (*iComp*) not yet processed. The four last components of the state record information that represents the manifests of the applications installed by the user, the certificates (*Cert*) with which they were signed and the set of permissions they define. The last component of the state stores the set of (native) applications installed in the Android system image, information that is relevant when granting permissions of level *Signature/System*.

A manifest (*Manifest*) is modelled as a 6-tuple whose members respectively declare application components (set of components, of type *Comp*, included in the application); optionally, the minimum version of the Android SDK required to run the application;

<sup>3</sup> A permanent delegated permission represents that an app has delegated permission to perform an operation on the resource identified by an URI. A temporary delegated permission refers to a permission that has been delegated to a component instance.

optionally, the version of the Android SDK targeted on development; the set of permissions it may need to run at its maximum capability; the set of permissions it declares; and the permission required to interact with its components, if any. Application components are all denoted by a component identifier. A content provider (`ContProv`), in addition, encompasses a mapping to the managed resources from the URIs assigned to them for external access. While the components constitute the static building blocks of an application, all runtime operations are initiated by component instances, which are represented in our model as members of an abstract type.

**Valid states.** The states defined in this way include some cases that are not relevant with the model we are trying to analyze. For example, we don't want a state where a preinstalled application and one installed by the user have the same identifier. In order to prevent this inconsistencies, we define a notion of valid state that captures several well-formedness conditions. It is formally defined as a predicate `valid_state` on the elements of type `AndroidST`. This predicate holds on a state  $s$  if the following conditions are met:

- all the components both in installed applications and in system applications have different identifiers;
- no component belongs to two different applications present in the device;
- no running component is an instance of a content provider;
- every temporally delegated permission has been granted to a currently running component and over a content provider present in the system;
- every running component belongs to an application present in the system;
- every application that sets a value for a resource is present in the system;
- the domains of the partial functions `Manifests`, `Certs` and `AppDefPS` are exactly the identifiers of the user-installed applications;
- the domains of the partial functions `AppPS` and `PermsGr` are exactly the identifiers of the applications in the system, both those installed by the users and the system applications;
- every installed application has an identifier different from those of the system applications, whose identifiers differ as well;
- all the permissions defined by applications have different identifiers;
- every partial function is indeed a function, that is, their domains don't have repeated elements;
- every individually granted permission is present in the system; and
- all the sent intents have different identifiers.

All these safety properties have a straightforward interpretation in our model. The full formal definition of the predicate is available in [19].

### 3.2 Action semantics

We modelled the different functionalities provided by the Android security system as a set of actions (of type `Action`) that determine how the system is able to transition from one state to another. Table 1 summarises the actions specified in our previous model that remained mostly the same since the new features didn't affect them whereas Table 2 groups those that are new or that suffered a big semantic change.

The behaviour of each action is specified in terms of a precondition ( $Pre : \text{AndroidST} \rightarrow \text{Action} \rightarrow Prop$ ) and a postcondition ( $Post : \text{AndroidST} \rightarrow \text{Action} \rightarrow \text{AndroidST} \rightarrow Prop$ ).

■ **Table 1** Legacy actions.

<code>install app m c lRes</code>	Install application with id <i>app</i> , whose manifest is <i>m</i> , is signed with certificate <i>c</i> and its resources list is <i>lRes</i> .
<code>uninstall app</code>	Uninstall the application with id <i>app</i> .
<code>read ic cp u</code>	The running comp. <i>ic</i> reads the resource corresponding to URI <i>u</i> from content provider <i>cp</i> .
<code>write ic cp u val</code>	The running comp. <i>ic</i> writes value <i>val</i> on the resource corresponding to URI <i>u</i> from content provider <i>cp</i> .
<code>startActivity i ic</code>	The running comp. <i>ic</i> asks to start an activity specified by the intent <i>i</i> .
<code>startActivityRes i n ic</code>	The running comp. <i>ic</i> asks to start an activity specified by the intent <i>i</i> , and expects as return a token <i>n</i> .
<code>startService i ic</code>	The running comp. <i>ic</i> asks to start a service specified by the intent <i>i</i> .
<code>sendBroadcast i ic p</code>	The running comp. <i>ic</i> sends the intent <i>i</i> as broadcast, specifying that only those components who have the permission <i>p</i> can receive it.
<code>sendOrdBroadcast i ic p</code>	The running comp. <i>ic</i> sends the intent <i>i</i> as an ordered broadcast, specifying that only those components who have the permission <i>p</i> can receive it.
<code>sendSBroadcast i ic</code>	The running comp. <i>ic</i> sends the intent <i>i</i> as a sticky broadcast.
<code>resolveIntent i app</code>	Application <i>app</i> makes the intent <i>i</i> explicit.
<code>stop ic</code>	The running comp. <i>ic</i> finishes its execution.
<code>grantP ic cp app u pt</code>	The running comp. <i>ic</i> delegates permanent permissions to application <i>app</i> . This delegation enables <i>app</i> to perform operation <i>pt</i> on the resource assigned to URI <i>u</i> from content provider <i>cp</i> .
<code>revokeDel ic cp u pt</code>	The running comp. <i>ic</i> revokes delegated permissions on URI <i>u</i> from content provider <i>cp</i> to perform operation <i>pt</i> .
<code>call ic sac</code>	The running comp. <i>ic</i> makes the API call <i>sac</i> .

■ **Table 2** New or modified actions.

<code>grant p app</code>	Grant the permission <i>p</i> to the application <i>app</i> with user confirmation.
<code>grantAuto p app</code>	Grant automatically the permission <i>p</i> to the application <i>app</i> (without user confirmation).
<code>revoke p app</code>	Remove an ungrouped permission <i>p</i> from the application <i>app</i> .
<code>revokePermGroup g app</code>	Remove the every permission of group <i>g</i> from the application <i>app</i> .
<code>hasPermission p app</code>	Check if the application <i>app</i> has the permission <i>p</i> .
<code>receiveIntent i ic app</code>	Application <i>app</i> receives the intent <i>i</i> , sent by the running comp. <i>ic</i> .
<code>verifyOldApp app</code>	Application <i>app</i> granted permissions are verified by the user

For instance, the axiomatic semantics of the new feature about automatic granting of permissions `grantAuto` is given by:

$$\begin{aligned}
Pre(s, \mathbf{grantAuto} \ p \ app) &\stackrel{\text{def}}{=} \\
&(\exists m : \mathbf{Manifest}, m = \mathit{getManifestForApp}(app, s) \\
&\quad \wedge \mathit{getPermissionId}(p) \in (\mathit{use} \ m)) \wedge \\
&(\mathit{isSystemPerm} \ p \ \vee \ \mathit{usrDefPerm} \ p) \wedge \\
&p \notin \mathit{grantedPerms}(app, s) \wedge \\
&\mathit{permLevel}(p) = \mathit{dangerous} \wedge \\
&(\exists g : \mathbf{PermGroup}, \mathit{getPermissionGroup}(p) = \mathit{Some} \ g \\
&\quad \wedge g \in \mathit{getAuthorizedGroups}(app, s)) \\
Post(s, \mathbf{grantAuto} \ p \ app, s') &\stackrel{\text{def}}{=} \\
&\mathit{grantPerm}(app, p, s, s') \wedge \\
&\mathit{sameOtherFieldsOnGrantAuto}(s, s')
\end{aligned}$$

The precondition establishes several conditions that must be fulfilled before this action is able to transition. The first one requires that the permission  $p$  is listed on the application's manifest (and this manifest, of course, is required to exist). Regarding the permission, it is also required that it is defined either by the user or the system, that its level is *dangerous* and that it has not been already granted to  $app$ . Up to this point, the precondition of  $\mathit{grantAuto}$  is exactly the same as the one of  $\mathit{grant}$ . The main difference is established by the following condition: the permission at issue should belong to a group  $g$  and the system should know that the user had previously authorized that group for automatic granting.

The postcondition of  $\mathbf{grantAuto} \ p \ a$  requires that for an initial state  $s$  and a final state  $s'$ , the individual permission  $p$  is granted to application  $app$ . This condition is enforced by the  $\mathit{grantPerm} \ a \ p \ s \ s'$  predicate which only alters the state in component that maps applications with their current *dangerous* permissions. Every other component of the state remains the same.

### 3.3 Executions

Whenever the system attempts to execute an action  $a$  over a valid state  $s$ , there are two possible outcomes. If the precondition holds, the system will transition to another state  $s'$  where the postcondition of  $a$  is established; but if the precondition is not satisfied on  $s$ , then the state remains unchanged and the system answers with an error message determined by the relation  $\mathit{ErrorMsg}^4$ .

Formally, the possible answers of the system are defined by the type  $\mathit{Response} \stackrel{\text{def}}{=} \mathit{ok} \mid \mathit{error} \ (ec : \mathit{ErrorCode})$  and the executions can be specified with this operational semantics:

$$\frac{\mathit{valid\_state}(s) \ Pre(s, a) \ Post(s, a, s')}{s \xrightarrow{a/ok} s'} \quad \frac{\mathit{valid\_state}(s) \ \mathit{ErrorMsg}(s, a, ec)}{s \xrightarrow{a/error(ec)} s}$$

One-step execution with error management preserves valid states.

► **Lemma 1** (Validity is invariant).

$$\forall (s \ s' : \mathbf{AndroidST})(a : \mathbf{Action})(r : \mathit{Response}), s \xrightarrow{a/r} s' \rightarrow \mathit{valid\_state}(s')$$

<sup>4</sup> Given a state  $s$ , an action  $a$  and an error code  $ec$ ,  $\mathit{ErrorMsg}(s, a, ec)$  holds iff  $\mathit{error} \ ec$  is an acceptable response when the execution of  $a$  is requested on state  $s$ .

### 3:10 Towards a Certified Reference Monitor of the Android 10 Permission System

The property is proved by case analysis on  $a$ , for each condition in  $valid\_state$ , using several auxiliary lemmas [19].

System state invariants, such as state validity, are useful to analyze other relevant properties of the model. In particular, the results presented in this work are obtained from valid states of the system.

#### 3.4 Reasoning over the specified model

In this section we present and discuss some properties about the Android 10 security framework. We focus on safety-related properties about the changes introduced on the later versions of Android (mainly Oreo and 10) rather than on security issues. Nevertheless, we also found potentially dangerous behaviours that may not be considered in the informal documentation of the platform and we formally reasoned about them as well. The full formal definition of these properties can be found in [19], along with the corresponding proofs.

On Table 3 we introduce helper functions and predicates used to define the properties that will follow.

■ **Table 3** Helper functions and predicates.

Function/Predicate	Description
$appHasPermission(app, p, s)$	holds iff $app$ is considered to have permission $p$ on state $s$ .
$canGrant(cp, u, s)$	holds iff the content provider $cp$ allows the delegation of permissions over the resource at URI $u$ on state $s$ .
$canStart(c', c, s)$	holds if the app containing component $c'$ (installed in $s$ ) has the required permissions to create a new running instance of $c$ .
$cmpProtectedByPerm(c)$	returns the permission by which the component $c$ is protected.
$componentIsExported(c)$	holds iff the component $c$ is exported and can be accessed from other applications.
$existsRes(cp, u, s)$	holds iff the URI $u$ belongs to the content provider $cp$ on $s$ .
$getAppFromCmp(c, s)$	given a component $c$ on $s$ , returns the app to which it belongs.
$getAppRequestedPerms(m)$	given the manifest $m$ of an app, returns the set of permissions it uses.
$getDefPermsApp(app, s)$	returns the set of permissions defined by $app$ on state $s$ .
$getGrantedPermsApp(app, s)$	returns the set of individual permissions granted to $app$ on $s$ .
$getAuthorizedGroups(app, s)$	returns the set of permission groups that have been authorized for automatic granting for $app$ on $s$ .
$getInstalledApps(s)$	returns the set of identifiers of the applications installed on $s$ .
$getManifestForApp(app, s)$	returns the manifest of application $app$ on state $s$ .
$getPermissionId(p)$	returns the identifier of permission $p$ .
$getPermissionLevel(p)$	returns the permission level of permission $p$ .
$getPermissionGroup(p)$	returns <i>Some</i> $g$ if the permission $p$ is grouped or <i>None</i> if not.
$getRunningComponents(s)$	returns the set of pairs consisting of a running instance id, and its associated component currently running on state $s$ .
$inApp(c, app, s)$	holds iff the component $c$ belongs to application $app$ on state $s$ .
$permissionRequiredRead(c)$	returns the permission required for reading the component.
$permSACs(p, sac)$	holds iff permission $p$ is required for performing the system call $sac$ (of type $SACall$ ).
$oldAppNotVerified(a, s)$	holds iff the application $a$ is considered old and the user hasn't verified it in state $s$ .

The first property that we proved establishes a safety condition about the automatic granting of grouped permissions. It states that the system is not able to transition with this action unless the group of the permission involved is already authorized.

► **Property 1** (Automatic grant only possible on authorized groups).

$$\forall (s, s' : \text{AndroidST})(p : \text{Perm})(g : \text{PermGroup})(app : \text{Appld}),$$

$$\text{getPermissionLevel}(p) = \text{dangerous} \wedge \text{getPermissionGroup}(p) = \text{Some } g \wedge$$

$$g \notin \text{getAuthorizedGroups}(app, s) \rightarrow \neg s \xrightarrow{\text{grantAuto } p \text{ app/ok}} s'$$

*Android's permission system ensures that an automatic granting can only occur on permissions that belong to authorized groups.*

However, a few questions arise when trying to formally describe the situations in which a group is authorized. For instance, there is at least one valid state where the system can automatically grant a grouped permission to an app even though that the application has no other permission of the same group granted at that moment. This means that an application can have a group authorized for automatic granting via a permission that no longer exists. This is not necessarily a security flaw. It may be a design principle to avoid asking the user to authorize the same group too many times, but the decision is not clear or disambiguated in the official documentation.

► **Property 2** (Auto-granting permission without having others of the same group).

$$\exists (s : \text{AndroidST})(p : \text{Perm})(g : \text{PermGroup})(app : \text{Appld}), \text{valid\_state}(s) \wedge$$

$$\text{getPermissionLevel}(p) = \text{dangerous} \wedge \text{getPermissionGroup}(p) = \text{Some } g \wedge$$

$$\neg(\exists(p' : \text{Perm}), p' \in \text{getGrantedPermsApp}(app, s) \wedge$$

$$\text{getPermissionGroup}(p') = \text{Some } g) \wedge \text{Pre}(s, \text{grantAuto } p \ a)$$

*System can automatically grant a permission even though there is currently no other permission of that group granted to the app.*

The next property formalizes the situation described in Section 2.2 about normal and dangerous permissions sharing a group. We believe that permissions with different protection levels should not be allowed to share a group, since it could lead to a privilege escalation scenario.

► **Property 3** (Dangerous permission automatically granted without explicit consent).

$$\forall (s, s' : \text{AndroidST})(a : \text{Appld})(m : \text{Manifest})(c : \text{Cert})(resources : \text{list Res})$$

$$(g : \text{PermGroup})(pDang \ pNorm : \text{Perm}), s \xrightarrow{\text{install } a \ m \ c \ resources/ok} s' \rightarrow$$

$$\text{getPermissionLevel}(pDang) = \text{dangerous} \rightarrow \text{getPermissionGroup}(pDang) = \text{Some } g \rightarrow$$

$$\text{getPermissionLevel}(pNorm) = \text{normal} \rightarrow \text{getPermissionGroup}(pNorm) = \text{Some } g \rightarrow$$

$$\{pDang, pNorm\} \subseteq \text{getAppRequestedPerms}(m) \rightarrow \text{Pre}(s', \text{grantAuto } pDang \ a)$$

*An application that uses a normal and a dangerous permission of the same group, can obtain the dangerous one automatically after being installed.*

Users are able to revoke permissions at runtime. However, the UI does not allow to revoke grouped permissions individually, the complete group is invalidated instead. We consider this behavior to be expected and desirable, and therefore, we proved that our model is consistent with it.

► **Property 4** (Revoking group revokes grouped individual permissions).

$$\forall (s, s' : \text{AndroidST})(g : \text{PermGroup})(app : \text{Appld}), s \xrightarrow{\text{revokePermGroup } g \text{ app/ok}} s' \rightarrow$$

$$\neg(\exists(p : \text{Perm}), p \in \text{getGrantedPermsApp}(app, s') \wedge \text{getPermissionGroup}(p) = \text{Some } g)$$

*Whenever a user revokes a permission group from an application, every individual permission that belongs to that group is revoked.*



The following property reasons about another change mentioned in Section 2.2. It formalizes a good behaviour about the unverified legacy applications.

► **Property 5** (Unverified old app cannot receive intents).

$$\forall (s, s' : \text{AndroidST}) (i : \text{Intent}) (ic : \text{iComp}) (app : \text{AppId}), \\ \text{oldAppNotVerified}(app, s) \rightarrow \neg s \xrightarrow{\text{receiveIntent } i \text{ ic } app/ok} s'$$

*An old application that hasn't been verified by the user yet cannot receive intents, meaning that it can't start activities as well.*

Finally, we include here a property that holds since version 6 of Android. Any application that wants to send information through the network must have the permission `INTERNET`, but since this permission is of level *normal*, the application just needs to declare it as used in its manifest. This will give access to the network in an implicit and irrevocable way. Once again, this has been criticized due to the potential information leakage it allows. The following property formally generalizes this situation and embodies a reasonable argument to roll back this security issue introduced in Android `Marshmallow`.

► **Property 6** (Internet access implicitly and irrevocably allowed).

$$\forall (s : \text{AndroidST}) (sac : \text{SACall}) (c : \text{Comp}) (ic : \text{iComp}) (p : \text{Perm}), \\ \text{valid\_state}(s) \rightarrow \text{permSAC}(p, sac) \rightarrow \\ \text{getPermissionLevel}(p) = \text{normal} \rightarrow \text{getPermissionId}(p) \in \\ \text{getAppRequestedPerms}(\text{getManifestForApp}(\text{getAppFromComp}(c, s), s)) \rightarrow \\ (ic, c) \in \text{getRunningComponents}(s) \rightarrow s \xrightarrow{\text{call } ic \text{ sac}/ok} s$$

*If the execution of an Android API call only requires permissions of level normal, it is enough for an application to list them as used on its manifest file to be allowed to perform such call.*

## 4 A certified reference validation mechanism

The implementation we developed in our previous model consisted in a set of `Coq` functions such that for every action in our axiomatic specification there exists a function which stands for it. In this work we kept this approach, updating those functions for which its axiomatic counterpart changed and adding new ones for the new actions `verifyOldApp` and `grantAuto`.

Functions that implement actions are basically state transformers. Their definition follows this pattern: first, it is checked whether the precondition of the action is satisfied in state  $s$ , and then, if that is the case, another function is called to return a state  $s'$  where the postcondition of the action holds. Otherwise, the state  $s$  is returned unchanged along with an appropriate response specifying an error code which describes the failure. More formally, the returned value has type  $\text{Result} \stackrel{\text{def}}{=} \{\text{resp} : \text{Response}, st : \text{AndroidST}\}$ . In Figure 2 we present, as an example, the function that implements the execution of the `grant` action. The `Coq` code of this function, together with that of the remaining ones, can be found in [19]<sup>5</sup>. The function `grant_pre` is defined as the nested validation of each of the properties of the precondition, specifying which error to throw when one of them doesn't hold. In general, every `<action>_pre` function is defined this way. The function `grant_post` implements the expected behaviour of the `grant` action: the permission `perm` is prepended to the list<sup>6</sup> of given permissions of the application `app` and, if that permission is grouped, that group is also added to the list of permissions groups authorized by the user on that application.

<sup>5</sup> We omit here the formal definition of these functions due to space constraints.

<sup>6</sup> We implement the sets in the model with lists of `Coq`.



```

Definition grant_safe(perm, app, s) : Result :=
  match grant_pre(perm, app, s) with
    | Some ec ⇒ {error(ec), s}
    | None ⇒ {ok, grant_post(perm, app, s)}
  end.

```

■ **Figure 2** The function that implements the **grant** action.

## Step

All of these functions are grouped into a *step* function, which basically acts as an action dispatcher<sup>7</sup>. Figure 3 show the structure of this function.

```

Definition step(s, a) :=
  match a with
    | ... ⇒ ...
    | grant perm app ⇒ grant_safe(perm, app, s)
    | ... ⇒ ...
  end.

```

■ **Figure 3** Structure of the **step** function.

## Traces

We have modeled the execution of the permission validation mechanism during a session of the system as a function that implements the execution of a list of actions starting in an initial system state. The output of the execution, a trace, is the corresponding sequence of states.

```

Function trace (s : AndroidST) (actions : list Action) : list AndroidST :=
  match actions with
    | nil ⇒ nil
    | action :: rest ⇒ let s' := (step s action).st in s' :: trace s' rest
  end.

```

## 4.1 Correctness of the implementation

We proceed now to outline the proof that our functional implementation of the security mechanisms of Android correctly implements the axiomatic model. This property has been formally stated as the following correctness theorem which in turn was verified using Coq [19].

► **Theorem 2** (Correctness of the reference validation mechanism).

$$\forall (s : \text{AndroidST}) (a : \text{Action}), \text{valid\_state}(s) \rightarrow s \xrightarrow{a/\text{step}(s,a).\text{resp}} \text{step}(s,a).\text{st}$$

The proof of this theorem starts by performing a case analysis on the (decidable) predicate  $\text{Pre}(s, a)$ . Then, in case that the predicate holds, we apply Lemma 3; otherwise we continue by applying Lemma 4.

<sup>7</sup> Mechanism to trigger actions, on a state, according to the type of event considered.

► **Lemma 3** (Correctness of valid execution).

$$\forall (s : \text{AndroidST}) (a : \text{Action}), \text{valid\_state}(s) \rightarrow \text{Pre}(s, a) \rightarrow \\ s \xrightarrow{a/ok} \text{step}(s, a).st \wedge \text{step}(s, a).resp = ok$$

► **Lemma 4** (Correctness of error execution).

$$\forall (s : \text{AndroidST}) (a : \text{Action}), \text{valid\_state}(s) \rightarrow \neg \text{Pre}(s, a) \rightarrow \exists (ec : \text{ErrorCode}), \\ \text{step}(s, a).st = s \wedge \text{step}(s, a).resp = \text{error}(ec) \wedge \text{ErrorMsg}(s, a, ec)$$

The proof of these lemmas proceeds by applying functional induction on  $\text{step}(s, a)$ . Then, in Lemma 3, the proof continues by applying the corresponding subproof of soundness of the function that implements each action; whereas in Lemma 4, a subproof about the existence of a proper error code is provided.

## 4.2 Reasoning over the certified reference validation mechanism

In this section we present several security properties we have stated and proved about the function  $\text{trace}$  defined in Section 4.

The first property states that in Android 10, if an application that is considered to be old (as we defined in Section 2.2) is able to run, then it has been verified and validated by the user previously.

► **Property 7** (Old applications must be verified).

$$\forall (\text{initState}, \text{lastState} : \text{AndroidST}) (\text{app} : \text{AppId}) (l : \text{list Action}), \text{valid\_state}(\text{initState}) \rightarrow \\ \text{app} \in \text{getInstalledApps}(\text{initState}) \rightarrow \text{oldAppNotVerified}(a, \text{initState}) \rightarrow \\ \text{canRun}(a, \text{lastState}) \rightarrow \text{last}(\text{trace}(\text{initState}, l), \text{initState}) = \text{lastState} \rightarrow \\ \text{uninstall } \text{app} \notin l \rightarrow \text{verifyOldApp } \text{app} \in l$$

*The only way for an old application to be able to execute is if the user verified it.*

The next property establishes that for an application to have **any** dangerous permission (grouped or ungrouped) it must be explicitly granted to it, either by the user or automatically by the system.

► **Property 8** (Dangerous permissions must be explicitly granted).

$$\forall (\text{initState}, \text{lastState} : \text{AndroidST}) (\text{app} : \text{AppId}) (p : \text{Perm}) (l : \text{list Action}), \\ \text{valid\_state}(\text{initState}) \rightarrow \text{app} \in \text{getInstalledApps}(\text{initState}) \rightarrow \\ \text{getPermissionLevel}(p) = \text{dangerous} \rightarrow \text{appHasPermission}(\text{app}, p, \text{lastState}) \rightarrow \\ \neg \text{appHasPermission}(\text{app}, p, \text{initState}) \rightarrow \text{uninstall } \text{app} \notin l \rightarrow \\ \text{last}(\text{trace}(\text{initState}, l), \text{initState}) = \text{lastState} \rightarrow (\text{grant } p \text{ } \text{app} \in l \vee \text{grantAuto } p \text{ } \text{app} \in l)$$

*The only way for an application to get a permission is if the user authorized it, or if the user authorized a group and the system is able to automatically grant it.*

The following property formally states that if an application used to have a permission that was later revoked, only re-granting it will allow the application to have it again.

► **Property 9** (Revoked permissions must be regranted).

$$\forall (\text{initState}, \text{sndState}, \text{lastState} : \text{AndroidST}) (\text{app} : \text{AppId}) (p : \text{Perm}) (l : \text{list Action}), \\ \text{valid\_state}(\text{initState}) \rightarrow \text{getPermissionLevel}(p) = \text{dangerous} \rightarrow \\ p \notin \text{getDefPermsForApp}(\text{app}, \text{initState}) \rightarrow \\ \text{step}(\text{initState}, \text{revoke } p \text{ } \text{app}).st = \text{sndState} \rightarrow \\ \text{step}(\text{initState}, \text{revoke } p \text{ } \text{app}).resp = ok \rightarrow \text{uninstall } \text{app} \notin l \rightarrow \text{grant } p \text{ } \text{app} \notin l \rightarrow \\ \text{grantAuto } p \text{ } \text{app} \notin l \rightarrow \text{last}(\text{trace}(\text{sndState}, l), \text{sndState}) = \text{lastState} \rightarrow \\ \neg \text{appHasPermission}(\text{app}, p, \text{lastState})$$

*If a permission is revoked from an application, only regranting it will allow the application to have it again.*

Whenever an application  $app$  receives a READ/WRITE permission  $perm$ , it also receives the right to delegate this permission to another application, say  $app'$ , to access that same resource on its behalf. However, if  $perm$  is later revoked from application  $app$ , there's a chance that  $app'$  still has access to that resource, since delegated permissions **are not recursively revoked**. The following property formalizes this situation and is a proof that the current specification allows a behavior which is arguably against the user's will.

► **Property 10** (Delegated permissions are not recursively revoked).

$$\begin{aligned} &\forall (s : \text{AndroidST})(p : \text{Perm})(app, app' : \text{Appld})(ic, ic' : \text{iComp})(c, c' : \text{Comp})(u : \text{Uri}) \\ &(cp : \text{CProvider}), \text{valid\_state}(s) \rightarrow \text{step}(s, \text{grant } p \text{ app}).\text{resp} = \text{ok} \rightarrow \\ &\text{getAppFromCmp}(c, s) = app \rightarrow \text{getAppFromCmp}(c', s) = app' \rightarrow \\ &(ic, c) \in \text{getRunningComponents}(s) \rightarrow (ic', c') \in \text{getRunningComponents}(s) \rightarrow \\ &\text{canGrant}(cp, u, s) \rightarrow \text{existsRes}(cp, u, s) \rightarrow \text{componentIsExported}(cp) \rightarrow \\ &\text{permissionRequiredRead}(cp) = \text{Some } p \rightarrow \end{aligned}$$

$$\begin{aligned} &\text{let opsResult} := \text{trace}(s, [\text{grant } p \text{ app}, \text{grantP } ic \text{ cp } app' \text{ u Read}, \\ &\text{revoke } p \text{ app}] \text{ in } \text{step}(\text{last}(\text{opsResult}, s), \text{read } ic' \text{ cp } u).\text{resp} = \text{ok} \end{aligned}$$

In Android 10, if a permission  $p$  is revoked for an application  $app$  not necessarily shall it be revoked for the applications to which  $app$  delegated  $p$ .

The purpose of the following property is to show that with runtime permissions introduced after **Android 6**, certain assertions on which a developer could rely in previous versions do not hold. For example, a running component may have the right of starting another one on a certain state, but may not be able to do so at a later time, even though no involved application was uninstalled. The property still holds on the latest version of Android.

► **Property 11** (The right to start an external component is revocable).

$$\begin{aligned} &\forall (\text{initState} : \text{AndroidST})(l : \text{listAction})(app, app' : \text{Appld})(c : \text{Comp})(act : \text{Activity}) \\ &(p : \text{Perm}), \text{valid\_state}(\text{initState}) \rightarrow \\ &\text{getPermissionLevel}(p) = \text{dangerous} \rightarrow \text{permissionIsGrouped}(p) = \text{None} \rightarrow \\ &app \neq app' \rightarrow p \notin \text{getDefPermsApp}(app, \text{initState}) \rightarrow \text{inApp}(c, app, \text{initState}) \rightarrow \\ &\text{inApp}(act, app', \text{initState}) \rightarrow \text{cmpProtectedByPerm}(act) = \text{Some } p \rightarrow \\ &\text{canStart}(c, act, \text{initState}) \rightarrow \exists (l : \text{list Action}), \text{uninstall } app \notin l \wedge \\ &\text{uninstall } app' \notin l \wedge \neg \text{canStart}(c, act, \text{last}(\text{trace}(\text{initState}, l), \text{initState})) \end{aligned}$$

A running component may have the right of starting another one on a certain state, but may not be able to do so at a later time.

## 5 Related work

Several analyses have been carried out concerning the security of the Android permission system. Plenty of them [11, 30, 13, 29, 23, 5] implement a static analysis tool that is capable of detecting overprivileges and unwanted information flow on a set of applications. This pragmatic approach may be helpful for Android users to keep their private information secure, but no properties about the system can be established. Recently, Mayrhofer *et al.* [22] described the Android security platform and documented the complex threat model and ecosystem it needs to operate, but no formal analysis was performed in it.

Few works study the aspects of the permission enforcing framework in a formal way. In particular, Shin *et al.* [25, 26] developed using Coq a framework that represents the Android permission system, similarly to what we did. Although, that work does not consider the different types of components, the interaction between a running instance and the system, the R/W operation on a content provider, the semantics of the permission delegation mechanism. Also, their work is based on an older version of the platform and some novel aspects, like the

management of runtime permissions or the verification of legacy applications, are not included. Similarly, Bagheri *et al.* [6] formalized Android’s permission protocol using Alloy [15]. The analysis performed, however, was based on the ability to automatically find counterexamples provided by the Alloy framework, which the authors claim to be tremendously helpful for identifying vulnerabilities. A Coq-based approach like ours, requires more human effort to identify a flaw but provides stronger guarantees on security and safety properties. Another formal work on Android is CrashSafe [16], where the authors formalized in Coq the inter-component communication mechanism and proved its safety with regard to failures (or *crashes*). This work, similarly to ours, focus on safety properties rather than security ones.

On the other hand, many works have addressed the problem of relating inductively defined relations and executable functions. In particular, Tollitte *et al.* [28] show how to extract a functional implementation from an inductive specification in Coq, and [9] exhibits a similar approach for Isabelle. Earlier, alternative approaches such as [7, 8] aim to provide reasoning principles for executable specifications. In [12], the verification of properties of imperative programs is performed using techniques based on the specialization of constrained logic programs. In this work we are able to develop independently the specification of the reference monitor and the implementation of the validation mechanism, considering that Coq provides a reasoning framework based on higher order logic to perform proofs of specifications and programs and a functional programming language. Other approaches could be considered to develop the formalization. For instance, a logical approach like the one used in [12]. However, a logical approach does not allow us to have the same functionalities in a unified formal environment.

Specifically, in this work we present a model of a reference monitor and demonstrate properties which shall hold for every correct implementation of the model. Then, we have developed a functional implementation in Coq of the reference validation mechanism and proved its correctness with respect to the specified reference monitor. Applying the program extraction mechanism provided by Coq we have also derived a certified Haskell prototype of the reference validation mechanism, which can be used to conduct verification activities on actual real implementations of the platform. The results presented in this paper extend the ones reported in [10, 21]. We have enriched the model presented in [10, 21] so as to consider the changes introduced in Android permission system by version Nougat, Oreo, Pie and 10.

## 6 Final remarks

We have enhanced the formal specification considered in our previous work [10] with the new features concerning the permission system that have been added during the later releases of Android. With a conservative approach, we first analyzed the validity of the already formulated properties and then established new ones about the novel changes; summing up a total of 14 valid properties, without including the auxiliary lemmas that have been separated just for modularization. Among these properties we included several that aim to highlight how formal methods help to disambiguate unclear behaviours that may be inferred from an informal specification. For instance, we found a potentially dangerous situation in which an application can gain access to every dangerous permission that shares group with a normal one, without explicit consent of the user (see Property 3). This scenario fits the model (informally) described in the official documentation of the platform.

We also enriched our previous functional implementation of the reference validation mechanism with these new characteristics and updated its correctness proof. As consequence, the derived Haskell prototype obtained using the program extraction mechanism provided by the proof assistant, has been updated as well. The full certified code is available in [19] and is about 23k LOC of Coq, including proofs.

One important goal of our work is to keep our formalization up to date with the later versions of Android in order to constitute a reliable framework for reasoning about its permission system. We aim to help to increase the confidence on Android’s security mechanisms by providing certified guarantees about the enforcement of this measures. The use of idealized models and certified prototypes is a good step forward but no doubt the definitive step is to be able to provide similar guarantees concerning actual implementations of the platform. We plan to use the certified extracted algorithm as a testing oracle and also to conduct verification activities on actual implementations of the platform, following the methodology proposed in [21]. In particular, we are investigating the use of that algorithm to compare the results of executing an action on a real Android platform and executing that same action on the correct program. This would allow us to monitor the actions performed in a real system and assessing whether the intended security policy is actually enforced by the particular implementation of the platform.

On September 8th 2020, Android 11 was released. This update includes features that continue increasing the security of the device, such as auto-resetting permissions from unused applications or one-time permissions for the most sensitive resources, like the microphone or camera. In future work, we intend to add this features to our model.

---

## References

- 1 J. P. Anderson. Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972. URL: <http://csrc.nist.gov/publications/history/ande72.pdf>.
- 2 Android Developers. *Application Fundamentals*. Available at: <http://developer.android.com/guide/components/fundamentals.html>. Last access: Feb. 2021.
- 3 Android Developers. *Permissions*. Available at: <http://developer.android.com/guide/topics/security/permissions.html>. Last access: Feb. 2021.
- 4 Android Developers. *Protection levels*. Available at: <https://developer.android.com/guide/topics/permissions/overview#normal-dangerous>. Last access: Feb. 2021.
- 5 H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, September 2015. doi:10.1109/TSE.2015.2419611.
- 6 Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing*, 30, November 2017. doi:10.1007/s00165-017-0445-z.
- 7 A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In M. Aagaard and J. Harrison, editors, *TPHOLs*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000. doi:10.1007/3-540-44659-1\_1.
- 8 G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *LNCS*, pages 114–129. Springer, 2006. doi:10.1007/11737414\_9.
- 9 Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 131–146. Springer, 2009. doi:10.1007/978-3-642-03359-9\_11.
- 10 Gustavo Betarte, Juan Campo, Felipe Gorostiaga, and Carlos Luna. *A Certified Reference Validation Mechanism for the Permission Model of Android: 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers*. Springer, July 2018. doi:10.1007/978-3-319-94460-9\_16.
- 11 P. Chester, C. Jones, M. Wiem Mkaouer, and D. E. Krutz. M-perm: A lightweight detector for android permission gaps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 217–218, May 2017. doi:10.1109/MOBILESoft.2017.23.

- 12 E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95:149–175, 2014. doi:10.1016/j.scico.2014.05.017.
- 13 Michael Gordon, Kim deokhwan, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. In *NDSS Symposium 2015*, January 2015. doi:10.14722/ndss.2015.23089.
- 14 International Data Corporation (IDC). Smartphone market share. Technical report, International Data Corporation (IDC), 2020.
- 15 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.
- 16 Wilayat Khan, Habib Ullah, Aakash Ahmad, Khalid Sultan, Abdullah Alzahrani, Sultan Khan, Mohammad Alhumaid, and Sultan Abdulaziz. Crashesafe: a formal model for proving crash-safety of android applications. *Human-centric Computing and Information Sciences*, 8, December 2018. doi:10.1186/s13673-018-0144-7.
- 17 P. Letouzey. *Programmation fonctionnelle certifiée – L’extraction de programmes dans l’assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.
- 18 Pierre Letouzey. A New Extraction for Coq. In *Proceedings of TYPES’02*, volume 2646 of *LNCS*, 2003.
- 19 Guido De Luca and Carlos Luna. Formal verification of the security model of Android 10: Coq code. Available at: <https://github.com/g-deluca/android-coq-model>. Last access: Feb. 2021.
- 20 Guido De Luca and Carlos Luna. Towards a certified reference monitor of the android 10 permission system. *CoRR*, abs/2011.00720, 2020. arXiv:2011.00720.
- 21 Carlos Luna, Gustavo Betarte, Juan Diego Campo, Camila Sanz, Maximiliano Cristiá, and Felipe Gorostiaga. A formal approach for the verification of the permission-based security model of android. *CLEI Electron. J.*, 21(2), 2018. doi:10.19153/cleiej.21.2.3.
- 22 René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kravovich. The android platform security model. *CoRR*, abs/1904.05572, 2019. arXiv:1904.05572.
- 23 Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE ’15*, pages 77–88, Piscataway, NJ, USA, 2015. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2818754.2818767>.
- 24 Open Handset Alliance. *Android project*. Available at: [//source.android.com/](http://source.android.com/). Last access: Feb. 2021.
- 25 W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. In *SocialCom’10*, pages 944–951, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/SocialCom.2010.140.
- 26 W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A first step towards automated permission-enforcement analysis of the android framework. In *SAM 2010*, pages 323–329. CSREA Press, 2010.
- 27 The Coq Team. *The Coq Proof Assistant Reference Manual – Version V8.12.0*, 2020. URL: <http://coq.inria.fr>.
- 28 P.-N. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 76–91. Springer, 2012. doi:10.1007/978-3-642-35308-6\_9.
- 29 Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security*, 21:1–32, April 2018. doi:10.1145/3183575.
- 30 S. Wu and J. Liu. Overprivileged permission detection for android applications. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2019. doi:10.1109/ICC.2019.8761572.



# Coinductive Proof Search for Polarized Logic with Applications to Full Intuitionistic Propositional Logic

José Espírito Santo ✉ 

Centre of Mathematics, University of Minho, Braga, Portugal

Ralph Matthes ✉ 

CNRS, Institut de Recherche en Informatique de Toulouse (IRIT), France

Luís Pinto ✉ 

Centre of Mathematics, University of Minho, Braga, Portugal

---

## Abstract

The approach to proof search dubbed “coinductive proof search”, and previously developed by the authors for implicational intuitionistic logic, is in this paper extended to  $LJP$ , a focused sequent-calculus presentation of polarized intuitionistic logic, including an array of positive and negative connectives. As before, this includes developing a coinductive description of the search space generated by a sequent, an equivalent inductive syntax describing the same space, and decision procedures for inhabitation problems in the form of predicates defined by recursion on the inductive syntax. We prove the decidability of existence of focused inhabitants, and of finiteness of the number of focused inhabitants for polarized intuitionistic logic, by means of such recursive procedures. Moreover, the polarized logic can be used as a platform from which proof search for other logics is understood. We illustrate the technique with  $LJT$ , a focused sequent calculus for full intuitionistic propositional logic (including disjunction). For that, we have to work out the “negative translation” of  $LJT$  into  $LJP$  (that sees all intuitionistic types as negative types), and verify that the translation gives a faithful representation of proof search in  $LJT$  as proof search in the polarized logic. We therefore inherit decidability of both problems studied for  $LJP$  and thus get new proofs of these results for  $LJT$ .

**2012 ACM Subject Classification** Theory of computation → Type theory; Theory of computation → Proof theory

**Keywords and phrases** Inhabitation problems, Coinduction, Lambda-calculus, Polarized logic

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.4

**Funding** The first and the last authors were partially financed by Portuguese Funds through FCT (Fundação para a Ciência e a Tecnologia) within the Projects UIDB/00013/2020 and UIDP/00013/2020. All authors got financial support by the COST action CA15123 EUTYPES.

**Acknowledgements** We would like to thank for the careful and thoughtful review by an anonymous referee.

## 1 Introduction and Motivation

An approach to proof search dubbed “coinductive proof search” has been developed by the authors [5, 7]. The approach is based on three main ideas: (i) the Curry-Howard paradigm of representation of proofs (by typed  $\lambda$ -terms) is extended to solutions of proof-search problems (a solution is a run of the proof search process that, if not completed, does not fail to apply bottom-up an inference rule, so it may be an infinite object); (ii) two typed  $\lambda$ -calculi are developed for the effect, one being obtained by a co-inductive reading of the grammar of proof terms, the other being obtained by enriching the grammar of proof terms with a formal fixed-point operator to represent cyclic behaviour, the first calculus acting as the universe



© José Espírito Santo, Ralph Matthes, and Luís Pinto;  
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 4; pp. 4:1–4:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for the mathematical definition of concepts pertaining to proof search (e. g., the existence of solutions for a given logical sequent), the second calculus acting as the finitary setting where algorithmic counterparts of those concepts can be found; (iii) formal (finite) sums are employed throughout to represent choice points, so not only solutions but even entire solution spaces are represented, both coinductively and finitarily.

The approach was developed systematically for intuitionistic implicational logic, delivering new solutions to inhabitation and counting problems, and proofs of the state-of-the-art coherence theorems, in the simply typed  $\lambda$ -calculus [8]; it also helped the investigation of new questions, like the various concepts of finiteness suggested by proof search [6].

The goal of this paper is to extend this approach to *polarized*, intuitionistic propositional logic with a rich choice of positive and negative connectives [17, 4], and to proof search in a full-fledged focused sequent calculus. Polarized logic can be used as a platform from which proof search for other logics is understood [15]. The extension to polarized logic also aims at obtaining results about proof search for full intuitionistic propositional logic.

In this paper, coinductive proof search is applied to *LJP*, a focused sequent-calculus presentation of polarized logic. The extension works smoothly, which is a sign of the robustness of the approach, that has been developed for a relatively simple logic. Only the luxuriant syntax (typical of focused systems, rich in various forms of judgments) puts a notational challenge, and we make a proposal for that. Unlike the case of implicational logic we described in previous work of ours, guardedness of the coinductively described expressions is not enforced by the grammar alone, and so it has to be made an extra assumption; and focusing suggests a refinement of our approach: formal sums are not needed in the inversion phases, and the infinity of solutions must go infinitely often through stable sequents (this can be expressed by a rather simple instance of the parity condition). In the end, we obtain for *LJP* decidability of provability, and decidability of finiteness of the number of proofs, with our typical two-staged decision procedure: a function that calculates the finitary representation (in the calculus with formal fixed points) of the solution space of the given logical sequent, composed with a syntax-directed, recursive predicate that tests the desired property.

As said, from the results about the polarized logic, we can extract results for other logics. We illustrate the technique with *LJT*, a focused sequent calculus for full intuitionistic propositional logic (including disjunction) [11, 3]. For that, we define the “negative translation” of *LJT* into *LJP*, that sees all intuitionistic formulas as negative formulas (an idea rooted in the  $!A \multimap B$  translation by Girard of intuitionistic logic into linear logic, and developed in various contexts [19, 15, 1]). While the translation of formulas is mostly dictated by polarity, there are subtle problems with a definition of the translation of proof terms without knowing the logical sequent they witness (see the definitions of  $DLV(t)$  and atomic and positive spines in Section 5). Soundness of a translation is its first aim, but we also crucially need to guarantee that the translation gives a faithful representation of proof search in *LJT* as proof search in *LJP*. In proving this result, we benefited from the language of proof terms developed for polarized logic in [4].

**Plan of the paper.** The sequent-calculus presentation of polarized logic from [4] is reviewed in Section 2. Coinductive proof search for *LJP* occupies Sections 3 and 4. Applications to full intuitionistic logic are extracted in Section 5. Section 6 concludes.



## 2 Background on the system *LJP* of polarized propositional logic

We introduce the sequent calculus *LJP* for polarized intuitionistic propositional logic (PIPL). *LJP* is a variant of the cut-free fragment of  $\lambda_G^\pm$  [4].

*Formulas* of *LJP* are as follows (unchanged from  $\lambda_G^\pm$ ):

$$\begin{array}{ll}
\text{(formulas)} & A ::= N \mid P \\
\text{(negative)} & N, M ::= C \mid a^- \\
\text{(composite negative)} & C ::= \uparrow P \mid P \supset N \mid N \wedge M \\
\text{(positive)} & P, Q ::= a^+ \mid \downarrow N \mid \perp \mid P \vee Q
\end{array}$$

Here, we assume a supply of (names of) atoms, denoted typically by  $a$ ; the markers  $-$  and  $+$  for polarity are added to the atom (name) as superscripts, giving rise to negative resp. positive atoms. The symbols  $\perp$ ,  $\wedge$  and  $\vee$  obviously stand for falsity, conjunction and disjunction,  $\supset$  stands for implication, and  $\uparrow$  and  $\downarrow$  are polarity shifts (as they are commonly denoted in the literature). We call right formulas or R-formulas positive formulas and negative atoms. The set of formulas is thus partitioned in two ways: into negative and positive formulas, and into composite negative and right formulas. The second partitioning plays an important role in *LJP*, more than in  $\lambda_G^\pm$ . We also use the notion of left formulas or L-formulas: they are either negative formulas or positive atoms.

*Proof terms* of *LJP* are organized in five syntactic categories as follows:

$$\begin{array}{ll}
\text{(values)} & v ::= z \mid \mathbf{thunk}(t) \mid \mathbf{in}_i^P(v) \\
\text{(terms)} & t ::= [e] \mid \ulcorner e \urcorner \mid \lambda p \mid \langle t_1, t_2 \rangle \\
\text{(co-values/spines)} & s ::= \mathbf{nil} \mid \mathbf{cothunk}(p) \mid v :: s \mid i :: s \\
\text{(co-terms)} & p ::= z^{a^+} . e \mid x^N . e \mid \mathbf{abort}^A \mid [p_1, p_2] \\
\text{(stable expressions)} & e ::= \mathbf{dlv}(t) \mid \mathbf{ret}(v) \mid \mathbf{coret}(x, s)
\end{array}$$

where  $i \in \{1, 2\}$ , and  $z$  and  $x$  range over countable sets of variables assumed to be disjoint, called positive resp. negative variables.<sup>1</sup> The syntax deviates from  $\lambda_G^\pm$  [4, Figure 4] in the following ways: the letters to denote values and covalues are now in lower case, the two expressions to type the cut rules are absent, and the last form of values (the injections) and **abort** come with type information, as well as the binding occurrences of variables in the first two forms of co-terms – all the other syntax elements do not introduce variable bindings, in particular, there is no binding in  $\lambda p$  or  $\mathbf{coret}(x, s)$ . Often we refer to all proof terms of *LJP* as *expressions*, and use letter  $T$  to range over expressions in this wide sense ( $T$  being reminiscent of terms, but not confined to the syntactic category  $t$ ). To shorten notation, we communicate  $\langle t_1, t_2 \rangle$  and  $[p_1, p_2]$  as  $\langle t_i \rangle_i$  and  $[p_i]_i$ , respectively.

We also use the typical letters for denoting elements of the syntactic categories as sorts: let  $S := \{v, t, s, p, e\}$  be their set, and use letter  $\tau$  to denote any element of  $S$ .

Since proof terms of *LJP* come with some extra type information as compared to  $\lambda_G^\pm$ , the typing rules will be adjusted accordingly. The typing relation will also be slightly reduced: it is assumed that the *Focus<sub>L</sub>*-rule of  $\lambda_G^\pm$  (the one typing the **coret** construction for proof terms) only applies if the right-hand side formula is an R-formula. This also means that *focus negative left* sequents can be restricted to R-formulas on the right-hand side, which we therefore do in *LJP*.

<sup>1</sup> At first sight, these proof terms are far removed from any familiar sort of  $\lambda$ -terms; and the fact that cut-elimination does not belong to this paper means that no reduction semantics will be given here to help grasping what they are. As detailed in [4], this language refines call-by-push-value [14], with the positive/negative distinction being related to the value/computation distinction. In Section 5 the translation of the more familiar proof terms from *LJT* into these proof terms gives some insight. Bear in mind proof terms are the cornerstone of coinductive proof search, as both the coinductive and the finitary representations of search spaces are based on them.

$$\begin{array}{c}
\frac{}{\Gamma, z : a^+ \vdash [z : a^+]} \quad \frac{\Gamma \Longrightarrow t : N}{\Gamma \vdash [\text{thunk}(t) : \downarrow N]} \quad \frac{\Gamma \vdash [v : P_i]}{\Gamma \vdash [\text{in}_i^{P_3-i}(v) : P_1 \vee P_2]} \quad i \in \{1, 2\} \\
\frac{\Gamma \vdash e : a^-}{\Gamma \Longrightarrow \lceil e \rceil : a^-} \quad \frac{\Gamma \vdash e : P}{\Gamma \Longrightarrow \lceil e \rceil : \uparrow P} \quad \frac{\Gamma \mid p : P \Longrightarrow N}{\Gamma \Longrightarrow \lambda p : P \supset N} \quad \frac{\Gamma \Longrightarrow t_i : N_i \text{ for } i = 1, 2}{\Gamma \Longrightarrow \langle t_i \rangle_i : N_1 \wedge N_2} \\
\frac{}{\Gamma[\text{nil} : a^-] \vdash a^-} \quad \frac{\Gamma \mid p : P \Longrightarrow R}{\Gamma[\text{cothunk}(p) : \uparrow P] \vdash R} \quad \frac{\Gamma \vdash [v : P] \quad \Gamma[s : N] \vdash R}{\Gamma[v :: s : P \supset N] \vdash R} \\
\frac{\Gamma[s : N_i] \vdash R}{\Gamma[i :: s : N_1 \wedge N_2] \vdash R} \quad i \in \{1, 2\} \quad \frac{\Gamma, z : a^+ \vdash e : A}{\Gamma \mid z^{a^+}.e : a^+ \Longrightarrow A} \quad \frac{\Gamma, x : N \vdash e : A}{\Gamma \mid x^N.e : \downarrow N \Longrightarrow A} \\
\frac{}{\Gamma \mid \text{abort}^A : \perp \Longrightarrow A} \quad \frac{\Gamma \mid p_1 : P_1 \Longrightarrow A \quad \Gamma \mid p_2 : P_2 \Longrightarrow A}{\Gamma \mid [p_i]_i : P_1 \vee P_2 \Longrightarrow A} \\
\frac{\Gamma \Longrightarrow t : C}{\Gamma \vdash \text{dlv}(t) : C} \quad \frac{\Gamma \vdash [v : P]}{\Gamma \vdash \text{ret}(v) : P} \quad \frac{\Gamma, x : N[s : N] \vdash R}{\Gamma, x : N \vdash \text{coret}(x, s) : R}
\end{array}$$

■ **Figure 1** Inductive definition of typing rules of *LJP*.

There are five forms of *sequents*, one for each syntactic category  $\tau$  of proof terms (the full names and the rationales of the categories are found in [4]):

$$\begin{array}{ccc}
\text{(focus negative left)} & \Gamma[s : N] \vdash R & \text{(focus positive right)} & \Gamma \vdash [v : P] \\
\text{(invert positive left)} & \Gamma \mid p : P \Longrightarrow A & \text{(invert negative right)} & \Gamma \Longrightarrow t : N \\
\text{(stable)} & \Gamma \vdash e : A & & 
\end{array}$$

The rules, given in Fig. 1, are the obvious adaptations of the ones in [4, Figures 1–3] (omitting the cut rules), given the more annotated syntax and the mentioned restrictions to R-formulas in some places. We recall that  $\Gamma$  is a context made of associations of variables with left formulas that respect polarity, hence these associations are either  $z : a^+$  or  $x : N$  (in other words, positive variables are assigned atomic types only). The extra annotations ensure uniqueness of typing in that, given the shown context, type and term information, there is at most one formula that can replace any of the placeholders in  $\Gamma[s : N] \vdash \cdot$ ,  $\Gamma \vdash [v : \cdot]$ ,  $\Gamma \mid p : \cdot \Longrightarrow \cdot$ ,  $\Gamma \Longrightarrow t : \cdot$  and  $\Gamma \vdash e : \cdot$ .

We also consider sequents without proof-term annotations, i. e.,  $\Gamma \vdash [P]$ ,  $\Gamma \Longrightarrow N$ ,  $\Gamma[N] \vdash R$ ,  $\Gamma \mid P \Longrightarrow A$  and  $\Gamma \vdash A$ , that we will call logical sequents. The letters  $\rho, \rho'$  etc. will range over  $\Gamma \vdash R$ , with an R-formula on the right-hand side. Those will be called R-stable sequents. (Such logical sequents cannot be proven by a proof term of the form  $\text{dlv}(t)$ .) Results about all forms of sequents can sometimes be presented uniformly, with the following notational device: If  $\sigma$  is any logical sequent and  $T$  a proof term of the suitable syntactic category, let  $\sigma(T)$  denote the sequent obtained by placing “ $T$  :” properly into  $\sigma$ , e. g., if  $\sigma = (\Gamma \mid P \Longrightarrow A)$ , then  $\sigma(p) = (\Gamma \mid p : P \Longrightarrow A)$  (the parentheses around sequents are often used for better parsing of the text). We sometimes indicate the syntactic category  $\tau$  of  $T$  as upper index of  $\sigma$ , e. g., an arbitrary logical sequent  $\Gamma \vdash A$  is indicated by  $\sigma^e$ .

We also use the set  $S$  of sorts to give a more uniform view of the different productions of the grammar of *LJP* proof terms. E. g., we consider  $\text{thunk}(\cdot)$  as a unary function symbol, which is typed/sorted as  $t \rightarrow v$ , to be written as  $\text{thunk}(\cdot) : t \rightarrow v$ . As another example, we see

co-pairing as binary function symbol  $[\cdot, \cdot] : p, p \rightarrow p$ . This notational device does not take into account variable binding, and we simply consider  $z^{a^+} \cdot \cdot$  as a unary function symbol for every  $z$  and every  $a$ . The positive variables  $z$  have no special role either in this view, so they are all nullary function symbols (i. e., constants) with sort  $v$ . Likewise, for every negative variable  $x$ ,  $\text{coret}(x, \cdot)$  is a unary function symbol sorted as  $s \rightarrow e$ . We can thus see the definition of proof terms of  $LJP$  as based on an infinite signature, with function symbols  $f$  of arities  $k \leq 2$ . The inductive definition of proof terms of  $LJP$  can then be depicted in the form of one rule scheme:

$$\frac{f : \tau_1, \dots, \tau_k \rightarrow \tau \quad T_i : \tau_i, 1 \leq i \leq k}{f(T_1, \dots, T_k) : \tau}$$

Later we will write  $f(T_i)_i$  in place of  $f(T_1, \dots, T_k)$  and assume that  $k$  is somehow known. Instead of writing the  $k$  hypotheses  $T_i : \tau_i$ , we will then just write  $\forall i, T_i : \tau_i$ .

### 3 Coinductive approach to proof search in the polarized system $LJP$

In this section, we adapt our coinductive approach to proof search from implicational intuitionistic logic to  $LJP$ . Due to the high number of syntactic categories and different constructors for proof terms, we use the extra notational devices from the end of Section 2 to ensure a uniform presentation of mostly similar rules that appear in definitions. Our previous development sometimes departs from such a uniformity, which is why we also widen the grammar of “forests”. This in turn asks for a mathematically more detailed presentation of some coinductive proofs that are subtle but lie at the heart of our analysis. (For reasons of limited space, that presentation was moved into Appendix A.5.)

#### 3.1 Search for inhabitants in $LJP$ , coinductively

System  $LJP_{\Sigma}^{\text{co}}$  extends the proof terms of  $LJP$  in two directions: there is a coinductive reading of the rules of the grammar of proof terms, and formal sums are added to the grammar as means to express alternatives. This general idea is refined when applied to the focused system  $LJP$ : the coinductive reading will be attached to stable expressions only; and the formal sums are not added to the categories of (co)terms, since (co)terms serve to represent the inversion phase in proof search, where choice is not called for.

The expressions in the wide sense of  $LJP_{\Sigma}^{\text{co}}$  are called *forests* and ranged by the letter  $T$ . They comprise five categories introduced by the simultaneous coinductive definition of the sets  $v_{\Sigma}^{\text{co}}$ ,  $t_{\Sigma}^{\text{co}}$ ,  $s_{\Sigma}^{\text{co}}$ ,  $p_{\Sigma}^{\text{co}}$ , and  $e_{\Sigma}^{\text{co}}$ . However, we will continue to use the sorts  $\tau$  taken from the set  $S$  that was introduced for  $LJP$ . This allows us to maintain the function-symbol view of  $LJP$  with the same symbols  $f$  that keep their typing/sorting. As said, only for the classes of values, spines and expressions, we add finite sums, denoted with the multiary function symbols  $\Sigma^{\tau}$  for  $\tau \in \{v, s, e\}$ . The definition of the set of forests, i. e., the expressions (in a wide sense) of  $LJP_{\Sigma}^{\text{co}}$  can thus be expressed very concisely as being obtained by only two rule schemes:

$$\frac{f : \tau_1, \dots, \tau_k \rightarrow \tau \quad \forall i. T_i : \tau_i}{f(T_1, \dots, T_k) : \tau} \text{ coinductive if } \tau = e \quad \frac{\forall i. T_i : \tau}{\sum_i^{\tau} T_i : \tau} \tau \in \{v, s, e\}$$

The doubly horizontal line indicates a possibly coinductive reading. As a first step, we read all these inference rules coinductively, but in a second step restrict the obtained infinitary expressions to obey the following property: infinite branches must go infinitely often through the  $e$ -formation rules coming from  $LJP$ , i. e., those depicted as unary function symbols  $f : \tau_1 \rightarrow e$  (also called the *inherited*  $e$ -formation rules – those for  $\text{dlv}(\cdot)$ ,  $\text{ret}(\cdot)$  and  $\text{coret}(x, \cdot)$ ).

This can be expressed as the *parity condition* (known from parity automata where this is the acceptance condition) based on priority 2 for any rule for those  $f : \tau_1 \rightarrow e$  and priority 1 for all the others. The parity condition requires that the maximum of the priorities seen infinitely often on a path in the (forest) construction is even, hence infinite cycling through the other syntactic categories and the summing operation for  $e$ -expressions is subordinate to infinite cycling through the inherited  $e$ -formation rules. Put less technically, we allow infinite branches in the construction of forests, but infinity is not allowed to come from infinite use solely of the “auxiliary” productions (for  $\tau \neq e$ ) or the additional sum operator for  $e$ , thus, in particular ruling out infinite pairing with angle brackets, infinite copairing with brackets or infinite spine composition by way of one of the  $::$  constructors – all of which would never correspond to typable proof terms – and also ruling out infinite stacks of finite sums.

Sums  $\sum_i^\tau T_i$  are required to be finite and therefore may also be denoted by  $T_1 + \dots + T_k$ , leaving  $\tau$  implicit. We write  $\mathbb{O}$  (possibly with the upper index  $\tau$  that obviously cannot be inferred from the summands) for empty sums. Sums are treated as sets of alternatives (so they are identified up to associativity, commutativity and idempotency – that incorporates  $\alpha$ -equivalence (this is still a  $\lambda$ -calculus, the presentation with function symbols  $f$  is a notational device) and bisimilarity coming from the full coinductive reading in the first step of the construction).

We now define an inductive notion of membership, hence restricting the notion we had in our previous papers on implicational logic.

► **Definition 1 (Membership).** *An LJP-expression  $T$  is a member of a forest  $T'$  when the predicate  $\text{mem}(T, T')$  holds, which is defined inductively as follows.*

$$\frac{\forall i. \text{mem}(T_i, T'_i)}{\text{mem}(f(T_i)_i, f(T'_i)_i)} \quad \frac{\text{mem}(T, T'_j)}{\text{mem}(T, T'_1 + \dots + T'_k)} \text{ for some } j$$

The intuition of this definition is obviously that the sums expressed by  $\sum_i^\tau$  represent alternatives out of which one is chosen for a concrete member.

The minimum requirement for this definition to be meaningful is that the five syntactic categories are respected: if  $\text{mem}(T, T')$  then  $T \in \tau$  iff  $T' \in \tau_{\Sigma}^{\text{co}}$ . This property holds since we tacitly assume that the sum operators are tagged with the respective syntactic category.

For a forest  $T$ , we call *finite extension* of  $T$ , which we denote by  $\mathcal{E}_{\text{fin}}(T)$ , the set of the (finite) members of  $T$ , i. e.,  $\mathcal{E}_{\text{fin}}(T) = \{T_0 \mid \text{mem}(T_0, T)\}$ . Properties of special interest in this paper are: (i)  $\text{exfinext}(T)$ , defined as:  $\mathcal{E}_{\text{fin}}(T)$  is nonempty; and  $\text{nofinext}$ , the complement of  $\text{exfinext}$ ; and (ii)  $\text{finfinext}(T)$ , defined as:  $\mathcal{E}_{\text{fin}}(T)$  is finite; and  $\text{inffinext}$ , the complement of  $\text{finfinext}$ . These predicates play an important role in Section 4.

In Fig. 2, analogously to our previous work [8], we inductively characterize  $\text{exfinext}$  and  $\text{finfinext}$ , and we coinductively characterize  $\text{nofinext}$  and  $\text{inffinext}$ . Note that the characterization of  $\text{finfinext}$  resp.  $\text{inffinext}$  depends upon the characterization of  $\text{nofinext}$  resp.  $\text{exfinext}$ . In Appendix A.1, it is shown that the characterizations in Fig. 2 are adequate, namely:  $\text{exfin} = \text{exfinext}$ ,  $\text{nofin} = \text{nofinext}$ ,  $\text{finfin} = \text{finfinext}$  and  $\text{inffin} = \text{inffinext}$ . As immediate consequences,  $\text{exfin}$  and  $\text{nofin}$  are complementary predicates, as are  $\text{finfin}$  and  $\text{inffin}$ , and additionally  $\text{nofin} \subseteq \text{finfin}$ .

Now, we are heading for the infinitary representation of all inhabitants of any logical sequent  $\sigma$  of LJP as a forest whose members are precisely those inhabitants (to be confirmed in Prop. 4). For all the five categories of logical sequents  $\sigma^\tau$ , we define the associated *solution space*  $\mathcal{S}(\sigma^\tau)$  as a forest, more precisely, an element of  $\tau_{\Sigma}^{\text{co}}$ , that is supposed to represent the space of solutions generated by an exhaustive and possibly non-terminating search process applied to that given logical sequent  $\sigma^\tau$ . This is by way of the following simultaneous coinductive definition. It is simultaneous for the five categories of logical sequents. For each category, there is an exhaustive case analysis on the formula argument.

$$\frac{\forall i. \text{exfin}(T_i)}{\text{exfin}(f(T_i)_i)} \quad \frac{\text{exfin}(T_j)}{\text{exfin}(\sum_i T_i)} \quad \frac{\text{nofin}(T_j)}{\text{nofin}(f(T_i)_i)} \quad \frac{\forall i. \text{nofin}(T_i)}{\text{nofin}(\sum_i T_i)}$$

$$\frac{\forall i. \text{finfin}(T_i)}{\text{finfin}(f(T_i)_i)} \quad \frac{\text{nofin}(T_j)}{\text{finfin}(f(T_i)_i)} \quad \frac{\forall i. \text{finfin}(T_i)}{\text{finfin}(\sum_i T_i)} \quad \frac{\text{inffin}(T_j)}{\text{inffin}(f(T_i)_i)} \quad \frac{\forall i. \text{exfin}(T_i)}{\text{inffin}(\sum_i T_i)} \quad \frac{\text{inffin}(T_j)}{\text{inffin}(\sum_i T_i)}$$

■ **Figure 2** Predicates `exfin`, `nofin`, `finfin` and `inffin`.

$$\begin{aligned} \mathcal{S}(\Gamma \vdash [a^+]) &:= \sum_{(z:a^+) \in \Gamma} z & \mathcal{S}(\Gamma \vdash [\perp]) &:= \mathbb{O}_v \\ \mathcal{S}(\Gamma \vdash [\downarrow N]) &:= \text{thunk}(\mathcal{S}(\Gamma \Rightarrow N)) & \mathcal{S}(\Gamma \vdash [P_1 \vee P_2]) &:= \sum_{i \in \{1,2\}} \text{in}_i^{P_3-i}(\mathcal{S}(\Gamma \vdash [P_i])) \\ \mathcal{S}(\Gamma \Rightarrow a^-) &:= \lceil \mathcal{S}(\Gamma \vdash a^-) \rceil & \mathcal{S}(\Gamma \Rightarrow P \supset N) &:= \lambda \mathcal{S}(\Gamma \mid P \Rightarrow N) \\ \mathcal{S}(\Gamma \Rightarrow \uparrow P) &:= \lceil \mathcal{S}(\Gamma \vdash P) \rceil & \mathcal{S}(\Gamma \Rightarrow N_1 \wedge N_2) &:= \langle \mathcal{S}(\Gamma \Rightarrow N_i) \rangle_i \\ \mathcal{S}(\Gamma[a^-] \vdash R) &:= \text{if } R = a^- \text{ then nil else } \mathbb{O}_s \\ \mathcal{S}(\Gamma[P \supset N] \vdash R) &:= \mathcal{S}(\Gamma \vdash [P]) :: \mathcal{S}(\Gamma[N] \vdash R) \\ \mathcal{S}(\Gamma[\uparrow P] \vdash R) &:= \text{cothunk}(\mathcal{S}(\Gamma \mid P \Rightarrow R)) \\ \mathcal{S}(\Gamma[N_1 \wedge N_2] \vdash R) &:= \sum_{i \in \{1,2\}} (i :: \mathcal{S}(\Gamma[N_i] \vdash R)) \\ \mathcal{S}(\Gamma \mid a^+ \Rightarrow A) &:= z^{a^+} \cdot \mathcal{S}(\Gamma, z : a^+ \vdash A) & \mathcal{S}(\Gamma \mid \perp \Rightarrow A) &:= \text{abort}^A \\ \mathcal{S}(\Gamma \mid \downarrow N \Rightarrow A) &:= x^N \cdot \mathcal{S}(\Gamma, x : N \vdash A) & \mathcal{S}(\Gamma \mid P_1 \vee P_2 \Rightarrow A) &:= [\mathcal{S}(\Gamma \mid P_i \Rightarrow A)]_i \\ \mathcal{S}(\Gamma \vdash C) &:= \text{dlv}(\mathcal{S}(\Gamma \Rightarrow C)) \\ \mathcal{S}(\Gamma \vdash a^-) &:= \sum_{(x:N) \in \Gamma} \text{coret}(x, \mathcal{S}(\Gamma[N] \vdash a^-)) \\ \mathcal{S}(\Gamma \vdash P) &:= \text{ret}(\mathcal{S}(\Gamma \vdash [P])) + \sum_{(x:N) \in \Gamma} \text{coret}(x, \mathcal{S}(\Gamma[N] \vdash P)) \end{aligned}$$

■ **Figure 3** Solution spaces for *LJP*.

► **Definition 2** (Solution spaces). *We define a forest  $\mathcal{S}(\sigma^\tau) \in \tau_{\Sigma}^{\text{co}}$  for every logical sequent  $\sigma^\tau$ , by simultaneous coinduction for all the  $\tau \in S$ . The definition is found in Fig. 3, where in the clauses for  $\mathcal{S}(\Gamma \mid a^+ \Rightarrow A)$  resp.  $\mathcal{S}(\Gamma \mid \downarrow N \Rightarrow A)$ , the variables  $z$  resp.  $x$  are supposed to be “fresh”.*

In the mentioned clauses, since the names of bound variables are considered as immaterial, there is no choice involved in this inversion phase of proof search, as is equally the case for  $\mathcal{S}(\Gamma \Rightarrow \cdot)$  – as should be expected from the deterministic way inversion rules are dealt with in a focused system like *LJP*.

► **Lemma 3** (Well-definedness of  $\mathcal{S}(\sigma)$ ). *For all logical sequents  $\sigma$ , the definition of  $\mathcal{S}(\sigma)$  indeed produces a forest.*

**Proof.** Well-definedness is not at stake concerning productivity of the definition since every corecursive call is under a constructor. As is directly seen in the definition, the syntactic categories are respected. Only the parity condition requires further thought. In Appendix A.2, we prove it by showing that all the “intermediary” corecursive calls to  $\mathcal{S}(\sigma)$  in the calculation of  $\mathcal{S}(\Gamma \vdash A)$  – which is the only case that applies inherited *e*-formation rules – lower the “weight” of the logical sequent, until a possible further call to some  $\mathcal{S}(\Gamma' \vdash A')$ . ◀

The members of a solution space are exactly the inhabitants of the sequent:

► **Proposition 4** (Adequacy of the coinductive representation). *For each  $\tau \in S$ , logical sequent  $\sigma^\tau$  and  $T$  of category  $\tau$ ,  $\text{mem}(T, \mathcal{S}(\sigma))$  iff  $\sigma(T)$  is provable in  $LJP$  (proof by induction on  $T$ ).*

The following definition is an immediate adaptation of the corresponding definition in [8].

► **Definition 5** (Inessential extension of contexts and R-stable sequents).

1.  $\Gamma \leq \Gamma'$  iff  $\Gamma \subseteq \Gamma'$  and  $|\Gamma| = |\Gamma'|$ , with  $|\Delta| := \{L \mid \exists y, (y : L) \in \Delta\}$  for an arbitrary context  $\Delta$  (where we write  $y$  for an arbitrary variable). That is,  $\Gamma \leq \Gamma'$  if  $\Gamma'$  only has extra bindings w. r. t.  $\Gamma$  that come with types that are already present in  $\Gamma$ .
2.  $\rho \leq \rho'$  iff for some  $\Gamma \leq \Gamma'$  and for some right-formula  $R$ ,  $\rho = (\Gamma \vdash R)$  and  $\rho' = (\Gamma' \vdash R)$ .

