19th International Conference on Types for Proofs and Programs

TYPES 2013, April 22–26, 2013, Toulouse, France

Edited by

Ralph Matthes

Aleksy Schubert

Editors

Ralph Matthes
IRIT
CNRS and Université de Toulouse
France
matthes@irit.fr

Aleksy Schubert
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw
Poland
alx@mimuw.edu.pl

ACM Classification 1998
D.1.1 Applicative (Functional) Programming, D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical Logic

ISBN 978-3-939897-72-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-72-9.

Publication date
July, 2014

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

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2013.i

ISBN 978-3-939897-72-9 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert

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

Preface

The 19th International Conference on Types for Proofs and Programs (TYPES 2013) was held in Toulouse, France from April 22 to 26, 2013, consisting of the main conference and several satellite events.

The following institutions helped with funding and/or in providing lecture halls and services that TYPES 2013 could take place (in alphabetic order):

- Institut de Recherche en Informatique de Toulouse (IRIT), `http://www.irit.fr/`
- Région Midi-Pyrénées, `http://www.midipyrenees.fr/`
- So Toulouse (an activity of Mairie de Toulouse), `http://www.sotoulouse.com/`
- Structure Fédérative de Recherche en Mathématiques et en Informatique de Toulouse (FREMIT), `http://www.irit.fr/FREMIT/`
- Université Paul Sabatier, Toulouse III, `http://www.univ-tlse3.fr/`
- Université Toulouse 1 Capitole, `http://www.ut-capitole.fr/`

The conference was attended by close to hundred scientists. The responsible person for local organisation was Ralph Matthes, and the program committee for the selection of the conference presentations consisted of

José Espírito Santo, University of Minho, Braga, Portugal,
Herman Geuvers, Radboud University Nijmegen, Netherlands,
Silvia Ghilezan, University of Novi Sad, Serbia,
Hugo Herbelin, PPS, INRIA Rocquencourt-Paris, France,
Martin Hofmann, Ludwig-Maximilians-Universität München, Germany,
Zhaohui Luo, Royal Holloway, University of London, UK,
Ralph Matthes, IRIT, CNRS and Université de Toulouse, France (co-chair),
Marino Miculan, University of Udine, Italy,
Bengt Nordström, Chalmers University of Technology, Göteborg, Sweden,
Erik Palmgren, Stockholm University, Sweden,
Andy Pitts, University of Cambridge, UK,
Sergei Soloviev, IRIT, Université de Toulouse, France (co-chair),
Paweł Urzyczyn, University of Warsaw, Poland,
Tarmo Uustalu, Institute of Cybernetics, Tallinn Technical University, Estonia.

The TYPES meetings were first organised in the late 1980's and were supported by a series of EU programmes from 1989 to 2008. Previous meetings were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal (2002), Turin (2003), Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Turin (2008), Aussois (2009), Warsaw (2010) and Bergen (2011).

Three invited talks and 34 contributed talks were given at the meeting, and we got 22 submissions to these open post-proceedings, out of which 13 papers were accepted. In this volume one can find papers on the following topics: analysis of the classical principles in intuitionistic calculi, type isomorphisms for intersection types, monads and their semantics in functional programming languages, realizability, extensions of type theory, extensions of linear logic, models of type theory, control operators in type systems, formal verification of programs, program extraction, compiler formalization and modelling of natural language features. All papers obtained at least two reviews, and up to six reviews, counting a second round of review.

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As editors of this post-proceedings volume, we would like to thank the authors of the paper submissions, whether accepted or not. And we gratefully acknowledge all 43 anonymous external referees for their valuable work. The overall very high quality of the resulting papers is in part due to their careful reading and commenting on the obtained material.

Concerning the handling, we owe thanks to Andrej Voronkov and the support team of EasyChair for this tool that is still quite helpful for preparing post-proceedings, and also to Marc Herbstritt whose competent and friendly guidance made it a very agreeable experience to work with Schloss Dagstuhl as editors of these post-proceedings.

June 2014, Ralph Matthes, Aleksy Schubert

Author Index

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert

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

Update Monads: Cointerpreting Directed Containers*

Danel Ahman[1] and Tarmo Uustalu[2]

1 **Laboratory for Foundations of Computer Science, University of Edinburgh**
10 Crichton Street, Edinburgh EH8 9AB, United Kingdom
d.ahman@ed.ac.uk

2 **Institute of Cybernetics at Tallinn University of Technology**
Akadeemia tee 21, 12618 Tallinn, Estonia
tarmo@cs.ioc.ee

Abstract

We introduce update monads as a generalization of state monads. Update monads are the compatible compositions of reader and writer monads given by a set and a monoid. Distributive laws between such monads are given by actions of the monoid on the set.

We also discuss a dependently typed generalization of update monads. Unlike simple update monads, they cannot be factored into a reader and writer monad, but rather into similarly looking relative monads.

Dependently typed update monads arise from cointerpreting directed containers, by which we mean an extension of an interpretation of the opposite of the category of containers into the category of set functors.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, F.3.3 Studies of Program Constructs

Keywords and phrases monads and distributive laws, reader and writer and state monads, monoids, monoid actions, directed containers

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.1

1 Introduction

In denotational semantics and functional programming, reader, writer and state monads [15] are well known and important. They are related to each other, but there is also something that may feel unsatisfactory: reader and writer monads are not instances of state monads and state monads are not combinations of reader and writer monads.

In this paper we introduce a generalization of state monads, which we call update monads, that overcome exactly this underachievement. Update monads are compatible compositions of reader and writer monads, they are specified by a set, a monoid and an action, defining a reader monad, a writer monad and a distributive law of the latter over the former. They collect computations that take an initial state to pair of an update (that is not applied!) and a return value.

We also discuss a dependently typed generalization of update monads. Dependently typed update monads arise from cointerpreting directed containers, by which we mean an extension

* This work was supported by the University of Edinburgh Principal's Career Development PhD Scholarship, the ERDF funded Estonian CoE project EXCS and ICT programme project Coinduction, the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12 and the Estonian Science Foundation grant no. 9475.

of an interpretation of the opposite of the category of containers into the category of set functors. Directed containers [2] are a description of comonoids in the category of containers and characterize those containers whose interpretation carries a comonad structure. In a directed container, each state has its own set of updates that are considered "safe" for that state.

The paper is organized as follows. We begin Section 2 by recapitulating reader, writer and state monads. We then introduce update monads, show that reader and writer monads are instances of update monads and that state monads related to them in an important way; we also discuss algebras of update monads. Next, we show that update monads are compatible compositions of reader and writer monads, which leads to a different characterization of their algebras. We also briefly discuss update monad maps. In Section 3, we look at the dependently typed generalization of update monads that results from cointerpreting directed containers. In Section 4, we compare update monads to a generalization of state monads by Kammar and Plotkin.

For self-containedness of the paper, in Appendix A, we review monoids, actions, monads and compatible compositions of monads. In Appendix B, we give a detailed proof of the main theorem of the paper. In Appendix C, we show how algebras of update monads can be described as models of Lawvere theories.

In this paper we develop the theory of update monads over the category **Set** of sets and functions. The development in Section 2 can be easily generalized to arbitrary Cartesian closed categories. The development in Section 3 can be similarly carried out in locally Cartesian closed categories.

2 Unifying reader, writer, state monads

2.1 Reader, writer, state monads

We recall the three "classic" families of monads of functional programming [15]—the reader, writer and state monads.

Reader monads

Every set S (of states) defines a monad (the *reader monad*) via

$$T\,X = S \to X$$

$$\eta : \forall\{X\}.\, X \to S \to X$$
$$\eta\, x = \lambda s.\, x$$

$$\mu : \forall\{X\}.\, (S \to (S \to X)) \to S \to X$$
$$\mu\, f = \lambda s.\, f\, s\, s$$

Here and in the following, we use Agda's [12] syntax of braces for implicit arguments, i.e., for those arguments we may want to skip when they are inferrable from other arguments.

Writer monads

Every monoid $(P, \mathsf{o}, \oplus)$ (of updates) defines a monad (the *writer monad*, also sometimes called the *complexity* monad) via

$$T\,X = P \times X$$

$$\eta : \forall\{X\}.\, X \to P \times X$$
$$\eta\, x = (\mathsf{o}, x)$$

$$\mu : \forall\{X\}.\, P \times (P \times X) \to P \times X$$
$$\mu\,(p, (p', x)) = (p \oplus p', x)$$

Note that we would not be able to do with just a set P to get a monad on this underlying functor, we need both the unit and multiplication of the monoid to the define the unit and multiplication and the monoid laws to prove the monad laws.[1]

State monads

Every set S defines a monad (the *state monad*) via

$$T\,X = S \to S \times X$$

$$\eta : \forall\{X\}.\, X \to S \to S \times X$$
$$\eta\, x = \lambda s.\, (s, x)$$

$$\mu : \forall\{X\}.\, (S \to S \times (S \to S \times X)) \to S \to S \times X$$
$$\begin{aligned}\mu\, f = \lambda s.\, &\mathsf{let}\ (s', g) = f\ s;\\ &\quad\ (s'', x) = g\ s'\\ &\mathsf{in}\ (s'', x)\end{aligned}$$

The usual explanation of the state monad is the following. $T\,X = S \to S \times X$ is the set of computations each taking an initial state to a pair of a final state and a return value.

Unifying the three?

Notice the similarities between these monads. The reader monad resembles the state monad, when we ignore the final, mutated, state. The writer monad resembles the state monad, when we ignore the initial state and require the final state to have monoidal structure. (This said, in the case of the writer monad, the values written accumulate, but in the case of the state monad, they replace each other.) Could we possibly unify the three properly? We can. We do this in the next section using monoid actions.

2.2 Update monads

Given a set S and a monoid $(P, \mathsf{o}, \oplus)$ together with a right action $\downarrow\colon S \times P \to S$ of the monoid on the set (in this paper we call such a triple a *(right) act*[2]), we are interested in the

[1] The monoid structure on P induces also a different non-isomorphic monad defined by $T^{\mathrm{r}}X = X \times P$, $\eta^{\mathrm{r}}\, x = (x, \mathsf{o})$, $\mu^{\mathrm{r}}\,((x, p), p') = (x, p \oplus p')$. The order of P and X in the product $T^{\mathrm{r}}\,X$ is not important here, as product is symmetric. But μ^{r} adds the two given monoid elements in the order reverse to μ.

[2] A set that a fixed monoid $(P, \mathsf{o}, \oplus)$ acts on is often called a $(P, \mathsf{o}, \oplus)$-set or a $(P, \mathsf{o}, \oplus)$-act. We are interested in varying both the set S and the monoid $(P, \mathsf{o}, \oplus)$ at the same time.

following monad, which we call the *update monad* for the act $(S, (P, \mathsf{o}, \oplus), \downarrow)$.

$$T\,X = S \to P \times X$$

$$\eta : \forall\{X\}.\, X \to S \to P \times X$$
$$\eta\, x = \lambda s.\, (\mathsf{o}, x)$$

$$\mu : \forall\{X\}.\, (S \to P \times (S \to P \times X)) \to S \to P \times X$$
$$\begin{aligned}\mu\, f = \lambda s.\, &\mathsf{let}\ (p, g) = f\, s;\\ &\quad\ (p', x) = g\, (s \downarrow p)\\ &\mathsf{in}\ (p \oplus p', x)\end{aligned}$$

A computation over X, i.e., an element of $T\,X$, is a function taking an initial state to an update produced and a return value. Notice that rather than returning the result of applying the update to the initial state, i.e., the final state, the function returns the actual update itself. These updates are only ever applied by the multiplication μ. This operation applies to the initial state s the update p defined by s, in order to thus obtain a new state $s \downarrow p$ that then further determines a new update p' to be composed with p.

▶ **Example 1.** It turns out that reader and writer monads are special cases of update monads.

We get the reader monad for a given set S when we take $(P, \mathsf{o}, \oplus)$ and $\downarrow$ trivial, i.e., $P = 1$. We then have $T\,X = S \to 1 \times X \cong S \to X$.

The writer monad for a given monoid $(P, \mathsf{o}, \oplus)$ is obtained by taking S and $\downarrow$ trivial, i.e., $S = 1$, so that $T\,X = 1 \to P \times X \cong P \times X$.

▶ **Example 2.** State monads fail to be a special case of update monads, but they are very close in an important way.

Recall that the free monoid on any semigroup[3] $(S, \bullet)$ is $(P, \mathsf{o}, \oplus)$ where

$$P = 1 + S$$

$$\mathsf{o} = \mathsf{inl}\ *$$

$$\mathsf{inl}\ * \oplus p = p$$
$$\mathsf{inr}\ s \oplus \mathsf{inl}\ * = \mathsf{inr}\ s$$
$$\mathsf{inr}\ s \oplus \mathsf{inr}\ s' = \mathsf{inr}\ (s \bullet s')$$

The actions of this monoid on any set S are determined by the actions of the inducing semigroup $(S, \bullet)$ on S. Recall that, in particular, $\bullet$ is an action of $(S, \bullet)$ on S, so it also induces an action of $(P, \mathsf{o}, \oplus)$ on S.

Notice also that, for any set S, the "overwrite" operation $\bullet$ defined by $s \bullet s' = s'$ gives a semigroup structure on S.[4]

Now let us fix some set S and let (T, η, μ) be the state monad for S. Let $(P, \mathsf{o}, \oplus)$ be the free monoid on the overwrite semigroup $(S, \bullet)$ and let $\downarrow$ be the action of $(P, \mathsf{o}, \oplus)$ on S induced by $\bullet$. Let $(T^\circ, \eta^\circ, \mu^\circ)$ be the update monad for S, $(P, \mathsf{o}, \oplus)$ and $\downarrow$.

It turns out that the state monad (T, η, μ) is characterized as the splitting of the following monad idempotent *idem* on $(T^\circ, \eta^\circ, \mu^\circ)$ that replaces the nil update with overwriting the

[3] Notice that we are talking about the free monoid on a semigroup here, not the free monoid on a set! This is about adjoining a unit element to the semigroup.

[4] In semigroup/monoid literature [10], this is called the right zero semigroup structure.

given state by itself:

$$\begin{aligned}&\mathit{idem} : \forall\{X\}.\, (S \to (1 + S) \times X) \to S \to (1 + S) \times X\\ &\mathit{idem}\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (\mathsf{inr}\ (\mathsf{case}\ p\ \mathsf{of}\ (\mathsf{inl} * \mapsto s; \mathsf{inr}\, s' \mapsto s')), x)\end{aligned}$$

Indeed the monad (T, η, μ) embeds into $(T^\circ, \eta^\circ, \mu^\circ)$ via a monad map sec:

$$\begin{aligned}&\mathit{sec} : \forall\{X\}.\, (S \to S \times X) \to S \to (1 + S) \times X\\ &\mathit{sec}\, f = \lambda s.\, \mathsf{let}\ (s', x) = f\, s\ \mathsf{in}\ (\mathsf{inr}\, s', x)\end{aligned}$$

The monad $(T^\circ, \eta^\circ, \mu^\circ)$ also projects onto the state monad (T, η, μ) via a monad map retr:

$$\begin{aligned}&\mathit{retr} : \forall\{X\}.\, (S \to (1 + S) \times X) \to S \to S \times X\\ &\mathit{retr}\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (\mathsf{case}\ p\ \mathsf{of}\ (\mathsf{inl} * \mapsto s; \mathsf{inr}\, s' \mapsto s'), x)\end{aligned}$$

And it is easy to check that $\mathit{sec} \circ \mathit{retr} = \mathit{idem}$ and $\mathit{retr} \circ \mathit{sec} = \mathsf{id}$.

▶ **Example 3.** We mentioned earlier that the functor $T_\circ$ given by $T_\circ\, X = P_\circ \times X$ is not a monad, if $P_\circ$ is just a set. We need the unit and multiplication of a monoid structure on $P_\circ$ for $T_\circ$ to carry the unit and multiplication of a monad.

But in fact on any set $P_\circ$ we have the the overwrite semigroup structure. Hence $T_\circ$ is at least a "monad without a unit" with multiplication $\mu_\circ : P_\circ \times (P_\circ \times X) \to P_\circ \times X$ given by $\mu_\circ\, (p, (p', x)) = (p', x)$.

We get a monad from $(T_\circ, \mu_\circ)$ by freely adjoining a unit. Concretely, we get a monad (T, η, μ) by defining

$$T\, X = X + P_\circ \times X \quad (\cong (1 + P_\circ) \times X)$$

$$\begin{aligned}&\eta : \forall\{X\}.\, X \to X + P_\circ \times X\\ &\eta\, x = \mathsf{inl}\, x\end{aligned}$$

$$\begin{aligned}&\mu : \forall\{X\}.\, (X + P_\circ \times X) + P_\circ \times (X + P_\circ \times X) \to X + P_\circ \times X\\ &\mu\, (\mathsf{inl}\, c) = c\\ &\mu\, (\mathsf{inr}\, (p, (\mathsf{inl}\, x))) = \mathsf{inr}\, (p, x)\\ &\mu\, (\mathsf{inr}\, (p, (\mathsf{inr}\, (p', x)))) = \mathsf{inr}\, (p', x)\end{aligned}$$

This is (up to isomorphism) the update monad for the set 1, the free monoid on the overwrite semigroup on $P_\circ$ and the trivial action. We call it the *overwrite monad*.

▶ **Example 4.** For any set S and monoid $(P, \mathsf{o}, \oplus)$, we have an action that does nothing: $s \downarrow p = s$. The multiplication operation of the corresponding update monad uses the same state twice:

$$T\, X = S \to P \times X$$

$$\begin{aligned}&\eta : \forall\{X\}.\, X \to S \to P \times X\\ &\eta\, x = \lambda s.\, (\mathsf{o}, x)\end{aligned}$$

$$\begin{aligned}&\mu : \forall\{X\}.\, (S \to P \times (S \to P \times X)) \to S \to P \times X\\ &\mu\, f = \lambda s.\, \mathsf{let}\ (p, g) = f\, s;\\ &\qquad\qquad (p', x) = g\, s\\ &\qquad\ \mathsf{in}\ (p \oplus p', x)\end{aligned}$$

▶ **Example 5.** Here is a cute example of update monads of with a clear programming meaning.

Let $(P, \mathsf{o}, \oplus)$ be the free monoid on a given set S explicitly defined by

$$P = S^*$$

$$\begin{aligned}&\mathsf{o} = []\\ &ss \oplus ss' = ss +\!\!+ ss'\end{aligned}$$

i.e., the set of lists over S, the empty list and concatenation. As the action $\downarrow : S \to S^* \to S$ we want to use

$$s \downarrow ss = \mathsf{last}\,(s :: ss)$$

(Note that :: can be given the type $S \times S^* \to S^+$ and last is a total function $S^+ \to S$.)

We get the following *state-logging* monad (similar to the one considered by Piróg and Gibbons [13], except that it only allows finite traces):

$$T\,X = S \to S^* \times X$$

$$\begin{aligned}&\eta : \forall\{X\}.\, X \to S \to S^* \times X\\ &\eta\, x = \lambda s.\, ([], x)\end{aligned}$$

$$\begin{aligned}&\mu : \forall\{X\}.\, (S \to S^* \times (S \to S^* \times X)) \to S \to S^* \times X\\ &\mu\, f = \lambda s.\, \mathsf{let}\ (ss, g) = f\, s;\\ &\qquad\qquad\ (ss', x) = g\,(\mathsf{last}\,(s :: ss))\\ &\qquad\ \ \mathsf{in}\ (ss +\!\!+ ss', x)\end{aligned}$$

A computation takes an initial state to the list of all intermediate states (excluding the initial state!) and the value returned.

Here and in the following we use Haskell notation for lists, with [] for nil, :: for cons and $+\!\!+$ for append. In addition we will write $es\rfloor n$ for taking n first elements of a list es, $n\lfloor es$ for taking n last elements, and es/n for removing n last elements of es (if $\mathsf{len}\, es \leq n$, then all elements are taken resp. removed).

▶ **Example 6.** Here is a minimally more involved concrete example of an update monad—for no-removal *buffers* of a fixed size N. This is the definition:

$$T\,X = E^{\leq N} \to E^* \times X$$

$$\begin{aligned}&\eta : \forall\{X\}.\, X \to E^{\leq N} \to E^* \times X\\ &\eta\, x = \lambda es.\, ([], x)\end{aligned}$$

$$\begin{aligned}&\mu : \forall\{X\}.\, (E^{\leq N} \to E^* \times (E^{\leq N} \to E^* \times X)) \to E^{\leq N} \to E^* \times X\\ &\mu\, f = \lambda es.\, \mathsf{let}\ (es', g) = f\, es;\\ &\qquad\qquad\ (es'', x) = g\,(es +\!\!+ (es'\rfloor(N - \mathsf{len}\, es)))\\ &\qquad\ \ \mathsf{in}\ (es' +\!\!+ es'', x)\end{aligned}$$

The buffer is used to store values drawn from some given set E and has size N. Therefore, we take as the states $S = E^{\leq N}$ lists of values of length at most N (for values stored in the buffer). The updates $P = E^*$ are simply lists of values to write into the buffer, with the nil update and composition of two updates given by $\mathsf{o} = []$, $es \oplus es' = es +\!\!+ es'$. The action, defined by $es \downarrow es' = es +\!\!+ (es'\rfloor(N - \mathsf{len}\, es))$, updates the buffer with

additional values, as long as there is free space. The updates that do not fully fit into the buffer are performed partially, so some suffix of the list of values to write may be dropped "silently". (An alternative buffer might prefer new values to old, which corresponds to choosing $es \downarrow es' = N\lfloor(es +\!\!+ es')$.)

▶ **Example 7.** To implement an unbounded stack, we can choose the set of states to be $S = E^*$ (values stored in the stack) and as the set of updates use $P = (1 + E)^*$, $\mathsf{o} = []$, $ps \oplus ps' = ps +\!\!+ ps'$ (sequences of pop and push instructions). The intended action $\downarrow$ is then

$$
\begin{aligned}
&es \downarrow [] = es \\
&es \downarrow (\mathsf{inl} * :: ps) = es/1 \downarrow ps \\
&es \downarrow (\mathsf{inr}\, e :: ps) = (es +\!\!+ [e]) \downarrow ps
\end{aligned}
$$

(notice that popping from the empty stack removes no element).

Alternatively, we can be more abstract in regards to updates and identify all those sequences of pop and push instructions that have the same net effect. An update is then a number of elements to remove from the stack and a list of new elements to add. We define

$$
\begin{aligned}
&P' = \mathsf{Nat} \times E^* \\
&\mathsf{o}' = (0, []) \\
&(n, es) \oplus' (n', es') = (n + (n' \dot{-} \mathsf{len}\, es), es/n' +\!\!+ es')
\end{aligned}
$$

$$
es \downarrow' (n', es') = es/n' +\!\!+ es'
$$

The monoid here is a Zappa-Szép product [5] of the monoids $(\mathsf{Nat}, 0, +)$ and $(E^*, [], +\!\!+)$. It arises from two matching actions of the two monoids on each other.

In Section 3, we will see that a dependently typed version of update monads can disallow over- and underflowing updates.

2.3 Algebras of update monads

By the definition of an algebra of a monad (see Appendix A.2), an algebra for the update monad for an act $(S, (P, \mathsf{o}, \oplus), \downarrow)$ is a set X with an operation

$$
act : (S \to P \times X) \to X
$$

satisfying the equations

$$
\begin{aligned}
&x = act\, (\lambda s.\, (\mathsf{o}, x)) \\
&act\, (\lambda s.\, (p\, s, act\, (\lambda s'.\, (p'\, s\, s', x\, s\, s')))) = act\, (\lambda s.\, (p\, s \oplus p'\, s\, (s \downarrow p\, s), x\, s\, (s \downarrow p\, s)))
\end{aligned}
$$

However it is quite easy to see that the same thing can also be described as a set X with two operations (see the interdefinability below)

$$
\begin{aligned}
&lkp : (S \to X) \to X \\
&upd : P \times X \to X
\end{aligned}
$$

satisfying the equations

$$
\begin{aligned}
&x = lkp\, (\lambda s.\, upd\, (\mathsf{o}, x)) \\
&upd\, (p, upd\, (p', x)) = upd\, (p \oplus p', x) \\
&lkp\, (\lambda s.\, upd\, (p\, s, lkp\, (\lambda s'.\, x\, s\, s'))) = lkp\, (\lambda s.\, upd\, (p\, s, x\, s\, (s \downarrow p\, s)))
\end{aligned}
$$

The intuition behind the design of the equation system is that every algebra expression should be rewritable into the form $\mathit{lkp}\,(\lambda s.\ \mathit{upd}\,(p\,s, x\,s))$. Seen as rewrite rules, the 1st equation enables one to prefix a given algebra expression with a pair of occurrences of *lkp* and *upd* whereas the 2nd and 3rd equations allow removal of all subsequent occurrences of *lkp* and *upd*.

The operations

$$\mathit{act} : (S \to P \times X) \to X$$

and

$$\begin{aligned}&\mathit{lkp} : (S \to X) \to X\\ &\mathit{upd} : P \times X \to X\end{aligned}$$

satisfying their respective axioms are interdefinable via

$$\begin{aligned}&\mathit{lkp}\,(\lambda s.\ x\,s) = \mathit{act}\,(\lambda s.\ (\mathsf{o}, x\,s))\\ &\mathit{upd}\,(p, x) = \mathit{act}\,(\lambda s.\ (p, x))\end{aligned}$$

and

$$\mathit{act}\,(\lambda s.\ (p\,s, x\,s)) = \mathit{lkp}\,(\lambda s.\ \mathit{upd}\,(p\,s, x\,s))$$

2.4 Update monads as a compatible composition of reader and writer monads

While state monads cannot be described as compositions of reader and writer monads, update monads are exactly that!