### 3.2 Search for inhabitants in $LJP$ , inductively

We are going to present a finitary version of  $LJP_\Sigma^{co}$  in the form of a system  $LJP_\Sigma^{\text{gfp}}$  of *finitary forests* that are again generically denoted by letter  $T$ . We are again making extensive use of our notational device introduced in Section 2. The letter  $f$  ranges over the function symbols in this specific view on  $LJP$ . Summation is added analogously as for  $LJP_\Sigma^{co}$ , and there are two more constructions for the category of expressions.

$$\frac{f : \tau_1, \dots, \tau_k \rightarrow \tau \quad \forall i. T_i : \tau_i}{f(T_1, \dots, T_k) : \tau} \quad \frac{\forall i. T_i : \tau \quad \tau \in \{v, s, e\}}{\sum_i^\tau T_i : \tau} \quad \frac{}{X^\rho : e} \quad \frac{T : e}{\text{gfp } X^\rho.T : e}$$

where  $X$  is assumed to range over a countably infinite set of *fixpoint variables* and  $\rho$  ranges over R-stable sequents, as said before. The conventions regarding sums  $\sum_i$  in the context of forests are also assumed for finitary forests. We stress that this is an all-inductive definition, and that w. r. t.  $LJP$ , the same finite summation mechanism is added as for  $LJP_\Sigma^{co}$ , but that the coinductive generation of stable expressions is replaced by formal fixed points whose binding and bound/free variables are associated with R-stable sequents  $\rho$  whose proof theory is our main aim.

Below are some immediate adaptations of definitions in our previous paper [8]. However, they are presented in the new uniform notation. Moreover, the notion of guardedness only arises with the now wider formulation of finitary forests that allows fixed-point formation for any finitary forest of the category of stable expression.

For a finitary forest  $T$ , let  $FPV(T)$  denote the set of freely occurring typed fixed-point variables in  $T$ , which can be described by structural recursion:

$$\begin{aligned} FPV(f(T_i)_i) &= FPV(\sum_i T_i) = \bigcup_i FPV(T_i) & FPV(X^\rho) &= \{X^\rho\} \\ FPV(\text{gfp } X^\rho.T) &= FPV(T) \setminus \{X^{\rho'} \mid \rho' \text{ R-stable sequent and } \rho \leq \rho'\} \end{aligned}$$

Notice the non-standard definition that considers  $X^{\rho'}$  also bound by  $\text{gfp } X^\rho$ , as long as  $\rho \leq \rho'$ . This special view on binding necessitates to study the following restriction on finitary forests: A finitary forest is called *well-bound* if, for any of its subterms  $\text{gfp } X^\rho.T$  and any free occurrence of  $X^{\rho'}$  in  $T$ ,  $\rho \leq \rho'$ .

► **Definition 6** (Interpretation of finitary forests as forests). *For a finitary forest  $T$ , the interpretation  $\llbracket T \rrbracket$  is a forest given by structural recursion on  $T$ :*

$$\begin{aligned} \llbracket f(T_1, \dots, T_k) \rrbracket &= f(\llbracket T_1 \rrbracket, \dots, \llbracket T_k \rrbracket) & \llbracket X^\rho \rrbracket &= \mathcal{S}(\rho) \\ \llbracket T_1 + \dots + T_k \rrbracket &= \llbracket T_1 \rrbracket + \dots + \llbracket T_k \rrbracket & \llbracket \text{gfp } X^\rho.T \rrbracket &= \llbracket T \rrbracket \end{aligned}$$

This definition may look too simple to handle the interpretation of bound fixed-point variables adequately, and in our previous paper [8] we called an analogous definition “simplified

$$\begin{aligned}
\mathcal{F}(\Gamma \vdash [a^+]; \Xi) &:= \sum_{(z:a^+) \in \Gamma} z & \mathcal{F}(\Gamma \vdash [\downarrow N]; \Xi) &:= \text{thunk}(\mathcal{F}(\Gamma \Longrightarrow N; \Xi)) \\
\mathcal{F}(\Gamma \vdash [\perp]; \Xi) &:= \mathbb{O}_v & \mathcal{F}(\Gamma \vdash [P_1 \vee P_2]; \Xi) &:= \sum_{i \in \{1,2\}} \text{in}_i^{P_3-i}(\mathcal{F}(\Gamma \vdash [P_i]; \Xi)) \\
\mathcal{F}(\Gamma \Longrightarrow a^-; \Xi) &:= \lceil \mathcal{F}(\Gamma \vdash a^-; \Xi) \rceil & \mathcal{F}(\Gamma \Longrightarrow P \supset N; \Xi) &:= \lambda \mathcal{F}(\Gamma \mid P \Longrightarrow N; \Xi) \\
\mathcal{F}(\Gamma \Longrightarrow \uparrow P; \Xi) &:= \lceil \mathcal{F}(\Gamma \vdash P; \Xi) \rceil & \mathcal{F}(\Gamma \Longrightarrow N_1 \wedge N_2; \Xi) &:= \langle \mathcal{F}(\Gamma \Longrightarrow N_i; \Xi) \rangle_i \\
\mathcal{F}(\Gamma[a^-] \vdash R; \Xi) &:= \text{if } R = a^- \text{ then nil else } \mathbb{O}_s \\
\mathcal{F}(\Gamma[\uparrow P] \vdash R; \Xi) &:= \text{cothunk}(\mathcal{F}(\Gamma \mid P \Longrightarrow R; \Xi)) \\
\mathcal{F}(\Gamma[P \supset N] \vdash R; \Xi) &:= \mathcal{F}(\Gamma \vdash [P]; \Xi) :: \mathcal{F}(\Gamma[N] \vdash R; \Xi) \\
\mathcal{F}(\Gamma[N_1 \wedge N_2] \vdash R; \Xi) &:= \sum_{i \in \{1,2\}} (i :: \mathcal{F}(\Gamma[N_i] \vdash R; \Xi)) \\
\mathcal{F}(\Gamma \mid a^+ \Longrightarrow A; \Xi) &:= z^{a^+} . \mathcal{F}(\Gamma, z : a^+ \vdash A; \Xi) && (z \text{ fresh}) \\
\mathcal{F}(\Gamma \mid \downarrow N \Longrightarrow A; \Xi) &:= x^N . \mathcal{F}(\Gamma, x : N \vdash A; \Xi) && (x \text{ fresh}) \\
\mathcal{F}(\Gamma \mid P_1 \vee P_2 \Longrightarrow A; \Xi) &:= [\mathcal{F}(\Gamma \mid P_i \Longrightarrow A; \Xi)]_i \\
\mathcal{F}(\Gamma \mid \perp \Longrightarrow A; \Xi) &:= \text{abort}^A \\
\mathcal{F}(\Gamma \vdash C; \Xi) &:= \text{dlv}(\mathcal{F}(\Gamma \Longrightarrow C; \Xi)) \\
\mathcal{F}(\Gamma \vdash a^-; \Xi) &:= \text{gfp } Y^\rho . \sum_{(x:N) \in \Gamma} \text{coret}(x, \mathcal{F}(\Gamma[N] \vdash a^-; \Xi, Y : \rho)) && (\rho = \Gamma \vdash a^-, Y \text{ fresh}) \\
\mathcal{F}(\Gamma \vdash P; \Xi) &:= \text{gfp } Y^\rho . \text{ret}(\mathcal{F}(\Gamma \vdash [P]; \Xi, Y : \rho)) && (\rho = \Gamma \vdash a^-, Y \text{ fresh}) \\
&& + \sum_{(x:N) \in \Gamma} \text{coret}(x, \mathcal{F}(\Gamma[N] \vdash P; \Xi, Y : \rho))
\end{aligned}$$

■ **Figure 4** All other cases of the finitary representation of solution spaces for *LJP*.

semantics” to stress that point. However, as in that previous paper, we can study those finitary forests for which the definition is “good enough” for our purposes of capturing solution spaces: we say a finitary forest  $T$  is *proper* if for any of its subterms  $T'$  of the form  $\text{gfp } X^\rho . T''$ , it holds that  $\llbracket T' \rrbracket = \mathcal{S}(\rho)$ .

To any free occurrence of an  $X^\rho$  in  $T$  is associated a *depth*: for this, we count the function symbols on the path from the occurrence to the root and notably do not count the binding operation of fixed-point variables and the sum operations. So,  $X^\rho$  only has one occurrence of depth 0 in  $X^\rho$ , likewise in  $\text{gfp } Y^{\rho'} . X^\rho$ .

We say a finitary forest  $T$  is *guarded* if for any of its subterms  $T'$  of the form  $\text{gfp } X^\rho . T''$ , it holds that every free occurrence in  $T''$  of a fixed-point variable  $X^{\rho'}$  that is bound by this fixed-point constructor has a depth of at least 1 in  $T''$ .

► **Definition 7** (Finitary solution spaces for *LJP*). *Let  $\Xi := \overrightarrow{X : \rho}$  be a vector of  $m \geq 0$  declarations ( $X_i : \rho_i$ ) where no fixed-point variable name occurs twice. The definition of the finitary forest  $\mathcal{F}(\sigma; \Xi)$  is as follows. If for some  $1 \leq i \leq m$ ,  $\rho_i =: (\Gamma_i \vdash R_i) \leq \sigma$  (i. e.,  $\sigma = \Gamma \vdash R_i$  and  $\Gamma_i \leq \Gamma$ ), then  $\mathcal{F}(\sigma; \Xi) = X_i^\sigma$ , where  $i$  is taken to be the biggest such index (notice that the produced  $X_i$  will not necessarily appear with the  $\rho_i$  associated to it in  $\Xi$ ). Otherwise,  $\mathcal{F}(\sigma; \Xi)$  is as displayed in Fig. 4. Then,  $\mathcal{F}(\sigma)$  denotes  $\mathcal{F}(\sigma; \Xi)$  with empty  $\Xi$ .*

Analogously to the similar result for implicational logic [7, Lemma 20], one can show that  $\mathcal{F}(\sigma; \Xi)$  is well-defined (the above recursive definition terminates) – some details are given in Appendix A.4. Notice that the “if-guard” in the above definition presupposes that  $\sigma$  is an R-stable sequent, hence for other forms of sequents, one necessarily has to apply the (mostly recursive) rules of Fig. 4.



$$\frac{P(\rho)}{\text{EF}_P(X^\rho)} \quad \frac{\forall i, \text{EF}_P(T_i)}{\text{EF}_P(f^*(T_i)_i)} \quad \frac{\text{EF}_P(T_j)}{\text{EF}_P(\sum_i T_i)} \quad \frac{\neg P(\rho)}{\text{NEF}_P(X^\rho)} \quad \frac{\text{NEF}_P(T_j)}{\text{NEF}_P(f^*(T_i)_i)} \quad \frac{\forall i, \text{NEF}_P(T_i)}{\text{NEF}_P(\sum_i T_i)}$$

■ **Figure 5**  $\text{EF}_P$  and  $\text{NEF}_P$  predicates.

► **Theorem 8** (Equivalence of representations for  $LJP$ ). *Let  $\sigma$  be a logical sequent and  $\Xi$  as in Def. 7. We have:*

1.  $\mathcal{F}(\sigma; \Xi)$  is guarded.
2.  $\mathcal{F}(\sigma; \Xi)$  is well-bound and  $\mathcal{F}(\sigma)$  is closed.
3.  $\mathcal{F}(\sigma; \Xi)$  is proper.
4.  $\llbracket \mathcal{F}(\sigma; \Xi) \rrbracket = \mathcal{S}(\sigma)$ ; hence the coinductive and the finitary representations are equivalent.

**Proof.** The proof is by structural induction on  $\mathcal{F}(\sigma; \Xi)$ . Items 1 and 2 are proved independently (the former is an easy induction, the latter on well-boundness uses in the two cases which generate  $\text{gfp}$ -constructions the lemma “if  $X^{\rho'}$  occurs free in  $\mathcal{F}(\sigma; \Xi)$ , then, for some  $\rho \leq \rho'$ ,  $X : \rho \in \Xi$ ”, also proved by structural induction on  $\mathcal{F}(\sigma; \Xi)$ , and from that lemma follows immediately that  $\mathcal{F}(\sigma)$  is closed). As in the proof of [8, Thm. 19], item 3 uses item 4, which can be proved independently, but some effort is saved if the two items are proved simultaneously. ◀

## 4 Deciding inhabitation problems in the polarized system $LJP$

Now we adapt to  $LJP$  our method [8] (until now only available for intuitionistic implication) to decide *type emptiness* (provability), and to decide *type finiteness* (only finitely many inhabitants). The presentation will look very different due to our notational device. Because of the wider notion of finitary forests that does not ensure guardedness through the grammar, some subtle technical refinements will be needed in the proofs (which will involve the Prop. 9 and are detailed in Appendix A.5). In the following, we write  $f^*$  to stand for a function symbol  $f$  or the prefix  $\text{gfp} X^\rho$ . of a finitary forest, the latter being seen as special unary function symbol.

### 4.1 Type emptiness

We consider complementary parameterized predicates on finitary forests  $\text{EF}_P(T)$  and  $\text{NEF}_P(T)$ , where the parameter  $P$  is a predicate on logical sequents. ( $P = \emptyset$  will be already an important case). The definition of the two predicates  $\text{EF}_P$  and  $\text{NEF}_P$  is inductive and presented in Fig. 5, although, as in [8], it is clear that they could equivalently be given by a definition by recursion over the term structure. Thus, the predicates  $\text{EF}_P$  and  $\text{NEF}_P$  are decidable if  $P$  is.

The following can be proven by routine induction on  $T$  (barely more than an application of de Morgan’s laws): for all  $T \in LJP_\Sigma^{\text{gfp}}$ ,  $\text{NEF}_P(T)$  iff  $\text{EF}_P(T)$  does not hold.

► **Proposition 9** (Finitary characterization).

1. If  $P \subseteq \text{exfin} \circ \mathcal{S}$  and  $\text{EF}_P(T)$  then  $\text{exfin}(\llbracket T \rrbracket)$ .
2. Let  $T \in LJP_\Sigma^{\text{gfp}}$  be well-bound, guarded and proper. If  $\text{NEF}_P(T)$  and for all  $X^\rho \in \text{FPV}(T)$ ,  $\text{exfin}(\mathcal{S}(\rho))$  implies  $P(\rho)$ , then  $\text{nofin}(\llbracket T \rrbracket)$ .
3. For any  $T \in LJP_\Sigma^{\text{gfp}}$  well-bound, guarded, proper and closed,  $\text{EF}_\emptyset(T)$  iff  $\text{exfin}(\llbracket T \rrbracket)$ .



$$\frac{P(\rho)}{\text{FF}_P(X^\rho)} \quad \frac{\forall i, \text{FF}_P(T_i)}{\text{FF}_P(f^*(T_i)_i)} \quad \frac{\text{NEF}_*(T_j)}{\text{FF}_P(f^*(T_i)_i)} \quad \frac{\forall i, \text{FF}_P(T_i)}{\text{FF}_P(\sum_i T_i)}$$

$$\frac{\neg P(\rho)}{\text{NFF}_P(X^\rho)} \quad \frac{\text{NFF}_P(T_j) \quad \forall i, \text{EF}_*(T_i)}{\text{NFF}_P(f^*(T_i)_i)} \quad \frac{\text{NFF}_P(T_j)}{\text{NFF}_P(\sum_i T_i)}$$

■ **Figure 6**  $\text{FF}_P$  and  $\text{NFF}_P$  predicates.

**Proof.**

1. is proved by induction on the predicate  $\text{EF}_P$  (or, equivalently, on  $T$ ). The base case for fixpoint variables needs the proviso on  $P$ , and all other cases are immediate by the induction hypothesis (notice the special case for  $f^*$  that is even simpler).
2. This needs a special notion of depth of observation for the truthfulness of `nofin` for forests. A more refined statement has to keep track of this observation depth in premise and conclusion, even taking into account the depth of occurrences of the bound fixed-point variables of  $T$ . This is presented with details in Appendix A.5.
3. For  $P = \emptyset$  resp. for closed  $T$ , the extra condition on  $P$  in part 1 resp. part 2 is trivially satisfied. We now use that `exfin` and `nofin` are complements, as are  $\text{NEF}_P$  and  $\text{EF}_P$ . ◀

► **Theorem 10** (Deciding the existence of inhabitants in  $LJP$ ). *A logical sequent  $\sigma$  of  $LJP$  is inhabited iff `exfin`( $\mathcal{S}(\sigma)$ ) iff  $\text{EF}_\emptyset(\mathcal{F}(\sigma))$ . Hence “ $\sigma$  is inhabited” is decided by deciding  $\text{EF}_\emptyset(\mathcal{F}(\sigma))$ . In other words, the inhabitation problem for  $LJP$  is decided by the computable predicate  $\text{EF}_\emptyset \circ \mathcal{F}$ .*

**Proof.** The first equivalence follows by Prop. 4 and `exfin` = `exfinext`. The second equivalence follows from Prop. 9.3, using all items of Theorem 8. Computability comes from computability of the recursive function  $\mathcal{F}$  and the equivalence of the inductively defined  $\text{EF}_\emptyset$  with a recursive procedure over the term structure of its argument. ◀

The theorem opens the way to using Prop. 9 with  $P := \text{EF}_\emptyset \circ \mathcal{F}$ . This is explored now, but will be needed only in the next subsection. The predicates  $\text{EF}_*$  and  $\text{NEF}_*$  on  $LJP_\Sigma^{\text{gfp}}$  are defined by  $\text{EF}_* := \text{EF}_P$  and  $\text{NEF}_* := \text{NEF}_P$  for  $P := \text{EF}_\emptyset \circ \mathcal{F}$  (which by Theorem 10 is equivalent to say  $P := \text{exfin} \circ \mathcal{S}$ ). We already know such  $P$  is decidable, hence, also  $\text{EF}_*$  and  $\text{NEF}_*$  are decidable. Additionally:

► **Lemma 11** (Sharp finitary characterization). *For all  $T \in LJP_\Sigma^{\text{gfp}}$ ,  $\text{EF}_*(T)$  iff `exfin`( $\llbracket T \rrbracket$ ).*

**Proof.** The direction from left to right follows immediately by Proposition 9.1. The other direction is equivalent to  $\text{NEF}_*(T)$  implies `nofin`( $\llbracket T \rrbracket$ ), which follows by an easy induction on the predicate  $\text{NEF}_*$  with the help of Theorem 10 in the base case  $T = X^\sigma$ . ◀

## 4.2 Type finiteness

Decision of type finiteness will be achieved by mimicking the development for deciding type emptiness, but will additionally require concepts and results from the latter. The finitary characterization of type finiteness is obtained through the complementary (parametrized) predicates  $\text{FF}_P$  and  $\text{NFF}_P$ , which are defined inductively on Fig. 6 and make use of the sharp finitary characterizations of emptiness and non-emptiness ( $\text{NEF}_*$  and  $\text{EF}_*$ ). That the two predicates are indeed complementary, i.e. that  $\text{FF}_P(T)$  iff  $\text{NFF}_P(T)$  does not hold, is again proved by routine induction on  $T$ .

► **Proposition 12** (Finitary characterization).

1. If  $P \subseteq \text{finfin} \circ \mathcal{S}$  and  $\text{FF}_P(T)$  then  $\text{finfin}(\llbracket T \rrbracket)$ .
2. Let  $T \in \text{LJP}_{\Sigma}^{\text{gfp}}$  be well-bound, guarded and proper. If  $\text{NFF}_P(T)$  and for all  $X^\rho \in \text{FPV}(T)$ ,  $\text{finfin}(\mathcal{S}(\rho))$  implies  $P(\rho)$ , then  $\text{inffin}(\llbracket T \rrbracket)$ .
3. For any  $T \in \text{LJP}_{\Sigma}^{\text{gfp}}$  well-bound, guarded, proper and closed,  $\text{FF}_\emptyset(T)$  iff  $\text{finfin}(\llbracket T \rrbracket)$ .

**Proof.** Each of the items follows analogously to the corresponding item of Proposition 9. In particular: 1 follows by induction on  $\text{FF}_P$ , and uses the fact  $\text{nofin} \subseteq \text{finfin}$ ; 2 needs a special notion of depth of observation for the truthfulness of  $\text{inffin}$  for forests, as detailed in Appendix A.5; 3 follows then by items 1 and 2, and uses the facts  $\text{finfin}$  and  $\text{inffin}$  are complements, as are  $\text{FF}_P$  and  $\text{NFF}_P$ . ◀

► **Theorem 13** (Deciding finiteness of inhabitants in  $\text{LJP}$ ). *A logical sequent  $\sigma$  of  $\text{LJP}$  has (only) finitely many inhabitants iff  $\text{finfin}(\mathcal{S}(\sigma))$  iff  $\text{FF}_\emptyset(\mathcal{F}(\sigma))$ . Hence “ $\sigma$  has (only) finitely many inhabitants” is decided by deciding  $\text{FF}_\emptyset(\mathcal{F}(\sigma))$ . In other words, the type finiteness problem for  $\text{LJP}$  is decided by the computable predicate  $\text{FF}_\emptyset \circ \mathcal{F}$ .*

**Proof.** The first equivalence follows by Prop. 4 and  $\text{finfin} = \text{finfinext}$ . The second equivalence follows from Prop. 12.3, using all items of Thm. 8. Computability comes from computability of the recursive function  $\mathcal{F}$ , decidability of  $\text{NEF}_*$ , and the equivalence of the inductively defined  $\text{FF}_\emptyset$  with a recursive procedure over the term structure of its argument. ◀

## 5 Applications to intuitionistic propositional logic with all connectives

One of the interests of polarized logic is that it can be used to analyze other logics [15]. This is also true of  $\text{LJP}$  and we illustrate it now, deriving algorithms for deciding the emptiness (provability) and the finiteness problems for  $\text{LJT}$  with all connectives. Such transfer of results from  $\text{LJP}$  will be immediate after the preparatory work that sets up an appropriate version of  $\text{LJT}$ , alongside with its embedding into  $\text{LJP}$ .

### 5.1 System $\text{LJT}$ of intuitionistic logic with all propositional connectives

The best known variant of the focused sequent calculus  $\text{LJT}$  for IPL is the one for implication only [10]. Variants including conjunction and disjunction as well can be found in [11, 3]. We present our own variant, still denoted  $\text{LJT}$ . *Formulas* of  $\text{LJT}$  are as follows:

$$\begin{aligned} \text{(intuitionistic formulas)} \quad A, B & ::= A \supset B \mid A \wedge B \mid R \\ \text{(right intuitionistic formulas)} \quad R & ::= a \mid \perp \mid A \vee B \end{aligned}$$

where  $a$  ranges over atoms, of which an infinite supply is assumed. A *positive* intuitionistic formula,  $P$ , is a non-atomic right intuitionistic formula.

*Proof terms* of  $\text{LJT}$  are organized in three syntactic categories as follows:

$$\begin{aligned} \text{(terms)} \quad t & ::= \lambda x^A. t \mid \langle t_1, t_2 \rangle \mid e \\ \text{(expressions)} \quad e & ::= xs \mid \text{in}_i^A(t) \\ \text{(spines)} \quad s & ::= \text{nil} \mid t :: s \mid i :: s \mid \text{abort}^R \mid [x_1^{A_1}. e_1, x_2^{A_2}. e_2] \end{aligned}$$

where  $i \in \{1, 2\}$ , and  $x$  ranges over a countable set of variables. We will refer to  $e_1$  and  $e_2$  in the latter form of spines as *arms*. Proof terms in any category are ranged over by  $T$ .

There are three forms of *sequents*,  $\Gamma \Longrightarrow t : A$  and  $\Gamma \vdash e : R$  and  $\Gamma[s : A] \vdash R$ , where, as usual,  $\Gamma$  is a context made of associations of variables with formulas. Therefore, a logical sequent  $\sigma$  in  $\text{LJT}$  may have three forms:  $\Gamma \Longrightarrow A$  and  $\Gamma \vdash R$  and  $\Gamma[A] \vdash R$ . The latter two

$$\begin{array}{c}
\frac{\Gamma, x : A \Longrightarrow t : B}{\Gamma \Longrightarrow \lambda x^A.t : A \supset B} \quad \frac{\Gamma \Longrightarrow t_i : A_i \text{ for } i = 1, 2}{\Gamma \Longrightarrow \langle t_1, t_2 \rangle : A_1 \wedge A_2} \quad \frac{\Gamma \vdash e : R}{\Gamma \Longrightarrow e : R} \quad \frac{\Gamma, x : A[s : A] \vdash R}{\Gamma, x : A \vdash xs : R} \\
\\
\frac{\Gamma \Longrightarrow t : A_i}{\Gamma \vdash \text{in}_i^{A_3-i}(t) : A_1 \vee A_2} \quad i \in \{1, 2\} \quad \frac{\Gamma \Longrightarrow t : A \quad \Gamma[s : B] \vdash R}{\Gamma[t :: s : A \supset B] \vdash R} \quad \frac{}{\Gamma[\text{nil} : a] \vdash a} \\
\\
\frac{}{\Gamma[\text{abort}^R : \perp] \vdash R} \quad \frac{\Gamma[s : A_i] \vdash R}{\Gamma[i :: s : A_1 \wedge A_2] \vdash R} \quad i \in \{1, 2\} \quad \frac{\Gamma, x_i : A_i \vdash e_i : R \text{ for } i = 1, 2}{\Gamma[x_1^{A_1}.e_1, x_2^{A_2}.e_2 : A_1 \vee A_2] \vdash R}
\end{array}$$

■ **Figure 7** Typing rules of *LJT*.

$$\begin{array}{l}
(A \supset B)^* = \downarrow A^* \supset B^* \quad (A \vee B)^\circ = \downarrow A^* \vee \downarrow B^* \\
(A \wedge B)^* = A^* \wedge B^* \quad \perp^\circ = \perp \\
P^* = \uparrow P^\circ \quad a^\circ = a^- \\
a^* = a^\circ \\
\\
(\lambda x^A.t)^* = \lambda(x^{A^*}.\text{DLV}(t^*)) \quad (xs)^* = \text{coret}(x, s^*) \\
\langle t_1, t_2 \rangle^* = \langle t_1^*, t_2^* \rangle \quad \text{in}_i^A(t)^* = \text{ret}(\text{in}_i^{\downarrow A^*}(\text{thunk}(t^*))) \\
e^* = \lceil e^{*\neg} \rceil, \text{ if } e \text{ is atomic} \\
e^* = \lceil e^* \rceil, \text{ if } e \text{ is positive} \\
\\
\text{nil}^* = \text{nil} \quad (\text{abort}^R)^* = \text{cothunk}(\text{abort}^{R^\circ}) \\
(t :: s)^* = \text{thunk}(t^*) :: s^* \quad [x_1^{A_1}.e_1, x_2^{A_2}.e_2]^* = \text{cothunk}([x_1^{A_1^*}.e_1^*, x_2^{A_2^*}.e_2^*]) \\
(i :: s)^* = i :: s^*
\end{array}$$

■ **Figure 8** Negative translation.

forms require a right formula to the right of the turnstile. The full definition of the typing rules of *LJT* is given in Fig. 7. As for *LJP*, the annotations guarantee that there is at most one formula that can replace the placeholders in  $\Gamma \Longrightarrow t : \cdot$ ,  $\Gamma \vdash e : \cdot$  and  $\Gamma[s : A] \vdash \cdot$ .

The characteristic feature of the design of *LJT* is the restriction of the type of spines to right formulas. Since the type of `nil` is atomic, spines have to be “long”; and the arms of spines cannot be lambda-abstractions nor pairs, which is enforced by restricting the arms of spines to be expressions, rather than general terms: this is the usefulness of separating the class of expressions from the class of terms. In the typing rules, the restriction to right formulas is generated at the *select rule* (the typing rule for `xs`); and the long form is forced by the identity axiom (the typing rule for `nil`) because it applies to atoms only.

We could not find in the literature the restriction of cut-free LJT we consider here, but Ferrari and Fiorentini [9] consider a presentation of IPL that enforces a similar use of right formulas, in spite of being given in natural deduction format and without proof terms. It is easy to equip this natural deduction system with proof terms and map it into *LJT*: the technique is fully developed in [4] for polarized logic, but goes back to [3]. Since the just mentioned system [9] is complete for provability, so is *LJT*.

System *LJT* can be embedded in *LJP*. We define the *negative translation*  $(\cdot)^* : LJT \rightarrow LJP$  in Fig. 8, comprising a translation of formulas and a translation of proof terms.

The translation of formulas uses an auxiliary translation of right intuitionistic formulas  $R$ :  $R^\circ$  is a right formula (and specifically,  $P^\circ$  is a positive formula). An intuitionistic formula  $A$  is mapped to a negative formula  $A^*$ , hence the name of the translation. At the level of proof terms: terms (resp. spines, expressions) are mapped to terms (resp. spines, stable expressions). Definitions like  $e^* = \ulcorner e^* \urcorner$  are meaningful if one thinks of the left  $e$  as being tagged with the injection into terms. Use is made of the derived construction  $\text{DLV}(t)$ , a stable expression of  $LJP$ , defined by  $\text{DLV}(\ulcorner e \urcorner) = e$  and  $\text{DLV}(t) = \text{dlv}(t)$  otherwise. Its derived typing rule is that  $\Gamma \vdash \text{DLV}(t) : N$  follows from  $\Gamma \Longrightarrow t : N$ .

The translation of proof terms is defined for legal proof terms in  $LJT$  only:  $T$  is *legal* if every expression  $e$  occurring in  $T$  is either atomic or positive; an expression  $xs$  is *atomic* (resp. *positive*) if  $s$  is atomic (resp. positive), whereas an injection is positive; and a spine  $s$  is *atomic* (resp. *positive*) if every “leaf” of  $s$  is `nil` or `aborta` (resp. an injection or `abortP`). Only when translating a legal  $T$  can we apply the definition of  $(\cdot)^*$  to  $e$  as a term.

Formally, the inductive definition of atomic and positive spines is as follows:

- `nil` is atomic; `aborta` is atomic; if  $s$  is atomic, then  $t :: s$  and  $i :: s$  are atomic; if, for each  $i = 1, 2$ ,  $e_i = y_i s_i$  and  $s_i$  is atomic, then  $[x_1^{A_1}.e_1, x_2^{A_2}.e_2]$  is atomic.
- `abortP` is positive; if  $s$  is positive, then  $t :: s$  and  $i :: s$  are positive; if, for each  $i = 1, 2$ ,  $e_i = y_i s_i$  and  $s_i$  is positive, or  $e_i = \text{in}_i^A(t)$ , then  $[x_1^{A_1}.e_1, x_2^{A_2}.e_2]$  is positive.

Suppose  $\Gamma[s : A] \vdash R$  is derivable. If  $R = a$  (resp.  $R = P$ ) then  $s$  is atomic (resp. positive). Hence any typable proof term of  $LJT$  is legal. Moreover, if  $\Gamma \vdash e : R$  then if  $e$  is atomic,  $R = a$  and if  $e$  is positive,  $R = P$ .

The negative translation is easily seen to be injective. In order to state other properties of the translation, we define the logical  $LJP$  sequent  $\sigma^*$  for every logical  $LJT$  sequent  $\sigma$ :  $(\Gamma \Longrightarrow A)^* = (\Gamma^* \Longrightarrow A^*)$  and  $(\Gamma \vdash R)^* = (\Gamma^* \vdash R^\circ)$  and  $(\Gamma[A] \vdash R)^* = (\Gamma^*[A^*] \vdash R^\circ)$ .

► **Proposition 14 (Soundness).** *For all  $T = t, e, s$  in  $LJT$ : if  $\sigma(T)$  is derivable in  $LJT$  then  $\sigma^*(T^*)$  is derivable in  $LJP$ .*

**Proof.** By simultaneous induction on derivations for  $\sigma(T)$ . ◀

For the converse property (faithfulness), we need to understand better the image of the negative translation, which we will call the *\*-fragment* of  $LJP$ . Consider the following subclass of formulas in  $LJP$ :

$$\begin{aligned} (\text{*}-\text{formulas}) \quad M, N & ::= a^- \mid \uparrow P \mid \downarrow N \supset M \mid N \wedge M \\ (\text{positive } \circ\text{-formulas}) \quad P & ::= \perp \mid \downarrow N \vee \downarrow M \end{aligned}$$

The positive  $\circ$ -formulas are separated because they are useful to define  $\circ$ -formulas  $R$ , which are either atoms  $a^-$  or positive  $\circ$ -formulas  $P$ . A  $*$ -formula  $N$  is a negative formula; a positive  $\circ$ -formula  $P$  is a positive formula; a  $\circ$ -formula  $R$  is a right formula. The negative translation, at the level of formulas, is a bijection from intuitionistic formulas to  $*$ -formulas, from positive intuitionistic formulas to positive  $\circ$ -formulas; and from right intuitionistic formulas to  $\circ$ -formulas. The respective inverse maps are denoted  $|\cdot|$ : they just erase the polarity shifts and the minus sign from atoms.

If we are interested in deriving in  $LJP$  logical sequents of the form  $\sigma^*$  only, then some obvious cuts can be applied to the grammar of proof terms of  $LJP$ , yielding the following grammar  $\mathcal{G}$  of *\*-proof terms*:

$$\begin{aligned} (\text{*}-\text{terms}) \quad t & ::= [e] \mid \ulcorner e \urcorner \mid \lambda(x^N.e) \mid \langle t_1, t_2 \rangle \\ (\text{*}-\text{spines}) \quad s & ::= \text{nil} \mid \text{cothunk}(\text{abort}^R) \mid \text{cothunk}([x_1^{N_1}.e_1, x_2^{N_2}.e_2]) \mid \text{thunk}(t) :: s \mid i :: s \\ (\text{*}-\text{expressions}) \quad e & ::= \text{dlv}(t) \mid \text{ret}(\text{in}_i^P(\text{thunk}(t))) \mid \text{coret}(x, s) \end{aligned}$$

Here the type annotations range over formulas in the  $*$ -fragment.

A *legal*  $*$ -proof term is one where  $\text{dlv}(t)$  is only allowed as the body of a  $\lambda$ -abstraction. Legal expressions are generated by a restricted variant of the grammar above:  $\text{dlv}(t)$  is forbidden as a  $*$ -expression *per se*, but, as a compensation, we introduce a second form of  $\lambda$ -abstraction,  $\lambda(x^N.\text{dlv}(t))$ .

There is a *forgetful* map from legal  $*$ -terms (resp. legal  $*$ -spines, legal  $*$ -expressions) to terms (resp. spines, expressions) of *LJT* that essentially erases term decorations, and is given in detail in Appendix A.6. The negative translation only generates legal  $*$ -proof terms; and, since the negative translation is just a process of decoration, the forgetful map is left inverse to it:  $|T^*| = T$ .

► **Proposition 15 (Faithfulness).** *For all  $T$  in LJP: if  $\sigma^*(T)$  is derivable in LJP, then  $T$  is legal and  $\sigma(|T|)$  is derivable in LJT and  $|T|^* = T$ .*

**Proof.** By simultaneous induction on  $T = t, s, r$  as generated by the grammar  $\mathcal{G}$  above. ◀

By faithfulness and injectivity of the negative translation, the implications in Proposition 14 are in fact equivalences. Moreover:

► **Corollary 16 (Reduction of counting and inhabitation problems).**

1. *There is a bijection between the set of those  $T \in LJT$  such that  $\sigma(T)$  is derivable in LJT and the set of those  $T' \in LJP$  such that  $\sigma^*(T')$  is derivable in LJP.*
2. *There is  $T \in LJT$  such that  $\sigma(T)$  is derivable in LJT iff there is  $T' \in LJP$  such that  $\sigma^*(T')$  is derivable in LJP.*

**Proof.** We prove the first item. The negative translation is the candidate for the bijection. Due to Proposition 14, it maps from the first set to the second. We already observed that the translation is injective. Proposition 15 guarantees that the translation is also surjective. The second item is an immediate consequence of the first. ◀

## 5.2 Deciding emptiness and finiteness in *LJT*

The “extraction” of the two decision procedures is immediate. Both procedures will be given by the composition of two recursive functions: first,  $\mathcal{F}$  calculates the finitary representation of the full solution space; second, recursing on the structure of this representation, a predicate ( $\text{EF}_\emptyset$  or  $\text{FF}_\emptyset$ ) is decided.

**Emptiness.** Given  $\sigma$  in *LJT*:  $\sigma$  is inhabited in *LJT* iff  $\sigma^*$  is inhabited in *LJP* (Cor. 16); iff  $\text{exfin}(\mathcal{S}(\sigma^*))$  (Prop. 4 and  $\text{exfin} = \text{exfinext}$ ); iff  $\text{EF}_\emptyset(\mathcal{F}(\sigma^*))$  (Thm. 10). The obtained algorithm is thus  $\text{EF}_\emptyset(\mathcal{F}(\sigma^*))$ . Recall from Subsec. 4.1 that, although predicate  $\text{EF}_\emptyset$  is given inductively, it can be equivalently given by recursion over the structure of finitary forests.

**Finiteness.** Given  $\sigma$  in *LJT*:  $\sigma$  has finitely many inhabitants in *LJT* iff  $\sigma^*$  has finitely many inhabitants in *LJP* (Cor. 16); iff  $\text{finfin}(\mathcal{S}(\sigma^*))$  (Prop. 4 and  $\text{finfin} = \text{finfinext}$ ); iff  $\text{FF}_\emptyset(\mathcal{F}(\sigma^*))$  (Thm. 13). The obtained algorithm is thus  $\text{FF}_\emptyset(\mathcal{F}(\sigma^*))$ . Again, here, we should think of  $\text{FF}_\emptyset$  as given by its recursive description.

**Discussion.** Complexity issues are not (yet) a concern of “coinductive proof search”. So far we privileged a conceptual approach, where the representation of the search space is separated from its analysis. This separation of concerns is reflected in the architecture of our decision procedures, given as the composition of  $\mathcal{F}$  with a recursive predicate adequate for the specific problem at hand. This organization is modular, with  $\mathcal{F}(\sigma^*)$  being reused, as we

move our attention to a different decision problem; but it is not optimized, because knowing the particular predicate we want to compose  $\mathcal{F}$  with, in general, suggests simplifications. Nevertheless, here are some comparisons with algorithms from the literature.

It is well-known that provability in full IPL is PSPACE-complete. In particular, [13] establishes a space bound  $\mathcal{O}(n \log n)$  for this problem based on a *contraction-free* proof system. Of course, this kind of efficiency cannot be expected from a naive implementation of our decision method above, as it would first fully compute through  $\mathcal{F}$  the finitary representation of the solution space. An immediate optimization would be to compute with  $\mathcal{F}$  lazily, and interleave it with the structural decision algorithm for  $\text{EF}_\emptyset$ , thus avoiding the explicit construction of the solution space. We wonder if such kind of optimization leads to a decision algorithm for provability in full IPL within PSPACE. Note that, if our sole interest was decision of provability, it would be better to start from variants of *LJT* like the systems *MJ<sup>Hist</sup>* [12] or *Nbu* [9], which, in particular, block application of context-expanding rules if the formulas to be added are already present in the context (like in *total discharge convention*). However, neither the latter systems nor contraction-free systems give an appropriate basis to address questions related to the *full set of normal proofs/inhabitants*.

The work of [18] is the only one we are aware of that deals with a question of type finiteness for full IPL (but  $\perp$  is not included). That work considers a cut-free *LJT*-presentation of IPL close to ours, but allowing more proofs, due to unrestricted RHS in its *contraction* rule (recall our version of *LJT* imposes an atom or disjunction on the RHS when a formula from the context is selected to the “focus”). The work [18] uses graphs to represent the search space, and such graphs are guaranteed to be finite only in the case where contexts are sets, in other words, when the total discharge convention is assumed. The decision of type finiteness is then based on traversal of this finite graph structure and exhaustive checking for the absence of “cyclic proof structures”. In our case, the decision comes by computing the result of the function  $\mathcal{F}$ , which gives the finitary forest representing the solution space, and then by deciding by a simple structural recursion the predicate  $\text{FF}_\emptyset$  on such a forest; but, again, one may compute with  $\mathcal{F}$  lazily and interleave it with the structural decision of  $\text{FF}_\emptyset$ . It should be noted that decision of type finiteness in [18] is part of more general algorithms that count the number of inhabitants of a type. In our case, counting of inhabitants is done by a function defined by structural recursion on finitary forests. This worked fine for the implicational fragment of *LJT* [8], and we anticipate no major obstacles in extending the idea to full *LJT*.

## 6 Final remarks

We have shown that “coinductive proof search” extends to polarized intuitionistic logic [17, 4]: the basic result about the equivalence of the coinductive and finitary representation of solution spaces is obtained, as well as decidability of some predicates (one of which is provability) through recursive predicates defined over the finitary syntax.

In the presence of disjunction, focused proofs fail to be *canonical* – in e. g. [16] (Subsec. 1.7) it is observed that types with a unique canonical inhabitant may have infinitely many focused inhabitants. So, we stress again, our algorithms for type finiteness refer to the finiteness of the number of *focused* inhabitants (which are all the inhabitants according to the specific proof systems considered in this paper). The next challenge is to try our approach with the even more sophisticated systems [16] that capture canonical inhabitants, and for that we find it useful to deal with *LJP* first.

But the study of  $LJP$  has other uses, as a platform to study other logics. We illustrated this view with  $LJT$ , a focused proof system for intuitionistic logic, by means of the negative translation of  $LJT$  into  $LJP$ . Variants of this translation were previously mentioned or sketched [19, 15], here we give a full treatment as a translation between languages of proof terms. By composing the properties of the negative translation with the results about polarized logic, we extract results about proof search in  $LJT$  (including notably disjunction).

Our negative translation is reminiscent of Girard’s translation of intuitionistic logic into linear logic. The latter translation may be seen as underlying other translations in the literature – see [1] for a study that involves polarized *linear* logic and even covers cut-elimination (our setting is cut-free and linearity plays no role). We also worked out a positive translation of cut-free  $LJQ$  [2] into  $LJP$ , but have no space to show it. This opens the way to the study of inhabitation problems relative to call-by-value  $\lambda$ -terms, and for that, the results obtained here about  $LJP$  will be reused.

In the context of intuitionistic implication, we obtained in [6] decidability of problems involving the concept of solution rather than inhabitant (including the problem of termination of proof search). As further future work, we plan to extend to  $LJP$  such decidability results.

---

## References

- 1 Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus. In Cristian S. Calude and Vladimiro Sassone, editors, *Proceedings of Theoretical Computer Science - 6th IFIP TC 1/WG 2.2 International Conference, TCS 2010, Brisbane, Australia, September 20-23, 2010*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 165–181. Springer, 2010.
- 2 Roy Dyckhoff and Stéphane Lengrand. Call-by-value lambda-calculus and LJQ. *J. Log. Comput.*, 17(6):1109–1134, 2007.
- 3 Roy Dyckhoff and Luís Pinto. A permutation-free sequent calculus for intuitionistic logic. Technical report, St Andrews University Computer Science Research Report CS/96, August 1996.
- 4 José Espírito Santo. The polarized  $\lambda$ -calculus. In Vivek Nigam and Mário Florido, editors, *11th Workshop on Logical and Semantic Frameworks with Applications, LSFA 2016, Porto, Portugal, January 1, 2016*, volume 332 of *Electronic Notes in Theoretical Computer Science*, pages 149–168. Elsevier, 2016. doi:10.1016/j.entcs.2017.04.010.
- 5 José Espírito Santo, Ralph Matthes, and Luís Pinto. A coinductive approach to proof search. In David Baelde and Arnaud Carayol, editors, *Proceedings Workshop on Fixed Points in Computer Science, FICS 2013, Turino, Italy, September 1st, 2013*, volume 126 of *EPTCS*, pages 28–43, 2013.
- 6 José Espírito Santo, Ralph Matthes, and Luís Pinto. Decidability of several concepts of finiteness for simple types. *Fundam. Inform.*, 170(1-3):111–138, 2019.
- 7 José Espírito Santo, Ralph Matthes, and Luís Pinto. A coinductive approach to proof search through typed lambda-calculi. *CoRR*, abs/1602.04382v3, 2020. arXiv:1602.04382v3.
- 8 José Espírito Santo, Ralph Matthes, and Luís Pinto. Inhabitation in simply-typed lambda-calculus through a lambda-calculus for proof search. *Mathematical Structures in Computer Science*, 29:1092–1124, 2019. Also found at HAL through <https://hal.archives-ouvertes.fr/hal-02360678v1>. doi:10.1017/S0960129518000099.
- 9 Mauro Ferrari and Camillo Fiorentini. Goal-oriented proof-search in natural deduction for intuitionistic propositional logic. *J. Autom. Reasoning*, 62(1):127–167, 2019.
- 10 Hugo Herbelin. A  $\lambda$ -calculus structure isomorphic to a Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1995.



- 11 Hugo Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de  $\lambda$ -termes et comme calcul de stratégies gagnantes*. Ph.D. thesis, University Paris 7, 1995.
- 12 Jacob Howe. *Proof Search Issues in Some Non-Classical Logics*. Ph.D. thesis, University of St. Andrews, available as University of St. Andrews Research Report CS/99/1, 1998.
- 13 Jörg Hudelmaier. An  $o(n \log n)$ -space decision procedure for intuitionistic propositional logic. *J. Log. Comput.*, 3(1):63–75, 1993.
- 14 Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.*, 19(4):377–414, 2006.
- 15 Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logic. *Theor. Comput. Sci.*, 410:4747–4768, 2009.
- 16 Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 243–255. ACM, 2015.
- 17 Robert J. Simmons. Structural focalization. *ACM Trans. Comput. Log.*, 15(3):21:1–21:33, 2014.
- 18 J. B. Wells and Boris Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*, volume 3573 of LNCS, pages 262–277. Springer, 2004.
- 19 Noam Zeilberger. Focusing and higher-order abstract syntax. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 359–369. ACM, 2008.

## A Appendix with some more technical details

### A.1 On the characterization of predicates on forests in Section 3.1

#### ► Lemma 17.

1. Given a forest  $T$ ,  $\text{exfin}(T)$  iff  $\text{nofin}(T)$  does not hold.
2. Given a forest  $T$ ,  $\text{finfin}(T)$  iff  $\text{inffin}(T)$  does not hold.

**Proof.** Both items are plain instances of the generic result in the style of De Morgan’s laws that presents inductive predicates as complements of coinductive predicates, by a dualization operation on the underlying clauses. ◀

The following lemma shows that the predicates  $\text{exfin}$  and  $\text{finfin}$  correspond to the intended meaning in terms of the finite extensions. Additionally, the lemma shows that the negation of  $\text{exfin}$  resp.  $\text{finfin}$  holds exactly for the forests which have no finite members resp. for the forests which have infinitely many finite members.

#### ► Lemma 18 (Coinductive characterization). Given a forest $T$ ,

1.  $\text{exfin}(T)$  iff  $\mathcal{E}_{\text{fin}}(T)$  is non-empty, i. e.,  $\text{exfin} = \text{exfinext}$  as sets of forests;
2.  $\text{finfin}(T)$  iff  $\mathcal{E}_{\text{fin}}(T)$  is finite, i. e.,  $\text{finfin} = \text{finfinext}$  as sets of forests.

**Proof.** Item 1 follows directly from the fact:  $\text{exfin}(T)$  iff  $\text{mem}(T_0, T)$  for some  $T_0$ . The left to right implication is proved by induction on  $\text{exfin}$ . (Recall  $\text{exfin}$  is a predicate on forests, but is defined inductively.) The right to left implication can be proved via the equivalent statement “for all  $T_0$ ,  $\text{mem}(T_0, T)$  implies  $\text{exfin}(T)$ ”, which follows by induction on  $LJP$  proof terms  $T_0$ . For the case of membership in sums, it is necessary to decompose them (thanks to priority 1) until membership in an expression  $f(T_i)_i$  is reached so that the argument for the first inductive clause of membership applies. Item 2 follows analogously, but also uses the fact  $\text{nofin} = \text{nofinext}$ , which is an immediate consequence of item 1 and Lemma 17. ◀



## A.2 On well-definedness of infinitary representation in Section 3.1

This section is dedicated to the proof of Lemma 3.

It remains to check the parity condition. As mentioned in the main text, this comes from the observation that all the “intermediary” corecursive calls to  $\mathcal{S}(\sigma)$  in the calculation of  $\mathcal{S}(\Gamma \vdash A)$  – which is the only case that applies inherited  $e$ -formation rules – lower the “weight” of the logical sequent, until a possible further call to some  $\mathcal{S}(\Gamma' \vdash A')$ .

► **Definition 19** (weight). *Weight of a formula:  $w(\perp, a^+) := 0$ ,  $w(a^-) := 1$ , and for composite formulas, add the weights of the components and add the following for the extra symbols:  $w(\downarrow, \wedge) := 0$ ,  $w(\vee) := 1$ ,  $w(\uparrow) := 2$ ,  $w(\supset) := 3$ . Then  $w(N) \geq 1$  and  $w(P) \geq 0$ .*

*Weight of context  $\Gamma$ : the sum of the weights of all the formulas associated with the variables.*

*Weight of logical sequent:  $w(\Gamma \vdash A) := w(\Gamma) + w(A)$ ,  $w(\Gamma \Longrightarrow N) := w(\Gamma) + w(N) - 1 \geq 0$ .  $w(\Gamma \vdash [P]) := w(\Gamma) + w(P)$ ,  $w(\Gamma|P \Longrightarrow A) := w(\Gamma) + w(P) + w(A) + 1$ ,  $w(\Gamma[N] \vdash R) := w(\Gamma) + w(N) + w(R)$ . Then for all  $\sigma$ ,  $w(\sigma) \geq 0$ .*

In preparation of Section A.4, we even show the following more general statement:

► **Lemma 20.** *Every direct corecursive call in the definition of  $\mathcal{S}(\sigma)$  to some  $\mathcal{S}(\sigma')$  for neither  $\sigma$  nor  $\sigma'$   $R$ -stable sequents lowers the weight of the logical sequent.*

**Proof.** We have to show the following inequalities:

$w(\Gamma \vdash C) > w(\Gamma \Longrightarrow C)$  (the rule introducing  $\text{dlv}(\cdot)$  is easy to overlook but not needed for the proof of Lemma 3): this is why  $\cdot \Longrightarrow \cdot$  has to weigh less

$w(\Gamma|a^+ \Longrightarrow A) > w(\Gamma, z : a^+ \vdash A)$ : this is why  $\cdot| \cdot \Longrightarrow \cdot$  has to weigh more (and variable names must not enter the weight of contexts  $\Gamma$ )

$w(\Gamma|\downarrow N \Longrightarrow A) > w(\Gamma, x : N \vdash A)$ :  $w(\downarrow) = 0$  suffices

$w(\Gamma \vdash [\downarrow N]) > w(\Gamma \Longrightarrow N)$ :  $w(\downarrow) = 0$  suffices

$w(\Gamma \vdash [P_1 \vee P_2]) > w(\Gamma \vdash [P_i])$ : trivial since  $w(\vee) > 0$

$w(\Gamma \Longrightarrow a^-) > w(\Gamma \vdash a^-)$  is not to be shown (and is wrong) since we hit the class of  $R$ -stable sequents

$w(\Gamma \Longrightarrow \uparrow P) > w(\Gamma \vdash P)$ : this works since  $\uparrow$  weighs more (given that  $\cdot \Longrightarrow \cdot$  weighs less), but this inequation is not needed either

$w(\Gamma \Longrightarrow P \supset N) > w(\Gamma|P \Longrightarrow N)$ : since both logical sequent weights are unfavourably modified, the weight of  $\supset$  has to be so high

$w(\Gamma \Longrightarrow N_1 \wedge N_2) > w(\Gamma \Longrightarrow N_i)$ : since  $w(N_{3-i}) \geq 1$

$w(\Gamma[\uparrow P] \vdash R) > w(\Gamma|P \Longrightarrow R)$ : this is why  $\uparrow$  has to weigh more (given that  $\cdot| \cdot \Longrightarrow \cdot$  weighs more)

$w(\Gamma[P \supset N] \vdash R) > w(\Gamma \vdash [P])$  and  $> w(\Gamma[N] \vdash R)$ : both are trivial since  $w(\supset) > 0$

$w(\Gamma[N_1 \wedge N_2] \vdash R) > w(\Gamma[N_i] \vdash R)$ : since  $w(N_{3-i}) \geq 1$

$w(\Gamma|P_1 \vee P_2 \Longrightarrow A) > w(\Gamma|P_i \Longrightarrow A)$ : trivial since  $w(\vee) > 0$  ◀

It is clear that this lemma guarantees the parity condition for all  $\mathcal{S}(\sigma)$ .

## A.3 On forest transformation for inessential extensions in Section 3.1

If  $\rho = (\Gamma \vdash R)$  and  $\rho' = (\Gamma' \vdash R)$ , then the result  $[\rho'/\rho]T$  of the *decontraction operation* applied to  $T$  is defined to be  $[\Gamma'/\Gamma]T$ , with the latter given as follows:

$$\begin{aligned}
[\Gamma'/\Gamma]f(T_1, \dots, T_k) &= f([\Gamma'/\Gamma]T_1, \dots, [\Gamma'/\Gamma]T_k) \quad \text{for } f \text{ neither } z \text{ nor } \text{coret}(x, \cdot) \\
[\Gamma'/\Gamma]\sum_i T_i &= \sum_i [\Gamma'/\Gamma]T_i \\
[\Gamma'/\Gamma]z &= z && \text{if } z \notin \text{dom}(\Gamma) \\
[\Gamma'/\Gamma]z &= \sum_{z' \in D_z} z' && \text{if } z \in \text{dom}(\Gamma) \\
[\Gamma'/\Gamma]\text{coret}(x, s) &= \text{coret}(x, [\Gamma'/\Gamma]s) && \text{if } x \notin \text{dom}(\Gamma) \\
[\Gamma'/\Gamma]\text{coret}(x, s) &= \sum_{x' \in D_x} \text{coret}(x', [\Gamma'/\Gamma]s) && \text{if } x \in \text{dom}(\Gamma)
\end{aligned}$$

■ **Figure 9** Corecursive equations for definition of decontraction.

► **Definition 21** (Decontraction). *Let  $\Gamma \leq \Gamma'$ . For a forest  $T$  of  $LJP_{\Sigma}^{\text{co}}$ , the forest  $[\Gamma'/\Gamma]T$  of  $LJP_{\Sigma}^{\text{co}}$  is defined by corecursion in Fig. 9, where, for  $w \in \text{dom}(\Gamma)$ ,*

$$D_w := \{w\} \cup \{w' : (w' : \Gamma(w)) \in (\Gamma' \setminus \Gamma)\} .$$

*In other words, the occurrences of variables (in the syntactic way they are introduced in the forests) are duplicated for all other variables of the same type that  $\Gamma'$  has in addition.*

► **Lemma 22** (Solution spaces and decontraction). *Let  $\rho \leq \rho'$ . Then  $\mathcal{S}(\rho') = [\rho'/\rho]\mathcal{S}(\rho)$ .*

**Proof.** Analogous to the proof for implicational logic [7]. Obviously, the decontraction operation for forests has to be used to define decontraction operations for all forms of logical sequents (analogously to the R-stable sequents, where only  $\Gamma$  varies). Then, the coinductive proof is done simultaneously for all forms of logical sequents. ◀

#### A.4 On termination of finitary representation in Section 3.2

Definition 7 contains recursive equations that are not justified by calls to the same function for “smaller” sequents, in particular not for the rules governing R-stable sequents as first argument. We mentioned that the proof of termination of an analogous function for implicational logic [7, Lemma 20] can be adapted to establish also termination of  $\mathcal{F}(\sigma; \Xi)$  for any valid arguments. Here, we substantiate this claim.

The difficulty comes from the rich syntax of  $LJP$ , so that the “true” recursive structure of  $\mathcal{F}(\rho; \Xi)$  – for R-stable sequents that spawn the formal fixed points – gets hidden through intermediary recursive calls with the other forms of logical sequents. However, we will now argue that all those can be seen as plainly auxiliary since they just decrease the “weight” of the problem to be solved.

► **Lemma 23.** *Every direct recursive call in the definition of  $\mathcal{F}(\sigma; \Xi)$  to some  $\mathcal{F}(\sigma'; \Xi')$  for neither  $\sigma$  nor  $\sigma'$  R-stable sequents lowers the weight of the first argument.*

**Proof.** This requires to check the very same inequations as in the proof of Lemma 20. ◀

The message of the lemma is that the proof search through all the other forms of logical sequents (including the form  $\Gamma \vdash C$ ) is by itself terminating. Of course, this was to be expected. Otherwise, we could not have “solved” them by a recursive definition in  $\mathcal{F}$  where only R-stable sequents ask to be hypothetically solved through fixed-point variables.

The present argument comes from an analysis that is deeply connected to *LJP*, it has nothing to do with an abstract approach of defining (infinitary or finitary) forests. As seen directly in the definition of  $\mathcal{F}$ , only by cycling finitely through the  $\text{dlv}(\cdot)$  construction is the context  $\Gamma$  extended in the arguments  $\sigma$  to  $\mathcal{F}$ . And the context of the last fixed-point variable in  $\Xi$  grows in lockstep.

It is trivial to observe that all the formula material of the right-hand sides lies in the same subformula-closed sets (see [7]) as the left-hand sides (in other words, the logical sequents in the recursive calls are taken from the same formula material, and there is no reconstruction whatsoever).

Therefore, the previous proof for the implicational case [7, Lemma 20] can be carried over without substantial changes. What counts are recursive calls with first argument an  $R$ -stable sequent for the calculation when the first argument is an  $R$ -stable sequent. In the implicational case, these “big” steps were enforced by the grammar for finitary forests (and the logical sequents  $\Gamma \vdash R$  had even only atomic  $R$  there, but this change is rather irrelevant for the proof (instead of counting atoms, one has to count  $R$  formulas for getting the measure, but this does not affect finiteness of it). The preparatory steps in the proof of [7, Lemma 20] are also easily adapted, where the  $\Gamma$  part of the first argument to  $\mathcal{F}$  takes the role of the context  $\Gamma$  in that proof.

## A.5 Completing the proofs of Props. 9.2 and 12.2 with extra concepts

First we prove Prop. 9.2. For this, we need an auxiliary concept with which we can formulate a refinement of that proposition. From the refinement, we eventually get Prop. 9.2.

We give a sequence of approximations from above to the coinductive predicate  $\text{nofin}$  whose intersection characterizes the predicate. The index  $n$  is meant to indicate to which observation depth of  $T$  we can guarantee that  $\text{nofin}(T)$  holds. For this purpose, we do not take into account the summation operation as giving depth. We present the notion as a simultaneous inductive definition.

$$\frac{}{\text{nofin}_0(T)} \quad \frac{\text{nofin}_n(T_j)}{\text{nofin}_{n+1}(f(T_i)_i)} \text{ for some } j \quad \frac{\forall i. \text{nofin}_{n+1}(T_i)}{\text{nofin}_{n+1}(\sum_i T_i)}$$

A guarantee up to observation depth 0 does not mean that the root symbol is suitable but the assertion is just void. Going through a function symbol requires extra depth. The child has to be fine up to a depth that is one less. As announced, the summation operation does not provide depth, which is why this simultaneous inductive definition cannot be seen as a definition of  $\text{nofin}_n$  by recursion over the index  $n$ .

By induction on the inductive definition, one can show that  $\text{nofin}_n$  is antitone in  $n$ , i. e., if  $\text{nofin}_{n+1}(T)$  then  $\text{nofin}_n(T)$ .

► **Lemma 24** (Closure under decontraction of each  $\text{nofin}_n$ ). *Let  $\rho \leq \rho'$  and  $n \geq 0$ . For all forests  $T$ ,  $\text{nofin}_n(T)$  implies  $\text{nofin}_n([\rho'/\rho]T)$ .*

**Proof.** By induction on the inductive definition – we profit from not counting sums as providing depth. ◀

► **Lemma 25** (Inductive characterization of absence of members). *Given a forest  $T$ . Then,  $\text{nofin}(T)$  iff  $\text{nofin}_n(T)$  for all  $n$ .*

**Proof.** From left to right, this is by induction on  $n$ . One decomposes (thanks to priority 1) the sums until one reaches finitely many expressions  $f(T_i)_i$  to which the induction hypothesis applies. From right to left, one proves coinductively  $R \subseteq \text{nofin}$ , for  $R := \{T : \forall n \geq 0, \text{nofin}_n(T)\}$ .

For example, in the case  $T = f(T_i)_i \in R$ , this amounts to showing  $T_j \in R$  for some  $j$ . The assumption  $\text{nofin}_1(f(T_i)_i)$  already implies the existence of at least one  $T_j$ . The proof is then indirect: if for all  $i$  we would have  $T_i \notin R$ , then, for each  $i$ , there would be an  $n_i$  s. t.  $\neg \text{nofin}_{n_i}(T_i)$ , and letting  $m$  be the maximum of these  $n_i$ 's,  $\neg \text{nofin}_m(T_i)$  by antitonicity; hence we would have  $\neg \text{nofin}_{m+1}(T)$ , but  $T \in R$ .  $\blacktriangleleft$

For  $T \in LJP_{\Sigma}^{\text{gfp}}$ , we write  $\mathcal{A}_n(T)$  for the following assumption: For every free occurrence of some  $X^\rho$  in  $T$  (those  $X^\rho$  are found in  $FPV(T)$ ) such that  $\neg P(\rho)$ , there is an  $n_0$  with  $\text{nofin}_{n_0}(\mathcal{S}(\rho))$  and  $d + n_0 \geq n$  for  $d$  the *depth* of the occurrence in  $T$  as defined earlier, where sums and generations of fixed points do not contribute to depth.

Notice that, trivially  $n' \leq n$  and  $\mathcal{A}_n(T)$  imply  $\mathcal{A}_{n'}(T)$ .

► **Lemma 26** (Ramification of Proposition 9.2). *Let  $T \in LJP_{\Sigma}^{\text{gfp}}$  be well-bound, proper and guarded and such that  $\text{NEF}_P(T)$  holds. Then, for all  $n \geq 0$ ,  $\mathcal{A}_n(T)$  implies  $\text{nofin}_n(\llbracket T \rrbracket)$ .*

**Proof.** By induction on the predicate  $\text{NEF}_P$  (which can also be seen as a proof by induction on finitary forests).

**Case  $T = X^\rho$ .** Then  $\llbracket T \rrbracket = \mathcal{S}(\rho)$ . Assume  $n \geq 0$  such that  $\mathcal{A}_n(T)$ . By inversion,  $\neg P(\rho)$ , hence, since  $X^\rho \in FPV(T)$  at depth 0 in  $T$ , this gives  $n_0 \geq n$  with  $\text{nofin}_{n_0}(\mathcal{S}(\rho))$ . Since  $\text{nofin}_m$  is antitone in  $m$ , we also have  $\text{nofin}_n(\llbracket T \rrbracket)$ .

**Case  $T = \text{gfp } X^\rho.T_1$ .**  $\text{NEF}_P(T)$  comes from  $\text{NEF}_P(T_1)$ . Let  $N := \llbracket T \rrbracket = \llbracket T_1 \rrbracket$ . As  $T$  is proper,  $N = \mathcal{S}(\rho)$ . We do the proof by a side induction on  $n$ . The case  $n = 0$  is trivial. So assume  $n = n' + 1$  and  $\mathcal{A}_n(T)$  and that we already know that  $\mathcal{A}_{n'}(T)$  implies  $\text{nofin}_{n'}(\mathcal{S}(\rho))$ . We have to show  $\text{nofin}_n(\mathcal{S}(\rho))$ , i. e.,  $\text{nofin}_n(\llbracket T_1 \rrbracket)$ . We use the main induction hypothesis on  $T_1$  with the same index  $n$ . Hence, it suffices to show  $\mathcal{A}_n(T_1)$ . Consider any free occurrence of some  $Y^{\rho'}$  in  $T_1$  such that  $\neg P(\rho')$ . We have to show that there is an  $n_0$  with  $\text{nofin}_{n_0}(\mathcal{S}(\rho'))$  and  $d + n_0 \geq n$  for  $d$  the depth of the occurrence in  $T_1$ .

*First sub-case:* the considered occurrence is also a free occurrence in  $T$ . Since we disregard fixed-point constructions for depth,  $d$  is also the depth in  $T$ . Because of  $\mathcal{A}_n(T)$ , we get an  $n_0$  as desired.

*Second sub-case:* the remaining case is with  $Y = X$  and, since  $T$  is well-bound,  $\rho \leq \rho'$ . As remarked before,  $\mathcal{A}_n(T)$  gives us  $\mathcal{A}_{n'}(T)$ . The side induction hypothesis therefore yields  $\text{nofin}_{n'}(\mathcal{S}(\rho))$ . By closure of  $\text{nofin}_n$  under decontraction, we get  $\text{nofin}_{n'}([\rho'/\rho]\mathcal{S}(\rho))$ , but that latter forest is  $\mathcal{S}(\rho')$  by Lemma 22. By guardedness of  $T$ , this occurrence of  $X^{\rho'}$  has depth  $d \geq 1$  in  $T_1$ . Hence,  $d + n' \geq 1 + n' = n$ .

**Case  $T = f(T_1, \dots, T_k)$  with a proper function symbol  $f$ .** Assume  $n \geq 0$  such that  $\mathcal{A}_n(T)$ . There is an index  $j$  such that  $\text{NEF}_P(T)$  comes from  $\text{NEF}_P(T_j)$ . Assume  $n \geq 0$  such that  $\mathcal{A}_n(T)$ . We have to show that  $\text{nofin}_n(\llbracket T \rrbracket)$ . This is trivial for  $n = 0$ . Thus, assume  $n = n' + 1$ . We are heading for  $\text{nofin}_{n'}(\llbracket T_j \rrbracket)$ . We use the induction hypothesis on  $T_j$  (even with this smaller index  $n'$ ). Therefore, we are left to show  $\mathcal{A}_{n'}(T_j)$ . Consider any free occurrence of some  $X^\rho$  in  $T_j$  such that  $\neg P(\rho)$ , of depth  $d$  in  $T_j$ . This occurrence is then also a free occurrence in  $T$  of depth  $d + 1$  in  $T$ . From  $\mathcal{A}_n(T)$ , we get an  $n_0$  with  $\text{nofin}_{n_0}(\mathcal{S}(\rho))$  and  $d + 1 + n_0 \geq n$ , hence with  $d + n_0 \geq n'$ , hence  $n_0$  is as required for showing  $\mathcal{A}_{n'}(T_j)$ .

**Case  $T = \sum_i T_i$ .**  $\text{NEF}_P(T)$  comes from  $\text{NEF}_P(T_i)$  for all  $i$ . Assume  $n \geq 0$  such that  $\mathcal{A}_n(T)$ . We have to show that  $\text{nofin}_n(\llbracket T \rrbracket)$ . This is trivial for  $n = 0$ . Thus, assume  $n = n' + 1$  and fix some index  $i$ . We have to show  $\text{nofin}_n(\llbracket T_i \rrbracket)$ . We use the induction hypothesis on  $T_i$  (with the

same index  $n$ ). Therefore, we are left to show  $\mathcal{A}_n(T_i)$ . Consider any free occurrence of some  $X^\rho$  in  $T_i$  such that  $\neg P(\rho)$ , of depth  $d$  in  $T_i$ . This occurrence is then also a free occurrence in  $T$  of depth  $d$  in  $T$ . From  $\mathcal{A}_n(T)$ , we get an  $n_0$  with  $\text{nofin}_{n_0}(\mathcal{S}(\rho))$  and  $d + n_0 \geq n$ , hence  $n_0$  is as required for showing  $\mathcal{A}_n(T_i)$ . (Of course, it is important that sums do not count for depth in finitary terms if they do not count for the index of the approximations to  $\text{nofin}$ . Therefore, this proof case is so simple.)  $\blacktriangleleft$

We return to Prop. 9.2:

**Proof.** Let  $T \in LJP_{\Sigma}^{\text{gfp}}$  be well-bound, proper and guarded, assume  $\text{NEF}_P(T)$  and that for all  $X^\rho \in FPV(T)$ ,  $\text{exfin}(\mathcal{S}(\rho))$  implies  $P(\rho)$ . We have to show  $\text{nofin}(\llbracket T \rrbracket)$ . By Lemma 25 it suffices to show  $\text{nofin}_n(\llbracket T \rrbracket)$  for all  $n$ . Let  $n \geq 0$ . By the just proven refinement, it suffices to show  $\mathcal{A}_n(T)$ . Consider any free occurrence of some  $X^\rho$  in  $T$  such that  $\neg P(\rho)$ , of depth  $d$  in  $T$ . By contraposition of the assumption on  $FPV(T)$  and by the complementarity of  $\text{nofin}$  and  $\text{exfin}$ , we have  $\text{nofin}(\mathcal{S}(\rho))$ , hence by Lemma 25  $\text{nofin}_n(\mathcal{S}(\rho))$ , and  $d + n \geq n$ , as required for  $\mathcal{A}_n(T)$ .  $\blacktriangleleft$

The whole development above can be replayed to prove Prop. 12.2. Now, the required auxiliary concept is  $\text{inffin}_n$ , which gives a sequence of approximations to the coinductive predicate  $\text{inffin}$ :

$$\frac{}{\text{inffin}_0(T)} \quad \frac{\text{inffin}_n(T_j) \quad \forall i. \text{exfin}(T_i)}{\text{inffin}_{n+1}(f(T_i)_i)} \text{ for some } j \quad \frac{\text{inffin}_{n+1}(T_j)}{\text{inffin}_{n+1}(\sum_i T_i)} \text{ for some } j$$

► **Lemma 27** (Antitonicity and closedness under decontraction of  $\text{inffin}_n$ ). *Given a forest  $T$  and  $n \geq 0$ ,*

1. *if  $\text{inffin}_{n+1}(T)$  then  $\text{inffin}_n(T)$ ;*
2. *for any  $\rho \leq \rho'$ ,  $\text{inffin}_n(T)$  implies  $\text{inffin}_n([\rho'/\rho]T)$ .*

**Proof.** Both items 1 and 2 follow by induction on the inductive definition of  $\text{inffin}_n$ , and 2 uses closedness of  $\text{exfin}$  under decontraction.  $\blacktriangleleft$

► **Lemma 28** (Inductive characterization of finiteness of members). *Given a forest  $T$ ,  $\text{inffin}(T)$  iff  $\text{inffin}_n(T)$  for all  $n \geq 0$ .*

**Proof.** Analogously to the proof of Lemma 25, the left to right direction follows by induction on  $n$ , and the right to left direction follows by proving coinductively  $R \subseteq \text{inffin}$ , for  $R := \{T : \forall n \geq 0, \text{inffin}_n(T)\}$ .  $\blacktriangleleft$

For  $T \in LJP_{\Sigma}^{\text{gfp}}$ , now  $\mathcal{A}_n(T)$  will stand for the assumption: For every free occurrence of some  $X^\rho$  in  $T$  (those  $X^\rho$  are found in  $FPV(T)$ ) such that  $\neg P(\rho)$ , there is an  $n_0$  with  $\text{inffin}_{n_0}(\mathcal{S}(\rho))$  and  $d + n_0 \geq n$  for  $d$  the depth of the occurrence in  $T$  as defined earlier, where sums and generations of fixed points do not contribute to depth. (The only change w. r. t. the definition of  $\mathcal{A}_n(T)$  above is the replacement of  $\text{nofin}_{n_0}$  by  $\text{inffin}_{n_0}$ .)

► **Lemma 29** (Ramification of Proposition 12.2). *Let  $T \in LJP_{\Sigma}^{\text{gfp}}$  be well-bound, proper and guarded and such that  $\text{NFF}_P(T)$  holds. Then, for all  $n \geq 0$ ,  $\mathcal{A}_n(T)$  implies  $\text{inffin}_n(\llbracket T \rrbracket)$ .*

**Proof.** By induction on the predicate  $\text{NFF}_P$ . All cases for  $T$  follow analogously to the corresponding cases of Lemma 26, with the help of Lemma 27. The case  $T = f(T_i)_i$  uses additionally Lemma 11.  $\blacktriangleleft$

Finally, Prop. 12.2 follows from Lemma 29 (in lockstep with the proof of Prop. 9.2 from Lemma 26) thanks to Lemma 28.

### A.6 Details on the forgetful map in Section 5

The *forgetful* map from legal \*-terms (resp. legal \*-spines, legal \*-expressions) to terms (resp. spines, expressions) of *LJT* is as follows:

$$\begin{array}{ll}
 |\lambda(x^N.\text{dlv}(t))| & = \lambda x^{|N|}.|t| & |\text{nil}| & = \text{nil} \\
 |\lambda(x^N.e)| & = \lambda x^{|N|}.|e| & |\text{cothunk}(\text{abort}^R)| & = \text{abort}^{|R|} \\
 |\langle t_1, t_2 \rangle| & = \langle |t_1|, |t_2| \rangle & \text{cothunk}([x_1^{|N_1|}.e_1, x_2^{|N_2|}.e_2]) & = [x_1^{|N_1|}.|e_1|, x_2^{|N_2|}.|e_2|] \\
 |\ulcorner e \urcorner| & = |e| & |\text{thunk}(t) :: s| & = |t| :: |s| \\
 |[e]| & = |e| & |i :: s| & = i :: |s| \\
 |\text{coret}(x, s)| & = x|s| & |\text{ret}(\text{in}_i^P(\text{thunk}(t)))| & = \text{in}_i^{|P|}(|t|)
 \end{array}$$

# Synthetic Completeness for a Terminating Seligman-Style Tableau System

Asta Halkjær From   

Technical University of Denmark, Kongens Lyngby, Denmark

---

## Abstract

Hybrid logic extends modal logic with nominals that name worlds. Seligman-style tableau systems for hybrid logic divide branches into blocks named by nominals to achieve a local proof style. We present a Seligman-style tableau system with a formalization in the proof assistant Isabelle/HOL. Our system refines an existing system to simplify formalization and we claim termination from this relationship. Existing completeness proofs that account for termination are either analytic or based on translation, but synthetic proofs have been shown to generalize to richer logics and languages. Our main result is the first synthetic completeness proof for a terminating hybrid logic tableau system. It is also the first formalized completeness proof for any hybrid logic proof system.

**2012 ACM Subject Classification** Theory of computation → Modal and temporal logics

**Keywords and phrases** Hybrid logic, Seligman-style tableau, synthetic completeness, Isabelle/HOL

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.5

**Supplementary Material** *Model (Isabelle/HOL formalization in the Archive of Formal Proofs (4900+ lines))*: [https://isa-afp.org/entries/Hybrid\\_Logic.html](https://isa-afp.org/entries/Hybrid_Logic.html)

archived at `swh:1:cnt:5a830ce17c70be797b343d9078bf19c43b0f2145`

**Acknowledgements** We thank Patrick Blackburn, Thomas Bolander, Torben Braüner, Klaus Frovin Jørgensen and Jørgen Villadsen for discussions and Lars Hupel, Jasmin Blanchette and the anonymous reviewers for comments on the paper.

## 1 Introduction

Hybrid logic increases the expressiveness of modal logic by adding a special sort of propositional symbol called *nominals* to the syntax. In regular modal logic we can only reference worlds indirectly through the modalities, but nominals, that are true at exactly one world, name worlds explicitly. A nominal  $i$  gives rise to the satisfaction operator  $@_i$  that states what world a formula is true “at.” These features make hybrid logic well suited for applications like temporal logic [3], description logic [5] and epistemic logics for social networks [24].

There are many proof systems for classical hybrid logic [4] and we focus on tableau systems in the following. Early work relied on loop checks to ensure termination [10] but Bolander and Blackburn introduced a calculus that guarantees finite branches through local restrictions [9]. Their completeness proof is *analytic*, meaning that they reason about open branches directly. Blackburn et al. [4] introduced the Seligman-style [25] system ST with a more local proof style than previous systems. Jørgensen et al. [21] later introduced a *synthetic* completeness proof for ST and showed that it scales with extensions to the logic. The synthetic approach involves reasoning about maximal consistent sets and their properties [13, 26] and this also opens the way for other developments, notably interpolation results [1].

Blackburn et al. [4] restricted ST into the terminating  $ST^*$  but showed completeness by translation from the system by Bolander and Blackburn [9]. The synthetic completeness proof for ST relies on a symmetry in branches that neither terminating system has. We present system  $ST^A$ , a refinement of  $ST^*$  suitable for formalization, which is formalized in the simple type theory of Isabelle/HOL [23]. Its proof of completeness fills a gap as the first synthetic completeness proof for a terminating tableau system for hybrid logic. It is also the first standalone completeness proof for a terminating Seligman-style system and, to our knowledge, the first formalization of any proof system for hybrid logic.



© Asta Halkjær From;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 5; pp. 5:1–5:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



The formalization provides absolute trust in the correctness of the completeness proof and serves as a companion to this paper, where the proofs can be seen in full detail.

Our system closely resembles  $ST^*$  but with restrictions that are simpler to formalize and we argue for termination based on this relationship. Formalizing termination remains future work since we want a direct proof, not one based on translation. Blanchette [6] gives an overview of efforts to formalize the metatheory of logical calculi and provers in Isabelle.

Other formalizations of hybrid logic itself exist. Doczkal and Smolka [12] formalized hybrid logic with nominals in constructive type theory using the proof assistant Coq. They gave algorithmic proofs of small model theorems and computational decidability of satisfiability, validity, and equivalence of formulas. In Isabelle/HOL, Linker [22] formalized the semantic embedding of a spatio-temporal multi-modal logic with a hybrid logic-inspired *at*-operator.

Our work is classical but hybrid logic also has a constructive variant. Braüner and de Paiva [11] defined intuitionistic hybrid logic, and a natural deduction system, and Galmiche and Salhi [19] showed its decidability via a sequent calculus. Jia and Walker [20] interpreted modal proofs as distributed programs with nominals denoting places in the network.

We formalized the synthetic completeness of  $ST$  with some of the simpler  $ST^*$  restrictions required for termination in our MSc thesis [17]. A short paper by From et al. [14] briefly described an even earlier version of the formalization and we mentioned the present completeness proof in a short presentation at Advances in Modal Logic 2020 [18].

The paper continues as follows. First, we give the syntax and semantics of basic hybrid logic (Section 2). We introduce the proof system, corresponding rule restrictions and some consequences (Section 3). Next, we show a number of properties of the system that are useful for the completeness proof (Section 4). After that, we prove completeness of the system and show how our proof relates to existing work (Section 5). We then show how  $ST^A$  relates to  $ST^*$  and argue for our choice of restrictions. From this relationship we claim that  $ST^A$  must be terminating by sketching a possible translation (Section 6). We briefly discuss some points about the formalization (Section 7) and conclude with future work (Section 8).

## 2 Syntax and Semantics

The well-formed formulas of the basic hybrid logic are given by the following grammar, where we use  $p$  as a propositional symbol and  $i, j, k, a, b$  for nominals.

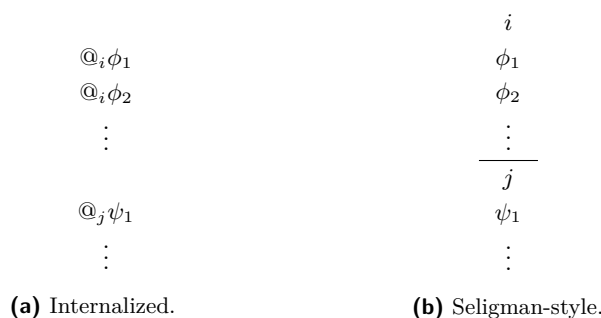
$$\phi, \psi ::= p \mid i \mid \neg\phi \mid \phi \vee \psi \mid \diamond\phi \mid @_i\phi$$

The  $\diamond$  operator is the usual possibility modality and  $@_i$  is the aforementioned *satisfaction operator*. A formula of the form  $@_i\phi$  is called a *satisfaction statement*.

We interpret the language on Kripke models  $\mathfrak{M} = (W, R, V)$ . The frame  $(W, R)$  consists of a non-empty set of worlds  $W$  and a binary accessibility relation  $R$  between them.  $V$  is the valuation of propositional symbols. An *assignment*  $g$  maps nominals to elements of  $W$ ; if  $g(i) = w$  we say that nominal  $i$  *denotes*  $w$ . Formula satisfiability is defined as follows:

$$\begin{array}{lll} \mathfrak{M}, g, w \models p & \text{iff} & w \in V(p) \\ \mathfrak{M}, g, w \models i & \text{iff} & g(i) = w \\ \mathfrak{M}, g, w \models \neg\phi & \text{iff} & \mathfrak{M}, g, w \not\models \phi \\ \mathfrak{M}, g, w \models \phi \vee \psi & \text{iff} & \mathfrak{M}, g, w \models \phi \text{ or } \mathfrak{M}, g, w \models \psi \\ \mathfrak{M}, g, w \models \diamond\phi & \text{iff} & \text{for some } w', wRw' \text{ and } \mathfrak{M}, g, w' \models \phi \\ \mathfrak{M}, g, w \models @_i\phi & \text{iff} & \mathfrak{M}, g, g(i) \models \phi \end{array}$$





■ **Figure 1** Internalized and Seligman-style tableau branches.

### 3 Our Seligman-Style Tableau System

Our proof system of choice is tableau. In tableau we decompose an initial set of *root* formulas into a tree structure and show unsatisfiability by reaching a contradiction on each branch. This is called “closing” the branch and a branch that cannot be closed remains “open.” If we can close every branch that emerges then the root formulas have a *closing tableau*.

A hybrid logic formula is true relative to a given world and our proof system must handle this. Internalized tableau systems, as depicted in Figure 1a, encode the information in every formula on the branch by working exclusively with satisfaction statements. We follow instead the Seligman style [25] adapted to tableau systems by Blackburn et al. [4]. Here, the information is attached to a group of formulas at once by dividing the branch into *blocks* as depicted in Figure 1b. The first formula on each block is ensured to be a nominal and called the *opening nominal*. It denotes the world that the formulas on the block are true at. We occasionally call a block’s opening nominal its “type” and use the following shorthands:

► **Definition 1** ( $\phi$  at  $i$ ). *If a formula  $\phi$  occurs on a block with opening nominal  $i$ , then we say that  $\phi$  occurs “on an  $i$ -block” or simply that  $\phi$  occurs “at  $i$ .”*

#### 3.1 Proof System

Figure 2 gives our tableau rules. We give the rule output below the *vertical* lines and the rule input above them. The opening nominal of the latest, *current*, block is given below the horizontal line. Above each input formula we write the opening nominal of the block it occurs on. When a rule has multiple input we write these pairs side by side. Any formula on the current block may be used as input under the same restrictions on opening nominals.

► **Example 2.** Consider the  $(\neg\neg)$  rule: if  $\neg\neg\phi$  occurs on an  $a$ -block and the current block is an  $a$ -block, then  $\phi$  is a legal extension of the branch. The intuition for the **Nom** rule is that the current opening nominal  $a$  occurs on a  $b$ -block so nominals  $a$  and  $b$  must denote the same world and it is sound to copy  $\phi$  from  $b$  to  $a$ . The  $(\diamond)$  rule witnesses its input formula,  $\diamond\phi$ , with a fresh *witnessing nominal*  $i$  by producing an accessibility formula,  $\diamond i$ , and a satisfaction statement,  $@_i\phi$ , saying that  $\phi$  holds at the reachable world denoted by  $i$ .

► **Remark 3.** In the internalized system, cf. Figure 1a, we may work on a formula prefixed by  $@_i$  one moment and one prefixed by  $@_k$  the next. The Seligman-style blocks give rise to a more local proof style by delegating this perspective switch, e.g. from  $i$  to  $k$ , to the **GoTo** rule that opens a new block with corresponding opening nominal.

The soundness proof for  $ST^A$  follows existing work [4, 14] (cf. the formalization).

$\frac{a}{\phi \vee \psi}$	$\frac{a}{\neg(\phi \vee \psi)}$	$\frac{a}{\neg\neg\phi}$	$\frac{a}{\diamond\phi}$	$\frac{a \quad a}{\neg\diamond\phi \quad \diamond i}$
$\frac{a}{/ \quad \backslash}$	$\frac{a}{ }$	$\frac{a}{ }$	$\frac{a}{ }$	$\frac{a}{ }$
$\phi \quad \psi$	$\neg\phi$	$\phi$	$\diamond i$	$\neg@_i\phi$
$\neg\psi$	$\neg\psi$	$@_i\phi$	$@_i\phi$	$@_i\phi$
( $\vee$ )	( $\neg\vee$ )	( $\neg\neg$ )	( $\diamond$ ) <sup>1</sup>	( $\neg\diamond$ )
$\frac{b \quad b}{a \quad \phi}$	$\frac{b \quad b}{\phi \quad \neg\phi}$	$\frac{b}{@_a\phi}$	$\frac{b}{@_a\phi}$	$\frac{b}{\neg@_a\phi}$
$\frac{a}{ }$	$\frac{a}{ }$	$\frac{a}{ }$	$\frac{a}{ }$	$\frac{a}{ }$
$i$	$\phi$	$\times$	$\phi$	$\neg\phi$
GoTo <sup>2</sup>	Nom	Closing	(@)	( $\neg$ @)

<sup>1</sup>  $i$  is fresh and  $\phi$  is not a nominal.

<sup>2</sup>  $i$  is not fresh.

■ **Figure 2** Our Seligman-style tableau system ST<sup>A</sup>.

### 3.2 Restrictions for Termination

Besides the side conditions, we need to impose the following four restrictions on the system to ensure that we eventually run out of applicable rules (inspired by Blackburn et al. [4]):

- S1** The output of a non-GoTo rule must include a formula new to the current block type.
- S2** The ( $\diamond$ ) rule can only be applied to input  $\diamond\phi$  on an  $a$ -block if  $\diamond\phi$  is not already witnessed at  $a$  by formulas  $\diamond i$  and  $@_i\phi$  for some *witnessing nominal*  $i$ .
- S3** We associate *potential*, a natural number  $n$ , with each line in the tableau. GoTo must decrement the number, the other rules increment it and we may start from any amount.
- S4** We parameterize the proof system by a fixed set of nominals  $A$  and impose the following:
  - a. The nominal introduced by the ( $\diamond$ ) rule is not in  $A$ .
  - b. For any nominal  $i$ , Nom only applies to a formula  $\phi = i$  or  $\phi = \diamond i$  when  $i \in A$ .

Restrictions S1 and S2 prevent us from applying the same rule to the same input repeatedly. We motivate restriction S3 by the following examples and restriction S4 in Section 3.3.

► **Example 4.** In Figure 3a we prove the validity of  $\neg@_i\phi \vee @_i\phi$  by constructing a closing tableau for its negation. We start from potential 0 in the fourth column. Notice how regular rule applications build up potential that is then discharged to open a new block on line 5.

► **Example 5.** In Figure 3b we start from the unsatisfiable formula  $@_i\neg i$  and potential  $n$ . Restriction S3 prevents infinite applications of GoTo and eventually forces us to make progress (or we might get stuck if no rules apply).

► **Remark 6.** The choice of a fresh opening nominal for the root block ensures that we do not close the branch because of an interplay between the formula itself and the opening nominal (imagine starting from  $\neg i$  on a block with opening nominal  $i$ ).

Given restrictions S3 and S4 we say that a branch has a closing tableau *with respect to* a set of allowed nominals  $A$  and potential  $n$ . We also introduce the following shorthand:

► **Definition 7** (Allowed  $\phi$ ). *A formula  $\phi$  is allowed by  $A$  if it meets condition S4b.*

$ \begin{array}{llll} 0. & a & & \\ 1. & \neg(\neg @_i \phi \vee @_i \phi) & [0] & \\ 2. & \neg\neg @_i \phi & (\neg\vee) 1 & [1] \\ 3. & \neg @_i \phi & (\neg\vee) 1 & [1] \\ 4. & @_i \phi & (\neg\neg) 2 & [2] \\ 5. & \frac{i}{\text{GoTo}} & [1] & \\ 6. & \neg\phi & (\neg@) 3 & [2] \\ 7. & \phi & (@) 4 & [3] \\ & \times & &  \end{array} $	$ \begin{array}{llll} 0. & a & & \\ 1. & \frac{@_i \neg i}{i} & \text{GoTo} & [n] \\ 2. & \frac{i}{i} & \text{GoTo} & [n-1] \\ 3. & \frac{i}{i} & \text{GoTo} & [n-2] \\ & \vdots & \vdots & \vdots \\ & \vdots & \vdots & \vdots \\ n+1. & \frac{i}{i} & \text{GoTo} & [0] \\ n+2. & \neg i & (@) 1 & [1] \\ & \times & &  \end{array} $
---	---

(a) Building up potential.

(b) Running out of potential.

■ **Figure 3** Two examples of potential.

### 3.3 Nominal Asymmetry

See Blackburn et al. [4] for why a restriction like S4 is needed. They conclude:

We ... have to enforce some control on the “direction” we allow the copying of formulas, so that we can establish a decreasing length argument. It is OK to copy a formula true at a nominal  $i$  to a nominal  $j$  if  $j$  generated  $i$ , but not if  $i$  generated  $j$  [4].

Essentially, we need to ensure that blocks of generated nominals contain strictly smaller formulas, so that any chain of them eventually terminates. It is the  $(\diamond)$  rule that *generates* a fresh nominal  $i$  by producing the formulas  $\diamond i$  and  $@_i \phi$ . Only **GoTo** can decompose either formula into the raw nominal  $i$ . Our restriction S4a ensures  $i \notin A$  so by S4b, nominal  $i$  cannot be copied to another block. Thus, unlike root nominals, the nominals generated by  $(\diamond)$  can only appear raw as opening nominals. Since **Nom** requires the opening nominal of the current block to appear on its own, formulas can only be copied *to* blocks with  $(\diamond)$ -generated opening nominals, not *from* them. This matches the quote. It also shows how generated nominals are treated differently, causing a “nominal asymmetry.”

We revisit termination in Section 6. For now, note that the *fixed* set  $A$  frees us from formalizing the *growing* set of nominals generated by  $(\diamond)$ . The reader may imagine the set  $A$  to contain all root nominals, as it will in Section 5, such that these can be copied freely.

## 4 Properties

We briefly remark on some properties of  $ST^A$  that are useful for the completeness proof. We start by noting that while restriction S3 allows us to start from any amount of potential, a single unit is always sufficient to close a branch. Then we lift the S1 and S2 restrictions by showing that unrestricted versions of the proof rules are admissible. This makes it simpler to show further properties of the system, since we do not have to worry about the restrictions any longer. Finally we show a structural property.

### 4.1 Sufficient Potential

That a single unit is sufficient is not surprising: simply never make a detour (i.e. two applications of **GoTo** in a row) and the other rule applications will build up the potential as needed. Similarly, given an existing tableau, construct a more “efficient” counterpart by collapsing sequences of **GoTo** so only the last one remains. **GoTo** serves no other purpose than starting a new block so any subsequent rule applications only depend on the final **GoTo**. The single starting unit may, however, be needed for an initial application of the rule.

► **Lemma 8** (A single unit of potential). *If branch  $\Theta$  closes with respect to  $A$  and potential  $n$  then  $\Theta$  closes with respect to  $A$  and potential 1.*

**Proof.** By induction on the closing tableau for  $\Theta$  (see the formalization for details). ◀

## 4.2 Strengthening

► **Lemma 9** (Strengthening). *Let  $\Theta$  be a branch and  $\Delta$  a set of occurrences of  $\phi$  on  $i$ -blocks in  $\Theta$ . Assume that at least one “lasting occurrence” of  $\phi$  at  $i$  is not in  $\Delta$ . If  $\Theta$  closes wrt.  $A$  and potential  $n$  then so does  $\Theta$  with all occurrences in  $\Delta$  removed.*

**Proof.** By induction on the construction of the closing tableau for  $\Theta$ . When an occurrence in  $\Delta$  is used as rule input, use the lasting occurrence of  $\phi$  instead to construct the tableau for the strengthened branch. No rule applications are invalidated, so the new branch closes under the same amount of potential. Similarly, we only apply rules that were applicable before, so restriction S2 cannot be violated. See the formalization for exact details. ◀

In the formalization we represent the set of occurrences as a set of indices into the branch. We state the lemma over such a set to make it work with the induction principle given by Isabelle/HOL. To lift restriction S1, fix the set of occurrences to contain only the rule output, which must occur elsewhere since S1 is violated, and apply the lemma to justify it.

## 4.3 Substitution

Next we show a substitution lemma. Note that substitution across a tableau can collapse formulas such that an occurrence suddenly violates restriction S1 and cannot be justified as before the substitution. This is why Lemma 9 is useful. But it also means that our substitution lemma will quantify existentially over the potential needed to close the transformed branch: we may need to start from more potential to account for the fewer rule applications. Another complication is that restriction S2 may suddenly be violated by this collapsing but, as we have also shown previously [14], collapsing witnessing nominals allows us to lift S2.

► **Definition 10** ( $\Theta\sigma$ ). *Given a substitution  $\sigma$ , i.e. a mapping from nominals to nominals, and a branch  $\Theta$ ,  $\Theta\sigma$  denotes the branch obtained by replacing every nominal  $i$  in  $\Theta$  by  $\sigma(i)$ .*

Substitutions are allowed to change the type of nominals, e.g. from numbers to strings, so in the following lemma we need to ensure that it leaves enough fresh nominals available.

► **Lemma 11** (Substitution). *Let  $\Theta$  be a branch,  $A$  be a finite set of allowed nominals and  $\sigma$  a substitution whose co-domain is at least as large as its domain. If  $\Theta$  closes with respect to  $A$  then  $\Theta\sigma$  closes with respect to the image of  $A$  under  $\sigma$ .*

**Proof.** By induction on the construction of the closing tableau for an arbitrary  $\sigma$ .

In the ( $\diamond$ ) case, let  $i$  be the generated witnessing nominal. After the (collapsing) substitution, the rule input may become witnessed by some nominal  $\sigma(j)$ , violating S2. In this case, utilize that we can pick  $\sigma$  in the induction hypothesis such that it maps  $i$  to  $\sigma(j)$ . By the side condition on ( $\diamond$ ), the image of  $A$  under the updated  $\sigma$  is the same, but now Lemma 9 justifies the rule output. The rest of the branch is unaffected since  $i$  is fresh.

If S2 is not violated, it may still be that  $\sigma(i)$  is no longer fresh like  $i$  was before the substitution. Therefore, use the finiteness of both the branch and  $A$ , and the size of the co-domain of  $\sigma$ , to obtain a fresh nominal  $k$ . Apply the induction hypothesis at  $\sigma$  mapping  $i$  to  $k$ . This guarantees that the ( $\diamond$ ) rule applies to justify the rule output. ◀

To lift S2, collapse the involved witnessing nominals in the same way as in the proof of Lemma 11 and apply Lemma 9. The finiteness assumption on  $A$  is stronger than we need, but we forgo generalization since we work with finite sets in Section 5 anyway.

#### 4.4 Branch Structure

The following lemma shows that we can add, contract and rearrange blocks on a branch without affecting the existence of a closing tableau. Such operations may violate both S1 and S2, but we have lifted these restrictions already, so we do not need to worry about them.

► **Lemma 12** (Adding, contracting and rearranging blocks). *Let  $\Theta$  be a branch consisting of the set of blocks  $\{B_1, \dots, B_n\}$  and let  $\Theta'$  be a branch whose blocks are a finite superset of  $\{B_1, \dots, B_n\}$ . If  $\Theta$  closes wrt. finite  $A$  then so does  $\Theta'$ .*

**Proof.** By induction on the construction of the closing tableau for arbitrary  $\Theta'$ . In each case we apply the induction hypothesis at  $\Theta'$  extended by  $B$ , where  $B$  is the current block of the original branch. This makes the opening nominals agree on the two branches, so that the original rule applies to the new branch as well. After applying this rule, we justify the  $B$  block by Lemma 9 and the **GoTo** rule. Lemma 11 resolves ( $\diamond$ ) cases where the fresh nominal is not fresh on the new branch since we can substitute it with another fresh nominal. ◀

## 5 Completeness

Our completeness proof is a synthesis of two approaches, both based on showing completeness via contradiction by constructing a model for formulas on open, exhausted branches.

Bolander and Blackburn reason about the shape of such branches directly from the proof rules in their terminating, internalized calculus [9]. Jørgensen et al., on the other hand, define Hintikka sets of blocks as an abstraction of their open, exhausted branches and show model existence for formulas in such sets. They show that any set of blocks without a closing tableau can be extended to a maximal consistent set of blocks and that these are Hintikka sets [21]. Their model construction, however, assumes that all nominals are treated uniformly, which our termination restrictions prevent (cf. Section 3.3). We define Hintikka sets of blocks that characterize open branches *exhausted with respect to* a set of allowed nominals  $A$ . We then abstract the model existence result by Bolander and Blackburn, which is compatible with such branches, and apply it to our Hintikka sets. In Section 5.4 we contrast our approach with the existing work but the proof itself is self-contained.

### 5.1 Hintikka Sets

Figure 4 shows our definition of Hintikka sets of blocks. We reuse the “at” notation from Definition 1 and suppress “in  $H$ ” for brevity. Our goal is to show a model existence result for formulas on blocks in such sets. **ProP** and **NomP** ensure consistency at the bottom by forbidding certain contradictions. The remaining requirements match the proof rules. The ones up to **Nom** ensure *downwards saturation* such that the satisfiability of a complex formula is guaranteed by conditions on its subformulas [21]. The novel condition **Nom** ensures *lateral saturation* of allowed formulas across blocks whose opening nominals denote the same world. This allows us to treat such blocks uniformly when it comes to allowed formulas.

► **Remark 13.** **Nom** replaces three requirements by Jørgensen et al. [21, (iv, v, vii)] that serve the same purpose for a smaller range of formulas.

- Prop** If nominal  $b$  occurs at  $a$  and prop. symbol  $p$  occurs at  $b$  then  $\neg p$  does not occur at  $a$ .
- NomP** If nominal  $i$  occurs at  $a$  then  $\neg i$  does not occur at  $a$ .
- NegN** If  $\neg\neg\phi$  occurs at  $a$  then  $\phi$  occurs at  $a$ .
- DisP** If  $\phi \vee \psi$  occurs at  $a$  then either  $\phi$  or  $\psi$  occurs at  $a$ .
- DisN** If  $\neg(\phi \vee \psi)$  occurs at  $a$  then both  $\neg\phi$  and  $\neg\psi$  occur at  $a$ .
- DiaP** If  $\diamond\phi$  occurs at  $a$  and  $\phi$  is not a nominal then for some  $i$ ,  $\diamond i$  and  $@_i\phi$  occur at  $a$ .
- DiaN** If  $\neg\diamond\phi$  and  $\diamond i$  both occur at  $a$  then  $\neg@_i\phi$  occurs at  $a$ .
- SatP** If  $@_a\phi$  occurs at  $b$  then  $\phi$  occurs at  $a$ .
- SatN** If  $\neg@_a\phi$  occurs at  $b$  then  $\neg\phi$  occurs at  $a$ .
- GoTo** If  $\phi$  occurs at  $a$  and  $i$  is a nominal in  $\phi$  then some block in  $H$  has opening nominal  $i$ .
- Nom** If  $\phi$  and nominal  $a$  both occur at  $b$  and  $\phi$  is *allowed* by  $A$  then  $\phi$  occurs at  $a$ .

■ **Figure 4** Eleven requirements for a set of blocks  $H$  to be a Hintikka set with respect to  $A$ .

### 5.1.1 Equivalence

Assume for the rest of the section that  $H$  is a Hintikka set with respect to the set of allowed nominals  $A$ . We define an equivalence between nominals:

► **Definition 14** (Equivalence). *Nominals  $i, j$  are equivalent,  $i \sim_H j$ , if  $j$  occurs at  $i$  in  $H$ .*

► **Note 15** ( $\sim$  and  $\phi$  at  $i$ ). In the following we typically suppress the subscript in  $\sim_H$  and likewise the fragment “in  $H$ ” in sentences like “ $\phi$  occurs at  $i$  in  $H$ ”.

The equivalence  $i \sim j$  only implies  $j \sim i$  if  $i \in A$  as otherwise **Nom** does not apply: only allowed nominals are symmetric. This motivates the restriction on the following lemma:

► **Lemma 16** (Equivalence relation).  *$\sim_H$  is an equivalence relation on the set of allowed opening nominals in  $H$ .*

**Proof.** *Reflexivity:*  $i \sim_H i$  for any opening nominal  $i$  in  $H$  since opening nominals occur on their own block. *Symmetry:* Assume  $i \sim_H j$  with  $i \in A$ . That is,  $j$  occurs at  $i$  in  $H$  so by **Nom**,  $i$  occurs at  $j$  in  $H$ :  $j \sim_H i$ . *Transitivity:* Assume  $i \sim_H j$  and  $j \sim_H k$  with  $i, k \in A$ . By symmetry,  $i$  occurs at  $j$  in  $H$ :  $j \sim_H i$ . Moreover,  $k \in A$  occurs at  $j$  in  $H$  so by **Nom**,  $k$  occurs at  $i$  in  $H$ :  $i \sim_H k$ . ◀

► **Note 17.** Due to the *GoTo* Hintikka restriction, any nominal occurring in  $H$  also occurs as opening nominal, so  $\sim_H$  is an equivalence relation on the allowed nominals in  $H$ .

### 5.1.2 Model Construction

Let  $|i|_{\sim_H}$  denote the set of nominals equivalent to  $i$  with respect to  $H$ .

We make use of the following shorthand in our model construction:

► **Definition 18** ( $\phi$  at  $a^*$ ). *We say that  $\phi$  occurs at a set of nominals  $a^* = \{a_0, a_1, \dots\}$  if it occurs at some nominal  $a_k \in a^*$  and that  $\phi$  occurs at all  $a^*$  if it occurs at all nominals in  $a^*$ .*

We can now define the model induced by Hintikka set  $H$  and allowed nominals  $A$ :

► **Definition 19** (The model  $\mathfrak{M}_{H,A}$  and assignment  $g_{H,A}$  induced by  $H$  and  $A$ ).

**Worlds** *The worlds of  $\mathfrak{M}_{H,A}$  are sets of equivalent nominals, written  $a^*$ , from  $H$ .*

**Assignment** The assignment  $g_{H,A}$  maps a nominal to the equivalence class of an equivalent, allowed nominal or a singleton set if no such nominal exists:

$$g_{H,A}(a) = \begin{cases} |b|_{\sim_H} & \exists b \in A. a \sim_H b \\ \{a\} & \text{otherwise} \end{cases}$$

**Reachability** From world  $a^*$  we can reach a world exactly if it is denoted by some nominal  $b$  that is reachable at  $a^*$  (as witnessed by  $\diamond b$  occurring at  $a^*$ ):

$$R_{H,A}(a^*) = \{g_{H,A}(b) \mid \exists a \in a^*. \diamond b \text{ occurs at } a \text{ in } H\}$$

**Valuation** Propositional symbol  $p$  holds at world  $a^*$  exactly if  $p$  occurs at  $a^*$  in  $H$ :

$$V_{H,A}(a^*)(p) = \exists a \in a^*. p \text{ occurs at } a \text{ in } H$$

### 5.1.3 Properties of the Model

Consider first a property of the assignment:

► **Lemma 20** (Non-empty assignment). *The induced assignment  $g_{H,A}$  is always non-empty.*

**Proof.** Fix an arbitrary nominal  $a$ . If  $g_{H,A}(a) = \{a\}$  the thesis holds immediately. So assume there is some  $b \in A$  such that  $a \sim_H b$  and  $g_{H,A}(a) = |b|$ .  $b \in |b|$  witnesses the thesis. ◀

The following lemma showcases the lateral saturation guaranteed by the **Nom** condition:

► **Lemma 21** (Assignment closure). *If  $\phi$  is allowed by  $A$  and  $\phi$  occurs at  $a$  in  $H$  then  $\phi$  occurs at all  $g_{H,A}(a)$  in  $H$  (and at least one such world exists).*

**Proof.** If  $g_{H,A}(a) = \{a\}$  the thesis holds immediately. So assume there is some  $b \in A$  where  $b$  occurs at  $a$  in  $H$  and  $g_{H,A}(a) = |b|$ . Then by Hintikka requirement **Nom**,  $\phi$  occurs not only at  $b$  in  $H$  but at all  $a \in |b|$  in  $H$ , proving the thesis. Lemma 20 gives the parenthetical. ◀

### 5.1.4 Model Existence

We can now prove model existence:

► **Lemma 22** (Model existence). *Let  $H$  be a Hintikka set with respect to allowed nominals  $A$ . We show two statements by mutual induction:*

- *If  $\phi$  occurs at  $i$  in  $H$  and  $\phi$  is allowed by  $A$  then  $\mathfrak{M}_{H,A}, g_{H,A}, g_{H,A}(i) \models \phi$ .*
- *If  $\neg\phi$  occurs at  $i$  in  $H$  and  $\phi$  is allowed by  $A$  then  $\mathfrak{M}_{H,A}, g_{H,A}, g_{H,A}(i) \not\models \phi$ .*

**Proof.** By induction on the structure of  $\phi$  for an arbitrary nominal  $i$ . The proof follows the one by Bolander and Blackburn [9]. We suppress subscripts for readability.

If  $p$  at  $i$  then  $p$  at  $g(i)$  by Lemma 21, which matches the valuation, so  $\mathfrak{M}, g, g(i) \models p$ .

If  $\neg p$  at  $i$  then  $\neg p$  at all  $g(i)$  so by **ProP**,  $p$  does not occur at  $g(i)$ , so  $\mathfrak{M}, g, g(i) \not\models p$ .

If  $a$  at  $i$  then from the assumption  $a \in A$  we have  $g(i) = |a|$  and  $g(a) = |a|$  and thereby  $g(i) = g(a)$  so  $\mathfrak{M}, g, g(i) \models a$ .

If  $\neg a$  at  $i$  then  $\neg a$  at  $g(i)$  by Lemma 21. Moreover,  $a \in A$  by assumption so from Lemma 21 we have that  $a$  occurs at all  $g(a)$ . We thus have  $\neg a$  at  $g(i)$  but  $a$  at all  $g(a)$  so by **NomN**,  $g(i) \neq g(a)$  and therefore  $\mathfrak{M}, g, g(i) \not\models a$ .

If  $\neg\phi$  at  $i$  then  $\mathfrak{M}, g, g(i) \not\models \phi$  by the induction hypothesis so  $\mathfrak{M}, g, g(i) \models \neg\phi$ .

If  $\neg\neg\phi$  at  $i$  then  $\phi$  at  $i$  by **NegN** and  $\mathfrak{M}, g, g(i) \not\models \neg\phi$  by the induction hypothesis.



The cases for  $\phi \vee \psi$ ,  $\neg(\phi \vee \psi)$ ,  $@_j\phi$  and  $\neg@_j\phi$  at  $i$  all follow similarly to  $\neg\phi$  and  $\neg\neg\phi$ .

If  $\diamond j$  at  $i$  then  $j \in A$  by assumption. Thus  $\diamond j$  at  $g(i)$  so  $g(i) R g(j)$  and  $\mathfrak{M}, g, g(i) \models \diamond j$ .

If  $\diamond\phi$  at  $i$  where  $\phi$  is not a nominal then by **DiaP** (and Lemma 21) there is some witnessing nominal  $k$  such that  $\diamond k$  and  $@_k\phi$  both appear at  $g(i)$ . By **SatP**,  $\phi$  then occurs at  $k$  and by the induction hypothesis at  $k$  we have  $\mathfrak{M}, g, g(k) \models \phi$ . From  $\diamond k$  at  $g(i)$  we have  $g(i) R g(k)$  so combined we get  $\mathfrak{M}, g, g(i) \models \diamond\phi$ .

If  $\neg\diamond\phi$  at  $i$  then  $\neg\diamond\phi$  at  $g(i)$  by Lemma 21. We need to show that all worlds reachable from  $g(i)$  falsify  $\phi$ . So assume for some arbitrary  $j$  that  $\diamond j$  occurs at some  $a \in g(i)$ . By **Nom**, we also have  $\neg\diamond\phi$  at  $a$  so by **DiaN** we get  $\neg@_j\phi$  at  $a$  and finally by **SatN** we have  $\neg\phi$  at  $j$ . The induction hypothesis at  $j$  then tells us that  $\mathfrak{M}, g, g(j) \not\models \phi$  as needed. Since  $j$  was chosen arbitrarily,  $\mathfrak{M}, g, g(i) \not\models \diamond\phi$ .

Each appeal to the induction hypothesis requires showing that the subformula is allowed by  $A$  but since it is a subformula this holds trivially.  $\blacktriangleleft$

## 5.2 Maximal Consistent Sets

Our next task is to follow the classical synthetic recipe: extend a consistent set of blocks to be maximally consistent, show that such sets fulfill all Hintikka requirements and thus that formulas in them are satisfiable. Consistency and maximality are standard but wrt.  $A$ :

► **Definition 23** (Consistency). *The set of blocks  $S$  is consistent wrt.  $A$  if there is no finite subset  $S' \subseteq S$  such that  $S'$  has a closing tableau wrt.  $A$  and any amount of potential.*

► **Definition 24** (Maximality). *The set of blocks  $S$  is maximal wrt.  $A$  if it is consistent wrt.  $A$  and for any block  $B \notin S$  the set  $S \cup \{B\}$  is inconsistent wrt.  $A$ .*

Besides maximally consistent, our constructed set will also be  $\diamond$ -saturated [21]:

► **Definition 25** ( $\diamond$ -Saturation). *The set of blocks  $S$  is  $\diamond$ -saturated if for any  $\phi$  at any  $a$  in  $S$ , where  $\phi$  is not a nominal, there is a nominal  $i$  such that  $@_i\phi$  and  $\diamond i$  both occur at  $a$  in  $S$ .*

We now construct our  $\diamond$ -saturated maximally consistent set and show it is a Hintikka set:

► **Definition 26** (Lindenbaum-Henkin construction). *Assume an enumeration of all blocks  $B_0, B_1, B_2 \dots$  in the language. From a consistent set  $S_0$  we build an infinite sequence of consistent sets  $S_0, S_1, S_2, \dots$  in the following way. Given  $S_n$ , construct  $S_{n+1}$  like so:*

$$S_{n+1} = \begin{cases} S_n & \text{if } S_n \cup \{B_n\} \text{ is inconsistent wrt. } A \\ S_n \cup \{B_n\} \cup \{B'\} & \text{otherwise, where } B' \text{ is a } \diamond\text{-witness for } B_n \end{cases}$$

A  $\diamond$ -witness for a block  $B$  is a block with the same opening nominal that witnesses all  $\diamond\phi$ -formulas in  $B$  using fresh and disallowed nominals (when  $\phi$  is not a nominal).

► **Lemma 27** (Lindenbaum-Henkin). *Let  $S_0$  be a consistent set of blocks with respect to finite  $A$  and over a finite set of nominals. Then  $\bigcup S_n$  as given by Definition 26 is a  $\diamond$ -saturated maximally consistent set.*

**Proof.** The three-part proof follows the one by Jørgensen et al. [21].

**Consistency.** Proof by contradiction. Assume  $\bigcup S_n$  is inconsistent. Then some finite subset  $S' \subseteq \bigcup S_n$  has a closing tableau. But the sequence  $S_0, S_1, S_2, \dots$  grows with respect to  $\subseteq$  so there must be an  $m$  such that  $S' \subseteq S_m$ . And since  $S_0$  is consistent, it follows by induction on  $m$  that  $S_m$  is too (each  $\diamond$ -witness preserves consistency due to the ( $\diamond$ ) rule). This contradicts the existence of an inconsistent, finite subset  $S'$ .



**Maximality.** Proof by contradiction. Assume that there is some block  $B_m \notin \bigcup S_n$  such that  $\bigcup S_n \cup \{B_m\}$  is still consistent. This block is part of the enumeration of blocks, but was not added to  $S_{m+1}$ . This can only be because  $S_m \cup \{B_m\}$  is inconsistent. However,  $S_m \cup \{B_m\} \subseteq \bigcup S_n \cup \{B_m\}$  contradicting the consistency of the right-hand side.

$\diamond$ -**Saturation.** Follows directly from the addition of  $\diamond$ -witnesses.  $\blacktriangleleft$

► **Lemma 28** (Smullyan-Fitting block lemma). *Assume  $S$  is a  $\diamond$ -saturated maximal consistent set of blocks wrt. a finite set  $A$  and a finite set of nominals. Then  $S$  is a Hintikka set.*

**Proof.** The proof follows the one by Jørgensen et al. [21] but we have fewer cases since we have fewer Hintikka requirements. The cases are straight-forward so we only exemplify three, with the last being the typical one. The remaining cases can be found in the formalization.

**Case ProP.** Proof of negation. Assume that  $b$  occurs at  $a$ ,  $p$  occurs at  $b$  and  $\neg p$  occurs at  $a$  in  $S$  for some  $a, b, p$ . The set  $S$  is assumed to be consistent but we can construct a closing tableau from these blocks by applying the **Nom** rule to get  $\neg p$  at  $b$  and immediately close due to the existing  $p$  at  $b$ .

**Case DiaP.** Follows directly from  $\diamond$ -saturation.

**Case Nom.** Assume that both  $\phi$  and  $a$  occur at  $b$  in  $S$  and that  $\phi$  is allowed by  $A$ . Assume towards a contradiction that  $\phi$  does not occur at  $a$  in  $S$ . Then by the maximality of  $S$ , we can find an inconsistent finite subset  $S' \cup \{([\phi], a)\} \subseteq S \cup \{([\phi], a)\}$  where  $([\phi], a)$  is an  $a$ -block that only contains  $\phi$ . If a closing tableau exists for  $S' \cup \{([\phi], a)\}$  then it also exists for the larger set  $S' \cup \{([\phi], a)\} \cup \{([\phi, a], b)\}$  (Lemma 12). But now the **Nom** rule tells us that  $\phi$  at  $a$  is redundant, so just  $S' \cup \{([\phi], a)\} \cup \{([\phi, a], b)\}$  is inconsistent. The **GoTo** rule gets us to  $S' \cup \{([\phi, a], b)\}$  and this set is trivially a subset of  $S$ , contradicting its consistency.  $\blacktriangleleft$

### 5.3 Tying It All Together

Completeness follows by constructing a model for any formula whose tableau does not close.

► **Theorem 29** (Completeness). *Assume that  $\phi$  is a valid formula and  $a$  is some nominal. Let  $A$  be the set containing all nominals in  $\phi$ . Then the branch consisting solely of  $\neg\phi$  on an  $a$ -block has a closing tableau with respect to  $A$  and 1 unit of potential.*

**Proof.** Assume towards a contradiction that the branch does not close. Then the set  $S_0 = \{([\neg\phi], a)\}$  is consistent with respect to  $A$ . We construct  $\bigcup S_n$ , which by Lemma 27 is a  $\diamond$ -saturated maximal consistent set of blocks, so by Lemma 28  $\bigcup S_n$  is a Hintikka set.

Since  $\neg\phi$  occurs at  $a$  in  $\bigcup S_n$ , we obtain from Lemma 22 a model that does not satisfy  $\phi$ , namely  $\mathfrak{M}_{H,A}, g_{H,A}, g_{H,A}(a) \not\models \phi$ . This contradicts our validity assumption, so the branch must close. By Lemma 8 it must close from a single unit of potential.  $\blacktriangleleft$

### 5.4 Relation to Existing Work

In this section we provide context for our induced model, Definition 19, and the corresponding Lemma 22. Readers less familiar with tableau systems for hybrid logic may skip this section. To refresh, Bolander and Blackburn give an analytic proof for a terminating, internalized calculus [9] and Jørgensen et al. give a synthetic proof for the non-terminating system ST [21].

### 5.4.1 Worlds

Jørgensen et al. have no restrictions on their **Nom** rule so they have no nominal asymmetry (cf. Section 3.3) and  $\sim_H$  is an equivalence relation on all nominals. They use representatives of such equivalence classes as their worlds [21]. Since  $\sim_H$  is only an equivalence relation on a subset of our nominals, we cannot use equivalence classes directly. Instead we use sets of equivalent nominals. Bolander and Blackburn use plain nominals as their worlds.

### 5.4.2 Assignment

Jørgensen et al. map each nominal  $i$  in  $H$  to its equivalence class  $|i|_{\sim_H}$  [21]. If we artificially fix  $A$  to contain all nominals in  $H$  then  $\sim_H$  becomes an equivalence relation on all nominals. Our assignment then reduces to its first clause and becomes equivalent to theirs.

Bolander and Blackburn map each nominal  $a$  to its “urfather”  $u(a)$ : either an equivalent “right nominal” or the nominal itself if no such nominal exists [9]. This is very similar to our assignment that maps each nominal to the equivalence class of an equivalent *allowed nominal* or the singleton set if no such nominal exists.

A *right nominal*, understood in terms of our setting, is a non-opening nominal that occurs on its own. Since there may be multiple equivalent right nominals, Bolander and Blackburn impose an ordering on them and always choose the smallest one to ensure that their assignment is well-defined [9]. Working with sets of nominals frees us from such concerns.

### 5.4.3 Reachability and the Bridge Rule

It is worthwhile to compare the three different reachability relations from the considered systems. By writing them in similar notation we get:

Jørgensen et al.	$ i  R_H  j $	iff $\diamond j$ occurs at $i$ in $H$
Bolander and Blackburn	$i R_H u(j)$	iff $\diamond j$ occurs at $i$ in $H$
The present paper	$i^* R_H g_{H,A}(j)$	iff $\diamond j$ occurs at $i^*$ in $H$

If we further note that  $g(j) = |j|$  for Jørgensen et al. [21] and  $g(j) = u(j)$  for Bolander and Blackburn [9] we see that the relations are all defined in the same way over the assignment: *a world is reachable iff it is denoted by a nominal  $j$  such that  $\diamond j$  occurs at the current world.* Only the treatment of the worlds differ. Since Jørgensen et al. use representatives of their sets they need the following Hintikka requirement to ensure well-definedness:

If there is an  $i$ -block in  $H$  with  $\diamond j$  on it, and a  $j$ -block in  $H$  with  $k$  on it, then there is an  $i$ -block in  $H$  with  $\diamond k$  on it [21, (vi)].

To see why, imagine that the premises hold but the conclusion does not. Then  $|i| R_H |j|$  and  $j \sim_H k$  but not  $|i| R_H |k|$  even though  $|j| = |k|$  by the second premise, so the choice of representative matters when it should not. In our setting we side-step the problem completely by having no representatives but quantifying existentially over the nominals in our worlds.

If we view the requirement as a rule, we get the known **Bridge** rule that produces  $\diamond k$  at  $i$  given  $\diamond j$  at  $i$  and nominal  $k$  at  $j$ . Jørgensen et al. prove the admissibility of **Bridge** as part of their completeness proof [21]. We include this result in the formalization (when  $j \in A$ ) because it is interesting in its own right [4] but do not need it for completeness.

### 5.4.4 Valuation

Our valuation is standard but our use of sets instead of representatives slightly complicates the **ProP** Hintikka requirement, where we take equivalence of nominals into account. For Jørgensen et al. the following suffices: “if there is an  $i$ -block in  $H$  with atomic formula  $a$  on it then there is no  $i$ -block in  $H$  with  $\neg a$  on it.” [21].

### 5.4.5 Model Existence

We turn now to the model existence result, Lemma 22, inspired by Blackburn and Bolander [9].

The two nominal cases and the  $\diamond j$  case rely on the involved nominals being in  $A$ . Bolander and Blackburn work with right nominals instead of allowed nominals [9]. This gives them the positive nominal case for free, since the formula in that case is a right nominal. In the negative nominal case, however, they need to rely on a special ( $\neg$ ) rule that upgrades a negated nominal, “ $@_i \neg a$ ”, to a right nominal “ $@_a a$ ”. They need this rule because of the nature of internalized tableau systems: the nominal  $i$  in a satisfaction statement  $@_i a$  has lower status than the right nominal  $a$ . The status of nominals in our system is not defined structurally but by the set  $A$ . Thus, we make the ( $\neg$ ) rule unnecessary by picking  $A$  carefully.

Finally, Bolander and Blackburn assume that the formula in question is not a  $\diamond j$  formula produced by the ( $\diamond$ ) rule. Our assumption  $j \in A$  matches this, since the ( $\diamond$ ) rule cannot generate an allowed nominal, but we are free from keeping track of actual rule applications.

## 6 Relation to $ST^*$

Here, we relate our restrictions S1-S4 to the restrictions R1-R5 and  $Nom^*$  rule in  $ST^*$  [4].

### 6.1 System $ST^*$

For reasons of space we introduce  $ST^*$  only briefly. To obtain  $ST^*$ , take the rules in Figure 2, add another rule called **Name** that introduces a fresh nominal to the branch and impose restrictions R1-R5 and  $Nom^*$  that we explain in the following. Since the rules of  $ST^A$  are a subset of  $ST^*$ , it is meaningful to compare the strength of our restrictions to those of  $ST^*$ .

Blackburn et al. [4] need the **Name** rule since they allow the very first block to have no opening nominal. We have dispensed with this flexibility to obtain a simpler formalization.

### 6.2 Restrictions R1-R5

Restriction R1 states that “a formula is never added to an  $i$ -block if it already occurs in an  $i$ -block on the same branch” [4]. This formulation is more ambiguous than our S1, which states when a rule is applicable. Any rule application outlawed by R1 is also outlawed by S1:

► **Lemma 30** (R1 implies S1). *If R1 outlaws a rule application then so does S1.*

**Proof.** R1 outlaws the rule application so it must include no formulas new to the block type. Therefore, S1 outlaws it too. ◀

Restriction R2 states that “the ( $\diamond$ ) rule can not be applied twice to the same formula occurrence” [4]. Note that formalizing this would require keeping track of ( $\diamond$ ) rule applications. This is why S2 is formulated in terms of branch content instead. It is at least as strict as R2:

► **Lemma 31** (R2 implies S2). *If R2 outlaws an application of ( $\diamond$ ) then so does S2.*

**Proof.** Assume that an application of the rule  $(\diamond)$  to formula  $\diamond\phi$  at  $a$  is outlawed by R2. This means that  $(\diamond)$  has already been applied to  $\diamond\phi$  at  $a$ . So for some nominal  $i$  there must be formulas  $@_i\phi$  and  $\diamond i$  witnessing  $\diamond\phi$  at  $a$ . Thus the application is also outlawed by S2. ◀

Restriction R3 applies to the omitted name rule so we have no equivalent of it [4].

Restriction R4 states that “the GoTo rule can not be applied twice in a row” [4]. Our counterpart is S3 that does allow repeated applications but still prevents repeating the rule ad infinitum (cf. Figure 3b). We see in Section 6.5 why this extra flexibility is desirable. For now recall the idea from Section 4.1 that any tableau with repeated applications of GoTo can be translated into one where just the final application remains. We have the following:

► **Lemma 32** (From S3 to R4). *A tableau satisfying S3 collapses into one that satisfies R4 where only finite sequences of GoTo are removed and all non-GoTo applications are preserved.*

**Proof.** By collapsing all sequences of GoTo applications into the last one (cf. Lemma 8). All such sequences are finite due to decreasing potential so “the last one” is well-defined. ◀

Finally, restriction R5 can be ignored here: it restricts the more liberal variants of rules  $(@)$  and  $(\neg@)$  in system ST to the versions present in  $ST^*$  and  $ST^A$  [4].

### 6.3 Nom\* and Allowed Nominals

We turn now to the Nom\* rule in  $ST^*$  and its relationship to our set of allowed nominals  $A$  in restriction S4. We first need the following by Blackburn et al. [4]: “A quasi-root subformula is a formula of the form  $\phi$ ,  $\neg\phi$ ,  $@_i\phi$  or  $\neg@_i\phi$  where  $\phi$  is a subformula of the root.”

Their Nom\* rule is then defined as follows:

Suppose  $i$  and  $j$  are nominals,  $\phi$  is a quasi-root subformula and  $j \neq i, \phi$ . If  $j$  and  $\phi$  both occur in  $i$ -blocks on a branch  $\Theta$ , then  $\phi$  can be added to any  $j$ -block on  $\Theta$  [4].

By inspecting the rules of  $ST^*$  and  $ST^A$  we see that only the  $(\diamond)$  rule can produce formulas that are not quasi-root subformulas [4]. As such, the only formulas that Nom\* does not allow us to copy are formulas  $i$  and  $\diamond i$  where  $i$  was introduced by  $(\diamond)$ . This is exactly what restriction S4 enforces on our Nom rule (cf. Section 3.3). So S4 is at least as strict:

► **Lemma 33** (Nom implies Nom\*). *Suppose that  $\phi$  and  $a$  both occur at  $b$  in a tableau constructed under the allowed set of nominals  $A$ . If Nom can add  $\phi$  to  $a$  then so can Nom\*.*

**Proof.** If  $\phi$  can be added by Nom it must be allowed by  $A$ . Thus  $\phi$  must be a quasi-root subformula. Moreover, since adding  $\phi$  to  $a$  does not violate S1 (or R1),  $a \neq \phi$  and likewise  $a \neq b$ . Ultimately, Nom\* can also add  $\phi$  to  $a$ . ◀

### 6.4 Termination

We have covered all differences between  $ST^*$  and  $ST^A$  and seen how the restrictions compare. This motivates the following unformalized theorem and proof sketch:

► **Theorem 34** ( $ST^A$  is terminating). *Any  $ST^A$  tableau is finite.*

**Proof.** Lemmas 30–33 imply that we can translate any  $ST^A$  tableau into an  $ST^*$  tableau of similar size by collapsing repeated applications of GoTo (and adding an initial application of the Name rule). Since all  $ST^*$  tableaux are finite [4] so must any  $ST^A$  tableau be. ◀

Blackburn et al. [4] exemplify a number of infinite branches possible in system ST and show that they are illegal in system  $ST^*$ . In support of the above theorem, we note that the sequences of rule applications leading to those infinite branches are also outlawed in  $ST^A$ .

$$\begin{array}{c}
 a \\
 \phi \\
 \hline
 a' \quad \text{GoTo} \\
 \phi' \quad \text{R} \\
 \hline
 i \quad \text{GoTo} \\
 \psi
 \end{array}
 \qquad
 \begin{array}{c}
 a \\
 \phi \\
 \hline
 a \quad \text{GoTo} \\
 \phi \quad \text{R} \\
 \hline
 \sigma(i) \quad \text{GoTo} \\
 \psi\sigma
 \end{array}$$

(a) Possible segment on original closing tableau.    (b) R becomes invalid causing two GoTos in a row.

■ **Figure 5** Unjustified GoTo after applying substitution  $\sigma$  that unifies  $a$  and  $a'$  as well as  $\phi$  and  $\phi'$ .

## 6.5 Restricting the GoTo Rule

We should motivate our choice of S3 over R4. As Section 4 shows, we typically show lemmas of the form “if branch  $\Theta$  has a closing tableau then so does  $f(\Theta)$ ”, where  $f$  is some operation like substitution or restructuring. In a proof by induction on the closing tableau under restriction R4 we need to show in each non-GoTo case that GoTo becomes applicable, since we need that assumption to discharge the GoTo case. However, the transformation may invalidate a previously valid rule application and prevent us from making this promise. Figure 5 depicts a possible case when proving the substitution lemma. Before the substitution, the application of rule R was legal, but afterwards it violates restriction R1. We can still justify the extension  $\phi$  with the Strengthening Lemma 9 but doing so does not make GoTo applicable afterwards.

We might give a more intricate transformation that also prunes detours but that would complicate an otherwise simple idea like substitution. We could also state the lemma in weaker terms that allow for a different branch structure, but we prefer to give straight-forward lemmas and transformations. Our S3 restriction resolves the issue by dealing with detours separately. Consider Figure 5 from the perspective of potential: we need to start from more potential to close the transformed branch since we lose a rule application, but we can simply do this, so the detour becomes benign. Thus, we can give the transformation we want, we just need to existentially quantify the potential required to close the resulting branch.

## 7 Formalization

In general, the formalization consists of close to 5000 lines in the *intelligible semi-automated reasoning* language Isar [27] and follows the structure of the paper. It is accepted into the Archive of Formal Proofs and thus kept up to date with new versions of Isabelle/HOL.

We formalize the logic as a deep embedding into higher-order logic by specifying the syntax as a datatype and the semantics as a predicate on that datatype (alongside a model and an assignment). Types in higher-order logic are non-empty so we represent the set of worlds as a type variable  $'w$ . Similarly, we use  $'a$  for the universe of propositional symbols and  $'b$  for the universe of nominals. We formalize a block as a list of formulas paired with its opening nominal and a branch as a list of blocks, where lists in Isabelle/HOL are finite, ordered sequences. We use the **inductive** command to specify the proof system as ten inductive cases. The command provides a predicate  $\vdash$  for whether or not a given branch closes with respect to a set  $A$  and potential  $n$ . Thus, we abstract away the concrete shape of a closing tableau and reason only about its existence. This suffices for formalizing completeness but not termination where we would need to inspect well-formed but infinite branches. However, it permits induction over the proof rules instead of the trickier coinduction.

Imagine that we formalized  $\text{ST}^*$  instead of  $\text{ST}^A$ . Section 6.5 motivated our choice of S3 over R4. Restriction R2 on the  $(\diamond)$  rule would require us to additionally index our predicate  $\vdash$  by a list of indices, each pointing to a formula occurrence that  $(\diamond)$  cannot be applied to. When

proving lemmas by induction, we would need to make suitable assumptions about this list. Instead, our formulation  $S2$  identifies the applicability of  $(\diamond)$  from the branch content itself, which we already know. The  $Nom^*$  rule considers quasi-root subformulas and would require us to remember the root segment of the tableau as we extend it, complicating induction proofs too. Our parameterization of the rules by the set  $A$  causes no such complications.

Imagine next that we adapted the completeness proof for  $ST^*$  to  $ST^A$ . That proof works by translation from a different system with an analytic completeness proof, which we would have to formalize as well. This could be done: Blanchette, Popescu and Traytel [7, 8] have formalized analytic completeness proofs for first-order logic in Isabelle/HOL. Instead, our *standalone* synthetic completeness proof joins a family of such proofs in Isabelle/HOL [2, 15, 16]. While possible, a similar proof for  $ST^*$  would, as described, be harder to formalize.

## 8 Conclusion and Future Work

We have presented a Seligman-style tableau system for hybrid logic with a formalization in Isabelle/HOL of its soundness and completeness and argued that it is terminating. The restrictions required for termination cause an asymmetry in branches that makes a previous synthetic completeness proof for hybrid logic tableau systems inapplicable. We have presented a novel proof that works in this case and described its relation to existing work. The use of plain sets instead of representatives in the model construction relieves us of some concerns about well-definedness. Our work is the first sound and complete formalized proof system for hybrid logic and the first synthetic proof for a terminating hybrid logic tableau system.

Blackburn et al. showed termination of  $ST^*$  by a translation of any branch into a terminating system and we claim termination of  $ST^A$  by possible translation into  $ST^*$ . We are currently working on a direct, formalized termination proof for  $ST^A$  through a decreasing measure argument in the style of Bolander and Blackburn [9]. This will allow code generation for a verified decision procedure based on the tableau system. We also want to explore extensions to the logic and investigate a Seligman-style system for intuitionistic hybrid logic.

---

### References

- 1 Carlos Areces, Patrick Blackburn, and Maarten Marx. Hybrid logics: Characterization, interpolation and complexity. *The Journal of Symbolic Logic*, 66(3):977–1010, 2001.
- 2 Stefan Berghofer. First-Order Logic According to Fitting. *Archive of Formal Proofs*, 2007. Formal proof development. URL: <http://isa-afp.org/entries/FOL-Fitting.html>.
- 3 Patrick Blackburn. Representation, reasoning, and relational structures: A hybrid logic manifesto. *Logic Journal of the IGPL*, 8(3):339–365, 2000.
- 4 Patrick Blackburn, Thomas Bolander, Torben Braüner, and Klaus Frovin Jørgensen. Completeness and termination for a Seligman-style tableau system. *Journal of Logic and Computation*, 27(1):81–107, 2017.
- 5 Patrick Blackburn and Miroslava Tzakova. Hybridizing concept languages. *Annals of Mathematics and Artificial Intelligence*, 24(1-4):23–49, 1998.
- 6 Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *Certified Programs and Proofs, CPP. Proceedings*, pages 1–13. ACM, 2019.
- 7 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Abstract completeness. *Archive of Formal Proofs*, 2014. Formal proof development. URL: [https://isa-afp.org/entries/Abstract\\_Completeness.html](https://isa-afp.org/entries/Abstract_Completeness.html).
- 8 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.

- 9 Thomas Bolander and Patrick Blackburn. Termination for Hybrid Tableaus. *Journal of Logic and Computation*, 17(3):517–554, 2007.
- 10 Thomas Bolander and Torben Braüner. Tableau-based decision procedures for hybrid logic. *Journal of Logic and Computation*, 16(6):737–763, 2006.
- 11 Torben Braüner and Valeria de Paiva. Intuitionistic hybrid logic. *Journal of Applied Logic*, 4(3):231–255, 2006.
- 12 Christian Doczkal and Gert Smolka. Constructive formalization of hybrid logic with eventualities. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs, CPP. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2011.
- 13 Melvin Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169. Springer Science & Business Media, 1983.
- 14 Asta Halkjær From, Patrick Blackburn, and Jørgen Villadsen. Formalizing a Seligman-style tableau system for hybrid logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 474–481, Cham, 2020. Springer International Publishing.
- 15 Asta Halkjær From. Epistemic Logic. *Archive of Formal Proofs*, 2018. Formal proof development. URL: [http://isa-afp.org/entries/Epistemic\\_Logic.html](http://isa-afp.org/entries/Epistemic_Logic.html).
- 16 Asta Halkjær From. A sequent calculus for first-order logic. *Archive of Formal Proofs*, 2019. Formal proof development. URL: [http://isa-afp.org/entries/FOL\\_Seq\\_Calc1.html](http://isa-afp.org/entries/FOL_Seq_Calc1.html).
- 17 Asta Halkjær From. Hybrid Logic. Master’s thesis, Technical University of Denmark, 2020.
- 18 Asta Halkjær From. Hybrid logic in the Isabelle proof assistant: Benefits, challenges and the road ahead. In Nicola Olivetti and Rineke Verbrugge, editors, *Short Papers: Advances in Modal Logic (AiML) 2020*, pages 23–27, 2020.
- 19 Didier Galmiche and Yakoub Salhi. Sequent calculi and decidability for intuitionistic hybrid logic. *Information and Computation*, 209(12):1447–1463, 2011.
- 20 Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). In David A. Schmidt, editor, *Programming Languages and Systems, ESOP, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2004.
- 21 Klaus Frovin Jørgensen, Patrick Blackburn, Thomas Bolander, and Torben Braüner. Synthetic completeness proofs for Seligman-style tableau systems. In *Advances in Modal Logic, Volume 11*, pages 302–321, 2016.
- 22 Sven Linker. Hybrid Multi-Lane Spatial Logic. *Archive of Formal Proofs*, 2017. Formal proof. URL: [http://isa-afp.org/entries/Hybrid\\_Multi\\_Lane\\_Spatial\\_Logic.html](http://isa-afp.org/entries/Hybrid_Multi_Lane_Spatial_Logic.html).
- 23 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
- 24 Jeremy Seligman, Fenrong Liu, and Patrick Girard. Facebook and the epistemic logic of friendship. In Burkhard C. Schipper, editor, *Proceedings of the 14th Conference on Theoretical Aspects of Rationality and Knowledge (TARK 2013)*, pages 229–238, Chennai, India, 2013.
- 25 Jerry Seligman. Internalization: The case of hybrid logics. *Journal of Logic and Computation*, 11(5):671–689, 2001.
- 26 Raymond M Smullyan. *First-Order Logic*. Dover Publications, 1995.
- 27 Makarius Wenzel. Isabelle/Isar – a generic framework for human-readable proof documents. *From Insight to Proof – Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar and Rhetoric*, 10(23):277–298, 2007.





# Encoding of Predicate Subtyping with Proof Irrelevance in the $\lambda\Pi$ -Calculus Modulo Theory

Gabriel Hondet ✉ 🏠

Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France

Frédéric Blanqui ✉ 🏠 

Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France

---

## Abstract

The  $\lambda\Pi$ -calculus modulo theory is a logical framework in which various logics and type systems can be encoded, thus helping the cross-verification and interoperability of proof systems based on those logics and type systems. In this paper, we show how to encode *predicate subtyping* and *proof irrelevance*, two important features of the PVS proof assistant. We prove that this encoding is correct and that encoded proofs can be mechanically checked by DEDUKTI, a type checker for the  $\lambda\Pi$ -calculus modulo theory using rewriting.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Type theory; Theory of computation  $\rightarrow$  Higher order logic; Theory of computation  $\rightarrow$  Equational logic and rewriting

**Keywords and phrases** Predicate Subtyping, Logical Framework, PVS, Dedukti, Proof Irrelevance

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.6

**Acknowledgements** The authors thank Gilles Dowek and the anonymous referees very much for their remarks.

## 1 Introduction

A substantial number of proof assistants can be used to develop formal proofs, but a proof developed in an assistant cannot, in general, be used in another one. This impermeability generates redundancy since theorems are likely to have one proof per proof assistant. It also prevents adoption of formal methods by industry because of the lack of standards and the difficulty to use adequately formal methods.

Logical frameworks are a part of the answer. Because of their expressiveness, different logics and proof systems can be stated in a common language. The  $\lambda\Pi$ -calculus modulo theory, or  $\lambda\Pi/\equiv$ , is such a logical framework. It is the simplest extension of simply typed  $\lambda$ -calculus with dependent types and arbitrary computation rules. Fixed-length vectors are a common example of dependent type, that can be represented in the  $\lambda\Pi$ -calculus as  $\forall n : \mathbb{N}, \text{Vec}(n)$ . The  $\lambda\Pi$ -calculus modulo theory already allows to formulate first order logic, higher order logic [5] or proof systems based on *Pure Type Systems* [12] such as MATITA [3], COQ [10] or AGDA [16].

PVS [28] is a proof assistant that has successfully been used in collaboration by academics and industrials to formalise and specify real world systems [27]. More precisely, PVS is an environment comprising a specification language, a type checker and a theorem prover. One of the specificities of PVS is its ability to blend type checking with theorem proving by requiring terms to validate arbitrary predicates in order to be attributed a certain type. This ability is a consequence of *predicate subtyping* [30]. It facilitates the development of specifications and provides a more expressive type system which allows to encode more constraints. For instance, one can define the inverse function  $\text{inv} : \mathbb{R}^* \rightarrow \mathbb{R}$ , where  $\mathbb{R}^*$  is a predicate subtype defined as reals which are not zero.



© Gabriel Hondet and Frédéric Blanqui;  
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 6; pp. 6:1–6:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

If predicate subtyping provides a richer type system, it also makes type checking of specifications undecidable. In [17], F. Gilbert paved the way of the expression of PVS into  $\lambda\Pi/\equiv$ : he formalised the core of PVS and provided a language of certificates for PVS whose type checking is decidable. However, the encoding in  $\lambda\Pi/\equiv$  of this language of certificates relies on *proof irrelevance*.