The update monad (T, η, μ) for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ is a compatible composition (in the sense of the definition given in Section A.3) of the reader monad (T_0, η_0, μ_0) for S and the writer monad (T_1, η_1, μ_1) for $(P, \mathsf{o}, \oplus)$: the underlying functor T is the functor composition $T_0 \cdot T_1$ and the unit η and multiplication μ relate to those of the reader and writer monad in a certain way, which implies, in particular, that $T_0 \cdot \eta_1$ is a monad map from (T_0, η_0, μ_0) to (T, η, μ) and $\eta_0 \cdot T_1$ is one from (T_1, η_1, μ_1) to (T, η, μ).

The corresponding distributive law θ of (T_1, η_1, μ_1) over (T_0, η_0, μ_0) is determined by the action $\downarrow$:

$$\begin{aligned}&\theta : \forall\{X\}.\ P \times (S \to X) \to S \to P \times X\\ &\theta\,(p, f) = \lambda s.\ (p, f\,(s \downarrow p))\end{aligned} \tag{1}$$

Moreover, every compatible composition of these two monads is an update monad, since every distributive law θ of (T_1, η_1, μ_1) over (T_0, η_0, μ_0) defines an action $\downarrow$ satisfying (1) via

$$\begin{aligned}&\downarrow : S \times P \to S\\ &s \downarrow p = \mathsf{snd}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s)\end{aligned} \tag{2}$$

while it follows directly from the definition of θ that

$$p = \mathsf{fst}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s) \tag{3}$$

Substituting the two definitions into each other the other way around yields identity too, so (1) and (2) give a bijective correspondence between the actions and the distributive laws.

▶ **Lemma 8.** *For any distributive law θ of the writer monad for $(P, \mathsf{o}, \oplus)$ over the reader monad for S, equation (3) holds.*

▶ **Theorem 9.** *Equations (1), (2) establish a bijective correspondence between the actions of* $(P, \mathsf{o}, \oplus)$ *on* S *and the distributive laws of the writer monad for* $(P, \mathsf{o}, \oplus)$ *over the reader monad for* S*.*

For proofs, see Appendix B.

The trivial action $s \downarrow p = s$ corresponds to the distributive law $\theta\,(p, f) = \lambda s.\,(p, f\,s)$, which is the strength of T_0.

As an instance of the general characterization of algebras of a compatible composition of two monads in terms of their algebras, we learn that an algebra of the update monad for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ can be specified as a set X carrying algebras of both the reader and writer monad, i.e., operations

$$\mathit{lkp} : (S \to X) \to X \qquad\qquad \mathit{upd} : P \times X \to X$$

satisfying the conditions

$$x = \mathit{lkp}\,(\lambda s.\,x) \qquad\qquad x = \mathit{upd}\,(\mathsf{o}, x)$$
$$\mathit{lkp}\,(\lambda s.\,\mathit{lkp}\,(\lambda s'.\,x\,s\,s')) = \mathit{lkp}\,(\lambda s.\,x\,s\,s) \qquad \mathit{upd}\,(p, \mathit{upd}(p', x)) = \mathit{upd}\,(p \oplus p', x)$$

plus an additional compatibility condition

$$\mathit{upd}\,(p, \mathit{lkp}\,(\lambda s'.\,x\,s')) = \mathit{lkp}\,(\lambda s.\,\mathit{upd}\,(p, x\,(s \downarrow p)))$$

This axiomatization of *lkp* and *upd* is quite different from the one we showed above—only one axiom is shared—, but nonetheless equivalent. One could also argue that it is more systematic and symmetric.

For the trivial action $s \downarrow p = s$, the compatibility condition becomes $\mathit{upd}\,(p, \mathit{lkp}\,(\lambda s'.\,x\,s'))$ $= \mathit{lkp}\,(\lambda s.\,\mathit{upd}\,(p, x\,s))$—the condition of models of the tensor of the Lawvere theories for reading and writing [9, Section 5].

It is important to notice that algebras (X, upd) of the reader monad are nothing but sets with a *left* action of $(P, \mathsf{o}, \oplus)$ while $(S, \downarrow)$ is a set with a *right* action of $(P, \mathsf{o}, \oplus)$.

2.5 Maps between update monads

What are maps between update monads like? It turns out that, for a suitable notion of act maps, every map between two given acts in the reverse direction defines a map between the corresponding update monads, but this mapping of act maps to monad maps is generally neither injective nor surjective.

We choose to define a *map* between two acts $(S', (P', \mathsf{o}', \oplus'), \downarrow')$ and $(S, (P, \mathsf{o}, \oplus), \downarrow)$ to be a function $t : S' \to S$ together with a monoid homomorphism $q : (P, \mathsf{o}, \oplus) \to (P', \mathsf{o}', \oplus')$ (notice the direction of q!) such that

$$t\,(s \downarrow' q\,p) = t\,s \downarrow p$$

holds.[5]

These pairs (t, q) are in a bijective correspondence with pairs (τ_0, τ_1) where τ_0 is a map between the reader monads (T_0, η_0, μ_0) and (T_0', η_0', μ_0') for S_0 resp. S_0' and τ_1 is a map

[5] More customarily, a map between these acts would be taken to be a function $t : S' \to S$ together with a monoid homomorphism $q : (P', \mathsf{o}', \oplus') \to (P, \mathsf{o}, \oplus)$ satisfying the condition $t\,(s \downarrow' p) = t\,s \downarrow q\,p$, see, e.g., [10, p. 54].

between the writer monads (T_1, η_1, μ_1) and (T_1', η_1', μ_1') for $(P, \mathsf{o}, \oplus)$ resp. $(P, \mathsf{o}', \oplus')$ satisfying the condition

$$\begin{array}{ccc} T_1 \cdot T_0 & \xrightarrow{\tau_1 \cdot \tau_0} & T_1' \cdot T_0' \\ {\scriptstyle\theta}\downarrow & & \downarrow{\scriptstyle\theta'} \\ T_0 \cdot T_1 & \xrightarrow[\tau_0 \cdot \tau_1]{} & T_0' \cdot T_1' \end{array}$$

where θ and θ' are the distributive laws defined by $\downarrow$ resp. $\downarrow'$.

Acts and act maps form a category.

Every act map (t, q) between $(S', (P', \mathsf{o}', \oplus'), \downarrow')$ and $(S, (P, \mathsf{o}, \oplus), \downarrow)$ determines a monad map τ between the corresponding update monads (T, η, μ) and (T, η', μ') via

$$\begin{array}{l} \tau : \forall \{X\}.\, (S \to P \times X) \to S' \to P' \times X \\ \tau\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, (t\, s)\ \mathsf{in}\ (q\, p, x) \end{array}$$

extending the mapping of acts to monads into a functor from the opposite of the category of acts to the category of monads on **Set**.

This functor is neither faithful nor full. To see the failure of faithfulness, let S be any set, but $S' = 0$, and let $(P, \mathsf{o}, \oplus)$, $(P', \mathsf{o}', \oplus')$ be arbitrary monoids with more than one monoid map between them. Let $\downarrow$ be arbitrary; as $\downarrow' : S' \times P' \to S'$ we can only choose the empty function. Now there is exactly one map $t : S' \to S$, namely the empty function. For any monoid map $q : (P, \mathsf{o}, \oplus) \to (P', \mathsf{o}', \oplus')$, the pair (t, q) is an act map, but the corresponding map τ between the update monads, with type $\tau : \forall \{X\}.\, (S \to P \times X) \to S' \to P' \times X$, sends any given map f to the empty map irrespective of the choice of q.

A simple counterexample to fullness is obtained by considering the reader monad for a given S, the update monad extension of the state monad for S (the update monad of Example 2) and the more interesting one of the two canonical embeddings between them. Concretely, we take S to be an arbitrary non-trivial set (i.e., not 0, not 1) and $S' = S$. We let $(P, \mathsf{o}, \oplus)$ and $\downarrow$ be trivial (i.e., $P = 1$) and we let $(P', \mathsf{o}', \oplus')$ and $\downarrow'$ be the free monoid on the overwrite semigroup on S and the action of P' on S given by overwriting. We define $\tau : \forall \{X\}.\, (S \to 1 \times X) \to S \to (1 + S) \times X$ by $\tau\, f = \lambda s.\, \mathsf{let}\ (*, x) = f\, s\ \mathsf{in}\ (\mathsf{inr}\, s, x)$. Now τ is a monad map, but it is not the image of any act map (t, q).

3 A dependently typed generalization

Recall the fixed-size no-removal buffer and stack of Examples 6 and 7. They can overflow and underflow. This raises a natural question: Is it possible to restrict the updates so that this is guaranteed to not happen?

This cannot be done with the definition given in Section 2.2. The reason is the non-dependence of updates on states. To remedy this, we now define a dependently-typed generalization of update monads. It is related to Abbott, Altenkirch and Ghani's containers [1] (equivalent to (simple, or non-dependent) polynomials [8]).

We recall that a *container* is a set S together with a S-indexed family P. A *map* between two containers (S, P) and (S', P') is given by functions $t : S \to S'$ and $q : \Pi\, \{s : S\}.\, P'\, (t\, s) \to P\ s$. Containers and container morphisms form a monoidal category **Cont**.

Any container determines a set functor $[\![S, P]\!]^{\mathrm{c}}$ (its *interpretation*) by

$$\begin{array}{l} [\![S, P]\!]^{\mathrm{c}}\, X = \Sigma s : S.\, P\, s \to X \\ [\![S, P]\!]^{\mathrm{c}}\, h\, (s, v) = (s, h \circ v) \end{array}$$

By associating a map (t, q) between containers (S, P) and (S', P') with a natural transformation $[\![t, q]\!]^{\mathrm{c}}$ between the functors $[\![S, P]\!]^{\mathrm{c}}$ and $[\![S', P']\!]^{\mathrm{c}}$ by

$$[\![t, q]\!]^{\mathrm{c}}\,(s, v) = (t\,s, v \circ q\,\{s\})$$

the mapping $[\![-]\!]^{\mathrm{c}}$ is extended into a fully faithful monoidal functor from **Cont** to [**Set**, **Set**].

In our previous work with Chapman [2], we introduced directed containers as characterization of containers that are comonads. A directed container is similar to an act, except that the monoid carrier and operations depend on elements of the set.

A *directed container* is a set S together with a S-indexed family P and operations

$$\downarrow : \Pi s : S.\, P\,s \to S$$
$$\mathsf{o} : \Pi\{s : S\}.\, P\,s$$
$$\oplus : \Pi\{s : S\}.\, \Pi p : P\,s.\, P\,(s \downarrow p) \to P\,s$$

satisfying the equations

$$s \downarrow \mathsf{o} = s$$
$$s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$$

$$p \oplus \mathsf{o} = p$$
$$\mathsf{o} \oplus p = p$$
$$(p \oplus p') \oplus p'' = p \oplus (p' \oplus p'')$$

Observe that, on the level of terms (with implicit arguments suppressed) these five equations are exactly those of a monoid and an action. But the typing is different. In fact, the 4th equation is only well-typed on the assumption of the 1st equation and similarly the well-typedness of the 5th equation depends on a proof of the 2nd equation. (In the renderings above, this is invisible, as we have also suppressed type-index conversions.) If none of $P\,s$, $\mathsf{o}\,\{s\}$ and $p \oplus \{s\}\,p'$ actually depends on s, the directed container is an act. If $s \downarrow p = s$, then it is a set together with a family of monoids.

A *map* between directed containers $(S, P, \downarrow, \mathsf{o}, \oplus)$ and $(S', P', \downarrow', \mathsf{o}', \oplus')$ is a map (t, q) between the underlying containers (S, P) and (S', P') such that

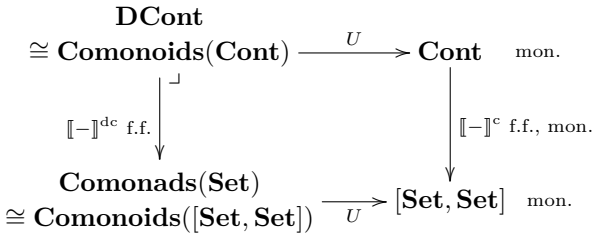
$$t\,(s \downarrow q\,p) = t\,s \downarrow'\, p$$
$$\mathsf{o} = q\,\mathsf{o}'$$
$$q\,p \oplus q\,p' = q\,(p \oplus'\, p')$$

These equation look like those for a monoid map and an action map, but are typed finer. In particular, the 3rd equation is well-typed on the assumption of the 1st equation. If the two directed containers are in fact acts and $q\,\{s\}\,p$ does not actually depend on s, then q is an act map.

Directed containers and directed container maps form a category **DCont** that turns out to be isomorphic to the category **Comonoids**(**Cont**) of comonoid objects in the monoidal category **Cont**.

This isomorphism together with monoidality of the functor $[\![-]\!]^{\mathrm{c}}$ implies that the functor $[\![-]\!]^{\mathrm{c}} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$ lifts to a functor $[\![-]\!]^{\mathrm{dc}}$ from **DCont** $\cong$ **Comonoids**(**Cont**) to **Comonads**(**Set**) $\cong$ **Comonoids**([**Set**, **Set**]) interpreting directed containers into comonads. From fully faithfulness of $[\![-]\!]^{\mathrm{c}}$ it follows that $[\![-]\!]^{\mathrm{dc}}$ is fully faithful too, i.e., the maps between the interpretations of two directed containers are in a bijection between the maps between these directed containers. More, $[\![-]\!]^{\mathrm{dc}}$ is the pullback of $[\![-]\!]^{\mathrm{c}}$ along the forgetful functor $U : \mathbf{Comonads}(\mathbf{Set}) \to [\mathbf{Set}, \mathbf{Set}]$, meaning that directed containers are in fact exactly the

containers whose interpretation carries a comonad structure. This is summarized in the following diagram.



For this paper, we have a reason to refocus from interpretation to "cointerpretation". Let us assign to every container (S, P) a set functor $\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}$ (its *cointerpretation*) by setting

$$\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}\, X = \Pi s : S.\, P\, s \times X$$
$$\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}\, h\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (p, h\, x)$$

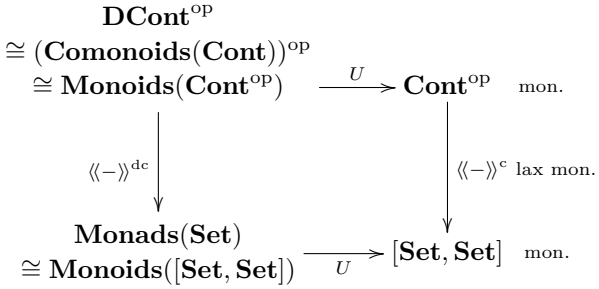
By associating with any container map (t, q) between (S', P') and (S, P) a natural transformation $\langle\!\langle t, q \rangle\!\rangle^{\mathrm{c}}$ between $\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}$ and $\langle\!\langle S', P' \rangle\!\rangle^{\mathrm{c}}$, which is easily done by taking

$$\langle\!\langle t, q \rangle\!\rangle^{\mathrm{c}}\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, (t\, s)\ \mathsf{in}\ (q\, \{s\}\, p, x)$$

we extend the mapping $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ to a functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ between $\mathbf{Cont}^{\mathrm{op}}$ and $[\mathbf{Set}, \mathbf{Set}]$.

The functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}} : \mathbf{Cont}^{\mathrm{op}} \to [\mathbf{Set}, \mathbf{Set}]$ is not as well-behaved as $[\![-]\!]^{\mathrm{c}} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$. First of all, $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ fails to be monoidal, it is only lax monoidal. Second, it is neither faithful nor full.

Nonetheless, the mere lax monoidality of $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ is enough to obtain a canonical cointerpretation mapping of directed containers into monads. It suffices to note that $\mathbf{DCont}^{\mathrm{op}} \cong (\mathbf{Comonoids}(\mathbf{Cont}))^{\mathrm{op}} \cong \mathbf{Monoids}(\mathbf{Cont}^{\mathrm{op}})$ and $\mathbf{Monads}(\mathbf{Set}) \cong \mathbf{Monoids}([\mathbf{Set}, \mathbf{Set}])$. Lax monoidality of $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}} : \mathbf{Cont}^{\mathrm{op}} \to [\mathbf{Set}, \mathbf{Set}]$ implies that $\langle\!\langle - \rangle\!\rangle^{\mathrm{c}}$ sends monoids to monoids and lifts to a functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}} : \mathbf{DCont}^{\mathrm{op}} \to \mathbf{Monads}(\mathbf{Set})$. This is summarized in the following diagram where the square commutes, but is not a pullback.



Explicitly, the functor $\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}}$ sends a directed container $(S, P, \downarrow, \mathsf{o}, \oplus)$ to the monad (T, η, μ) (the corresponding *(dependently typed) update monad*) given by

$$T\, X = \langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}\, X = \Pi s : S.\, P\, s \times X$$

$$\eta : \forall \{X\}.\, X \to \Pi s : S.\, P\, s \times X$$
$$\eta\, x = \lambda s.\, (\mathsf{o}, x)$$

$$\mu : \forall \{X\}.\, (\Pi s : S.\, P\, s \times (\Pi s' : S.P\, s' \times X)) \to \Pi s : S.\, P\, s \times X$$
$$\begin{aligned} \mu\, f = \lambda s.\, &\mathsf{let}\ (p, g) = f\, s; \\ &\quad (p', x) = g\, (s \downarrow p) \\ &\mathsf{in}\ (p \oplus p', x) \end{aligned}$$

On the level of terms, the definitions of the unit and multiplication look exactly as those we gave in Section 2.2 for the update monad for an act, but their types are finer.

A map (t, q) between directed containers $(S', P', \downarrow', \mathsf{o}', \oplus')$ and $(S, P, \downarrow, \mathsf{o}, \oplus)$ is sent by $\langle\!\langle - \rangle\!\rangle^{\mathrm{dc}}$ to the natural transformation $\langle\!\langle t, q \rangle\!\rangle^{\mathrm{dc}} = \langle\!\langle t, q \rangle\!\rangle^{\mathrm{c}}$ between the functors $\langle\!\langle S, P \rangle\!\rangle^{\mathrm{c}}$ and $\langle\!\langle S', P' \rangle\!\rangle^{\mathrm{c}}$, which is also a monad map between $\langle\!\langle S, P, \downarrow, \mathsf{o}, \oplus \rangle\!\rangle^{\mathrm{dc}}$ and $\langle\!\langle S', P', \downarrow', \mathsf{o}', \oplus' \rangle\!\rangle^{\mathrm{dc}}$. This way of specifying maps between dependently typed update monads generalizes the one we described in Section 2.5 for maps between simply typed update monads.

The intuitive advantage of dependently typed update monads over simply typed update monads lies in the idea of updates *enabled* (or safe) in a state. In the simply typed case, any update has to apply to any state.

In the dependently typed setting, any initial state $s : S$ determines its own set of updates $P\,s$ enabled in it. And, according to the type of $\downarrow$, only those updates are applicable to s. This means that we are not forced to invent outcomes for updates that should morally only be allowed in some states.

▶ **Example 10.** In the example of the buffer (Example 6), we chose to write only a prefix of a given list into the buffer, if it had no space left for the full list. This is clearly a dangerous design, as values get discarded silently. Another option would have been to introduce a special error state. But with a dependently typed update monad, we can do much better.

The non-overflowing fixed-size no-removal buffer monad is given by the following data:

$$T\,X = \Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times X$$

$$\eta : \forall \{X\}.\, X \to \Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times X$$
$$\eta\, x = \lambda es.\, []$$

$$\mu : \forall \{X\}.\, (\Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times (\Pi es' : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es'} \times X)) \to \Pi es : E^{\leq N}.\, E^{\leq N - \mathsf{len}\, es} \times X$$
$$\begin{aligned}\mu\, f = \lambda es.\, \mathsf{let}\ & (es', g) = f\, es; \\ & (es'', x) = g\, (es +\!\!+ es') \\ \mathsf{in}\ & (es' +\!\!+ es'', x)\end{aligned}$$

The states are lists of length at most n as before: $S = E^{\leq N}$. But the updates, acceptable lists of values to write, now depend on these states in a natural way. Namely, for a state $es : E^{\leq N}$ of the buffer, the enabled updates are $P\, es = E^{\leq N - \mathsf{len}\, es}$, i.e., lists that can be appended to es without exceeding the length limit N. The means that the action does not have to truncate, it is just concatenation: $es \downarrow es' = es +\!\!+ es'$.

▶ **Example 11.** Similarly, we can amend our two stack monads from Example 7 to be non-underflowing.

As before, the set of states $S = E^*$ is given by lists of elements of E. Regarding the monoid of updates, we can define $P\, es = \{ps : (1 + E)^* \mid \mathsf{removes}\, ps \leq \mathsf{len}\, es\}$ where

$$\begin{aligned}&\mathsf{removes}\, [] = 0 \\ &\mathsf{removes}\, (\mathsf{inl}\, * :: ps) = \mathsf{removes}\, ps + 1 \\ &\mathsf{removes}\, (\mathsf{inr}\, e :: ps) = \mathsf{removes}\, ps \dot{-} 1\end{aligned}$$

Alternatively, we can define $P'\, es = [0..\mathsf{len}\, es] \times E^*$.

Differently from simply typed update monads, dependently typed update monads subsume state monads.

▶ **Example 12.** Given a set S, define $P\,s = S$, $s \downarrow s' = s'$, $\mathsf{o}\,\{s\} = s$, $s' \oplus \{s\}\,s'' = s''$.

The update monad for this directed container is the state monad for S.

In Example 2, we noted that the state monad for S fails to be a simply typed update monad, because $P = S$ is just a semigroup, not a monoid. With the dependently typed notion, we can afford a different unit element $\mathsf{o}\,\{s\} = s : P\,s = S$ for each $s : S$, overcoming this obstacle.

The monad morphisms *idem*, *sec* and *retr* are morphisms of dependently typed update monads.

An (EM-)algebra for the dependently typed update monad for the directed container $(S, P, \downarrow, \mathsf{o}, \oplus)$ is a set X with an operation

$$act : (\Pi s : S.\, P\,s \times X) \to X$$

satisfying the equations

$$x = act\,(\lambda s.\,(\mathsf{o}\,\{s\}, x))$$
$$act\,(\lambda s.\,(p\,s, act\,(\lambda s'.\,(p'\,s\,s', x\,s\,s')))) = act\,(\lambda s.\,(p\,s \oplus \{s\}\,p'\,s\,(s \downarrow p\,s), x\,s\,(s \downarrow p\,s)))$$

Again the equations look exactly the same as in the simply typed case, but the types are different.

It is not clear to us whether dependently typed update monads admit a useful decomposition similar to the decomposition of simple update monads into reader and writer monads. One possibility is to resort to relative monads of Altenkirch et al. [4]: dependently typed update monads can be described as compatible compositions of certain relative monads.

Given a directed container $(S, P, \downarrow, \mathsf{o}, \oplus)$. Define $J_0 : [S, \mathbf{Set}] \to \mathbf{Set}$ by $J_0\,X = \Pi s : S.\, X\,s$ and $J_1 : \mathbf{Set} \to [S, \mathbf{Set}]$ by $J_1\,X\,s = X$ (notice that J_0 is right adjoint to J_1). Now on J_0 we have a rather trivial but nonetheless reader-like relative monad $(T_0, \eta_0, (-)_0^*)$ given by $T_0\,X = \Pi s : S.\, X\,s = J_0\,X$, $\eta_0\,\{X\} = \mathsf{id}\,\{J_0\,X\}$, $k_0^* = k$. On J_1 at the same time we can define a writer-like relative monad $(T_1, \eta_1, (-)_1^*)$ by $T_1\,X\,s = P\,s \times X$, $\eta_1\,\{X\}\,\{s\}\,x = (\mathsf{o}\,\{s\}, x)$, $k_1^*\,\{s\}\,(p, x) = \mathsf{let}\ (p', y) = k\,\{s \downarrow p\}\,x\ \mathsf{in}\ (p \oplus \{s\}\,p', y)$. The dependently typed update monad is a compatible composition of the two relative monads.

4 Kammar and Plotkin's generalization of state monads

Kammar and Plotkin[6] have proposed a generalization of state monads that is related to ours. Similarly to us, they employ monoids and monoid actions. Kammar and Plotkin's monad for an act $(S, (P, \mathsf{o}, \oplus), \downarrow)$ is defined by

$$T\,X = \Pi s : S.\,(s \downarrow P) \times X$$

$$\eta : \forall\{X\}.\, X \to \Pi s : S.\,(s \downarrow P) \times X$$
$$\eta\,x = \lambda s.\,(s, x)$$

$$\mu : \forall\{X\}.\,(\Pi s : S.\,(s \downarrow P) \times (\Pi s' : S.\,(s' \downarrow P) \times X)) \to \Pi s : S.\,(s \downarrow P) \times X$$
$$\mu\,f = \lambda s.\,\mathsf{let}\ (s', g) = f\,s;$$
$$\qquad\qquad (s'', x) = g\,s'$$
$$\qquad \mathsf{in}\ (s'', x)$$

[6] O. Kammar and G. Plotkin. Take action for your state: effective conservative restrictions. Slides from Scottish Programming Language Seminar, Strathclyde, Nov. 2010.