The following work proposes an encoding of proof irrelevant equivalences into the  $\lambda\Pi$ -calculus modulo theory. It also inspects the completion of such equations into a confluent rewrite system. The resulting rewrite system can be used to provide an encoding of PVS into DEDUKTI, a type-checker for the  $\lambda\Pi$ -calculus modulo theory based on rewriting [4].

### Related work

An encoding or “simulation” of predicate subtyping *à la* PVS into HOL can be found in [20]. The objective of that work was to get some facilities provided by predicate subtyping into HOL rather than providing a language of certificates, and proof checking hence remains undecidable. Moreover, predicate subtypes are not represented by types but by theorems.

In [32], predicate subtyping is weakened into a language named RUSSELL to be then converted into CIC. This conversion amounts to the insertion of coercions and unsolved meta-variables, the latter embody PVS *type correctness conditions* (TCC). The equational theory used in the CIC encoding is richer than ours since it includes surjective pairing  $e = \text{pair } T \ U \ (\text{fst } T \ U \ e) \ (\text{snd } T \ U \ e)$  and  $\eta$ -equivalence  $f = \lambda x, f \ x$  in addition to proof irrelevance.

In [36], proof irrelevance is embedded into Luo’s ECC [25] and its dependent pairs. Pairs and dependent pair types come in two flavours, the proof irrelevant one and the normal one. The flavour is noted by an annotation, and proof irrelevance is implemented by a reduction which applies only on annotated pairs. The article presents as well an application to PVS.

On a slightly more practical side, the automated first-order prover ACL2 [21] reproduces the system of “guards” provided by predicate subtyping into its logic based on COMMON LISP with the concept of *gold symbols*. Approximately, a symbol is gold if all its TCC have been solved.

Some theories – often based on Martin-Löf’s Type Theory – blend together a decidable (called *definitional* or *intensional*) equality with an undecidable (said *extensional*) equality. In [29], a judgement “ $A$  is provable” is introduced, to say that a proof of  $A$  exists, but no attention is paid to what it is. Similarly, [1] introduces proof irrelevance in Martin-Löf’s logical framework using a function to distinguish propositions  $A$  from “proof-irrelevant propositions  $\text{Prf}(A)$ ”. While  $A$  can be inhabited by several normal terms,  $\text{Prf}(A)$  is inhabited by only one normal form noted  $\star$ , to which all terms of  $\text{Prf}(A)$  reduce. Still in Martin-Löf’s type theory, [31] provides proof irrelevance for predicate subtyping (here called *subset types*) for two different presentations, one is intensional, and the other extensional. The interested reader may have a look at NUPRL [11], an implementation of Martin-Löf’s Type Theory with extensional equality and subset types.

Proof irrelevance has also been added to LF to provide a new system LFI in [24], where proof irrelevance is used in the context of *refinement types*. In LFI, proof irrelevance is not limited to propositions, nor it is attached to a certain type: terms are irrelevant based on the function they are applied to. A similar system is implemented in AGDA [33].

More generally, concerning proof irrelevance in proof assistants, COQ and AGDA [18] each have a sort for proof irrelevant propositions ( $\text{SProp}$  for COQ and  $\text{Prop}$  for AGDA [33]). LEAN [14] is by design proof irrelevant, and MATITA supports proof irrelevance as well [2, Section 9.3].

## Outline

Encoding predicate subtyping requires a clear definition of it, which is done in Section 2. Predicate subtyping is encoded into  $\lambda\Pi/\equiv$  using the signatures provided in Section 3. This encoding is put in use into some examples as well. The encoding is proved correct in Section 4: any well typed term of the source language can be encoded into  $\lambda\Pi/\equiv$ , and its type in  $\lambda\Pi/\equiv$  is the encoding of its type in the source language. Finally, we show that a type checker for the  $\lambda\Pi$ -calculus modulo rewriting can be used to type check terms that have been encoded as described in Section 3.

## 2 PVS-Cert: A Minimal System With Predicate Subtyping

Because of its size, encoding the whole of PVS cannot be achieved in one step. Consequently, F. Gilbert in his PhD [17] extracted, formalised and studied a subsystem of PVS which captures the essence of predicate subtyping named PVS-CERT. Unlike PVS, PVS-CERT contains proof terms, which has for consequence that type checking is decidable in PVS-CERT while it is not in PVS. Hence PVS-CERT is a good candidate to be a logical system in which PVS proofs and specifications can be encoded to be rechecked by external tools.

In this paper, we use an equational presentation of PVS-CERT, that is, we use equations rather than reduction rules and slightly change the syntax of terms. We describe PVS-CERT, as done in [17], namely the addition of predicate subtyping over simple type theory.

### 2.1 Type Systems Modulo Theory

To describe PVS-CERT and  $\lambda\Pi/\equiv$  in a uniform way, we will use the notion of *Type Systems Modulo* described in [8]. Type Systems Modulo are an extension of *Pure Type Systems* [7] with symbols of fixed arity whose types are given by a *typing signature*  $\Sigma$ , and an arbitrary conversion relation  $\equiv$  instead of just  $\beta$ -conversion  $\equiv_\beta$ .

The terms of such a system are characterised by a finite set of *sorts*  $\mathcal{S}$ , a countably infinite set of variables  $\mathcal{V}$  and a signature  $\Sigma$ . The set of terms  $\mathcal{T}(\Sigma, \mathcal{S}, \mathcal{V})$  is inductively defined in Figure 1.

$$M, N, T, U ::= s \in \mathcal{S} \mid x \in \mathcal{V} \mid M N \mid \lambda x : T, M \mid (x : T) \rightarrow U \mid f(\vec{M})$$

$$\text{with } \Sigma(f) = (\vec{x}, \vec{T}, U, s)$$

■ **Figure 1** Terms of the type system characterised by  $\mathcal{S}, \mathcal{V}$  and  $\Sigma$ .

The contexts are noted  $\Gamma ::= \emptyset \mid \Gamma, v : T$  and the judgements  $\Gamma \vdash WF$  or  $\Gamma \vdash M : T$ . The typing rules are given in Figure 2 and depend on

- *axioms*  $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$  to type sorts;
- *product rules*  $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$  to type dependent products;
- a typing signature  $\Sigma$  which defines the function symbols and how to type their applications;
- a convertibility relation  $\equiv$ .

**Notations.** Rewriting relations are noted  $\hookrightarrow_R$ , where  $R$  is a set of rewriting rules.  $\hookrightarrow_R$  is the closure of  $R$  by substitution and context.  $\equiv_R$  is the symmetric, reflexive and transitive closure of  $\hookrightarrow_R$ . The substitution of  $x$  by  $N$  in  $M$  is noted  $\{x \mapsto N\} M$ . We use a vectorised notation

$$\begin{array}{c}
 \text{empty} \frac{}{\emptyset \vdash WF} \quad \text{decl} \frac{\Gamma \vdash T : s}{\Gamma, v : T \vdash WF} v \notin \Gamma \quad \text{var} \frac{\Gamma \vdash WF}{\Gamma \vdash v : T} v : T \in \Gamma \\
 \\
 \text{conv} \frac{\Gamma \vdash M : U \quad \Gamma \vdash T : s \quad T \equiv U}{\Gamma \vdash M : T} \quad \text{sort} \frac{\Gamma \vdash WF}{\Gamma \vdash s_1 : s_2} (s_1, s_2) \in \mathcal{A} \\
 \\
 \text{prod} \frac{\Gamma \vdash T : s_1 \quad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash (x : T) \rightarrow U : s_3} (s_1, s_2, s_3) \in \mathcal{P} \\
 \\
 \text{abst} \frac{\Gamma \vdash (x : T) \rightarrow U : s \quad \Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T, M : (x : T) \rightarrow U} \quad \text{app} \frac{\Gamma \vdash M : (x : T) \rightarrow U \quad \Gamma \vdash N : T}{\Gamma \vdash M N : \{x \mapsto N\} U} \\
 \\
 \text{sig} \frac{\overrightarrow{x : \vec{T}} \vdash U : s \quad \left( \Gamma \vdash t_i : \left\{ (x_j \mapsto t_j)_{j < i} \right\} T_i \right)_i \quad \Sigma(f) = (\overrightarrow{x : \vec{T}}, U, s)}{\Gamma \vdash f(\vec{t}) : \left\{ \overrightarrow{x \mapsto \vec{t}} \right\} U}
 \end{array}$$

■ **Figure 2** Typing rules of a TYPE SYSTEM MODULO.

for products  $\overrightarrow{(x : \vec{T})} \rightarrow U$  to represent the dependent product  $(x_1 : T_1) \rightarrow (x_2 : T_2) \rightarrow \dots (x_n : T_n) \rightarrow U$ ; and more generally for any construction that can be extended to a finite sequence, such as a parallel substitution  $\left\{ \overrightarrow{x \mapsto \vec{N}} \right\} M$ . A mapping  $\Sigma(f) = (\overrightarrow{x : \vec{T}}, U, s)$  can also be written  $\overrightarrow{x : \vec{T}} \vdash_{\Sigma} f(\vec{x}) : U : s$ . For all relations on terms  $R$  and  $S$ , we write  $RS = \{(t, u) \mid \exists v, tRv \wedge vSu\}$  the composition of  $R$  and  $S$ , and  $R^*$  the reflexive and transitive closure of  $R$ .

## 2.2 Simple Type Theory

PVS and PVS-CERT are both based on simple type theory, which can be represented by the PTS  $\lambda\text{HOL}$  [7]:

- $\mathcal{S}^{\lambda\text{HOL}} = \{\text{Prop}, \text{Type}, \text{Kind}\}$ ,
- $\mathcal{A}^{\lambda\text{HOL}} = \{(\text{Prop}, \text{Type}), (\text{Type}, \text{Kind})\}$ ,
- $\mathcal{P}^{\lambda\text{HOL}} = \{(\text{Prop}, \text{Prop}, \text{Prop}), (\text{Type}, \text{Type}, \text{Type}), (\text{Type}, \text{Prop}, \text{Prop})\}$ ,
- $\Sigma^{\lambda\text{HOL}} = \emptyset$ ,
- $\equiv^{\lambda\text{HOL}}$  is the reflexive, transitive and symmetric closure of the  $\beta$ -equation

$$((\lambda x, M) N) = \{x \mapsto N\} M \quad (\beta)$$

## 2.3 Predicate Subtyping

Predicate subtyping has two main benefits for a specification language. The first is to provide a richer type system thanks to the entanglement of type-checking and proof-checking. In consequence, any property can be encoded in the type system, which allows to easily create “guards” such as `tail : nonempty_stack → stack` where `nonempty_stack` is a predicate subtype defined from a predicate `empty?`. It is also essential in the expression of mathematics: the judgement  $M : T$  is akin to the statement  $M \in T$  in the usual language of mathematics when  $T$  is a set defined by comprehension such as  $E = \{n : \mathbb{N} \mid P(n)\}$ . With predicate subtyping, we can represent the set  $E$  by the type  $(\text{psub } \mathbb{N} \ P)$ , and the judgement  $\Gamma \vdash M : \text{psub } \mathbb{N} \ P$  is derivable if term  $M$  contains a proof of  $P(n)$  for some

*n.* The other benefit of predicate subtyping, which is essential in PVS developments, is that it separates the process of writing specifications from the proving phase. In PVS, this separation appears through *type correctness conditions* (TCC): the development of specifications creates proof obligations that may be solved at any time. This separation is also visible in usual mathematical developments, where if we want to prove that  $t \in E$ , we prove once that  $P(t)$  is valid to then forget the proof and simply use  $t$ .

The type system of PVS-CERT can be seen as  $\lambda$ HOL with a non empty signature  $\Sigma^{\text{PVS}}$  defined in Figure 3 and a richer equivalence  $\equiv_{\text{PVS}}$  that will be discussed in the next paragraph.

$$T : \text{Type}, p : T \rightarrow \text{Prop} \vdash \text{psub } T \ p \quad : \text{Type} \quad : \text{Kind} \quad (1)$$

$$T : \text{Type}, p : T \rightarrow \text{Prop}, m : T, h : p \ m \vdash \text{pair } T \ p \ m \ h : \text{psub } T \ p \quad : \text{Type} \quad (2)$$

$$T : \text{Type}, p : T \rightarrow \text{Prop}, m : \text{psub } T \ p \vdash \text{fst } T \ p \ m \quad : T \quad : \text{Type} \quad (3)$$

$$T : \text{Type}, p : T \rightarrow \text{Prop}, m : \text{psub } T \ p \vdash \text{snd } T \ p \ m \quad : p \ (\text{fst } T \ p \ m) : \text{Type} \quad (4)$$

■ **Figure 3** Signature  $\Sigma^{\text{PVS}}$  of PVS-CERT.

A predicate subtype ( $\text{psub } T \ U$ ) is defined from a *supertype*  $T$  and predicate  $U$  which binds a variable of type  $T$  to a proposition. Terms inhabiting a predicate subtype ( $\text{psub } T \ U$ ) are built with the pair construction ( $\text{pair } T \ U \ M \ N$ ) where  $M$  is a term of the supertype  $T$  and  $N$  is a proof of  $(U \ M)$ . While the pair construction allows to coerce a term from any type to a predicate subtype, the converse, that is the coercion from a type to its supertype is done with  $\text{fst}$ , the left projection of the pair. The right projection,  $\text{snd}$ , provides a witness that the left projection of the pair validates the predicate defining the subtype. Unlike PVS-CERT, PVS does not use coercions  $\text{pair}$ ,  $\text{fst}$  and  $\text{snd}$ . In PVS, subtyping is implicit: terms do not have a unique type, and the choice of this type is left to the type checker.

► **Remark 1.** Unlike the original presentation of PVS-CERT in [17], this one annotates  $\text{fst}$  and  $\text{snd}$ , using  $\text{fst } T \ p \ m$  instead of  $\text{fst } m$  to ease the well-definedness proof of the translation of PVS-CERT terms (Proposition 4).

### Equations and Proof Irrelevant Pairs

So far, no real difference has been evinced between PVS-CERT and dependent pairs: predicate subtype ( $\text{psub } T \ p$ ) may be encoded as the dependent pair type  $\Sigma x : T, p \ x$  [17, Definition 4.2.3]. The difference lies in the equivalence relations and the fact that PVS-CERT implements *proof irrelevance* in pairs.

The equivalence of PVS-CERT is noted  $\equiv_{\text{PVS}}$  and contains Equations (5), (6), and ( $\beta$ ) which provide *proof irrelevance*:

$$\text{pair } t \ u \ m \ h_0 = \text{pair } t \ u \ m \ h_1 \quad (5)$$

$$\text{fst } t_0 \ u_0 \ (\text{pair } t_1 \ u_1 \ m \ h) = m \quad (6)$$

We will now motivate the use of these equations in PVS-CERT. Proofs contained in terms are essential for typing purposes. On the other hand, these proofs are a burden regarding equivalence of terms. Were these proofs taken into account (as  $\equiv_{\beta}$  does), too many terms would be distinguished. For example, consider two terms  $t = \text{pair } \mathbb{N} \ \text{Even } 2 \ h$  and  $t' = \text{pair } \mathbb{N} \ \text{Even } 2 \ h'$  typed as even numbers. Then  $t$  and  $t'$  are not considered equal because they don't have the same proof ( $h$  and  $h'$ ) that 2 is even. We end up with one even number 2 per proof that 2 is even.

As stated in [13], most mathematicians seek convertibility of  $t$  and  $t'$  and care more about what  $h$  and  $h'$  prove than the proofs themselves. To this end, PVS-CERT has *proof irrelevant* pairs: proofs attached to terms are not taken into account when checking the equivalence of two pairs. This property is embedded in the equivalence relation  $\equiv_{\text{PVS}}$  used in the conversion rule of PVS-CERT which must verify Equation (5).

Equation (6) allows the projection to compute, but because of proof irrelevance, we cannot allow the right projection to compute, otherwise, all terms of type *Prop* would be considered equivalent.

A proof of  $T \equiv_{\beta} U$  or  $T \equiv U$  can use untyped intermediate terms, which can be problematic when one wants to prove some property on typed terms only. In the case of  $\equiv_{\beta}$ , the problem is solved by using the fact that  $\hookrightarrow_{\beta}$  is confluent, that is  $\equiv_{\beta} = \hookrightarrow_{\beta}^* \hookrightarrow_{\beta}^*$ . We now prove a similar property for  $\equiv_{\text{PVS}}$ :

► **Lemma 2** (Properties of the PVS-CERT conversion). *Let  $\hookrightarrow_{\beta\text{fst}} = \hookrightarrow_{\beta} \cup \hookrightarrow_{\text{fst}}$  where  $\hookrightarrow_{\text{fst}}$  is the closure by substitution and context of Equation (6) oriented from left to right, and let  $\leftrightarrow_{pi}$  be the closure by substitution and context of Equation (5) and  $=_{pi} = \leftrightarrow_{pi}^*$ .*

*For all relation on terms  $R$ , let  $R^{ty}$  be the restriction of  $R$  to typable terms. Then:*

- $\equiv_{\text{PVS}} \subseteq \hookrightarrow_{\beta\text{fst}}^* =_{pi} \hookrightarrow_{\beta\text{fst}}^*$
- $\hookrightarrow_{\beta\text{fst}}$  preserves typing: if  $\Gamma \vdash_{\text{PVS}} M : T$  and  $M \hookrightarrow_{\beta\text{fst}} M'$ , then  $\Gamma \vdash_{\text{PVS}} M' : T$
- $\equiv_{\text{PVS}}^{ty} \subseteq \left( \hookrightarrow_{\beta\text{fst}}^{ty} \right)^* \left( \leftrightarrow_{pi}^{ty} \right)^* \left( \hookrightarrow_{\beta\text{fst}}^{ty} \right)^*$ ,

**Proof.** A relation  $\hookrightarrow$  is confluent modulo some relation  $E$  if  $\hookrightarrow^* \hookrightarrow^* \subseteq \hookrightarrow^* E \hookrightarrow^*$ . If  $E = \emptyset$ , we simply say that  $\hookrightarrow$  is confluent.

First note that  $\hookrightarrow_{\beta\text{fst}}$  is confluent since it can be seen as a Combinatory Reduction System that is orthogonal (i.e. whose rules are left-linear and non-overlapping) [22].

We now prove that  $\leftrightarrow_{pi}$  steps can be postponed:  $\leftrightarrow_{pi} \hookrightarrow_{\beta\text{fst}} \subseteq \hookrightarrow_{\beta\text{fst}}^* =_{pi}$ , where  $\hookrightarrow_{\beta\text{fst}}^*$  is the reflexive closure of  $\hookrightarrow_{\beta\text{fst}}$ . Assume that the  $\leftrightarrow_{pi}$  step is at position  $p$  and the  $\hookrightarrow_{\beta\text{fst}}$  step is at position  $q$ . If  $p$  and  $q$  are disjoint, this is immediate. If  $p$  is above  $q$ , we have  $\text{pair } T \ U \ M \ N_1 \leftrightarrow_{pi} \text{pair } T \ U \ M \ N_2$  and either  $\text{pair } T \ U \ M \ N_2 \hookrightarrow_{\text{fst}} M$  or  $\text{pair } T \ U \ M \ N_2 \hookrightarrow_{\beta\text{fst}} \text{pair } T' \ U' \ M' \ N'_2$ . In the first case,  $\text{pair } T \ U \ M \ N_1 \hookrightarrow_{\text{fst}} M$ . In the second case,  $\text{pair } T \ U \ M \ N_1 \hookrightarrow_{\beta\text{fst}}^* \text{pair } T' \ U' \ M' \ N_1 \leftrightarrow_{pi} \text{pair } T' \ U' \ M' \ N'_2$ . Finally, if  $q$  is above  $p$ , we have  $(\lambda x : T, M)N \leftrightarrow_{pi} (\lambda x : T', M')N' \hookrightarrow_{\beta\text{fst}} \{x \mapsto N'\} M'$  and  $(\lambda x : T, M)N \hookrightarrow_{\beta\text{fst}} \{x \mapsto N\} M =_{pi} \{x \mapsto N'\} M'$ , and similarly in the case of a fst step.

Hence, (1)  $\hookrightarrow_{\beta\text{fst}}$  is confluent modulo  $=_{pi}$ , that is,  $\equiv_{\text{PVS}} \subseteq \hookrightarrow_{\beta\text{fst}}^* =_{pi} \hookrightarrow_{\beta\text{fst}}^*$ .

We now prove that (2)  $\hookrightarrow_{\beta}$  preserves typing. To this end, it suffices to prove that, if  $(x : T) \rightarrow U$  and  $(x : T') \rightarrow U'$  are typable, and  $(x : T) \rightarrow U \equiv_{\text{PVS}} (x : T') \rightarrow U'$ , then  $T \equiv_{\text{PVS}} T'$  and  $U \equiv_{\text{PVS}} U'$  (see [9] for more details), which follows from (1).

We now prove that (3)  $\hookrightarrow_{\text{fst}}$  preserves typing. Assume that  $\text{fst } T_0 \ P_0 (\text{pair } T_1 \ P_1 \ M \ N)$  is of type  $C$ . By inversion of typing rules,  $\text{pair } T_1 \ P_1 \ M \ N$  is of type  $\text{psub } T_0 \ P_0$  and  $T_0 \equiv_{\text{PVS}} C$ . By inversion again,  $M$  is of type  $T_1$  and  $\text{psub } T_0 \ U_0 \equiv_{\text{PVS}} \text{psub } T_1 \ P_1$ . By (1),  $T_0 \equiv_{\text{PVS}} T_1$  and  $P_0 \equiv_{\text{PVS}} P_1$ . Therefore,  $M$  is of type  $C$ .

Next, note that (4)  $=_{pi} = \leftrightarrow_{pi}$  where  $\leftrightarrow_{pi}$  consists in applying several  $\leftrightarrow_{pi}$  steps at disjoint positions. Indeed, if  $t = \text{pair } T \ P \ M \ N_1 \leftrightarrow_{pi} u = \text{pair } T \ P \ M (\dots (\text{pair } T' \ P' \ M' \ N'_1) \dots) \leftrightarrow_{pi} v = \text{pair } T \ P \ M (\dots (\text{pair } T' \ P' \ M' \ N'_2) \dots)$ , then  $t \leftrightarrow_{pi} v$  as well.

Moreover, we have (5)  $\leftrightarrow_{pi}^{ty} = (\leftrightarrow_{pi}^{ty})^*$ . Indeed,  $A \leftrightarrow_{pi}^{ty} B$  means that we can obtain  $B$  from  $A$  by replacing some subterms of  $A$ , that are typable since  $A$  is typable, by some subterms of  $B$ , that are typable since  $B$  is typable.

We can now conclude as follows. Assume that  $A \equiv_{\text{PVS}}^{ty} B$ . By (1), there are  $A'$  and  $B'$  such that  $A \xrightarrow{\beta_{\text{fst}}}^* A' =_{\text{pi}} B' \xrightarrow{\beta_{\text{fst}}}^* B$ . By (2), (3), (4) and (5),  $A(\xrightarrow{\beta_{\text{fst}}}^{ty})^* A'(\xrightarrow{\beta_{\text{fst}}}^{ty})^* B'(\xrightarrow{\beta_{\text{fst}}}^{ty})^* B$ . ◀

### 3 Encoding PVS-Cert in $\lambda\Pi/\equiv$

We provide an encoding of PVS-CERT into the logical framework  $\lambda\Pi/\equiv$ . This encoding allows to express terms of PVS-CERT into  $\lambda\Pi/\equiv$ . Because logical frameworks strive to remain minimal, constructions such as pair or psub are not built-in: they must be expressed into the language of the logical framework through an encoding. We hence define the symbols allowing to emulate predicate subtyping using the terms of  $\lambda\Pi/\equiv$ .

#### Definition of $\lambda\Pi/\equiv$

$\lambda\Pi/\equiv$  is the family of Type Systems Modulo whose sorts, axioms and product rules are:

- sorts  $\mathcal{S}^{\lambda\Pi} = \{\text{TYPE}, \text{KIND}\}$ ,
- axiom  $\mathcal{A}^{\lambda\Pi} = \{(\text{TYPE}, \text{KIND})\}$ ,
- product rules  $\mathcal{P}^{\lambda\Pi} = \{(\text{TYPE}, \text{TYPE}, \text{TYPE}), (\text{TYPE}, \text{KIND}, \text{KIND})\}$ .

#### 3.1 Encoding Simple Type Theory

The encoding of  $\lambda\text{HOL}$  given in Figures 4 and 5 follows the method settled in [12] for pure type systems.

In the following, we write the function symbols of a signature in blue and the other constructions of  $\lambda\Pi/\equiv$  in black, to better distinguish them.

The general idea is to manipulate types and terms of  $\lambda\text{HOL}$  as terms of  $\lambda\Pi/\equiv$ . Sorts are both objectified as **type** and **prop** and encoded as types by **Kind**, **Type** and **Prop** in Equations (7)–(11). Sorts as types are used to type sorts as objects to encode the axioms in  $\mathcal{A}$ . Terms of type *Type* are encoded as terms of type **Type**. These encoded types can be interpreted as  $\lambda\Pi/\equiv$  types with function **El** (12). Similarly, propositions are reified as terms of type **prop** and interpreted by function **Prf**. For instance, given a  $\lambda\text{HOL}$  type  $T$  and a  $\lambda\text{HOL}$  proposition  $P$  both encoded as  $\lambda\Pi/\equiv$  terms, the abstractions  $\lambda x : \text{El } T, x$  and  $\lambda h : \text{Prf } P, h$  are valid  $\lambda\Pi/\equiv$  terms. The signature exposed in Figure 4 is noted  $\Sigma^{\lambda\text{HOL}}$ .

Equations (18)–(20) are used to map encoded products to  $\lambda\Pi/\equiv$  products. Equation (17) makes sure that the objectified sort **prop** is the same as the sort **Prop** when interpreted as a type.

#### 3.2 Encoding Predicate Subtyping

Predicate subtypes are defined in Equation (21) as encoded types (i.e. terms of type **Type**) built from encoded type  $t$  and predicate defined on  $t$ . Pairs are encoded in Equation (22), where the second argument is the predicate that defines the type of the pair. The two projections are encoded in Equations (23) and (24), and we note the signature of Figure 6  $\Sigma^{\text{psub}}$ .

The signature used to encode PVS-CERT into  $\lambda\Pi/\equiv$  is  $\Sigma^{\text{PC}} = \Sigma^{\lambda\text{HOL}} \cup \Sigma^{\text{psub}}$ . The terms of the encoding are thus the terms of  $\mathcal{T}(\Sigma^{\text{PC}}, \mathcal{S}^{\lambda\Pi}, \mathcal{V})$ . The typing rules are those of  $\lambda\Pi/\equiv$  with the signature  $\Sigma^{\text{PC}}$  and the congruence  $\equiv_{\lambda\Pi}$  generated by Equations (5), (6), (17)–(20), and  $(\beta)$  where, in Equations (5) and (6), psub, pair and fst (PVS-CERT symbols in black) are replaced by **psub**, **pair** and **fst** ( $\lambda\Pi/\equiv$  symbols in blue).

$$\vdash \text{Kind} : \text{TYPE} : \text{KIND} \quad (7)$$

$$\vdash \text{Type} : \text{TYPE} : \text{KIND} \quad (8)$$

$$\vdash \text{Prop} : \text{TYPE} : \text{KIND} \quad (9)$$

$$\vdash \text{type} : \text{Kind} : \text{TYPE} \quad (10)$$

$$\vdash \text{prop} : \text{Type} : \text{TYPE} \quad (11)$$

$$t : \text{Type} \vdash \text{El } t : \text{TYPE} : \text{KIND} \quad (12)$$

$$p : \text{Prop} \vdash \text{Prf } p : \text{TYPE} : \text{KIND} \quad (13)$$

$$t : \text{Type}, p : \text{El } t \rightarrow \text{Prop} \vdash \forall t p : \text{Prop} : \text{KIND} \quad (14)$$

$$p : \text{Prop}, q : \text{Prf } p \rightarrow \text{Prop} \vdash p \Rightarrow q : \text{Prop} : \text{KIND} \quad (15)$$

$$t : \text{Type}, u : \text{El } t \rightarrow \text{Type} \vdash t \rightsquigarrow u : \text{Type} : \text{KIND} \quad (16)$$

■ **Figure 4** Signature  $\Sigma^{\lambda\text{HOL}}$  of the encoding of  $\lambda\text{HOL}$  into  $\lambda\Pi/\equiv$ .

$$\text{El prop} = \text{Prop} \quad (17)$$

$$\text{Prf}(\forall t p) = (x : \text{El } t) \rightarrow \text{Prf}(p x) \quad (18)$$

$$\text{Prf}(p \Rightarrow q) = (h : \text{Prf } p) \rightarrow \text{Prf}(q h) \quad (19)$$

$$\text{El}(t \rightsquigarrow u) = (x : \text{El } t) \rightarrow \text{El}(u x) \quad (20)$$

■ **Figure 5** Equations of the encoding of  $\lambda\text{HOL}$  into  $\lambda\Pi/\equiv$ .

### 3.3 Translation of PVS-Cert Terms Into $\lambda\Pi/\equiv$ Terms

► **Definition 3** (Translation). *Let  $\Gamma$  be a well formed context.*

- *The term translation of the terms  $M$  typable in  $\Gamma$ , noted  $[M]_{\Gamma}$ , is defined in Figures 7 and 8.*
- *The type translation of  $\text{Kind}$  and the terms  $M$  typable by a sort in  $\Gamma$ , noted  $\llbracket M \rrbracket_{\Gamma}$ , is defined in Figure 9.*
- *The context translation  $\llbracket \Gamma \rrbracket$  is defined by induction on  $\Gamma$  as*

$$\llbracket \emptyset \rrbracket = \emptyset; \quad \llbracket \Gamma, x : T \rrbracket = \llbracket \Gamma \rrbracket, x : \llbracket T \rrbracket_{\Gamma}$$

► **Proposition 4.** *The translation function  $[\cdot]$ , that maps a context and a PVS-CERT term typable in this context to a  $\lambda\Pi/\equiv$  term is well-defined.*

**Proof.** After Lemma 2 and [8, Lemma 41], the types of a term are unique up to equivalence. Moreover, the arguments of the translation function are decreasing with respect to the (strict) subterm relation. ◀

### 3.4 Examples of Encoded Theories

We provide here some examples that take advantage of proof irrelevance or predicate subtyping. While these examples could have been presented in PVS-CERT, we unfold them into the encoding of PVS-CERT into  $\lambda\Pi/\equiv$  to show how it can be used in practice. All examples are



$$t : \text{Type}, p : \text{El } t \rightarrow \text{Prop} \vdash \text{psub } t p \quad : \text{Type} \quad : \text{TYPE} \quad (21)$$

$$t : \text{Type}, p : \text{El } t \rightarrow \text{Prop}, m : \text{El } t, h : \text{Prf}(p m) \vdash \text{pair } t p m h : \text{El}(\text{psub } t p) \quad : \text{TYPE} \quad (22)$$

$$t : \text{Type}, p : \text{El } t \rightarrow \text{Prop}, m : \text{El}(\text{psub } t p) \vdash \text{fst } t p m \quad : \text{El } t \quad : \text{TYPE} \quad (23)$$

$$t : \text{Type}, p : \text{El } t \rightarrow \text{Prop}, m : \text{El}(\text{psub } t p) \vdash \text{snd } t p m \quad : \text{Prf}(p(\text{fst } t p m)) : \text{TYPE} \quad (24)$$

■ **Figure 6** Signature  $\Sigma^{\text{psub}}$  of the encoding of predicate subtyping into  $\lambda\Pi/\equiv$ .

$$\begin{aligned} [x]_{\Gamma} &= x \\ [\text{Prop}]_{\Gamma} &= \text{prop} \\ [\text{Type}]_{\Gamma} &= \text{type} \\ [M N]_{\Gamma} &= [M]_{\Gamma} [N]_{\Gamma} \\ [\lambda x : T, M]_{\Gamma} &= \lambda x : \text{El } [T]_{\Gamma}, [M]_{\Gamma, x:T} \\ [(x : T) \rightarrow U]_{\Gamma} &= [T]_{\Gamma} \rightsquigarrow (\lambda x : [T]_{\Gamma}, [U]_{\Gamma, x:T}) \\ &\quad \text{when } \Gamma \vdash_{\text{PVS}} T : \text{Type} \text{ and } \Gamma, x : T \vdash_{\text{PVS}} U : \text{Type} \\ [(x : T) \rightarrow P]_{\Gamma} &= \forall [T]_{\Gamma} (\lambda x : [T]_{\Gamma}, [P]_{\Gamma, x:T}) \\ &\quad \text{when } \Gamma \vdash_{\text{PVS}} T : \text{Type} \text{ and } \Gamma, x : T \vdash_{\text{PVS}} P : \text{Prop} \\ [(h : P) \rightarrow Q]_{\Gamma} &= [P]_{\Gamma} \Rightarrow (\lambda h : [P]_{\Gamma}, [Q]_{\Gamma, h:P}) \\ &\quad \text{when } \Gamma \vdash_{\text{PVS}} P : \text{Prop} \text{ and } \Gamma, h : P \vdash_{\text{PVS}} Q : \text{Prop} \end{aligned}$$

■ **Figure 7** Translation from  $\lambda\text{HOL}$  to  $\lambda\Pi/\equiv$ .

$$\begin{aligned} [\text{psub } T P]_{\Gamma} &= \text{psub } [T]_{\Gamma} [P]_{\Gamma} & [\text{fst } T P M]_{\Gamma} &= \text{fst } [T]_{\Gamma} [P]_{\Gamma} [M]_{\Gamma} \\ [\text{pair } T P M N]_{\Gamma} &= \text{pair } [T]_{\Gamma} [P]_{\Gamma} [M]_{\Gamma} [N]_{\Gamma} & [\text{snd } T P M]_{\Gamma} &= \text{snd } [T]_{\Gamma} [P]_{\Gamma} [M]_{\Gamma} \end{aligned}$$

■ **Figure 8** Translation from PVS-CERT to  $\lambda\Pi/\equiv$ .

$$\begin{aligned} \llbracket T \rrbracket_{\Gamma} &= \text{El } [T]_{\Gamma} & \text{when } \Gamma \vdash_{\text{PVS}} T : \text{Type}; & \llbracket \text{Kind} \rrbracket &= \text{Kind} \\ \llbracket T \rrbracket_{\Gamma} &= \text{Prf } [T]_{\Gamma} & \text{when } \Gamma \vdash_{\text{PVS}} T : \text{Prop}; & \llbracket \text{Type} \rrbracket &= \text{Type} \end{aligned}$$

■ **Figure 9** Translation of types from PVS-CERT to  $\lambda\Pi/\equiv$ .

## 6:10 Predicate Subtyping with Proof Irrelevance in $\lambda\Pi/\equiv$

```
symbol stack : Type;          symbol empty : El stack;          symbol t : Type;
symbol nonempty_stack?(s : El stack) := s ≠ empty;
symbol nonempty_stack := psub nonempty_stack?;
symbol push : El stack → El t → El nonempty_stack;
symbol pop : El nonempty_stack → El stack;
symbol pop_push(x : El t)(s : El stack) : Prf(pop(push x s) = s);
symbol pop2push2(x y : El t)(s : El stack)
  : Prf(pop(pair (pop(push x (fst(push y s)))) ?0) = s) := ...;
```

■ **Figure 10** Specification for stacks.

available as DEDUKTI files<sup>1</sup> and can be type-checked with LAMBDAPI<sup>2</sup>. In the examples, the first two arguments of `fst`, `pair` and `snd` are implicit.

► **Example 5** (Stacks with predicate subtypes). This example comes from the language reference manual of PVS [26] and illustrates the use of predicate subtyping and the generation of TCC through a specification of stacks in Figure 10.

Predicate subtyping is used to define the type of nonempty stacks, which allows the function `pop` to be total. Symbol `pop_push` is an axiom that uses Leibniz equality `=` on stacks. In the definition of the theorem `pop2push2`, term `?0` is a meta-variable that must be instantiated with a proof that the first argument of the pair is not empty, and represents, in the encoding, the TCC generated by PVS. We can thus see that the concept of TCC of PVS has a clear and explicit representation in the encoding, allowing its benefits to be transported to  $\lambda\Pi/\equiv$ .

► **Example 6** (Bounded lists and proof irrelevance). This example is inspired by sorted lists in the AGDA manual [33]<sup>3</sup>. Because we have not encoded dependent types, we cannot encode the type of lists bounded by a variable. We thus declare the bound in the signature. The specification is given in Figure 11.

We first notice that the predicate subtype allows to encode the proof `head ≤ bound` passed as a standalone argument in AGDA in the type of an argument in our encoding, providing a shorter type for `bcons`. In Figure 12, we define two (non-convertible) axioms `p1` and `p2` as proofs of `zero ≤ suc bound`, and two lists containing `zero` but proved to be bounded by `suc bound` using `p1` for `ℓ1` and `p2` for `ℓ2`. Type checking `ℓi` requires axioms `pi`. These axioms are like TCC's in PVS. Assuming that one wants to prove `ℓ1 = ℓ2`, had we lacked proof irrelevance, we would have had to prove that `p1 ≡ p2`, which is not possible. In our case, the equality is simply the result of `refl ℓ1`.

## 4 Correctness of the Encoding

In this section, we prove that the encoding is correct: if a PVS-CERT type is inhabited then its translation is inhabited too. Any type-checker for  $\lambda\Pi/\equiv$  could thus be used to recheck PVS-CERT typings. However, to make sure that our encoding is faithful (the encoding that

<sup>1</sup> <https://github.com/Deducteam/personoj/paper/>

<sup>2</sup> <https://github.com/Deducteam/lambdapi>, commit 0875521

<sup>3</sup> <https://agda.readthedocs.io/en/v2.5.4/language/irrelevance.html>

```

symbol zero : El ℕ;
symbol suc(n : El ℕ) : El ℕ;
symbol ≤ (nm : El ℕ) : Prop;

symbol bound := ...;
symbol blist : Type;
symbol bnil : El blist;

symbol bounded := psub(λn, n ≤ bound);
symbol bcons(head : El bounded)(tail : El blist) : El blist;

```

■ **Figure 11** Specification of sorted lists.

```

symbol p1 : Prf(zero ≤ suc bound);
symbol p2 : Prf(zero ≤ suc bound);

symbol ℓ1 := bcons(pair zero p1) bnil;
symbol ℓ2 := bcons(pair zero p2) bnil;

```

■ **Figure 12** Definition of two sorted lists with different proofs.

maps any PVS-CERT term to the same well-typed ground term is correct, but useless), completeness (also called conservativity) ought to be proved too: a PVS-CERT type is inhabited whenever its encoding is inhabited. However, as completeness is often difficult to establish (see [3, 34]), we leave it for future work.

In the following,

- $s$  stands for *Type*, *Prop* or *Kind*;
- $T, U$  designate terms of type *Type*;
- $M, N, t, u$  designate expressions that have a type  $T : \textit{Type}$ ;
- $P, Q$  are propositions of type *Prop*, or predicates of type  $T \rightarrow \textit{Prop}$ ;
- $h$  stands for a proof typed by a proposition.

Typing judgements in PVS-CERT are noted with  $\vdash_{\text{PVS}}$ , and typing judgements in  $\lambda\Pi/\equiv$  are noted with  $\vdash_{\lambda\Pi/\equiv}$ .

► **Lemma 7** (Preservation of substitution). *If  $\Gamma, x : U, \Delta \vdash_{\text{PVS}} M : T$  and  $\Gamma \vdash_{\text{PVS}} N : T$ , then  $\{x \mapsto N\} M \vdash_{\Gamma, \{x \mapsto N\} \Delta} = \{x \mapsto [N]_{\Gamma}\} [M]_{\Gamma, x : U, \Delta}$ .*

**Proof.** By structural induction on  $M$ . ◀

► **Lemma 8** (Preservation of equivalence). *Let  $M$  and  $N$  be two well typed terms in  $\Gamma$ .*

1. *If  $M \overset{\text{PVS}}{\leftrightarrow} N$ , then  $[M]_{\Gamma} \overset{\lambda\Pi}{\equiv} [N]_{\Gamma}$ .*
2. *If  $M \overset{\text{PVS}}{\equiv} N$ , then  $[M]_{\Gamma} \overset{\lambda\Pi}{\equiv} [N]_{\Gamma}$ .*

**Proof.** Each item is proved separately.

1. Taking back the notations of the proof of Lemma 2, we show that
  - a. computational steps of  $\overset{ty}{\leftrightarrow}_{\beta\text{fst}}$  are preserved,
  - b. equational steps of  $\overset{ty}{\leftrightarrow}_{\rho_i}$  are preserved.

These two properties are shown by induction on a context  $C$  such that  $M = C[\hat{M}] R C[\hat{N}] = N$  where  $R$  is any of the two relations applied at the head of  $\hat{M}$  and  $\hat{N}$ . We will only detail the base cases of inductions, the other cases being straightforward.

**Preservation of Computation** There are two possible cases,

**Case**  $M = ((\lambda x, t) u) \hookrightarrow_{\beta} \{x \mapsto u\} t$ , we have,

$$[(\lambda x : U, t) u]_{\Gamma} = ((\lambda x : \llbracket U \rrbracket_{\Gamma}, [t]_{\Gamma, x:U}) [u]_{\Gamma}) = \{x \mapsto [u]_{\Gamma}\} [t]_{\Gamma} \equiv_{\lambda\Pi} [\{x \mapsto u\} t]_{\Gamma}$$

where the equivalence is given by Lemma 7.

**Case**  $M = \text{fst } T_1 P_1 (\text{pair } T_0 P_0 t h) \hookrightarrow_{\text{fst}} t$ , we have the following equalities

$$\begin{aligned} [\text{fst } T_1 P_1 (\text{pair } T_0 P_0 t h)]_{\Gamma} &= \text{fst } [T_1]_{\Gamma} [P_1]_{\Gamma} [\text{pair } T_0 P_0 t h]_{\Gamma} \\ &= \text{fst } [T_1]_{\Gamma} [P_1]_{\Gamma} (\text{pair } [T_0]_{\Gamma} [P_0]_{\Gamma} [t]_{\Gamma} [h]_{\Gamma}) \\ &\equiv_{\lambda\Pi} [t]_{\Gamma} \end{aligned}$$

with the last equivalence provided by Equation (6).

**Preservation of Proof Irrelevance** Assume that  $M = \text{pair } T P t h \leftrightarrow_{pi} \text{pair } T P t h'$

$$[\text{pair } T P t h]_{\Gamma} = \text{pair } [T]_{\Gamma} [P]_{\Gamma} [t]_{\Gamma} [h]_{\Gamma} \equiv_{\lambda\Pi} \text{pair } [T]_{\Gamma} [P]_{\Gamma} [t]_{\Gamma} [h']_{\Gamma} = [\text{pair } T P t h']_{\Gamma}$$

where the equivalence is given by Equation (5).

2. By Lemma 2, we know that there are  $H_0$  and  $H_1$  such that  $M(\hookrightarrow_{\beta\text{fst}}^{ty})^* H_0(\hookrightarrow_{pi}^{ty})^* H_1(\hookrightarrow_{\beta\text{fst}}^{ty})^* N$ . For  $R \in \{\leftrightarrow_{pi}, \hookrightarrow_{\beta\text{fst}}\}$ , we have  $t(R^{ty})^* u \Rightarrow [t] \equiv_{\lambda\Pi} [u]$  by induction on the number of  $R^{ty}$  steps, using Item 1 for the base case. Therefore,  $[M]_{\Gamma} \equiv_{\lambda\Pi} [H_0]_{\Gamma} \equiv_{\lambda\Pi} [H_1]_{\Gamma} \equiv_{\lambda\Pi} [N]_{\Gamma}$ , which gives, by transitivity of  $\equiv_{\lambda\Pi}$ ,  $[M]_{\Gamma} \equiv_{\lambda\Pi} [N]_{\Gamma}$ . ◀

► **Theorem 9 (Correctness).** *If  $\Gamma \vdash_{\text{PVS}} M : T$ , then  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} [M]_{\Gamma} : \llbracket T \rrbracket_{\Gamma}$ . For all  $\Gamma$ , if  $\Gamma \vdash_{\text{PVS}} WF$ , then  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} WF$ .*

**Proof.** By induction on the typing derivation of  $\Gamma \vdash_{\text{PVS}} M : T$  and case distinction on the last inference rule.

**empty**  $\emptyset \vdash_{\text{PVS}} WF$

We have  $\llbracket \emptyset \rrbracket = \emptyset$  and  $\emptyset \vdash_{\lambda\Pi/\equiv} WF$ .

**decl**  $\frac{\Gamma \vdash_{\text{PVS}} T : s}{\Gamma, v : T \vdash_{\text{PVS}} WF} v \notin \Gamma$

We have  $\llbracket \Gamma, v : T \rrbracket = \llbracket \Gamma \rrbracket, v : \llbracket T \rrbracket_{\Gamma}$ . By induction hypothesis, we have  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} [T]_{\Gamma} : [s]_{\Gamma}$ , for  $s \in \mathcal{S}$  and hence  $[s]_{\Gamma}$  is either **Prop** by conversion (because **El prop**  $\equiv_{\lambda\Pi}$  **Prop**), **Type** or **Kind**. If  $s$  is **Kind**, then  $T$  is **Type**. Since  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} \text{Type} : \text{TYPE}$  because  $\Sigma^{\text{PC}}(\text{Type}) = (\vec{0}, (\text{Type}, \text{TYPE}))$ , we can derive with the declaration rule  $\llbracket \Gamma, v : T \rrbracket \vdash_{\lambda\Pi/\equiv} WF$  because  $\llbracket \text{Type} \rrbracket = \text{Type}$ . Otherwise,  $s$  is **Type** or **Prop** and  $\llbracket T \rrbracket = \xi [T]_{\Gamma}$  where  $\xi$  is **El** or **Prf**. By typing of **El** or **Prf** (with the signature),  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} \llbracket T \rrbracket_{\Gamma} : \text{TYPE}$  and finally,  $\llbracket \Gamma, v : T \rrbracket \vdash_{\lambda\Pi/\equiv} WF$  by application of the declaration rule.

**var**  $\frac{\Gamma \vdash_{\text{PVS}} WF}{\Gamma \vdash_{\text{PVS}} v : T} v : T \in \Gamma$

By definition,  $[v] = v$  and by induction hypothesis,  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} WF$ . Since  $v : T \in \Gamma$ , by definition, there is  $\Delta \subsetneq \Gamma$ ,  $\Delta \vdash_{\text{PVS}} WF$  such that,  $v : [T]_{\Delta} \in \llbracket \Gamma \rrbracket$ . Hence  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} v : [T]_{\Delta}$  and finally  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} v : [T]_{\Gamma}$  because contexts are well formed.

**sort**  $\frac{\Gamma \vdash_{\text{PVS}} WF}{\Gamma \vdash_{\text{PVS}} s_1 : s_2} (s_1, s_2) \in \mathcal{A}$

First,  $[s_1]$  is either **prop** or **type**. In the former case,  $[s_2] = \text{Type}$  and because  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} WF$  (by induction hypothesis) and  $\Sigma^{\text{PC}}(\text{prop}) = (\vec{0}, (\text{Type}, \text{TYPE}))$ , we have  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} \text{prop} : \text{Type}$ . The same procedure holds for  $s_1 = \text{Type}$  and  $s_2 = \text{Kind}$ .

$$\text{prod} \frac{\Gamma \vdash_{\text{PVS}} T : s_1 \quad \Gamma, x : T \vdash_{\text{PVS}} U : s_2}{\Gamma \vdash_{\text{PVS}} (x : T) \rightarrow U : s_3} (s_1, s_2, s_3) \in \mathcal{P}$$

We only detail for the product (*Type, Prop, Prop*), others being processed similarly. We have  $[(x : T) \rightarrow U]_{\Gamma} = \forall [T]_{\Gamma} (\lambda x : [T]_{\Gamma}, [U]_{\Gamma, x:T})$ . By induction hypothesis,  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [T] : [\text{Type}]$ , and thus  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [T] : \text{Type}$  by definition. By induction hypothesis,  $[\Gamma, x : T] \vdash_{\lambda\Pi/\equiv} [U] : [\text{Prop}]$ , and thus  $[\Gamma], x : [T]_{\Gamma} \vdash_{\lambda\Pi/\equiv} [U] : \text{Prop}$  by definition of  $[\cdot]$  and conversion which yields  $[\Gamma] \vdash_{\lambda\Pi/\equiv} \lambda x : [T]_{\Gamma}, [U]_{\Gamma, x:T} : [T]_{\Gamma} \rightarrow \text{Prop}$ .

To finish, we obtain  $[\Gamma] \vdash_{\lambda\Pi/\equiv} \lambda x : [T]_{\Gamma}, [U]_{\Gamma, x:T} : (\text{El } [T]_{\Gamma}) \rightarrow \text{Prop}$  by conversion. Using the typing signature  $\Sigma^{\text{PC}}$ ,  $[\Gamma] \vdash_{\lambda\Pi/\equiv} \forall [T]_{\Gamma} (\lambda x, [T]_{\Gamma}[U]_{\Gamma, x:T}) : \text{Prop}$  which becomes, by conversion  $\text{Prop} \equiv_{\lambda\Pi} \text{El prop}$  and definition of  $[\cdot]_{\Gamma}$ :  $\text{El prop} = [\text{Prop}]$ , hence,  $[\Gamma] \vdash_{\lambda\Pi/\equiv} \forall [T]_{\Gamma} (\lambda x, [T]_{\Gamma}[U]_{\Gamma, x:T}) : [\text{Prop}]$

$$\text{abst} \frac{\Gamma, v : T \vdash_{\text{PVS}} M : U \quad \Gamma \vdash_{\text{PVS}} (v : T) \rightarrow U : s}{\Gamma \vdash_{\text{PVS}} \lambda v : T, M : (v : T) \rightarrow U}$$

We have  $[\lambda v : T, M]_{\Gamma} = \lambda v : [T]_{\Gamma}, [M]_{\Gamma}$ . By induction hypothesis,  $[\Gamma, v : T] \vdash_{\lambda\Pi/\equiv} [M]_{\Gamma, v:T} : [U]_{\Gamma, v:T}$  and by definition of  $[\cdot]$ ,  $[\Gamma], v : [T]_{\Gamma} \vdash_{\lambda\Pi/\equiv} [M]_{\Gamma, v:T} : [U]_{\Gamma, v:T}$ . Applying the abstraction rule in  $\lambda\Pi/\equiv$ , we obtain  $[\Gamma] \vdash_{\lambda\Pi/\equiv} \lambda v : [T]_{\Gamma}, [M]_{\Gamma, v:T} : (v : [T]_{\Gamma}) \rightarrow [U]_{\Gamma, v:T}$  (with the product well typed in  $\lambda\Pi/\equiv$  since  $[U]$  and  $[T]$  are both of type **TYPE** and thus the product is of type **TYPE** as well).

Finally, we proceed by case distinction on sorts  $s_T$  and  $s_U$  such that  $\Gamma \vdash_{\text{PVS}} T : s_T$  and  $\Gamma \vdash_{\text{PVS}} U : s_U$ . We will detail the case  $(s_T, s_U) = (\text{Type}, \text{Prop})$ . We have  $(v : [T]_{\Gamma}) \rightarrow [U]_{\Gamma, v:T} \equiv_{\lambda\Pi} \text{Prf}(\forall [T]_{\Gamma} (\lambda x : [T]_{\Gamma}, [U]_{\Gamma, v:T})) = [(v : T) \rightarrow U]_{\Gamma}$  which allows to conclude.

$$\text{app} \frac{\Gamma \vdash_{\text{PVS}} M : (v : T) \rightarrow U \quad \Gamma \vdash_{\text{PVS}} N : T}{\Gamma \vdash_{\text{PVS}} M N : \{v \mapsto N\} U}$$

By induction hypothesis and conversion, we have  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [M]_{\Gamma} : (v : [T]_{\Gamma}) \rightarrow [U]_{\Gamma, v:T}$  (shown by case distinction on the sorts of  $T$  and  $U$ ) and  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [N]_{\Gamma} : [T]_{\Gamma}$ . Since  $[M N]_{\Gamma} = [M] [N]$ , we obtain using the application rule  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [M N] : \{v \mapsto [N]_{\Gamma}\} [U]_{\Gamma, v:T}$  and by Lemma 7, we obtain  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [M N] : [\{v \mapsto N\} U]_{\Gamma}$ .

$$\text{conv} \frac{\Gamma \vdash_{\text{PVS}} M : U \quad \Gamma \vdash_{\text{PVS}} T : s \quad T \equiv_{\text{PVS}} U}{\Gamma \vdash_{\text{PVS}} M : T}$$

By hypothesis, there is a type  $U$  such that  $\Gamma \vdash_{\text{PVS}} M : U$ , and  $T \equiv_{\text{PVS}} U$ , and there is a sort  $s$  such that  $\Gamma \vdash_{\text{PVS}} T : s$ . By induction hypothesis,  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [M]_{\Gamma} : [U]_{\Gamma}$ .

We now prove that if  $T \equiv_{\text{PVS}} U$ , then  $[T]_{\Gamma} \equiv_{\lambda\Pi} [U]_{\Gamma}$  and  $\Gamma \vdash_{\lambda\Pi/\equiv} [T] : \text{TYPE}$ : it will allow us to conclude using the conversion rule in  $\lambda\Pi/\equiv$ .

By Lemma 2, we have  $T \xrightarrow{*}_{\beta\text{fst}} T' =_{\text{pi}} U' \xrightarrow{*}_{\beta\text{fst}} U$  and  $T(\xrightarrow{*}_{\beta\text{fst}} T')^* (\xrightarrow{*}_{\text{pi}} U')^* (\xrightarrow{*}_{\beta\text{fst}} U)^*$ . Because  $\xrightarrow{*}_{\beta\text{fst}}$  preserves typing (Lemma 2), we have  $\Gamma \vdash_{\text{PVS}} U' : s$ . By [8, Lemma 43],  $\Gamma \vdash_{\text{PVS}} T : s$ . By Lemma 8,  $[T]_{\Gamma} \equiv_{\lambda\Pi} [U]_{\Gamma}$

If  $s = \text{Prop}$ , then  $[T]_{\Gamma} = \text{Prf } [T]_{\Gamma} \equiv_{\lambda\Pi} \text{Prf } [U]_{\Gamma} = [U]_{\Gamma}$ . Moreover we have  $[\Gamma] \vdash_{\lambda\Pi/\equiv} [T]_{\Gamma} : \text{TYPE}$  because, by induction hypothesis,  $[T]_{\Gamma} : [\text{Prop}] = \text{El } [\text{Prop}] = \text{El prop} = \text{Prop}$ , and  $(p : \text{Prop} \vdash_{\Sigma^{\text{PC}}} \text{Prf } p : \text{TYPE} : \text{KIND})$ . If  $s = \text{Type}$ ,  $[T]_{\Gamma} = \text{El } [T]_{\Gamma} \equiv_{\lambda\Pi} \text{El } [U]_{\Gamma} = [U]_{\Gamma}$ . By induction hypothesis,  $[T]_{\Gamma} : [\text{Type}]_{\Gamma} = \text{Type}$ . If  $s = \text{Kind}$ , then  $T = U = \text{Type}$  (*Type* is the only inhabitant of *Kind*). Finally,  $[\text{Type}] = \text{Type} : \text{TYPE}$ .

$$\text{sig} \frac{\overrightarrow{x : \vec{T}} \vdash U : s \quad \left( \Gamma \vdash t_i : \left\{ (x_j \mapsto t_j)_{j < i} \right\} T_i \right)_i \quad \Sigma(f) = (\overrightarrow{x, \vec{T}}, U, s)}{\Gamma \vdash f(\vec{t}) : \left\{ \overrightarrow{x \mapsto \vec{t}} \right\} U}$$

We first observe from Figure 6 that for each  $f \in \Sigma^{\text{PVS}}$ , we have a counterpart symbol  $\hat{f} \in \Sigma^{\text{PC}}$  such that if  $\Sigma^{\text{PVS}}(f) = (\overrightarrow{x : \vec{T}}, U, s)$ , then  $\Sigma^{\text{PC}}(\hat{f}) = (\overrightarrow{x, \llbracket T \rrbracket}, \llbracket U \rrbracket_{x:\vec{T}}, \text{TYPE})$ .

By induction hypothesis, for each  $i$ , we have  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} [t_i]_{\Gamma} : \llbracket \{(x_j \mapsto t_j)_{j < i}\} T_i \rrbracket_{\Gamma}$  which we can write as, thanks to Lemma 7,  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} [t_i]_{\Gamma} : \{(x_j \mapsto [t_j]_{\Gamma})_{j < i}\} \llbracket T_i \rrbracket_{\Gamma}$ .

Now, using the signature rule, we are able to conclude  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} \hat{f} \overrightarrow{[t]_{\Gamma}} : \overrightarrow{\{x \mapsto [t]\}} \llbracket U \rrbracket$ .

By Lemma 7, we obtain  $\llbracket \Gamma \rrbracket \vdash_{\lambda\Pi/\equiv} \hat{f} \overrightarrow{[t]_{\Gamma}} : \llbracket \overrightarrow{\{x \mapsto t\}} U \rrbracket$ . Moreover, we have taken care to define the translation in Figure 8 such that  $\llbracket f(\vec{t}) \rrbracket = \hat{f} \overrightarrow{[t]}$ .  $\blacktriangleleft$

## 5 Mechanised Type Checking

The encoding of PVS-CERT into  $\lambda\Pi/\equiv$  can be used to proof check terms of PVS-CERT using a type checker for  $\lambda\Pi/\equiv$ . But because of the rule

$$\frac{\Gamma \vdash t : B \quad \Gamma \vdash A : s \quad A \equiv B}{\Gamma \vdash t : A} \quad (\lambda\Pi/\equiv\text{-conv})$$

type checking is decidable only if  $\equiv$  is. A decidable relation equivalent to  $\equiv$  can be obtained using the convertibility relation stemming from the rewriting relation of a convergent rewrite system, yielding the type system  $\lambda\Pi/R$  ( $R$  for *rewriting*). Consequently, while type checkers cannot be provided for  $\lambda\Pi/\equiv$  in general, they can for  $\lambda\Pi/R$ , as can be seen with DEDUCTI<sup>4</sup>. Such rewrite systems can be obtained through *completion procedures* [6]. However, completion procedures rely on a well-founded order that cannot be provided here because of Equation (5) which cannot be oriented since each side of the equation has a free variable which is not in the other side.

A possible solution would be to rewrite all proofs of a pair to a canonical proof with a rule of the form

$$\text{pair } t p m h \hookrightarrow \text{pair } t p m (\text{canon } t p m)$$

where  $t : \text{Type}, p : \text{El } t \rightarrow \text{Prop}, m : \text{El } t \vdash \text{canon } t p m : \text{Prf}(p m) : \text{TYPE}$ . But this creates a rewrite rule that duplicates three variables.

Otherwise, as noted in [23], the addition of a symbol to the signature can circumvent the issue. Hence, we add a symbol for proof irrelevant pairs, and make it equal to pairs

$$t : \text{Type}, p : \text{El } t \rightarrow \text{Prop}, m : \text{El } t \vdash \text{pair}^{\dagger} t p m : \text{El}(\text{psub } t p) : \text{TYPE} \quad (25)$$

$$\text{pair } t p m h = \text{pair}^{\dagger} t p m \quad (26)$$

thus  $(\text{pair } t p m h) \equiv (\text{pair}^{\dagger} t p m) \equiv (\text{pair } t p m h')$ . The new set of identities given by Equations (6), (17)–(20), and (26) can be completed into a rewrite system  $R$  which is equivalent to the equations:

► **Proposition 10.** *Let  $\hookrightarrow_R$  be the closure by context and substitution of the rewrite rules of Figure 13, and  $\equiv_R$  be the smallest equivalence containing  $\hookrightarrow_R$ . Then, for all  $M, N \in \mathcal{T}(\Sigma^{\text{PC}}, \mathcal{S}^{\lambda\Pi}, \mathcal{V})$ , if  $M \equiv_{\lambda\Pi} N$  then  $M \equiv_R N$ .*

<sup>4</sup> <https://github.com/Deducteam/lambdapi.git>

$$\begin{array}{ll}
(\lambda x : T, t) u \hookrightarrow \{x \mapsto u\} t & (27) \quad \text{El prop} \leftrightarrow \text{Prop} & (30) \\
\text{pair } t p m h \hookrightarrow \text{pair}^\dagger t p m & (28) \quad \text{Prf}(\forall t p) \hookrightarrow (x : \text{El } t) \rightarrow \text{Prf}(p x) & (31) \\
\text{fst } t_0 p_0 (\text{pair}^\dagger t_1 p_1 m) \hookrightarrow m & (29) \quad \text{El}(t \rightsquigarrow u) \hookrightarrow (x : \text{El } t) \rightarrow \text{El}(u x) & (32) \\
& & \text{Prf}(p \Rightarrow q) \hookrightarrow (h : \text{Prf } p) \rightarrow (\text{Prf}(q h)) & (33)
\end{array}$$

■ **Figure 13** Rewrite system  $R$  resulting from the completion of the equations of the encoding of PVS-CERT in  $\lambda\Pi/\equiv$ .

**Proof.** It suffices to prove that every equation of PVS-CERT is included in  $\equiv_R$ . This is immediate for the Equations (17)–(20) and  $(\beta)$  since they are equal to the rules (27) and (30)–(33). For the Equation (5), we have  $\text{pair } t p m h_0 \hookrightarrow_R \text{pair}^\dagger t p m \leftrightarrow_R \text{pair } t p m h_1$ . Finally, for the Equation (6), we have  $\text{fst } t_0 p_0 (\text{pair } t_1 p_1 m h) \hookrightarrow_R \text{fst } t_0 p_0 (\text{pair}^\dagger t_1 p_1 m) \hookrightarrow_R m$ . ◀

- **Remark 11.** ■ Rewrite system  $R$  is confluent because it is orthogonal.
- Termination of  $R$  is required to obtain the decidability of  $\equiv_R$ . A possible approach to prove it would be to extend the termination model of  $\lambda\text{HOL}$  described in [15].
  - In order to prove the completeness of the encoding, that is, the fact that a type is inhabited whenever its encoding is, it could be useful to have the reciprocal implication, that is, if  $M \equiv_R N$  and  $M, N \in \mathcal{T}(\Sigma^{\text{PC}}, \mathcal{S}^{\lambda\Pi}, \mathcal{V})$ , then  $M \equiv_{\lambda\Pi} N$ . We leave this for future work too.

A priori, the introduction of  $\text{pair}^\dagger$  allows one to craft terms that cannot be proof checked in PVS-CERT. Indeed, given a predicate `Even` on natural numbers, the term  $(\text{pair}^\dagger \mathbb{N} \text{ Even } 3)$  is the encoding of  $(\text{pair } \mathbb{N} \text{ Even } 3 h)$  which cannot be type checked in PVS-CERT since there is no proof  $h$  that 3 is even. However, DEDUKTI relies on a system of modules and tags attached to symbols to define where and how symbols can be used. A symbol tagged *protected* cannot be used to build terms outside of the module where it is defined, but it may appear during type checking because of conversion, a trick first introduced in [35] and used also for encoding Cumulative Type Systems in  $\lambda\Pi/\equiv$  [34]. In our case, one may protect  $\text{pair}^\dagger$  in the module that defines the encoding of PVS-CERT, so that users of the encoding are forced to use `pair`.

## Conclusion

This work provides an encoding of predicate subtyping with proof irrelevance into the  $\lambda\Pi$ -calculus modulo theory,  $\lambda\Pi/\equiv$  [4]. We first recall PVS-CERT, an extension of higher-order logic with predicate subtyping and proof irrelevance [17]. We then provide a  $\lambda\Pi/\equiv$  signature to encode terms of PVS-CERT, and prove that the encoding is correct: if a PVS-CERT type is inhabited, then its translation in  $\lambda\Pi/\equiv$  is inhabited too. Finally, we show that the equational theory of our encoding is equivalent to a confluent set of rewrite rules which enable us to use DEDUKTI to type check encoded specifications.

However, two important problems are left open. First, is our encoding complete, that is, is a PVS-CERT type inhabited if its translation is? Second, is the confluent rewrite system used in the encoding terminating? We believe that these two properties hold but leave their difficult study for future work.

## Perspectives

The encoding of PVS-CERT in  $\lambda\Pi/R$  is the stepping stone towards an automatic translator from PVS to DEDUKTI. Indeed, PVS does not have proof terms in its syntax, and consequently type checking is undecidable. The creation of PVS-CERT allows to convert PVS terms to a syntax whose type checking is decidable. This was the work of F. Gilbert in [17]. Now we are able to express this decidable syntax in  $\lambda\Pi/R$  and hence in DEDUKTI. However, the type system proposed here only allows to coerce from a type to its direct supertype or a subtype, that is, we can go from  $(\text{psub } (\text{psub } \iota P) Q)$  to  $\text{psub } \iota P$  in one coercion, but we cannot coerce from  $(\text{psub } (\text{psub } \iota P) Q)$  to  $\iota$ , whereas PVS can. Consequently, an algorithm to elaborate the correct sequence of coercions is needed to obtain terms that can be type checked in DEDUKTI.

Other features of PVS can be integrated into PVS-CERT and the encoding: dependent types like  $(\text{psub } \textit{list } (\lambda\ell, \text{length } \ell = n))$ , recursive definitions of functions, and dependent records. With those features encoded, almost all the standard library<sup>5</sup> of PVS can be translated to DEDUKTI.

Finally, while the previous points were concerned with the translation of specifications from PVS, we may also want to translate proofs developed in PVS. These proofs are witnesses of *type correctness conditions* (TCC), which are required to type check terms. Since PVS is a highly automated prover, proof terms often come from application of complex tactics that cannot be mimicked into DEDUKTI. However, proof terms may either be provided by hand, emulating the interaction provided by TCC's, or we may call external solvers [19].

---

## References

- 1 Andreas Abel, Thierry Coquand, and Miguel Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. *Log. Methods Comput. Sci.*, 7(2), 2011. doi:10.2168/LMCS-7(2:4)2011.
- 2 Andrea Asperti, Wilmer Ricciotti, and Claudio Sacerdoti Coen. Matita tutorial. *J. Formaliz. Reason.*, 7(2):91–199, 2014. doi:10.6092/issn.1972-5787/4651.
- 3 A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École Polytechnique, France, 2015. URL: <https://tel.archives-ouvertes.fr/tel-01235303/>.
- 4 A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the  $\lambda\pi$ -calculus modulo theory, 2019. Draft. URL: <http://lsv.fr/~dowek/Publi/expressing.pdf>.
- 5 Ali Assaf and Guillaume Burel. Translating HOL to dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*, volume 186 of *EPTCS*, pages 74–88, 2015. doi:10.4204/EPTCS.186.8.
- 6 Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
- 7 Henk Barendregt and Kees Hemerik. Types in Lambda Calculi and Programming Languages. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 1990. doi:10.1007/3-540-52592-0\_53.
- 8 F. Blanqui. *Théorie des types et réécriture*. PhD thesis, Université Paris-Sud, France, 2001. URL: <http://tel.archives-ouvertes.fr/tel-00105522>.

---

<sup>5</sup> <http://www.cs.rug.nl/~gr1/ar06/prelude.html>



- 9 Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1), 2005. doi:10.1017/S0960129504004426.
- 10 M. Boespflug and G. Burel. CoqInE: translating the calculus of inductive constructions into the lambda-Pi-calculus modulo. In *Proceedings of the 2nd International Workshop on Proof eXchange for Theorem Proving*, CEUR Workshop Proceedings 878, 2012. URL: <http://ceur-ws.org/Vol-878/paper3.pdf>.
- 11 Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. URL: <http://dl.acm.org/citation.cfm?id=10510>.
- 12 D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007. doi:10.1007/978-3-540-73228-0\_9.
- 13 N.G. de Bruijn. Some Extensions of Automath: The AUT-4 Family. In R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 283–288. Elsevier, 1994. doi:10.1016/S0049-237X(08)70209-X.
- 14 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6\_26.
- 15 Gilles Dowek. Models and termination of proof reduction in the lambda pi-calculus modulo theory. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPICs*, pages 109:1–109:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ICALP.2017.109.
- 16 G. Genestier. encoding agda programs using rewriting. In *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 167, 2020. doi:10.4230/LIPICs.FSCD.2020.31.
- 17 Frederic Gilbert. *Extending higher-order logic with predicate subtyping : application to PVS*. Theses, Université Sorbonne Paris Cité, 2018. URL: <https://tel.archives-ouvertes.fr/tel-02058937>.
- 18 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without k. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290316.
- 19 Mohamed Yacine El Haddad, Guillaume Burel, and Frédéric Blanqui. EKSTRAKTO A tool to reconstruct dedukti proofs from TSTP files (extended abstract). In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 27–35, 2019. doi:10.4204/EPTCS.301.5.
- 20 Joe Hurd. Predicate subtyping with predicate sets. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLS 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2001. doi:10.1007/3-540-44755-5\_19.
- 21 Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997. doi:10.1109/32.588534.
- 22 Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993. doi:10.1016/0304-3975(93)90091-7.
- 23 D. Knuth and P. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning. Symbolic Computation (Artificial Intelligence)*. Springer, 1983.

- 24 William Lovas and Frank Pfenning. Refinement types for logical frameworks and their interpretation as proof irrelevance. *Log. Methods Comput. Sci.*, 6(4), 2010. doi:10.2168/LMCS-6(4:5)2010.
- 25 Zhaohui Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, UK, 1990. URL: <http://hdl.handle.net/1842/12487>.
- 26 S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.
- 27 Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag. URL: <http://www.csl.sri.com/papers/fmtrends98/>.
- 28 Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, 1997.
- 29 Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 221–230. IEEE Computer Society, 2001. doi:10.1109/LICS.2001.932499.
- 30 John M. Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.*, 24(9):709–720, 1998. doi:10.1109/32.713327.
- 31 Anne Salvesen and Jan M. Smith. The strength of the subset type in martin-löf’s type theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS ’88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 384–391. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5135.
- 32 Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006. doi:10.1007/978-3-540-74464-1\_16.
- 33 The Agda Team. *Agda Manual*. URL: <https://agda.readthedocs.io/>.
- 34 F. Thiré. *Interoperability between proof systems using the Dedukti logical framework*. PhD thesis, Université Paris-Saclay, France, 2020.
- 35 F. Thiré and G. Férey. Proof irrelevance and predicate subtyping in dedukti. [https://eatypes.cs.ru.nl/eatypes\\_pmwiki/uploads/Main/books-of-abstracts-TYPES2019.pdf](https://eatypes.cs.ru.nl/eatypes_pmwiki/uploads/Main/books-of-abstracts-TYPES2019.pdf), p. 106, 2019. Abstract of a talk given at the TYPES conference.
- 36 Benjamin Werner. On the strength of proof-irrelevant type theories. *Log. Methods Comput. Sci.*, 4(3), 2008. doi:10.2168/LMCS-4(3:13)2008.

# $\Lambda$ -Symsym: An Interactive Tool for Playing with Involutions and Types

Furio Honsell ✉

University of Udine, Italy

Marina Lenisa ✉

University of Udine, Italy

Ivan Scagnetto ✉

University of Udine, Italy

---

## Abstract

We present the web portal  **$\Lambda$ -symsym**, available at <http://158.110.146.197:31780/automata/>, for experimenting with *game semantics* of  $\lambda^!$ -calculus, and its *normalizing elementary sub-calculus*, the  $\lambda^{EAL}$ -calculus. The  $\lambda^!$ -calculus is a generalization of the  $\lambda$ -calculus obtained by introducing a modal operator  $!$ , giving rise to a *pattern*  $\beta$ -reduction. Its sub-calculus corresponds to an applicatively closed class of terms normalizing in an elementary number of steps, in which all *elementary functions* can be encoded. The game model which we consider is the *Geometry of Interaction* model  $\mathcal{I}$  introduced by Abramsky to study reversible computations, consisting of *partial involutions* over a very simple language of moves.

Given a  $\lambda^!$ - or a  $\lambda^{EAL}$ -term,  $M$ ,  **$\Lambda$ -symsym** provides:

- an abstraction algorithm  $\mathcal{A}^!$ , for compiling  $M$  into a term,  $\mathcal{A}^!(M)$ , of the *linear combinatory logic*  $\mathbf{CL}^!$ , or the normalizing combinatory logic  $\mathbf{CL}^{EAL}$ ;
- an interpretation algorithm  $\llbracket \cdot \rrbracket^{\mathcal{I}}$  yielding a specification of the partial involution  $\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}}$  in the model  $\mathcal{I}$ ;
- an algorithm,  $\mathcal{I}2\mathcal{T}$ , for synthesizing from  $\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}}$  a type,  $\mathcal{I}2\mathcal{T}(\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}})$ , in a *multimodal, intersection type assignment discipline*,  $\vdash_!$ ;
- an algorithm,  $\mathcal{T}2\mathcal{I}$ , for synthesizing a specification of a partial involution from a type in  $\vdash_!$ , which is an inverse to the former.

We conjecture that  $\vdash_! M : \mathcal{I}2\mathcal{T}(\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}})$ .  **$\Lambda$ -symsym** permits to investigate experimentally the fine structure of  $\mathcal{I}$ , and hence the game semantics of the  $\lambda^!$ - and  $\lambda^{EAL}$ -calculi. For instance, we can easily verify that the model  $\mathcal{I}$  is a  $\lambda^!$ -algebra in the case of *strictly linear*  $\lambda$ -terms, by checking all the necessary equations, and find counterexamples in the general case.

We make this tool available for readers interested to *play* with games (-semantics). The paper builds on earlier work by the authors, the type system being an improvement.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Linear logic

**Keywords and phrases** game semantics, lambda calculus, involutions, linear logic, implicit computational complexity

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.7

**Supplementary Material** *Software (Tool)*: <http://158.110.146.197:31780/automata/>

**Acknowledgements** The authors would like to thank the anonymous referees for their helpful comments.

## 1 Introduction

We present the web portal  **$\Lambda$ -symsym**, available at <http://158.110.146.197:31780/automata/>, for experimenting, in the spirit of [24], with the *game semantics* of the  $\lambda^!$ -calculus, and its *normalizing elementary affine sub-calculus*, the  $\lambda^{EAL}$ -calculus. The  $\lambda^!$ -calculus is a generalization of the  $\lambda$ -calculus obtained by introducing a co-monadic modal operator  $!$ , which gives rise to a *pattern*  $\beta$ -reduction. Its sub-calculus corresponds to an applicatively closed class of terms normalizing in an elementary number of steps. All *elementary functions*



© Furio Honsell, Marina Lenisa, and Ivan Scagnetto;  
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 7; pp. 7:1–7:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

can be encoded in the  $\lambda^{EAL}$ -calculus, cf [20, 6]. The game model that we consider consists of *partial involutions*, i.e. *partial symmetric functions*. These were introduced by Abramsky in [3] to provide a *Geometry of Interaction* model,  $\mathcal{I}$ , of reversible computation for *linear combinatory logic*, see also [4, 5].

In a series of papers by the authors, [12, 10, 11, 18], we have explored and extended Abramsky’s model  $\mathcal{I}$ . In particular, we have introduced:

- a generalized co-monadic  $\lambda$ -calculus, the  $\lambda^!$ -calculus, which subsumes also the *elementary affine* and *linear affine* calculi [26, 7, 6, 15, 16];
- *abstraction algorithms* for compiling the above calculi in corresponding fragments of an extended linear combinatory logic,  $\mathbf{CL}^!$  or the normalizing  $\mathbf{CL}^{EAL}$ ;
- a semantics of these combinatory logics in terms of partial involutions, and hence a semantics of the  $\lambda^!$ -calculus via the abstraction algorithm;
- a *multi-modal intersection type systems*  $\vdash_!$  for the  $\lambda^!$ -calculus;
- an algorithm  $\mathcal{T}2\mathcal{I}$  for obtaining a specification of a partial involution, given a type in the language of  $\vdash_!$ , and an algorithm  $\mathcal{I}2\mathcal{T}$  for synthesizing a type given a specification of a partial involution, which are “morally” *mutual inverses*.

Moreover, we have shown that *linear application* between involutions, i.e. the game analogue of Girard’s “Execution Formula”, amounts to a kind of *resolution* between types in  $\vdash_!$ . In particular, in the case of strictly linear terms  $M, N$ , we have that  $\mathcal{I}2\mathcal{T}(\llbracket MN \rrbracket^{\mathcal{I}}) = \mathcal{I}2\mathcal{T}(\llbracket M \rrbracket^{\mathcal{I}} \cdot \llbracket N \rrbracket^{\mathcal{I}}) = \mathcal{R}es(\mathcal{I}2\mathcal{T}(\llbracket M \rrbracket^{\mathcal{I}}), \mathcal{I}2\mathcal{T}(\llbracket N \rrbracket^{\mathcal{I}}))$ , [12]. In this perspective, types deriving from  $\mathcal{I}2\mathcal{T}$  can be viewed as a kind of *principal types* for terms in the  $\lambda^!$ -calculus.

In this paper, we outline the implementation of the algorithms above and present the on-line tool  **$\Lambda$ -symsym**<sup>1</sup> for experimenting with them in a user-friendly interactive setting, thus allowing for the investigation of the fine structure and the peculiarities of the symmetric game semantics of  $\lambda^!$ -calculus. The sheer combinatorial complexity of applying linear application by hand, in effect, makes it essential to use a machine for checking equalities and finding counterexamples.

**$\Lambda$ -symsym** easily provides evidence for the following facts.

- The model  $\mathcal{I}$  of partial involutions does not satisfy any form of the  $\xi$ -rule, if the terms involved are not *strictly linear*. Namely, we can check all the necessary equations for  $\mathcal{I}$  to be a  $\lambda^!$ -algebra in the *strictly linear* case and provide counterexamples otherwise.
- Partial involutions keep track of the history of the execution, such as the erasures of arguments or the comonad !.

In particular, given a normalizing  $\lambda^!$ -term, or a  $\lambda^{EAL}$ -term,  $M$ , the web tool provides a web application for compiling  $M$  into a term of *linear combinatory logic*,  $\mathcal{A}^!(M)$  or  $\mathcal{A}^{EAL}(M)$ , respectively. Another web application provides the interpretation of  $\mathcal{A}^!(M)$  as a specification,  $\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}}$ , of a *partial involution* over a very simple language of moves, in the model  $\mathcal{I}$  of [3]. Yet another application permits to synthesize from  $\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}}$  a type,  $\mathcal{I}2\mathcal{T}(\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}})$ , in a *multimodal, intersection type discipline*. The type assignment system is given in the form of a general type system  $\vdash_!$ , for which we conjecture that  $\vdash_! M : \mathcal{I}2\mathcal{T}(\llbracket \mathcal{A}^!(M) \rrbracket^{\mathcal{I}})$ . In this paper we do not discuss the other web applications made available by the tool  **$\Lambda$ -symsym** for dealing with the other  $\lambda^!$ -calculi of [18], and non normalizing terms.

We wrote this paper to encourage interested readers to enjoy utilizing  **$\Lambda$ -symsym**. We made it available for people to play with games (-semantics)!

---

<sup>1</sup> We combine  $\Lambda$  to the magic word of the formula *Open Sesame*, **افتح يا سمسم** (*iftah ya symsym*), in the Arab collection of novels *1001 Nights*, **ألف ليلة وليلة** (*kitab alf laylah wa-laylah*), which reminds that our semantics are symmetric partial functions.

**Synopsis.** In Section 2, we introduce in a self-contained format the formal systems used in the web tool, namely the  $\lambda^!$ -calculus, and the  $\lambda^{EAL}$ -calculus, the linear combinatory logics  $\mathbf{CL}^!$  and  $\mathbf{CL}^{EAL}$ ,  $!$ -intersection-types, and the  $\vdash_!$  type discipline. In Subsection 2.1 we give the corresponding abstraction algorithms. In Section 3, we present the game theoretic semantics of the linear combinatory logic  $\mathbf{CL}^!$ , and hence of its normalizing sublogic  $\mathbf{CL}^{EAL}$ , in terms of partial involutions. We introduce also the crucial concept of specification of an involution. In Subsection 3.1, we present the two algorithms  $\mathcal{I}2\mathcal{T}$  and  $\mathcal{T}2\mathcal{I}$ , for mapping specifications of partial involutions into types and back. In Section 4, we illustrate the web tool  $\mathbf{\Lambda}$ -**symsym** and present its web interface. In Subsection 4.2 we discuss remarkable example sessions. Finally, in Section 5, we draw some conclusions and outline future directions. In the Appendix A, we give the Erlang code corresponding to the implementation of linear application of involutions. The complete code of our web tool is available in [13].

## 2 The $\lambda^!$ -calculus, Linear Combinatory Logic, $!$ -intersection-types

In this section we recall the formal systems used in the web tool. Most of these appear already in earlier papers by the authors, see [12, 10, 11, 18], but for the definition of the type assignment system  $\vdash_!$ .

We start with the  $\lambda^!$ -calculus and its sub-calculus.

► **Definition 1** ( $\lambda^!$ -calculus,  $\lambda^{EAL}$ -calculus).

- The language  $\mathbf{\Lambda}^!$  of the  $\lambda^!$ -calculus is inductively defined from variables  $x, y, z, \dots$  and constants  $c, \dots$ , and it is closed under the following formation rules:

$$\frac{M \in \mathbf{\Lambda}^! \quad N \in \mathbf{\Lambda}^!}{MN \in \mathbf{\Lambda}^!} \text{ (app)} \qquad \frac{M \in \mathbf{\Lambda}^!}{!M \in \mathbf{\Lambda}^!} \text{ (!)}$$

$$\frac{M \in \mathbf{\Lambda}^! \quad \mathcal{O}_{\leq 1}^!(x, M)}{\lambda x.M \in \mathbf{\Lambda}^!} \text{ (\lambda)} \qquad \frac{M \in \mathbf{\Lambda}^! \quad \mathcal{O}_{\geq 1}(x, M)}{\lambda!x.M \in \mathbf{\Lambda}^!} \text{ (\lambda!)}$$

where  $FV(M)$  denotes the set of free variables of  $M$ , and  $\mathcal{O}_{?}^{\#}(x, M)$  ( $\mathcal{O}_{?}^{\#}(M)$ ), for  $\#$  denoting either  $!$  or blank, and  $?$  denoting  $\leq 1$  or  $\geq 1$  or no constraint, means that the variable  $x$  (the free variables of  $M$ ) appears at most once ( $\leq 1$ ) or at least once ( $\geq 1$ ) in  $M$ , respectively, and, when  $\#$  is not blank, that it cannot appear in the scope of  $!$ .

We denote by  $\equiv$  syntactical identity on  $\lambda$ -terms.

- The language  $\mathbf{\Lambda}^{EAL}$  of the  $\lambda^{EAL}$ -calculus is the restriction of  $\mathbf{\Lambda}^!$  obtained by considering rules (app), ( $\lambda$ ), and the following versions of the rules ( $\lambda!$ ) and (!):

$$\frac{M \in \mathbf{\Lambda}^{EAL} \quad \mathcal{O}_{\geq 1}(x, M) \quad x \text{ is in the scope of a single } !}{\lambda!x.M \in \mathbf{\Lambda}^{EAL}} \qquad \frac{M \in \mathbf{\Lambda}^{EAL} \quad \mathcal{O}^!(M)}{!M \in \mathbf{\Lambda}^{EAL}}$$

- The reduction rules of the  $\lambda^!$ -calculus include the restrictions of the standard ( $\beta$ )-rule and ( $\xi$ )-rule to linear abstractions, the pattern- $\beta$ -reduction rule, which define the behaviour of the  $!$  pattern abstraction operator, the corresponding ( $str!$ ) structural rule, and the ( $\xi!$ ) rule, namely:

$$\begin{aligned} (\beta) \quad (\lambda x.M)N &\rightarrow_! M[N/x] & (\beta!) \quad (\lambda!x.M)N &\rightarrow_! M[N/x] \\ (\xi) \quad \frac{M \rightarrow_! N \quad \lambda x.M \in \mathbf{\Lambda}^!}{\lambda x.M \rightarrow_! \lambda x.N} & (\xi!) \quad \frac{M \rightarrow_! N \quad \lambda!x.M \in \mathbf{\Lambda}^!}{\lambda!x.M \rightarrow_! \lambda!x.N} & (str!) \quad \frac{M \rightarrow_! N}{!M \rightarrow_! !N} \\ (appl_L) \quad \frac{M_1 \rightarrow_! M'_1}{M_1 M_2 \rightarrow_! M'_1 M_2} & (appl_R) \quad \frac{M_2 \rightarrow_! M'_2}{M_1 M_2 \rightarrow_! M_1 M'_2} \end{aligned}$$

We denote by  $\rightarrow_!^*$  the reflexive and transitive closure of  $\rightarrow_!$ .

The reduction rules of the  $\lambda^{EAL}$ -calculus are the restrictions of the rules of the  $\lambda^!$ -calculus to  $\mathbf{\Lambda}^{EAL}$ -terms. We denote by  $\rightarrow_{EAL}$  the corresponding reduction relation.

We define strong normal forms as the irreducible terms which do not contain *stuck redexes*, i.e. subterms of the shape  $(!M)N$  or  $(\lambda!x.M)P$ , where  $P$  is irreducible and not a  $!$ -term.

► **Definition 2** (Strong Normal Forms). *The set  $SN^!$  of strong normal forms is inductively defined as follows:*

$$\frac{M_i \in SN^! \quad \forall i \in \{1, \dots, n\}}{xM_1 \dots M_n \in SN^!} \qquad \frac{M \in SN^!}{!M \in SN^!}$$

$$\frac{M \in SN^! \quad \lambda x.M \in \Lambda^!}{\lambda x.M \in SN^!} \qquad \frac{M \in SN^! \quad \lambda!x.M \in \Lambda^!}{\lambda!x.M \in SN^!}$$

Of course many terms containing stuck redexes are typable in the typing system of Definition 6 below, and receive the appropriate involution semantics, e.g.  $\lambda x.(\lambda!y.y)x$  behaves as  $\lambda!x.x$ , but these intricacies are not worthwhile considering for the purposes of this paper.

The involution semantics of terms in  $\Lambda^!$  and in  $\Lambda^{EAL}$  will be given via a compilation in linear combinatory logic.

► **Definition 3** (Linear Combinatory Logics).

■ *The language of the extended linear combinatory logic  $CL^!$  consists of variables  $x, y, \dots$ , distinguished constants (combinators)  $B, C, I, K, W, D, \delta, F$ , and it is closed under application and  $!$ -promotion, i.e.:*

$$\frac{M \in CL^! \quad N \in CL^!}{MN \in CL^!} \qquad \frac{M \in CL^!}{!M \in CL^!}$$

*In  $CL^!$ -terms, we associate  $\cdot$  to the left and we assume  $!$  to have order of precedence greater than  $\cdot$ . Combinators reduce according to the following pattern reduction rules:*

$$\begin{array}{llll} BMNP \rightarrow_! M(NP) & IM \rightarrow_! M & CMNP \rightarrow_! (MP)N & KMN \rightarrow_! M \\ WM!N \rightarrow_! M!N!N & \delta!M \rightarrow_! !M & D!M \rightarrow_! M & F!M!N \rightarrow_! !(MN) \\ \frac{M \rightarrow_! M'}{C[M] \rightarrow_! C[M']} & \text{for any context } C[\ ] & & \end{array}$$

*where one-hole contexts are defined as usual.*

■ *The language of elementary affine combinatory logic  $CL^{EAL}$  includes only the combinators  $B, I, C, K, W, F$ , and is closed under the rules of application and  $!$ -promotion. The reduction relation restricted to  $CL^{EAL}$  is denoted by  $\rightarrow_{EAL}$ .*

*We use the notation **Comb** to denote any of the combinators in the above definition.*

We recall the following proposition because it is not so immediate:

► **Proposition 4** (Normalization [18]). *The  $\lambda^{EAL}$ -calculus and  $CL^{EAL}$  linear combinatory logic are strongly normalizing.*

The type discipline introduced in this section is a refinement of the ones introduced in [10, 12, 11] and originated in the process of relating *partial involutions* and *principal types*. It amounts essentially to a *multimodal intersection type discipline*, which generalizes [8].

► **Definition 5** (Type Language). *The language of types,  $Type^!$ , is a two sorted language given by the following grammars, where  $\omega$  is a constant, and  $\alpha, \beta, \dots$  and  $i, j, \dots$  are type and index variables respectively:*

$$(Type^! \ni) \sigma, \tau ::= \omega \mid \alpha \mid \dots \mid \sigma \multimap \tau \mid \widehat{\sigma}$$

$$(\widehat{Type^!} \ni) \widehat{\sigma}, \widehat{\tau} ::= !_u \sigma \mid \widehat{\sigma} \wedge \widehat{\tau}$$

$$u, v ::= \epsilon \mid i \mid j \mid \dots \mid \langle u, v \rangle.$$

*The syntactic category  $\widehat{\sigma}$ -types isolates types whose main constructor is  $!$  or  $\wedge$ . Moreover, we consider the equivalence relation on types  $\sim$  induced by  $\omega \sim \sigma$ , for any type  $\sigma$  which contains only the constant  $\omega$  and no type variables.*



► **Definition 6** (Type System).

The type system  $\vdash_!$  derives judgements  $\Gamma \vdash_! M : \tau$ , where the environment  $\Gamma$  is a set of pairs of the following shapes:  $x : \sigma$ ,  $!x : \hat{\sigma}$ ; each variable occurs at most once in  $\Gamma$  either as a linear variable or a  $!$ -variable.  $\text{dom}(\Gamma)$  denotes the set of variables which appear in  $\Gamma$ .

**Assumption rule:**

$$\frac{}{x : \alpha \vdash_! x : \alpha} \text{ (Var)}$$

**Introduction and elimination rules:**

$$\frac{\Gamma \vdash_! M : \sigma \quad i \text{ fresh}}{\widehat{!}_i(\Gamma) \vdash_! M : !_i \sigma} \text{ (!Intro)} \qquad \frac{\Gamma \vdash_! !M : !_i \sigma \quad !(\Gamma)}{\widehat{!}_{[\epsilon/i]}(\Gamma) \vdash_! M : \sigma[\epsilon/i]} \text{ (!-Elim)}$$

$$\frac{\Gamma_1 \vdash_! !M : \hat{\sigma} \quad \Gamma_2 \vdash_! !M : \hat{\tau} \quad !(\Gamma_1) \quad !(\Gamma_2)}{\Gamma_1 \wedge \Gamma_2 \vdash_! !M : \hat{\sigma} \wedge \hat{\tau}} \text{ (\wedge-Intro)}$$

$$\frac{\Gamma \vdash_! M : \hat{\sigma} \wedge \hat{\tau}}{\Gamma \vdash_! M : \hat{\sigma}} \text{ (\wedge-E-left)} \qquad \frac{\Gamma \vdash_! M : \hat{\sigma} \wedge \hat{\tau}}{\Gamma \vdash_! M : \hat{\tau}} \text{ (\wedge-E-right)}$$

$$\frac{\Gamma_1 \vdash_! M : \sigma \multimap \tau \quad \Gamma_2 \vdash_! N : \sigma \quad \bigcap_!(\Gamma_1, \Gamma_2)}{\Gamma_1 \wedge \Gamma_2 \vdash_! MN : \tau} \text{ (MP)}$$

$$\frac{\Gamma, x : \sigma \vdash_! M : \tau}{\Gamma \vdash_! \lambda x. M : \sigma \multimap \tau} \text{ (\lambda)} \qquad \frac{\Gamma, !x : \hat{\sigma} \vdash_! M : \tau}{\Gamma \vdash_! \lambda !x. M : \hat{\sigma} \multimap \tau} \text{ (\lambda!)}$$

$$\frac{\Gamma \vdash_! M : \sigma}{U(\Gamma) \vdash_! M : U(\sigma)} \text{ (Inst)}$$

**Structural-rules:**

$$\frac{\Gamma \vdash_! M : \sigma \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : \tau \vdash_! M : \sigma} \text{ (weak)} \qquad \frac{\sigma \sim \omega}{\Gamma \vdash_! M : \sigma} \text{ (\omega)}$$

where

- $!(\Gamma)$  means that all variables in  $\Gamma$  are banged ( $\Gamma$  is possibly empty);
- by a slight abuse of notation

$$\widehat{!}_u(\Gamma, x : \tau) = \widehat{!}_u(\Gamma), !x : \widehat{!}_u(\tau) \quad \text{and} \quad \begin{cases} \widehat{!}_u(\tau) = !_u \tau & \text{if } \tau \notin \widehat{\text{Type}} \\ \widehat{!}_u(!_v \tau) = !_v \tau & \\ \widehat{!}_u(\hat{\tau} \wedge \hat{\sigma}) = \widehat{!}_u(\hat{\tau}) \wedge \widehat{!}_u(\hat{\sigma}); & \end{cases}$$

- $\widehat{!}_{[\epsilon/i]}(\Gamma, !x : \tau) = \widehat{!}_{[\epsilon/i]}(\Gamma), !x : \tau[\epsilon/i]$ ;
- $\bigcap_!(\Gamma_1, \Gamma_2)$  means that each variable in the intersection of  $\text{dom}(\Gamma_1)$  and  $\text{dom}(\Gamma_2)$  is  $!$ -prefixed both in  $\Gamma_1$  and  $\Gamma_2$ ;
- for  $\Gamma_1, \Gamma_2$  such that  $\bigcap_!(\Gamma_1, \Gamma_2)$ , the environment  $\Gamma_1 \wedge \Gamma_2$  is defined by:
  - if  $x : \sigma \in \Gamma_1$  or  $x : \sigma \in \Gamma_2$ , then  $x : \sigma \in \Gamma_1 \wedge \Gamma_2$
  - if  $!x : \sigma \in \Gamma_1$  and  $x \notin \text{dom}(\Gamma_2)$ , or  $!x : \sigma \in \Gamma_2$  and  $x \notin \text{dom}(\Gamma_1)$ , then  $!x : \sigma \in \Gamma_1 \wedge \Gamma_2$
  - if  $!x : \sigma_1 \in \Gamma_1$  and  $!x : \sigma_2 \in \Gamma_2$ , then  $!x : \sigma_1 \wedge \sigma_2 \in \Gamma_1 \wedge \Gamma_2$ ;
- $U$  is a substitution, mapping type/index variables to types/indexes, respectively.

We will denote by  $\text{Type}^{!-}$  and  $\vdash_!^-$  the set of types without  $\omega$  and the type system without the  $\omega$ -rule, respectively.

The type system  $\vdash_{EAL}$  is obtained by giving up the rule (!-Elim) and restricting the rules of  $\vdash_!$  to  $\lambda$ -terms in  $\mathbf{\Lambda}^{EAL}$ .

Some comments and remarks on the above definitions are in order:

- The type connective  $\wedge$  is neither commutative nor idempotent nor associative.
- The notion of *type equivalence*  $\sim$  defined above for the system of Definition 6 is a sharp restriction of the usual type equivalence on intersection types, *cf.* [8]. The traditional rule does not hold in the involution model of Section 3.
- Not all terms of  $\Lambda^!$  can be typed in  $\vdash_1^-$ , *i.e.* without the  $\omega$ -rule, *e.g.*  
 $\vdash_1 (\lambda!x.x!x)(\lambda!x.x!x) : \omega \quad \vdash_1 \lambda!x!y.(\lambda!z.z)(xy) : (\alpha \multimap \omega) \rightarrow (\alpha \multimap \omega)$ .  
 The former is not normalizing, while in the latter pattern matching fails and hence it does not have a strong normal form. Notice however that the type which can be assigned to it records the fact that the two variables can be applied. The meaningfulness of such types is the reason for weakening the traditional definition of type equivalence  $\sim$  as in the system of Definition 6. The issue is rather intricate and will be taken up in Section 4.2.2.
- Rule (Inst) could be made redundant by allowing arbitrary types in rule (Var) and arbitrary indices in rule (!Intro).
- One could consider also second or higher order (possibly recursive) type systems. Their typing strength is subsumed by the  $\wedge$ -rules.

A relevant class of types is that of *binary types*, which, as we will see in Section 3.1, arise naturally from specifications of partial involutions:

► **Definition 7.**

- A type  $\tau \in \text{Type}^!$  is *binary* if each type variable appears at most twice in  $\tau$ .
- A judgement  $\Gamma \vdash_1 M : \tau$  is *binary* if each type variable occurs at most twice in it.

A slight modification of Theorem 30 [11] allows to prove that:

- **Proposition 8.** *If  $\Gamma \vdash_1 M : \tau$  is derivable, then there exist a binary judgement  $\Gamma' \vdash_1 M : \tau'$  and a substitution  $U$  such that  $U(\Gamma') = \Gamma$  and  $U(\tau') = \tau$ .*

## 2.1 Abstraction Algorithm

The abstraction algorithm for mapping terms of the  $\lambda^!$ -calculus into the linear combinatory algebra  $\mathbf{CL}^!$  is a refinement of the standard abstraction algorithm, but this needs to be carefully spelled out so as to factor out as an abstraction algorithm also from  $\lambda^{EAL}$  to  $\mathbf{CL}^{EAL}$ .

► **Definition 9** (Abstraction Algorithms  $\mathcal{A}^!$ ,  $\mathcal{A}^{EAL}$ ).

- Let  $\Lambda_{\mathbf{CL}}^!$  denote the set of terms of the  $\lambda^!$ -calculus obtained by taking combinators as the constants of the calculus. We define  $\mathcal{A}^! : \Lambda_{\mathbf{CL}}^! \rightarrow \mathbf{CL}^!$  as in Figure 1, where  $\#$  denotes the modality ! or no modality.
- Let  $\Lambda_{\mathbf{CL}}^{EAL}$  be the set of  $\lambda^{EAL}$ -terms obtained by taking  $\mathbf{CL}^{EAL}$ -combinators as constants. We define  $\mathcal{A}^{EAL} : \Lambda_{\mathbf{CL}}^{EAL} \rightarrow \mathbf{CL}^{EAL}$  as the restriction of  $\mathcal{A}^!$  to terms in  $\Lambda_{\mathbf{CL}}^{EAL}$ .
- Vice versa, we denote by  $(\ )_{\lambda} : \mathbf{CL}^! \rightarrow \Lambda^!$  the natural mapping of a  $\mathbf{CL}$ -term into a  $\lambda$ -term obtained by replacing, in place of each combinator, the corresponding  $\Lambda^!$ -term as follows:

$$\begin{array}{llll} (\mathbf{B})_{\lambda} = \lambda xyz.x(yz) & (\mathbf{C})_{\lambda} = \lambda xyz.(xz)y & (\mathbf{I})_{\lambda} = \lambda x.x & (\mathbf{K})_{\lambda} = \lambda xy.x \\ (\mathbf{W})_{\lambda} = \lambda x!y.x!y!y & (\mathbf{D})_{\lambda} = \lambda!x.x & (\delta)_{\lambda} = \lambda!x!!x & (\mathbf{F})_{\lambda} = \lambda!x!y.!(xy). \end{array}$$

The following theorem justifies the above definition. For ease of readability we will often use the notation  $\lambda^* \#x_1 \dots \#x_n.M$ , for  $M \in \mathbf{CL}^!$ , to denote  $\mathcal{A}^!(\lambda \#x_1 \dots \#x_n.M)$ , where  $\#$  stands for either the modality ! or no modality (see [18] for more details).



$$\begin{array}{l}
\mathcal{A}^!(M) \\
= \left\{ \begin{array}{ll}
x & \text{if } M \equiv x \\
\mathbf{Comb} & \text{if } M \equiv \mathbf{Comb} \\
\mathbf{I} & \text{if } M \equiv \lambda x.x \\
\mathbf{F}(!I) & \text{if } M \equiv \lambda!x.!x \\
\mathbf{D} & \text{if } M \equiv \lambda!x.x \\
\mathbf{K}c & \text{if } M \equiv \lambda x.c \\
\mathbf{K}y & \text{if } M \equiv \lambda x.y \\
\mathbf{C}(\mathcal{A}^!(\lambda x.M_1))\mathcal{A}^!(M_2) & \text{if } M \equiv \lambda x.M_1M_2 \text{ and } x \in FV(M_1) \\
\mathbf{B}\mathcal{A}^!(M_1)\mathcal{A}^!(\lambda x.M_2) & \text{if } M \equiv \lambda x.M_1M_2 \text{ and } x \in FV(M_2) \\
\mathbf{K}(\mathcal{A}^!(M_1)\mathcal{A}^!(M_2)) & \text{if } M \equiv \lambda x.M_1M_2 \text{ and } x \notin FV(M_1M_2) \\
\mathcal{A}^!(\lambda\#x.\mathcal{A}^!(\lambda\#y.M_1)) & \text{if } M \equiv \lambda\#x.\lambda\#y.M_1 \\
\mathbf{W}(\mathbf{B}(\mathbf{C}\mathcal{A}^!(\lambda!x.M_1))(\mathcal{A}^!(\lambda!x.M_2))) & \text{if } M \equiv \lambda!x.M_1M_2 \text{ and } x \in FV(M_1), x \in FV(M_2) \\
\mathbf{B}\mathcal{A}^!(M_1)\mathcal{A}^!(\lambda!x.M_2) & \text{if } M \equiv \lambda!x.M_1M_2 \text{ and } x \notin FV(M_1), x \in FV(M_2) \\
\mathbf{C}\mathcal{A}^!(\lambda!x.M_1)\mathcal{A}^!(M_2) & \text{if } M \equiv \lambda!x.M_1M_2 \text{ and } x \in FV(M_1), x \notin FV(M_2) \\
\mathbf{F}!(\lambda x.\mathcal{A}^!(M_1)) & \text{if } M \equiv \lambda!x.!M_1 \text{ and } x \text{ occurs once in } M_1 \text{ and } \mathcal{O}^!(M_1) \\
\mathbf{W}\mathcal{A}^!(\lambda!y.\lambda!x.!M_1[y/x^1]) & \text{if } M \equiv \lambda!x.!M_1[x^1], x \text{ occurs in } M_1 \text{ more than once} \\
& \text{and } x^1 \text{ is the leftmost occurrence of } x \text{ in } M_1 \text{ and } \mathcal{O}^!(M_1) \\
\mathbf{B}(\mathbf{F}!(\mathcal{A}^!(\lambda!x.M_1)))\delta & \text{if } M \equiv \lambda!x.!M_1 \text{ and } \neg\mathcal{O}^!(M_1) \\
\mathcal{A}^!(M_1)\mathcal{A}^!(M_2) & \text{if } M = M_1M_2 \\
! \mathcal{A}^!(M_1) & \text{if } M = !M_1 \\
\mathbf{FAIL} & \text{otherwise}
\end{array} \right.
\end{array}$$

■ **Figure 1** The Abstraction Algorithm  $\mathcal{A}^! : \Lambda_{\mathbf{CL}}^! \rightarrow \mathbf{CL}^!$ .

► **Theorem 10** (Soundness of the Abstraction Algorithms  $\mathcal{A}^!$  and  $\mathcal{A}^{EAL}$ ).

- Let  $M \in \mathbf{CL}^!$ , let  $\#M_1, \dots, \#M_n \in \Lambda^!$ ,  $n \geq 0$ , then
$$(\lambda^* \#x_1 \dots \#x_n.M)\mathcal{A}^!(\#M_1) \dots \mathcal{A}^!(\#M_n) \rightarrow_!^* M[\mathcal{A}^!(M_1)/x_1, \dots, \mathcal{A}^!(M_n)/x_n].$$
- Let  $M \in \mathbf{CL}^{EAL}$ , let  $\#M_1, \dots, \#M_n \in \Lambda^{EAL}$ ,  $n \geq 0$ , then  $\mathcal{A}^{EAL}(M) \in \Lambda^{EAL}$  and moreover:
$$(\lambda^* \#x_1 \dots \#x_n.M)\mathcal{A}^{EAL}(\#M_1) \dots \mathcal{A}^{EAL}(\#M_n) \rightarrow_!^* M[\mathcal{A}^{EAL}(M_1)/x_1, \dots, \mathcal{A}^{EAL}(M_n)/x_n].$$

The abstraction algorithm in Fig. 1 is implemented in  **$\Lambda$ -symsym**, see Section 4.

### 3 Game Semantics for Combinatory Logics and $\lambda$ -calculi

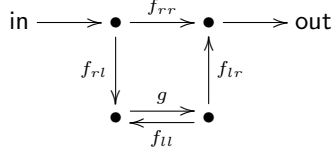
In this section, we present the *game theoretic* semantics of the linear combinatory logic  $\mathbf{CL}^!$ , and hence of its sublogic  $\mathbf{CL}^{EAL}$ , in terms of *partial involutions*. Then, via the abstraction algorithm, we obtain a game semantics for terms in  $\Lambda^!$  and  $\Lambda^{EAL}$ . Finally, we present two algorithms,  $\mathcal{I}2\mathcal{T}$  and  $\mathcal{T}2\mathcal{I}$ , which relate schematic representations of partial involutions and types.

We start by introducing the model of partial involutions,  $\mathcal{I}$ , originally defined by Abramsky in [3] in order to study reversible computations, and studied in [12, 11, 18]. The notion of application on this model amounts to the categorical trace on a subcategory of the category of relations, introduced in [23] (see also [2]).

► **Definition 11** (The Model of Partial Involutions,  $\mathcal{I}$ ).

- $T_\Sigma$ , the language of moves, is defined by the signature  $\Sigma_0 = \{\epsilon\}$ ,  $\Sigma_1 = \{l, r\}$ ,  $\Sigma_2 = \{\langle, \rangle\}$  (where  $\Sigma_i$  is the set of constructors of arity  $i$ ); terms  $r(t)$  are output words, while terms  $l(t)$  are input words (often denoted simply by  $rt$  and  $lt$ );

- $\mathcal{I}$  is the set of partial involutions over  $T_\Sigma$ , i.e. the set of all partial functions  $f : T_\Sigma \rightarrow T_\Sigma$  such that  $f(t) = t' \Leftrightarrow f(t') = t$ ;
- the operation of replication is defined by  $!f = \{(\langle t, t_1 \rangle, \langle t, t_2 \rangle) \mid t \in T_\Sigma \wedge (t_1, t_2) \in f\}$ ;
- the notion of linear application is defined by  $f \cdot g = f_{rr} \cup (f_{ri}; g; (f_{li}; g)^*; f_{lr})$ , where  $f_{ij} = \{(\langle t_1, t_2 \rangle \mid (i(t_1), j(t_2)) \in f\}$ , for  $i, j \in \{r, l\}$  (see diagram below), where “;” denotes postfix composition of relations.



- **Proposition 12** ([3]).  $\mathcal{I}$  is closed under  $!$ -replication and linear application.

Next we give the notion of *schematic specification* of the graph of a partial involution, which was introduced in [18] and recall some of its main properties. Schematic specifications induce a substructure of  $\mathcal{I}$  which is still closed under application and replication, and it is sufficient to interpret all combinators, see [18] for full details.

- **Definition 13** (Schematic Specification). Let  $T_\Sigma[Var]$  denote the set of terms generated by the signature  $\Sigma \setminus \{\epsilon\}$  starting from a set of variables  $Var$ , where we distinguish between term variables  $x, y, \dots$  and index variables  $i, j, \dots$ , i.e.:

$(T_\Sigma[Var] \ni) t ::= x \mid y \mid \dots \mid lt \mid rt \mid \langle u, t \rangle$  where  $u ::= \epsilon \mid i \mid j \mid \dots \mid lu \mid ru \mid \langle u, v \rangle$ . A specification  $P$  of a partial involution is a (possibly infinite) irreflexive set of pairs  $\{t_1 \leftrightarrow t'_1, t_2 \leftrightarrow t'_2, \dots\}$ , where:

- (i)  $t_i, t'_i \in T_\Sigma[Var]$ ;
- (ii) a single pair  $t_i \leftrightarrow t'_i$  represents both  $(t_i, t'_i)$  and  $(t'_i, t_i)$ ;
- (iii) in any pair  $t_i \leftrightarrow t'_i$  exactly one term variable appears, once in  $t_i$  and once in  $t'_i$ ;
- (iv) for any pairs  $t_i \leftrightarrow t'_i, t_j \leftrightarrow t'_j$ ,  $t_i$  is not an instance of  $t_j$ .

Schematic specifications generate graphs of partial involutions as follows:

- **Definition 14** ( $\mathcal{G}$ ). Given a schematic specification  $P$ , the set  $\mathcal{G}(P)$  is the symmetric closure of the set  $\{(U(t_1), U(t_2)) \mid (t_1 \leftrightarrow t_2) \in P \wedge U : T_\Sigma[Var] \rightarrow T_\Sigma\}$ .

The following lemma follows immediately from the above definitions:

- **Lemma 15.** If  $P$  is a specification, then  $\mathcal{G}(P)$  is the graph of a partial involution on  $T_\Sigma$ .

Not all partial involutions are generated by specifications, e.g. the relation  $\{r\epsilon \leftrightarrow l\epsilon\}$  is trivially not induced by any specification.

A notion of composition on specifications, which corresponds to composition on partial involutions can be defined as follows:

- **Definition 16** (Composition of Specifications). Let  $P, P'$  be specifications. We define

$$P; P' = \{(t, t') \mid \exists (t_1, t_2) \in P, (t'_1, t'_2) \in P', U. (U = \mathcal{U}(t_2, t'_1) \wedge t = U(t_1) \wedge t' = U(t'_2))\},$$

where  $\mathcal{U}(t_2, t'_1)$  is the most general unifier of  $t_2$  and  $t'_1$ .

This definition essentially amounts to composition in Clause Algebras, as in [19].

The above definitions permits to define a notion of application between specifications,  $P \cdot P'$ , which mimics the one between partial involutions in Definition 11.

$$\begin{aligned}
\llbracket \mathbf{B} \rrbracket^{\mathcal{I}} &= \{r^3x \leftrightarrow lrx, l^2x \leftrightarrow rlr, rl^2x \leftrightarrow r^2lx\} \\
\llbracket \mathbf{I} \rrbracket^{\mathcal{I}} &= \{lx \leftrightarrow rx\} \\
\llbracket \mathbf{C} \rrbracket^{\mathcal{I}} &= \{l^2x \leftrightarrow r^2lx, lrlx \leftrightarrow rlx, lr^2x \leftrightarrow r^3x\} \\
\llbracket \mathbf{K} \rrbracket^{\mathcal{I}} &= \{lx \leftrightarrow r^2x\} \\
\llbracket \mathbf{F} \rrbracket^{\mathcal{I}} &= \{l\langle i, rx \rangle \leftrightarrow r^2\langle i, x \rangle, l\langle i, lx \rangle \leftrightarrow rl\langle i, x \rangle\} \\
\llbracket \mathbf{W} \rrbracket^{\mathcal{I}} &= \{r^2x \leftrightarrow lr^2x, l^2\langle i, x \rangle \leftrightarrow rl\langle li, x \rangle, lrl\langle i, x \rangle \leftrightarrow rl\langle ri, x \rangle\} \\
\llbracket \delta \rrbracket^{\mathcal{I}} &= \{l\langle \langle i, j \rangle, x \rangle \leftrightarrow r\langle i, \langle j, x \rangle \rangle\} \\
\llbracket \mathbf{D} \rrbracket^{\mathcal{I}} &= \{l\langle \epsilon, x \rangle \leftrightarrow rx\} \\
\llbracket MN \rrbracket^{\mathcal{I}} &= \llbracket M \rrbracket^{\mathcal{I}} \cdot \llbracket N \rrbracket^{\mathcal{I}} \\
\llbracket !M \rrbracket^{\mathcal{I}} &= !\llbracket M \rrbracket^{\mathcal{I}}
\end{aligned}$$

■ **Figure 2** Game Semantics for  $\mathbf{CL}^!$ .

► **Lemma 17** ([18]). *Let  $P, P'$  be specifications. Then*

- (i)  $\mathcal{G}(P; P') = \mathcal{G}(P); \mathcal{G}(P')$ ;
- (ii)  $\mathcal{G}(P \cdot P') = \mathcal{G}(P) \cdot \mathcal{G}(P')$ .

One can easily check that:

► **Lemma 18** ([18]). *The substructure of  $\mathcal{I}$  induced by schematic specifications is closed under application and replication.*

Now we are in the position of giving the interpretation in  $\mathcal{I}$  of  $\mathbf{CL}^!$ , by defining the schematic specifications of the graphs of the combinators.

► **Definition 19** (Game Semantics of  $\mathbf{CL}^!$ ). *The combinatory logic  $\mathbf{CL}^!$  is interpreted in  $\mathcal{I}$  as in Figure 2, where we use the abbreviation  $l^n t$  and  $r^n t$  for the terms  $\underbrace{l \dots l}_n t$ , and  $\underbrace{r \dots r}_n t$ .*

In following proposition for (i) see [3], for (ii) see [11].

► **Proposition 20.**

- (i)  $\mathcal{I}$  is a model of  $\mathbf{CL}^!$ , and hence of the combinatory sublogic  $\mathbf{CL}^{\mathbf{EAL}}$ .
- (ii)  $\mathcal{I}$  is a model of all the reduction rules of  $\mathbf{\Lambda}^!$ , but the  $\xi$ -rules.

Using the semantics for  $\mathbf{CL}^!$ -combinators defined in Figure 2, and the abstraction algorithm described in Section 2.1, one can derive the partial involution corresponding to a given  $\lambda$ -term, by repeatedly applying our implementation of linear application on involutions (see Appendix A). However, not all interpretations of  $\mathbf{CL}^!$ -combinators are partial involutions with *finite* specifications, since graphs of finite specifications are not closed under linear application. *E.g.* the fixed point combinator  $(\lambda^*!x!y.y!(x!y!y!x))!(\lambda^*!x!y.y!(x!y!y!x))$  has an infinite specification. This is taken care of in  $\mathbf{\Lambda}$ -**symsym** by outputting the stream of pairs in the partial specification.

### 3.1 Correspondence Algorithms, $\mathcal{I}2\mathcal{T}$ , $\mathcal{T}2\mathcal{I}$

Here we present the two algorithms,  $\mathcal{I}2\mathcal{T}$  and  $\mathcal{T}2\mathcal{I}$ , which relate schematic representations of partial involutions and types in  $Type^!$ . The first algorithm, given a schematic specification of a partial involution, yields a binary type, the latter, given a binary type, provides a schematic specification of a partial involution. The two transformations are one inverse of the another over suitable domains of schematic specifications and types.

We begin by explaining how the algorithm  $\mathcal{I}2\mathcal{T}$  works. First of all, we need the following  $build\_type\_tree()$  function, in order to build a *skeleton* type tree from a term  $t \in T_{\Sigma}[Var]$ . The function definition is syntax driven:

$$\begin{aligned}
& build\_type\_tree(t) \\
& = \begin{cases} \alpha_x & \text{if } t \equiv x \\ build\_type\_tree(t') \rightarrow \_ & \text{if } t \equiv lt' \\ \_ \rightarrow build\_type\_tree(t') & \text{if } t \equiv t' \\ !_u build\_type\_tree(t') & \text{if } \tau \equiv \langle u, t' \rangle \\ & \text{and } u \text{ does not contain } l/r \text{ constructors} \\ build\_type\_tree(\langle u', t' \rangle) \wedge \_ & \text{if } t \equiv \langle u, t' \rangle \text{ and } l \text{ is the leftmost } l/r \text{ constructor} \\ & \text{in } u \text{ and } u' \text{ is obtained from } u \text{ by erasing it} \\ \_ \wedge build\_type\_tree(\langle u', t' \rangle) & \text{if } t \equiv \langle u, t' \rangle \text{ and } r \text{ is the leftmost } l/r \text{ constructor} \\ & \text{in } u \text{ and } u' \text{ is obtained from } u \text{ by erasing it} \end{cases}
\end{aligned}$$

where the underscore characters ( $\_$ ) above stand for “missing parts” of the resulting type tree. Such holes will be (hopefully) filled in by successive unifications, as explained below.

Hence, for each rule  $t_i \leftrightarrow t'_i$  (for  $i = 1, \dots, n$ ) of a given specification, we compute the two (partial) types  $build\_type\_tree(t_i)$  and  $build\_type\_tree(t'_i)$ . Finally, we try to unify all such partial types, over all rules. If we succeed, we infer a type representing the original partial specification; otherwise, we fail. Notice that the possible final type may still contain “holes”. *E.g.* partial specifications of erasing terms do not exhibit any rule for the variable which is erased.

Due to the peculiar syntax of types in  $Type^!$ , unification of types is not a plain unification *à la* Robinson. Indeed, we have subcases which may need to spawn  $\wedge$ -constructors in order to instantiate new instances of some types. For instance, a  $!$ -subtype like  $!_u\sigma$  may need to spawn new instances of  $!_u\sigma$ , becoming  $!_{u'}\sigma' \wedge !_{u''}\sigma'' \wedge \dots$ , in order to unify against another type. In our implementation, we implement such spawnings in order to ease the unification process as much as possible. At the end there can be some leftovers of those spawning activities; thus, we implement a sort of “garbage collector” which takes care of those dummy  $\wedge$ ’s, before returning the final type-candidate to the user.

Another subtle issue comes from the fact that the order in which the pairs of the specification are processed may imply success or failure in the unification process (because the spawning of  $\wedge$ -constructors must trigger at the right moment). Hence, we restart the procedure until a type is found, permuting in all possible ways the pairs in the specification.

Finally, the type synthesis algorithm may fail, totally or partially. Total failure can be easily detected, since the algorithm returns  $\Omega$ , meaning that it was not able to build even a partial skeleton. On the other hand, partial failure occurs when the algorithm does not succeed in recovering a complete type tree from the partial specification, i.e., there remain some “holes” to be filled in. Such holes are marked by placeholders of the form  $\Omega_i$ , where  $i$  is some integer index. Such  $\Omega_i$ ’s correspond to the underscore characters generated by the  $build\_type\_tree()$  function above which have not been instantiated by the type unification algorithm. We could safely replace them with a plain  $\Omega$ , since the type  $\omega$  is a *wild card*, but we do not make this choice in the web tool, see Section 4.2.2.

The other way round is dealt with by the algorithm  $\mathcal{T}2\mathcal{I}$ . Here the input can be any type and the output is a specification of a symmetric relation; if the input type is binary, then the result is the specification of a partial involution. The algorithm processes all type variables occurring in a given type by means of the function  $var\_type2PI$ , which produces a list  $L$  of partial specification terms. If the type is binary the list has at most length 2. The

partial specification rules for that type variable are obtained by combining in all possible ways the terms of  $L$ . Repeating this procedure for all type variables of a type, we recover the corresponding partial specification. We give only  $var\_type2PI$ :

$$var\_type2PI(\alpha, \tau) = \begin{cases} [x_\alpha] & \text{if } \tau \equiv \alpha \\ l(var\_type2PI(\alpha, \tau')) ++ r(var\_type2PI(\alpha, \tau'')) & \text{if } \tau \equiv \tau' \rightarrow \tau'' \\ & \text{and } \alpha \in fv(\tau'), \alpha \in fv(\tau'') \\ l(var\_type2PI(\alpha, \tau')) & \text{if } \tau \equiv \tau' \rightarrow \tau'' \\ & \text{and } \alpha \in fv(\tau') \setminus fv(\tau'') \\ r(var\_type2PI(\alpha, \tau'')) & \text{if } \tau \equiv \tau' \rightarrow \tau'' \\ & \text{and } \alpha \in fv(\tau'') \setminus fv(\tau') \\ add\_index(u, var\_type2PI(\alpha, \bigwedge_{i=1}^n !_{v_i} \tau_i)) & \text{if } \tau \equiv \bigwedge_{i=1}^n !_{<u, v_i>} \tau_i \\ compose\_index(l, var\_type2PI(\alpha, \tau')) ++ \\ compose\_index(r, var\_type2PI(\alpha, \tau'')) & \text{if } \tau \equiv \tau' \wedge \tau'' \\ & \text{and } \alpha \in fv(\tau'), \alpha \in fv(\tau'') \\ & \text{and ! indices in } \tau \text{ have different } \pi_1 \text{'s} \\ compose\_index(l, var\_type2PI(\alpha, \tau')) & \text{if } \tau \equiv \tau' \wedge \tau'' \\ & \text{and } \alpha \in fv(\tau') \setminus fv(\tau'') \\ & \text{and ! indices in } \tau \text{ have different } \pi_1 \text{'s} \\ compose\_index(r, var\_type2PI(\alpha, \tau'')) & \text{if } \tau \equiv \tau' \wedge \tau'' \\ & \text{and } \alpha \in fv(\tau'') \setminus fv(\tau') \\ & \text{and ! indices in } \tau \text{ have different } \pi_1 \text{'s} \\ add\_index(u, var\_type2PI(\alpha, \tau')) & \text{if } \tau \equiv !_u \tau' \text{ and } \alpha \in fv(\tau') \\ [] & \text{otherwise} \end{cases}$$

where:

- $fv(\tau)$  is the set of free variables of type  $\tau$ ;
- in the expression  $\tau \equiv \bigwedge_{i=1}^n \tau_i$  the right-hand side term denotes the atomic components of  $\tau$ ;
- $++$  is the list concatenation operator;
- the functions  $l()$  and  $r()$ , when applied to a list (i.e., the return type of  $var\_type2PI()$ ), distribute, respectively the  $l$  and  $r$  constructors at the head of all elements of the list, i.e., each  $t \in L$  will be substituted by  $lt$  (resp.  $rt$ );
- $compose\_index(c, L)$  (where  $c$  is either  $l$  or  $r$ ) is a function distributing the constructor  $c$  to all the pairs of the list  $L$ , i.e., each  $\langle u, t \rangle \in L$  will be substituted by  $\langle cu, t \rangle$ ;
- $add\_index(u, t)$  is a function distributing the index  $u$  to all elements of the list  $L$ , i.e. each  $tinL$  will be substituted by  $\langle u, t \rangle$ .

Notice that in dealing with types whose outermost constructor is  $\wedge$ , we distinguish between the cases where all atomic components have a common prefix in their  $!$ 's, denoting that promotions have been carried after the  $\wedge$  operations, and the cases where there is no such common prefix, where the  $\wedge$ -constructor was indeed the last to be used in producing the type.

It is easy to check that:

► **Lemma 21.**

1. For any specification  $P$  of a partial involution,  $\mathcal{I}2\mathcal{T}(P)$  is a binary type.
2. For any binary type  $\tau$ ,  $\mathcal{T}2\mathcal{I}(\tau)$  is a specification of a partial involution.

The two transformations,  $\mathcal{I}2\mathcal{T}$  and  $\mathcal{T}2\mathcal{I}$ , are the inverse of one another in the sense:

► **Conjecture 22.**

1. Let  $M \in \Lambda^!$  be a closed term which reduces to a term in strong normal form, without the use of the  $\xi$ -rules, then:  $\vdash M : \mathcal{I}2\mathcal{T}(\llbracket M \rrbracket^{\mathcal{I}})$  and  $\mathcal{T}2\mathcal{I}(\mathcal{I}2\mathcal{T}(\llbracket M \rrbracket^{\mathcal{I}})) = \llbracket M \rrbracket^{\mathcal{I}}$ .
2. Let  $\tau$  be a binary type, then  $\mathcal{I}2\mathcal{T}(\mathcal{T}2\mathcal{I}(\tau)) = \tau$ .

When a closed term  $M \in \Lambda^!$  needs the  $\xi$ -rules to reduce to strong normal form its semantics as a partial involution does not necessarily yield an immediately meaningful type. We will discuss this issue further in Section 5.

Conjecture 22 applies also when we restrict to terms of  $\Lambda^{EAL}$ .

## 4 The Web Tool $\Lambda$ -symsym

In the following, we will illustrate how to use the web tool,  **$\Lambda$ -symsym**, available at <http://158.110.146.197:31780/automata/>, in order to experiment with the algorithms provided in the previous section. We will provide excerpts from various sessions illustrating how to convert  $\lambda^!$ -terms to combinators, from the latter to partial involutions, and finally to types. Moreover, we will also show how to infer a partial involution from a type. The web tool is implemented in the language Erlang, [17]. We chose this language because of its rich pattern matching features, its flexible constructors such as the *set data type* and for the large number of libraries which allow for a rapid prototyping. The fine details of the implementation appear in [13]

### 4.1 Encoding $\lambda$ -terms, partial involutions and types

First we give the syntax representation; the following tables contain the correspondences between the syntax we use with “pencil and paper”, the actual input syntax and the internal representation used by our algorithms.

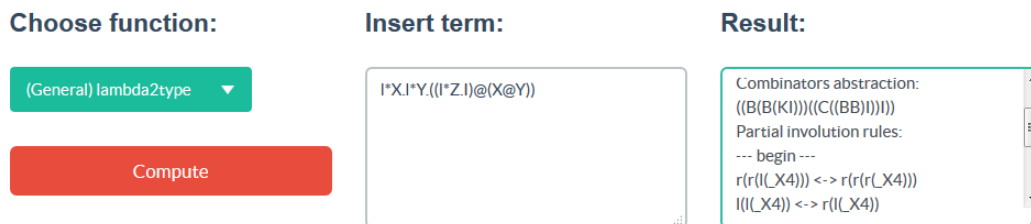
$\lambda$ -terms		
On the paper	Input syntax	Internal representation
$x$	X	{var, "X"}
$C$	C	{comb, C}
$MN$	M@N	{lapp, M, N}
$\lambda x.M$	l*X.M	{abs, {var, "X"}, M}
$\lambda!x.M$	l*!X.M	{abs_b, {var, "X"}, M}
$!M$	!M	{bang, M}

Partial Involutions		
On the paper	Input syntax	Internal representation
$\epsilon$	e	e
$x$	X	{var, "X"}
$lM$	lM	{l, M}
$rM$	rM	{r, M}
$\langle M, N \rangle$	<M,N>	{p, M, N}
$M \leftrightarrow N$	M <-> N	[{M,N},{N,M}]

Types		
On the paper	Input syntax	Internal representation
$\alpha$	A	{var, "A"}
$\alpha \longrightarrow \beta$	A -> B	{map, A, B}
$\alpha \wedge \beta$	A /\ B	{cap, A, B}
$!_i \alpha$	!I M	{bang, I, M}



■ **Figure 3** The interface of the web tool.

### 4.1.1 The web interface

Figure 3 depicts the web interface we can use to call the algorithms. It has 4 components:

1. a green drop-down menu allowing the user to choose the function to execute (e.g., `lambda2type()` in the general setting, as it appears in Figure 3); it provides a number of functions for the general case,  $\Lambda^!$  and for the  $\Lambda^{EAL}$  case, namely:
  - (i) the function `lambda2type()` which given a term produces its combinatory logic translation, its semantics as a specification of a partial involution, and the type corresponding to the specification;
  - (ii) the function `show_partial_involution` which produces a specification of a partial involution, given a type;
  - (iii) the function `show_type` which yields a type given a partial involution;
  - (iv) more functions which have not been discussed in this paper for dealing with polynomial terms, and Hindley-Milner Types;
2. a text area where the user can write the argument to pass to the selected function;
3. another text area where the result will appear after clicking the button “Compute”;
4. a red button (“Compute”) to start the computation.

## 4.2 Sessions

### 4.2.1 Strictly Linear Combinators in $\mathcal{I}$ are a $\lambda$ -algebra

One can see that the partial involutions corresponding to the two terms in input are equal. This is one of the many equations which need to be checked to show that the  $\xi$  rule holds.

```
Input:
l*X.l*Y.l*Z.X@(Y@Z)
Combinators abstraction:
((C((BB)((BB)I)))(C((BB)I)I))
Partial involution rules:
--- begin ---
l(r(X7)) <-> r(r(r(X7)))
l(l(X7)) <-> r(l(r(X7)))
r(l(l(X7))) <-> r(r(l(X7)))
--- end ---
Principal type:
((X10 -> X9) -> ((X8 -> X10) -> (X8 -> X9)))
```

```
Input:
B
Partial involution rules:
--- begin ---
r(r(r(X1))) <-> l(r(X1))
l(l(X1)) <-> r(l(r(X1)))
r(l(l(X1))) <-> r(r(l(X1)))
--- end ---
Principal type:
((X4 -> X3) -> ((X2 -> X4) -> (X2 -> X3)))
```

### 4.2.2 The $\xi$ -rule fails for erasers

The following example shows that the  $\xi$ -rule fails in  $\mathcal{I}$  when terms erase variables. The session on the right hand side shows that erased subterms might still leave an echo in the partial involution. This is the reason why we have not taken the standard  $\omega$ -equivalence

as for intersection types. Please bear in mind that, when an inferred type features some  $\_Omega_i$  variables (for  $i=1,2,3,\dots$ ), these variables stand for placeholders for any type. They are “holes” in the type tree structure which arise since partial involutions do not provide unnecessary information.

```
Input:
1*X.1*Y.((1*Z.I)@(X@Y))
Combinators abstraction:
((B(B(KI)))(C((BB)I)I))
Partial involution rules:
--- begin ---
r(r(1(_X4))) <-> r(r(r(_X4)))
l(1(_X4)) <-> r(1(_X4))
--- end ---
Principal type:
((_X5 -> _Omega3) -> (_X5 -> (_X6 -> _X6)))
```

```
Input:
1*X.1*Y.1*Z.Z
Combinators abstraction:
(K(KI))
Partial involution rules:
--- begin ---
r(r(1(_X2))) <-> r(r(r(_X2)))
--- end ---
Principal type:
(_Omega1 -> (_Omega2 -> (_X3 -> _X3)))
```

### 4.2.3 The $\xi$ -rule fails when modalities are erased

As was the case in the above example, also in this case there remains an echo in the partial involution when modalities are erased.

```
Input:
1*X.D@!X
Combinators abstraction:
((BD)(F!(I)))
Partial involution rules:
--- begin ---
l(<e, _X1>) <-> r(_X1)
--- end ---
Principal type:
((!e _X2) -> _X2)
```

```
Input:
1*X.D@!(D@!X)
Combinators abstraction:
((BD)((B(F!((BD)(F!(I))))))d)
Partial involution rules:
--- begin ---
l(<<e, e>, _X1>) <-> r(_X1)
--- end ---
Principal type:
((!<e, e> _X2) -> _X2)
```

### 4.2.4 The $\xi$ -rule fails in general

The following sessions illustrate that the  $\xi$ -rule fails in the encoding of standard  $\lambda$ -calculus via application in the Kleisli category. This issue will be briefly discussed in Section 5.

```
Input:
1*X.(1*!Y.Y@!Y)@(X@!X)
Combinators abstraction:
((B(W((C((BB)D))(F!(I)))))(B(F!(W((C((BB)D))(F!(I))))))d)
Partial involution rules:
--- begin ---
l(<<l(e), l(e)>, r(r(_X15))) <-> r(_X15)
l(<<_X15, l(e)>, l(<_X16, _X17>)) <-> l(<<_X15, r(_X16)>, _X17)
l(<<l(e), l(e)>, r(l(<_X15, _X16>)) <-> l(<<r(_X15), l(e)>, r(_X16)>))
--- end ---
Principal type:
(((!<e, e> (_Omega6 -> ((!_X17 _X18) -> _X21))) /\ (!<_X22, e> ((!_X19 _X20) -> _Omega10))) /\
  /\ (((!<_X22, _X19> _X20) /\ (!<_X17, e> (_Omega4 -> _X18))) /\
    /\ (!<_X22, e> ((!_X19 _X20) -> _Omega10)))
  -> _X21
```

but

```
Input:
1*X.X@!X@!(X@!X)
Combinators abstraction:
(W((C((BB)(W((C((BB)D))(F!(I)))))(B(F!(W((C((BB)D))(F!(I))))))d)
Partial involution rules:
--- begin ---
r(_X14) <-> l(<l(1(e)), r(r(_X14)))>
l(<l(1(e)), l(<_X14, _X15>)) <-> l(<l(r(_X14)), _X15>)
l(<l(1(e)), r(l(<_X14, _X15>)) <-> l(<r(<_X14, l(e)>), r(_X15)>)
l(<r(<_X14, l(e)>), l(<_X15, _X16>)) <-> l(<r(<_X14, r(_X15)>), _X16>)
--- end ---
Principal type:
(((!e ((!_X20 _X21) -> ((!_X17 _X23) -> _X24))) /\ (!_X20 _X21)) /\
  /\ (((!<_X17, e> ((!_X18 _X19) -> _X23)) /\ (!<_X17, _X18> _X19))) -> _X24
```



### 4.2.5 Church Binary Words in the $\lambda^{EAL}$ -calculus

This is the encoding of Church binary word  $\langle 0, 0, 1 \rangle$  as would appear in [6].

```

Input:
1*!X.1*!Y.!(1*Z.X@(X@(Y@Z)))
Combinators abstraction:
((BF)(W(BF)(F!(((C((BB)((BB)I))))((C((BB)((BB)I)))((C((BB)I)I))))))))
Partial involution rules:
--- begin ---
1(<l(_X9), r(_X10)>) <-> r(r(<_X9, r(_X10)>))
1(<l(_X9), l(_X10)>) <-> l(<r(_X9), r(_X10)>)
1(<r(_X9), l(_X10)>) <-> r(l(<_X9, r(_X10)>))
r(r(<_X9, l(_X10)>)) <-> r(l(<_X9, l(_X10)>))
--- end ---
Principal type:
(((!_X11 (_X15 -> _X14)) /\ (!_X11 (_X16 -> _X15))) -> ((!_X11 (_X12 -> _X16)) -> (!_X11 (_X12 -> _X14))))

```

## 5 Final Remarks

In this paper we have introduced the web tool **A-symsym** which implements various algorithms for computing the game semantics, in terms of involutions, of  $\lambda^!$ - and  $\lambda^{EAL}$ -terms and their types. Using **A-symsym** we have uncovered many peculiarities of game semantics. The fact that in Abramsky's model,  $\mathcal{I}$ , the  $\xi$ -rule fails in the general case, but not in the strictly linear case is perhaps the most remarkable one. Actually the portal **A-symsym** makes available algorithms, similar to the ones introduced in this paper, for dealing with the  $\lambda^{LAL}$ -calculus, introduced by the authors in [18], which captures in our framework the polynomial calculi underpinning *Light Linear Logic*, cf [20, 26]. Furthermore, partial involutions easily generalize to *symmetric partial relations*, see [18]. In this broader setting one can model ML-types. **A-symsym** provides algorithms also for this case.

We list, in a cursory manner, a number of intriguing issues deserving future attention.

- There is a plausible duality between types in  $\vdash^!$  and *partial involutions*, which should build up to a framework such as the one in [14, 1] for domain theory and intersection types, cf [9].
- Conjecture 22 appears to be very difficult to prove in full generality. A proof should relate two dual alternative ways of carrying out *unification*. In analogy to Abramsky's terminology, we could call the *wave-style* traditional unification and the *particle-style* unification, which goes on when linearly applying two specifications of involutions.
- The failure of the  $\xi$ -rule in the model  $\mathcal{I}$  uncovers the fact that sometimes, rather than computing the *most general unifier* between the corresponding types, linear application of partial involutions stops short of that and just computes some sort of *least general ancestor* of the type. This has not been fully appreciated yet. In this respect, exploring the connections of the present work with Girard's Clause Algebras and the Execution Formula as in [19] should prove fruitful.
- The  $\lambda^!$ -calculus is worthwhile investigating *per se*, but many well established notions need to be generalized, most notably normal forms, because of stuck terms. The paper [21] should provide some suggestions.
- J.-Y. Girard in [20] introduced a *polynomial time set-theory*, which bears some relation to the one of Fitch as presented in [25], which we have studied in [22]. The experience gathered with  $\lambda^{LAL}$ -calculus should improve our understanding of its applicability.

In conclusion we think that the fine structure of game semantics of the  $\lambda$ -calculus needs more investigation. This could be quite rewarding, provided more concrete experiments are carried out. **A-symsym** is our contribution to making them a little more feasible.

## References

- 1 Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1):1–77, 1991. doi:10.1016/0168-0072(91)90065-T.
- 2 Samson Abramsky. Retracing some paths in process algebra. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 1–17, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- 3 Samson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005. doi:10.1016/j.tcs.2005.07.002.
- 4 Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.
- 5 Samson Abramsky and Marina Lenisa. Linear realizability and full completeness for typed lambda-calculi. *Annals of Pure and Applied Logic*, 134(2):122–168, 2005. doi:10.1016/j.apal.2004.08.003.
- 6 Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing Polynomial and Exponential Complexity Classes in Elementary Lambda-Calculus. In Josep Diaz, Ivan Lanese, and Davide Sangiorgi, editors, *Theoretical Computer Science*, pages 151–163, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- 7 Patrick Baillot and Kazushige Terui. A Feasible Algorithm for Typing in Elementary Affine Logic. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 55–70, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 8 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983. URL: <http://www.jstor.org/stable/2273659>.
- 9 Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.
- 10 A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, and I. Scagnetto. Reversible Computation and Principal Types in  $\lambda!$ -calculus. *The Bulletin of Symbolic Logic*, 25(2):931–940, 2019.
- 11 Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, Marina Lenisa, and Ivan Scagnetto.  $\lambda!$ -calculus, Intersection Types, and Involutions. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSCD.2019.15.
- 12 Alberto Ciaffaglione, Furio Honsell, Marina Lenisa, and Ivan Scagnetto. The involutions-as-principal types/application-as-unification Analogy. In *LPAR*, volume 57, pages 254–270, 2018.
- 13 Alberto Ciaffaglione, Furio Honsell, Marina Lenisa, and Ivan Scagnetto. Web Appendix with Erlang code. <http://www.dimi.uniud.it/scagnett/pubs/automata-erlang.pdf>, 2020. Accessed: 2020-01-19.
- 14 M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium '82*, volume 112 of *Studies in Logic and the Foundations of Mathematics*, pages 241–262. Elsevier, 1984. doi:10.1016/S0049-237X(08)71819-6.
- 15 Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Light Logics and the Call-by-Value Lambda Calculus. *Logical Methods in Computer Science*, Volume 4, Issue 4, 2008. doi:10.2168/LMCS-4(4:5)2008.
- 16 Paolo Coppola and Simone Martini. Optimizing Optimal Reduction: A Type Inference Algorithm for Elementary Affine Logic. *ACM Trans. Comput. Logic*, 7(2):219–260, 2006. doi:10.1145/1131313.1131315.
- 17 Erlang Ecosystem Foundation. Erlang official website. <http://www.erlang.org>, 2020. Accessed: 2020-01-19.

- 18 I. Scagnetto F. Honsell, M. Lenisa. Types-as-game strategies for Hilbert style Computational Complexity. (submitted), 2020.
- 19 Jean-Yves Girard. Geometry of interaction III: accommodating the additives. *London Mathematical Society Lecture Note Series*, pages 329–389, 1995.
- 20 Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.
- 21 Furio Honsell and Marina Lenisa. Semantical analysis of perpetual strategies in  $\lambda$ -calculus. *Theoretical Computer Science*, 212(1):183–209, 1999. doi:10.1016/S0304-3975(98)00140-6.
- 22 Furio Honsell, Marina Lenisa, Luigi Liquori, and Ivan Scagnetto. Implementing Cantor’s Paradise. In Atsushi Igarashi, editor, *Programming Languages and Systems*, pages 229–250, Cham, 2016. Springer International Publishing.
- 23 André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 119(3), pages 447–468. [Cambridge, Eng.] Cambridge Philosophical Society., 1996.
- 24 Robin Milner. Is Computing an Experimental Science? *Journal of Information Technology*, 2(2):58–66, 1987. doi:10.1177/026839628700200202.
- 25 Dag Prawitz. *Natural deduction: A proof-theoretical study*. Courier Dover Publications, 2006.
- 26 Kazushige Terui. *Light logic and polynomial time computation*. PhD thesis, PhD thesis, Keio University, 2002.