Here $s \downarrow P = \{s \downarrow p \mid p \in P\} \subseteq S$ is the orbit of s along the action $\downarrow$ of the monoid $(P, \mathsf{o}, \oplus)$ on the set S. Notice that η and μ are defined just as for the state monad for S, only the typing is finer. In particular, the monoid structure and the action only appear in the types.

Reader and state monads are special cases of this unification, while writer monads are not. (Remember that, in contrast, simply typed update monads cover reader and writer monads, but not state monads.)

Kammar and Plotkin's monad for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ turns out to be the middle monad in the epi-mono factorization of the obvious monad map τ between the simply typed update monad for $(S, (P, \mathsf{o}, \oplus), \downarrow)$ and the state monad for S. For a given state, τ just applies to a given initial state the update that it produces.

$$\tau : \forall\{X\}.\, (S \to P \times X) \to S \to S \times X$$
$$\tau\, f = \lambda s.\, \mathsf{let}\ (p, x) = f\, s\ \mathsf{in}\ (s \downarrow p, x)$$

Just as the state monad, Kammar and Plotkin's monad is an instance of a dependently typed update monad. The appropriate directed container is $(S, P', \downarrow', \mathsf{o}', \oplus')$ where

$$P'\, s = s \downarrow P$$

$$\downarrow' : \Pi s : S.\, \Pi s' : s \downarrow P.\, S$$
$$s \downarrow'\, s' = s'$$

$$\mathsf{o}' : \Pi\{s : S\}.\, s \downarrow P$$
$$\mathsf{o}'\, \{s\} = s$$

$$\oplus' \colon \Pi s : S.\, \Pi s' : s \downarrow P.\, s' \downarrow P \to s \downarrow P$$
$$s' \oplus'\, \{s\}\, s'' = s''$$

5 Conclusion and future work

We have presented some facts about a class of monads that we call update monads. We hope that those convince the reader that the concept is meaningful and elegant. Although we arrived at update monads thinking about cointerpretation of directed containers, in hindsight we think that they are above all a simple, but instructive unification of the reader, writer and state monads. This unification helps explain how they and some further special monads interrelate and why.

As future work, we wish to generalize this work to monoidal closed categories (replacing unique comonoids with arbitrary comonoids) and to presheaf categories (replacing directed containers with directed indexed containers).

Acknowledgements. Danel Ahman thanks Ohad Kammar for discussions. Tarmo Uustalu acknowledges the feedback from Ichiro Hasuo, Zhenjiang Hu and their colleagues at the University of Tokyo and National Institute of Informatics.

References

1 M. Abbott, T. Altenkirch, N. Ghani. Containers: constructing strictly positive types. *Theor. Comput. Sci.*, v. 342, n. 1, pp. 3–27, 2005.

2 D. Ahman, J. Chapman, T. Uustalu. When is a container a comonad? In L. Birkedal, ed., *Proc. of 15th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2012 (Tallinn, March 2012)*, v. 7213 of *Lect. Notes in Comput. Sci.*, pp. 74–88. Springer, 2012. Journal version to appear in *Log. Methods in Comput. Sci.*

3 D. Ahman, T. Uustalu. Distributive laws of directed containers. *Progress in Informatics*, v. 10, pp. 3–18, 2013.

4 T. Altenkirch, J. Chapman, T. Uustalu. Monads need not be endofunctors. In L. Ong, ed., *Proc. of 13th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2010 (Paphos, March 2010)*, v. 6014 of *Lect. Notes in Comput. Sci.*, pp. 297-311. Springer, 2010. Journal version to appear in *Log. Methods in Comput. Sci.*

5 M. G. Brin. On the Zappa-Szép product. *Commun. in Algebra*, v. 33, n. 2, pp. 393–424, 2005.

6 M. Barr and C. Wells. *Toposes, Triples and Theories*, v. 278 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1984.

7 J. Beck. Distributive laws. In B. Eckmann, ed., *Seminar on Triples and Categorical Homology, ETH 1966/67*, v. 80 of *Lect. Notes in Math.*, pp. 119–140. Springer, 1969.

8 N. Gambino, M. Hyland. Wellfounded trees and dependent polynomial functors. In S. Berardi, M. Coppo, F. Damiani, eds., *Revised Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES 2003 (Torino, Apr./May 2003)*, v. 2085 of *Lect. Notes in Comput. Sci.*, pp. 210–225. Springer, 2004.

9 M. Hyland, G. Plotkin, J. Power. Combining effects: sum and tensor. *Theor. Comput. Sci.*, v. 357, no. 1, pp. 70–99, 2006.

10 M. Kilp, U. Knauer, A. V. Mikhalev. *Monoids, Acts and Categories: With Applications to Wreath Products and Graphs*, v. 29 of *De Gruyter Expositions in Mathematics*. De Gruyter, 2000.

11 S. Mac Lane. *Categories for the Working Mathematician*, v. 5 of *Graduate Texts in Mathematics*. Springer, 1971.

12 U. Norell. Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology, 2007.

13 M. Piróg, J. Gibbons. Monads for behavior. In D. Kozen, M. Mislove, eds., *Proc. of MFPS XXIX (New Orleans, LA, June 2013)*, v. 298 of *Electron. Notes in Theor. Comput. Sci.*, pp. 309–324. Elsevier, 2013.

14 G. D. Plotkin, J. Power. Notions of computation determine monads. In M. Nielsen, U. Engberg, eds., *Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2002 (Grenoble, April 2002)*, v. 2303 of *Lect. Notes in Comput. Sci.*, pp. 342–356. Springer, 2002.

15 P. Wadler. The essence of functional programming. In *Proc. of 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 1992 (Albuquerque, NM, Jan. 1992)*, pp. 1–14. ACM Press, 1992.

A Background

A.1 Monoids, actions

We recall that a *monoid* is a set P together with two operations

$$\mathsf{o} : P$$
$$\oplus : P \times P \to P$$

satisfying

$$p \oplus \mathsf{o} = p$$
$$\mathsf{o} \oplus p = p$$
$$(p \oplus p') \oplus p'' = p \oplus (p' \oplus p'')$$

A *map* between two monoids $(P, \mathsf{o}, \oplus)$ and $(P', \mathsf{o}', \oplus')$ is a map

$$q : P \to P'$$

satisfying

$$q\,\mathsf{o} = \mathsf{o}'$$
$$q\,(p \oplus p') = q\,p \oplus' q\,p'$$

A *right action* of a monoid $(P, \mathsf{o}, \oplus)$ on a set S is an operation

$$\downarrow : S \times P \to S$$

satisfying the conditions

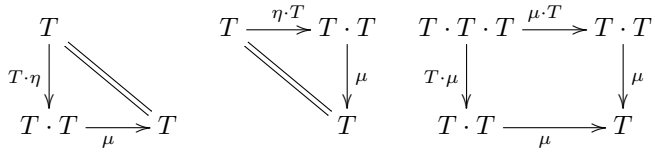
$$s \downarrow \mathsf{o} = s$$
$$s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$$

Similarly, a *left action* of $(P, \mathsf{o}, \oplus)$ on S is an operation $\uparrow : P \times S \to S$ satisfying $\mathsf{o} \uparrow s = s$, $(p \oplus p') \uparrow s = p \uparrow (p' \uparrow s)$.

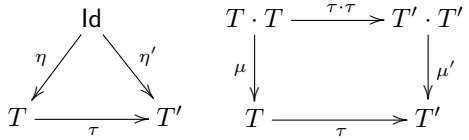
A.2 Monads, monad algebras

We recall the definitions of monads and algebras for monads. For thorough expositions, we refer the reader to the books by Barr and Wells [6, Ch. 3] and Mac Lane [11, § VI].

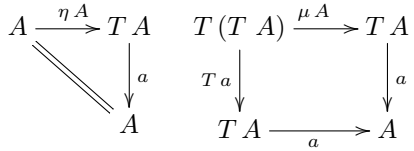
A *monad* on a category $\mathbb{C}$ is given by an endofunctor T on $\mathbb{C}$ and natural transformations $\eta : \mathsf{Id} \to T$ and $\mu : T \cdot T$ satisfying the conditions



A *map* between monads (T, η, μ) and (T', η', μ') on the same category $\mathbb{C}$ is a natural transformation $\tau : T \to T'$ satisfying the conditions



An *(Eilenberg-Moore) algebra* of a monad (T, η, μ) is an object A together with a map $a : T\,A \to A$ satisfying the conditions



For any object A, there is a free algebra of the monad (T, η, μ) on A: the algebra $(T\,A, \mu\,A)$ together with the map $\eta\,A : A \to T\,A$.

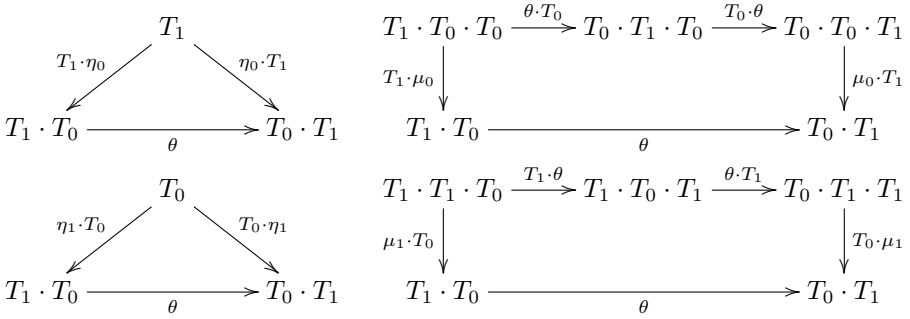
A.3 Distributive laws and compatible compositions of monads

Distributive laws, compatible compositions and liftings are due to Beck [7]. They are also discussed in the book of Barr and Wells [6, Ch. 9].

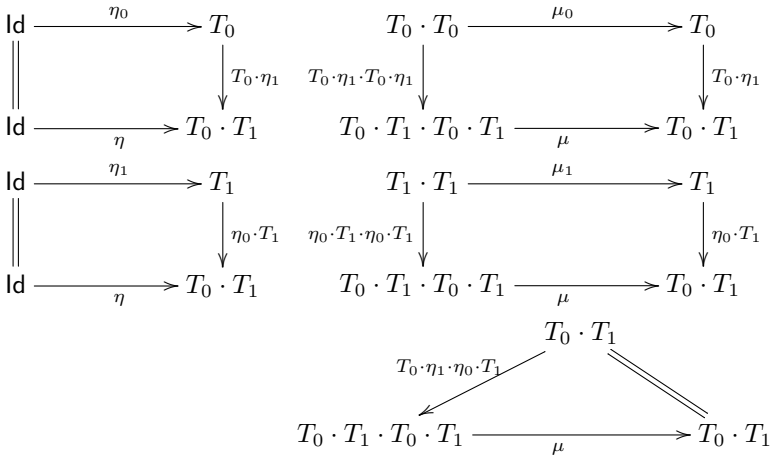
A *distributive law* between two monads (T_0, η_0, μ_0) and (T_1, η_1, μ_1) on the same category $\mathbb{C}$ is a natural transformation

$$\theta : T_1 \cdot T_0 \to T_0 \cdot T_1$$

satisfying the conditions



A *compatible composition* of monads (T_0, η_0, μ_0) and (T_1, η_1, μ_1) on $\mathbb{C}$ is a monad structure (η, μ) on the endofunctor $T_0 \cdot T_1$ satisfying the conditions



Notice that the first two conditions say that $T_0 \cdot \eta_1$ is a morphism between the monads (T_0, η_0, μ_0) and $(T_0 \cdot T_1, \eta, \mu)$ and the next two say that $\eta_0 \cdot T_1$ is a morphism between the monads (T_1, η_1, μ_1) and $(T_0 \cdot T_1, \eta, \mu)$. Notice also that the 1st and 3rd conditions really say the same, namely, that $\eta = \eta_0 \cdot \eta_1$. The most significant condition is the 5th, the so-called middle unital law.

Distributive laws and compatible compositions are in a bijective correspondence. A distributive law θ determines a compatible composition (η, μ) via

$$\eta = \mathsf{Id} \xrightarrow{\eta_0 \cdot \eta_1} T_0 \cdot T_1$$
$$\mu = T_0 \cdot T_1 \cdot T_0 \cdot T_1 \xrightarrow{T_0 \cdot \theta \cdot T_1} T_0 \cdot T_0 \cdot T_1 \cdot T_1 \xrightarrow{\mu_0 \cdot \mu_1} T_0 \cdot T_1$$

Conversely, a compatible composition (η, μ) defines a distributive law θ via

$$\theta = T_1 \cdot T_0 \xrightarrow{\eta_0 \cdot T_1 \cdot T_0 \cdot \eta_1} T_0 \cdot T_1 \cdot T_0 \cdot T_1 \xrightarrow{\mu} T_0 \cdot T_1$$

Given a distributive law θ between two monads (T_0, η_0, μ_0) and (T_1, η_1, μ_1), a pair of their algebras (A, a_0) and (A, a_1) with the same carrier is *matching*, if it satisfies the condition

$$\begin{array}{ccccc} T_1\,(T_0\,A) & \xrightarrow{\theta\,A} & T_0\,(T_1\,A) & \xrightarrow{T_0\,a_1} & T_0\,A \\ {\scriptstyle T_1\,a_0}\downarrow & & & & \downarrow{\scriptstyle a_0} \\ T_1\,A & & \xrightarrow[a_1]{} & & A \end{array}$$

Matching pairs of algebras are in a bijective correspondence with algebras of the compatible composition.

A matching pair of algebras (A, a_0, a_1) defines an algebra (A, a) via

$$a = \; T_0\,(T_1\,A) \xrightarrow{T_0\,a_1} T_0\,A \xrightarrow{a_0} A$$

An algebra (A, a) induces a matching pair (A, a_0, a_1) via

$$a_0 = \; T_0\,A \xrightarrow{T_0\,(\eta_1\,A)} T_0\,(T_1\,A) \xrightarrow{a} A$$
$$a_1 = \; T_1\,A \xrightarrow{\eta_0\,(T_1\,A)} T_0\,(T_1\,A) \xrightarrow{a} A$$

The counterpart under this bijection of the free algebra $(T_0\,(T_1\,A), \mu\,A)$ of the compatible composition on A is the matching pair $(T_0\,(T_1\,A), \bar{\mu}_0\,A, \bar{\mu}_1\,A)$ where

$$\bar{\mu}_0 = \; T_0 \cdot T_0 \cdot T_1 \xrightarrow{\mu_0 \cdot T_1} T_0 \cdot T_1$$
$$\bar{\mu}_1 = \; T_1 \cdot T_0 \cdot T_1 \xrightarrow{\theta \cdot T_1} T_0 \cdot T_1 \cdot T_1 \xrightarrow{T_0 \cdot \mu_1} T_0 \cdot T_1$$

B Proof of the main theorem

B.1 Proof of Lemma 8

$$\begin{array}{ll} & p \\ = & \\ & \mathsf{fst}\,((\lambda s'.\,(p, *))\,s) \\ = & \quad \{\text{def. of } \eta_0\} \\ & \mathsf{fst}\,((\eta_0 \cdot T_1)\,\{1\}\,(p, *)\,s) \\ = & \quad \{\text{distr. law eq. 1 for } \theta\} \\ & \mathsf{fst}\,((\theta \circ T_1 \cdot \eta_0)\,\{1\}\,(p, *)\,s) \\ = & \quad \{\text{def. of } \eta_0\} \\ & \mathsf{fst}\,(\theta\,\{1\}\,(p, \lambda s'.\,*)\,s) \\ = & \quad \{\text{defs. of } T_1,\, T_0\} \\ & \mathsf{fst}\,((\theta\,\{1\} \circ (T_1 \cdot T_0)\,(\lambda s'.\,*))\,(p, f)\,s) \\ = & \quad \{\text{naturality of } \theta\} \\ & \mathsf{fst}\,(((T_0 \cdot T_1)\,(\lambda s'.\,*) \circ \theta\,\{X\})\,(p, f)\,s) \\ = & \quad \{\text{defs. of } T_0,\, T_1\} \\ & \mathsf{fst}\,(\theta\,\{X\})\,(p, f)\,s) \end{array}$$

B.2 Proof of Theorem 9

Given an action $\downarrow$, we must verify that $\theta : \forall\{X\}.\,T_1\,(T_0\,X) \to T_0\,(T_1\,X)$ defined by $\theta\,\{X\}\,(p, f) = \lambda s.\,(p, f\,(s \downarrow p))$ is a distributive law.

Proof of naturality of θ.

$$
\begin{array}{ll}
 & ((T_0 \cdot T_1)\, g \circ \theta\, \{X\})\, (p, f) \\
= & \quad \{\text{def. of } \theta\} \\
 & (T_0 \cdot T_1)\, g\, (\lambda s.\, (p, f\, (s \downarrow p))) \\
= & \quad \{\text{def. of } T_0,\, T_1\} \\
 & \lambda s.\, (p, g\, (f\, (s \downarrow p))) \\
= & \quad \{\text{def. of } \theta\} \\
 & \theta\, \{Y\}\, (p, g \circ f) \\
= & \quad \{\text{def. of } T_1,\, T_0\} \\
 & (\theta\, \{Y\} \circ (T_1 \cdot T_0)\, g)\, (p, f)
\end{array}
$$

Proof of distributive law equation 1 for θ.

$$
\begin{array}{ll}
 & (\theta \circ T_1 \cdot \eta_0)\, \{X\}\, (p, x) \\
= & \quad \{\text{defs. of } T_1,\, \eta_0\} \\
 & \theta\, \{X\}\, (p, \lambda s'.\, x) \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\, (p, (\lambda s'.\, x)\, (s \downarrow p)) \\
= & \\
 & \lambda s.\, (p, x) \\
= & \quad \{\text{def. of } \eta_0\} \\
 & (\eta_0 \cdot T_1)\, \{X\}\, (p, x)
\end{array}
$$

Proof of distributive law equation 2 for θ.

$$
\begin{array}{ll}
 & (\theta \circ T_1 \cdot \mu_0)\, \{X\}\, (p, f) \\
= & \quad \{\text{defs. of } T_1,\, \mu_0\} \\
 & \theta\, \{X\}\, (p, \lambda s'.\, f\, s'\, s') \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\, (p, f\, (s \downarrow p)\, (s \downarrow p)) \\
= & \quad \{\text{def. of } \mu_0\} \\
 & (\mu_0 \cdot T_1)\, \{X\}\, (\lambda s.\, \lambda s'.\, (p, f\, (s \downarrow p)\, (s' \downarrow p))) \\
= & \quad \{\text{defs. of } T_0,\, \theta\} \\
 & (\mu_0 \cdot T_1 \circ T_0 \cdot \theta)\, \{X\}\, (\lambda s.\, (p, f\, (s \downarrow p))) \\
= & \quad \{\text{def. of } \theta\} \\
 & (\mu_0 \cdot T_1 \circ T_0 \cdot \theta \circ \theta \cdot T_0)\, \{X\}\, (p, f)
\end{array}
$$

Proof of distributive law equation 3 for θ.

$$
\begin{array}{ll}
 & (\theta \circ \eta_1 \cdot T_0)\, \{X\}\, f \\
= & \quad \{\text{def. of } \eta_1\} \\
 & \theta\, \{X\}\, (\mathsf{o}, f) \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\, (\mathsf{o}, f\, (s \downarrow \mathsf{o})) \\
= & \quad \{\text{action eq. 1 for } \downarrow\} \\
 & \lambda s.\, (\mathsf{o}, f\, s) \\
= & \quad \{\text{defs. of } T_0,\, \eta_1\} \\
 & (T_0 \cdot \eta_1)\, \{X\}\, f
\end{array}
$$

Proof of distributive law equation 4 for θ.

$$
\begin{array}{ll}
 & (\theta \circ \mu_1 \cdot T_0)\,\{X\}\,(p,(p',f)) \\
= & \quad \{\text{def. of } \mu_1\} \\
 & \theta\,\{X\}\,(p \oplus p', f) \\
= & \quad \{\text{def. of } \theta\} \\
 & \lambda s.\,(p \oplus p', f\,(s \downarrow (p \oplus p'))) \\
= & \quad \{\text{action eq. 2 for } \downarrow\} \\
 & \lambda s.\,(p \oplus p', f\,((s \downarrow p) \downarrow p')) \\
= & \quad \{\text{defs. of } T_0,\ \mu_1\} \\
 & (T_0 \cdot \mu_1)\,\{X\}\,(\lambda s.\,(p,(p', f\,((s \downarrow p) \downarrow p')))) \\
= & \quad \{\text{def. of } \theta\} \\
 & (T_0 \cdot \mu_1 \circ \theta \cdot T_1)\,\{X\}\,(p, \lambda s.\,(p', f\,(s \downarrow p'))) \\
= & \quad \{\text{defs. of } T_1,\ \theta\} \\
 & (T_0 \cdot \mu_1 \circ \theta \cdot T_1 \circ T_1 \circ \theta)\,\{X\}\,(p,(p',f))
\end{array}
$$

Given a distributive law θ, we must verify that $\downarrow\colon S \times P \to S$ defined by $s \downarrow p = \mathsf{snd}\,(\theta\,\{S\}\,(p, \lambda s'.\,s')\,s)$ is an action.

Proof of action law 1 for $\downarrow$.

$$
\begin{array}{ll}
 & s \downarrow \mathsf{o} \\
= & \quad \{\text{def. of } \downarrow\} \\
 & \mathsf{snd}\,(\theta\,\{S\}\,(\mathsf{o}, \lambda s'.\,s')\,s) \\
= & \quad \{\text{def. of } \eta_1\} \\
 & \mathsf{snd}\,((\theta \circ \eta_1 \cdot T_0)\,\{S\}\,(\lambda s'.\,s')\,s) \\
= & \quad \{\text{distr. law eq. 3 for } \theta\} \\
 & \mathsf{snd}\,((T_0 \cdot \eta_1)\,\{S\}\,(\lambda s'.\,s')\,s) \\
= & \quad \{\text{defs. of } T_0,\ \eta_1\} \\
 & \mathsf{snd}\,((\lambda s'.\,(\mathsf{o}, s'))\,s) \\
= & \\
 & s
\end{array}
$$

Proof of action law 2 for $\downarrow$.

$$
\begin{array}{ll}
 & s \downarrow (p \oplus p') \\
= & \quad \{\text{def. of } \downarrow\} \\
 & \mathsf{snd}\,(\theta\,\{S\}\,(p \oplus p', \lambda s'.\,s')\,s) \\
= & \quad \{\text{def. of } \mu_1\} \\
 & \mathsf{snd}\,((\theta \circ \mu_1 \cdot T_0)\,\{S\}\,(p,(p', \lambda s'.\,s'))\,s) \\
= & \quad \{\text{distr. law eq. 4 for } \theta\} \\
 & \mathsf{snd}\,((T_0 \cdot \mu_1 \circ \theta \cdot T_1 \circ T_1 \cdot \theta)\,\{S\}\,(p,(p', \lambda s'.\,s'))\,s) \\
= & \quad \{\text{def. of } T_1, \text{ Lemma 8, def. of } \downarrow\} \\
 & \mathsf{snd}\,((T_0 \cdot \mu_1 \circ \theta \cdot T_1)\,\{S\}\,(p, \lambda s'.\,(p', s' \downarrow p'))\,s) \\
= & \quad \{\text{defs. of } T_0,\ T_1\} \\
 & \mathsf{snd}\,((T_0 \cdot \mu_1 \circ \theta \cdot T_1 \circ (T_1 \cdot T_0)\,(\lambda s'.\,(p', s' \downarrow p')))\,\{S\}\,(p, \lambda s'.\,s'))\,s) \\
= & \quad \{\text{naturality of } \theta\} \\
 & \mathsf{snd}\,((T_0 \cdot \mu_1 \circ (T_0 \cdot T_1)\,(\lambda s'.\,(p', s' \downarrow p')) \circ \theta)\,\{S\}\,(p, \lambda s'.\,s'))\,s) \\
= & \quad \{\text{Lemma 8, def. of } \downarrow\} \\
 & \mathsf{snd}\,((T_0 \cdot \mu_1 \circ (T_0 \cdot T_1)\,(\lambda s'.\,(p', s' \downarrow p')))\,\{S\}\,(\lambda s'.\,(p, s' \downarrow p))\,s)
\end{array}
$$

$$
\begin{aligned}
&= \quad \{\text{defs. of } T_0,\, T_1\} \\
&\quad \mathsf{snd}\,((T_0 \cdot \mu_1)\,\{S\}\,(\lambda s'.\,(p, (p', (s' \downarrow p) \downarrow p')))\,s) \\
&= \quad \{\text{defs. of } T_0,\, \mu_1\} \\
&\quad \mathsf{snd}\,((\lambda s'.\,(p \oplus p', (s' \downarrow p) \downarrow p'))\,s) \\
&= \\
&\quad (s \downarrow p) \downarrow p'
\end{aligned}
$$

Finally, we have to check that the correspondence is bijective.

$$
\begin{aligned}
&\quad s \downarrow' p \\
&= \quad \{\text{def. of } \downarrow'\} \\
&\quad \mathsf{snd}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s) \\
&= \quad \{\text{def. of } \theta\} \\
&\quad \mathsf{snd}\,(p, \mathsf{id}\,\{S\}\,(s \downarrow p)) \\
&= \\
&\quad s \downarrow p
\end{aligned}
$$

$$
\begin{aligned}
&\quad \theta'\,\{X\}\,(p, f) \\
&= \quad \{\text{def. of } \theta'\} \\
&\quad \lambda s.\,(p, f\,(s \downarrow p)) \\
&= \quad \{\text{Lemma 8, def. of } \downarrow\} \\
&\quad \lambda s.\,(\mathsf{fst}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s), f\,(\mathsf{snd}\,(\theta\,\{S\}\,(p, \mathsf{id}\,\{S\})\,s))) \\
&= \quad \{\text{defs. of } T_0,\, T_1\} \\
&\quad ((T_0 \cdot T_1)\,f \circ \theta\,\{S\})\,(p, \mathsf{id}\,\{S\}) \\
&= \quad \{\text{naturality of } \theta\} \\
&\quad (\theta\,\{X\} \circ (T_1 \cdot T_0)\,f)\,(p, \mathsf{id}\,\{S\}) \\
&= \quad \{\text{defs. of } T_0,\, T_1\} \\
&\quad \theta\,\{X\}\,(p, f)
\end{aligned}
$$

C Algebras of update monads as models of Lawvere theories

In Sections 2.3 and 2.4, we showed three different equivalent definitions of algebras for the update monad for a given act $(S, (P, \mathsf{o}, \oplus), \downarrow)$. An algebra is the same as a model of the (generally large) Lawvere theory corresponding to this monad. Each of the three definitions corresponds to a particular presentation of this Lawvere theory.

The first presentation is given by one operation

$$act : S \to S \to P$$

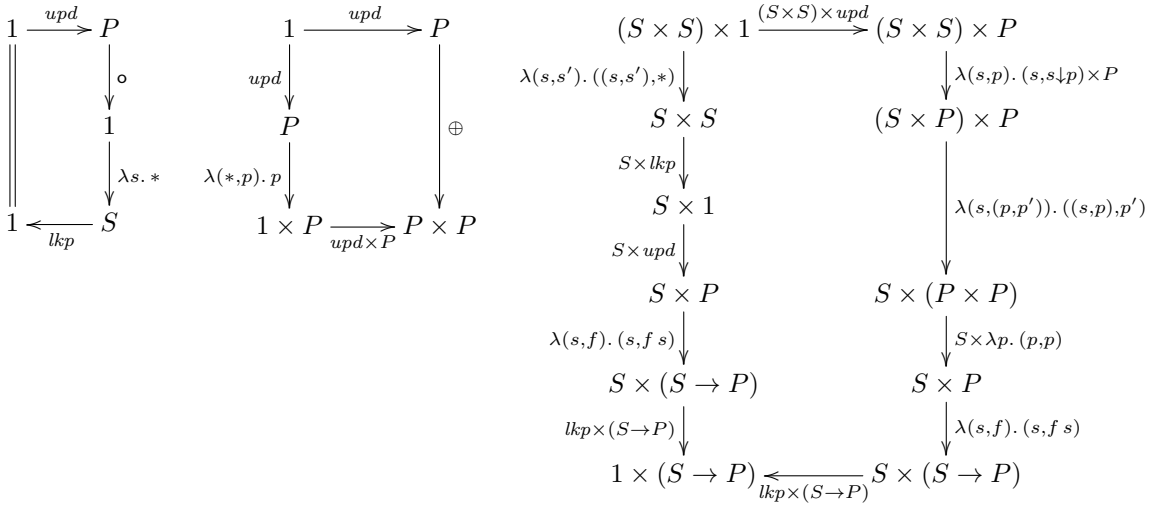
and two equations

$$
\begin{array}{ccc}
1 & \xrightarrow{\lambda s.\,*} & S \\
\| & & \downarrow{\scriptstyle act} \\
1 & \xleftarrow[\lambda *.\lambda s.\,\mathsf{o}]{} & S \to P
\end{array}
\qquad\qquad
\begin{array}{ccc}
S \times S & \xrightarrow{\lambda(s,f).\,(s, f\,s)} & S \times (S \to S) \\
\downarrow{\scriptstyle S \times act} & & \\
S \times (S \to P) & & \\
\downarrow{\scriptstyle \lambda(s,f).\,(s, f\,s)} & & \downarrow{\scriptstyle act \times (S \to S)} \\
S \times (S \to S \to P) & & \\
\downarrow{\scriptstyle act \times (S \to S \to P)} & & \\
(S \to P) \times (S \to S \to P) & \xleftarrow[\lambda(f,g).\,(\lambda s.\,f\,s \oplus g\,s\,(s \downarrow f\,s), \lambda s.\,s \downarrow f\,s)]{} & (S \to P) \times (S \to S)
\end{array}
$$

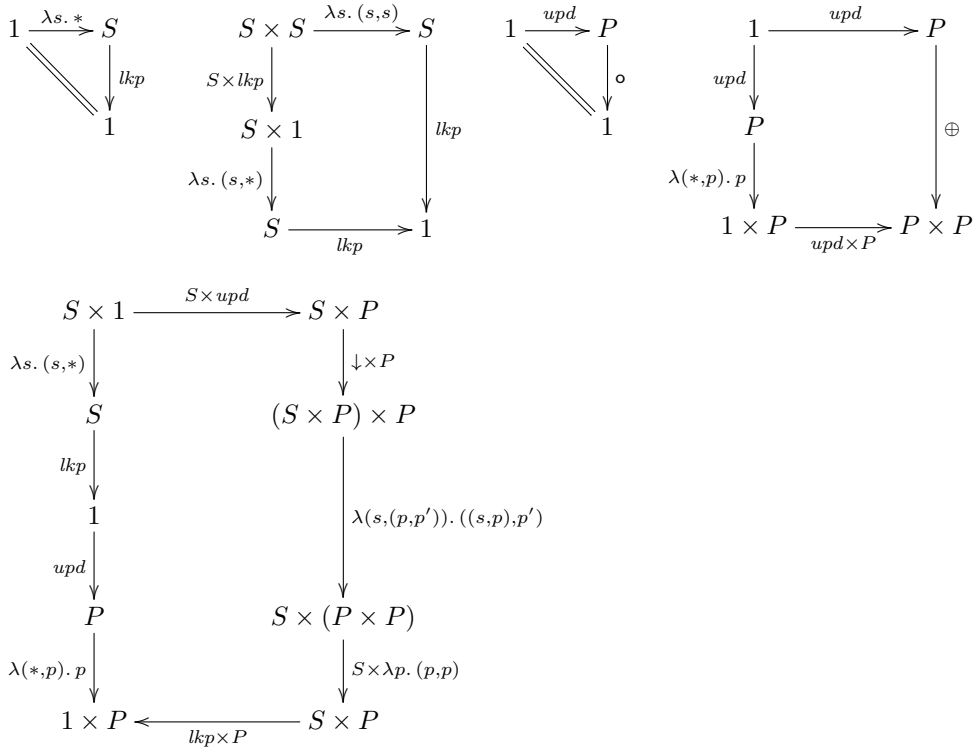
The second and third resemble Melliès's and Plotkin's variations[7] of Plotkin and Power's presentation of the theory of (global) state [14]. The second is given by two operations

$lkp : S \to 1$
$upd : 1 \to P$

satisfying



The third has the same operations, but different equations:



[7] P.-A. Melliès. String diagrams in logic and computer science. Slides from lecture 6 from course held at ITU Copenhagen, Apr. 2011. G. Plotkin. Algebraic effects. Slides from Logic and Interaction, Marseille, Feb. 2012.

A "Game Semantical" Intuitionistic Realizability Validating Markov's Principle

Federico Aschieri[*,1] **and Margherita Zorzi**[†,2]

1 **Laboratoire de l'Informatique du Parallélisme (UMR 5668, CNRS, UCBL) École Normale Supérieure de Lyon – Université de Lyon, France**
2 **Dipartimento di Informatica, Università di Verona, Italy**

Abstract

We propose a very simple modification of Kreisel's modified realizability in order to computationally realize Markov's Principle in the context of Heyting Arithmetic. Intuitively, realizers correspond to arbitrary strategies in Hintikka-Tarski games, while in Kreisel's realizability they can only represent winning strategies. Our definition, however, does not employ directly game semantical concepts and remains in the style of functional interpretations. As term calculus, we employ a purely functional language, which is Gödel's System T enriched with some syntactic sugar.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Markov's principle, intuitionistic realizability, Heyting arithmetic, game semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.24

1 Introduction

1.1 Markov's Argument

Given a recursive function $f : \mathbb{N} \to \mathbb{N}$, if it is impossible that for every natural number n, $f(n) \neq 0$, then there exists an n such that $f(n) = 0$. This classically true statement has come to be universally known as Markov's Principle, and was introduced by Markov in the context of his theory of Constructive Recursive Mathematics (see e.g. [23]). Markov's original argument for it was simply the following: if it is not possible that for all n, $f(n) \neq 0$, then by computing in sequence $f(0), f(1), f(2), \ldots$, one will eventually hit a number n such that $f(n) = 0$, which can be *effectively* recognized as a witness. For the rest of the paper we shall consider the formalization of Markov's principle in Heyting Arithmetic, that is the axiom scheme

$$\mathsf{MP} : \neg\forall x^{\mathtt{N}} P \to \exists x^{\mathtt{N}} P^{\perp}$$

where P is a decidable predicate and $P^{\perp}$ its negation.

Markov's justification of his own principle is hardly satisfying from a constructive point of view; the intuitionistic school of Brouwer, indeed, rejected it. It is true that, following

* This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR)
† Partially supported by LINTEL (Linear Techniques For The Analysis Of Languages), `https://sites.google.com/site/tolintel/`

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 24–44

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

Markov's argument, one can recursively realize MP using Kleene's realizability interpretation [13], thus providing a computational interpretation of it. However, a Kleene realizer just blindly searches for a witness of the conclusion, without even considering the possible constructive content of a proof of the premise. In other terms, such a realizer does not embody the meaningful transformation of a proof of $\neg\forall x^{\mathtt{N}}P$ into a proof of $\exists x^{\mathtt{N}}P^{\perp}$ which is demanded by the Brouwer-Heyting-Kolmogorov reading of logical constants [23]. However, when added to Heyting Arithmetic, MP gives rise to a *constructive* system enjoying the disjunction and the existential witness property [21] (if a disjunction is derivable, one of the disjoint is derivable too, and if an existential statement is derivable, so it is one instance of it). So a better interpretation of MP can and must be provided. In this article we shall try to answer, in particular, to the following question: is it possible to realize Markov's Principle just using a functional language and a simple intuitionistic realizability?

1.2 Gödel's Dialectica Interpretation

A much more refined constructive justification of Markov's Principle was in fact introduced by Gödel [10]. Indeed, the idea behind Gödel's Dialectica Interpretation is so refined, that it forms the basis for all subsequent constructive interpretations of MP [5, 11]. As pointed out by Diller [6], a very satisfying constructive justification of MP is indeed hidden in the Dialectica, and is the following. A formal proof of $\neg\forall x^{\mathtt{N}}P$ is a natural deduction of $\perp$ from the hypothesis $\forall x^{\mathtt{N}}P$. If we consider a normal form of this proof, we have actually a deduction of $\perp$ from finitely many instances $P(t_1),\dots,P(t_n)$; so one of them must be false and we get a t_i such that $P^{\perp}(t_i)$, and t_i reduces to some numeral n. Thus, as required in the BHK semantics, from any proof of the premise of Markov's Principle one can effectively extract a witness for the conclusion, without having to run a blindfold process.

More in detail, Gödel's interpretation of implication allows one to describe a realizer of the premise $\neg\forall x^{\mathtt{N}}P$ of MP as a functional mapping a witness for $\forall x^{\mathtt{N}}P$ (essentially, something void) into a possible counterexample to $\forall x^{\mathtt{N}}P$. If this counterexample works, one witness $\exists x^{\mathtt{N}}P^{\perp}$, otherwise one has refuted the realizer of the premise of MP.

Gödel's Dialectica is thus very interesting and, rather remarkably, allows to computationally interpret any proof in Heyting Arithmetic plus MP with a term in a simple and purely functional language, Gödel's system T. However, in spite of its simple interpretation of MP, the Dialectica is a rather involved translation, which burdens a lot the reading of implication, making it particularly painful to unravel in presence of nested implications in the translated formula, as it is often the case. It is also quite cumbersome to decorate natural deductions with Gödel realizers. Is it really needed all this complication if one wants just to interpret Markov's Principle?

1.3 Kreisel's Modified Realizability

Inspired by Gödel's interpretation, Kreisel put forward his modified realizability [14, 15] as a simplification of the Dialectica, which is actually equivalent to it in the case of formulas without implications (Oliva [17]). In modified realizability, the familiar BHK reading of implication is restored – which originates the main simplification – and the term assignment for proofs can be taken as a pleasant intuitionistic Curry-Howard correspondence. Unfortunately, Kreisel introduced modified realizability with the *specific aim* of showing that Markov's principle is not realizable in the syntactical model made by the terms of Gödel's T. So one is left with a very good intuitionistic realizability, which is not able to concretely realize MP.

1.4 Modified Realizability and Friedman's Translation

The Friedman translation is a strikingly simple device introduced by Friedman [7] in order to prove closure of intuitionistic systems S under Markov's rule:

$$S \vdash \neg\forall x^{\mathbb{N}} P \implies S \vdash \exists x^{\mathbb{N}} \neg P$$

where P is any decidable quantifier free formula. While combining Friedman's translation with modified realizability allows to interpret any *fixed* instance of MP, the situation does not improve too much because it is not possible to validate the full axiom scheme MP. In other terms, if a proof contains more than one instance of MP, combining Friedman's translation with modified realizability is not enough to interpret it.

Indeed, one possible solution to this issue, due to Coquand-Hofmann [5], is to first make Friedman's translation more flexible by using a somewhat unusual forcing [3] and then combining the result with modified realizability. We seek however a simpler and less ad hoc modification of modified realizability.

1.5 Game Semantics and Functional Interpretations

What's wrong with modified realizability? The problem is that it is not a refined game semantics, which is really the framework needed to explain constructively classical principles (see e.g. [4, 1]). Instead, the Dialectica is better suited to represent dialogues among players – i.e. proofs and tests – which arise in classical game semantics.

The standard way to associate a game to an arithmetical formula A is to consider an interaction between two players who debate A; the first player – usually called Eloise – tries to show that it is true, while the second player – usually called Abelard – tries to show that the formula is false. Thus, Eloise wins when true atomic formulas are on the board while Abelard wins with false ones. In the case of formulas of the shape $A \vee B, \exists x^{\mathbb{N}} A$, Eloise moves: in the first case by choosing A or B and in the second case by choosing an instance $A(n)$, where n is intended to be a witness for the existential quantifier. In the case of formulas of the shape $A \wedge B, \forall x^{\mathbb{N}} A$, Abelard moves: in the first case by choosing A or B and in the second case by choosing an instance $A(n)$, where n is intended to be a counterexample to the universal quantifier. This kind of game was introduced by Hintikka [12] and it is also known as Tarski game.

As far as $\to$-free formulas are concerned, modified realizability and Dialectica agree (Oliva [17]): a realizer represents in both cases a winning strategy for Eloise, that is, a way of selecting moves that allows Eloise to win every play, no matter how Abelard plays. But in the case of formulas of the shape $A \to B$, according to modified realizability, Abelard should give Eloise a *winning* strategy for A, and then the game for B is played; while according to Dialectica, Abelard should give Eloise *some* strategy for A and then the game for B is played, and either Eloise wins this game, or "temporarily" looses it, but still with the possibility of winning the whole game if she manages to show that the strategy offered by Abelard was not winning. This second way of formulating the game for $\to$ is much better, since the first one is not concretely playable: how to establish effectively whether the strategy given by Abelard to Eloise is winning? In the case of Dialectica, Abelard is given a chance to play the game for the premise A without necessarily having to play in *the best* way possible, but just at *his best*, as in real life games.

1.6 A Game Semantical Twist of Modified Realizability

The goal of the present paper is to tweak modified realizability in such a way that its game semantical content is improved and made more similar to the one of Dialectica, while retaining the simplicity and the appeal of Kreisel's original definition. One should allow realizers to be not only winning strategies, but arbitrary ones, thus allowing poor Abelard to have more chances to play in the game for the formula $A \to B$. True realizers – among which those extracted from proofs – should be winning strategies, but in the concept of realizability should appear also weaker realizers, that is, arbitrary strategies.

1.7 Plan of the paper

In Section §2 the term calculus $\mathcal{T}$ in which realizers are written and the language of the arithmetical theory $\mathsf{HA}^\omega + \mathsf{MP}$ are introduced. In Section §3 we give our definition of realizability. In Section §4 an extensionality property of $\mathcal{T}$ is introduced and discussed as a crucial tool for studying the realizer of the Markov's Principle, defined in Section §5. Section §6 is devoted to prove our main result, that every theorem of $\mathsf{HA}^\omega + \mathsf{MP}$ is realizable; also the relationship between our notion of realizability and truth is discussed. Conclusions and considerations about future works are in Section §7.

2 The Term Calculus

In this section we introduce the typed lambda calculus $\mathcal{T}$ in which realizers are written. System $\mathcal{T}$ is obtained from Gödel's T (see [8, 9]) by adding a new atomic type U and new operations on it. The basic objects of $\mathcal{T}$ are numerals ($\mathsf{S}\dots\mathsf{S}0$), booleans ($\mathtt{True}, \mathtt{False}$) and its basic computational constructs are primitive recursion at all types (R), if-then-else (if), pairs, as in Gödel's T. Terms of the form $\mathsf{if}_A\, t_1\, t_2\, t_3$ will be sometimes written in the far more legible form $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$. $\mathcal{T}$, which is formally described in Figure 1, also includes:

- two denumerable sets of constants of type U, namely $\top_0, \top_1, \top_2, \dots$ and $\bot_0, \bot_1, \bot_2, \dots$;
- two constants tt and ff of type $\mathtt{N} \to \mathsf{U}$: they transform numerals n into, respectively, the constants $\top_n$ and $\bot_n$; these are also the only constructs of the system that can generate constants of type U;
- the constant quote of type $\mathsf{U} \to \mathtt{N}$: quote takes as argument any constant $\top_n$ or $\bot_n$ and transform it into the numeral n. In other terms, quote takes a constant of type U and returns its Gödel number, which is its position in the enumeration. However – and this will be crucial in the following! – quote is not able to tell *from which* enumeration its argument comes from and it just returns its position n, which may thus refer to the ordering $\top_0, \top_1, \top_2, \dots$ as well as the ordering $\bot_0, \bot_1, \bot_2, \dots$. Therefore, quote is partially blind with respect to constants of type U: of its argument $\top_n$ or $\bot_n$, it sees only something like $?_n$ – i.e. the index n.

These non-standard features of $\mathcal{T}$ notwithstanding, the type U and all the constants $\mathsf{tt}, \mathsf{ff}, \mathsf{quote}$ are just syntactic sugar. Indeed, the type U can be encoded in Gödel's T as $\mathtt{Bool} \times \mathtt{N}$; then $\top_n$ and $\bot_n$ can be encoded respectively as $\langle \mathtt{True}, n\rangle$ and $\langle \mathtt{False}, n\rangle$; tt and ff can be encoded respectively as $\lambda x^{\mathtt{N}}\, \langle \mathtt{True}, x\rangle$ and $\lambda x^{\mathtt{N}}\, \langle \mathtt{False}, x\rangle$, and quote as the term $\lambda x^{\mathtt{Bool}\times\mathtt{N}}\, \pi_1(x)$. The typing rules for $\mathsf{tt}, \mathsf{ff}, \mathsf{quote}$ fully agree with the above encodings. So its clear that $\mathcal{T}$ is still a purely functional language; however, in order to be able to reason about it in a more refined way, we have found necessary to add the new type and constants as primitive constructs.

Types

$$\sigma, \tau ::= \mathtt{N} \mid \mathtt{Bool} \mid \mathsf{U} \mid \sigma \to \tau \mid \sigma \times \tau$$

Constants

$$\top_0, \top_1, \top_2, \ldots$$
$$\bot_0, \bot_1, \bot_2, \ldots$$
$$c ::= \mathsf{R}_\tau \mid \mathsf{if}_\tau \mid 0 \mid \mathsf{S} \mid \mathtt{True} \mid \mathtt{False} \mid \top_i\,(i \in \mathbb{N}) \mid \bot_i\,(i \in \mathbb{N}) \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{quote}$$

Terms

$$t, u ::= c \mid x^\tau \mid tu \mid \lambda x^\tau u \mid \langle t, u\rangle \mid \pi_0 u \mid \pi_1 u$$

Typing Rules for Variables and Constants

$$x^\tau : \tau \mid 0 : \mathtt{N} \mid \mathsf{S} : \mathtt{N} \to \mathtt{N} \mid \mathtt{True} : \mathtt{Bool} \mid \mathtt{False} : \mathtt{Bool} \mid$$
$$\top_i : \mathsf{U} \text{ for every } i \in \mathbb{N} \mid \bot_i : \mathsf{U} \text{ for every } i \in \mathbb{N}$$
$$\mathsf{tt} : \mathtt{N} \to \mathsf{U} \mid \mathsf{ff} : \mathtt{N} \to \mathsf{U} \mid \mathsf{quote} : \mathsf{U} \to \mathtt{N}$$
$$\mathsf{if}_\tau : \mathtt{Bool} \to \tau \to \tau \to \tau \mid \mathsf{R}_\tau : \tau \to (\mathtt{N} \to (\tau \to \tau)) \to \mathtt{N} \to \tau$$

Typing Rules for Composed Terms

$$\frac{t : \sigma \to \tau \qquad u : \sigma}{tu : \tau} \qquad \frac{u : \tau}{\lambda x^\sigma u : \sigma \to \tau} \qquad \frac{u : \sigma \qquad t : \tau}{\langle u, t\rangle : \sigma \times \tau} \qquad \frac{u : \tau_0 \times \tau_1}{\pi_i u : \tau_i}\ i \in \{0, 1\}$$

Reduction Rules All the usual reduction rules for simply typed lambda calculus (see Girard [8]) plus the rules for recursion, if-then-else and projections

$$\mathsf{R}_\tau uv0 \mapsto u \quad \mathsf{R}_\tau uv\mathsf{S}(t) \mapsto vt(\mathsf{R}_\tau uvt) \quad \mathsf{if}_\tau\,\mathtt{True}\,u\,v \mapsto u \quad \mathsf{if}_\tau\,\mathtt{False}\,u\,v \mapsto v \quad \pi_i\langle u_0, u_1\rangle \mapsto u_i, i = 0, 1$$

plus the following ones, assuming n be a numeral:

$$\mathsf{tt}n \mapsto \top_n \qquad \mathsf{ff}n \mapsto \bot_n$$
$$\mathsf{quote}\,\top_m \to m \qquad \mathsf{quote}\,\bot_m \to m$$

Figure 1 The extension $\mathcal{T}$ of Gödel's system T.

It is easy provable that $\mathcal{T}$ is strongly normalizing and has the uniqueness-of-normal-form property:

▶ **Theorem 1** (Strong Normalization and Weak Church-Rosser). *The system $\mathcal{T}$ enjoys strong normalization and weak-Church-Rosser (uniqueness of normal forms) for all closed terms of atomic types* $\mathtt{N}, \mathtt{Bool}$ *or* U.

Proof. By the translation of $\mathcal{T}$ into T. ◀

The following normal form theorem for $\mathcal{T}$ also holds.

▶ **Theorem 2** (Normal Form Property for $\mathcal{T}$). *Assume A is either an atomic type* $\mathtt{N}, \mathtt{Bool}, \mathsf{U}$ *or a product type. Then any closed normal term $t \in \mathcal{T}$ of type A is: a numeral $n : \mathtt{N}$, or a boolean* $\mathtt{True}, \mathtt{False} : \mathtt{Bool}$, *or a constant $\top_i : \mathsf{U}$, or a constant $\bot_i : \mathsf{U}$, or a pair $\langle u, v\rangle : B \times C$.*

Proof. As in Lemma 5 in [2]. ◀

From now onwards, for every pair of terms t, u of System $\mathcal{T}$, we shall write $t = u$ if they are the same term modulo the equality rules corresponding to the reduction rules of System $\mathcal{T}$ (equivalently, if they have the same normal form).

Finally, we define two sets of terms:

$$\mathbb{T} := \{t \mid t \text{ is a term of } \mathcal{T} \text{ and } t = \top_i \text{ for some } i \in \mathbb{N}\}$$

and

$$\mathbb{\perp\!\!\!\perp} := \{t \mid t \text{ is a term of } \mathcal{T} \text{ and } t = \bot_i \text{ for some } i \in \mathbb{N}\}.$$

2.1 Language of $\mathsf{HA}^\omega + \mathsf{MP}$

We now define the language of the arithmetical theory $\mathsf{HA}^\omega + \mathsf{MP}$.

▶ **Definition 3** (Language of $\mathsf{HA}^\omega + \mathsf{MP}$). The language $\mathcal{L}$ of $\mathsf{HA}^\omega + \mathsf{MP}$ is defined as follows.

1. The terms of $\mathcal{L}$ are all $t \in \mathcal{T}$.
2. The atomic formulas of $\mathcal{L}$ are all $Q \in \mathcal{T}$ such that $Q : \mathtt{Bool}$.
3. The formulas of $\mathcal{L}$ are built from atomic formulas of $\mathcal{L}$ by the connectives $\vee, \wedge, \to \forall, \exists$ as usual, with quantifiers possibly ranging over variables x^τ, y^τ, z^τ of arbitrary finite type τ of T.