## A An Erlang implementation of application between specifications of partial involutions

The definition of linear application between specifications of partial involutions follows the definition in [3], namely the flow diagram in Definition 11 (see Section 3). Given  $f, g, f \cdot g = f_{rr} \cup f_{rl}; g; (f_u; g)^*; f_{lr}$ , where  $f_{ij} = \{(u, v) \mid (i(u), j(v)) \in f\}$  for  $i, j \in \{l, r\}$ . Hence, we must begin by implementing a function `extract`, which infers l- and r- rewriting rules from the partial involution represented by the list L, according to `Op1` and `Op2`:

```
extract(L,Op1,Op2) ->
  case L of
    [] -> [];
    [{e,_}|T] -> extract(T,Op1,Op2);
    [{_,e}|T] -> extract(T,Op1,Op2);
    [{p,_,_}|T] -> extract(T,Op1,Op2);
    [{_,{p,_,_}|T] -> extract(T,Op1,Op2);
    [{T1,T2}|T] -> {O1,S1}=T1,
                  {O2,S2}=T2,
                  if
                    (O1==Op1) and (O2==Op2) -> [{S1,S2} | extract(T,Op1,Op2)];
                    true -> extract(T,Op1,Op2)
                  end
  end
end.
```

Thus, if F represents a partial involution, then `extract(F,r,l)` will compute  $F_{rl}$ .

Then, we define the *core* function `composeRuleList` which composes rule  $R1 \rightarrow R2$  with all the rules in L (exploiting the natural unification and substitution functions):

```
composeRuleList(R1,R2,L) ->
  case L of
    [] -> [];
    [{S1,S2}|T] -> {ExitStatus,MGU}=unify(R2,S1,[]),
                  if
                    (ExitStatus==ok) -> [{subListTerm(MGU,R1),subListTerm(MGU,S2)}
                                         | composeRuleList(R1,R2,T)];
                    true -> composeRuleList(R1,R2,T)
                  end
  end
end.
```

## 7:18 $\Lambda$ -Symsym

In order to avoid possible clashes between variable names, the `alpha` function defined below replaces all variables in `Ruleset1` which also occur in `Ruleset2` with freshly generated ones:

```
alpha(Ruleset1, Ruleset2) ->
  Vars1=ruleListVars(Ruleset1),
  Vars2=ruleListVars(Ruleset2),
  FreshSubst=separateVars(Vars1, Vars2),
  subListRuleset(FreshSubst, Ruleset1).
```

`alpha` is used fruitfully in the definition of `compose` which computes all possible chainings between rewriting rules of `L1` and `L2`:

```
compose(L1, L2) ->
  L1_Fresh=alpha(L1, L2),
  compose_fresh(L1_Fresh, L2).

compose_fresh(L1_Fresh, L2) ->
  case L1_Fresh of
    [] -> [];
    [H1|T1] -> {R1, R2}=H1,
                composeRuleList(R1, R2, L2)++compose_fresh(T1, L2)
  end.
```

The function `star` capitalizes on the definition of `compose`, in order to implement the computation of  $H; (F; G)^*$ :

```
star(H, F, G) ->
  S=compose(H, F),
  if
    S=[] -> H;
    true -> T=compose(S, G),
           if
             T=[] -> H;
             true -> H++star(T, F, G)
           end
  end.
```

At this point, the implementation of linear application, according to Definition 11 of Section 3, is straightforward:

```
lapp(F, G) ->
  FRR=extract(F, r, r),
  FRL=extract(F, r, l),
  FLL=extract(F, l, l),
  FLR=extract(F, l, r),
  FRL_G=compose(FRL, G),
  FRL_G_STAR=star(FRL_G, FLL, G),
  FRR++compose(FRL_G_STAR, FLR).
```

# Why Not W?

Jasper Hugunin  

Carnegie Mellon University, Pittsburgh, PA, USA

---

## Abstract

In an extensional setting,  $W$  types are sufficient to construct a broad class of inductive types, but in intensional type theory the standard construction of even the natural numbers does not satisfy the required induction principle. In this paper, we show how to refine the standard construction of inductive types such that the induction principle is provable and computes as expected in intensional type theory without using function extensionality. We extend this by constructing from  $W$  an internal universe of codes for inductive types, such that this universe is itself an inductive type described by a code in the next larger universe. We use this universe to mechanize and internalize our refined construction.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Type theory

**Keywords and phrases** dependent types, intensional type theory, inductive types,  $W$  types

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.8

**Supplementary Material** *Software (Source Code):*

<https://github.com/jashug/WhyNotW/releases/tag/v0.1>

archived at [swh:1:rel:75acce4d588f3622361f0adc3f1255ac24147669](https://swh.io/1:rel:75acce4d588f3622361f0adc3f1255ac24147669)

**Acknowledgements** I want to thank Jon Sterling and the anonymous reviewers for their helpful feedback.

## 1 Introduction

In intensional type theory with only type formers  $0$ ,  $1$ ,  $2$ ,  $\Sigma$ ,  $\Pi$ ,  $W$ ,  $\text{Id}$  and  $U$ , can the natural numbers be constructed?

The  $W$  type [12] captures the essence of induction (that we have a collection of possible cases, and for each case there is a collection of sub-cases to be handled inductively), and in extensional type theory it is straightforward to construct familiar inductive types out of it, including the natural numbers [6]. Taking the elements of the two-element type  $2$  to be  $\hat{0}$  and  $\hat{S}$ , we define

$$\tilde{\mathbb{N}} = W_{b:2}(\text{case } b \text{ of } \{\hat{0} \mapsto 0, \hat{S} \mapsto 1\}). \quad (1)$$

(the tilde distinguishes the standard construction from our refined construction of the natural numbers in Section 2)

However, as is well known [6, 10, 13, 14], in intensional type theory we cannot prove the induction principle for  $\tilde{\mathbb{N}}$  without some form of function extensionality. The obstacle is in the  $\hat{0}$  case, where we end up needing to prove  $P f$  for an arbitrary  $f : 0 \rightarrow \tilde{\mathbb{N}}$ , when we only know  $P (x \mapsto \text{case } x \text{ of } \{\})$ .

Can this obstacle be avoided? The answer turns out to be yes; in this paper, we show that refining the standard construction allows the natural numbers and many other inductive types to be constructed from  $W$  in intensional type theory. <sup>1</sup>

---

<sup>1</sup> These results have been formalized in Coq 8.12 [17]: see the link to supplementary material in the top matter of this article.



© Jasper Hugunin;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 8; pp. 8:1–8:9

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

### Type-theoretic notations and assumptions

We work in a standard intensional type theory with dependent function types  $\Pi_{a:A} B[a]$  (also written  $\forall_{a:A} B[a]$ ,  $(a : A) \rightarrow B[a]$ , non-dependent version  $A \rightarrow B$ , constructed as  $(x \mapsto y[x])$  or  $(\lambda x. y[x])$ ), dependent pair types  $\Sigma_{a:A} B[a]$  (also written  $(a : A) \times B[a]$ , non-dependent version  $A \times B$ , constructed as  $(x, y)$ , destructed as  $\mathbf{fst} p$ ,  $\mathbf{snd} p$ ), finite types  $0$ ,  $1$  (with inhabitant  $\star$ ),  $2$  (with inhabitants  $\mathbf{ff}$  and  $\mathbf{tt}$ , aliased to  $\hat{0}$  and  $\hat{S}$  when we are talking about constructing the natural numbers),  $\mathbb{W}$  types  $\mathbb{W}_{a:A} B[a]$  (constructor  $\mathbf{sup} a f$  for  $a : A$  and  $f : B[a] \rightarrow \mathbb{W}_a B[a]$ ), identity types  $\mathbf{Id}_A x y$  (constructor  $\mathbf{refl}$ , destruction of  $e : \mathbf{Id} x y$  keeps  $x$  fixed and generalizes over  $y$  and  $e$ ), and a universe  $\mathbf{U}$ . We define the coproduct  $A + B$  as  $\sum_{b:2} \mathbf{case} b \text{ of } \{\mathbf{ff} \mapsto A, \mathbf{tt} \mapsto B\}$ , and notate the injections as  $\mathbf{inl}$  and  $\mathbf{inr}$ .

Function extensionality is the principle that  $\forall_x \mathbf{Id} (f x) (g x)$  implies  $\mathbf{Id} f g$ , and uniqueness of identity proofs is the principle that  $\mathbf{Id}_{\mathbf{Id} x y} p q$  is always inhabited. We do *not* assume either of these principles.

We require strict  $\beta$ -rules for all type formers, and strict  $\eta$  for  $\Sigma$  (that  $p = (\mathbf{fst} p, \mathbf{snd} p)$ ) and  $\Pi$  (that  $f = (x \mapsto f x)$ ). For convenience we will also assume strict  $\eta$  for  $1$  (that  $u = \star$ ).

## 2 Constructing $\mathbb{N}$ (for real this time)

We run into problems in the  $\hat{0}$  case because we don't know that  $f = (x \mapsto \mathbf{case} x \text{ of } \{\})$  for an arbitrary  $f : 0 \rightarrow \tilde{\mathbb{N}}$ . To solve those problems, we will assume them away. To construct  $\mathbb{N}$ , we will first define a predicate  $\mathbf{canonical} : \tilde{\mathbb{N}} \rightarrow \mathbf{U}$  such that  $\mathbf{canonical}(\mathbf{sup} \hat{0} f)$  implies  $\mathbf{Id}(x \mapsto \mathbf{case} x \text{ of } \{\}) f$ . We then let  $\mathbb{N} = \sum_{x:\tilde{\mathbb{N}}} \mathbf{canonical} x$  be the canonical elements of  $\tilde{\mathbb{N}}$  (with  $\tilde{\mathbb{N}}$  defined by Equation (1)). This predicate will be defined by induction on  $\mathbb{W}$ , so we can start out with

$$\mathbf{canonical}(\mathbf{sup} x f) = ? : \mathbf{U} \quad (x : 2, f : \dots \rightarrow \tilde{\mathbb{N}}, \text{ may use } \mathbf{canonical}(f i) : \mathbf{U}).$$

The obvious next thing to do is to split by cases on  $x : 2$ :

$$\begin{aligned} \mathbf{canonical}(\mathbf{sup} \hat{0} f) &= ? : \mathbf{U} && (f : 0 \rightarrow \tilde{\mathbb{N}}, \text{ may use } \mathbf{canonical}(f i)), \\ \mathbf{canonical}(\mathbf{sup} \hat{S} f) &= ? : \mathbf{U} && (f : 1 \rightarrow \tilde{\mathbb{N}}, \text{ may use } \mathbf{canonical}(f i)). \end{aligned}$$

We need canonical terms to be *hereditarily* canonical, that is, we want to include the condition that all sub-terms are canonical. For the  $\hat{S}$  case, thanks to the strict  $\eta$  rules for  $1$  and  $\Pi$ , the types  $\mathbf{canonical}(f \star)$  and  $(i : 1) \rightarrow \mathbf{canonical}(f i)$  are equivalent; we can use either one. This will be the only condition we need for the  $\hat{S}$  case, so we can complete this part of the definition:

$$\mathbf{canonical}(\mathbf{sup} \hat{S} f) = \mathbf{canonical}(f \star).$$

The  $\hat{0}$  case is the interesting one. The blind translation of “every sub-term is canonical” is  $(i : 0) \rightarrow \mathbf{canonical}(f i)$ , but this leads to the same problem as before: without function extensionality we can't work with functions out of  $0$ . Luckily, we have escaped the rigid constraints of the  $\mathbb{W}$  type former, and have the freedom to translate the recursive condition as simply  $1$ . No sub-terms of zero, no conditions necessary!

$$\mathbf{canonical}(\mathbf{sup} \hat{0} f) = ? : \mathbf{U} \quad (f : 0 \rightarrow \tilde{\mathbb{N}})$$

That is all well and good, but we can't forget why we are here in the first place: we need  $\mathbf{Id}(x \mapsto \mathbf{case} x \text{ of } \{\}) f$ . Luckily, there is a hole just waiting to be filled:

$$\mathbf{canonical}(\mathbf{sup} \hat{0} f) = \mathbf{Id}(x \mapsto \mathbf{case} x \text{ of } \{\}) f.$$

$$\begin{aligned}
\tilde{\mathbb{N}} &= W_{b,2}(\text{case } b \text{ of } \{\hat{0} \mapsto 0, \hat{S} \mapsto 1\}) : \mathbb{U}, \\
\text{canonical} &: \tilde{\mathbb{N}} \rightarrow \mathbb{U}, \\
\text{canonical}(\text{sup } \hat{0} f) &= \text{Id } (x \mapsto \text{case } x \text{ of } \{\}) f, \\
\text{canonical}(\text{sup } \hat{S} f) &= \text{canonical}(f \star), \\
\mathbb{N} &= \Sigma_{x:\tilde{\mathbb{N}}} \text{canonical } x : \mathbb{U}, \\
\mathbb{O} &= (\text{sup } \hat{0} (x \mapsto \text{case } x \text{ of } \{\}), \text{refl}) : \mathbb{N}, \\
\mathbb{S} &= n \mapsto (\text{sup } \hat{S} (\star \mapsto \text{fst } n), \text{snd } n) : \mathbb{N} \rightarrow \mathbb{N}.
\end{aligned}
\tag{2}$$

$$\mathbb{N} = \Sigma_{x:\tilde{\mathbb{N}}} \text{canonical } x : \mathbb{U}, \tag{3}$$

$$\mathbb{O} = (\text{sup } \hat{0} (x \mapsto \text{case } x \text{ of } \{\}), \text{refl}) : \mathbb{N}, \tag{4}$$

$$\mathbb{S} = n \mapsto (\text{sup } \hat{S} (\star \mapsto \text{fst } n), \text{snd } n) : \mathbb{N} \rightarrow \mathbb{N}. \tag{5}$$

■ **Figure 1** The complete definition of  $\mathbb{N}$ .

## Induction

Now we are ready for the finale: induction for  $\mathbb{N}$  with the right computational behavior.

Assume we are given a type  $P[n]$  which depends on  $n : \mathbb{N}$ , along with terms  $\text{ISO} : P[\mathbb{O}]$  and  $\text{ISS} : \forall n:\mathbb{N} P[n] \rightarrow P[\mathbb{S} n]$ . Our mission is to define a term  $\text{rec}\mathbb{N} : \forall n:\mathbb{N} P[n]$ . Happily, the proof goes through if we simply follow our nose.

We begin by performing induction on  $\text{fst } n : \tilde{\mathbb{N}}$ , and then case on  $\hat{0}$  vs  $\hat{S}$ , just like the definition of `canonical`.

$$\begin{aligned}
\text{rec}\mathbb{N}(\text{sup } \hat{0} f, y) &= ? : P[(\text{sup } \hat{0} f, y)] & (f : 0 \rightarrow \tilde{\mathbb{N}}, y : \text{Id } (x \mapsto \text{case } x \text{ of } \{\}) f), \\
\text{rec}\mathbb{N}(\text{sup } \hat{S} f, y) &= ? : P[(\text{sup } \hat{S} f, y)] & (f : 1 \rightarrow \tilde{\mathbb{N}}, y : \text{canonical}(f \star)).
\end{aligned}$$

(where we may make recursive calls  $\text{rec}\mathbb{N}(f i, y')$  for any  $i$  and  $y'$ )

In the  $\hat{S}$  case,  $f = (\star \mapsto f \star)$  by the  $\eta$  rules for 1 and  $\Pi$ , and thus  $(\text{sup } \hat{S} f, y) = \mathbb{S} (f \star, y)$ . We can thus define

$$\text{rec}\mathbb{N}(\text{sup } \hat{S} f, y) = \text{ISS } (f \star, y) (\text{rec}\mathbb{N}(f \star, y)).$$

The  $\hat{0}$  case is again the interesting one, but it is only a little tricky. We know  $\text{ISO} : P[(\text{sup } \hat{0} (x \mapsto \text{case } x \text{ of } \{\}), \text{refl})]$ , and we want  $P[(\text{sup } \hat{0} f, y)]$ . But since we have  $y : \text{Id } (x \mapsto \text{case } x \text{ of } \{\}) f$ , this is a direct application of the eliminator for `Id`. We thus complete the definition of  $\text{rec}\mathbb{N}$  with

$$\text{rec}\mathbb{N}(\text{sup } \hat{0} f, y) = \text{case } y \text{ of } \{\text{refl} \mapsto \text{ISO}\}.$$

Examining the definitions, we can see that as long as we have strict  $\eta$  for  $\Sigma$  and strict  $\beta$  for `Id`,  $\text{rec}\mathbb{N} \mathbb{O} = \text{ISO}$  and  $\text{rec}\mathbb{N} (\mathbb{S} n) = \text{ISS } n (\text{rec}\mathbb{N} n)$ . Thus we have indeed defined the natural numbers with the expected induction principle and computational behavior in terms of the  $W$  type.

► **Theorem 1.** *The natural numbers can be constructed in intensional type theory with only type formers  $0, 1, 2, \Sigma, \Pi, W, \text{Id}$  and  $\mathbb{U}$ , such that the induction principle has the expected computational behavior.*

## 3 The General Case

Above, we have refuted a widely held intuition about the expressiveness of intensional type theory with  $W$  as the only primitive inductive type. Once we know we can construct the natural numbers, that we can construct lots of other inductive types is much less surprising.

## 8:4 Why Not W?

Given	
a type $P[n]$ depending on $n : \mathbb{N}$ ,	(6)
$\text{ISO} : P[0]$ ,	(7)
$\text{ISS} : \forall_{n:\mathbb{N}} P[n] \rightarrow P[S\ n]$ ,	(8)
we have	
$\text{rec}\mathbb{N} : \forall_{n:\mathbb{N}} P[n]$ ,	
$\text{rec}\mathbb{N}(\text{sup } \hat{0}f, y) = \text{case } y \text{ of } \{\text{refl} \mapsto \text{ISO}\}$ ,	(9)
$\text{rec}\mathbb{N}(\text{sup } \hat{S}f, y) = \text{ISS}(f \star, y) (\text{rec}\mathbb{N}(f \star, y))$ ,	
$\text{rec}\mathbb{N} 0 = \text{ISO}$ ,	(10)
$\text{rec}\mathbb{N}(S\ n) = \text{ISS } n (\text{rec}\mathbb{N } n)$ .	(11)

■ **Figure 2** Induction for  $\mathbb{N}$ .

Nevertheless, for completeness we define below an internal type of codes for inductive types along with the construction from  $\mathbb{W}$  types of the interpretation of those codes. For convenience, in this section we assume that we have not just one universe  $\mathbb{U}$  but an infinite cumulative tower of universes  $\mathbb{U}_0 : \mathbb{U}_1 : \dots : \mathbb{U}_i : \mathbb{U}_{i+1} : \dots$  all closed under  $0$ ,  $1$ ,  $2$ ,  $\Sigma$ ,  $\Pi$ ,  $\mathbb{W}$ , and  $\text{Id}$  such that  $A : \mathbb{U}_i$  implies  $A : \mathbb{U}_{i+1}$ .

The end result is a universe of inductive types which is self-describing, or “levitating” in the sense of [4].

### 3.1 Inductive Codes

We will let  $\text{Code}_i : \mathbb{U}_{i+1}$  be the type of codes for inductive types in  $\mathbb{U}_i$ , and implement it for now as a primitive inductive type. In Section 3.4 we will show how to construct  $\text{Code}$  itself from  $\mathbb{W}$ .

To define  $\text{Code}$ , we adapt the axiomatization of induction-recursion from [7]. Thus  $\text{Code}_i$  is generated by the constructors

$$\text{nil} : \text{Code}_i, \quad \text{nonind} : (A : \mathbb{U}_i) \rightarrow (A \rightarrow \text{Code}_i) \rightarrow \text{Code}_i, \quad \text{ind} : \mathbb{U}_i \rightarrow \text{Code}_i \rightarrow \text{Code}_i.$$

Looking at  $\mathbb{U}_i$  as the usual category of types and functions, a code  $A : \text{Code}_i$  defines an endofunctor  $F_A : \mathbb{U}_i \rightarrow \mathbb{U}_i$  defined by recursion on  $A$  by

$$F_{\text{nil}} X = 1, \tag{12}$$

$$F_{\text{nonind}(A,B)} X = \Sigma_{a:A} F_{(B\ a)} X, \tag{13}$$

$$F_{\text{ind}(I_x,B)} X = (I_x \rightarrow X) \times F_B X. \tag{14}$$

► **Example 2.** We can define a code for the natural numbers as

$$\text{“}\mathbb{N}\text{”} = \text{nonind}(2, b \mapsto \text{case } b \text{ of } \{\hat{0} \mapsto \text{nil}, \hat{S} \mapsto \text{ind}(1, \text{nil})\}) : \text{Code}_0.$$

Each code also defines a polynomial functor  $G_A X = \Sigma_{s:S_A} (P_A\ s \rightarrow X)$ , which is what is used in the standard construction:



$$S_{\text{nil}} = 1 \qquad P_{\text{nil}} \star = 0 \qquad (15)$$

$$S_{\text{nonind}(A,B)} = \Sigma_{a:A} S_{(B \ a)} \qquad P_{\text{nonind}(A,B)}(a, b) = P_{(B \ a)} b \qquad (16)$$

$$S_{\text{ind}(\text{Ix}, B)} = S_B \qquad P_{\text{ind}(\text{Ix}, B)} b = \text{Ix} + P_B b. \qquad (17)$$

$$G_A X = \Sigma_{s:S_A} (P_A s \rightarrow X) \qquad \tilde{\text{El}} A = \text{W}_{s:S_A} P_A. \qquad (18)$$

The idea here is that  $S_A$  collects up all the non-inductive data, and then  $P_A$  counts the number of inductive sub-cases.

There is an easy-to-define natural transformation  $\epsilon : F \Rightarrow G$ , and it even has a left inverse on objects, but without function extensionality  $\epsilon$  does not have a right inverse (roughly speaking,  $\epsilon$  is not surjective); there are usually terms  $g : G X$  not in the image of  $\epsilon$ . This is exactly the problem we ran into in the case of the natural numbers: the map  $(\star \mapsto (x \mapsto \text{case } x \text{ of } \{\})) : 1 \rightarrow (0 \rightarrow X)$  is not surjective. (The above  $S$ ,  $P$ , and  $\epsilon$  roughly correspond to Lemma 3 in [6])

The last component we need is  $\text{All}_A s : (Q : P_A s \rightarrow \mathbb{U}_j) \rightarrow \mathbb{U}_j$  (for universe level  $j \geq i$ ), the quantifier “holds at every position” (a refinement of  $\forall_p, Q p$ ):

$$\text{All}_{\text{nil}} \star Q = 1, \qquad (19)$$

$$\text{All}_{\text{nonind}(A,B)}(a, b) Q = \text{All}_{(B \ a)} b Q, \qquad (20)$$

$$\text{All}_{\text{ind}(\text{Ix}, B)} b Q = (\forall_i, Q (\text{inl } i)) \times \text{All}_B b (Q \circ \text{inr}). \qquad (21)$$

Noting that  $\text{snd}(\epsilon t) : P(\text{fst}(\epsilon t)) \rightarrow X$  enumerates the sub-terms of  $t : F X$ , we find that  $\text{All}(Q \circ \text{snd}(\epsilon t))$  lifts a predicate  $Q : X \rightarrow \mathbb{U}_j$  to a predicate over  $t : F X$ .

► **Lemma 3.** *There is an equivalence  $r$  (à la Voevodsky, a function with contractible fibers)*

$$r : F(\Sigma_{x:X} C x) \simeq \Sigma_{(t:F X)} \text{All}(C \circ \text{snd}(\epsilon t)). \qquad (22)$$

**Proof.** Follows easily by induction on the code  $A$ . We use equivalences à la Voevodsky as a concrete definition of coherent equivalences, which are the “right” way to define type equivalence in the absence of UIP. ◀

### 3.2 The General Construction

We are finally ready to define the true construction of inductive types  $\text{El} : \text{Code} \rightarrow \mathbb{U}_i$ . As with natural numbers, we define a “canonicity” predicate on  $\tilde{\text{El}} A$ , which says that “all subterms are canonical, and this node is in the image of  $\epsilon$ ”. This translates as:

$$\text{canonical}(\text{sup } sf) = \text{All}(\text{canonical} \circ f) \times (t : F(\tilde{\text{El}} A)) \times \text{Id}_{G(\tilde{\text{El}} A)}(\epsilon t)(s, f) : \mathbb{U}_i, \qquad (23)$$

and thus we finally have

$$\text{El } A = \Sigma_{x:\tilde{\text{El}} A} \text{canonical } x. \qquad (24)$$

For the constructors, we expect to have  $\text{intro} : F(\text{El } A) \rightarrow \text{El } A$ , which we define by

$$\text{intro } x = (\text{sup } (\epsilon(\text{fst}(r x))), (\text{snd}(r x), \text{fst}(r x), \text{refl})). \qquad (25)$$

using the equivalence  $r$  from Lemma 3 to split  $x : F(\text{El } A)$  into  $\text{fst}(r x) : F(\tilde{\text{El}} A)$  and  $\text{snd}(r x) : \text{All}(\text{canonical} \circ \text{snd}(\epsilon \text{fst}(r x)))$ .

### 3.3 General Induction

When we go to define the induction principle for  $\text{El } A$ , we are given  $P : \text{El } A \rightarrow \mathbb{U}_j$  for some  $j \geq i$  and the induction step  $\text{IS} : \forall_{(x:F(\text{El } A))} \text{All}(P \circ \text{snd}(\epsilon x)) \rightarrow P(\text{intro } x)$ , and want to define  $\text{rec} : \forall_{(x:\text{El } A)} P x$ . The definition proceeds by induction on  $\text{fst } x$ :

$$\text{rec}(\text{sup } sf, (h, t, e)) = ? : P(\text{sup } sf, (h, t, e)) \quad h : \text{All}(\text{canonical} \circ f) \quad e : \text{Id}(\epsilon t)(s, f),$$

and we have induction hypothesis  $H = p \mapsto c \mapsto \text{rec}(f p, c) : \Pi_p \Pi_c P(f p, c)$ . Next, we destruct the identity proof  $e$ , generalizing over both  $h$  and  $H$ , leaving us with

$$\text{rec}(\text{sup}(\epsilon t), (h, t, \text{refl})) = ? : P(\text{sup}(\epsilon t), (h, t, \text{refl})),$$

for  $t : F(\tilde{\text{El}} A)$ ,  $h : \text{All}(\text{canonical} \circ \text{snd}(\epsilon t))$ , and  $H : \Pi_p \Pi_c P(\text{snd}(\epsilon t) p, c)$ . The last step to bring us in line with the definition of  $\text{intro}$  is to use the equivalence from Lemma 3 to replace  $(t, h)$  with  $r x$  for some  $x : F(\text{El } A)$ , leaving us with

$$\text{rec}(\text{sup}(\epsilon(\text{fst}(r x))), (\text{snd}(r x), \text{fst}(r x), \text{refl})) = ? : P(\text{intro } x)$$

and induction hypothesis  $H : \Pi_p \Pi_c P(\text{snd}(\epsilon(\text{fst}(r x))) p, c)$ . We can then apply  $\text{IS}$ , but that leaves us with an obligation to prove  $\text{All}(P \circ \text{snd}(\epsilon x))$ . Fortunately, it is easy to show by induction on the code  $A$  that our hypothesis  $H$  is sufficient to dispatch this obligation.

This completes the definition of the induction principle, and it can be observed on concrete examples like the natural numbers to have the expected computational behavior. We can also prove a propositional equality  $\text{Id}(\text{rec}(\text{intro } x))(\text{IS } x(\text{rec} \circ \text{snd}(\epsilon x)))$  witnessing the expected computation rule, and observe on concrete examples that this witness computes to reflexivity. The details of this construction have all been formalized in Coq.

### 3.4 Bootstrapping

In Section 3.1 we postulated the type  $\text{Code}_i$  to be a primitive inductive type, which leads to the question of whether the general construction we have proposed is *really* constructing inductive types out of  $\mathbb{W}$  or whether it is making sneaky use of the inductive structure of  $\text{Code}_i$  to perform the construction.

As a first observation,  $\text{Code}_i : \mathbb{U}_{i+1}$  while  $\text{El} : \text{Code}_i \rightarrow \mathbb{U}_i$ , thus  $\text{Code}_i$  can't appear as data in  $\text{El } A$ : it is too big! However, this argument doesn't show that we can completely eliminate  $\text{Code}_i$  from the construction.

Next, we observe that the inductive type  $\text{Code}_i$  itself has a code “ $\text{Code}_i$ ” :  $\text{Code}_{i+1}$ :

$$\begin{aligned} \text{“Code}_i\text{”} = & \text{nonind}((1 + \mathbb{U}_i) + \mathbb{U}_i, t \mapsto \text{case } t \text{ of } \{ \\ & \text{inl}(\text{inl } \star) \mapsto \text{nil}, & (\text{case nil}) \\ & \text{inl}(\text{inr } A) \mapsto \text{ind}(A, \text{nil}), & (\text{case nonind}) \\ & \text{inr } Ix \mapsto \text{ind}(1, \text{nil}), & (\text{case ind}) \\ & \}). \end{aligned}$$

Then we can propose to define  $\text{Code}_i = \text{El} \text{“Code}_i\text{”}$ , but this is a circular definition: we define  $\text{Code}_i$  by using recursion on  $\text{Code}_{i+1}$ . What we really want, and in some ways should be able to expect, is that  $\text{El} \text{“Code}_i\text{”}$  *computes* to a normal form which no longer mentions  $\text{Code}$  but is expressed purely in terms of  $\mathbb{W}$ . We could then tie the knot by defining  $\text{Code}_i$  to be what  $\text{El} \text{“Code}_i\text{”}$  *will compute to*, once we have defined  $\text{El}$ .

There is just one minor, rather technical problem to resolve, which is that currently  $\text{El}$  (which is defined by recursion on codes) gets stuck on  $\text{El}(\text{case } t \text{ of } \{ \dots \})$  which is used to branch on constructor tags; we are missing some sort of commuting conversion [9, section 10]. Fortunately, this problem is easy to work around by reifying the operation of branching on constructor tags as part of  $\text{Code}$ . We add another constructor

$$\text{choice} : \text{Code}_i \rightarrow \text{Code}_i \rightarrow \text{Code}_i, \quad F_{\text{choice}(A,B)} X = F_A X + F_B X \quad (26)$$

which encodes the simple binary sum of functors, specializing the dependent sum of functors  $\text{nonind}(2, b \mapsto \text{case } b \text{ of } \{ \dots \})$  (but with all proofs essentially the same). With this in hand, we can define

$$\begin{aligned} \text{"Code}_i\text{"} = & \text{choice}(\text{choice}( & & (27) \\ & \text{nil}, & & (\text{case nil}) \\ & \text{choice}( & & \\ & \quad \text{nonind}(\mathbb{U}_i, A \mapsto \text{ind}(A, \text{nil})), & & (\text{case nonind}) \\ & \quad \text{ind}(1, \text{ind}(1, \text{nil}))), & & (\text{case choice}) \\ & \text{nonind}(\mathbb{U}_i, \text{Ix} \mapsto \text{ind}(1, \text{nil}))). & & (\text{case ind}) \end{aligned}$$

With this adjustment, the structure of the code is not hidden inside  $\text{case}$ , and the computation of  $\text{El}$  “Code<sub>*i*</sub>” proceeds to completion without becoming stuck, resulting in a term which does not mention  $\text{Code}$  at all. From there, we can define  $\text{El}$  such that  $\text{El}$  “Code<sub>*i*</sub>” =  $\text{Code}_i$ , as in [4] but with no invisible cables, just the  $\mathbb{W}$  type.

► **Theorem 4.** *In intensional type theory with type formers  $0, 1, 2, \Sigma, \Pi, \mathbb{W}, \text{Id}$  and an infinite tower of universes  $\mathbb{U}_i$ , we can construct terms  $\text{Code}_i : \mathbb{U}_{i+1}$  and  $\text{El} : \text{Code}_i \rightarrow \mathbb{U}_i$  such that  $\text{El } A$  is an inductive type, and we can also construct terms “Code<sub>*i*</sub>” :  $\text{Code}_{i+1}$  such that  $\text{El}$  “Code<sub>*i*</sub>” =  $\text{Code}_i$ . Furthermore,  $\text{Code}_i$  is not trivial: it contains codes for natural numbers, lists, binary trees, and many other inductive types, including inductive types such as  $\mathbb{W}$  that have infinitary inductive arguments.*

## 4 Discussion

### 4.1 Composition

Being codes for functors, one may ask if  $\text{Code}_i$  is closed under composition of functors? As with the codes for inductive-recursive types we have modified, without function extensionality we do not appear to have composition (for similar reasons as considered in [8]). Indeed, experiments suggest that the general construction of a class of inductive types closed under composition of the underlying functors essentially requires function extensionality. Even worse, to get definitional computation rules for the resulting inductive types, all our attempts have required that transporting over  $\text{funext}(x \mapsto \text{refl})$  computes to the identity, a property which not even cubical type theory [5] satisfies (it is satisfied, however, by observational type theory [2]). Thus, we do not know how to combine a class of inductive types closed under composition constructed from the  $\mathbb{W}$  type as we have in Section 3 with the the principle of Univalence [16] while maintaining good computational behavior.

We do however wish to emphasize that the construction in Section 3 (which is not closed under composition) is completely compatible with Univalence, and could be implemented in cubical type theory as long as an identity type with strict  $\beta$  rule is used.

## 4.2 Canonicity

Despite being constructed from  $W$  types, our natural numbers enjoy the canonicity property (that for every closed term  $n$  of type  $\mathbb{N}$ , either  $n = 0$  or  $n = S m$  for some closed  $m : \mathbb{N}$ ), at least as long as  $2$  and  $\text{Id}$  enjoy canonicity (closed  $b : 2$  implies  $b = \hat{0}$  or  $b = \hat{1}$ , and closed  $e : \text{Id } x \ y$  implies  $e = \text{refl}$  and  $x = y$ ). The trick is that when we have some representation of zero, it looks like  $(\text{sup } \hat{0} f, e)$ , where  $e$  is a closed term of type  $\text{Id } (x \mapsto \text{case } x \text{ of } \{ \}) f$ , and thus by canonicity for  $\text{Id}$ , this must be  $(\text{sup } \hat{0} (x \mapsto \text{case } x \text{ of } \{ \}), \text{refl}) = 0$ .

However, in a situation like cubical type theory where function extensionality holds,  $\text{Id}$  no longer enjoys canonicity, and neither does our construction of the natural numbers.

## 4.3 Problems

What are the problems with using this construction as the foundation for inductive types in a proof assistant? While we have shown bare possibility, this is not an obviously superior solution when compared to the inductive schemes present in proof assistants today.

The construction is complex, which has the possibility of confusing unification and other elaboration algorithms. While the reduction behavior simulates the expected such, the reduction engine has to make many steps to simulate one step of a primitive inductive type, which can lead to a large slowdown. As an example, we observed the general construction slow down from seconds to check to half an hour when replacing primitive inductive types the bootstrapped definition of `Code`. Understanding exactly why this slowdown happens and how to alleviate it is an important question to be answered before attempting to apply this construction in practice.

There are also some (fairly esoteric) limitations to the expressivity of this construction. Nested inductive types such as `Inductive tree := node : list tree → tree` do not appear to be constructible, nor do mutual inductive types landing in a mixture of impredicative and predicative sorts at different levels, and nor do inductive-inductive types.

## 4.4 Setoids

In [15], Palmgren uses  $W$  types to construct a setoid model of extensional type theory in intensional type theory, including the natural numbers. In contrast, we have different goals (we are not concerned with extensional type theory), and our construction has different properties: we construct the natural numbers as a set not a setoid, with definitional computation rules and canonicity rather than working only up to an extensional setoid notion of equality. Other work on setoid models includes [11] and [1].

## 4.5 Conclusion

We have shown that intensional type theory with  $W$  and  $\text{Id}$  types is more expressive than was previously believed. It supports not only the natural numbers, but a whole host of inductive types, generated by an internal type of codes, which is itself an inductive type coded for by itself (one universe level up). This brings possibilities for writing generic programs acting on inductive types internally (like in [3]), and perhaps simplifies the general study of extensions of intensional type theory: once you know  $W$  works, you know lots of inductive types work.

Thus we return to the titular question: why not use  $W$  as the foundation of inductive types, for example in a proof assistant like Coq or Agda? Equipped with this result, one can no longer say that it is impossible.

## References


- 1 Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 412–420. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782636.
- 2 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.
- 3 Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.
- 4 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010. doi:10.1145/1863543.1863547.
- 5 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2015.5.
- 6 Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf’s type theory. *Theoretical Computer Science*, 176(1):329–335, 1997. doi:10.1016/S0304-3975(96)00145-4.
- 7 Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA ’99, L’Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999. doi:10.1007/3-540-48959-2\_11.
- 8 Neil Ghani, Conor McBride, Fredrik Nordvall Forsberg, and Stephan Spahn. Variations on inductive-recursive definitions. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 63:1–63:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.63.
- 9 Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1990. Translated and with appendices by Paul Taylor and Yves Lafont. URL: <http://www.paultaylor.eu/stable/prot.pdf>.
- 10 Healfdene Goguen and Zhaohui Luo. Inductive data types: Well-ordering types revisited. *Logical Environments*, 1992. URL: <https://www.cs.rhul.ac.uk/home/zhaohui/WTYPES93.pdf>.
- 11 Martin Hoffman. *Extensional Constructs in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.
- 12 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by G. Sambin of a series of lectures given in Padua, 1980.
- 13 Conor McBride. W-types: good news and bad news, March 2010. URL: <https://mazzo.li/epilogue/index.html%3Fp=324.html>.
- 14 Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf’s Type Theory*. Oxford University Press, 1990.
- 15 Erik Palmgren. From type theory to setoids and back, 2019. arXiv:1909.01414.
- 16 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: <https://homotopytypetheory.org/book/>.
- 17 The Coq Development Team. The Coq proof assistant, version 8.12.0, 2020. doi:10.5281/zenodo.4021912.



# Subtype Universes

Harry Maclean  

Royal Holloway, University of London, UK

Zhaohui Luo 

Royal Holloway, University of London, UK

---

## Abstract

---

We introduce a new concept called a subtype universe, which is a collection of subtypes of a particular type. Amongst other things, subtype universes can model bounded quantification without undecidability. Subtype universes have applications in programming, formalisation and natural language semantics. Our construction builds on coercive subtyping, a system of subtyping that preserves canonicity. We prove Strong Normalisation, Subject Reduction and Logical Consistency for our system via transfer from its parent system  $UTT[C]$ . We discuss the interaction between subtype universes and other sorts of universe and compare our construction to previous work on Power types.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Type theory

**Keywords and phrases** Type theory, coercive subtyping, subtype universe

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.9

## 1 Introduction

In this paper we define a new sort of universe, which we call a subtype universe. The key idea is embodied in the following two pseudo-rules:

$$\begin{array}{c} \text{UNIVERSE-FORM} \\ \Gamma \vdash A : \mathbf{Type} \\ \hline \Gamma \vdash U(A) : \mathbf{Type} \end{array} \qquad \begin{array}{c} \text{UNIVERSE-INTRO} \\ \Gamma \vdash B \leq A \\ \hline \Gamma \vdash B : U(A) \end{array}$$

The first rule states that for any type  $A$ , there is a type  $U(A)$  which we call the subtype universe of  $A$ . The second rule states that any subtype of  $A$  is an object of  $U(A)$ .  $U(A)$  is therefore a type representing a collection of all subtypes of  $A$ . It is similar to universes such as  $Type_0$  in that its objects are types (technically, names for types), but whilst  $Type_0$  contains all types (at the time of formation, at least), the membership of a type in  $U(A)$  is based on the presence of a subtyping judgement between the type in question and  $A$ .

Subtype universes provide a simple model for *bounded quantification*, a concept first introduced by Cardelli and Wegner for the language Fun[5]. Bounded quantification extends the notion of parametric polymorphism with support for subtypes. In a system with support for subtyping, the bounded quantifier  $\Pi A \leq B. T$  binds a type  $A$  in the body  $T$  under the constraint that  $A$  is a subtype of  $B$ . In essence, bounded quantification allows a function to be defined over all subtypes of a particular type.

A typical use of bounded quantification is in writing operations on records. Consider a system with record types similar to [13], although for simplicity without dependence of record fields on each other. We write record types as  $R := \langle \rangle \mid \langle R, l : A \rangle$  and records as  $r := \langle \rangle \mid \langle r, l = a : A \rangle$ . Record types each have a corresponding kind  $RType[L]$ , where  $L$  is the set of labels occurring in the record type. We can define the following function, which translates a one-dimensional point by a given amount to the right.

$$\begin{aligned} \text{translate} X : Nat &\rightarrow \langle x : Nat \rangle \rightarrow \langle x : Nat \rangle \\ \text{translate} X(n, r) &= \text{set}(r, x, r.x + n) \end{aligned}$$



© Harry Maclean and Zhaohui Luo;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 9; pp. 9:1–9:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$\langle x : Nat \rangle$  is a record type containing a single field  $x$  of type  $Nat$ .  $set(r, l, a)$  is a primitive operation on records which updates the record  $r$ , setting the value of the field labelled  $l$  to  $a$ .  $r.l$  selects the value in the record  $r$  corresponding to the label  $l$ .

We may wish to apply this function to coordinates in higher dimensions, such as objects of type  $\langle x : Nat, y : Nat \rangle$ . This prompts us to consider a subtyping relation for records, and there is a natural one arising from the record restriction operator  $[r]$ , which removes the outermost field of a record:  $\langle R, l : A \rangle \leq_{\square} R$ . In other words, any extension of a record type  $R$  by additional fields produces a subtype of  $R$ . Thus we have  $\langle x : Nat, y : Nat \rangle \leq \langle x : Nat \rangle$ . We can now apply  $translateX$  to objects of  $\langle x : Nat, y : Nat \rangle$ , but this will implicitly downcast them, as the result type is  $\langle x : Nat \rangle$ .

Bounded quantification solves this problem by introducing quantification over subtypes. In this way the original subtype is named and can be given in the result type. With bounded quantification we can write the type of  $translateX$  as

$$translateX : \Pi R \leq \langle x : Nat \rangle. Nat \rightarrow R \rightarrow R$$

which describes a function that takes an argument of some arbitrary record type  $R$  constrained to be a subtype of  $\langle x : Nat \rangle$ . When a function of this type is applied to an argument of type  $\langle x : Nat, y : Nat \rangle$ ,  $R$  is instantiated to  $\langle x : Nat, y : Nat \rangle$  and the result is an object of the same type.

System  $F_{\leq}[4]$  is System F[9, 22] extended with subtyping and bounded quantification, and is a foundation for much of the research on subtyping in functional programming languages. However there is one drawback: typechecking in  $F_{\leq}$  is undecidable [20]. The crux of the problem is the rule for subtyping between bounded quantifiers:

$$\frac{\Gamma \vdash A_1 \leq B_1 \quad \Gamma, x \leq A_1 \vdash B_2 \leq A_2}{\Gamma \vdash \Pi(x \leq B_1). B_2 \leq \Pi(x \leq A_1). A_2}$$

When combined with a  $Top$  type, of which every type is a subtype, this rule causes the subtyping relation to become undecidable, which in turn causes typechecking to become undecidable [2]. Various modifications have been proposed to get around this problem[6, 23]. For example, disallowing  $Top$  in the bounds of quantifiers or requiring  $A_1 = B_1$  in the rule above. Each has its own trade-offs in terms of expressiveness and algorithmic practicality.

Because of this undecidability result and the difficulties in extending System F with bounded quantification, many researchers have thought that extending dependent type theories with bounded quantification would also be problematic, or at least, it would not be an easy task. This has turned out to be mistaken. We take up this challenge in this paper and show that bounded quantification can be modelled by subtype universes in a way that maintains nice meta-theoretic properties. Moreover, our system is a full dependent type theory, providing richer types than  $F_{\leq}$ .

With subtype universes, bounded quantification can be modelled using normal  $\Pi$  types: the type  $\Pi X : U(A). B$  is equivalent to  $\Pi X \leq A. B$ . However it is important to note that  $\Pi$  types cannot model all uses of subtype universes. A subtype universe  $U(A)$  is a type of types, whereas  $\Pi$  is a type of functions. Using subtype universes we can construct types such as  $A \rightarrow U(A)$ . The right hand side of this function type is a type whose objects are subtypes of  $A$ , and there is no equivalent to this using  $\Pi$  types.

We use coercive subtyping, which is a subtyping system well suited to type theories due to its preservation of canonicity. The system  $UTT[C]$  is an extension of  $UTT$  [11] with coercive subtyping. We further extend this system with support for subtype universes, forming the system  $UTT[C]_U$ . The extension consists of a handful of new syntactic forms and six new



typing rules, which are described in Section 2. Working in a dependent type theory rather than a weaker language allows us to apply the concept to a wide range of fields. Section 3 describes examples applicable to programming, formalisation and natural language semantics. Section 4 proves several important meta-theoretic properties, including logical consistency and strong normalisation. Section 5 discusses the design decisions we have taken and some interesting alternatives.

## 2 Subtype Universes

In Martin-Löf’s intuitionistic type theory [17] the concept of a universe is introduced to represent a collection of types which is closed under specific type-forming operations. Typically one starts by defining a group of base type-forming operations and then one defines a universe  $Type_0$  which contains the closure of these operations.  $Type_0$  is itself a type, and can be used in combination with other type-forming operations to form new types. For example, we can construct the polymorphic identity function  $\Pi T : Type_0. \Pi x : T. T$ . A function with this type can be applied to any type in  $Type_0$ , but not to  $Type_0$  itself. We can construct a more powerful type for the identity function by repeating the process: we define a new universe  $Type_1$  which contains the closure of all type-forming operations *including*  $Type_0$ . We can then construct the type  $\Pi(T : Type_1). \Pi(x : T). T$ , objects of which can be applied to  $Type_0$ . This process can be iterated indefinitely, forming a hierarchy of predicative universes, each one stronger than the previous one. This hierarchy allows us to quantify over arbitrarily large collections of types, providing great proof-theoretic strength. At the same time, the absence of a “type of all types” means that it neatly avoids Girard’s Paradox[8]. Intuitively, this construction of universes is an application of the reflection principle well known to set theory, and there are analogous constructions in other fields (such as the Grothendieck universes of category theory). Universes are typically expressed in either *Tarski* style or *Russell* style. Tarski style is more explicit, and to avoid ambiguity it is the style we use here.

### 2.1 Tarski style universes

The Tarski formulation introduces a new type for each universe, objects of which are names for other types. Alongside, we introduce a family of operators to map names to their corresponding types. We will briefly walk through this construction, as our system builds on some of the concepts. Firstly, to represent each universe we form a type  $Type_i$ , where  $i$  is a positive integer indicating the level of the universe.

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash Type_i : \mathbf{Type}}$$

Objects of the universe  $Type_i$  are names of other types. For each universe we introduce a reflection operator  $\mathbf{T}_i$  which maps names in  $Type_i$  to their corresponding types.

$$\frac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{T}_i(a) : \mathbf{Type}}$$

Now whenever we introduce a new type to the system, we also introduce its name in each universe. For example, if we introduce a type  $Nat$  of natural numbers then we would also add the following axiom, which states that each universe contains a name  $nat$  for  $Nat$ .

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash \mathit{nat}_i : \mathit{Type}_i}$$

We also add rules defining how  $\mathbf{T}_i$  behaves on these new names:

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash \mathbf{T}_i(\mathit{nat}_i) = \mathit{Nat} : \mathbf{Type}}$$

Finally, for each universe we introduce a lifting operator  $\mathbf{t}_i$  which “lifts” names from a universe  $\mathit{Type}_i$  into the successive universe  $\mathit{Type}_{i+1}$ .

$$\frac{\Gamma \vdash a : \mathit{Type}_i}{\Gamma \vdash \mathbf{t}_i(a) : \mathit{Type}_{i+1}}$$

$$\frac{\Gamma \vdash a : \mathit{Type}_i}{\Gamma \vdash \mathbf{T}(\mathbf{t}_i(a)) = \mathbf{T}(a) : \mathbf{Type}}$$

Thus (informally) for the type  $\mathit{Nat}$  we have a name  $\mathit{nat}_0$  in  $\mathit{Type}_0$ , and  $\mathbf{T}_0(\mathit{nat}_0)$  yields  $\mathit{Nat}$ . We can apply the lifting operator  $\mathbf{t}_1$  to this name, giving  $\mathbf{t}_1(\mathit{nat}_0) = \mathit{nat}_1$ , and of course  $\mathbf{T}_1(\mathit{nat}_1) = \mathit{Nat}$ . We can lift again, giving  $\mathbf{t}_2(\mathit{nat}_1) = \mathit{nat}_2$ , and so on.

## 2.2 Coercive subtyping

Coercive subtyping [12, 15] is a model of subtyping in type theories which expresses the subtyping relationship via a specific *coercion*, which is a function from the subtype to the supertype. It is a powerful form of subtyping which is particularly well suited to type theories with canonical objects, as it preserves canonicity [15]. A type theory  $T$  can be extended with coercive subtyping by adding two new judgement forms, for subtyping and subkinding. We will focus on the former; for a full description of this extension we refer the reader to [15]. The subtyping judgement is written  $\Gamma \vdash A \leq_c B : \mathbf{Type}$ . It declares that  $A$  is a proper subtype of  $B$  via a coercion  $c : (A)B$ .<sup>1</sup> There are associated rules that define subtyping to be congruent and transitive, among other things. A particularly important addition is the coercive definition rule:

$$\frac{\Gamma \vdash f : (x : B)C \quad \Gamma \vdash a : A \quad \Gamma \vdash A \leq_c B}{\Gamma \vdash f(a) = f(c(a)) : [c(a)/x]C}$$

This rule states that if  $A \leq_c B$ , then a function  $f$ , despite having domain  $B$ , may be applied to an object  $a$  of  $A$ . When this happens, it is equal to  $f(c(a))$ , which is the application of  $f$  to the coercion of  $a$  to an object of  $B$ , using the specific coercion  $c$ . This is the primary mechanism by which coercive subtyping relations are put to work.

<sup>1</sup> The systems described in this paper are defined in the meta-level framework LF, which is a typed version of Martin-Löf’s Logical Framework. Where appropriate we will use LF syntax. In brief,  $(x : A)B$  is the type of a meta-level function from  $A$  to  $B$ , where  $x : A$  is bound in  $B$ .  $[x : A]b$  is a meta-level function from an object of  $A$  to an expression  $b$ , where  $x : A$  is bound in  $b$ . See [11] for a full description of the language.

Finally, we add a set  $\mathbb{C}$  of axiomatic subtyping judgements. The only restriction on  $\mathbb{C}$  is that it must be *coherent*. This means that all coercions between any two types  $A$  and  $B$  must be the same, i.e.  $A \leq_c B$  and  $A \leq_{c'} B$  implies  $c = c'$ . We can then write  $T[\mathbb{C}]$  for the system formed by extending  $T$  with coercive subtyping and a coherent set  $\mathbb{C}$  of axiomatic subtyping judgements. A key property of this process is that  $T[\mathbb{C}]$  is a conservative extension of  $T$ : any  $T$ -judgement is derivable in  $T[\mathbb{C}]$  if and only if it is derivable in  $T$ .

Coercive subtyping is a conservative extension of type theory, as stated in the following result, where  $T$  is either the type theory UTT [11] or Martin-Löf's type theory [19]. For example, a corollary of the following theorem is that  $T[\mathbb{C}]$  is logically consistent if  $T$  is.

► **Theorem 1** (Conservativity [15]). *For any coherent set  $\mathbb{C}$  of coercion judgements,  $T[\mathbb{C}]$  is equivalent to a system that is a conservative extension of the type theory  $T$ .*

Extending the type theory UTT [11] with coercive subtyping yields the system  $\text{UTT}[\mathbb{C}]$  (“replacing”  $T$  by  $\text{UTT}$ ), and it is this system that we build on in this paper. Specifically, we extend  $\text{UTT}[\mathbb{C}]$  with additional syntax and rules concerning subtype universes, yielding a system we call  $\text{UTT}[\mathbb{C}]_U$ .

### 2.3 The system $\text{UTT}[\mathbb{C}]_U$

Our system is an extension of UTT [11], although any type theory with a predicative universe hierarchy is suitable. The extension consists of some new syntactic forms and six new typing rules. The syntactic forms are  $U_i(A)$ ,  $\mathbf{T}_{U_i}^A$ ,  $n(A)$  and  $u_i(A)$ .  $U_i(A)$  is a subtype universe parameterised by the type  $A$ .  $\mathbf{T}_{U_i}^A$  is an operator parameterised by the subtype universe  $U_i(A)$ .  $n(A)$  is a meta-level operation which gives the name for the type  $A$ . Similarly,  $u_i(A)$  is a meta-level operation which gives the name for the subtype universe  $U_i(A)$ . These forms are given meaning via six new typing rules. The rules are given in Figure 1. They can be divided into four groups, which we call formation (U-FORM), introduction (U-INTRO), reflection (U-REFL1, U-REFL2) and predicativity (U-PRED1, U-PRED2). Note that these are convenient labels rather than precise categorisations.

The formation rule (U-FORM) introduces a subtype universe  $U_i(A)$  for every type  $A$  which has a name in a traditional universe  $\text{Type}_i$ . We apply a single restriction in the form of the side condition  $\mathcal{L}_\Gamma(A) = i$ , which requires that the type level of  $A$  is equal to  $i$ .

Type levels are explained in Definition 2, but informally the level of a type is the index of the smallest traditional universe in which the type has a name. For example,  $\text{Bool}$  has type level 0 whilst  $\text{Type}_0$  has type level 1. Type levels are important because they allow us to determine “size” of a type. By annotating every subtype universe with a type level, we syntactically expose a lower bound for the type when placing it in the traditional universe hierarchy. For example, we cannot allow the subtype universe  $U_2(\text{Type}_1)$  to have a name in  $\text{Type}_0$ , as  $\text{Type}_1$  is a strictly larger type. We prevent this by ensuring that any subtype universe  $U_i(A)$  has a name only in the traditional universe  $\text{Type}_{i+1}$  (see rules (U-PRED1) and (U-PRED2)).

► **Definition 2** (Type Level). *For any type  $A$  in a context  $\Gamma$ , in  $\text{UTT}[\mathbb{C}]$  or  $\text{UTT}[\mathbb{C}]_U$ , its type level  $\mathcal{L}_\Gamma(A)$  is defined as follows:*

- If  $\exists P. \Gamma \vdash \text{Prf}(P) = A : \text{Type}$ ,  $\mathcal{L}_\Gamma(A) =_{df} -1$
- Otherwise,  $\mathcal{L}_\Gamma(A)$  is the least  $m$  such that  $\exists a. \Gamma \vdash \mathbf{T}_m(a) = A : \text{Type}$

Since every type in  $\text{UTT}[\mathbb{C}]$  and  $\text{UTT}[\mathbb{C}]_U$  has a name in some universe, Definition 2 is well defined for all types.

$\frac{\text{U-FORM}}{\Gamma \vdash \mathbf{T}_i(a) = A : \mathbf{Type}} (\mathcal{L}_\Gamma(A) = i)$	
$\frac{\text{U-INTRO}}{\Gamma \vdash n(B) : U_i(A)} (\mathcal{L}_\Gamma(B) \leq \mathcal{L}_\Gamma(A))$	
$\frac{\text{U-REFL1}}{\Gamma \vdash \mathbf{T}_{U_i}^A(n(B)) : \mathbf{Type}}$	
$\frac{\text{U-REFL2}}{\Gamma \vdash \mathbf{T}_{U_i}^A(n(B)) = B : \mathbf{Type}} (\mathcal{L}_\Gamma(B) \leq \mathcal{L}_\Gamma(A))$	
$\frac{\text{U-PRED1}}{\Gamma \vdash u_i(A) : \mathbf{Type}_{i+1}}$	$\frac{\text{U-PRED2}}{\Gamma \vdash \mathbf{T}_{i+1}(u_i(A)) = U_i(A) : \mathbf{Type}}$

■ **Figure 1** The typing rules for subtype universes. The extension of  $\text{UTT}[\mathbb{C}]$  by these rules forms the system  $\text{UTT}[\mathbb{C}]_U$ .

It is important to note that because proof types (types of the form  $\mathbf{Prf}(P)$  for some proposition  $P$ ) have a defined type level of  $-1$ , we cannot form subtype universes of them. This is because the premiss of (U-FORM) is  $\Gamma \vdash \mathbf{T}_i(a) = A : \mathbf{Type}$ , where  $i = \mathcal{L}_\Gamma(A)$ . There is no operator  $\mathbf{T}_{-1}$  in  $\text{UTT}[\mathbb{C}]$ , and therefore we cannot derive this judgement for proof types. Proof types can have names in other subtype universes, if there exists a corresponding subtyping relation, but the inverse is not possible. Intuitively, proof types are not data types and one usually does not consider subtyping relationships between them. We therefore do not consider subtype universes of a proof type. This decision is a point in the design space and there are alternative options. We discuss some of these in Section 5.

The introduction rule (U-INTRO) defines the membership of subtype universes. If a type  $B$  is a subtype of  $A$ , then its name, given by  $n(B)$ , is an object of the subtype universe of  $A$ . As we shall see, we will be able to convert from  $n(B)$  to  $B$ . In this way we represent the concept that  $B$  is a “member” of  $U_i(A)$ . Again, there is an additional restriction on this rule: the type level of  $B$  must not be greater than the type level of  $A$ . This restriction ensures that we can translate derivations in our system into derivations in  $\text{UTT}[\mathbb{C}]$ , and is critical in proving some meta-theoretic properties, as we will describe shortly.

We now have a connection between subtypes of  $A$  and their corresponding names in  $U_i(A)$ . The reflection rules (U-REFL1) and (U-REFL2) complete the circle by relating the names back to their subtypes. (U-REFL1) introduces an operator  $\mathbf{T}_{U_i}^A$ , which is parameterised by a type  $A$  and its type level  $i$ . For any object  $n(B)$  in  $U_i(A)$ ,  $\mathbf{T}_{U_i}^A(n(B))$  is a type. (U-REFL2) then tells us what type:  $\mathbf{T}_{U_i}^A(n(B))$  is equal to the type  $B$ . This rule has the

same side condition as (U-INTRO), which has no effect on the semantics but simplifies the metatheory. Together, these four rules are effectively a translation of (UNIVERSE-FORM) and (UNIVERSE-INTRO) into the Tarski universe formulation.

Finally, we relate subtype universes to the traditional universe hierarchy. Rules (U-PRED1) and (U-PRED2) state that a subtype universe  $U_i(A)$  has a name in the universe  $Type_{i+1}$ . Placing subtype universes into the traditional universes is a design choice, rather than a necessary construction. We discuss this approach and alternatives in Section 5.

### 3 Applications

Subtype universes have a clear use as a way to model bounded quantification, and this section describes some examples in programming, formalisation and natural language semantics. As subtype universes are first-class types in the system they are inherently more flexible than bounded quantifiers and we expect there are other applications to be discovered.

#### 3.1 Bounded Quantification

With subtype universes we can straightforwardly construct an equivalent to the bounded quantifier  $\Pi A \leq B$ . Continuing our running example, the function *translateX* can be given the following type:

$$translateX : \Pi r : U_i(\langle x : Nat \rangle). Nat \rightarrow \mathbf{T}_{U_i}^{\langle x : Nat \rangle}(r) \rightarrow \mathbf{T}_{U_i}^{\langle x : Nat \rangle}(r)$$

Applying *translateX* to  $\langle x = 1, y = 2 \rangle$  gives a result of type  $\mathbf{T}_{U_i}^{\langle x : Nat \rangle}(n(\langle x : Nat, y : Nat \rangle))$ , which reduces to  $\langle x : Nat, y : Nat \rangle$ . Thus we retain the information that the result is an two-dimensional coordinate.

This kind of extensibility has many applications in programming, where it is useful to be able to deal with *partially specified* data. As a software system is evolved, data is often embellished with new fields. Functions like *translateX* will continue to work as new fields are added to the records it is applied to, allowing for easy and type-safe system extension.

Being a type itself, a subtype universe provides more flexibility than bounded quantification. For example, a subtype universe can appear in both the domain and codomain position of a function type, whereas bounded quantification is only valid in the domain. For example, we can construct types such as  $A \rightarrow U_0(A)$ , which are functions from objects of type  $A$  to subtypes of  $A$ . Another example is the type

$$U_0(A) \rightarrow U_0(B) \rightarrow U_0(\Sigma(A, [x : A]B))$$

Given a subtype of  $A$  and a subtype of  $B$ , a function of this type will return a subtype of their sum,  $\Sigma(A, [x : A]B)$ .

#### 3.2 Extending predicates to subtypes

With coercive subtyping it is straightforward for a predicate  $P : (x : A)Prop$  on some type  $A$  to be extended to all subtypes of  $A$ , since we can always convert objects of a subtype to objects of the supertype. For example, given  $B \leq_c A$  and  $b : B$  then  $P(b)$  becomes  $P(c(b))$  after coercion insertion, which is well typed.

However this fact is not expressed in the type. We rely on meta-level reasoning to know that the domain of  $P$  is implicitly extended to all subtypes of  $A$ . If we rewrite  $P$  to use subtype universes we can better express this property:

$$P : (t : U_i(A))(x : \mathbf{T}_{U_i}^A(t))Prop$$

Here  $P$  is now a predicate on subtypes of  $A$ . Its first argument is the name of a type in the subtype universe for  $A$ , and its second argument is an object of that type, as before. We can apply  $P$  both to  $A$  and its subtypes:  $P(n(A), a)$  and  $P(n(B), b)$  are both well typed, assuming  $a : A$  and  $b : B$ .

### 3.3 Natural language semantics

As well as applications in programming, subtype universes have proved to be useful in formalising the semantics of natural language. In order to describe this application we will first introduce some basic concepts in natural language semantics. Then we will describe how subtype universes can model gradable adjectives.

#### 3.3.1 Montague Grammar

The seminal treatment of natural language semantics is the system developed by Montague in the 1970s [18]. Known as Montague Grammar, this system uses an embedding of higher-order logic in the simply typed lambda calculus to model sentences of natural language. Language constructs are divided into categories: sentences, verb phrases, noun phrases, and common nouns, amongst others. Each category is assigned a type with respect to the two atomic types  $e$  and  $t$ , representing objects and propositions respectively. A complete sentence (e.g. “Socrates is a man”) is regarded as a proposition, and thus has type  $t$ . Verb phrases such as “is a man” form complete propositions when supplied with an object, and therefore have type  $e \rightarrow t$ . Common nouns are interpreted as predicates. For example, the common noun “man” is represented by the function  $\lambda x.man(x)$ . Common nouns therefore have the type  $e \rightarrow t$ . We can also make use of logical operators such as implication ( $\Rightarrow$ ), universal quantification ( $\forall$ ) and existential quantification ( $\exists$ ): the sentence “all men are mortal” can be expressed as  $\forall x. man(x) \Rightarrow mortal(x)$ .

There are two notable downsides to this approach. Firstly, the use of a single type  $e$  for all objects means there is no distinction between different classes of objects, and we can therefore form nonsensical sentences such as “the colour green plays football”. Secondly, the interpretation of common nouns (and indeed, noun phrases) is not intuitive. One would naturally expect common nouns to be interpreted simply as objects.

#### 3.3.2 MTT Semantics

An alternative model of natural language based on Modern Type Theories (MTTs) [21][14] provides a solution to these problems. In MTT semantics, common nouns are interpreted as types in a type theory such as UTT. The interpretation of “man” is as the type  $Man$ , and the sentence “Socrates is a man” is interpreted as  $Socrates : Man$ . We can construct as many types as necessary to precisely describe the context, for example a type  $Man$  representing men and a type  $Human$  representing humans. This naturally leads to problems when a particular object can be seen to inhabit multiple types. For example, both of the judgements  $Socrates : Man$  and  $Socrates : Human$  seem reasonable. To solve this we can apply coercive subtyping. We might define  $Man \leq_c Human$  via some coercion  $c$ , and then we have  $Socrates : Man$  and  $c(Socrates) : Human$ . A full comparison of Montague grammar with MTT semantics is not within the scope of this paper; we refer the reader to [7] for details.

In the context of MTT semantics, subtype universes turn out to be useful in modelling gradable adjectives [16]. Gradable adjectives (words such as “tall”) can be interpreted as predicates which involve comparison of a measurement on the entity with some threshold

$$\begin{aligned}
\delta &: \text{UTT}[\mathbb{C}]_U \rightarrow \text{UTT}[\mathbb{C}] \\
\delta(U_i(A)) &= \text{Type}_i \\
\delta(u_i(A)) &= \text{type}_i \\
\delta(\mathbf{T}_{U_i}^A) &= \mathbf{T}_i \\
\delta(\mathbf{Type}) &= \mathbf{Type} \\
\delta(\text{El}(A)) &= \text{El}(\delta(A)) \\
\delta([x : A]B) &= [x : \delta(A)]\delta(B) \\
\delta((x : A)B) &= (x : \delta(A))\delta(B) \\
\delta(f(x)) &= \delta(f)(\delta(x)) \\
\delta(c) &= c
\end{aligned}$$

■ **Figure 2** The transformation  $\delta$  converts terms in  $\text{UTT}[\mathbb{C}]_U$  to terms in  $\text{UTT}[\mathbb{C}]$ .

value. In the case of “tall”, the measurement is the height of the argument. Furthermore, the threshold often varies based on the type of the argument. The threshold height that is considered tall for a human is very different from the height considered tall for a building. We will describe how “tall” can be interpreted in  $\text{UTT}[\mathbb{C}]_U$ , thereby motivating the use of subtype universes in formal semantics.

We first collect together into a new universe  $T$  all the common nouns for which it makes sense to measure a height.  $T$  may contain, amongst others, (names for) the types *Human* and *Building*. Each of these may have subtypes such as  $\text{Man} \leq \text{Human}$ . We will define a function *height* which measures the height of an object of some type in the universe  $T$ , and a function  $\xi$  which calculates the threshold height for a particular type to be considered tall:

$$\begin{aligned}
\text{height} &: \Pi A : T. \Pi B : U_0(A). B \rightarrow \text{Nat} \\
\xi &: \Pi A : T. U_0(A) \rightarrow \text{Nat}
\end{aligned}$$

Note that *height* and  $\xi$  are defined over all subtypes of all types in  $T$ . For simplicity, we assume that all types in  $T$  have names in the universe  $\text{Type}_0$ . We can then define *tall* as follows:

$$\begin{aligned}
\text{tall} &: \Pi A : T. \Pi B : U_0(A). B \rightarrow \text{Prop} \\
\text{tall}(A, B, x) &= \text{height}(A, B, x) \geq \xi(A, B)
\end{aligned}$$

Compared to other approaches subtype universes provide a simpler semantic construction for gradable adjectives, and may be useful in modelling other linguistic features.

## 4 Metatheory

The system  $\text{UTT}[\mathbb{C}]_U$  retains many of the nice meta-theoretic properties of its base system  $\text{UTT}[\mathbb{C}]$ : logical consistency, strong normalisation and subject reduction. In particular, logical consistency and strong normalisation results can be transferred from  $\text{UTT}[\mathbb{C}]$  because our new rules are admissible in  $\text{UTT}[\mathbb{C}]$  after applying a simple syntactic transformation. This transformation is named  $\delta$ , and it converts terms of  $\text{UTT}[\mathbb{C}]_U$  to terms of  $\text{UTT}[\mathbb{C}]$ , as shown in Figure 2.

For any type  $A$ ,  $\delta$  converts the universe  $U_i(A)$  to  $Type_i$ , which is a valid type in  $UTT[\mathbb{C}]$ . For any object  $A$ ,  $\delta$  converts  $u_i(A)$  to  $type_i$ . For any object  $a$ ,  $\delta$  converts an application of the subtype universe lifting operator  $\mathbf{T}_{U_i}^A(a)$  to an application of the traditional universe lifting operator  $\mathbf{T}_i(a)$ . For any constant  $c$ ,  $\delta$  leaves  $c$  unchanged. The translation is extended to other syntax forms by recursion on their structure, ensuring that the result contains none of the syntax introduced by our extension. Using  $\delta$  we can transform the typing rules in Figure 1, producing rules in the syntax of  $UTT[\mathbb{C}]$ . These are shown in Figure 3.

In this section we will need to refer to derivations in both  $UTT[\mathbb{C}]_U$  and  $UTT[\mathbb{C}]$ . To avoid confusion we will use  $\vdash_U$  for judgements in  $UTT[\mathbb{C}]_U$  and  $\vdash$  for judgements in  $UTT[\mathbb{C}]$ . Contexts in  $UTT[\mathbb{C}]_U$  will be written  $\Gamma$  whereas contexts in  $UTT[\mathbb{C}]$  will typically be of the form  $\delta(\Gamma)$ .

We first note that the side condition  $\mathcal{L}_\Gamma(B) \leq \mathcal{L}_\Gamma(A)$  in (U-INTRO) and (U-REFL2) provides information about the traditional universes in which  $A$  and  $B$  have names.

► **Lemma 3.** *If  $\Gamma \vdash \mathbf{T}_i(a) = A : \mathbf{Type}$ ,  $\Gamma \vdash B : \mathbf{Type}$  and  $\mathcal{L}_\Gamma(B) \leq \mathcal{L}_\Gamma(A)$  then there exists a name  $b : Type_j$  such that  $\Gamma \vdash \mathbf{T}_j(b) = B : \mathbf{Type}$  for some  $j \leq i$ .*

**Proof.** There are six cases to consider:

1.  $\mathcal{L}_\Gamma(B) = \mathcal{L}_\Gamma(A) = -1$
2.  $\mathcal{L}_\Gamma(B) = -1, \mathcal{L}_\Gamma(A) \geq 0$
3.  $\mathcal{L}_\Gamma(B) \geq 0, \mathcal{L}_\Gamma(A) = -1$
4.  $\mathcal{L}_\Gamma(B) = \mathcal{L}_\Gamma(A) \geq 0$
5.  $\mathcal{L}_\Gamma(B) > \mathcal{L}_\Gamma(A) \geq 0$
6.  $\mathcal{L}_\Gamma(B) < \mathcal{L}_\Gamma(A) \geq 0$

Cases 3 and 5 violate the premiss  $\mathcal{L}_\Gamma(B) \leq \mathcal{L}_\Gamma(A)$  and can be discounted.

For cases 1 and 2 we follow the same reasoning: via the definition of type level (Definition 2) there exists some  $P_B$  such that  $\Gamma \vdash \mathbf{Prf}(P_B) = B : \mathbf{Type}$ . Thus we have  $\Gamma \vdash \mathbf{T}_0(\mathbf{t}_0(P_B)) = B : \mathbf{Type}$ . We also have  $i \geq 0$  (otherwise  $\mathbf{T}_i$  would not be defined) and therefore we can choose  $j = 0$  and  $b = \mathbf{t}_0(P_B)$  to satisfy the conclusion.

Cases 4 and 6 also follow the same reasoning: we use the definition of type level to get  $\exists b'. \Gamma \vdash \mathbf{T}_{\mathcal{L}_\Gamma(B)}(b') = B : \mathbf{Type}$ . Choosing  $j = \mathcal{L}_\Gamma(B)$  and  $b = b'$  satisfies the conclusion. ◀

► **Lemma 4.** *The rules in Figure 3 are admissible in  $UTT[\mathbb{C}]$ .*

**Proof.** The conclusions of ( $\delta$ -U-FORM), ( $\delta$ -U-PRED1) and ( $\delta$ -U-PRED2) are all axioms in  $UTT$ , so these rules are trivially admissible. ( $\delta$ -U-REFL1) follows from the definition of  $\mathbf{T}_i$ , which has type  $(Type_i)\mathbf{Type}$ . This leaves ( $\delta$ -U-INTRO) and ( $\delta$ -U-REFL2).

For ( $\delta$ -U-REFL2), we consider the premiss  $\delta(\Gamma) \vdash \mathbf{T}_i(a) = \delta(A) : \mathbf{Type}$  and side condition  $\mathcal{L}_{\delta(\Gamma)}(\delta(B)) \leq \mathcal{L}_{\delta(\Gamma)}(\delta(A))$ . By Lemma 3 we can deduce  $\exists j \leq i. \delta(\Gamma) \vdash \mathbf{T}_j(b) = \delta(B) : \mathbf{Type}$ , where  $b$  is the name for  $\delta(B)$  in  $Type_j$ . Since the traditional universes form a cumulative hierarchy we can conclude  $\delta(\Gamma) \vdash \mathbf{T}_i(b') = \delta(B) : \mathbf{Type}$ , where  $b'$  is the name for  $\delta(B)$  in  $Type_i$ .

For ( $\delta$ -U-INTRO), we follow the same derivation. From  $\delta(\Gamma) \vdash \mathbf{T}_i(b) = \delta(B) : \mathbf{Type}$  we can then conclude  $\delta(\Gamma) \vdash b : Type_i$ . ◀

► **Theorem 5 (Logical consistency).** *There is no proof  $M$  such that  $\Gamma \vdash_U M : \forall X : Prop.X$ .*

**Proof.** By contradiction. Assume there is an  $M$  such that  $\vdash_U M : \forall X : Prop.X$ . Then it follows that  $\vdash \delta(M) : \delta(\forall X : Prop.X)$ , and therefore  $\vdash \delta(M) : \forall X : Prop.X$ . This implies that  $UTT[\mathbb{C}]$  is logically inconsistent, which is false [15]. ◀



$$\begin{array}{c}
\delta\text{-U-FORM} \\
\frac{\delta(\Gamma) \vdash \mathbf{T}_i(a) = \delta(A) : \mathbf{Type}}{\delta(\Gamma) \vdash \mathbf{Type}_i : \mathbf{Type}} \quad (\mathcal{L}_{\delta(\Gamma)}(\delta(A)) = i) \\
\\
\delta\text{-U-INTRO} \\
\frac{\delta(\Gamma) \vdash \mathbf{T}_i(a) = \delta(A) : \mathbf{Type} \quad \delta(\Gamma) \vdash \delta(B) \leq_{\delta(c)} \delta(A) : \mathbf{Type}}{\delta(\Gamma) \vdash n(B) : \mathbf{Type}_i} \quad (\mathcal{L}_{\delta(\Gamma)}(\delta(B)) \leq \mathcal{L}_{\delta(\Gamma)}(\delta(A))) \\
\\
\delta\text{-U-REFL1} \quad \delta\text{-U-REFL2} \\
\frac{\delta(\Gamma) \vdash n(B) : \mathbf{Type}_i}{\delta(\Gamma) \vdash \mathbf{T}_i(n(B)) : \mathbf{Type}} \quad \frac{\delta(\Gamma) \vdash \mathbf{T}_i(a) = \delta(A) : \mathbf{Type} \quad \delta(\Gamma) \vdash \delta(B) \leq_{\delta(c)} \delta(A) : \mathbf{Type}}{\delta(\Gamma) \vdash \mathbf{T}_i(n(B)) = \delta(B) : \mathbf{Type}} \quad (\mathcal{L}_{\delta(\Gamma)}(\delta(B)) \leq \mathcal{L}_{\delta(\Gamma)}(\delta(A))) \\
\\
\delta\text{-U-PRED1} \quad \delta\text{-U-PRED2} \\
\frac{\delta(\Gamma) \vdash \mathbf{Type}_i : \mathbf{Type}}{\delta(\Gamma) \vdash \mathbf{type}_i : \mathbf{Type}_{i+1}} \quad \frac{\delta(\Gamma) \vdash \mathbf{Type}_i : \mathbf{Type}}{\delta(\Gamma) \vdash \mathbf{T}_{i+1}(\mathbf{type}_i) = \mathbf{Type}_i : \mathbf{Type}}
\end{array}$$

■ **Figure 3** The rules in Figure 1 after transformation by  $\delta$ .

► **Definition 6** (Reduction). We write  $M \rightsquigarrow N$  to mean that applying a single step of reduction to the expression  $M$  yields the expression  $N$ . We write  $M \rightsquigarrow^* N$  to mean that applying zero or more steps of reduction to the expression  $M$  yields the expression  $N$ .

► **Lemma 7** ( $\delta$  preserves one-step reduction). For every term  $M$  in  $UTT[\mathbb{C}]_U$ , if  $M \rightsquigarrow N$  then in  $UTT[\mathbb{C}]$  we have  $\delta(M) \rightsquigarrow \delta(N)$ .

**Proof.** See appendix A. ◀

► **Lemma 8** ( $\delta$  preserves multi-step reduction). For every term  $M$  in  $UTT[\mathbb{C}]_U$ , if  $M \rightsquigarrow^* N$  then in  $UTT[\mathbb{C}]$  we have  $\delta(M) \rightsquigarrow^* \delta(N)$ .

**Proof.** There are two cases to consider: a reduction sequence of zero steps and a reduction sequence of one or more steps. For the former case we have  $M = N$  and therefore it follows that  $\delta(M) = \delta(N)$  and hence  $\delta(M) \rightsquigarrow^* \delta(N)$ . The latter case follows from Lemma 7 by induction. ◀

► **Theorem 9** (Strong Normalisation). If  $\Gamma \vdash_U M : A$  then  $M$  is strongly normalisable. In other words, every sequence of reductions starting from  $M$  is finite.

**Proof.** Assume that Strong Normalisation does not hold for  $UTT[\mathbb{C}]_U$  - i.e. there exists a term  $M$  in  $UTT[\mathbb{C}]_U$  with an infinite reduction sequence. By Lemma 8 it follows that there is a corresponding infinite reduction sequence in  $UTT[\mathbb{C}]$  for  $\delta(M)$ . This is a contradiction because Strong Normalisation holds for  $UTT[\mathbb{C}]$ , as it is a conservative extension of  $UTT$  [15]. ◀

► **Theorem 10** (Subject Reduction). *If  $\Gamma \vdash_U M : A$  and  $M \rightsquigarrow N$ , then  $\Gamma \vdash_U N : A$ .*

**Proof.**  $\text{UTT}[\mathbb{C}]_U$  is an extension of  $\text{UTT}[\mathbb{C}]$ , for which subject reduction holds. Therefore it is sufficient to show that the new syntax forms and rules we have introduced preserve this property. We proceed by induction on the structure of terms. Most cases follow straightforwardly by the induction hypothesis, except for the two base cases induced by the reduction rules *urefl* and *upred*, derived from the typing rules (U-REFL2) and (U-PRED2). Given the premises of each rule we can derive identical types for either side of the equality in the conclusion. For *urefl* we have  $M \equiv \mathbf{T}_{U_i}^A(n(B))$ ,  $N \equiv B$  and  $M \rightsquigarrow^{\text{urefl}} N$ . From the premises of (U-REFL2) we can then derive  $\Gamma \vdash_U M : \mathbf{Type}$  and  $\Gamma \vdash \_UN : \mathbf{Type}$ :

$$\frac{\Gamma \vdash_U \mathbf{T}_i(a) = A : \mathbf{Type} \quad \Gamma \vdash_U B \leq_c A : \mathbf{Type}}{\Gamma \vdash_U n(B) : U_i(A)} \text{ (U-INTRO)}$$

$$\frac{\Gamma \vdash_U n(B) : U_i(A)}{\Gamma \vdash_U \mathbf{T}_{U_i}^A(n(B)) : \mathbf{Type}} \text{ (U-REFL1)}$$

$$\frac{\Gamma \vdash_U \mathbf{T}_{U_i}^A(n(B)) : \mathbf{Type}}{\Gamma \vdash_U M : \mathbf{Type}} \text{ (definition of } M \text{)}$$

$$\frac{\Gamma \vdash_U A \leq_c B : \mathbf{Type}}{\Gamma \vdash_U B : \mathbf{Type}} \text{ (definition of } N \text{)}$$

$$\frac{\Gamma \vdash_U B : \mathbf{Type}}{\Gamma \vdash_U N : \mathbf{Type}} \text{ (definition of } N \text{)}$$

For *upred* we have  $M \equiv \mathbf{T}_{i+1}(u_i(A))$ ,  $N \equiv U_i(A)$  and  $M \rightsquigarrow^{\text{upred}} N$ . From the premiss of (U-PRED2) we already have  $\Gamma \vdash_U N : \mathbf{Type}$ . We can then derive  $\Gamma \vdash_U M : \mathbf{Type}$  as follows:

$$\frac{\Gamma \vdash_U U_i(A) : \mathbf{Type}}{\Gamma \vdash_U u_i(A) : \text{Type}_{i+1}} \text{ (U-PRED1)}$$

$$\frac{\Gamma \vdash_U u_i(A) : \text{Type}_{i+1}}{\Gamma \vdash_U \mathbf{T}_{i+1}(u_i(A)) : \mathbf{Type}} \text{ (definition of } \mathbf{T}_{i+1} \text{)}$$

$$\frac{\Gamma \vdash_U \mathbf{T}_{i+1}(u_i(A)) : \mathbf{Type}}{\Gamma \vdash_U M : \mathbf{Type}} \text{ (definition of } M \text{)}$$

The four new syntactic forms,  $U_i(A)$ ,  $u_i(A)$ ,  $n(B)$  and  $\mathbf{T}_{U_i}^A(n(B))$ , are irreducible under the existing reduction rules of  $\text{UTT}[\mathbb{C}]$  and so do not affect the subject reduction property of the original rules. Therefore subject reduction holds for  $\text{UTT}[\mathbb{C}]_U$ . ◀

## 5 Discussion on design choices

Whilst most aspects of our system follow directly from the pseudo rules given in Section 2 or from metatheoretic constraints (for example, the annotation of subtype universes with type levels), some parts reflect specific design choices that could be modified. These areas concern how subtype universes interact with other universes, such as the traditional predicative universes  $\text{Type}_i$ , the impredicative universe *Prop* and even other subtype universes. We might ask if a particular subtype universe can have a name in the universe  $\text{Type}_1$ , or vice versa. Or even: can we construct subtype universes containing names for other subtype universes? Whilst these questions are formally interesting, we have not identified any clear applications of alternative designs.

Firstly, our rules permit universes such as  $U_1(\text{Type}_0)$ , meaning that judgements like  $\text{Type}_0 : U_1(\text{Type}_0)$  are derivable. In the opposite direction, the rules (U-PRED1) and (U-PRED2) define that  $U_i(A) : \text{Type}_{i+1}$ . However, these two rules are optional. We include them so that the traditional universes continue to allow quantification over “all types”, but the system remains standing if they are removed. There are three main options for this point in the design.

The first option is that subtype universes have names in the same traditional universe as the type they are parameterised by - i.e.  $U_i(A) : Type_i$ . It is straightforward to show that this option leads to inconsistency when translated to UTT[C]. Applying  $\delta$  to the judgement  $\Gamma \vdash U_i(A) : Type_i$  yields  $\delta(\Gamma) \vdash \delta(U_i(A)) : \delta(Type_i)$ , which simplifies to  $\delta(\Gamma) \vdash Type_i : Type_i$ .

The second option is that subtype universes are contained in the traditional universe directly “above” the universe of the type they are parameterised by:  $U_i(A) : Type_{i+1}$ . So for some base type  $T : Type_0$ ,  $U_0(T) : Type_1$ . When translated to UTT this judgement becomes (in the general case)  $\Gamma \vdash Type_i : Type_{i+1}$  which is a derivable judgement in UTT. This, therefore, seems the most natural option for the relation between subtype universes and traditional universes, not least because  $Type_{i+1}$  is the smallest universe in which we can place  $U_i(A)$  without encountering paradoxes.

The third option is not to include subtype universes as objects in the traditional universe hierarchy. This is equivalent to removing the rules (U-PRED1) and (U-PRED2). The resulting system is still admissible in UTT, and therefore retains the desired meta-theoretic properties. However this weakens the traditional universes, because there are now types that they cannot capture (the subtype universes).

Subtype universes can be formed from any type in our system, with the except of proof types of propositions, i.e. types of the form  $\mathbf{Prf}(P)$ . We have made this decision to simplify the typing rules, but there is also an intuitive argument that subtyping between proof types is not desirable. It is important to note that whilst we cannot construct subtype universes of proof types, these types can still have names in the subtype universes of other types. Subtype universes of proof types can be supported by a small modification to the system: removing the first clause from Definition 2. This has the auxiliary effect of making the system independent of  $Prop$  entirely. We have not fully explored the relationship between subtype universes and the impredicative universe  $Prop$  (and its related types); further work is needed in this area.

Subtyping as a relation has the property of transitivity:

$$\frac{\Gamma \vdash T \leq_c A : \mathbf{Type} \quad \Gamma \vdash A \leq_{c'} B : \mathbf{Type}}{\Gamma \vdash T \leq_{c' \circ c} C : \mathbf{Type}}$$

By analogy with set theory, we might expect there to be a corresponding subtyping relation between the subtype universes of  $A$  and  $B$ . The (informal) reasoning for this is as follows: every type with a name in  $U_i(A)$  is a subtype of  $A$ , and therefore by transitivity it is also a subtype of  $B$ , and must have a name in  $U_j(B)$ . To formalise this argument we must find a coercion between names of  $T$  in  $U_i(A)$  and names of  $T$  in  $U_j(B)$ . This is not generally derivable in our system as-is, but we can support it by the addition of the following rule:

$$\text{U-EQUIV} \quad \frac{\Gamma \vdash A \leq_c B : \mathbf{Type} \quad \Gamma \vdash \mathbf{T}_{U_i}^A(t) = T : \mathbf{Type} \quad \Gamma \vdash \mathbf{T}_{U_j}^B(t') = T : \mathbf{Type}}{\Gamma \vdash \mathbf{t}_{i,j}^{A,B}(t) = t' : U_j(B)} \quad (\mathcal{L}_\Gamma(A) \leq \mathcal{L}_\Gamma(B))$$

Here we introduce a lifting operator  $\mathbf{t}_{i,j}^{A,B}$  which takes a name in the universe  $U_i(A)$  to a name of the same type in  $U_j(B)$ . This operator can act as a coercion between universes, allowing us to state the following subtype relation:

$$\Gamma \vdash U_i(A) \leq_{\mathbf{t}_{i,j}^{A,B}} U_j(B)$$

By (U-PRED2) we have  $\mathcal{L}_\Gamma(U_j(B)) = j + 1$ , and therefore we can derive

$$\begin{aligned} & \Gamma \vdash U_{j+1}(U_j(B)) : \mathbf{Type} \\ \Gamma \vdash \mathbf{T}_{U_j}^B(n(U_i(A))) = U_i(A) : \mathbf{Type} \end{aligned}$$

## 6 Conclusion

Subtype universes are a novel and useful construct, providing amongst other things a decidable alternative to bounded quantification. We have seen that a coercive subtyping system can be extended to support subtype universes with the addition of six typing rules. Our implementation builds on UTT[C], an existing system supporting coercive subtyping. We prove logical consistency, strong normalisation and subject reduction for our system.

We have extended UTT[C] with subtype universes in a way that preserves its nice metatheoretic properties. In this process we rely on the existence of the predicative universes  $Type_i$  in order to syntactically convert subtype universes into predicative universes. However it is important to note that this is done only to make the metatheoretic proofs straightforward. An early abstract of this paper [16] specified a simpler system where subtype universes were not annotated with a type level. This formulation was entirely independent of UTT’s predicative universes, but proving the admissibility of the typing rules was difficult. As a result, it is possible that subtype universes can be formulated without predicative universes; this would be an interesting subject for further work.

Although we believe that the system  $UTT[C]_U$  has the Church-Rosser property, we have not succeeded in proving it. We also leave unanswered the question of exactly why a system with subtype universes enjoys decidable typechecking whilst traditional bounded quantification does not. Notably absent in our system is the maximal type  $Top$  of which every other type is a subtype. Indeed it is not clear how one could introduce such a type in a system with a predicative universe hierarchy, for the same reason that we cannot introduce a type of all types. This distinction is worthy of further investigation.

**Related Work.** Subtype universes bear similarities to Cardelli’s power type [3]  $Power(A)$ , a type containing all subtypes of  $A$ . Power types are constructed in the context of a system of structural subtyping, where subtyping relations are determined by the structure of types rather than by arbitrary axioms. This is motivated by the desire for types to be “self-describing”, simplifying typechecking and enabling features like the type-safe (de)serialisation. There is no distinct subtyping relation; the judgement  $\vdash A : Power(B)$  reads “ $A$  is a subtype of  $B$ ” and is abbreviated  $\vdash A \leq B$ . A notable typing rule is that of Power Subtyping, written:

$$\frac{\Gamma \vdash B \leq A}{\Gamma \vdash Power(B) \leq Power(A)}$$

This rule states that if  $B$  is a subtype of  $A$  then the power type of  $B$  is a subtype of the power type of  $A$ . The conclusion can also be written  $\Gamma \vdash Power(B) : Power(Power(A))$ . Written this way, it is clear that this has the same effect as the rule (U-EQUIV) described in Section 5.

Another important aspect of Cardelli’s system is that it includes the axiom  $Type : Type$ , well known to be logically inconsistent. A full discussion of the problems with this property is beyond the scope of this paper; we merely note that, as an extension of UTT, our system does not share this property.

Aspinall’s  $\lambda_{Power}$  [1] is a predicative and simplified alternative to Cardelli’s system, but it has been difficult to prove some of its metatheoretic properties (such as subject reduction).

## References

- 1 David Aspinall. Subtyping with power types. In *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000 Proceedings*, volume 1862 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2000. doi:10.1007/3-540-44622-2\_10.
- 2 David Aspinall and Adriana B. Compagnoni. Subtyping dependent types (summary). In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 86–97. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561307.
- 3 Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 70–79. ACM Press, 1988. doi:10.1145/73560.73566.
- 4 Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Inf. Comput.*, 109(1-2):4–56, 1994. doi:10.1006/inco.1994.1013.
- 5 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.
- 6 Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 151–162. ACM Press, 1994. doi:10.1145/174675.177844.
- 7 Stergios Chatzikyriakidis and Zhaohui Luo. On the interpretation of common nouns: Types versus predicates. In *Modern perspectives in type-theoretical semantics*, pages 43–70. Springer, 2017.
- 8 Thierry Coquand. An analysis of girard's paradox. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 227–236. IEEE Computer Society, 1986.
- 9 Jean-Yves Girard. *Interprétation fonctionnelle et Élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.
- 10 Healfdene Goguen. The metatheory of UTT. In *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 60–82. Springer, 1994. doi:10.1007/3-540-60579-7\_4.
- 11 Zhaohui Luo. *Computation and reasoning - a type theory for computer science*, volume 11 of *International series of monographs on computer science*. Oxford University Press, 1994.
- 12 Zhaohui Luo. Coercive subtyping. *J. Log. Comput.*, 9(1):105–130, 1999. doi:10.1093/logcom/9.1.105.
- 13 Zhaohui Luo. Dependent record types revisited. In *Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants, MLPA '09*, page 30–37, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1735813.1735819.
- 14 Zhaohui Luo. Formal semantics in modern type theories with coercive subtyping. *Linguistics and Philosophy*, 35(6):491–513, 2012.
- 15 Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Inf. Comput.*, 223:18–42, 2013. doi:10.1016/j.ic.2012.10.020.
- 16 Harry Maclean and Zhaohui Luo. Subtype universes (extended abstract). *26th International Conference on Types for Proofs and Programs (TYPES20)*, 2020.
- 17 Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.
- 18 Richard Montague. *Formal Philosophy: Selected Papers of Richard Montague*. Springer, 1974.
- 19 Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory. An introduction*. Oxford University Press, 1990.
- 20 Benjamin C. Pierce. Bounded quantification is undecidable. *Inf. Comput.*, 112(1):131–165, 1994. doi:10.1006/inco.1994.1055.
- 21 Aarne Ranta. *Type-Theoretical Grammar*. Oxford University Press, 1994.

- 22 John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. doi:10.1007/3-540-06859-7\_148.
- 23 Sergei G. Vorobyov. Structural decidable extensions of bounded quantification. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95*, page 164–175, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/199448.199479.

## A Proof of Lemma 7

► **Lemma** ( $\delta$  preserves one-step reduction). For every term  $M$  in  $\text{UTT}[\mathbb{C}]_U$ , if  $M \rightsquigarrow N$  then in  $\text{UTT}[\mathbb{C}]$  we have  $\delta(M) \rightsquigarrow \delta(N)$ .

**Proof.** By induction on the terms of  $\text{UTT}[\mathbb{C}]_U$ . For each reduction step  $M \rightsquigarrow^R N$  in  $\text{UTT}[\mathbb{C}]_U$  via a computation rule  $R$  we will show that there is a reduction  $\delta(M) \rightsquigarrow^S \delta(N)$  in  $\text{UTT}[\mathbb{C}]$  via a (possibly identical) rule  $S$ . In the special case where  $\delta(M) = M$ , we will show that  $\delta(N) = N$ . There are eight reduction (or computation) rules in  $\text{UTT}$  [10]:

1.  $([x : K]k')k \rightsquigarrow^\beta [k/x]k'$
2.  $E_\forall(A, P, R, f, \Lambda(A, P, g)) \rightsquigarrow^{E_\forall} f(g)$
3.  $\mathbf{T}_{i+1}(\text{type}_i) \rightsquigarrow^{\text{type}_i} \text{Type}_i$
4.  $\mathbf{T}_0(\text{prop}) \rightsquigarrow^{\text{prop}} \text{Prop}$
5.  $\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) \rightsquigarrow^{\mathbf{t}_{i+1}} \mathbf{T}_i(a)$
6.  $\mathbf{T}_0(\mathbf{t}_0(P)) \rightsquigarrow^{\text{prf}} \mathbf{Prf}(P)$
7. the computation rule for inductive types  $\mathbf{E}[\bar{\Theta}]$
8.  $\mathbf{T}_i(\mu_i[\bar{\Theta}]) \rightsquigarrow^\mu \mathcal{M}[\bar{\Theta}]$

Our extension adds two more:

1.  $\mathbf{T}_{U_i}^A(n(B)) \rightsquigarrow^{\text{urefl}} B$
2.  $\mathbf{T}_{i+1}(u_i(A)) \rightsquigarrow^{\text{upred}} U_i(A)$

We will consider the last two rules in detail. The others follow straightforwardly from the definition of  $\delta$ . By induction we can therefore extend the result to all expressions in  $\text{UTT}[\mathbb{C}]_U$ .

► **Case 1** ( $M \rightsquigarrow^{\text{urefl}} N$ ). This rule eliminates  $n(B)$ , the name for a type  $B$  in the subtype universe  $U_i(A)$ .

$$\begin{aligned} \mathbf{T}_{U_i}^A(n(B)) &\rightsquigarrow^{\text{urefl}} B \\ \delta(\mathbf{T}_{U_i}^A(n(B))) &= \delta(\mathbf{T}_{U_i}^A(\delta((n(B)))))) \\ &= \mathbf{T}_i(n(\delta(B))) \\ &\rightsquigarrow^X \delta(B) \end{aligned}$$

where  $X$  stands for the relevant reduction rule reflecting a name in a traditional universe to its type. For example, if  $B$  is an inductive type then  $X$  stands for the rule  $\rightsquigarrow^\mu$ .

► **Case 2** ( $M \rightsquigarrow^{\text{upred}} N$ ). This rule eliminates  $u_i(A)$ , the name for the subtype universe  $U_i(A)$ .

$$\begin{aligned} \mathbf{T}_{i+1}(u_i(A)) &\rightsquigarrow^{\text{upred}} U_i(A) \\ \delta(\mathbf{T}_{i+1}(u_i(A))) &= \delta(\mathbf{T}_{i+1}(\delta((u_i(A)))))) \\ &= \mathbf{T}_{i+1}(\text{type}_i) \\ &\rightsquigarrow^{\text{type}_i} \text{Type}_i \\ &= \delta(U_i(A)) \end{aligned}$$

◀

# Two Applications of Logic Programming to Coq

Matteo Manighetti ✉


Inria, Palaiseau, France

LIX, Ecole Polytechnique, Palaiseau, France

Dale Miller ✉ 🏠

Inria, Palaiseau, France

LIX, Ecole Polytechnique, Palaiseau, France

Alberto Momigliano ✉ 🏠 

Dipartimento di Informatica, University of Milan, Italy

---

## Abstract

The logic programming paradigm provides a flexible setting for representing, manipulating, checking, and elaborating proof structures. This is particularly true when the logic programming language allows for bindings in terms and proofs. In this paper, we make use of two recent innovations at the intersection of logic programming and proof checking. One of these is the *foundational proof certificate* (FPC) framework which provides a flexible means of defining the semantics of a range of proof structures for classical and intuitionistic logic. A second innovation is the recently released Coq-Elpi plugin for Coq in which the Elpi implementation of  $\lambda$ Prolog can send and retrieve information to and from the Coq kernel. We illustrate the use of both this Coq plugin and FPCs with two example applications. First, we implement an FPC-driven sequent calculus for a fragment of the Calculus of Inductive Constructions and we package it into a tactic to perform property-based testing of inductive types corresponding to Horn clauses. Second, we implement in Elpi a proof checker for first-order intuitionistic logic and demonstrate how proof certificates can be supplied by external (to Coq) provers and then elaborated into the fully detailed proof terms that can be checked by the Coq kernel.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Logic and verification

**Keywords and phrases** Proof assistants, logic programming, Coq,  $\lambda$ Prolog, property-based testing

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.10

**Supplementary Material** *Software (Source Code)*: <https://github.com/proofcert/fpc-elpi>  
archived at `swh:1:dir:2da6ff73379af393bef8a3a3a6419d07906af713`

**Acknowledgements** We thank Enrico Tassi for his help with Coq-Elpi. His comments and those of the anonymous reviewers on an early draft of this paper have also been very helpful.

## 1 Introduction

Recently, Enrico Tassi et al. developed the Elpi implementation [21] of  $\lambda$ Prolog [47], and more recently, Tassi has made Elpi available as the Coq-Elpi plugin [62] (<https://github.com/LPCIC/coq-elpi>) to the Coq proof assistant. This implementation of  $\lambda$ Prolog extends earlier ones in primarily two directions: First, Elpi adds a notion of constraints and constraints handling rules, which makes it more expressive than the Teyjus implementation [51] of  $\lambda$ Prolog. Second, the plugin version of Elpi comes equipped with a quotation and anti-quotation syntax for mixing Coq expressions with  $\lambda$ Prolog program elements and an API for accessing the Coq environment, including its type checker.

The logic programming interpreter that underlies Elpi provides several convenient features for the kind of meta-programming tasks that can arise within modern proof assistants. For example,  $\lambda$ Prolog provides a declarative and direct treatment of abstract syntax that contains bindings, including capture-avoiding substitution, unification, and recursive programming. Elpi spares the programmer from dealing with low-level aspects of the representation of binders (e.g., De Bruijn indexes) while still having clean and effective ways to manipulate binding structures.



© Matteo Manighetti, Dale Miller, and Alberto Momigliano;  
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 10; pp. 10:1–10:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



## 10:2 Two Applications of Logic Programming to Coq

Since relations (not functions) are central in  $\lambda$ Prolog, Elpi is capable of providing direct support for the many relations that have a role in implementation and usage of proof assistants. Such relations include typing (e.g.,  $\Gamma \vdash M : \sigma$ ), evaluation (e.g., natural semantics specifications [37, 33]), and interaction (e.g., structured operational semantics [49, 57]).

Felty has also made the point that LCF-style tactics and tacticals can be given an elegant and natural specification using the higher-order relational specifications provided by  $\lambda$ Prolog [23]. Some recent implementations built using Elpi support the usefulness of higher-order logic programming as a meta-programming language for proof assistants in general [20, 32] and, in particular, for Coq via the Coq-Elpi plugin [18, 63].

In this paper, we present two applications of Elpi within Coq. With these applications, we shall illustrate that Elpi is useful not only because of its meta-programming features but also because it soundly implements a higher-order intuitionistic logic: such implementations of higher-order logic have long been known to provide powerful and flexible approaches to implementing many different logics and their proof systems [24, 53]. Following that tradition, the Elpi system makes it possible to encode the proofs and proof theory of various subsets of the logic behind Coq (see also [22]).

While other meta-programming frameworks based on functional programming such as *MetaCoq* [61] can and have been used for related endeavors, we believe (together with [20]) that they would require much more boilerplate code.

Before we can present these examples, we first highlight the rather striking differences in notions of computing and reasoning that arise on each side of the Coq-Elpi API. We will also present a quick summary of the key proof theory concepts that are used by our example applications.

### 2 Two cultures

When studying structural proof theory, one learns quickly that many concepts come in pairs: negative/positive, left/right, bottom-up/top-down, premises/conclusion, introduction/elimination, etc. When we examine the larger setting of this project of linking a logic programming engine with Coq and its kernel, we find a large number of new pairings that are valuable to explicitly discuss.

#### 2.1 Proof theory vs type theory

In many ways, proof theory is more elementary and low-level than most approaches to type theory. For example, type theories usually answer the question “What is a proof?” with the response “a (dependently) typed  $\lambda$ -term”. That is, when describing a type theory, one usually decides that a proof is a certain kind of  $\lambda$ -term within the system. In contrast, proof theory treats logical propositions and proofs as separate. For example, proof theory does not assume that there are terms within the logic that describe proofs.

Gentzen’s discovery that the key to treating classical and intuitionistic logics in the same proof system was identifying the structural rules of the weakening and contraction and placing them on the right-side of sequents [29]. This discovery led him away from natural deduction to multiple-conclusion sequent calculus. This same innovation of Gentzen’s also opened the way to another key proof-theoretic discovery, that of linear logic and linear negation [30]. While sequent calculus provides an elegant presentation of full linear logic, most treatments of linear logic in type theory have been limited to single-conclusion sequents [5] or restricted, multiple-conclusion variants [36].



As is often observed, however, sequent calculus is too low-level to be used explicitly as capturing the “essence of a proof”. Fortunately, the notion of *focused proof systems* [1, 42], makes it possible to collect and join the micro-level inference rules of the sequent calculus into large-scale, synthetic inference rules. As a result, using such proof systems, it is possible to extract from sequent calculus not only natural deduction proofs [56], but also proof nets [13], and Herbrand-style expansion trees [12]. The role of focused proof systems to characterize classes of proof structures is described in the next section and used in our two example applications of Elpi with Coq.

## 2.2 Proof search vs proof normalization

Gentzen-style proofs are used to model computation in at least two different ways. The functional programming paradigm, following the Curry-Howard correspondence, models computation abstractly as  $\beta$ -reduction within natural deduction proofs: that is, computation is modeled as *proof normalization*. On the other hand, the logic programming paradigm, following the notion of goal-directed proof search, models computation abstractly as a regimented search for cut-free sequent calculus proofs: that is, computation is modeled as *proof search*.

Proof search has features that are hard to capture in the proof normalization setting. In particular, Gentzen’s *eigenvariables* are a kind of proof-level binder. In the proof normalization setting, such eigenvariables are instantiated during proof normalization. However, during the search for cut-free proofs, eigenvariables do not vary, and they are part of the syntax of terms and formulas. As a result, they can be used in the treatment of bindings in data structures more generally. Such eigenvariables can be used by the logic programming paradigm to provide a natural and powerful approach to computing with bindings within syntax.

It is worth noting that the role of the cut rule and cut-elimination is different in these two approaches to computing. In the proof normalization paradigm, the cut rule can be used to model  $\beta$  reduction, especially via the explicit substitution approach [40]. In the proof search paradigm, since computing involves the search for cut-free proofs, the cut rule plays no part in the performance of computation. However, cut and cut-elimination can be used to reason about computation: for example, cut-elimination can be used to prove “substitution lemmas for free” that arise in the study of operational semantics [28].

## 2.3 $\lambda$ Prolog vs Coq

Given that  $\lambda$ Prolog and Coq both result from combining the  $\lambda$ -calculus with logic, it is important to understand some of their differences. The confusion around the term *higher-order abstract syntax* (HOAS), is a case in point. In the functional programming setting, including the Coq system, the HOAS approach leads to encoding binding structures within terms using functions. The earliest such encodings were unsatisfactory since they would allow for *exotic terms* [19] and for structures on which induction was not immediately possible [59]. Later approaches yielded non-canonical and complex encodings [17, 35], as well as sophisticated type theories [55]. All of these could support inductive and coinductive reasoning. In the logic programming setting, particularly  $\lambda$ Prolog, HOAS is well supported since bindings are allowed with terms ( $\lambda$ -bindings), formulas (quantifiers), and proofs (eigenvariables). (In fact, the original paper on HOAS [54] was inspired by  $\lambda$ Prolog.) For this reason, the term  *$\lambda$ -tree syntax* was introduced to name this particular take on HOAS [46]. The Abella proof assistant [3] was designed, in part, to provide inductive and co-inductive inference involving specifications using the  $\lambda$ -tree syntax approach.

Another difference between  $\lambda$ Prolog and functional programming can be illustrated by considering how they are used in the specification of tactics. The origin story for the ML functional programming language was that it was the meta-language for implementing the LCF suite of tactics and tacticals [31]. To implement tactics, ML adopted not only novel features such as polymorphically typed higher-order functional programming but also the non-functional mechanisms of failure and exception handling. While  $\lambda$ Prolog is also based on ML-style polymorphically typed higher-order relational programming, it also comes with a completely declarative version of failure and backtracking search. Combining those features along with its support of unification (even in the presence of term-level bindings),  $\lambda$ Prolog provides a rather different approach to the specification of tactics [23].

### 3 Proof theory and proof certificates

In this section, we introduce the main ideas from *focused proof systems*, *foundational proof certificates*, and the *Coq-Elpi* plugin that we need for this paper.

#### 3.1 Proofs for the Horn fragment

A *Horn clause* is a formula of the form  $\forall \bar{x}_1. A_1 \supset \forall \bar{x}_2. A_2 \supset \forall \bar{x}_n. A_n \supset A_0$  where  $\forall \bar{x}_i$  denote a list of universal quantifiers ( $i \in \{1, \dots, n\}$ ) and  $A_0, \dots, A_n$  are atomic formulas. It is well known that the following set of sequent calculus proof rules are complete for both classical and intuitionistic logic when one is attempting to prove that a given atomic formula  $A$  is provable from a set  $\mathcal{P}$  of Horn clauses.

$$\frac{D \in \mathcal{P} \quad \mathcal{P} \Downarrow D \vdash A}{\mathcal{P} \vdash A} \text{decide} \quad \frac{}{\mathcal{P} \Downarrow A \vdash A} \text{init}$$

$$\frac{\mathcal{P} \Downarrow D[t/x] \vdash A}{\mathcal{P} \Downarrow \forall x. D \vdash A} \forall L \quad \frac{\mathcal{P} \vdash B \quad \mathcal{P} \Downarrow D \vdash A}{\mathcal{P} \Downarrow B \supset D \vdash A} \supset L$$

Here, we use two different styles of sequents. The sequent  $\mathcal{P} \vdash A$  is the usual sequent which we generally use as the end sequent (the conclusion) of a proof. The sequent  $\mathcal{P} \Downarrow D \vdash A$  is a *focused* sequent in which the formula  $D$  is the *focus* of the sequent. The two left introduction rules and the initial rule can only be applied to the focused formula. This latter point is in contrast to Gentzen's sequent calculus where these rules can involve *any* formula on the left of the  $\vdash$ . The fact that this proof system is complete for both classical and intuitionistic logic (when restricted to the Horn clause fragment) follows from rather simple considerations of Horn clauses [50] and from the completeness of uniform proofs [48] or LJT proofs [34]. The use of the term *focus* comes from Andreoli's proof system for linear logic [1].

Figure 1 contains an annotated version of these proof rules: the annotations help us connect to elements of the Coq proof system.

1. Instead of having separate connectives for  $\forall$  and  $\supset$ , we have the dependent product connective  $(x : A)D$ .
2. We account for *computation* inside atoms by generalizing the *init* rule to allow type-level conversion.
3. We have incorporated *proof certificates* [16] (using the schematic variable  $\Xi$ ) along with *expert predicates* (predicates with the  $e$  subscript). We explain these in Section 3.2.
4. The inference rules are annotated by terms structures that can be given directly to the Coq kernel for checking.
5. We have added various premises which are responsible for interacting with Coq.

$$\begin{array}{c}
\frac{\text{Ind}[p] (\Gamma_I := \Gamma_C) \in E \quad (\text{head } A) : T \in \Gamma_I \quad k : D \in \Gamma_C \quad \Xi_1 \Downarrow l : D \vdash A \quad \text{decide}_e(\Xi, \Xi_1)}{\Xi \vdash k \ l : A} \\
\\
\frac{E[] \vdash_{CIC} A : s \quad \text{sort}_e(\Xi,)}{\Xi \vdash A : s} \quad \frac{E[] \vdash_{CIC} A =_{\beta\delta\iota\xi} A' \quad \text{initial}_e(\Xi)}{\Xi \Downarrow [] : A \vdash A'} \\
\\
\frac{\Xi_1 \vdash t : B \quad \Xi_2 \Downarrow l : D[t/x] \vdash A \quad \text{prod}_e(\Xi, \Xi_1, \Xi_2, t)}{\Xi \Downarrow (t :: l) : (x : B)D \vdash A}
\end{array}$$

■ **Figure 1** Specification of a core calculus.

The proof system described in Figure 1 (and implemented in Figure 2) corresponds to a subset of the Calculus of Inductive Constructions in which inductive definitions are limited to Horn clauses. This system is inspired by the calculus for proof search in Pure Type Systems introduced in [41], based in turn on ideas stemming from focusing (in particular, uniform proofs [48] and the LJT calculus [34]). Similar to that calculus, we have a term language that includes terms and lists of terms, and two typing judgments for the two categories. This style of proof terms coincides with the idea behind the *spine calculus* [11]. The main novelty of our proof system here is that proof terms and proof certificates are used simultaneously in all inference rules.

The proof system is parameterized by Coq’s global environment  $E$ , here a set of constant and inductive definitions; following Coq’s reference manual, inductive definition are denoted by  $\text{Ind}[p](\Gamma_I := \Gamma_C)$ , where  $\Gamma_I$  determines the names and types of the (possibly mutually) inductive type and  $\Gamma_C$  the names and types of its constructors; finally  $p$  denotes the number of parameters and plays here no role. The local context is empty, since we are only dealing with types that correspond to Horn clauses, and atomic types are inductively defined. In fact, we do not have a  $\forall$  rule on the right, although the proof theory would gladly allow it. This means that there are no bound variables in our grammar of terms. Terms are always applied to a (possibly empty) list of arguments. We delegate to Coq’s type checking the enforcement of the well-sortedness of inductive types. The *decide* rule, as in the previous proof system for Horn logic, given an atom, selects a clause on which to backchain on: we lookup the constructors of an inductive definition from the global environment, one that matches the head symbol of the atom we aim to backchain on, and then call the latter judgment that will find a correct instantiation, if any. The rules for backchaining include the (conflation of the) left introduction rules for  $\forall$  and  $\supset$ , as well as the *init* rule, which incorporates Coq’s conversion.

It may be at first surprising that there are no introduction rules for propositional connectives, nor equality for that matter. However, one of the beauties of the Calculus of Inductive Construction is that they are, in fact, defined inductively and therefore the *decide* rule will handle those. Thus, the syntax of proof-terms is rather simple.

### 3.2 Proof certificate checking

Figure 2 contains the Elpi implementation of the inference rules in Figure 1:  $\Xi \vdash t : A$  corresponds to `check Cert (go A T)` and  $\Xi \Downarrow l : D \vdash A$  corresponds to `check Cert (bc D A L)`. The code in that figure mixes both Coq-specific and FPC-specific items. We describe both of these separately.

## 10:6 Two Applications of Logic Programming to Coq

```

kind goal      type.
type go        term → term → goal.
type bc        term → term → list term → goal.
type check     cert → goal → o.

```

```

check Cert (go (sort S) A):-
  sortE Cert,
  coq.typecheck A (sort S) ok.
check Cert (go A Tm) :-
  coq.safe-dest-app A (global (indt Prog)) _,
  coq.env.indt Prog _ _ _ Kn KTypes,
  decideE Kn Cert Cert' K,
  std.zip Kn KTypes Clauses,
  std.lookup Clauses K D,
  check Cert' (bc D A L),
  Tm = (app [global (indc K)|L]).
check Cert (bc (prod _ B D) A [Tm|L]) :-
  prodE Cert Cert1 Cert2 Tm,
  check Cert1 (bc (D Tm) A L),
  check Cert2 (go B Tm).
check Cert (bc A A' []) :-
  initialE Cert,
  coq.unify-eq A A' ok.

```

■ **Figure 2** Implementation of the core calculus.

### 3.2.1 Coq-specific code

Coq terms are accessed through the Coq-Elpi API, and their representation in  $\lambda$ Prolog takes advantage of native  $\lambda$ Prolog constructs such as lists and binders. The following is part of the Coq-Elpi API signature of constants that we use.

```

kind term  type. % reification of Coq terms
kind gref  type. % reification of refs to globals
type global gref → term. % coercion to term
type indt  inductive → gref. % reification of inductive types
type indc  constructor → gref. % reification of their constructors
type app   list term → term. % reification of nary application
type prod  name → term → (term → term) → term. % reification of dependent product

```

Note that `prod` encodes dependent products by taking a name for pretty printing, a term and a  $\lambda$ Prolog abstraction from terms to terms: i.e.,  $(x : B)D$  is encoded by `prod 'x' B (x \ D x)`; when, in the implementation of the product-left rule, we apply `D` to the variable `Tm`, we get a new term that can be used to continue backchaining. This application is obtained via meta-level substitution, in the style of HOAS. In this sense, our calculus uses *implicit* substitutions, rather than explicit ones as in the LJT and PTSC's tradition; this is consistent with proof search in our application being cut-free, whereas explicit substitutions are linked to cuts. The `decideE` predicate, has, among others, the role of selecting which constructor to focus on from the inductive type. The kernel will successively obtain the type of the selected constructor, and initiate the backchaining phase. The latter is terminated when the focused atom unifies with the current goal, via Elpi's primitive `coq.unify-eq`.

### 3.2.2 FPC-specific code

The *foundational proof certificates* (FPC) framework was proposed in [16] as a flexible approach to specifying a range of proof structures in first-order classical and intuitionistic logics. Such specifications are also executable using a simple logic programming interpreter. As a result of using logic programming, proof certificates in this framework are allowed to lack details that can be reconstructed during the checking phase. For example, substitution instances of quantifiers do not need to be explicitly described within a certificate since unification within the logic programming checker is often capable of reconstructing such substitutions.

In this and the next section, we shall only use a much reduced subset of the FPC framework: in essence, an FPC will be used as a simple mechanism for bounding the search for proofs. In our examples, a proof certificate, denoted by the schematic variable  $\Xi$ , is a particular term that is threaded throughout a logic programming interpreter. For example, the inference rules in Figure 1 are augmented with an additional premise that invokes an *expert predicate* which is responsible for extracting relevant information from a proof certificate  $\Xi$  as well as constructing continuation certificates, such as,  $\Xi_1$  and  $\Xi_2$ . For example, the premise  $\text{prod}_e(\Xi, \Xi_1, \Xi_2, t)$  calls the expert for products which should extract from the certificate  $\Xi$  a substitution term  $t$  and two continuation certificates  $\Xi_1$  and  $\Xi_2$  for the two premises of this rule. If the certificate  $\Xi$  does not contain an explicit substitution term, the expert predicate can simply return a logic variable which would denote any term that satisfies subsequent unification problems arising in completing the check of this certificate.

In our case here, an FPC is a collection of  $\lambda$ Prolog clauses that provide the remaining details not supplied in Figures 1 and 2: that is, the exact set of constructors for the type of certificates `cert` as well as the specification of the expert predicates listed *ibidem*. The top of Figure 3 displays two FPCs, both of which can be used to describe proofs where we bound the dimension of a proof. For example, the first FPC dictates that the query `(check (qheight 5) A)` is provable in the kernel using the clauses in Figures 2 and 3 if and only if the height of that proof is 5 or less. Similarly, the second FPC can be used to bound the total number of instances of the decide rule in a proof. (Obviously, such proof certificates do not contain, for example, substitution terms.)

As it has been described in [6], it is also possible to pair together two different proof certificates, defined by two different FPC definitions, and do the proof checking in parallel. This means that we can build an FPC that *restricts* proofs satisfying two FPCs simultaneously. In particular, the infix constructor `<c>` in Figure 3 forms the pair of two proof certificates and the pairing experts for the certificate `Cert1 <c> Cert2` simply request that the corresponding experts also succeed for both `Cert1` and `Cert2`. Thus, the query `check ((qheight 4) <c> (qsize 10)) A` will succeed if there is a proof of `A` that has a height less than or equal to 4 while also being of size less than or equal to 10.

### 3.3 A Prolog-like tactic

Thanks to the Coq-Elpi interface, in particular to the “main” procedure `solve`, we can package the  $\lambda$ Prolog code for the checker as a tactic that can be called as any other tactic in a Coq script.

## 10:8 Two Applications of Logic Programming to Coq

```

type qheight int → cert.
type qsize   int → int → cert.
type <c>     cert → cert → cert. infixr <c> 5.

ttE (qheight _).
sortE (qheight _).
prodE (qheight H) (qheight H) (qheight H) T.
decideE Kn (qheight H) (qheight H') K :- std.mem Kn K, H > 0, H' is H - 1.
%
ttE (qsize In In).
sortE (qsize In In).
prodE (qsize In Out) (qsize In Mid) (qsize Mid Out) T.
decideE Kn (qsize In Out) (qsize In' Out) K :- std.mem Kn K, In > 0, In' is In - 1.
%
ttE (A <c> B) :- ttE A, ttE B.
sortE (A <c> B) :- sortE A, sortE B.
prodE (C1 <c> C2) (D1 <c> D2) (E1 <c> E2) T :- prodE C1 D1 E1 T, prodE C2 D2 E2 T.
decideE Kn (A <c> B) (C <c> D) K :- decideE Kn A C K, decideE Kn B D K.

```

■ **Figure 3** Sample FPCs.

```

Elpi Tactic dprolog.
Elpi Accumulate lp:{{
  solve [str " height", int N] [goal _ Ev G _] _ :-
    coq.say "Goal:" {coq.term→ string G},
    check (qheight N) (go G Term),
    Ev = Term,
    coq.say "Proof:" {coq.term→ string Ev}.
... (* Other clauses for different fpc omitted *)
}}.

```

The glue code between Coq-Elpi and the implementation of our calculus is straightforward: the goal consists of a quadruple of a (here inactive) context, an *ev*, a type (goal) and a list of extra information, also inactive. In addition, we supply the certificate: it consists of an integer (or two in the case of pairing) and a string to identify the “resource” FPC that we will use in this case. We just need to call `check` on the goal `G`, together with the certificate, in order to obtain a reconstructed proof term. We do not call the reconstruction directly on the *ev* because Coq-Elpi ensures that *ev*s manipulated by  $\lambda$ Prolog are well-typed at all times; since we cannot guarantee that, as we work with partially reconstructed term, we get around this by an explicit unification.

The following example shows how we can use the above tactic to do FPC-driven logic programming modulo conversion in Coq and return a Coq proof-term:

```

Inductive insert (x:nat) : list nat → list nat → Prop :=
| i_n : insert x [] [x]
| i_s : ∀ y : nat, ∀ ys, x <= y → insert x (y :: ys) (x :: y :: ys)
| i_c : ∀ y : nat, ∀ ys rs, y <= x → insert x ys rs → insert x (y :: ys) (x :: rs).
Lemma i1: ∃ R, insert 2 ([0] ++ [1]) R.
elpi dprolog height 10.
Qed.
Print i1.
ex_intro (fun R : list nat ⇒ insert 2 ([0] ++ [1]) R) [0; 1; 2]
  (i_c 2 0 [1] [1; 2] (1e_S 0 1 (1e_S 0 0 (1e_n 0)))
   (i_c 2 1 [] [2] (1e_S 1 1 (1e_n 1)) (i_n 2)))

```

The `dprolog` tactic implements some of the features of Coq's `eauto`; it is programmable and as such not restricted to depth-first search, since it follows the dictates of the given FPC; for example we could easily add iterative-deepening search. Furthermore, FPCs can provide a *trace* that may be more customizable than the one offered by `(e)auto`'s hard-wired `Debug` facility.

## 4 Revisiting property-based testing for Coq

We have presented in a previous paper [9] a proof-theoretical reconstruction of property-based testing (PBT) [26] of relational specifications, adopting techniques from foundational proof certificates to account for several features of this testing paradigm: from various *generation* strategies, to *shrinking* and fault localization.

Given the connection that Coq-Elpi offers between logic programming and the internals of Coq, it is natural to extend the FPC-driven logic programming interpreter of the previous section to perform PBT over **Inductive** types.

While nothing prevents us from porting all the PBT features that we have accounted for in [9], for the sake of this paper we will implement only FPC corresponding to different flavors of *exhaustive* generation, as adopted, e.g., in `SmallCheck` [60] and `αCheck` [14, 15], and their combination. Note however that it would take no more than two lines of code in the `decideE` expert to implement a form of random data generation in the sense of randomized backtracking [25].

Of course, Coq already features `QuickChick` [52] (<https://softwarefoundations.cis.upenn.edu/qc-current>), which is a sophisticated and well-supported PBT tool, based on a different perspective: being a clone of Haskell's `QuickCheck`, it emphasizes testing executable (read *decidable*) specifications with random generators. While current research [39] aims to increase automation, it is fair to say that testing with `QuickChick`, in particular relational specifications, is still very labor intensive. We do not intend to compete with `QuickChick` at this stage, but we shall see that we can test immediately **Inductive** definitions that corresponds to pure Horn programs, without having to provide a decidability proof for those definitions. Furthermore, we are not committed to a fixed random generation strategy, which, in general, requires additional work in the configuration of generators and shrinkers.

### 4.1 PBT as proof reconstruction

If we view a Horn property (in uncurried form) as a many-sorted logical formula

$$\forall x_1 : \tau_1 \dots x_n : \tau_n, A_1 \wedge \dots \wedge A_m \supset B \quad (*)$$

where the  $A_i$  and  $B$  are predicates defined using Horn clause specifications, a counter-example to this conjecture consists of a witness of the negated formula

$$\exists x_1 : \tau_1 \dots x_n : \tau_n, A_1 \wedge \dots \wedge A_m \wedge \neg B \quad (**)$$

In our Coq setting  $A_i$  and  $B$  will be propositions, while the  $\tau_j$  are honest-to-goodness datatypes. We will treat the two quite differently, in so far as elements of those datatypes will be *generated*, while predicates will only be *checked*. This distinction is reminiscent of bi-directional type-checking and plays also a part in interpreting the negation sign. In a proof system for intuitionistic logic extended with fixed points [2], negation corresponds to the usual intuitionistic interpretation, which is what Coq supports. However, for the sake of PBT and as we argued in [9], we can identify a proof certificate for **(\*\*)** simply with a proof



## 10:10 Two Applications of Logic Programming to Coq

certificate for  $\exists x_1 : \tau_1 \dots x_n : \tau_n, A_1 \wedge \dots \wedge A_m$  and we can resort to negation-as-failure to check that the conclusion does not hold without caring for any evidence for the latter. This also means that we do not produce a Coq proof term for the refutation of our property – and neither does QuickChick, which runs at the OCaml level – although we can return the witness for the existential.

### 4.2 An Elpi tactic for PBT

We will invoke the tactic in a proof environment where the overall goal is the property that the system-under-test (SUT) should meet. This means that, after **intro** has been used to introduce the relevant hypotheses, the user specifies which variables of the environment should be used for generating data and which for executing the specification. In addition to this, the user should specify all the certificate information that will guide the data generation phase. Concretely, for the property  $(*)$  and the specification of a FPC, the call to the tactic will be:

```
elpi dep_pbt <fpc> (A1 ∧ ⋯ ∧ Am) (x1) ⋯ (xn).
```

The tactic calls **check** with the given FPC on the dependent variables and delegates to a vanilla meta-interpreter (see Section A.1 in the appendix) the test of the hypotheses and of the negation of the goal:

```
interp (A1 ∧ ⋯ ∧ Am), not (interp B)
```

where **not** is λProlog’s negation-as-failure operator.

To exemplify, let us add to the previous specification of list insertion a definition of ordered list:

```
Inductive ordered : list nat → Prop :=  
| o_n : ordered []  
| o_s : ∀ x : nat, ordered [x]  
| o_c : ∀ (x y : nat), ∀ xs, ordered xs → x <= y → ordered (x :: y :: xs).
```

A property we may wish to check before embarking on a formal proof is whether insertion preserves ordered-ness:

```
Conjecture ins_ord: ∀ (x : nat) xs rs, ordered xs → insert x xs rs → ordered rs.  
intros x xs rs Ho Hi.  
elpi dep_pbt (height 5) (Ho ∧ Hi) (x) (xs).  
Abort.
```

In this query the tactic tests the hypotheses **Ho** and **Hi** against data **x**, **xs** generated exhaustively up to a height at most 5 from the library **Inductive** definitions of **nat** and **list**. We do not generate values for **rs**, since by (informal) mode information we know that it will be computed. Since we did slip in an error, our tactic reports a counter-example, namely **Proof Term**: `[0, [0; 1; 0]]`, which unpacks to **x** = 0 and **xs** = `[0; 1; 0]`. As the latter is definitely not an ordered list, this points to a quite evident bug in the definition of **ordered**. We leave the fix to the reader.

In order to generate the PBT query, some pre-processing is needed. In particular, we turn variables inhabiting datatypes into λProlog logic variables when they appear inside a specification, and generate queries for each of these logic variables in association with the type of the data variable it corresponds to. In order to realize this pre-processing step, we leverage extensively λProlog’s higher order programming features. The substitutions are handled with the technique of copy clauses [44].



Note that testing the above conjecture with QuickChick would have required much more setup: if we wished to proceed relationally as above, we would have had to provide a proof of decidability of the relevant notions. Were we to use functions, then we would have to implement a generator and shrinker for ordered lists, since automatic derivation of the former does not (yet) work for this kind of specification.

For a more significant case study, let us turn to the semantics of programming languages, where PBT has been used extensively and successfully [38]. Here we will consider a far simpler example, a typed arithmetic language featuring numerals with predecessor and test-for-zero, and Booleans with if-then-else, which comes from the *Software Foundations* book series (<https://softwarefoundations.cis.upenn.edu/plf-current/Types.html>). Whereas this example is admittedly quite simple-minded, it has, among others, been used as a benchmark for evaluating QuickChick’s automation of generators under invariants [39], and to be amenable to the tool, the specification had to be massaged non-insignificantly.

```
Inductive tm : Type :=
| ttrue : tm | tfalse : tm | tif : tm → tm → tm → tm | tzero : tm | tsucc : tm → tm
| tpred : tm → tm | tiszero : tm → tm.
Inductive typ : Type := | TBool : typ | TNat : typ.
```

The completely standard static and small step dynamic semantics rules are reported in appendix A.2.

While it is obvious that *subject expansion* fails for this calculus, it is gratifying to have it confirmed by our tactic, with counterexample  $e = \text{tif } t\text{true } t\text{zero } t\text{true}$ :

```
Conjecture subexp: ∀ e e' t, step e e' → has_type e' t → has_type e t.
intros e e' t HS HT.
elpi dep_pbt (height 2) (HS ∧ HT) (e).
Abort.
```

Another way to assess the fault detection capability of a PBT setup is via *mutation analysis* [10], whereby localized bugs are purposely inserted, with the view that they should be caught (“killed”) by a “good enough” testing suite. Following on an exercise in the aforementioned chapter of SF, we modify the typing relation by adding the following (nonsensical) clause:

```
Module M1.
Inductive has_type : tm → typ → Prop :=
...
| T_SuccBool : ∀ t, has_type t TBool → has_type (tsucc t) TBool.
end M1.
```

Some of the desired properties for our SUT now fails: not only type uniqueness, but also progress with counterexample  $e = \text{tsucc } t\text{true}$ :

```
Definition progress (e : tm) (Has_type : tm → typ → Prop) (Step : tm → tm → Prop) :=
  ∀ t, Has_type e t → notstuck e Step.
Conjecture progress_m1: ∀ e, progress e M1.has_type step.
unfold progress.
intros e t Ht.
elpi dep_pbt (height 2) (Ht) (e).
Abort.
```

To make the example slightly more interesting, we now move to an *intrinsically-typed* representation [4] of our object language, where by indexing terms with object types, we internalize the typing judgment into the syntax:

## 10:12 Two Applications of Logic Programming to Coq

```
Inductive tm : typ → Type :=
| ttrue : tm TBool | tfalse : tm TBool | tzero : tm TNat | tsucc : tm TNat → tm TNat
| tpred : tm TNat → tm TNat | tiszero : tm TNat → tm TBool
| tif: ∀ (T : typ), tm TBool → tm T → tm T → tm T.
```

Now, the operational semantics is by construction type-preserving, but bugs can still occur, see variations 3 in the same chapter that falsifies determinism of evaluation:

```
Module M3.
Inductive step : ∀ {T: typ}, tm T → tm T → Prop :=
...
| ST_Funny2 : ∀ T t1 t2 t2' t3,          (*bug*)
  t2 ⇒ t2' → (tif T t1 t2 t3) ⇒ (tif T t1 t2' t3)
End M3.
Goal ∀ (T : typ) (x y1 y2 : tm T), M3.step x y1 → M3.step x y2 → y1 = y2.
intros T x y1 y2 H1 H2.
elpi dep_pbt pair 3 5 (H1 ∧ H2) (x).
Abort.
Counterexample:
x = (tif TBool ttrue (tiszero tzero) ttrue
```

While we can deal with this encodings seamlessly, QuickChick's automatic derivation of generators is not applicable to dependent types, forcing us again either to provide decidability proofs for all judgments affected by the mutation or to embark in some non-trivial dependent functional programming, possibly based on monad transformers.

### 5 Elaboration of external proof certificates for the Coq kernel

The trusted base of Coq is its kernel, which is a type-checking program that certifies that a dependently typed  $\lambda$ -term has a given type. If type checking succeeds, the formula corresponding to that type is, in fact, accepted by Coq users as a theorem of intuitionistic logic (along with any axioms that have been asserted). The rest of the Coq system, especially its tactic language, is designed to help a human user build proofs-as- $\lambda$ -terms that can be checked by the kernel.

There are many theorem provers for intuitionistic logic [58] for which a successful proof is not the kind of detailed  $\lambda$ -term required by the Coq kernel. Often, such provers provide no information about the proofs they discover. To the extent that some evidence is output after a successful run, such evidence is usually just a trace of some key aspects of a proof, where some details are often not included. For example:

1. Substitution instances of quantifiers might not be recorded in a proof since such instances can, in principle, be reconstructed using unification.
2. Detailed typing information might not need to be stored within a proof since types can often be reconstructed during proof checking [45].
3. Some simplifications steps might be applied within a proof without recording which rewrites were used. A simple non-deterministic proof-search engine might be expected to reconstruct an equivalent simplification.

A majority of the external and automatic theorem provers for intuitionistic logic do not involve induction. Instead, they go beyond Horn clause by permitting formulas with no restriction on occurrences of  $\forall$  and  $\supset$ . In that case, we need to modify the focused proof rules that we have seen in Section 3.1 by adding the following rules.

<code>kind deb</code>	<code>type.</code>
<code>type lambda</code>	<code>deb → deb.</code>
<code>type apply</code>	<code>int → list deb → deb.</code>
<code>type idx</code>	<code>int → index.</code>
<code>type lc</code>	<code>int → deb → cert.</code>
<code>type args</code>	<code>int → list deb → cert.</code>

<code>impC</code>	<code>(lc C (lambda D)) (lc C D).</code>
<code>impE</code>	<code>(args C (A:: As)) (lc C A) (args C As).</code>
<code>initialE</code>	<code>(args C []).</code>
<code>decideE</code>	<code>(lc C (apply H A)) (args C A) (idx V) :- V is C - H - 1.</code>
<code>storeC</code>	<code>(lc C D) (x\ lc C' D) (x\ idx C) :- C' is C + 1.</code>

■ **Figure 4** The FPC definition of De Bruijn notation as proof evidence.

$$\frac{\mathcal{P}, B \vdash D}{\mathcal{P} \vdash B \supset D} \supset R \quad \frac{\mathcal{P} \vdash D[y/x]}{\mathcal{P} \vdash \forall x.D} \forall R$$

As usual, the  $\forall R$  rule has the restriction that the eigenvariable  $y$  is not free in its conclusion.

As has been detailed in earlier work on foundational proof certificates, this richer notion of proof system can provide for richer proof certificates. The main differences with what we have seen before is that the left-hand sides of sequents can now grow during the proof checking process. When reading the right introduction rule for  $\supset$  from conclusion to premise, we shall say that the antecedent of the implication is *stored* in the left side of the context. When this store action occurs, an *index* is used by the store command to name that new, left-hand formula occurrence. In this extended situation, the decide expert uses the index of an assumption in order to enter a focus phase of inference. A full proof checking kernel for first-order intuitionistic logic has been given in [16] so we do not reproduce it here.

To give an example, consider using untyped  $\lambda$ -terms encoded using De Bruijn's notation as proof certificates for propositional intuitionistic logic over just  $\supset$ . The fact that such terms can be used as proof certificates for such formulas (denoting simple types) can be formally defined using the FPC description in Figure 4. Using the constants provided in that figure, the untyped  $\lambda$ -term  $\lambda x(x(\lambda y(y(\lambda z(x(\lambda u z))))))$  can be encoded as the following Elpi term of type `deb`.

```
(lambda (apply 0 [lambda (apply 0 [lambda (apply 2 [lambda (apply 1 [])])])]) )
```

Using the constructor `lc` and `args`, terms in De Bruijn syntax (terms of type `deb`) are incorporated into proof certificates (terms of type `cert`) along with other integer arguments that are needed to compute offsets to address bound variables.

We describe here briefly how to use the technology behind FPCs and logic programming in order to provide a flexible approach to connecting external provers of first-order intuitionistic logic to Coq. Following the general outline that has been described in [6, 7], we assume that the following steps are taken.

1. Modify an external prover to output some form of proof evidence (proof certificate) for formulas it claims are theorems.
2. Develop a formal definition of the semantics behind such proof certificates using the FPC framework. The FPC for De Bruijn expressions given in Figure 4 is an example of this step.
3. Check proofs by executing the logic programming checker that is parameterized by the particular FPC definition.

As we have mentioned in Section 3.2.2, the logic programming setting allows parallel checking and synthesizing of a pair of certificates. That is, during the checking of one certificate, it is possible to synthesize, for example, a fully detailed term that is appropriate for handing to the Coq kernel. If one is interested only in building Coq kernel proof structures, we can bypass the use of an explicit pairing operation and build the synthesis of such proof structures directly into the FPC proof checker. We took exactly this step in Figure 1 where proof checking involved both proof certificates as well as Coq terms. If one is interested in checking only one kind of external proof structure, then the FPC for that structure could also be built into the checker (via, say, partial evaluation of logic programs [43]).

Continuing with the previous example, consider an external theorem prover for propositional intuitionistic logic which returns proof structures as untyped  $\lambda$ -terms using De Bruijn’s notation. Using the FPC provided in Figure 4 and the proof certificate checker in the file `ljf-dep.mod`, of the repository <https://github.com/proofcert/fpc-elpi>, the De Bruijn term displayed above can be elaborated into a proper proof for the following Coq theorem.

**Theorem** `dneg_peirce_mid` :  $\forall P Q : \mathbf{Prop}, (((((P \rightarrow Q) \rightarrow P) \rightarrow P) \rightarrow Q) \rightarrow Q)$ .

We note that this proof certificate checker and Coq proof synthesizer is rather compact, comprising less than 90 lines of Elpi code.

## 6 Conclusion and future work

This paper follows a line of research starting in the late 1980s and gaining more steam in the last five years, which advocates the usefulness of proof theory and higher-order logic programming for the many tasks concerning the development, enrichment, and even formal verification of proof assistants. The development of the Coq-Elpi plug-in has made this connection tighter.

We have presented two applications of this synergy: one supporting an out-of-the-box way to do property-based testing for inductive relations; the other geared towards providing a flexible approach to connecting external provers of first-order intuitionistic logic to Coq.

The code reported in Fig. 1 is a simplification for exposition purposes of the real implementation of the kernel. Following ideas from bidirectional type checking, we have factored out the product left rule in  $\forall - L$  and  $\supset - L$ , where the former delegates to Coq the *check* that the instantiation term  $t$  is well-typed w.r.t.  $B$ , while in the latter, proof search will *generate* such a term, given the type  $B$ . There are also other minor tweaks, such as a rule performing weak-head reduction, allowing us to handle directly existential goals.

There are many avenues of development for this line of research. We would like to exploit one of the distinguishing features of Elpi: the *delay* mechanism. The use of constraints for data generation is well developed [27] and we could try to leverage it to improve our PBT tactic to generate partially instantiated terms, without recurring to needed narrowing as in `LazySmallCheck` [60]. On the more practical side, it would be worthwhile to investigate *random* generation, following the ideas in [25, 9].

Finally, it makes sense to tie together the two threads of this paper and provide a way of checking and elaborating proof evidence for intuitionistic logic *over* (inductively) defined atoms using previously proved lemmata, that is capturing most of the features of `eauto`. This can be pushed further up to FPC for (co)inductive proofs [8].

The source files mentioned in this paper are available at <https://github.com/proofcert/fpc-elpi>.

## References

- 1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992. doi:10.1093/logcom/2.3.297.
- 2 David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1):2:1–2:44, 2012. doi:10.1145/2071368.2071370.
- 3 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014. doi:10.6092/issn.1972-5787/4650.
- 4 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reason.*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.
- 5 G. Bierman. *On Intuitionistic Linear Logic*. PhD thesis, Computing Laboratory, University of Cambridge, 1994.
- 6 Roberto Blanco, Zakaria Chihani, and Dale Miller. Translating between implicit and explicit versions of proof. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 255–273. Springer, 2017. doi:10.1007/978-3-319-63046-5\_16.
- 7 Roberto Blanco, Matteo Manighetti, and Dale Miller. FPC-Coq: Using Elpi to elaborate external proof evidence into Coq proofs. Technical Report hal-02974002, Inria, July 2020. Presented at the Coq Workshop 2020. URL: <https://hal.inria.fr/hal-02974002>.
- 8 Roberto Blanco and Dale Miller. Proof outlines as proof certificates: a system description. In Iliano Cervesato and Carsten Schürmann, editors, *Proceedings First International Workshop on Focusing*, volume 197 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–14. Open Publishing Association, 2015. doi:10.4204/EPTCS.197.2.
- 9 Roberto Blanco, Dale Miller, and Alberto Momigliano. Property-based testing via proof reconstruction. In E. Komendantskaya, editor, *Principles and Practice of Programming Languages 2019 (PPDP '19)*, October 2019. doi:10.1145/3354166.3354170.
- 10 Matteo Cavada, Andrea Colò, and Alberto Momigliano. MutantChick: Type-preserving mutation analysis for Coq. In *CILC*, volume 2710 of *CEUR Workshop Proceedings*, pages 105–112. CEUR-WS.org, 2020. URL: <http://ceur-ws.org/Vol-2710/short2.pdf>.
- 11 Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003. doi:10.1093/logcom/13.5.639.
- 12 Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A multi-focused proof system isomorphic to expansion proofs. *J. of Logic and Computation*, 26(2):577–603, 2016. doi:10.1093/logcom/exu030.
- 13 Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong, editors, *Fifth International Conference on Theoretical Computer Science*, volume 273 of *IFIP*, pages 383–396. Springer, September 2008. doi:10.1007/978-0-387-09680-3\_26.
- 14 James Cheney and Alberto Momigliano.  $\alpha$ Check: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311–352, 2017. doi:10.1017/S1471068417000035.
- 15 James Cheney, Alberto Momigliano, and Matteo Pessina. Advances in property-based testing for  $\lambda$ prolog. In *TAP@STAF*, volume 9762 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2016. doi:10.1007/978-3-319-41135-4\_3.
- 16 Zakaria Chihani, Dale Miller, and Fabien Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017. doi:10.1007/s10817-016-9380-6.
- 17 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. 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 143–156. ACM, 2008. doi:10.1145/1411204.1411226.