We denote with $\bot$ the atomic formula $\mathtt{False}$. With $P^\bot$ we denote the complement of the predicate P, that is, $\mathsf{if}\ P\ \mathsf{then}\ \mathtt{False}\ \mathsf{else}\ \mathtt{True}$. If P is an atomic formula of $\mathcal{L}$ in the free variables $x_1^{\tau_1}, \dots, x_n^{\tau_n}$ and $t_1 : \tau_1, \dots, t_n : \tau_n$ are terms of $\mathcal{L}$, with $P(t_1, \dots, t_n)$ we shall denote the atomic formula $P[t_1/x_1, \dots, t_n/x_n]$.

3 Realizability

For every formula A of $\mathcal{L}$, we are now going to define what type $|A|$ realizers of A must have.

▶ **Definition 4** (Types for realizers). For each formula A of $\mathcal{L}$ we define a type $|A|$ of $\mathcal{T}$ by induction on A:

$$|P| = \mathtt{U} \text{ if } P \text{ is atomic} \qquad |A \wedge B| = |A| \times |B| \qquad |A \to B| = |A| \to |B|$$
$$|A \vee B| = \mathtt{Bool} \times (|A| \times |B|) \qquad |\forall x^\tau A| = \tau \to |A| \qquad |\exists x^\tau A| = \tau \times |A|$$

We remark that any HA^ω term of type $|A|$, by definition, can be taken to represent an arbitrary strategy for Eloise in the Hintikka-Tarski game for A. For example, a term

$$t : |\forall x^\tau A| = \tau \to |A|$$

takes a move $u : \tau$ by Abelard, corresponding to the game $A[u/x^\tau]$, and gives Eloise the strategy tu to follow for the continuation. A term

$$t : |\exists x^\tau A| = \tau \times |A|$$

gives Eloise a move to play, $\pi_0 t = u$, and a strategy $\pi_1 t$ for continuing the game $A[u/x^\tau]$. For precise definitions of Hintikka-Tarski games and strategies we refer to [1]. In this paper, however, we do not need to examine these concepts in further detail, because game semantical notions will be just used as guidelines to understand intuitively the realizability that we are going to introduce.

Let now $\mathsf{p}_0 := \pi_0 : \sigma_0 \times (\sigma_1 \times \sigma_2) \to \sigma_0$, $\mathsf{p}_1 := \pi_0\pi_1 : \sigma_0 \times (\sigma_1 \times \sigma_2) \to \sigma_1$ and $\mathsf{p}_2 := \pi_1\pi_1 : \sigma_0 \times (\sigma_1 \times \sigma_2) \to \sigma_2$ be the three canonical projections from $\sigma_0 \times (\sigma_1 \times \sigma_2)$.

We define the realizability relation $t \Vdash F$, where $t \in \mathcal{T}$ and F is a formula:

▶ **Definition 5** (Realizability). For each closed formula F and closed term $t : |F|$ of System $\mathcal{T}$, we define a relation $t \Vdash F$ of HA^ω by induction on F as follows:

1. $t \Vdash Q$ if and only if ($Q = \mathtt{True}$ and $t \in \mathbb{T}$) or ($Q = \mathtt{False}$ and $t \in \mathbb{\perp\!\!\!\perp}$) for Q atomic formula;
2. $t \Vdash A \wedge B$ if and only if $\pi_0 t \Vdash A$ and $\pi_1 t \Vdash B$;
3. $t \Vdash A \vee B$ if and only if $\mathsf{p}_0 t = \mathtt{True}$ and $\mathsf{p}_1 t \Vdash A$ or $\mathsf{p}_0 t = \mathtt{False}$ and $\mathsf{p}_2 t \Vdash B$;
4. $t \Vdash A \to B$ if and only if for all u, if $u \Vdash A$, then $tu \Vdash B$;
5. $t \Vdash \exists x^\tau A$ if and only if $\pi_0 t = u$ for $u : \tau$ closed term of HA^ω and $\pi_1 t \Vdash A[u/x]$;
6. $t \Vdash \forall x^\tau A$ if and only if for all closed term $u : \tau$ of HA^ω, $tu \Vdash A[u/x]$.

We remark that the clauses 2–6 of our realizability relation coincide exactly with those of modified realizability for the corresponding formulas. Our definition tweaks modified realizability in two other ways. Firstly, instead of considering Gödel's T as canonical term model, we take $\mathcal{T}$. Secondly, we modify in a crucial way the realizability condition for atomic formulas. In modified realizability P is realizable by any term if it is true, while not realizable if it is false; in our case, a realizer of P is a term which just computes the truth value of P and returns it under the form of a constant belonging to $\mathbb{T}$ or to $\mathbb{\perp\!\!\!\perp}$.

In game semantical language, a realizer of P just determines the outcome of the Hintikka-Tarski game for P, returning a constant belonging to $\mathbb{T}$ or to $\mathbb{\perp\!\!\!\perp}$ according as to whether Eloise or Abelard wins. The intuition is that, as anticipated in the introduction, we want arbitrary strategies to realize formulas. This forces atomic formulas to be realizable regardless of their truth value, and we just need the truth value to be reflected by realizers. Of course, realizer coming from proofs will have an extra condition that will prevent them from realizing false formulas, as we shall soon see. We shall also show that any closed formula of HA^ω is realizable: any strategy t for A can be mapped into a realizer t_A of A which follows the strategy t. All that implies a crucial change in the meaning with respect to modified realizability also for implication. Since arbitrary strategies can be turned into realizers, a realizer of $A \to B$ will map not only winning strategies for A into winning strategies for B, but also realizers/arbitrary-strategies for A into realizers/arbitrary-strategies for B.

▶ **Definition 6** (Translation of Arbitrary Strategies)**.** Let A be any formula and $t : |A|$ any term of HA^ω containing all the free variables of A. We define by induction on A a term t_A of $\mathcal{T}$ with free variables containing those of A:

- If P is atomic, then
$$t_P := \mathsf{if}\ P\ \mathsf{then}\ \top_0\ \mathsf{else}\ \bot_0$$
- $t_{A\wedge B} := \langle (\pi_0 t)_A, (\pi_1 t)_B \rangle$ $\quad t_{A\vee B} := \langle \mathsf{p}_0 t, (\mathsf{p}_1 t)_A, (\mathsf{p}_2 t)_B \rangle$ $\quad t_{A\to B} := \lambda x^{|A|}.\,(tx)_B$
- $t_{\forall x^\tau A} := \lambda x^\tau.\,(tx)_A$ $\quad t_{\exists x^\tau A} := \langle \pi_0 t, (\pi_1 t)_{A[\pi_0 t/x^\tau]} \rangle$

where x is fresh.

▶ **Proposition 7** (Arbitrary Strategies and Realizability)**.** *Let A be any closed formula and $t : |A|$ any closed term of HA^ω. Then*
$$t_A \Vdash A$$

Proof. We proceed by induction on A. We cover only few representative cases, the others being similar.

1. $A = P$, with P atomic. Then
$$t_P := \mathsf{if}\ P\ \mathsf{then}\ \top_0\ \mathsf{else}\ \bot_0$$
Now, if $P = \texttt{True}$, then $t_P = \top_0 \in \mathbb{T}$, so $t_P \Vdash P$; if $P = \texttt{False}$, then $t_P = \bot_0 \in \mathbb{\perp\!\!\!\perp}$, so $t_P \Vdash P$.
2. $A = B \to C$. Then
$$t_A := \lambda x^{|B|}.\,(tx)_C$$
Now, suppose $u \Vdash B$. We have to show $t_A u \Vdash C$. But it is easy to see that
$$t_A u = (tx)_C[u/x] = (tu)_C$$
and by inductive hypothesis $(tu)_C \Vdash C$. We thus conclude by Lemma 9 that $t_A \Vdash A$.

3. $A = \exists x^\tau B$. Then

$$t_A := \langle \pi_0 t, (\pi_1 t)_{B[\pi_0 t/x^\tau]} \rangle$$

Since by inductive hypothesis

$$(\pi_1 t)_{B[\pi_0 t/x^\tau]} \Vdash B[\pi_0 t/x^\tau]$$

we conclude by Lemma 9 that $t_A \Vdash A$.

◀

In the following, we will focus on a particular class of terms, called *proof-like*. These are the terms that are extracted from the actual proofs, and that neither contain any constant from the set $\bot\!\!\!\bot$ nor have the possibility of generating them with a constant ff.

▶ **Definition 8** (Proof-like Terms)**.** A proof-like term is a term t of $\mathcal{T}$ which does not contain constants of the form $\bot_i$ $(i \in \mathbb{N})$ or ff.

In the following, the "true" realizers will be proof-like terms. They actually represent winning strategies, that is, they carry sound constructive information about the formula they realize.

The concept of proof-like realizer is also crucial to determine a meaningful interaction between strategies in the definition of realizability for implication. For instance, suppose that some proof-like term t realizes a formula $A \to B$, where for simplicity A and B are $\to$-free. Let u be a realizer of A. Then tu must realize B. Since tu is not necessarily proof-like, tu may not represent a winning strategy for B. For example, assume $B = \forall x^{\mathtt{N}} \exists y^{\mathtt{N}} P(x, y)$; then there could be a numeral n such that if we let $m = \pi_0(tun)$, then $P(n, m) = \mathtt{False}$. n is a *test* that refutes the realizer tu, when seen as a strategy for B. Now, the term $\pi_1(tun)$, which realizes $P(n, m)$, must reduce to a constant in $\bot\!\!\!\bot$. Since t is proof-like, such a constant must be produced by the term u in the reduction of $\pi_1(tun)$; namely, a test must be produced that refutes u as well, when seen a strategy. For example, if $A = \exists x^{\mathtt{N}} \forall y^{\mathtt{N}} Q(x, y)$, in the reduction of $\pi_1(tun)$, $\pi_1 u$ must be applied to some numeral j such that $\pi_0 u = i$ and $Q(i, j) = \mathtt{False}$. In that case, a constant in $\bot\!\!\!\bot$ is produced, and it may actually be the constant which is the normal form of $\pi_1(tun)$.

We point out that this behaviour of realizers of implications is analogous to that of terms witnessing the Dialectica interpretation of implications.

The next Lemma tells that realizability respects the notion of equality of $\mathcal{T}$ terms: if two terms can be proved equal in $\mathcal{T}$, then then they realize the same formulas.

▶ **Lemma 9.** *If $t_1 = t_2$ and $u_1 = u_2$ are valid in $\mathcal{T}$, then $t_1 \Vdash A[u_1/x]$ if and only if $t_2 \Vdash A[u_2/x]$ for each formula A.*

Proof. By induction on the formula A. ◀

4 Extensionality

Proving that Markov's Principle is realizable by a proof-like term is by no means trivial. The goal of this section is to introduce a key tool that will let us describe an important kind of extensionality property of System $\mathcal{T}$. Afterwards, we shall be able to reason in a more sophisticated way about terms of $\mathcal{T}$, and in particular about the realizer of MP that we shall propose.

A basic feature of typed functional lambda calculi is extensionality: in concrete computations, there is no way to discriminate syntactically different terms if, denotationally, they

represent the same function. For example, suppose t and u are two terms of T of type $\mathtt{N}^2 \to \mathtt{N}$ implementing in different way the addition function. For instance, t may perform recursion on the first argument and u on the second. The two terms represent the same function, but they are syntactically different normal forms. Nevertheless, any term $\Psi : (\mathtt{N}^2 \to \mathtt{N}) \to \mathtt{N}$ of T will not be able to discriminate t and u: Ψt and Ψu will convert to the same numeral.

Another characteristic of typed lambda calculi is the impossibility of distinguishing different *mute* constants, which are the constants whose associated reduction rules cannot leak any information about their shape. If we take a term t and permute its mute constants obtaining t', the normal form of t' can be obtained from the normal form of t by the same permutation of constants. To put it differently, mute constants can be moved around and duplicated inside a term, but they have no influence whatsoever on the evolution of the computation. Now, while the constants $\mathtt{True}, \mathtt{False}, \mathsf{S}, 0$ can be discriminated (by if and R), the constants of the form $\top_n, \bot_n$ do not. They are not completely mute since their indexes can be recognized by quote, but their main form ($\bot, \top$) cannot be determined by any reduction rule.

All these considerations lead us to the concept of extensionality modulo a relation $\mathcal{R}$ over the base type U. Here, $\mathcal{R}$ relates terms which should be regarded as *almost*, or *observationally*, equal. If we take the usual definition of extensionality and, instead of fixing it to be equality at type U, we let it to be $\mathcal{R}$, we determine a more flexible concept of extensionality, relating objects which can well be different, but cannot be computationally distinguished. Now, let us consider any reflexive binary relation $\mathcal{R}$ between closed terms of type U of $\mathcal{T}$. $\mathcal{R}$ is said to be *saturated with respect to equality* if for every t_1, t_2, u_1, u_2, if $t_1 \mathcal{R} t_2$ and $t_1 = u_1$ and $t_2 = u_2$, then $u_1 \mathcal{R} u_2$.

▶ **Definition 10** (Extensionality Modulo a Relation)**.** Let t and u two closed terms of $\mathcal{T}$ of type ρ and $\mathcal{R}$ a reflexive relation between closed terms of type U of $\mathcal{T}$ saturated with respect to equality. We define the *extensionality relation* $t \sim_{\mathcal{R}} u$ by induction on the type ρ:

- If $\rho = \mathsf{U}$, then $t \sim_{\mathcal{R}} u$ if and only if $t \mathcal{R} u$;
- If $\rho = \mathtt{N}$, then $t \sim_{\mathcal{R}} u$ if and only if $t = u$;
- If $\rho = \mathtt{Bool}$, then $t \sim_{\mathcal{R}} u$ if and only if $t = u$;
- If $\rho = \tau \to \sigma$, then $t \sim_{\mathcal{R}} u$ if and only if $\forall v : \tau \, \forall w : \tau.\ v \sim_{\mathcal{R}} w$ implies $tv \sim_{\mathcal{R}} uw$;
- If $\rho = \tau \times \sigma$, then $t \sim_{\mathcal{R}} u$ if and only if $\pi_0 t \sim_{\mathcal{R}} \pi_0 u$ and $\pi_1 t \sim_{\mathcal{R}} \pi_1 u$.

Intuitively, a closed term t of $\mathcal{T}$ is extensional modulo $\mathcal{R}$ if $t \sim_{\mathcal{R}} t$. Let us now prove that the relation $\sim_{\mathcal{R}}$ as well is saturated with respect to equality.

▶ **Lemma 11.** *Given u_1, u_2, t_1, t_2 closed terms of $\mathcal{T}$ of type σ, suppose $u_1 = t_1$, $u_2 = t_2$ and $u_1 \sim_{\mathcal{R}} u_2$. Then $t_1 \sim_{\mathcal{R}} t_2$.*

Proof. By induction on the type σ.

- $\sigma = \mathsf{U}$: the thesis follows by saturation of the relation $\mathcal{R}$.
- $\sigma = \mathtt{N}$ or $\rho = \mathtt{Bool}$: by Definition 10, $u_1 = u_2$, so $t_1 = t_2$ and we conclude $t_1 \sim_{\mathcal{R}} t_2$.
- $\sigma = \rho \to \tau$. Let us consider any pair of terms $r : \rho$ and $s : \rho$ such that $r \sim_{\mathcal{R}} s$. By Definition 10 of the extensionality relation and by the fact that $u_1 \sim_{\mathcal{R}} u_2$, it holds that $u_1 r \sim_{\mathcal{R}} u_2 s$. Now, we can apply the inductive hypothesis to the type τ of the terms $u_1 r$ and $u_2 s$: since $u_1 r = t_1 r$ and $u_2 s = t_2 s$, we have $t_1 r \sim_{\mathcal{R}} t_2 s$. Therefore, by Definition 10, $t_1 \sim_{\mathcal{R}} t_2$.
- $\sigma = \rho \times \tau$. By Definition 10, $u_1 \sim_{\mathcal{R}} u_2$ implies that $\pi_0 u_1 \sim_{\mathcal{R}} \pi_0 u_2$ and $\pi_1 u_1 \sim_{\mathcal{R}} \pi_1 u_2$. Since $\pi_0 u_1 = \pi_0 t_1, \pi_0 u_2 = \pi_0 t_2, \pi_1 u_1 = \pi_1 t_1, \pi_1 u_2 = \pi_1 t_2$, by applying the inductive hypothesis on the types ρ and τ one has that $\pi_0 t_1 \sim_{\mathcal{R}} \pi_0 t_2$ and $\pi_1 t_1 \sim_{\mathcal{R}} \pi_1 t_2$. Thus, by Definition 10, $t_1 \sim_{\mathcal{R}} t_2$. ◀

The following proposition says that any closed term of $\mathcal{T}$ in which quote does not occur, is extensional modulo $\mathcal{R}$, where $\mathcal{R}$ is any reflexive binary relation between terms. The proof of the extensionality of the constant quote requires instead the definition of a particular relation $\mathcal{R}$ and will be formalized in Lemma 14. Since $\sim_{\mathcal{R}}$ can be seen as a *logical relation*, in the sense of Plotkin, our proposition can be seen as yet another incarnation of the usual Fundamental Theorem of logical relations (see e.g. [16]).

▶ **Proposition 12** (Extensionality). *Let t be a term of $\mathcal{T}$ with free variables among $x_1, \dots, x_k$ and assume that the constant* quote *does not occur in t. If $u_1, \dots, u_k, v_1, \dots, v_k$ are closed terms of $\mathcal{T}$ such that $u_1 \sim_{\mathcal{R}} v_1, \dots, u_k \sim_{\mathcal{R}} v_k$, then $t[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} t[v_1/x_1 \dots v_k/x_k]$.*

Proof. By induction on the structure of t.

1. t is a variable x_i for some $i \in [1,k]$. Trivially, $x_i[u_1/x_1 \dots u_k/x_k] = u_i \sim_{\mathcal{R}} v_i = x_i[v_1/x_1 \dots v_k/x_k]$.
2. t is an application $t_1 t_2$. Suppose $t_1 : \tau \to \sigma$ and $t_2 : \tau$. By inductive hypothesis, one has $t_1[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} t_1[v_1/x_1 \dots v_k/x_k]$ and $t_2[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} t_2[v_1/x_1 \dots v_k/x_k]$. By Definition 10 of the extensionality relation
$$t_1[u_1/x_1 \dots u_k/x_k]t_2[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} t_1[v_1/x_1 \dots v_k/x_k]t_2[v_1/x_1 \dots v_k/x_k]$$
which is to say
$$t_1 t_2[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} t_1 t_2[v_1/x_1 \dots v_k/x_k]\,.$$
3. t is $\lambda z^{\sigma} w$. Let us consider any two terms r_1, r_2 of type σ such that $r_1 \sim_{\mathcal{R}} r_2$. By inductive hypothesis, it holds that
$$w[u_1/x_1 \dots u_k/x_k\ r_1/z] \sim_{\mathcal{R}} w[v_1/x_1 \dots v_k/x_k\ r_2/z]\,.$$
Since
$$(\lambda z^{\sigma} w)[u_1/x_1 \dots u_k/x_k]r_1 = w[u_1/x_1 \dots u_k/x_k\ r_1/z]$$
$$(\lambda z^{\sigma} w)[v_1/x_1 \dots v_k/x_k]r_2 = w[v_1/x_1 \dots v_k/x_k\ r_2/z]$$
by Lemma 11 we obtain
$$(\lambda z^{\sigma} w)[u_1/x_1 \dots u_k/x_k]r_1 \sim_{\mathcal{R}} (\lambda z^{\sigma} w)[v_1/x_1 \dots v_k/x_k]r_2$$
and thus the thesis.
4. t is a pair $\langle t_1, t_2 \rangle$. Then, for $i = 0, 1$, by induction hypothesis
$$\pi_i(t[u_1/x_1 \dots u_k/x_k]) = t_i[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} t_i[v_1/x_1 \dots v_k/x_k] = \pi_i(t[v_1/x_1 \dots v_k/x_k])$$
and thus by Lemma 11 we obtain
$$\pi_i(t[u_1/x_1 \dots u_k/x_k]) \sim_{\mathcal{R}} \pi_i(t[v_1/x_1 \dots v_k/x_k])$$
and thus the thesis.
5. t is $\pi_i w$, $i = 0, 1$. By inductive hypothesis,
$$w[u_1/x_1 \dots u_k/x_k] \sim_{\mathcal{R}} w[u_1/x_1 \dots u_k/x_k]$$
and by Definition 10 of extensionality we have the thesis.
6. t is a constant such as 0 : `N`, `True` : `Bool`, `False` : `Bool`: we conclude $t \sim_{\mathcal{R}} t$ by Definition 10.

7. t is the constant $\mathsf{S} : \mathtt{N} \to \mathtt{N}$. Given two terms $w_1, w_2 : \mathtt{N}$ such that $w_1 \sim_{\mathcal{R}} w_2$, by definition of extensionality relation, $w_1 = w_2$. Then clearly $\mathsf{S}w_1 \sim_{\mathcal{R}} \mathsf{S}w_2$ and we obtain the thesis by Definition 10.
8. t is $\bot_i$, $\top_i$ ($i \in \mathbb{N}$): $\bot_i \sim_{\mathcal{R}} \bot_i$ and $\top_i \sim_{\mathcal{R}} \top_i$ follows by reflexivity of the relation $\mathcal{R}$.
9. t is the constant $\mathsf{tt} : \mathtt{N} \to \mathsf{U}$. Let us consider two terms w_1 and w_2 of type $\mathtt{N}$ such that $w_1 \sim_{\mathcal{R}} w_2$. By Definition 10, $w_1 = w_2$, i.e they have the same numeral, say m, as normal form. Therefore, $\mathsf{tt}\, w_1 = \top_m \sim_{\mathcal{R}} \top_m = \mathsf{tt}\, w_2$ and, by Lemma 11 and definition of the extensionality relation, $\mathsf{tt} \sim_{\mathcal{R}} \mathsf{tt}$.
10. t is the constant ff: as for the previous case.
11. t is the constant if_τ. Let us consider $r_1 : \mathtt{Bool}, r_2 : \tau, r_3 : \tau$ and $s_1 : \mathtt{Bool}, s_2 : \tau, s_3 : \tau$ terms of $\mathcal{T}$ such that $r_1 \sim_{\mathcal{R}} s_1$, $r_2 \sim_{\mathcal{R}} s_2$ and $r_3 \sim_{\mathcal{R}} s_3$. We want to prove that $\mathsf{if}_\tau\, r_1\, r_2\, r_3 \sim_{\mathcal{R}} \mathsf{if}_\tau\, s_1\, s_2\, s_3$. By Definition 10, $r_1 \sim_{\mathcal{R}} s_1$ implies that $r_1 = s_1$, i.e. r_1 and s_1 both reduces to either $\mathtt{True}$ or $\mathtt{False}$.
There are two cases, according to the normal form of r_1 and s_1. If $r_1 = s_1 = \mathtt{True}$, then $\mathsf{if}_\tau\, r_1\, r_2\, r_3 = r_2 \sim_{\mathcal{R}} s_2 = \mathsf{if}_\tau\, s_1\, s_2\, s_3$ and the thesis follows by Lemma 11. If $r_1 = s_1 = \mathtt{False}$: symmetric to the previous case.
12. t is the constant R_τ. Let us consider $r_1, s_1 : \tau, r_2, s_2 : \mathtt{N} \to (\tau \to \tau), r_3, s_3 : \mathtt{N}$ terms of $\mathcal{T}$ such that $r_1 \sim_{\mathcal{R}} s_1$, $r_2 \sim_{\mathcal{R}} s_2$ and $r_3 \sim_{\mathcal{R}} s_3$. We want to prove that $\mathsf{R}_\tau\, r_1\, r_2\, r_3 \sim_{\mathcal{R}} \mathsf{R}_\tau\, s_1\, s_2\, s_3$. By Definition 10, $r_3 \sim_{\mathcal{R}} s_3$ implies that $r_3 = s_3$ and therefore r_3 and s_3 reduce to the same numeral: we argue by induction on it. If $r_3 = s_3 = 0$, then $\mathsf{R}_\tau\, r_1\, r_2\, 0 = r_1 \sim_{\mathcal{R}} s_1 = \mathsf{R}_\tau\, s_1\, s_2\, s_3$ and one can conclude by Lemma 11. If $r_3 = s_3 = \mathsf{S}(m)$, then

$$\mathsf{R}_\tau\, r_1\, r_2\, r_3 = \mathsf{R}_\tau\, r_1\, r_2\, \mathsf{S}(m) = r_2\, m\, (\mathsf{R}_\tau\, r_1\, r_2\, m)$$

$$\mathsf{R}_\tau\, s_1\, s_2\, s_3 = \mathsf{R}_\tau\, s_1\, s_2\, \mathsf{S}(m) = s_2\, m\, (\mathsf{R}_\tau\, s_1\, s_2\, m)$$

By induction hypothesis $\mathsf{R}_\tau r_1\, r_2\, m \sim_{\mathcal{R}} \mathsf{R}_\tau s_1\, s_2\, m$ and Definition 10, $r_2\, m\, (\mathsf{R}_\tau\, r_1\, r_2\, m) \sim_{\mathcal{R}} s_2\, m\, (\mathsf{R}_\tau\, s_1\, s_2\, m)$ and the thesis follows by Lemma 11.

◀

▶ **Corollary 13.** *Let t be any closed term of $\mathcal{T}$. If* $\mathsf{quote} \sim_{\mathcal{R}} \mathsf{quote}$, *$t \sim_{\mathcal{R}} t$.*

Proof. Clearly, for some fresh variable $z : \mathsf{U}$, $t = (t[z/\,\mathsf{quote}\,])[\,\mathsf{quote}\,/z]$. Thus, by Proposition 12 applied to $t[z/\,\mathsf{quote}\,]$, we obtain $t \sim_{\mathcal{R}} t$. ◀

In Section 6 we will prove that every theorem in $\mathsf{HA}^\omega + \mathsf{MP}$ is realizable and in particular that a *proof-like* realizer $\mathfrak{r}$ of Markov's Principle $\neg\forall x^{\mathtt{N}} P \to \exists x^{\mathtt{N}} P^\bot$ can be defined. In this case, the extensionality relation plays a crucial role. Our realizer $\mathfrak{r}$ of Markov's Principle will have to map a realizer of $\neg\forall x^{\mathtt{N}} P$ into a realizer of $\exists x^{\mathtt{N}} P^\bot$. In other words, given a realizer of $\neg\forall x^{\mathtt{N}} P$, $\mathfrak{r}$ must in some way extract from it either a counterexample for $\forall x^{\mathtt{N}} P$ to be used as a witness of $\exists x^{\mathtt{N}} P^\bot$, or a constant in $\mathbb{\bot}$, by which one can realize everything.

So let us examine a realizer of $\forall x^{\mathtt{N}} P \to \bot$. It takes as input a realizer of $\forall x^{\mathtt{N}} P$ and returns a realizer of $\bot$. A tentative first plan to define $\mathfrak{r}$ may thus be to construct a realizer of $\forall x^{\mathtt{N}} P$ in order to obtain a realizer of $\bot$, that is, a constant in $\mathbb{\bot}$. A realizer of $\forall x^{\mathtt{N}} P$ is indeed easily definable in $\mathcal{T}$ as follows:

$$\mathbf{test}_{\lambda x.P} ::= \lambda x^{\mathtt{N}}.\ \mathsf{if}\ P\ \mathsf{then}\ \mathsf{tt}x\ \mathsf{else}\ \mathsf{ff}x$$

It behaves the expected way: when fed with a numeral m it evaluates $P[m/x]$ yielding $\top_m$ if $P[m/x] = \mathtt{True}$ and $\bot_m$ if $P[m/x] = \mathtt{False}$.

Thus we are done... aren't we? Unfortunately, no. Clearly, $\mathbf{test}_{\lambda x.P}$ is not proof-like, since it contains the subterm ff and so it may evaluate to $\bot_i$ for some numeral i. As previously

said, only proof-like terms will be considered realizers/winning strategies and $\mathfrak{r}$ is forbidden to contain a term such as $\mathbf{test}_{\lambda x.P}$.

We have thus to formulate a new plan for constructing $\mathfrak{r}$. The idea is to use extensionality. We want to alter $\mathbf{test}_{\lambda x.P}$ in such a way that it behaves extensionally as before but at the same time it is proof-like! With that in mind, we modify the term $\mathbf{test}_{\lambda x.P}$ like this:

$$\mathbf{mtest}_{\lambda x.P} ::= \lambda x^{\mathtt{N}}.\mathsf{if}\ P\ \mathsf{then}\ \mathsf{tt}x\ \mathsf{else}\ \mathsf{tt}x$$

While that may appear like a crazy attempt, it works. The term $\mathbf{mtest}_{\lambda x.P}$ is indeed proof-like, and differs from $\mathbf{test}_{\lambda x.P}$ only for the fact that it returns a constant in $\mathbb{T}$ also when $P[m/x]$ is false. That would be a great difference in another situation, but here it is not the case: $\mathbf{test}_{\lambda x.P}$ and $\mathbf{mtest}_{\lambda x.P}$ are equal up to a subterm of the form $\mathsf{tt}x$ or $\mathsf{ff}x$, which yields mute constants – constants that cannot be discriminated by any term in $\mathcal{T}$. In other words, $\mathbf{mtest}_{\lambda x.P}$ behaves observationally, i.e. extensionally, like $\mathbf{test}_{\lambda x.P}$, provided the relation $\mathcal{R}$ is suitable chosen.

In order to prove that $\mathbf{mtest}_{\lambda x.P} \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.P}$, $\mathcal{R}$ will be defined to hold either on pairs of equal terms (and this captures the case in which the evaluation of P on the given input n yields $\mathtt{True}$ and both $\mathbf{mtest}_{\lambda x.P}$ and $\mathbf{test}_{\lambda x.P}$ evaluates to $\mathsf{tt}n$) or on pair of discordant constants $(\top_k, \bot_k)$, where the index k is a numeral such that $P[k/x] = \mathtt{False}$. These constants are considered to be "equal" by the terms of our system and their index k is a counterexample to the formula $\forall x^{\mathtt{N}} P$ and therefore a correct witness for $\exists x^{\mathtt{N}} P^{\bot}$. Notice that k can be extracted both from $\top_k$ and $\bot_k$ by the constant quote, which is not able to produce any information about the argument but the index itself. The same constant quote is extensional modulo the relation $\mathcal{R}$ just introduced. All these notions are formalized in the following lemma:

▶ **Lemma 14** (Test Equivalence). *Let us consider the terms* $\mathbf{mtest}_{\lambda x.P}$ *and* $\mathbf{test}_{\lambda x.P}$ *defined above and the saturated-with-respect-to-equality relation*

$$\mathcal{R} ::= \{(t_1, t_2) \mid t_1 = t_2 \text{ or } (t_1 = \top_k, t_2 = \bot_k \text{ and } P[k/x] = \mathtt{False} \text{ for some numeral } k)\}$$

where we assume that the only free variable of P *is* x*. Then:*

1. $\mathbf{mtest}_{\lambda x.P} \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.P}$
2. $\mathsf{quote} \sim_{\mathcal{R}} \mathsf{quote}$

Proof.

1. Let us consider two closed term $s : \mathtt{N}$ and $r : \mathtt{N}$ such that $s \sim_{\mathcal{R}} r$. By Theorem 2 and by Definition 10, s and r reduce to the same numeral, say $n : \mathtt{N}$. We want to prove that $\mathbf{mtest}_{\lambda x.P}\, n \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.P}\, n$.
   Two cases occur:
   - $P[n/x] = \mathtt{True}$. Then $\mathbf{mtest}_{\lambda x.P}\, n = \mathsf{tt}n = \top_n = \mathsf{tt}n = \mathbf{test}_{\lambda x.P}\, n$. By definition of $\mathcal{R}$, $\mathbf{mtest}_{\lambda x.P}\, n\ \mathcal{R}\ \mathbf{test}_{\lambda x.P}\, n$, which is to say $\mathbf{mtest}_{\lambda x.P}\, n \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.P}\, n$.
   - $P[n/x] = \mathtt{False}$. Then

     $$\mathbf{mtest}_{\lambda x.P}\, n = \mathsf{tt}n = \top_n \sim_{\mathcal{R}} \bot_n = \mathsf{ff}n = \mathbf{test}_{\lambda x.P}\, n$$

     Therefore, by Lemma 11 $\mathbf{mtest}_{\lambda x.P}\, n \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.P}\, n$.

   Finally, by Definition 10 and Lemma 11, one can conclude $\mathbf{mtest}_{\lambda x.P} \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.P}$.

2. Let us consider two terms u_1 and u_2 of type U such that $u_1 \sim_{\mathcal{R}} u_2$. By Theorem 2 and by Definition 10:
   - either $u_1 = u_2$, and clearly $\mathsf{quote}\, u_1 = \mathsf{quote}\, u_2$ and, by Definition 10, $\mathsf{quote}\, u_1 \sim_{\mathcal{R}} \mathsf{quote}\, u_2$;
   - or $u_1 = \top_k$, $u_2 = \bot_k$ for some k and $P[k/x] = \mathtt{False}$. Also in this case $\mathsf{quote}\, u_1 = \mathsf{quote}\, u_2 = k$ and, by Definition 10, $\mathsf{quote}\, u_1 \sim_{\mathcal{R}} \mathsf{quote}\, u_2$.

   Finally, by Definition 10 and Lemma 11, one can conclude $\mathsf{quote} \sim_{\mathcal{R}} \mathsf{quote}$. ◀

5 A Realizer of Markov's Principle

We are now ready to define the realizer $\mathfrak{r}$ of Markov's Principle. $\mathfrak{r}$ takes as argument a realizer z of $\neg\forall x^{\mathtt{N}} P$ and we want $\mathfrak{r}$ to pass the term $\mathbf{mtest}_{\lambda x.P}$ as argument to z. Of course, $\mathbf{mtest}_{\lambda x.P}$ is not a realizer of $\forall x^{\mathtt{N}} P$, which is required in order to obtain with certitude a realizer of $\bot$ from z. However, it is extensionally equal to the realizer $\mathbf{test}_{\lambda x.P}$, which is enough. Now, let us consider $z\,\mathbf{mtest}_{\lambda x.P}$. The informal reasoning is the following (for a detailed argument see the proof of the Adequacy Theorem 15 or come back after having read it for intuitive explanations of the formal details). $z\,\mathbf{mtest}_{\lambda x.P}$ is extensionally equal to $z\,\mathbf{test}_{\lambda x.P}$, for $\mathcal{R}$ chosen as in Lemma 14, and $z\,\mathbf{test}_{\lambda x.P}$ must normalize to a constant in $\perp\!\!\!\perp$, say $\bot_k$. That constant is ultimately generated either by $\mathbf{test}_{\lambda x.P}$ or already by z. In this latter case, also $z\,\mathbf{mtest}_{\lambda x.P}$ will be able to produce $\bot_k$, and we are done, we can realize everything. In the former case, $z\,\mathbf{mtest}_{\lambda x.P}$ should reduce to $\top_k$, with k witness for $\exists x^{\mathtt{N}} P^{\bot}$, because $z\,\mathbf{mtest}_{\lambda x.P}\ \mathcal{R}\ z\,\mathbf{test}_{\lambda x.P}$; then k can be extracted by quote applied to $z\,\mathbf{mtest}_{\lambda x.P}$.

For those reasons, we are lead to define $\mathfrak{r}$ as:

$$\lambda z^{(\mathtt{N}\to\mathsf{U})\to\mathsf{U}} \langle\, \mathsf{quote}\,(z\,\mathbf{mtest}_{\lambda x.P}), \mathsf{if}\ P^{\bot}[\,\mathsf{quote}\,(z\,\mathbf{mtest}_{\lambda x.P})/x]\ \mathsf{then}\ \mathtt{tt}0\ \mathsf{else}\ z(\mathbf{mtest}_{\lambda x.P}) \rangle$$

$\mathfrak{r}$ just tests whether the numeral $k = \mathsf{quote}\,(z\,\mathbf{mtest}_{\lambda x.P})$ is a witness for $\exists x^{\mathtt{N}} P^{\bot}$; if it is the case, then $\mathtt{tt}0 = \top_0$ realizes $P^{\bot}[k/x]$, otherwise $z\,\mathbf{mtest}_{\lambda x.P}$ realizes $\bot$ and thus $P^{\bot}[k/x]$.

5.1 Curry-Howard Correspondence for $\mathsf{HA}^{\omega} + \mathsf{MP}$

In Figure 2, we define a standard natural deduction system for $\mathsf{HA}^{\omega} + \mathsf{MP}$ (see [19], for example) together with a term assignment in the spirit of Curry-Howard correspondence for intuitionistic logic.

We replace purely universal axioms (i.e., Π^0_1-axioms) with *sound Post rules*, which are inferences of the form

$$\frac{\Gamma \vdash A_1 \ \Gamma \vdash A_2 \ \cdots \ \Gamma \vdash A_n}{\Gamma \vdash A}$$

where $A_1, \ldots, A_n, A$ are atomic formulas of $\mathcal{T}$ such that for every substitution

$$\sigma = [t_1/x_1, \ldots, t_k/x_k]$$

of closed terms $t_1, \ldots, t_k$ of $\mathcal{T}$, $A_1\sigma = \ldots = A_n\sigma = \mathtt{True}$ implies $A\sigma = \mathtt{True}$. Any other axiomatic presentation of HA^{ω} would have worked just fine, but Post rules allows to define in a uniform way a more flexible deduction system, which is very useful when coding actual

Contexts With Γ we denote contexts of the form $x_1 : A_1, \ldots, x_n : A_n$, with $x_1, \ldots, x_n$ proof variables and $A_1, \ldots, A_n$ formulas of $\mathcal{T}$.

Axioms $\Gamma, x : A \vdash x^{|A|} : A$

Conjunction $\dfrac{\Gamma \vdash u : A \quad \Gamma \vdash t : B}{\Gamma \vdash \langle u, t\rangle : A \wedge B}$ $\qquad \dfrac{\Gamma \vdash u : A \wedge B}{\Gamma \vdash \pi_0 u : A}$ $\qquad \dfrac{\Gamma \vdash u : A \wedge B}{\Gamma \vdash \pi_1 u : B}$

Implication $\dfrac{\Gamma \vdash u : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash ut : B}$ $\qquad \dfrac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x^{|A|} u : A \to B}$

Disjunction Intro. $\dfrac{\Gamma \vdash u : A}{\Gamma \vdash \langle \mathtt{True}, u, d^{|B|}\rangle : A \vee B}$ $\qquad \dfrac{\Gamma \vdash u : A}{\Gamma \vdash \langle \mathtt{False}, d^{|A|}, u\rangle : A \vee B}$

Disjunction Elim. $\dfrac{\Gamma \vdash u : A \vee B \quad \Gamma \vdash w_1 : A \to C \quad \Gamma \vdash w_2 : B \to C}{\Gamma \vdash \mathsf{if}\ \mathsf{p}_0 u\ \mathsf{then}\ w_1(\mathsf{p}_1 u)\ \mathsf{else}\ w_2(\mathsf{p}_2 u) : C}$

Universal Quantification $\dfrac{\Gamma \vdash u : \forall \alpha^\tau A}{\Gamma \vdash ut : A[t/\alpha^\tau]}$ $\qquad \dfrac{\Gamma \vdash u : A}{\Gamma \vdash \lambda \alpha^\tau u : \forall \alpha^\tau A}$

where t is a term of $\mathcal{T}$ and $\alpha^{\mathtt{N}}$ does not occur free in any formula B occurring in Γ.

Existential Quantification $\dfrac{\Gamma \vdash u : A[t/\alpha^\tau]}{\Gamma \vdash \langle t, u\rangle : \exists \alpha^\tau . A}$ $\qquad \dfrac{\Gamma \vdash u : \exists \alpha^\tau . A \quad \Gamma \vdash t : \forall \alpha^\tau . A \to C}{\Gamma \vdash t(\pi_0 u)(\pi_1 u) : C}$

where α^τ is not free in C.

Induction $\dfrac{\Gamma \vdash u : A(0) \quad \Gamma \vdash v : \forall \alpha^{\mathtt{N}} . A(\alpha) \to A(\mathsf{S}(\alpha))}{\Gamma \vdash \lambda \alpha^{\mathtt{N}} \mathsf{R} u v \alpha : \forall \alpha^{\mathtt{N}} A}$

Booleans $\dfrac{\Gamma \vdash u : A(\mathtt{True}) \quad \Gamma \vdash v : A(\mathtt{False})}{\Gamma \vdash \lambda \alpha^{\mathtt{Bool}}\ \mathsf{if}\ x\ \mathsf{then}\ u\ \mathsf{else}\ v : \forall \alpha^{\mathtt{Bool}} A}$

Post Rules $\dfrac{\Gamma \vdash u_1 : A_1\ \Gamma \vdash u_2 : A_2\ \cdots\ \Gamma \vdash u_n : A_n}{\Gamma \vdash \mathsf{if}\ A\ \mathsf{then}\ \mathsf{tt}0\ \mathsf{else}\ \mathsf{if}\ A_1^\perp\ \mathsf{then}\ u_1\ \mathsf{else}\ \ldots \mathsf{if}\ A_n^\perp\ \mathsf{then}\ u_n\ \mathsf{else}\ \mathsf{tt}0 : A}$

where $n > 0$ and $A_1, A_2, \ldots, A_n, A$ are atomic formulas and the rule is a sound Post rule.

Post Rules with no Premises $\dfrac{}{\Gamma \vdash \mathsf{tt}0 : A}$

where A is an atomic formula of $\mathcal{T}$ and an axiom of equality or a classical propositional tautology.

MP $\dfrac{}{\Gamma \vdash \mathfrak{r} : \neg \forall x^{\mathtt{N}} P \to \exists x^{\mathtt{N}} P^\perp}$

where $\mathfrak{r} = \lambda z^{(\mathtt{N}\to\mathsf{U})\to\mathsf{U}} \langle\, \mathsf{quote}\,(z\, \mathbf{mtest}_{\lambda x.P}), \mathsf{if}\ P^\perp[\,\mathsf{quote}\,(z\, \mathbf{mtest}_{\lambda x.P})/x]\ \mathsf{then}\ \mathsf{tt}0\ \mathsf{else}\ z\, \mathbf{mtest}_{\lambda x.P} \rangle$

Figure 2 Terms Assignment Rules for $\mathsf{HA}^\omega + \mathsf{MP}$.

mathematical proofs. Let now $\mathsf{eq} : \mathtt{N}^2 \to \mathtt{Bool}$ a term of Gödel's system T representing equality between natural numbers. Among the Post rules, we have the Peano axioms

$$\frac{\Gamma \vdash \mathsf{eq}\ \mathsf{S}(x)\, \mathsf{S}(y)}{\Gamma \vdash \mathsf{eq}\ x\, y} \qquad \frac{\Gamma \vdash \mathsf{eq}\ 0\, \mathsf{S}(x)}{\Gamma \vdash \bot}$$

and axioms of equality

$$\frac{}{\Gamma \vdash \mathsf{eq}\ x\, x} \qquad \frac{\Gamma \vdash \mathsf{eq}\, x\, y \quad \Gamma \vdash \mathsf{eq}\, y\, z}{\Gamma \vdash \mathsf{eq}\ x\, z} \qquad \frac{\Gamma \vdash A(x) \quad \Gamma \vdash \mathsf{eq}\, x\, y}{\Gamma \vdash A(y)}$$

and for every A_1, A_2 such that $A_1 = A_2$ is an equation of system $\mathcal{T}$ (equivalently, A_1, A_2 have the same normal form in T), we have the rule

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_2}.$$

We also have a Post rule

$$\frac{\Gamma \vdash A_1 \ \Gamma \vdash A_2 \ \cdots \ \Gamma \vdash A_n}{\Gamma \vdash A}$$

for every classical propositional tautology $A_1 \to \ldots \to A_n \to A$, where for $i = 1, \ldots, n$, A_i, A are atomic formulas obtained as combination of other atomic formulas by the Gödel's system T closed terms representing boolean connectives. For example, given terms $\Rightarrow_{\mathtt{Bool}}, \wedge_{\mathtt{Bool}}, \vee_{\mathtt{Bool}} : \mathtt{Bool} \to \mathtt{Bool} \to \mathtt{Bool} \ldots$ representing boolean connectives, one can form, out of atomic formulas A and B, the atomic formulas $\Rightarrow_{\mathtt{Bool}} AB$ and $\wedge_{\mathtt{Bool}} AB$. Using infix notations, we have for example the rules

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}, \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \Rightarrow_{\mathtt{Bool}} B}, \qquad \frac{\Gamma \vdash A \wedge_{\mathtt{Bool}} B}{\Gamma \vdash A}.$$

Finally, we have a rule of case reasoning for booleans. For any formula $A(\alpha^{\mathtt{Bool}})$ he have the axiom:

$$\frac{\Gamma \vdash A(\mathtt{True}) \ \ \Gamma \vdash A(\mathtt{False})}{\Gamma \vdash \forall \alpha^{\mathtt{Bool}} A}.$$

We remark that some of the Post rules, for example many of those for eq, are derivable from others. We remark that the negations $^{\perp}$ and $\neg$, and the disjunctions $\vee_{\mathtt{Bool}}$ and $\vee$ have the same meaning but they are syntactically different: for every atomic formula P, we consider $P^{\perp}$ and $P \vee_{\mathtt{Bool}} P^{\perp}$ as atomic formulas, while $\neg P$ and $P \vee P^{\perp}$ as compound formulas. But one can show that, for every atomic formula P, $\mathsf{HA}^{\omega} \vdash P^{\perp} \leftrightarrow \neg P$: it is enough to derive $\mathsf{HA}^{\omega} \vdash \mathtt{True}^{\perp} \leftrightarrow \neg\mathtt{True}$ and $\mathsf{HA}^{\omega} \vdash \mathtt{False}^{\perp} \leftrightarrow \neg\mathtt{False}$, then use the rule of case reasoning for booleans to obtain $\mathsf{HA}^{\omega} \vdash \forall \alpha^{\mathtt{Bool}} \alpha^{\perp} \leftrightarrow \neg\alpha$ and conclude with the elimination of $\forall$ applied to P. We can derive $\mathsf{HA}^{\omega} \vdash \mathtt{True}^{\perp} \to \neg\mathtt{True}$ as follows:

$$\frac{\dfrac{\mathtt{True}^{\perp}, \mathtt{True} \vdash \mathtt{True}^{\perp} = \mathsf{if}\ \mathtt{True}\ \mathsf{then}\ \mathtt{False}\ \mathsf{else}\ \mathtt{True}}{\mathtt{True}^{\perp}, \mathtt{True} \vdash \mathtt{False}}}{\mathtt{True}^{\perp} \vdash \mathtt{True} \to \mathtt{False} = \neg\mathtt{True}}$$

and $\mathsf{HA}^{\omega} \vdash \neg\mathtt{True} \to \mathtt{True}^{\perp}$ as follows:

$$\frac{\dfrac{\neg\mathtt{True} \vdash \neg\mathtt{True} \qquad \neg\mathtt{True} \vdash \mathtt{True}}{\neg\mathtt{True} \vdash \mathtt{False}}}{\neg\mathtt{True} \vdash \mathtt{True}^{\perp} = \mathsf{if}\ \mathtt{True}\ \mathsf{then}\ \mathtt{False}\ \mathsf{else}\ \mathtt{True}}$$

$\mathsf{HA}^{\omega} \vdash \mathtt{False}^{\perp} \leftrightarrow \neg\mathtt{False}$ can be derived even more easily, since $\neg\mathtt{False} = \mathtt{False} \to \mathtt{False}$ is derivable and

$$\frac{\vdash \mathtt{True}}{\vdash \mathtt{False}^{\perp} = \mathsf{if}\ \mathtt{False}\ \mathsf{then}\ \mathtt{False}\ \mathsf{else}\ \mathtt{True}}$$

Moreover, $P \vee_{\mathtt{Bool}} P^{\perp}$ is an axiom, while we may derive $\mathsf{HA}^{\omega} \vdash P \vee P^{\perp}$ again by case reasoning for booleans.

If τ is any type of $\mathcal{T}$, we denote with d^{τ} a dummy term of type τ, defined by $d^{\mathbb{N}} = 0$, $d^{\mathtt{Bool}} = \mathtt{False}$, $d^{\mathbb{U}} = \top_0$, $d^{\sigma \to \rho} = \lambda z^{\sigma}.d^{\rho}$ (with z^{σ} any variable of type σ), $d^{\sigma \times \rho} = \langle d^{\sigma}, d^{\rho} \rangle$.

6 Main Results

6.1 The Adequacy Theorem

We now prove our main result, namely, that every theorem of $\mathsf{HA}^\omega + \mathsf{MP}$ is realizable by a proof-like term. This derives as an easy corollary from the Adequacy Theorem 15. In the Adequacy Theorem we will exploit the extensionality relation defined in Section 4.

As usual in adequacy proofs for realizability, we prove a stronger version of the theorem, suitable to be proved by induction.

▶ **Theorem 15** (Adequacy). *Assume that* $\Gamma \vdash w : A$ *in* $\mathsf{HA}^\omega + \mathsf{MP}$, *with* $\Gamma = x_1 : A_1, \dots, x_n : A_n$ *and suppose that all the free variables occurring in* Γ *and* $w : A$ *are among* $\alpha_1 : \tau_1, \dots, \alpha_k : \tau_k$. *For any choice of closed terms* $r_1 : \tau_1, \dots, r_k : \tau_k$ *of system* $\mathcal{T}$, *if there are terms* $t_1, \dots, t_n$ *such that, for* $i = 1, \dots, n$

$$t_i \Vdash A_i[r_1/\alpha_1, \dots, r_k/\alpha_k]$$

then

$$w[t_1/x_1^{|A_1|}, \dots, t_n/x_n^{|A_n|}, r_1/\alpha_1, \dots, r_k/\alpha_k] \Vdash A[r_1/\alpha_1, \dots, r_k/\alpha_k]\,.$$

Proof.

▶ Notation 1. For any term v and formula B, we denote $v[t_1/x_1^{|A_1|} \cdots t_n/x_n^{|A_n|}\ r_1/\alpha_1 \cdots r_k/\alpha_k]$ with $\overline{v}$ and $B[r_1/\alpha_1 \cdots r_k/\alpha_k]$ with $\overline{B}$. We have $|\overline{B}| = |B|$ for all formulas B.

We proceed by induction on the derivation of $\Gamma \vdash w : A$. Let r be the last rule applied in the derivation.