- 18 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in coq with elpi (system description). In *FSCD*, volume 167 of *LIPICs*, pages 34:1–34:21. Schloss Dagstuhl, 2020. doi:10.4230/LIPICs.FSCD.2020.34.
- 19 Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer, April 1995. doi:10.1007/BFb0014049.
- 20 Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing HOL in an higher order logic programming language. In *LFMTP*, pages 4:1–4:10. ACM, 2016. doi:10.1145/2966268.2966272.
- 21 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable,  $\lambda$ prolog interpreter. In *LPAR*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015. doi:10.1007/978-3-662-48899-7\_32.
- 22 Amy Felty. Encoding the calculus of constructions in a higher-order logic. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 233–244. IEEE, June 1993. doi:10.1109/LICS.1993.287584.
- 23 Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, 1993. doi:10.1007/BF00881900.
- 24 Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in *Lecture Notes in Computer Science*, pages 61–80, Argonne, IL, 1988. Springer. doi:10.1007/BFb0012823.
- 25 Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 383–405. Springer, 2015. doi:10.1007/978-3-662-46669-8\_16.
- 26 George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, pages 74–80, July 1997. doi:10.1145/263244.263267.
- 27 Fabio Fioravanti, Maurizio Proietti, and Valerio Senni. Efficient generation of test data structures using constraint logic programming and program transformation. *J. Log. Comput.*, 25(6):1263–1283, 2015. doi:10.1093/logcom/ext071.
- 28 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012. doi:10.1007/s10817-011-9218-1.
- 29 Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935. doi:10.1007/BF01201353.
- 30 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. doi:10.1016/0304-3975(87)90045-4.
- 31 Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. doi:10.1007/3-540-09724-4.
- 32 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory in higher order constraint logic programming. *Mathematical Structures in Computer Science*, 29(8):1125–1150, 2019. doi:10.1017/S0960129518000427.
- 33 John Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123–152, April 1993. doi:10.1017/S0956796800000666.
- 34 Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1994. doi:10.1007/BFb0022247.
- 35 Furio Honsell, Marino Miculan, and Ivan Scagnetto.  $\pi$ -calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001. doi:10.1016/S0304-3975(00)00095-5.



- 36 Martin Hyland and Valeria de Paiva. Full intuitionistic linear logic (extended abstract). *Annals of Pure and Applied Logic*, 64(3):273–291, 11 November 1993. doi:10.1016/0168-0072(93)90146-5.
- 37 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987. doi:10.1007/BFb0039592.
- 38 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Raskind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103691.
- 39 Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, 2018. doi:10.1145/3158133.
- 40 Stéphane Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, University of St. Andrews, December 2006. URL: <https://tel.archives-ouvertes.fr/tel-00134646>.
- 41 Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A sequent calculus for type theory. In *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2006. doi:10.1007/11874683\_29.
- 42 Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009. doi:10.1016/j.tcs.2009.07.041.
- 43 J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3):217–242, 1991. doi:10.1016/0743-1066(91)90027-M.
- 44 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991. doi:10.1093/logcom/1.4.497.
- 45 Dale Miller. Proof checking and logic programming. *Formal Aspects of Computing*, 29(3):383–399, 2017. doi:10.1007/s00165-016-0393-z.
- 46 Dale Miller. Mechanized metatheory revisited. *Journal of Automated Reasoning*, 63(3):625–665, October 2019. doi:10.1007/s10817-018-9483-3.
- 47 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. doi:10.1017/CB09781139021326.
- 48 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. doi:10.1016/0168-0072(91)90068-w.
- 49 Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
- 50 Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, 1990. doi:10.1145/96559.96570.
- 51 Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of  $\lambda$ Prolog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer. doi:10.1007/3-540-48660-7\_25.
- 52 Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *ITP 2015*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015. doi:10.1007/978-3-319-22102-1\_22.
- 53 Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989. doi:10.1007/BF00248324.
- 54 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988. doi:10.1145/53990.54010.

- 55 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in Lecture Notes in Computer Science, pages 15–21, 2010. doi:10.1007/978-3-642-14203-1\_2.
- 56 Elaine Pimentel, Vivek Nigam, and Jo ao Neto. Multi-focused proofs with different polarity assignments. In Mario Benevides and Rene Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015)*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 163–179, 2016. doi:10.1016/j.entcs.2016.06.011.
- 57 Gordon D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, 1981.
- 58 Thomas Rathus, Jens Otten, and Christoph Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1):261–271, 2007. doi:10.1007/s10817-006-9060-z.
- 59 C. Röckl, D. Hirschhoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proc. FOSSACS’01*, volume 2030 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2001. doi:10.1007/3-540-45315-6\_24.
- 60 Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In Andy Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008. doi:10.1145/1411286.1411292.
- 61 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5):947–999, 2020. doi:10.1007/s10817-019-09540-0.
- 62 Enrico Tassi. Elpi: an extension language for Coq. CoqPL 2018: The Fourth International Workshop on Coq for Programming Languages, 2018. URL: <https://hal.inria.fr/hal-01637063/>.
- 63 Enrico Tassi. Deriving proved equality tests in Coq-Elpi: Stronger induction principles for containers in Coq. In *ITP*, volume 141 of *LIPICs*, pages 29:1–29:18. Schloss Dagstuhl, 2019. doi:10.4230/LIPICs.ITP.2019.29.

## A Appendix

In this appendix we list some definitions and pieces of code that we have mentioned in the main paper.

### A.1 The vanilla meta-interpreter

We report below the encoding of the vanilla meta-interpreter used in the testing phase of the `dep_pbt` tactic. Differently from Fig. 2 we appeal, via quotations, to Coq’s defined connectives. An atomic proposition is one defined **Inductively**.

```
type interp term → o.
type backchain term → term → o.
```

```
interp {{True}}.
interp (sort _).
interp {{lp:G1 ∧ lp:G2}} :- interp G1, interp G2.
interp {{lp:G1 ∨ lp:G2}} :- interp G1; interp G2.
interp {{lp:T1 = lp:T2}} :- coq.unify-eq T1 T2 ok.
interp {{ex (lp:G)}} :- interp (G X).
```



```

interp Atom :-
  atomic Atom,
  coq.safe-dest-app Atom (global (indt Prog)) _,
  coq.env.indt Prog _ _ _ _ KTypes,
  std.mem KTypes D, backchain D Atom.
backchain A A' :- atomic A, coq.unify-eq A A' ok.
backchain D A :- is_imp D A D', !, backchain D' A, interp Ty.
backchain D A :- is_uni D D', backchain (D' X) A.

```

## A.2 Semantics of the typed arithmetic language

We list the rules for static and dynamic semantics of the language mentioned in Section 4.2 and related notions:

```

Inductive has_type : tm → typ → Prop :=
| T_True : has_type ttrue TBool
| T_Fls : has_type tfalse TBool
| T_Test : ∀ t1 t2 t3 T,
  has_type t1 TBool → has_type t2 T → has_type t3 T → has_type (tif t1 t2 t3) T
| T_Zro : has_type tzero TNat
| T_Scc : ∀ t1, has_type t1 TNat → has_type (tsucc t1) TNat
| T_Prd : ∀ t1, has_type t1 TNat → has_type (tpred t1) TNat
| T_Iszro : ∀ t1, has_type t1 TNat → has_type (tiszero t1) TBool.

```

```

Inductive nvalue : tm → Prop :=

```

```

| nv_zero : nvalue tzero
| nv_succ : ∀ t, nvalue t → nvalue (tsucc t).

```

```

Inductive bvalue : tm → Prop :=

```

```

| bv_t : bvalue ttrue
| bv_f : bvalue tfalse.

```

Reserved Notation "t1 '=>' t2" (at level 40).

```

Inductive step : tm → tm → Prop :=

```

```

| ST_IfTrue : ∀ t1 t2, (tif ttrue t1 t2) ==> t1
| ST_IfFalse : ∀ t1 t2, (tif tfalse t1 t2) ==> t2
| ST_If : ∀ t1 t1' t2 t3,
  t1 ==> t1' → (tif t1 t2 t3) ==> (tif t1' t2 t3)
| ST_Succ : ∀ t1 t1',
  t1 ==> t1' → (tsucc t1) ==> (tsucc t1')
| ST_PredZero : (tpred tzero) ==> tzero
| ST_PredSucc : ∀ t1,
  nvalue t1 → (tpred (tsucc t1)) ==> t1
| ST_Pred : ∀ t1 t1',
  t1 ==> t1' → (tpred t1) ==> (tpred t1')
| ST_IszeroZero : (tiszero tzero) ==> ttrue
| ST_IszeroSucc : ∀ t1,
  nvalue t1 → (tiszero (tsucc t1)) ==> tfalse
| ST_Iszero : ∀ t1 t1',
  t1 ==> t1' → (tiszero t1) ==> (tiszero t1')

```

where "t1 '=>' t2" := (step t1 t2).

```

Inductive notstuck (e : tm) (Step : tm → tm → Prop) : Prop :=

```

```

| pn : nvalue e → notstuck e Step
| pb : bvalue e → notstuck e Step
| ps e' : Step e e' → notstuck e Step.

```



# Duality in Intuitionistic Propositional Logic

Paweł Urzyczyn  

Institute of Informatics, University of Warsaw, Poland

---

## Abstract

It is known that provability in propositional intuitionistic logic is PSPACE-complete. As PSPACE is closed under complements, there must exist a LOGSPACE-reduction from refutability to provability. Here we describe a direct translation: given a formula  $\varphi$ , we define  $\bar{\varphi}$  so that  $\bar{\varphi}$  is provable if and only if  $\varphi$  is not.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Proof theory; Theory of computation  $\rightarrow$  Constructive mathematics; Theory of computation  $\rightarrow$  Complexity theory and logic

**Keywords and phrases** Intuitionistic logic, Complexity

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2020.11

## Introduction

A dual concept to proof theory is refutation theory [9] where one asks how to refute or disprove a formula. Various refutation systems occur in the literature, e.g. [2, 4, 9] to derive formal refutations. This paper takes a look from another angle: we ask if one can internalize refutability as provability. A positive answer to this question may depend on the particular logic, the intuitionistic propositional calculus (IPC) being a most promising case. Indeed, the PSPACE-completeness of IPC means that non-provability is LOGSPACE reducible to provability and vice versa. Here we show how to construct, for a given formula  $\varphi$ , another formula  $\bar{\varphi}$  which is provable if and only if  $\varphi$  is not. The construction works in logarithmic space, in particular in polynomial time.

The inspiration for our approach comes from a computational interpretation of logic, which can be seen as yet another side of Curry-Howard isomorphism, namely the equivalence:

$$\textit{Proof construction} \quad \Leftrightarrow \quad \textit{Computation}$$

This paradigm is implicitly exploited by many authors, especially in hardness and undecidability proofs, e.g. [5], but it is rarely explicitly formulated. The idea is extremely simple: the task to prove a judgment of the form

$$\Gamma \vdash \tau$$

is nothing else than a configuration of a machine, where

- $\tau$  is the present internal state, and
- $\Gamma$  is the contents of memory.

The use of machine memory has to be cautious: usually assumptions are non-disposable (unless we deal with some substructural logic) and one cannot verify that an assumption is not available. On the other hand, proof search algorithms naturally use both nondeterministic choices and universal splits (recursive calls). Machines adequate for IPC should therefore be defined as alternating automata operating on write-once binary registers. Every register represents an assumption: the value true means that the assumption is available. Registers can only be accessed as positive guards: to execute an action the machine may have to check that a given register is set to true. A register cannot be checked for the value false nor unset to false. A variant of this model is mentioned in [6], an elaborated first-order version is developed in [7]. The Wajsberg/Ben-Yelles algorithm for IPC, like in [10], can easily



© Paweł Urzyczyn;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Types for Proofs and Programs (TYPES 2020).

Editors: Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch; Article No. 11; pp. 11:1–11:10

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be implemented on such monotone automata [8]. On the other hand, for every monotone automaton  $M$  one can construct a formula  $\varphi_M$  such that  $\varphi_M$  is a theorem of IPC if and only if  $M$  halts. The formula  $\varphi_M$  can be defined using implication as the only connective. So it is just a simple type and only of order 3.

Under this understanding, our construction in this paper can be seen as complementation of monotone automata: given an automaton  $M$  define another automaton  $\overline{M}$  so that  $\overline{M}$  halts if and only if  $M$  does not halt. Such interpretation inspired our presentation below which can be easily translated to the language of automata. This could make the whole development somewhat more concise and technically direct, but we decided to remain within the language of propositional logic, to demonstrate its flexibility.

Our goal is to define a formula  $\overline{\varphi}$  that has a proof when a given  $\varphi$  has none. What  $\overline{\varphi}$  actually states is that  $\varphi$  has no *normal* proof *without repeated judgments* (and therefore of bounded size). To handle the first aspect we use lambda-notation for proofs and we appeal to normalization. To control proof size we found it convenient to define a restricted version of natural deduction rules (Figure 1) where additional annotations are used to disallow cycles.

## Natural deduction

We consider propositional formulas built from the connectives  $\wedge$ ,  $\vee$ ,  $\rightarrow$  and  $\perp$ . Variables and  $\perp$  are called *atoms*. Negation  $\neg\alpha$  is defined as  $\alpha \rightarrow \perp$ . We assume that implication is right-associative, i.e., we write  $\alpha \rightarrow \beta \rightarrow \gamma$  for  $\alpha \rightarrow (\beta \rightarrow \gamma)$ . If  $\mathcal{S} = \{\alpha_1, \dots, \alpha_k\}$  then  $\mathcal{S} \rightarrow \beta$  abbreviates any formula of the form  $\alpha_1 \rightarrow \dots \rightarrow \alpha_k \rightarrow \beta$  (disregarding the order of premises). Notation for sets of formulas is simplified, e.g.  $\Gamma, \Sigma$  stands for  $\Gamma \cup \Sigma$  and  $\Gamma, \alpha$  for  $\Gamma \cup \{\alpha\}$ .

Our natural deduction calculus (Figure 1) derives judgments of the form  $\Gamma \vdash \alpha [\Sigma]$ , where  $\Gamma$  and  $\Sigma$  are sets of formulas, and  $\alpha$  is a formula. The meaning of  $\Gamma \vdash \alpha [\Sigma]$  is that the ordinary judgment  $\Gamma \vdash \alpha$  is provable without (directly) addressing proof goals in  $\Sigma$ . To see this, one reads the rules upwards, in the order of proof search. Then, at every step, the set  $\Sigma$  of forbidden goals is expanded by the current goal unless a new assumption is added; then  $\Sigma$  is reset to  $\emptyset$ . This protocol ensures that no judgment can be repeated on any proof branch. Note that the rules are “upward-preserving” in that all assumptions occurring in conclusion must occur in the premises as well.

A convenient proof notation for propositional intuitionistic logic is an extended lambda-calculus as e.g. in [1]. From this point of view, natural deduction becomes a type-assignment (or, perhaps more adequately, “term-assignment” or “proof-assignment”) system (Figure 2), deriving judgments  $\Gamma \vdash M : \alpha [\Sigma]$  with the additional term component  $M$ . (N.B. we identify  $\alpha$ -convertible terms.) Strictly speaking,  $\Gamma$  can no longer be just a set of formulas and must be understood as a *type environment*, i.e., a set of variable declarations  $(x : \alpha)$ . Fortunately, we do not need to consider environments  $\Gamma$  involving more than one declaration of the same  $\alpha$ . To make it precise, we say that an environment is *simple* when  $(x : \alpha), (y : \alpha) \in \Gamma$  implies  $x = y$ . Simple environments can thus be identified with sets of formulas. In Figure 2, we assume  $\Gamma$  simple in all rules,<sup>1</sup> so the notation  $\alpha \in \Gamma$  (resp.  $\alpha \notin \Gamma$ ) can safely be read as “there is (resp. is not) a variable of type  $\alpha$  in  $\Gamma$ ”. Note that in rule  $(W \rightarrow_2)$  it is assumed that  $\gamma \in \Gamma$  despite the lambda-introduction.

<sup>1</sup> The insightful reader should note that we do not claim subject reduction for the system in Figure 2, cf. e.g. [3]; the existence of normal forms is inherited from the standard system.

$$\begin{array}{c}
\text{(Ax)} \quad \Gamma \vdash \alpha [\Sigma] \quad [\alpha \in \Gamma] \qquad \qquad \qquad \text{(E}\perp\text{)} \quad \frac{\Gamma \vdash \perp [\Sigma, \alpha]}{\Gamma \vdash \alpha [\Sigma]} \quad [\perp \notin \Sigma] \\
\\
\text{(E}\wedge_1\text{)} \quad \frac{\Gamma \vdash \alpha \wedge \beta [\Sigma, \alpha]}{\Gamma \vdash \alpha [\Sigma]} \quad [\alpha \wedge \beta \notin \Sigma] \qquad \qquad \text{(E}\wedge_2\text{)} \quad \frac{\Gamma \vdash \beta \wedge \alpha [\Sigma, \alpha]}{\Gamma \vdash \alpha [\Sigma]} \quad [\beta \wedge \alpha \notin \Sigma] \\
\\
\text{(W}\wedge\text{)} \quad \frac{\Gamma \vdash \gamma [\Sigma, \gamma \wedge \delta] \quad \Gamma \vdash \delta [\Sigma, \gamma \wedge \delta]}{\Gamma \vdash \gamma \wedge \delta [\Sigma]} \quad [\gamma, \delta \notin \Sigma] \\
\\
\text{(E}\vee\text{)} \quad \frac{\Gamma \vdash \gamma \vee \delta [\Sigma, \alpha] \quad \Gamma, x : \gamma \vdash \alpha [\emptyset] \quad \Gamma, y : \delta \vdash \alpha [\emptyset]}{\Gamma \vdash \alpha [\Sigma]} \quad [\gamma \vee \delta \notin \Sigma, \gamma, \delta \notin \Sigma] \\
\\
\text{(W}\vee_1\text{)} \quad \frac{\Gamma \vdash \gamma [\Sigma, \gamma \vee \delta]}{\Gamma \vdash \gamma \vee \delta [\Sigma]} \quad [\gamma \notin \Sigma] \qquad \qquad \text{(W}\vee_2\text{)} \quad \frac{\Gamma \vdash \delta [\Sigma, \gamma \vee \delta]}{\Gamma \vdash \gamma \vee \delta [\Sigma]} \quad [\delta \notin \Sigma] \\
\\
\text{(E}\rightarrow\text{)} \quad \frac{\Gamma \vdash \beta \rightarrow \alpha [\Sigma, \alpha] \quad \Gamma \vdash \beta [\Sigma, \alpha]}{\Gamma \vdash \alpha [\Sigma]} \quad [\beta, \beta \rightarrow \alpha \notin \Sigma] \\
\\
\text{(W}\rightarrow_1\text{)} \quad \frac{\Gamma, \gamma \vdash \delta [\emptyset]}{\Gamma \vdash \gamma \rightarrow \delta [\Sigma]} \quad [\gamma \notin \Gamma] \qquad \qquad \text{(W}\rightarrow_2\text{)} \quad \frac{\Gamma \vdash \delta [\Sigma, \gamma \rightarrow \delta]}{\Gamma \vdash \gamma \rightarrow \delta [\Sigma]} \quad [\delta \notin \Sigma, \gamma \in \Gamma]
\end{array}$$

■ **Figure 1** Natural deduction.

It is convenient to use term notation to express properties of proofs. But the principal use of lambda-terms is that they normalize, and thus proof search can be restricted to lambda-terms in normal form [1].

We are very relaxed regarding the notation. For example we write  $\Gamma \vdash M : \alpha [\Sigma]$  when it is convenient to mention the proof  $M$ , and  $\Gamma \vdash \alpha [\Sigma]$  when  $M$  is not relevant. Lambda-terms are, for simplicity, written in Curry-style (without type decoration) but types are always implicit, and can be marked e.g. as superscripts, whenever it is useful. The notation  $\Gamma \vdash \alpha$  and  $\Gamma \vdash M : \alpha$  means ordinary intuitionistic provability and term-assignment as in [1]. Substitution of  $N$  for free occurrences of  $x$  in  $M$  is written  $M[x := N]$ .

The following definition is needed for the proof of completeness of our system (Lemma 2). Let a term  $M$  be typable in a simple environment  $\Gamma$ . The set  $U_\Gamma(M)$  of types *directly used in*  $M$  with respect to  $\Gamma$  is defined by induction below. Informally, members of  $U_\Gamma(M)$  are (with one exception) types of proper subterms of  $M$ , not in scope of a variable binding in  $M$ . The exception is a lambda-abstraction representing an unnecessary (duplicated) assumption.

$$\begin{array}{l}
U_\Gamma(x) = \emptyset; \\
U_\Gamma(P^{\gamma \rightarrow \delta} M^\gamma) = U_\Gamma(P) \cup U_\Gamma(M) \cup \{\gamma \rightarrow \delta, \gamma\}; \\
U_\Gamma(\lambda x^\gamma. N^\delta) = U_\Gamma(N[x := y]) \cup \{\delta\}, \text{ when } (y : \gamma) \in \Gamma, \text{ and } U_\Gamma(\lambda x^\gamma. N^\delta) = \emptyset, \text{ otherwise;} \\
U_\Gamma(M[\alpha]) = U_\Gamma(M) \cup \{\perp\}; \\
U_\Gamma(P^{\gamma \wedge \delta} \{1\}) = U_\Gamma(P^{\gamma \wedge \delta} \{2\}) = U_\Gamma(P) \cup \{\gamma \wedge \delta\}; \\
U_\Gamma(\langle M^\gamma, N^\delta \rangle) = U_\Gamma(M) \cup U_\Gamma(N) \cup \{\gamma, \delta\}; \\
U_\Gamma(\langle M^\gamma \rangle_1) = U_\Gamma(M) \cup \{\gamma\}, \text{ and } U_\Gamma(\langle M^\delta \rangle_2) = U_\Gamma(M) \cup \{\delta\}; \\
U_\Gamma(P^{\gamma \vee \delta} [x.M, y.N]) = U_\Gamma(P) \cup \{\gamma \vee \delta\}.
\end{array}$$

$$\begin{array}{c}
 \text{(Ax)} \quad \Gamma, x : \alpha \vdash x : \alpha [\Sigma] \qquad \qquad \qquad \text{(E}\perp\text{)} \quad \frac{\Gamma \vdash P : \perp [\Sigma, \alpha]}{\Gamma \vdash P[\alpha] : \alpha [\Sigma]} [\perp \notin \Sigma] \\
 \\
 \text{(E}\wedge_1\text{)} \quad \frac{\Gamma \vdash P : \alpha \wedge \beta [\Sigma, \alpha]}{\Gamma \vdash P\{1\} : \alpha [\Sigma]} [\alpha \wedge \beta \notin \Sigma] \qquad \text{(E}\wedge_2\text{)} \quad \frac{\Gamma \vdash P : \beta \wedge \alpha [\Sigma, \alpha]}{\Gamma \vdash P\{2\} : \alpha [\Sigma]} [\beta \wedge \alpha \notin \Sigma] \\
 \\
 \text{(W}\wedge\text{)} \quad \frac{\Gamma \vdash M : \gamma [\Sigma, \gamma \wedge \delta] \quad \Gamma \vdash N : \delta [\Sigma, \gamma \wedge \delta]}{\Gamma \vdash \langle M, N \rangle : \gamma \wedge \delta [\Sigma]} [\gamma, \delta \notin \Sigma] \\
 \\
 \text{(E}\vee\text{)} \quad \frac{\Gamma \vdash P : \gamma \vee \delta [\Sigma, \alpha] \quad \Gamma, x : \gamma \vdash M : \alpha [\emptyset] \quad \Gamma, y : \delta \vdash N : \alpha [\emptyset]}{\Gamma \vdash P[x.M, y.N] : \alpha [\Sigma]} [\gamma \vee \delta \notin \Sigma, \gamma, \delta \notin \Gamma] \\
 \\
 \text{(W}\vee_1\text{)} \quad \frac{\Gamma \vdash M : \gamma [\Sigma, \gamma \vee \delta]}{\Gamma \vdash \langle M \rangle_1 : \gamma \vee \delta [\Sigma]} [\gamma \notin \Sigma] \qquad \text{(W}\vee_2\text{)} \quad \frac{\Gamma \vdash M : \delta [\Sigma, \gamma \vee \delta]}{\Gamma \vdash \langle M \rangle_2 : \gamma \vee \delta [\Sigma]} [\delta \notin \Sigma] \\
 \\
 \text{(E}\rightarrow\text{)} \quad \frac{\Gamma \vdash P : \beta \rightarrow \alpha [\Sigma, \alpha] \quad \Gamma \vdash M : \beta [\Sigma, \alpha]}{\Gamma \vdash PM : \alpha [\Sigma]} [\beta, \beta \rightarrow \alpha \notin \Sigma] \\
 \\
 \text{(W}\rightarrow_1\text{)} \quad \frac{\Gamma, x : \gamma \vdash M : \delta [\emptyset]}{\Gamma \vdash \lambda x M : \gamma \rightarrow \delta [\Sigma]} [\gamma \notin \Gamma] \qquad \text{(W}\rightarrow_2\text{)} \quad \frac{\Gamma \vdash M : \delta [\Sigma, \gamma \rightarrow \delta]}{\Gamma \vdash \lambda x M : \gamma \rightarrow \delta [\Sigma]} [\delta \notin \Sigma, \gamma \in \Gamma]
 \end{array}$$

■ **Figure 2** Extended lambda-calculus.

► **Lemma 1.** *If  $\Gamma \vdash M : \alpha$ , and  $\beta \in U_\Gamma(M)$ , then  $\Gamma \vdash N : \beta$ , for some term  $N$ , shorter than  $M$ . In particular, if  $M$  is the shortest term of type  $\alpha$  in  $\Gamma$ , then  $\alpha \notin U_\Gamma(M)$ .*

**Proof.** Easy induction with respect to  $M$ . ◀

► **Lemma 2.** *The system in Figure 2 is sound and complete in the following sense:*

- *If  $\Gamma \vdash M : \alpha [\Sigma]$ , for some  $\Sigma$ , then  $\Gamma \vdash M : \alpha$ ;*
- *If  $\Gamma \vdash M : \alpha$ , and  $\Gamma$  is simple, then  $\Gamma \vdash M : \alpha [\Sigma]$ , for all  $\Sigma$  with  $\Sigma \cap U_\Gamma(M) = \emptyset$ . In particular,  $\Gamma \vdash M : \alpha$  is equivalent to  $\Gamma \vdash M : \alpha [\emptyset]$ .*

**Proof.** The first part follows by a simple erasure of the unnecessary annotations. The second part we prove by induction with respect to the size of a smallest possible witness  $M$  such that  $\Gamma \vdash M : \alpha$  holds. From Lemma 1 we know that  $\alpha \notin U_\Gamma(M)$ .

If  $M$  is a variable then the claim holds trivially. Assume that  $M = P[\alpha]$  with  $P$  of type  $\perp$ . Then  $\alpha \notin U_\Gamma(M) = U_\Gamma(P) \cup \{\perp\}$ , hence  $(\Sigma, \alpha) \cap U_\Gamma(P) = \emptyset$ . Observe that  $P$  is the shortest term of type  $\perp$ , hence  $\Gamma \vdash P : \perp [\Sigma, \alpha]$  holds by induction. Also  $\Sigma \cap U_\Gamma(M) = \emptyset$ , so  $\perp \notin \Sigma$ , rule (E $\perp$ ) applies, and yields  $\Gamma \vdash M : \alpha [\Sigma]$ .

Consider the case  $M = P[x.R, y.N]$  with  $P$  of type  $\gamma \vee \delta$ . Then  $\Gamma \vdash P : \gamma \vee \delta [\Sigma, \alpha]$  holds by induction, because  $\Sigma, \alpha$  is disjoint with  $U_\Gamma(P)$ , as  $\alpha \notin U_\Gamma(M) \supseteq U_\Gamma(P)$ . Now note that  $\gamma, \delta \notin \Gamma$ , as otherwise either  $R$  or  $N$  would make a shorter inhabitant of  $\alpha$  than  $M$ . It follows that environments  $\Gamma, \gamma$  and  $\Gamma, \delta$  are simple. Hence the judgments  $\Gamma, x : \gamma \vdash R : \alpha [\emptyset]$  and  $\Gamma, y : \delta \vdash N : \alpha [\emptyset]$  also hold by induction, because the empty set is disjoint with everything. To apply rule (E $\vee$ ) we check that  $\gamma \vee \delta \notin \Sigma$  because  $\gamma \vee \delta \in U_\Gamma(M)$ .

As another example consider  $\alpha = \gamma \rightarrow \delta$ , and let  $M = \lambda x N$ . Then  $\Gamma, x : \gamma \vdash N : \delta$ , and we have two cases depending on whether  $\gamma \in \Gamma$  or not. If  $\gamma \notin \Gamma$  then  $\Gamma, x : \gamma$  is a simple environment, and  $\Gamma, x : \gamma \vdash N : \delta[\emptyset]$  holds by the induction hypothesis. Thus  $\Gamma \vdash \lambda x N : \gamma \rightarrow \delta[\Sigma]$  using rule  $(W \rightarrow_1)$ .

If  $\gamma \in \Gamma$ , say  $(y : \gamma) \in \Gamma$ , then  $\Gamma, x : \gamma$  is not simple. But then  $\Gamma \vdash N[x := y] : \delta$ . The term  $N[x := y]$  is of the same size as  $N$ , so it is still a smallest possible term of type  $\delta$ . Now  $U_\Gamma(N[x := y]) \cap (\Sigma, \gamma \rightarrow \delta) = \emptyset$  because  $\gamma \rightarrow \delta \notin U_\Gamma(M) \supseteq U_\Gamma(N[x := y])$  and  $U_\Gamma(M) \cap \Sigma = \emptyset$ . So we can apply the induction hypothesis to  $\Gamma \vdash N[x := y] : \delta$  and infer  $\Gamma \vdash N[x := y] : \delta[\Sigma, \gamma \rightarrow \delta]$ . Since  $\delta \in U_\Gamma(M)$ , we have  $\delta \notin \Sigma$ , so rule  $(W \rightarrow_2)$  yields  $\Gamma \vdash \lambda y. N[x := y] : \gamma \rightarrow \delta[\Sigma]$  and it remains to observe that  $\lambda y. N[x := y]$  is just the same term as  $\lambda x N$ . Other cases are similar. ◀

## The construction

In what follows we fix a formula  $\varphi$  and we define a formula  $\bar{\varphi}$  to satisfy the equivalence:

$$\not\vdash \varphi \quad \Leftrightarrow \quad \vdash \bar{\varphi}. \quad (*)$$

Let  $\varphi$  be of length  $n$  and let  $\mathcal{S}$  be the set of all subformulas of  $\varphi$ . Then  $\mathcal{S}$  has at most  $n$  elements. For  $\alpha, \beta \in \mathcal{S}$ , and  $t = 0, \dots, n$ , the following propositional symbols may occur in  $\bar{\varphi}$ :

- $D_{\alpha,t}$  – “Refute  $\alpha$  in  $n - t$  phases”;
- $D'_{\alpha,t}$  – “Refute  $\alpha$  in  $n - t$  phases without addressing goal  $\alpha$  again”;
- $P_{\alpha,t}$  – “Assumption  $\alpha$  present in phase  $t$ ”;
- $N_{\alpha,t}$  – “Assumption  $\alpha$  not added in phase  $t$ ”;
- $X_{\alpha,t}$  – “Goal  $\alpha$  cannot be derived in phase  $t$  using the axiom rule”;
- $W_{\alpha,t}, W_{\alpha,t}^1, W_{\alpha,t}^2$  – “Goal  $\alpha$  cannot be derived in phase  $t$  by introduction”;
- $E_{\alpha,\beta,t}$  – “Goal  $\alpha$  cannot be derived in phase  $t$  by elimination of  $\beta$ ”.

Atoms subscripted by  $t$  are called  $t$ -atoms. The intuitive meaning of those is given above. However, the role of  $D_{\alpha,t}$  is twofold and depends on whether  $D_{\alpha,t}$  occurs as a proof goal or as an assumption. While proving  $D_{\alpha,t}$  amounts to disproving  $\alpha$ , an assumption of  $D_{\alpha,t}$  should be read as disqualifying  $\alpha$  as a possible proof goal.

When  $\beta \in \mathcal{S}$ ,  $t \leq n$ ,  $\Gamma \subseteq \mathcal{S}$ , we use the abbreviations:

- $\mathcal{A}_{\beta,t} = \{P_{\beta,t}\} \cup \{N_{\alpha,t} \mid \alpha \in \mathcal{S} \wedge \alpha \neq \beta\}$  – “The unique assumption added in phase  $t$  is  $\beta$ ”;
- $\mathcal{N}_{\beta,t,\downarrow} = \{N_{\beta,u} \mid u \leq t\}$  – “Formula  $\beta$  not assumed until phase  $t$ ”;
- $\mathcal{N}_{\Gamma,t,\downarrow} = \{N_{\beta,u} \mid u \leq t \wedge \beta \notin \Gamma\}$  – “No formula outside of  $\Gamma$  assumed until phase  $t$ ”;
- $\mathcal{P}_{\Gamma,t} = \{P_{\gamma,t} \mid \gamma \in \Gamma\}$  – “Formulas in  $\Gamma$  assumed in phase  $t$ ”;
- $\mathcal{D}_{\Sigma,t} = \{D_{\beta,t} \mid \beta \in \Sigma\}$  – “Goals in  $\Sigma$  are forbidden in phase  $t$ ”.

The formula  $\bar{\varphi}$  to be defined has the form:

$$\bar{\varphi} = \Delta \rightarrow D_{\varphi,0},$$

where  $\Delta$  is the set consisting of the following implicational formulas (called *axioms*):

1.  $N_{\alpha,0}$ , for all  $\alpha \in \mathcal{S}$ ;
2.  $P_{\alpha,t} \rightarrow P_{\alpha,t+1}$ , for all  $\alpha \in \mathcal{S}$ ,  $t < n$ ;
3.  $(D_{\alpha,t} \rightarrow D'_{\alpha,t}) \rightarrow D_{\alpha,t}$ , for all  $\alpha \in \mathcal{S}$ , and all  $t \leq n$ ;
4.  $\mathcal{M}_{\alpha,t} \rightarrow D'_{\alpha,t}$ , for all  $\alpha \in \mathcal{S}$ , and all  $t \leq n$ , where the set  $\mathcal{M}_{\alpha,t}$  consists of the atoms:
  - $X_{\alpha,t}$ ;
  - $E_{\alpha,\perp,t}$ , and  $E_{\alpha,\beta \wedge \alpha,t}$ ,  $E_{\alpha,\alpha \wedge \beta,t}$ ,  $E_{\alpha,\beta \vee \gamma,t}$ ,  $E_{\alpha,\beta \rightarrow \alpha,t}$ , for all  $\beta, \gamma \in \mathcal{S}$ ;
  - $W_{\alpha,t}$ , in case  $\alpha$  is not an atom;

## 11:6 Duality in Intuitionistic Propositional Logic

5.  $\mathcal{N}_{\alpha,t\downarrow} \rightarrow \mathcal{X}_{\alpha,t}$ ;
6.  $\mathcal{D}_{\gamma,t} \rightarrow \mathcal{W}_{\alpha,t}$ , and  $\mathcal{D}_{\delta,t} \rightarrow \mathcal{W}_{\alpha,t}$ , for  $\alpha = \gamma \wedge \delta$ ;
7.  $\mathcal{D}_{\gamma,t} \rightarrow \mathcal{D}_{\delta,t} \rightarrow \mathcal{W}_{\alpha,t}$ , for  $\alpha = \gamma \vee \delta$ ;
8.  $\mathcal{W}_{\alpha,t}^1 \rightarrow \mathcal{W}_{\alpha,t}^2 \rightarrow \mathcal{W}_{\alpha,t}$ ,  $\mathcal{P}_{\gamma,t} \rightarrow \mathcal{W}_{\alpha,t}^1$ ,  $(\mathcal{A}_{\gamma,t+1} \rightarrow \mathcal{D}_{\delta,t+1}) \rightarrow \mathcal{W}_{\alpha,t}^1$ ,  $\mathcal{N}_{\gamma,t\downarrow} \rightarrow \mathcal{W}_{\alpha,t}^2$ ,  
and  $\mathcal{D}_{\delta,t} \rightarrow \mathcal{W}_{\alpha,t}^2$ , for  $\alpha = \gamma \rightarrow \delta$ ;
9.  $\mathcal{D}_{\perp,t} \rightarrow \mathcal{E}_{\alpha,\perp,t}$ ;
10.  $\mathcal{D}_{\alpha\wedge\beta,t} \rightarrow \mathcal{E}_{\alpha,\alpha\wedge\beta,t}$ , and  $\mathcal{D}_{\beta\wedge\alpha,t} \rightarrow \mathcal{E}_{\alpha,\beta\wedge\alpha,t}$ , for all  $\beta \in \mathcal{S}$ ;
11.  $\mathcal{D}_{\beta,t} \rightarrow \mathcal{E}_{\alpha,\beta\rightarrow\alpha,t}$ , and  $\mathcal{D}_{\beta\rightarrow\alpha,t} \rightarrow \mathcal{E}_{\alpha,\beta\rightarrow\alpha,t}$ , for all  $\beta \in \mathcal{S}$ ;
12.  $\mathcal{D}_{\gamma\vee\delta,t} \rightarrow \mathcal{E}_{\alpha,\gamma\vee\delta,t}$ ,  $\mathcal{P}_{\gamma,t} \rightarrow \mathcal{E}_{\alpha,\gamma\vee\delta,t}$ ,  $\mathcal{P}_{\delta,t} \rightarrow \mathcal{E}_{\alpha,\gamma\vee\delta,t}$ ,  $(\mathcal{A}_{\delta,t+1} \rightarrow \mathcal{D}_{\alpha,t+1}) \rightarrow \mathcal{E}_{\alpha,\gamma\vee\delta,t}$ , and  
 $(\mathcal{A}_{\gamma,t+1} \rightarrow \mathcal{D}_{\alpha,t+1}) \rightarrow \mathcal{E}_{\alpha,\gamma\vee\delta,t}$ , for all  $\delta, \gamma \in \mathcal{S}$ .

The main equivalence (\*) follows from Lemma 3 below, for  $\Gamma = \Sigma = \emptyset$ ,  $\alpha = \varphi$ , and  $t = 0$ . (Note that  $\mathcal{N}_{\emptyset,0\downarrow} \subseteq \Delta$ ,  $\mathcal{P}_{\emptyset,0} = \mathcal{D}_{\emptyset,0} = \emptyset$ .)

► **Lemma 3.** For  $|\Gamma| = t$ , and  $\alpha \notin \Sigma$ :

$$\Gamma \not\vdash \alpha [\Sigma] \quad \text{iff} \quad \Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t} \vdash \mathcal{D}_{\alpha,t}.$$

### Proof of Lemma 3

We begin with some additional notation and preparatory lemmas. Consider a set of atoms of shape  $\Xi = \mathcal{N}, \mathcal{P}, \mathcal{D}$ , where  $\mathcal{N}$ ,  $\mathcal{P}$ , and  $\mathcal{D}$  consist, respectively, of atoms of the form  $\mathcal{N}_{\alpha,u}$ ,  $\mathcal{P}_{\alpha,u}$ , and  $\mathcal{D}_{\alpha,u}$ . Write  $\max \Xi$  for the largest  $u$  such that a  $u$ -atom occurs in  $\Xi$ . For  $t = \max \Xi$ , define  $|\Xi|_t = |\mathcal{N}|_{t\downarrow}, |\mathcal{P}|_t, |\mathcal{D}|_t$ , where:

$$|\mathcal{N}|_{t\downarrow} = \bigcup \{ \mathcal{N}_{\alpha,t\downarrow} \mid \mathcal{N}_{\alpha,t\downarrow} \subseteq \mathcal{N} \}, \quad |\mathcal{P}|_t = \{ \mathcal{P}_{\alpha,t} \mid \exists u \leq t. \mathcal{P}_{\alpha,u} \in \mathcal{P} \}, \quad |\mathcal{D}|_t = \{ \mathcal{D}_{\alpha,t} \mid \mathcal{D}_{\alpha,t} \in \mathcal{D} \}.$$

The set  $|\Xi|_t$  consists of atoms (either occurring in  $\Xi$  or derivable from  $\Xi$ ) that are relevant towards a  $t$ -atomic proof goal, as stated in Lemma 5.

► **Lemma 4.** Let  $\Xi = \mathcal{N}, \mathcal{P}, \mathcal{D}$  be as above, with  $t = \max \Xi$ . If  $\Delta, \Xi \vdash \mathcal{P}_{\alpha,u}$ , for some  $u \leq t$ , then there is  $v \leq u$  such that  $\mathcal{P}_{\alpha,v} \in \mathcal{P}$ . In particular,  $\mathcal{P}_{\alpha,t} \in |\mathcal{P}|_t$ .

**Proof.** Induction with respect to the size of a normal term  $M$  such that  $\Delta, \Xi \vdash M : \mathcal{P}_{\alpha,u}$ . If  $M$  is a variable then  $v = u$ . Otherwise  $M = xN$ , where  $x$  is a variable of type  $\mathcal{P}_{\alpha,u-1} \rightarrow \mathcal{P}_{\alpha,u}$  and  $N$  has type  $\mathcal{P}_{\alpha,u-1}$ , as only axioms (2) have targets of the form  $\mathcal{P}_{\alpha,u}$ . We apply the induction hypothesis to  $N$ . ◀

We define the *weight* of a term  $M$  as the number of symbols in  $M$ , excluding parentheses and occurrences of variables of type (2).

► **Lemma 5.** Let  $\Xi = \mathcal{N}, \mathcal{P}, \mathcal{D}$  be as above, with  $t = \max \Xi$ , and let  $\mathcal{C}_t$  be a  $t$ -atom, not of the form  $\mathcal{N}_{\alpha,t}$ . If  $\Delta, \Xi \vdash M : \mathcal{C}_t$ , and  $M$  is normal, then also  $\Delta, |\Xi|_t \vdash M' : \mathcal{C}_t$ , where the weight of the proof  $M'$  does not exceed the weight of  $M$ .

**Proof.** Induction with respect to the weight of a normal term  $M$  such that  $\Delta, \Xi \vdash M : \mathcal{C}_t$ . Clearly,  $M$  must be an application of an assumption variable to zero, one or more arguments. If  $M$  is a variable (has no arguments), then  $\mathcal{C}_t \in \Xi$ . Then also  $\mathcal{C}_t \in |\Xi|_t$ , by definition (recall that atoms  $\mathcal{N}_{\alpha,t}$  are excluded), and we can take  $M' = M$ .

Otherwise let  $x$  be the head variable of  $M$ . The type of  $x$  is one of the axioms in  $\Delta$ . Assume first that  $x : \mathcal{P}_{\alpha,t-1} \rightarrow \mathcal{P}_{\alpha,t}$ . Then  $M$  has type  $\mathcal{P}_{\alpha,t}$ , whence  $\mathcal{P}_{\alpha,t} \in |\mathcal{P}|_t$ , by Lemma 4. We take the appropriate variable as  $M'$ .

Now  $M = x\vec{N}$ , for some vector  $\vec{N}$  of arguments. If types of these arguments are  $t$ -atoms in  $\Xi$ , other than  $\mathcal{N}_{\alpha,u}$ , then we apply the induction hypothesis to each component of  $\vec{N}$ .



Atoms  $N_{\alpha,u}$  only occur as arguments in the axioms:  $\mathcal{N}_{\alpha,t\downarrow} \rightarrow X_{\alpha,t}$  and  $\mathcal{N}_{\gamma,t\downarrow} \rightarrow W_{\gamma \rightarrow \delta, t}^2$ . If one of them is the type of  $x$  then  $\mathcal{N}_{\alpha,t\downarrow}$  (resp.  $\mathcal{N}_{\gamma,t\downarrow}$ ) must be a subset of  $\Xi$ , more precisely a subset of  $\mathcal{N}$ . But then it is actually a subset of  $|\mathcal{N}|_{t\downarrow}$ , so we can take  $M' = M$ .

There are four cases when an assumption of an axiom is not an atom. The case of axiom (3) is simple: then  $M = x(\lambda y N)$  with  $y : D_{\alpha,t}$  and  $N : D'_{\alpha,t}$ . In this case we apply the induction hypothesis to  $\Delta, \Xi, D_{\alpha,t} \vdash N : D'_{\alpha,t}$ .

A less obvious case is when the head variable of  $M$  has e.g. type  $(\mathcal{A}_{\gamma,t+1} \rightarrow D_{\delta,t+1}) \rightarrow W_{\alpha,t}^1$ . Then we have  $\Delta, \Xi, \mathcal{A}_{\gamma,t+1} \vdash N : D_{\delta,t+1}$ , for a subterm  $N$  of  $M$ , to which we apply the induction hypothesis. This yields

$$\Delta, |\mathcal{N}, \mathcal{N}'|_{(t+1)\downarrow}, |\mathcal{P}, P_{\gamma,t+1}|_{t+1}, |\mathcal{D}|_{t+1} \vdash N' : D_{\delta,t+1}$$

where  $\mathcal{N}' = \{N_{\alpha,t+1} \mid \alpha \in \mathcal{S} \wedge \alpha \neq \gamma\}$ . We want to show  $\Delta, |\Xi|_t, \mathcal{A}_{\gamma,t+1} \vdash N'' : D_{\delta,t+1}$ , that is,

$$\Delta, |\mathcal{N}|_{t\downarrow}, |\mathcal{P}|_t, |\mathcal{D}|_t, \mathcal{A}_{\gamma,t+1} \vdash N'' : D_{\delta,t+1}. \quad (\dagger)$$

First observe that  $|\mathcal{D}|_{t+1} = \emptyset \subseteq |\mathcal{D}|_t$ . We also have  $|\mathcal{N}, \mathcal{N}'|_{(t+1)\downarrow} \subseteq |\mathcal{N}|_{t\downarrow} \cup \mathcal{A}_{\gamma,t+1}$ . Indeed, if  $N_{\alpha,u} \in |\mathcal{N}, \mathcal{N}'|_{(t+1)\downarrow}$  then the whole set  $\mathcal{N}_{\alpha,(t+1)\downarrow}$  is contained in  $\mathcal{N} \cup \mathcal{N}'$ . Thus, for all  $v$ , if  $v \leq t$  then  $N_{\alpha,v} \in \mathcal{N}$ , and if  $v = t+1$  then  $N_{\alpha,v} \in \mathcal{N}'$ , in particular  $\alpha \neq \gamma$ . It follows that  $N_{\alpha,u} \in |\mathcal{N}|_t$ , for  $u \leq t$ , and  $N_{\alpha,u} \in \mathcal{A}_{\gamma,t+1}$  when  $u = t+1$ .

However, it is not the case that  $|\mathcal{P}, P_{\gamma,t+1}|_{t+1} \subseteq |\mathcal{P}|_t, \mathcal{A}_{\gamma,t+1}$ , so we cannot take  $N'' = N'$ . But the missing atoms  $P_{\beta,t+1}$  are derivable (for free) from  $P_{\beta,t} \in |\mathcal{P}|_t$ . That is, we can replace in  $N'$  any occurrence of a variable of type  $P_{\beta,t+1}$  by an application of one or more axioms of type (2) without adding more weight. We conclude that the judgment  $(\dagger)$  holds with some term  $N''$  of the same weight as  $N'$ .  $\blacktriangleleft$

**Proof of Lemma 3 (“if” part).** Assume  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t} \vdash D_{\alpha,t}$ , where  $|\Gamma| = t$ , and  $\alpha \notin \Sigma$ . Suppose towards contradiction that  $\Gamma \vdash \alpha[\Sigma]$ . We proceed by induction with respect to the weight of a normal proof of  $D_{\alpha,t}$ . Of course  $D_{\alpha,t} \notin \mathcal{D}_{\Sigma,t}$ , so the proof is an application of the axiom  $(D_{\alpha,t} \rightarrow D'_{\alpha,t}) \rightarrow D_{\alpha,t}$ . Hence  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t}, D_{\alpha,t} \vdash D'_{\alpha,t}$ , and this means that  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t} \vdash C_t$ , for every atom  $C_t \in \mathcal{M}_t$ . Indeed, no other axiom targets  $D'_{\alpha,t}$ , and we have  $\mathcal{D}_{\Sigma,t}, D_{\alpha,t} = \mathcal{D}_{\Sigma,\alpha,t}$ .

In particular,  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t} \vdash X_t$ , and that can only happen when  $\mathcal{N}_{\alpha,t\downarrow} \subseteq \mathcal{N}_{\Gamma,t\downarrow}$ , i.e., when  $\alpha \notin \Gamma$ . It follows that the judgment  $\Gamma \vdash \alpha[\Sigma]$  is not an axiom.

Since  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t} \vdash E_{\alpha,\perp,t}$ , it must be the case that  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t} \vdash D_{\perp,t}$ , whence  $\Gamma \not\vdash \perp[\Sigma, \alpha]$  by induction. Thus,  $\Gamma \vdash \alpha[\Sigma]$  cannot be obtained from rule  $(E_{\perp})$ .

Suppose  $\Gamma \vdash \alpha[\Sigma]$  is derived using  $(E_{\wedge_1})$  in the last step. Then  $\Gamma \vdash \alpha \wedge \beta[\Sigma, \alpha]$ , for some  $\beta$  such that  $\alpha \wedge \beta \notin \Sigma$ . But  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t} \vdash E_{\alpha,\alpha \wedge \beta,t}$  implies that  $D_{\alpha \wedge \beta,t}$  is derivable, whence  $\Gamma \not\vdash \alpha \wedge \beta[\Sigma, \alpha]$ , by the induction hypothesis.

In a similar fashion we exclude all rules where  $\Gamma$  remains unchanged in the premises. Let us consider rule  $(E_{\vee})$ . For any  $\gamma, \delta$  we have a proof of  $E_{\alpha,\gamma \vee \delta}$  using one of the five available axioms (12) targeting this atom.

Suppose that  $E_{\alpha,\gamma \vee \delta}$  was derived using the axiom  $D_{\gamma \vee \delta,t} \rightarrow E_{\alpha,\gamma \vee \delta,t}$ . This means that  $D_{\gamma \vee \delta,t}$  was proved in the same environment. If  $\gamma \vee \delta \notin \Sigma$  then  $\Gamma \not\vdash \gamma \vee \delta[\Sigma, \alpha]$ , by induction, and rule  $(E_{\vee})$  is not applicable. This rule is also excluded when  $\gamma \vee \delta \in \Sigma$ .

Axiom  $P_{\gamma,t} \rightarrow E_{\alpha,\gamma \vee \delta,t}$  can be used only if  $P_{\gamma,t} \in \mathcal{P}_{\Gamma,t}$ , i.e., when  $\gamma \in \Gamma$ . Then rule  $(E_{\vee})$  is not applicable too (and similarly for  $\delta$ ). So  $\gamma, \delta \notin \Gamma$  and we must have used e.g. the axiom  $(\mathcal{A}_{\delta,t+1} \rightarrow D_{\alpha,t+1}) \rightarrow E_{\alpha,\gamma \vee \delta,t}$ . It follows that  $M$  is of shape  $y(\lambda \vec{x} N)$ , where

$$\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t}, D_{\alpha,t}, \mathcal{A}_{\delta,t+1} \vdash N : D_{\alpha,t+1}.$$

## 11:8 Duality in Intuitionistic Propositional Logic

From Lemma 5 we obtain:

$$\Delta, |\mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t}, \mathbf{D}_{\alpha,t}, \mathcal{A}_{\delta,t+1}|_{t+1} \vdash N' : \mathbf{D}_{\alpha,t+1},$$

that is:

$$\Delta, |\mathcal{N}_{\Gamma,t\downarrow} \cup \{\mathbf{N}_{\beta,t+1} \mid \beta \neq \delta\}|_{(t+1)\downarrow}, |\mathcal{P}_{\Gamma,t}, \mathbf{P}_{\delta,t+1}|_{t+1}, |\mathcal{D}_{\Sigma,t}, \mathbf{D}_{\alpha,t}|_{t+1} \vdash N' : \mathbf{D}_{\alpha,t+1}.$$

Now we should observe that:

- $|\mathcal{D}_{\Sigma,t}, \mathbf{D}_{\alpha,t}|_{t+1} = \emptyset = \mathcal{D}_{\emptyset,t+1}$ ;
- $|\mathcal{N}_{\Gamma,t\downarrow} \cup \{\mathbf{N}_{\beta,t+1} \mid \beta \neq \delta\}|_{(t+1)\downarrow} \subseteq \mathcal{N}_{\Gamma,\delta,(t+1)\downarrow}$ ;
- $|\mathcal{P}_{\Gamma,t}, \mathbf{P}_{\delta,t+1}|_{t+1} = \mathcal{P}_{\Gamma,\delta,t+1}$ .

Therefore, we can write:

$$\Delta, \mathcal{N}_{\Gamma,\delta,(t+1)\downarrow}, \mathcal{P}_{\Gamma,\delta,t+1}, \mathcal{D}_{\emptyset,t+1} \vdash N' : \mathbf{D}_{\alpha,t+1}.$$

Since  $N'$  is smaller than  $M$  in weight, we have  $\Gamma, \delta \not\vdash \alpha[\emptyset]$  by the induction hypothesis, and rule (E $\vee$ ) is now excluded too. In other cases we proceed in an analogous way. ◀

**Proof of Lemma 3 (“only if” part).** Let  $\Gamma \not\vdash \alpha[\Sigma]$ . We prove  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t} \vdash \mathbf{D}_{\alpha,t}$ , by induction with respect to two parameters:

- (1) the cardinality  $n - t$  of  $\mathcal{S} - \Gamma$ ,      (2) the cardinality of  $\mathcal{S} - \Sigma$ .

We need to show that all atoms in  $\mathcal{M}_{\alpha,t}$  can be derived from  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t}, \mathbf{D}_{\alpha,t}$ , which is the same as  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t}$ . Then we can obtain  $\mathbf{D}_{\alpha,t}$  using the axioms  $\mathcal{M}_{\alpha,t} \rightarrow \mathbf{D}'_{\alpha,t}$  and  $(\mathbf{D}_{\alpha,t} \rightarrow \mathbf{D}'_{\alpha,t}) \rightarrow \mathbf{D}_{\alpha,t}$ .

We begin with  $\mathbf{X}_{\alpha,t}$ . From  $\Gamma \not\vdash \alpha[\Sigma]$  it follows that  $\alpha \notin \Gamma$ , hence all formulas  $\mathbf{N}_{\alpha,u}$ , for  $u \leq t$ , are in  $\mathcal{N}_{\Gamma,t\downarrow}$ , and the axiom  $\mathcal{N}_{\alpha,t\downarrow} \rightarrow \mathbf{X}_{\alpha,t}$  can be used to prove  $\mathbf{X}_{\alpha,t}$ .

In order to prove  $\mathbf{E}_{\alpha,\perp,t}$ , we consider two cases. If  $\perp \notin \Sigma$  then we observe that  $\Gamma \not\vdash \perp[\Sigma, \alpha]$ , as otherwise rule (E $\perp$ ) could be used to derive  $\Gamma \vdash \alpha[\Sigma]$ . By the induction hypothesis we have  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\alpha,t} \vdash \mathbf{D}_{\perp,t}$ , because the cardinality of  $\Gamma$  is unchanged and the cardinality of  $\Sigma, \alpha$  is greater by one than that of  $\Sigma$ . Our goal is accomplished with the axiom  $\mathbf{D}_{\perp,t} \rightarrow \mathbf{E}_{\alpha,\perp,t}$ . The case  $\perp \in \Sigma$  is simpler, because then  $\mathbf{D}_{\perp,t}$  just belongs to  $\mathcal{D}_{\Sigma,t}$ .

Consider a constant  $\mathbf{E}_{\alpha,\beta \rightarrow \alpha,t}$ . If  $\beta \in \Sigma$  or  $\beta \rightarrow \alpha \in \Sigma$  then  $\mathbf{D}_{\beta,t} \in \mathcal{D}_{\Sigma,t}$  or  $\mathbf{D}_{\beta \rightarrow \alpha,t} \in \mathcal{D}_{\Sigma,t}$ , and  $\mathbf{E}_{\alpha,\beta \rightarrow \alpha,t}$  follows easily. Otherwise, one of the premises of rule (E $\rightarrow$ ) does not hold, and one can apply the induction hypothesis to derive either  $\mathbf{D}_{\beta,t}$  or  $\mathbf{D}_{\beta \rightarrow \alpha,t}$ . The induction could fail when  $\beta = \alpha$ , in which case  $\beta \in \Sigma, \alpha$ . But then we already have  $\mathbf{D}_{\beta,t} = \mathbf{D}_{\alpha,t} \in \mathcal{D}_{\Sigma,t}$ .

As the next example we consider the atom  $\mathbf{W}_{\alpha}$ , where  $\alpha = \gamma \rightarrow \delta$ . We should derive the two atoms  $\mathbf{W}_{\alpha,t}^1$  and  $\mathbf{W}_{\alpha,t}^2$ .

If  $\gamma \in \Gamma$  then  $\mathbf{P}_{\gamma,t} \in \mathcal{P}_{\alpha,t}$  and  $\mathbf{P}_{\gamma,t} \rightarrow \mathbf{W}_{\alpha,t}^1$  implies  $\mathbf{W}_{\alpha,t}^1$ . Otherwise  $\Gamma, \gamma \not\vdash \delta[\emptyset]$ , as  $\gamma \rightarrow \delta$  should not be derivable using rule (W $\rightarrow_1$ ). We can apply the induction hypothesis, because the set  $\Gamma, \gamma$  is larger than  $\Gamma$ . Thus,  $\Delta, \mathcal{N}_{\Gamma,\gamma,(t+1)\downarrow}, \mathcal{P}_{\Gamma,\gamma,t+1}, \mathcal{D}_{\emptyset,t+1} \vdash \mathbf{D}_{\delta,t+1}$ . Observe that  $\mathcal{N}_{\Gamma,\gamma,(t+1)\downarrow} \subseteq \mathcal{N}_{\Gamma,t\downarrow} \cup \mathcal{A}_{\gamma,t+1}$ , and that all atoms  $\mathbf{P}_{\sigma,t+1} \in \mathcal{P}_{\Gamma,\gamma,t+1}$  can be derived from the set  $\mathcal{P}_{\Gamma,t}, \mathcal{A}_{\gamma,t+1}$ , because  $\mathbf{P}_{\gamma,t+1} \in \mathcal{A}_{\gamma,t+1}$ . Therefore  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,t}, \mathbf{D}_{\alpha,t}, \mathcal{A}_{\gamma,t+1} \vdash \mathbf{D}_{\delta,t+1}$ . Using the axiom  $(\mathcal{A}_{\gamma,t+1} \rightarrow \mathbf{D}_{\delta,t+1}) \rightarrow \mathbf{W}_{\alpha,t}^1$  we obtain what we need.

The atom  $\mathbf{W}_{\gamma \rightarrow \delta,t}^2$  is easily derived from  $\mathbf{D}_{\delta,t}$  in case  $\delta \in \Sigma$ . Similarly, if  $\gamma \notin \Gamma$  then we can use the axiom  $\mathcal{N}_{\gamma,t\downarrow} \rightarrow \mathbf{W}_{\alpha,t}^2$ . So we assume  $\delta \notin \Sigma$ ,  $\gamma \in \Gamma$  and we apply induction to  $\Gamma \not\vdash \delta[\Sigma, \gamma \rightarrow \delta]$ . This yields  $\Delta, \mathcal{N}_{\Gamma,t\downarrow}, \mathcal{P}_{\Gamma,t}, \mathcal{D}_{\Sigma,\gamma \rightarrow \delta,t} \vdash \mathbf{D}_{\delta,t}$ , and we use the axiom  $\mathbf{D}_{\delta,t} \rightarrow \mathbf{W}_{\alpha,t}^2$  to complete the job. Other cases are similar. ◀

## A simple example

For a formula  $\varphi$  of length  $n$ , the “dual” formula  $\bar{\varphi}$  is of size  $\mathcal{O}(n^3)$  with a decently large constant, and may be quite incomprehensible even for short  $\varphi$ . We therefore consider an extremely simple example  $\varphi = (p \rightarrow p) \rightarrow p$ . By our definition we have  $\bar{\varphi} = \bar{\Delta} \rightarrow \mathcal{D}_{\varphi,0}$ , where  $\bar{\Delta}$  abbreviates the sequence of all axioms (1–12). Not all of them are actually needed, and some can be simplified in this case. For example, a normal proof of  $\varphi$  itself cannot be an elimination, because only subformulas of  $\varphi$  are used and neither  $\perp$  or  $\vee$  occurs in  $\varphi$ . Hence the set  $\mathcal{M}_{\varphi,t}$  in (4) reduces to two elements (cf. type of  $X_4$  below). Here we only list the relevant part of  $\bar{\Delta}$ , in the form of variable declarations. We use the abbreviation  $\alpha = p \rightarrow p$ .

1.  $X_1 : \mathbf{N}_{\varphi,0}, Z_1 : \mathbf{N}_{\alpha,0}, Y_1 : \mathbf{N}_{p,0};$
3.  $X_3 : (\mathbf{D}_{\varphi,0} \rightarrow \mathbf{D}'_{\varphi,0}) \rightarrow \mathbf{D}_{\varphi,0}, Y_3 : (\mathbf{D}_{p,1} \rightarrow \mathbf{D}'_{p,1}) \rightarrow \mathbf{D}_{p,1}, U_3 : (\mathbf{D}_{\varphi,1} \rightarrow \mathbf{D}'_{\varphi,1}) \rightarrow \mathbf{D}_{\varphi,1};$
4.  $X_4 : \mathbf{X}_{\varphi,0} \rightarrow \mathbf{W}_{\varphi,0} \rightarrow \mathbf{D}'_{\varphi,0}, Y_4 : \mathbf{X}_{p,1} \rightarrow \mathbf{E}_{p,\varphi,1} \rightarrow \mathbf{E}_{p,\alpha,1} \rightarrow \mathbf{D}'_{p,1}, U_4 : \mathbf{X}_{\varphi,1} \rightarrow \mathbf{W}_{\varphi,1} \rightarrow \mathbf{D}'_{\varphi,1};$
5.  $X_5 : \mathbf{N}_{\varphi,0} \rightarrow \mathbf{X}_{\varphi,0}, Y_5 : \mathbf{N}_{p,0} \rightarrow \mathbf{N}_{p,1} \rightarrow \mathbf{X}_{p,1}, U_5 : \mathbf{N}_{\varphi,0} \rightarrow \mathbf{N}_{\varphi,1} \rightarrow \mathbf{X}_{\varphi,1};$
8.  $X_8^1 : \mathbf{W}_{\varphi,0}^1 \rightarrow \mathbf{W}_{\varphi,0}^2 \rightarrow \mathbf{W}_{\varphi,0}, X_8^3 : (\mathbf{P}_{\alpha,1} \rightarrow \mathbf{N}_{\varphi,1} \rightarrow \mathbf{N}_{p,1} \rightarrow \mathbf{D}_{p,1}) \rightarrow \mathbf{W}_{\varphi,0}^1, X_8^4 : \mathbf{N}_{\alpha,0} \rightarrow \mathbf{W}_{\varphi,0}^2,$   
 $U_8^1 : \mathbf{W}_{\varphi,1}^1 \rightarrow \mathbf{W}_{\varphi,1}^2 \rightarrow \mathbf{W}_{\varphi,1}, U_8^2 : \mathbf{P}_{\alpha,1} \rightarrow \mathbf{W}_{\varphi,1}^1, U_8^5 : \mathbf{D}_{p,1} \rightarrow \mathbf{W}_{\varphi,1}^2.$
11.  $Y_{11}^1 : \mathbf{D}_{p,1} \rightarrow \mathbf{E}_{p,\alpha,1}, Y_{11}^2 : \mathbf{D}_{\varphi,1} \rightarrow \mathbf{E}_{p,\varphi,1}.$

A proof of  $\mathbf{D}_{\varphi,0}$  can now be presented as the lambda-term:

$$X_3(\lambda x:\mathbf{D}_{\varphi,0}. X_4(X_5 X_1)(X_8^1 T (X_8^4 Z_1))),$$

where  $T = X_8^3(\lambda w:\mathbf{P}_{\alpha,1} \lambda x_1:\mathbf{N}_{\varphi,1} \lambda y_1:\mathbf{N}_{p,1}. Y_3(\lambda y:\mathbf{D}_{p,1}. Y_4(Y_5 Y_1 y_1) S (Y_{11}^1 y)))$  has type  $\mathbf{W}_{\varphi,0}^1$ , and  $S = Y_{11}^2(U_3(\lambda u:\mathbf{D}_{\varphi,1}. U_4(U_5 X_1 x_1)(U_8^1(U_8^2 w)(U_8^5 y))))$  has type  $\mathbf{E}_{p,\varphi,1}$ .

The above term represents the following refutation of  $\varphi$ . First check that  $\varphi$  is not assumed (this is the meaning of the subterm  $X_5 X_1$ ). Then check that  $\varphi$  cannot be obtained by introduction from  $\alpha \vdash p$ . Since  $\alpha$  is not yet assumed ( $X_8^4 Z_1$ ) we now assume it (variable  $w$ ) but not the other formulas (variables  $x_1, y_1$ ). The goal  $p$  is now addressed for the first time and is marked as forbidden in this phase (variable  $y$ ). It is easy to check that  $p$  is not an assumption ( $Y_5 Y_1 y_1$ ) and that it cannot be derived by elimination from  $\alpha$ : indeed, the latter requires re-addressing the goal  $p$  in the same environment ( $Y_{11}^1 y$ ).

The subterm  $S$  refutes the possibility that  $p$  is obtained by elimination from  $\varphi$ , because an attempt to derive  $\alpha \vdash \varphi$  will fail. Indeed,  $\varphi$  must be obtained by introduction from  $\alpha \vdash p$  (the subterms  $U_8^2 w$  and  $U_5 X_1 x_1$  certify that  $\alpha$  has already been assumed, but  $\varphi$  has not). But  $p$  is a forbidden goal ( $U_8^5 y$ ), hence using the introduction rule is illegal.

## Conclusion

We have demonstrated a logarithmic space algorithm to construct a “dual” formula  $\bar{\varphi}$  for any given propositional formula  $\varphi$ , so that  $\bar{\varphi}$  is provable in IPC if  $\varphi$  is not. The construction is inspired by an automata-theoretic view of proof-search. This can be seen as alternative to introducing rules to derive refutability: just apply the old ones towards a different task.

The formula  $\bar{\varphi}$  uses only implication and (as a simple type) is of order (depth) at most 3. Since  $\bar{\varphi}$  is provable iff so is  $\varphi$ , we conclude that IPC provability reduces to provability of formulas of particularly simple form.<sup>2</sup> The formula  $\bar{\varphi}$  is not equivalent to  $\varphi$ , but is computable in logarithmic space (note the analogy with CNF-SAT).

<sup>2</sup> Of course that can be done much simpler in a direct way [8].

Intuitionistic propositional logic can represent combinatorial problems as easily (or better) as classical propositional satisfiability, and it is far more expressive because it reaches beyond the class NP. Provability in IPC reduces to the case of order three. Those should be relatively easy to simplify and manipulate by various heuristics (like joining and deleting some formula components). It is about time for an intuitionistic analogue of Davis-Putnam algorithm. This issue is raised in a subsequent work [8].

---

### References

- 1 Philippe de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Information and Computation*, 178(2):441–464, 2002.
- 2 Camillo Fiorentini and Mauro Ferrari. A forward unprovability calculus for intuitionistic propositional logic. In R. A. Schmidt and C. Nalon, editors, *Proc. TABLEAUX 2017*, volume 10501 of *LNAI*, pages 114–130. Springer, 2017.
- 3 D. Leivant. Assumption classes in natural deduction. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 25:1–4, 1979.
- 4 Luis Pinto and Roy Dyckhoff. Loop-free construction of counter-models for intuitionistic propositional logic. In R. Behara, M. Fritsch and R.G. Lintz, editors, *Symposia Gaussiana, Conf. A, 1993*, pages 225–232. De Gruyter, 1995.
- 5 Sylvain Schmitz. Implicational relevance logic is 2-EXPTIME-complete. *J. Symb. Log.*, 81:641–661, 2016.
- 6 Aleksy Schubert, Wil Dekkers, and Hendrik Pieter Barendregt. Automata theoretic account of proof search. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPICs*, pages 128–143. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015.
- 7 Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. How hard is positive quantification? *ACM Trans. Comput. Log.*, 17(4):30:1–30:29, 2016.
- 8 Aleksy Schubert, Paweł Urzyczyn, and Konrad Zdanowski. Between proof construction and SAT-solving, 2021. To appear.
- 9 Tomasz Skura. Refutation systems in propositional logic. In D.M. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 16, pages 115–157. Springer, 2011.
- 10 Paweł Urzyczyn. Intuitionistic games: Determinacy, completeness, and normalization. *Studia Logica*, 104(5):957–1001, 2016.