1. r is an axiom for variables. For some i, $w = x_i^{|A_i|}$ and $A = A_i$. So $\overline{w} = t_i \Vdash \overline{A_i} = \overline{A}$.
2. r is the $\wedge I$ rule, then $w = \langle u, t\rangle$, $A = B \wedge C$, $\Gamma \vdash u : B$ and $\Gamma \vdash t : C$. Therefore, $\overline{w} = \langle \overline{u}, \overline{t}\rangle$. By induction hypothesis, $\pi_0\overline{w} = \overline{u} \Vdash \overline{B}$ and $\pi_1\overline{w} = \overline{t} \Vdash \overline{C}$; so, by Lemma 9, $\overline{w} \Vdash \overline{B} \wedge \overline{C} = \overline{A}$.
3. r is a $\wedge E$ rule, say left, then $\Gamma \vdash u : A \wedge B$, $w = \pi_0 u$. Since $\overline{u} \Vdash \overline{A} \wedge \overline{B}$ by induction hypothesis, if $\overline{w} = \pi_0\overline{u}$ we can conclude $w \Vdash \overline{A}$.
4. r is the $\to E$ rule, then $\Gamma \vdash u : B \to A$ and $\Gamma \vdash t : B$ $w = ut$, . So $\overline{w} = \overline{u}\overline{t} \Vdash \overline{A}$, for $\overline{u} \Vdash \overline{B} \to \overline{A}$ and $\overline{t} \Vdash \overline{B}$ by induction hypothesis.
5. r is the $\to I$ rule, then $w = \lambda x^{|B|}u$, $A = B \to C$ and $\Gamma, x : B \vdash u : C$. Suppose now that $t \Vdash \overline{B}$; we have to prove that $\overline{w}t \Vdash \overline{C}$. By induction hypothesis on u, $\overline{u} \Vdash \overline{C}$. One has

$$\begin{aligned}\overline{w}t &= (\lambda x^{|B|}u)[t_1/x_1^{|A_1|} \cdots t_n/x_n^{|A_n|}\ r_1/\alpha_1 \cdots r_k/\alpha_k]t\\ &= (\lambda x^{|B|}u)t[t_1/x_1^{|A_1|} \cdots t_n/x_n^{|A_n|}\ r_1/\alpha_1 \cdots r_k/\alpha_k]\\ &= u[t/x^{|B|}][t_1/x_1^{|A_1|} \cdots t_n/x_n^{|A_n|}\ r_1/\alpha_1 \cdots r_k/\alpha_k]\\ &= \overline{u}\,.\end{aligned}$$

   Then since $\overline{u} = \overline{w}t$, by Lemma 9, $\overline{w}t \Vdash \overline{C}$.
6. r is a $\vee I$ rule, say left (the other case is symmetric), then $w = \langle \mathtt{True}, u, d^{|C|}\rangle$, $A = B \vee C$ and $\Gamma \vdash u : B$. So, $\overline{w} = \langle \mathtt{True}, \overline{u}, d^{|C|}\rangle$ and hence $\pi_0\overline{w}[s] = \mathtt{True}$. $\overline{u} \Vdash \overline{B}$ follows with the help of induction hypothesis.
7. r is a $\vee E$ rule, then

   $$w = \mathsf{if}\ \mathsf{p}_0 u\ \mathsf{then}\ w_1(\mathsf{p}_1 u)\ \mathsf{else}\ w_2(\mathsf{p}_2 u)$$

   and $\Gamma \vdash u : B \vee C$, $\Gamma \vdash w_1 : B \to A$, $\Gamma \vdash w_2 : C \to A$.
   Assume $\mathsf{p}_0\overline{u} = \pi_0\overline{u} = \mathtt{True}$. By inductive hypothesis $\overline{u} \Vdash \overline{B} \vee \overline{C}$, $\overline{w_1} \Vdash \overline{B} \to \overline{A}$ and $\overline{w_2} \Vdash \overline{C} \to \overline{A}$. Therefore, $\mathsf{p}_1\overline{u} \Vdash \overline{B}$. Hence $\overline{w} = \overline{w_1}(\mathsf{p}_1\overline{u})$.

Since $\overline{w_1} \Vdash \overline{B} \to \overline{A}$ and $\mathsf{p}_1\overline{u} \Vdash \overline{B}$, by definition of realizability, $\overline{w_1}(\mathsf{p}_1\overline{u}) \Vdash \overline{A}$. By $\overline{w} = \overline{w_1}((\mathsf{p}_1\overline{u}))$ and Lemma 9, also $\overline{w} \Vdash \overline{A}$.
Symmetrically, if $\mathsf{p}_0\overline{u} = \mathtt{False}$, we obtain again $\overline{w} \Vdash \overline{A}$.

8. r is the $\forall E$ rule, then $w = ut$, $A = B[t/\alpha^\tau]$ and $\Gamma \vdash u : \forall\alpha^\tau B$. So, $\overline{w} = \overline{u}\overline{t}$. By inductive hypothesis $\overline{u} \Vdash \forall\alpha^\tau\overline{B}$ and so we can conclude that $\overline{u}\overline{t} \Vdash \overline{B}[\overline{t}/\alpha^\tau]$.
9. r is the $\forall I$ rule, then $w = \lambda\alpha^\tau u$, $A = \forall\alpha^\tau B$ and $\Gamma \vdash u : B$ (with α^τ not occurring free in the formulas of Γ). So, $\overline{w} = \lambda\alpha^\tau\overline{u}$, since $\alpha \neq \alpha_1, \ldots, \alpha_k$. Let $t : \tau$ be a closed term of HA^ω; by Lemma 9, it is enough to prove that $\overline{w}t = \overline{u}[t/\alpha^\tau]$, $\overline{u}[t/\alpha^\tau] \Vdash \overline{B}[t/\alpha^\tau]$, which amounts to show that the induction hypothesis can be applied to u. We observe that, since $\alpha \neq \alpha_1, \ldots, \alpha_k$, for $i = 1, \ldots, n$ we have

$$t_i \Vdash \overline{A_i} = \overline{A_i}[t/\alpha^\tau]\,.$$

10. r is the $\exists E$ rule, then $w = t(\pi_0 u)(\pi_1 u)$, $\Gamma \vdash t : \forall\alpha^\tau : B \to C$ and $\Gamma \vdash u : \exists\alpha^\tau.B$.
By inductive hypothesis $\overline{u} \Vdash \exists\alpha^{\mathtt{N}}.\overline{B}$, $\pi_0\overline{u} = v$ for v term in HA^ω and hence $\pi_1\overline{u} \Vdash \overline{B}[v/\alpha^\tau]$. Then

$$\overline{t}v(\pi_1\overline{u}) \Vdash \overline{C}[v/\alpha^\tau] = \overline{C}\,.$$

We thus obtain by $\overline{w} = \overline{t}(\pi_0\overline{u})(\pi_1\overline{u})$ and by Lemma 9 that $\overline{w} \Vdash \overline{C}$.

11. r is the $\exists I$ rule, then $w = \langle t, u\rangle$, $A = \exists\alpha^\tau B$, $\Gamma \vdash u : B[t/\alpha^\tau]$. So, $\overline{w} = \langle\overline{t}, \overline{u}\rangle$; and, indeed, $\pi_1\overline{w} = \overline{u} \Vdash \overline{B}[\overline{t}/\alpha^\tau]$ by induction hypothesis. By Lemma 9 we conclude the thesis.
12. r is the induction rule. Therefore $w = \lambda\alpha^{\mathtt{N}}\mathsf{R}uv\alpha$, $A = \forall\alpha^{\mathtt{N}}B$, $\Gamma \vdash u : B(0)$ and $\Gamma \vdash v : \forall\alpha^{\mathtt{N}}.B(\alpha) \to B(\mathsf{S}(\alpha))$. So, $\overline{w} = \lambda\alpha^{\mathtt{N}}\mathsf{R}\overline{u}\overline{v}\alpha$.
We have to prove that $\overline{w}u \Vdash \overline{B}[n/\alpha]$ for all closed term u of type $\mathtt{N}$.
Let n be the normal form of u: by Lemma 2 n is a numeral. A plain induction shows that

$$\overline{w}n = \mathsf{R}\overline{u}\overline{v}n \Vdash \overline{B}[n/\alpha]$$

for $\overline{u} \Vdash \overline{B}(0)$ and $\overline{v}i \Vdash \overline{B}(i) \to \overline{B}(\mathsf{S}(i))$ for all numerals i by induction hypothesis. If we set $i = n$, the thesis follows by Lemma 9 and $\overline{w}u = \overline{w}n$.

13. r is the rule for booleans, then $w = \lambda\alpha^{\mathtt{Bool}}\,\mathsf{if}\ \alpha\ \mathsf{then}\ u\ \mathsf{else}\ v$, $\Gamma \vdash u : B(\mathtt{True})$, $\Gamma \vdash v : B(\mathtt{False})$ and $A = \forall\alpha^{\mathtt{Bool}}B$. By inductive hypothesis, $\overline{u} \Vdash \overline{B}(\mathtt{True})$ and $\overline{v} \Vdash \overline{B}(\mathtt{False})$. So, $\overline{w} = \lambda\alpha^{\mathtt{Bool}}\mathsf{if}\ \alpha\ \mathsf{then}\ \overline{u}\ \mathsf{else}\ \overline{v}$. Let $t : \mathtt{Bool}$ be a closed term of HA^ω; by Lemma 9, it is enough to prove that

$$\overline{w}t = (\mathsf{if}\ t\ \mathsf{then}\ \overline{u}\ \mathsf{else}\ \overline{v}) \Vdash \overline{B}[\overline{t}/\alpha^{\mathtt{Bool}}]\,.$$

By Lemma 2, there are two cases:
- the normal form of t is $\mathtt{True}$. Then $\overline{w}t = (\mathsf{if}\ \mathtt{True}\ \mathsf{then}\ \overline{u}\ \mathsf{else}\ \overline{v})$ reduces to $\overline{u}$: the thesis follows by Lemma 9 and the inductive hypothesis on u.
- the normal form of t is $\mathtt{False}$. Then $\overline{w}t$ reduces to $\overline{v}$: the thesis follows by Lemma 9 and the inductive hypothesis on v.

14. r is a Post rule, then $w = \mathsf{if}\ A\ \mathsf{then}\ \mathsf{tt}0\ \mathsf{else}\ \mathsf{if}\ A_1^\perp\ \mathsf{then}\ u_1\ \mathsf{else}\ \ldots\,\mathsf{if}\ A_n^\perp\ \mathsf{then}\ u_n\ \mathsf{else}\,\mathsf{tt}0$.
By inductive hypothesis, for $i = 1, \ldots, n$, $\overline{u_i} \Vdash \overline{A_i}$. There are two cases:
- if $\overline{A} = \mathtt{True}$, then $\overline{w} = \mathsf{tt}0 = \top_0 \in \mathbb{T}$ and thus $\overline{w} \Vdash \overline{A}$.
- if $\overline{A} = \mathtt{False}$, then there exists $j \in [1, n]$ such that $\overline{A}_j = \mathtt{False}$ and $\overline{u}_j \in \perp\!\!\!\perp$. Thus $\overline{w} = \overline{u}_j$ and the thesis follows by Lemma 9 and the inductive hypothesis.

15. r is the MP axiom, then for some atomic formula Q

$$\overline{w} = \lambda z^{(\mathtt{N}\to\mathsf{U})\to\mathsf{U}}\langle\,\mathsf{quote}\,(z\,\mathbf{mtest}_{\lambda x.Q}), \mathsf{if}\ Q^{\perp}[\,\mathsf{quote}\,(z\,\mathbf{mtest}_{\lambda x.Q})/x]\ \mathsf{then}\ \mathtt{tt}0\ \mathsf{else}\ z\,\mathbf{mtest}_{\lambda x.Q}\rangle$$

and $\overline{A} = \neg\forall x^{\mathtt{N}}Q \to \exists x^{\mathtt{N}}Q^{\perp}$. Let $u : (\mathtt{N}\to\mathsf{U})\to\mathsf{U}$ be a closed term of $\mathcal{T}$ such that $u \Vdash (\forall x^{\mathtt{N}}Q)\to\perp$. We have to prove that

$$\overline{w}u = \langle\,\mathsf{quote}\,(u\,\mathbf{mtest}_{\lambda x.Q}), \mathsf{if}\ Q^{\perp}[\,\mathsf{quote}\,(u\,\mathbf{mtest}_{\lambda x.Q})/x]\ \mathsf{then}\ \mathtt{tt}0\ \mathsf{else}\ u\,\mathbf{mtest}_{\lambda x.Q}\rangle \Vdash \exists x^{\mathtt{N}}Q^{\perp}$$

By Theorem 2, assume $\mathsf{quote}\,(u\,\mathbf{mtest}_{\lambda x.Q}) = m$, with m numeral. There are two cases:

- m is a witness for $\exists x^{\mathtt{N}}Q^{\perp}$, that is, $Q^{\perp}[m/x] = \mathtt{True}$. Then

  $$\pi_1(\overline{w}u) = \mathsf{if}\ Q^{\perp}[m/x]\ \mathsf{then}\ \mathtt{tt}0\ \mathsf{else}\ u\,\mathbf{mtest}_{\lambda x.Q} = \top_0 \in \mathbb{T}$$

  and by Lemma 9 we can conclude $\overline{w}u \Vdash \exists x^{\mathtt{N}}Q^{\perp}$.
- m is not a witness for $\exists x^{\mathtt{N}}Q^{\perp}$, that is, $Q^{\perp}[m/x] = \mathtt{False}$ and

  $$\pi_1(\overline{w}u) = \mathsf{if}\ Q^{\perp}[m/x]\ \mathsf{then}\ \mathtt{tt}0\ \mathsf{else}\ u\,\mathbf{mtest}_{\lambda x.Q} = u\,\mathbf{mtest}_{\lambda x.Q}$$

  In order to obtain the thesis, we have to prove that $u\,\mathbf{mtest}_{\lambda x.Q} \Vdash Q^{\perp}[m/x]$. We have that $\mathbf{test}_{\lambda x.Q} \Vdash \forall x^{\mathtt{N}}Q$ and so $u\,\mathbf{test}_{\lambda x.Q} \Vdash \perp$. Therefore $u\,\mathbf{test}_{\lambda x.Q} = \perp_n$, for some numeral n. Let us define the saturated relation $\mathcal{R}$ defined as in Lemma 14

  $$\mathcal{R} ::= \{(t_1,t_2) \mid t_1 = t_2 \text{ or } (t_1 = \top_i, t_2 = \perp_i \text{ and } Q[i/x] = \mathtt{False} \text{ for some } i)\}$$

  By the Test Equivalence Lemma 14, $\mathbf{mtest}_{\lambda x.Q} \sim_{\mathcal{R}} \mathbf{test}_{\lambda x.Q}$, $\mathsf{quote} \sim_{\mathcal{R}} \mathsf{quote}$; therefore, by Corollary 13, $u \sim_{\mathcal{R}} u$ and by Definition 10, $u\,\mathbf{mtest}_{\lambda x.Q} \sim_{\mathcal{R}} u\,\mathbf{test}_{\lambda x.Q}$, which implies $u\,\mathbf{mtest}_{\lambda x.Q}\ \mathcal{R}\ u\,\mathbf{test}_{\lambda x.Q}$. Now, $u\,\mathbf{test}_{\lambda x.Q} = \perp_n$ and it cannot be that $u\,\mathbf{mtest}_{\lambda x.Q} = \top_n$, because by assumption $\mathsf{quote}\,(u\,\mathbf{mtest}_{\lambda x.Q}) = m$ and we would thus have $m = n$, with again by assumption

  $$Q[m/x] = \mathtt{True}$$

  By definition of $\mathcal{R}$, this forces $u\,\mathbf{mtest}_{\lambda x.Q} = u\,\mathbf{test}_{\lambda x.Q}$. Therefore, $u\,\mathbf{mtest}_{\lambda x.Q} \in \perp\!\!\!\perp$. We conclude that $u\,\mathbf{mtest}_{\lambda x.Q} \Vdash Q^{\perp}[m/x]$.

◀

Since all the terms decorating the inference rules of $\mathsf{HA}^{\omega} + \mathsf{MP}$ are proof-like, as an easy corollary of Theorem 15 we obtain the main theorem:

▶ **Theorem 16.** *If A is a closed formula and $\mathsf{HA}^{\omega} + \mathsf{MP} \vdash t : A$, then $t \Vdash A$, with t proof-like term of $\mathcal{T}$.*

6.2 Realizability and Truth

We now want to investigate the relationship between realizability and truth. We have already seen in Proposition 7 that any formula is realizable. Here, we want to show that our notion of realizability is consistent at least when realizers come from proofs in $\mathsf{HA}^{\omega} + \mathsf{MP}$: whenever a formula not containing $\to$ is realized by a proof-like term, it is also true, for a suitable notion of truth. Intuitively, we consider a formula of HA^{ω} to hold if it is true in the canonical syntactical model in which quantifiers of type τ range over the closed terms of HA^{ω} of type τ. In particular, the truth of arithmetical formulas is exactly the standard arithmetical truth over $\mathbb{N}$. We now give the obvious definition.

▶ **Definition 17** (Truth in the Syntactical Model). Given a closed formula F of HA^ω, we define by induction over F its truth value $[\![F]\!] \in \{\mathtt{True}, \mathtt{False}\}$.

- If P is atomic, $[\![P]\!] = \mathtt{True}$ if $P = \mathtt{True}$, $[\![P]\!] = \mathtt{False}$ otherwise.
- $[\![A \wedge B]\!] = \mathtt{True}$ if $[\![A]\!] = [\![B]\!] = \mathtt{True}$, $[\![A \wedge B]\!] = \mathtt{False}$ otherwise.
- $[\![A \vee B]\!] = \mathtt{True}$ if $[\![A]\!] = \mathtt{True}$ or $[\![B]\!] = \mathtt{True}$, $[\![A \vee B]\!] = \mathtt{False}$ otherwise.
- $[\![A \to B]\!] = \mathtt{True}$ if $[\![A]\!] = \mathtt{True}$ implies $[\![B]\!] = \mathtt{True}$, $[\![A \to B]\!] = \mathtt{False}$ otherwise.
- $[\![\forall x^\tau A]\!] = \mathtt{True}$ if for all closed terms $t : \tau$ of HA^ω, $[\![A[t/x]]\!] = \mathtt{True}$, $[\![\forall x^\tau A]\!] = \mathtt{False}$ otherwise.
- $[\![\exists x^\tau A]\!] = \mathtt{True}$ if there exists a closed term $t : \tau$ of HA^ω such that $[\![A[t/x^\tau]]\!] = \mathtt{True}$, $[\![\exists x^\tau A]\!] = \mathtt{False}$ otherwise.

We are now ready show the consistency of our notion of realizability.

▶ **Proposition 18** (Consistency of Realizability). *Let F be a closed $\to$-free formula and let t be a proof-like term such that $t \Vdash F$. Then $[\![F]\!] = \mathtt{True}$.*

Proof. By induction on F.

1. $F = P$, with P atomic. Since t is proof-like, no term in the reduction tree of t can contain a constant in $\bot\!\!\!\bot$. Therefore, $t \notin \bot\!\!\!\bot$, and since $t \Vdash P$, it must be that $P = \mathtt{True}$.
2. $F = A \wedge B$. Since $t \Vdash A \wedge B$, we have that $\pi_0 t \Vdash A$ and $\pi_1 t \Vdash B$. By induction hypothesis, $[\![A]\!] = \mathtt{True}$ and $[\![B]\!] = \mathtt{True}$. Therefore, $[\![A \wedge B]\!] = \mathtt{True}$.
3. $F = A \vee B$. Since $t \Vdash A \vee B$, we have that $\mathsf{p}_1 t \Vdash A$ or $\mathsf{p}_2 t \Vdash B$. By induction hypothesis, $[\![A]\!] = \mathtt{True}$ or $[\![B]\!] = \mathtt{True}$. Therefore, $[\![A \vee B]\!] = \mathtt{True}$.
4. $F = \forall x^\tau A$. Since $t \Vdash \forall x^\tau A$, we have that for all closed terms u of HA^ω, $tu \Vdash A[u/x^\tau]$. By induction hypothesis, for all closed terms u of HA^ω, $[\![A[u/x^\tau]]\!] = \mathtt{True}$. Therefore, $[\![\forall x^\tau A]\!] = \mathtt{True}$.
5. $F = \exists x^\tau A$. Since $t \Vdash \exists x^\tau A$, we have that for $\pi_0 t = u$ for some closed term u of HA^ω, and $\pi_1 t \Vdash A[u/x^\tau]$. By induction hypothesis, $[\![A[u/x^\tau]]\!] = \mathtt{True}$. Therefore, $[\![\exists x^\tau A]\!] = \mathtt{True}$.

◀

Proposition 18 is very important since ensure that proof-like realizers produce *correct* constructive content for the formulas they realize. For instance, if $t \Vdash \exists x^\tau A$, then $\pi_0 t = u$ for some closed term u of HA^ω and $[\![A[u/x]]\!] = \mathtt{True}$. Thus, our realizability can be used to extract in an effective way sound witnesses from proofs in $\mathsf{HA}^\omega + \mathsf{MP}$ of $\to$-free formulas. Proposition 18 is not true for all formulas, since the Axiom of Choice is realizable, as in Kreisel's modified realizability, but not true in the syntactical model. But we conjecture that Proposition 18 can be strengthened further and that many kind of formulas containing implications are true when realized. However, for reasons of space and complexity we do not address this matter here.

7 Concluding Remarks and Further Works

As remarked in the introduction, there are several constructive interpretations of Markov's Principle [10, 5, 11]. While the semantics are quite different from each other, it is quite clear that the computational mechanisms employed by the extracted programs are essentially the same. Our realizability is no exception and exploits, as all the other interpretations, a proof of $\neg\forall x^{\mathbb{N}} P$ in order to get a witness for $\exists x^{\mathbb{N}} P^\perp$.

However, it is clear that our realizability is intensionally different from the Dialectica, it is simpler and the term assignment for extracting programs is much lighter. It remains to

establish the exact relationship between the two notions: are they equivalent? We conjecture that in most cases there is a translation between realizers of formulas in our sense and terms witnessing their Dialectica interpretation.

Our realizability appears also less ad hoc then Avigad's smooth version [3] of Coquand-Hofmann translation, which requires an usual forcing style definition, with conditions being set of purely universal formulas. With that approach one must always refer to these conditions, which are used to interpret Markov's Principle, even when considering other formulas or axiom schemes (for example, one may like to interpret countable choice, which has nothing to do with MP).

We also remark that our realizability has not been formulated as a syntactical formula translation. Indeed it is not trivial to formalize it in such a way, since we have employed several syntactical tools, as the notion of proof-like term and the normalization theorem. However, we claim to be able to formulate realizability as a formula translation in the style of modified realizability. Once formalized, we also claim that our realizability can be used to obtain with new methods some conservativity results, for example the one stating that $\mathsf{HA}^\omega + \mathsf{MP}$ is conservative over HA^ω for $\to$-free arithmetical formulas.

Finally, compared with Herbelin [11], we employ a purely functional language, while he uses exception handling mechanisms.

Another way of extending this work is to interpret the generalized Markov's Principle:

$$\mathsf{GMP} : \neg\forall x^\tau P \to \exists x^\tau P^\perp .$$

It is indeed reasonable that the methods of this paper can be refined in order to interpret also this axiom.

References

1 F. Aschieri: *A Constructive Analysis of Learning in Peano Arithmetic*, Annals of Pure and Applied Logic **162**(11), 2012.

2 F. Aschieri, S. Berardi: *A New Use of Friedman's Translation: Interactive Realizability*, Logic, Construction, Computation, Ontos Mathematical Logic **3**, 11–50, 2011.

3 J. Avigad: *Interpreting Classical Theories in Constructive Ones*, Journal of Symbolic Logic **65**, 1785–1812, 2000.

4 T. Coquand: *A Semantic of Evidence for Classical Arithmetic*, Journal of Symbolic Logic **60**, 325–337,1995.

5 T. Coquand, M. Hofmann: *A New Way of Establishing Conservativity of Classical Systems over their Intuitionistic Versions*, Mathematical Structures in Computer Science **9**(4), 323–333, 1999.

6 J. Diller: *Logical Problems of Functional Intepretations*, Annals of Pure and Applied Logic **114**, 27–42, 2002.

7 H. Friedman: *Classically and Intuitionistically Provable Recursive Functions*, Lecture Notes in Mathematics **669**, 21–27, 1978.

8 J.-Y. Girard, Y. Lafont, P. Taylor: *Proofs and Types*, Cambridge University Press. 1989.

9 Girard, J.-Y.: *Proof Theory and Logical Complexity*, Studies in Proof Theory, Bibliopolis, 1987.

10 K. Gödel: *Über eine bisher noch nicht benutzte Erweiterung des finiten Standpunktes*, Dialectica **12**, 280–287, 1958.

11 H. Herbelin: *An Intuitionistic Logic that Proves Markov's Principle*, Proceedings of Logic in Computer Science, 50–56, 2010.

12 J. Hintikka, G. Sandu: *Game-Theoretical Semantics* in Handbook of Language and Computation, MIT Press, 1997.

13 S. C. Kleene: *On the interpretation of intuitionistic number theory.* Journal of Symbolic Logic **10**(4),109–124, 1972.

14 G. Kreisel: *Interpretation of Analysis by Means of Constructive Functionals of Finite Types.* Constructivity in Mathematics, 101–128, North-Holland, 1959.

15 G. Kreisel: *On Weak Completeness of Intuitionistic Predicate Logic*, Journal of Symbolic Logic **27**, 1962.

16 J. Mitchell: *Foundations for Programming Languages*, MIT Press, 2000.

17 P. Oliva: *Unifying Functional Intepretations*, Notre Dame Journal of Formal Logic **47**(2), 263–290, 2006.

18 D. Prawitz: *Ideas and Results in Proof Theory.* In Fenstad, ed., Proceedings of the 2nd Scandinavian Logic Symposium, 235–307, North-Holland, 1972.

19 M. H. Sorensen, P. Urzyczyn: *Lectures on the Curry-Howard isomorphism*, Studies in Logic and the Foundations of Mathematics **149**, Elsevier, 2006.

20 A. Troelstra: *Notions of Realizability for Intuitionistic Arithmetic and Intuitionistic Arithmetic in all Finite Types*, in Fenstad, ed., Proceedings of the 2nd Scandinavian Logic Symposium, 369–405, North-Holland,1972.

21 A. Troelstra: *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, Lectures Notes in Mathematics **344**, Springer-Verlag, 1973.

22 A. Troelstra: *Realizability*, in S. Buss, ed., Handbook of Proof Theory, Studies in Logic and in the Foundation of Mathematics, Elsevier, 1998.

23 A. Troelstra, D. van Dalen: *Constructivism in Mathematics Volume I*, North-Holland, 1988.

Formally Verified Implementation of an Idealized Model of Virtualization

Gilles Barthe[1], Gustavo Betarte[2], Juan Diego Campo[2], Jesús Mauricio Chimento[3], and Carlos Luna[2]

1 **IMDEA Software, Madrid, Spain**
gilles.barthe@imdea.org
2 **InCo, Facultad de Ingeniería, Universidad de la República, Uruguay**
{gustun,jdcampo,cluna}@fing.edu.uy
3 **FCEIA, Universidad Nacional de Rosario, Argentina**
checholcc@gmail.com

Abstract

VirtualCert is a machine-checked model of virtualization that can be used to reason about isolation between operating systems in presence of cache-based side-channels. In contrast to most prominent projects on operating systems verification, where such guarantees are proved directly on concrete implementations of hypervisors, VirtualCert abstracts away most implementations issues and specifies the effects of hypervisor actions axiomatically, in terms of preconditions and postconditions. Unfortunately, seemingly innocuous implementation issues are often relevant for security. Incorporating the treatment of errors into VirtualCert is therefore an important step towards strengthening the isolation theorems proved in earlier work. In this paper, we extend our earlier model with errors, and prove that isolation theorems still apply. In addition, we provide an executable specification of the hypervisor, and prove that it correctly implements the axiomatic model. The executable specification constitutes a first step towards a more realistic implementation of a hypervisor, and provides a useful tool for validating the axiomatic semantics developed in previous work.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.4.6 Security and Protection

Keywords and phrases virtualization, cache and TLB, executable specification, error management, isolation

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.45

1 Introduction

Virtualization is a prominent technology that allows high-integrity, safety-critical, systems and untrusted, non-critical, systems to coexist securely on the same platform and efficiently share its resources. To achieve the strong security guarantees requested by these application scenarios, virtualization platforms impose a strict control on the interactions between their guest systems. While this control theoretically guarantees isolation between guest systems, implementation errors and side-channels often lead to breaches of confidentiality, allowing a malicious guest system to obtain secret information, such as a cryptographic key, about another guest system.

Over the last few years, there have been significant efforts to prove that virtualization platforms deliver the expected, strong, isolation properties between operating systems. The most prominent efforts in this direction are within the Hyper-V [13, 19] and L4.verified [17]

projects, which aim to derive strong guarantees for concrete implementations: more specifically, Murray *et al.* [24] recently presented a machine-checked information flow security proof for the seL4 microkernel.

Earlier work

In [4, 5], we have pursued a complementary approach in which verification of isolation properties is conducted in an idealized model of virtualization, named VirtualCert [28]. In comparison with the Hyper-V and L4.verified projects, our proofs are based on an axiomatization of the semantics of a hypervisor, and abstract away many details from the implementation; on the other hand, our model integrates caches and Translation Lookaside Buffers (TLBs), two security relevant components that are not considered in these works. Specifically, we formalize using the Coq proof assistant [31] the semantics of a hypervisor. The semantics accounts for cache-based side-channels, by allowing that a malicious operating system can draw observations from the history of the cache; the treatment of cache-based side-channels is inspired from earlier work on physically observable cryptography [23], but is specialized to caches and TLBs. Then, we prove that, for a wide range of replacement and write policies, flushing the cache upon switching between guest operating systems ensures OS isolation and prevents access-driven cache-based attacks [34].

Contributions

The axiomatic semantics of [4, 5] only considers correct execution. The first contribution of this paper is an implementation of a hypervisor in the programming language of Coq, and a proof that it realizes the axiomatic semantics. Although it remains idealized and far from a realistic hypervisor, the implementation arguably provides a useful mechanism for validating the axiomatic semantics.

The implementation is total, in the sense that it computes for every state and action a new state or an error. Thus, soundness is proved with respect to an extended axiomatic semantics in which transitions may lead to errors. The second contribution of this paper is a proof that OS isolation remains valid for executions that may trigger errors.

Formal language and notation used

The Coq proof assistant [31, 9] is a free open source software that provides a (dependently typed) functional programming language and a reasoning framework based on higher order logic to perform proofs of programs. As examples of its applicability, Coq has been used as a framework for formalizing programming environments and designing special platforms for software verification: the Gemalto and Trusted Logic companies obtained the level CC EAL 7 of certification for their formalization, developed in Coq, of the security properties of the JavaCard platform [11, 10, 1]; Leroy and others developed in Coq a certified optimizing compiler for a large subset of the C programming language [20]; Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [7].

We developed our specification in the Calculus of Inductive Constructions (CIC) [14, 15, 27] – formal language that combines a higher-order logic and a richly-typed functional programming language – using Coq.

We freely use enumerated types, option types, lists, streams and records. Enumerated types and (parametric) sum types are defined using Haskell-like notation; for example, we define for every type T the type $\mathit{option}\ T \stackrel{\text{def}}{=} \mathit{None} \mid \mathit{Some}\ (t : T)$. Record types are of the form $\{l_1 : T_1, \ldots, l_n : T_n\}$, whereas their elements are of the form $\langle t_1, \ldots, t_n \rangle$. Field selection

and field update are respectively written as $r.l$ and $r'[l := v]$; we also use simultaneous field update, which is defined in the usual way. We make an extensive use of partial maps, and bounded partial maps: the type of partial maps from objects of type A into objects of type B is written $A \mapsto B$, and the type of partial maps from A to B whose domain is of size smaller or equal to k (where k is a natural number) is written as $A \mapsto_k B$. Application of a map m on an object a of type A is denoted $m[a]$ and map update is written $m[a := b]$, where b overwrites the value, if any, associated to the key a.

Organization of the paper

The rest of the paper is organized as follows. Section 2 provides a brief account of the basic components of the idealized model focusing on the memory model and the notion of state that has been formalized. Section 3 describes the formal axiomatic and executable semantics of the hypervisor and outlines the proof of correctness of the implementation. In section 4 we present the isolation theorems for the model extended with execution errors. Section 5 discusses related work and concludes.

The formal development can be found in [28], and can be verified using Coq.

2 Background

In this section we provide insights into the basic structures of VirtualCert, namely, the memory model and the set of (valid) states.

Memory model

The formalized memory model includes the main memory of the platform, various kinds of memory spaces, and the cache and the TLB. Our modelling choices are guided by Xen [3], and specifically, by Xen on ARM [16]. As shown in Figure 1 there are three different types of memory addresses: i) the machine addresses (written *madd*) model real hardware memory on the host machine and it is never directly accessed by the guest operating systems, ii) the physical addresses (*padd*) are an abstraction provided by the hypervisor, in order for the guest operating systems to use a contiguous memory space when dealing with its memory pages. The mapping between physical and machine addresses is managed exclusively by the hypervisor, and is transparent to the guest operating systems, and iii) the virtual addresses (*vadd*) are used by applications running on guest operating systems. Each OS has a designated portion of its virtual address space that is reserved for the hypervisor to attend hypercalls. A hypercall interface allows OSs to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. The hypervisor maintains page tables that map virtual addresses to machine addresses in special memory pages. The operating systems must call the hypervisor to modify these mappings.

The figure also shows the cache and the TLB. The cache is indexed by a virtual address, modeling a Virtually Indexed Virtually Tagged (VIVT) cache, and holds a (partial) copy of memory pages. The TLB is used in conjunction with the current page table of the active OS to map virtual to machine addresses. In [5], we present a brief overview of cache management, where we describe different cache types and alternatives policies for implementing cache content management, in particular concerning the update and replacement of cache information.

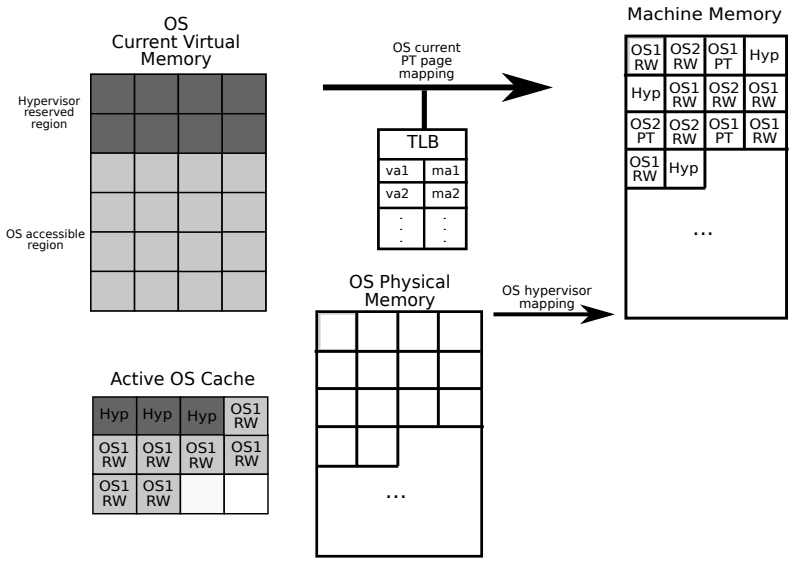


Figure 1 Memory model of the platform.

Platform states

States are modeled as records:

$$
\begin{aligned}
State \stackrel{\text{def}}{=} \{ \quad & oss && : oss\_map, \\
& active\_os && : os\_ident, \\
& mode && : exec\_mode, \\
& activity && : os\_activity, \\
& hypervisor && : hypervisor\_map, \\
& memory && : machine\_memory, \\
& cache && : cache\_vivt, \\
& tlb && : tlb\_struct \} .
\end{aligned}
$$

We define a type *os_ident* of identifiers for guest OSs and a predicate *trusted_os* that separates between trusted and untrusted OSs. The state contains information about each guest OS such as its current page table, and whether the OS has a pending hypercall to be resolved. Formally this information is captured by a mapping *oss_map* that associates OS identifiers with objects of type *os*, where

$$
\begin{aligned}
& os \stackrel{\text{def}}{=} \{curr\_page : padd, hcall : option\ Hyper\_call\} , \\
& oss\_map \stackrel{\text{def}}{=} os\_ident \ \mapsto \ os .
\end{aligned}
$$

The state also stores the current active operating system, and the execution mode of the CPU (*user* or *supervisor* mode). Guest operating systems execute in *user* mode (where some privileged instructions are not available) and the hypervisor executes in *supervisor* mode. The activity registers whether the active OS is currently *running* or *waiting* for a hypercall to be resolved. The mapping, that given an OS returns the corresponding mapping from physical to machine addresses, is formalized as an object of the type *hypervisor_map*, where

$$
hypervisor\_map \stackrel{\text{def}}{=} os\_ident \ \mapsto \ (padd \mapsto \ madd) .
$$

The real platform memory is formalized as a mapping that associates to a machine address a page. A memory page consists of a page content (either a readable/writable value, an OS

page table mapping, or nothing) and a reference to the page owner (the hypervisor, an OS, or none). Formally:

$$
\begin{aligned}
&\mathit{machine\_memory} \stackrel{\text{def}}{=} \mathit{madd} \mapsto \mathit{page}\,,\\
&\mathit{content} \stackrel{\text{def}}{=} \mathit{RW}\ (v : \mathit{option}\ \mathit{Value}) \mid \mathit{PT}\ (\mathit{va\_to\_ma} : \mathit{vadd} \mapsto \mathit{madd}) \mid \mathit{Other}\,,\\
&\mathit{page\_owner} \stackrel{\text{def}}{=} \mathit{Hyp} \mid \mathit{Os}\ (\mathit{osi} : \mathit{os\_ident}) \mid \mathit{No\_Owner}\,,\\
&\mathit{page} \stackrel{\text{def}}{=} \{\ \mathit{page\_content} : \mathit{content}, \mathit{page\_owned\_by} : \mathit{page\_owner}\ \}\,.
\end{aligned}
$$

Finally, the cache and the TLB of the platform are formalized as partial maps, whose domains are bounded in size with positive fixed constants *size_cache* and *size_tlb*:

$$
\begin{aligned}
&\mathit{cache\_vivt} \stackrel{\text{def}}{=} \mathit{vadd} \mapsto_{\mathit{size\_cache}} \mathit{page}\,,\\
&\mathit{tlb\_struct} \stackrel{\text{def}}{=} \mathit{vadd} \mapsto_{\mathit{size\_tlb}} \mathit{madd}\,.
\end{aligned}
$$

We define a notion of valid state, through the predicate *valid_state* on states, that captures essential properties of the platform. The definition is provided in Appendix A.1.

3 Verified implementation

In this section we first provide a short account of the axiomatic semantics of the hypervisor, to proceed to motivate the extension of the model with execution errors. Then we describe the executable specification and show that it constitutes a correct implementation of the behavior specified by the idealized model.

3.1 Actions semantics

The axiomatic semantics of the hypervisor is modeled by defining a set of actions, and providing their semantics as state transformers. Table 1 summarises a small subset of the actions specified in our model. The complete set of actions is included in Appendix A.2. Actions can be classified as follows:

- hypervisor calls `new`, `delete`, `pin`, `unpin` and `lswitch`;
- change of the active OS by the hypervisor (`switch`);
- access, from an OS or the hypervisor, to memory pages (`read` and `write`);
- update of page tables by the hypervisor on demand of an untrusted OS or by a trusted OS directly (`new` and `delete`);
- changes of the execution mode (`chmod`, `ret_ctrl`); and
- changes in the hypervisor memory mapping (`pin` and `unpin`), which are performed by the hypervisor on demand of an untrusted OS or by a trusted OS directly. These actions model (de)allocation of resources.

The behaviour of actions is specified by a precondition *Pre* and by a postcondition *Post* of respective types:

$$
\begin{aligned}
\mathit{Pre} &: \mathit{State} \to \mathit{Action} \to \mathit{Prop}\,,\\
\mathit{Post} &: \mathit{State} \to \mathit{Action} \to \mathit{State} \to \mathit{Prop}\,.
\end{aligned}
$$

Figure 2 provides the axiomatic semantics of the `write` action.

The precondition of the action `write` *va val* says that there exists a machine address *ma* such that *va* is associated to it (*va_mapped_to_ma*) and that the page associated to it in the memory is readable/writable (*is_RW*); that the guest OS activity must be running; and

Table 1 Actions.

`read_hyper` *va*	The hypervisor reads virtual address *va*.
`write` *va val*	A guest OS writes value *val* in virtual address *va*.
`new_tr` *va pa*	The virtual address *va* is mapped to the machine address *ma* in the memory mapping of the trusted active OS, where *pa* translates to *ma* for the active OS.
`switch` *o*	The hypervisor sets *o* to be the active OS.
`lswitch_untr` *o pa*	The hypervisor changes the current memory mapping of the untrusted active OS, to be the one located at physical address *pa*.
`hcall` *c*	An untrusted OS requires privileged service *c* to be executed by the hypervisor.
`pin_untr` *o pa t*	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.
`unpin_untr` *o pa*	The memory page that corresponds to physical address *pa* (for the untrusted OS *o*) is un-registered.

$$
\begin{array}{l}
Pre\ s\ (\texttt{write}\ va\ val) \stackrel{\text{def}}{=} \exists\ (ma : madd),\\
\quad va\_mapped\_to\_ma(s, va, ma)\ \wedge\ is\_RW(s.memory[ma].page\_content)\ \wedge\\
\quad os\_accessible(va)\ \wedge\ s.activity = running\\
\\
Post\ s\ (\texttt{write}\ va\ val)\ s' \stackrel{\text{def}}{=} \exists\ (ma : madd)\ (pg : page),\\
\quad \texttt{let}\ new\_pg\ :=\ \{RW(Some\ val), pg.page\_owned\_by\}\ \texttt{in}\\
\quad va\_mapped\_to\_pg\_cache(s, va, pg)\ \wedge\ va\_mapped\_to\_ma\_cache(s, va, ma)\ \wedge\\
\quad s' = s \left[\begin{array}{rcl} mem & := & (s.memory[ma := new\_pg]),\\ cache & := & cache\_add(fix\_cache\_synonym(s.cache, ma), va, new\_pg),\\ tlb & := & tlb\_add(s.tlb, va, ma)\end{array}\right]
\end{array}
$$

Figure 2 Axiomatic specification of action `write`.

that *va* must be accessible by the active guest OS (*os_accessible*). Its postcondition sets up that the only variations in the state after executing this action can be produced in the value of the page associated to *ma* in memory, and in the values stored in the cache and the TLB. It is not hard to see that, as the cache uses a write-through policy, both the memory and the cache are updated when a write is performed. As explained in [5], a cache c_2 is the result of updating a cache c_1 with a pair *va* and *pg*, written $c_2 = cache\_add(c_1, va, pg)$, iff

$$
\begin{array}{l}
pg\ =\ c_2[va]\ \wedge\\
\forall\ (va' : vadd)\ (pg' : page),\ va\ \neq\ va'\ \rightarrow\ pg'\ =\ c_2[va']\ \rightarrow\ pg'\ =\ c_1[va']\,.
\end{array}
$$

The definition of $c_2 = tlb\_add(c_1, va, ma)$ is analogous. Moreover, in order to avoid aliasing problems we fix synonyms before adding a new entry into the cache using the function *fix_cache_synonym*. The result of $fix\_cache\_synonym(c_1, ma)$ is a cache c_2 whose indexes (virtual addresses) are translated to machine addresses ma' which differ from ma. We recall that we are modeling a VIVT cache.

3.2 Error management

There can be attempts to execute an action on a state that does not verify the precondition of that action. In the presence of one such situation the system answers with a corresponding error code. These error codes are defined in our model by the enumerated type *ErrorCode*.

Table 2 Preconditions and error codes.

Action	*Failure*	*Error Code*
`write` *va val*	$s.aos\_activity \neq running$	$wrong\_os\_activity$
	$\neg\ va\_mapped\_to\_ma(s, va, ma)$	$invalid\_vadd$
	$\neg\ os\_accessible(va)$	$no\_access\_va\_os$
	$\neg\ is\_RW(s.memory[ma].page\_content)$	$wrong\_page\_type$
`new_tr` *va pa*	$s.aos\_activity \neq running$	$wrong\_os\_activity$
	$\neg\ os\_accessible(va)$	$no\_access\_va\_os$
	$\neg\ trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ page\_of\_OS(s.active\_os, pa, ma)$	$wrong\_owner$
`lswitch_untr` *osi pa*	$s.aos\_activity \neq waiting$	$wrong\_os\_activity$
	$trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ is\_PT(s.memory[ma].page\_content)$	$wrong\_page\_type$
	$\neg\ lswitch\_hypercall(s.oss[osi].hcall)$	$wrong\_pending\_hcall$
	$\neg\ page\_of\_OS(s.activeos, pa, ma)$	$wrong\_owner$
`unpin_untr` *osi pa*	$s.aos\_activity \neq waiting$	$wrong\_os\_activity$
	$trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ page\_unpin\_hypercall(s.oss[osi].hcall)$	$wrong\_pending\_hcall$
	$\neg\ pa\_not\_curr\_page(s, s.oss, pa)$	$wrong\_currpage\_add$
	$s.hypervisor[osi][pa] \neq\ ma$	$invalid\_madd$
	$\neg\ no\_va\_mapped\_to\_ma(s, osi, ma)$	$invalid\_vadd$

We define the relation between an error code and the unfulfilled precondition of an action with the predicate *ErrorMsg*. Formally,

$$ErrorMsg : State \to Action \to ErrorCode \to Prop$$

where *ErrorMsg* s a ec means that the execution of the action a in the state s generates the error ec. In Table 2 we show some examples about error codes associated to unverified preconditions of some actions of our model. Notice that in the case of the `write` action, for instance, to each of the propositions that compose the precondition of that action there corresponds an element of *ErrorCode* that indicates the failure of the state s to satisfy that proposition.

Executions with error management

Executing an action a over a state s produces a new state s' and a corresponding answer r (denoted $s \xrightarrow{a/r} s'$), where the relation between the former state and the new one is given by the postcondition relation *Post*.

$$\frac{valid\_state(s) \qquad Pre(s, a) \qquad Post(s, a, s')}{s \xrightarrow{a/ok} s'}$$

$$\frac{valid\_state(s) \qquad ErrorMsg(s, a, ec)}{s \xrightarrow{a/error\ \ ec} s}$$

Whenever an action occurs for which the precondition holds, the (valid) state may change in such a way that the action postcondition is established. The notation $s \xrightarrow{a/ok} s'$ may be read

as *the execution of the action a in a valid state s results in a new state s′*. However, if the precondition is not satisfied, then the state s remains unchanged and the system answer is the error message determined by the relation $ErrorMsg$.

Formally, the possible answers of the system are defined by the following type:

$$Response \stackrel{\text{def}}{=} ok : Response \mid error : ErrorCode \rightarrow Response$$

where *ok* is the answer resulting from a successful execution of an action.

One-step execution with error management preserves valid states, that is to say, the state resulting from the execution of an action is also a valid one.

▶ **Lemma 1** (Validity is invariant).
$\forall\ (s\ s' : State)(a : Action)(r : Response),$
$valid\_state(s)\ \rightarrow s \xrightarrow{a/r} s' \rightarrow valid\_state(s')\ .$

Platform 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 platform.

3.3 Executable specification

The executable specification of the hypervisor has been written using the Coq proof assistant and it ultimately amounts to the definition of functions that implement action execution. The functions have been defined so as to conform to the axiomatic specification of action execution as provided by the idealized model. The implementation of the hypervisor consists of a set of `Coq` functions, such that for every predicate involved in the axiomatic specification of action execution there exists a function which stands for the functional counterpart of that predicate. An important characteristics of our formalization is that the definition of state that is used for defining the executable semantics of the hypervisor is exactly the same as the one introduced in the idealized model. This simplifies the formal proof of soundness between the inductive and the functional semantics of the hypervisor. The execution of the virtualization platform consists of a (potentially infinite) sequence of action executions starting in an (initial) platform state. The output of the execution is the corresponding sequence of memory states (the trace of execution) obtained while executing the sequence of actions.

3.3.1 Action execution

The execution of actions has been implemented as a *step* function, that given a memory state s and an action a invokes the function that implements the execution of a in s, which in turn returns an object res of type *Result*:

$$Result \stackrel{\text{def}}{=} \{\ resp\ :\ Response, st\ :\ State\ \}$$

where *res.resp* is either an error code *ec*, if the precondition of the actions does not hold in state s, or otherwise the value *ok*, and the state *res.st* represents the execution effect. The *step* function acts basically as an action dispatcher. Figure 3, which shows the structure of the dispatcher, details the branch corresponding to the dispatching of action `write`, which is the action we shall use along this section to illustrate the working of the implementation.

The functions invoked in the branches, like *write_safe*, are state transformers whose definition follows this pattern: first it is checked whether the precondition of the action is

```
Definition step s a :=
  match a with
        | ... ⇒ ...
        | Write va val ⇒ write_safe(s, va, val)
        | ... ⇒ ...
  end.
```

Figure 3 The *step* function.

```
Definition write_safe (s : state) (va : vadd) (val : value) : Result :=
  match write_pre(s, va, val) with
        | Some ec ⇒ {error(ec), s}
        | None ⇒ {ok, write_post(s, va, val)}
  end.
```

Figure 4 Execution of `write` action.

```
Definition write_pre (s : state) (va : vadd) (val : value) : option ErrorCode :=
  match get_os_ma(s, va) with
    | None ⇒ Some invalid_vadd
    | Some ma
      ⇒ match page_type(s.memory, ma) with
        | Some RW
          ⇒ match aos_activity(s) with
            | Waiting ⇒ Some wrong_os_activity
            | Running
              ⇒ if vadd_accessible(s, va)
                then None
                else Some no_access_va_os
            end
        | _ ⇒ Some wrong_page_type
        end
  end.
```

Figure 5 Validation of `write` action precondition.

satisfied in state s, and then, if that is the case, the function that implements the execution of the action is invoked, otherwise, the state s, unchanged, is returned along with an appropriate response.

In Figure 4 we show the definition of the function that implements the execution of the `write` action. The Coq code of this function, together with that of the remaining functions, can be found in [28].

The function *write_pre* is defined as the nested validation of each of the properties of the precondition (see Figure 5). The function *write_post*, shown in Figure 6, implements the expected behavior of the `write` action: when a new value has to be written in a certain virtual address *va*, first it must be checked whether *va* is in the cache (i.e. is an index of the cache). If that is the case, then the function updates both the cache and the memory,

```
Definition write_post (s : state) (va : vadd) (val : value) : state :=
  match s.cache[va] with
  | Value old_pg ⇒
    let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
    let val_ma := va_mapped_to_ma_system(s, va) in
    match val_ma with
    | Value ma ⇒
      s · [ mem := s.memory[ma := new_pg],
            cache := fcache_add(fix_cache_synonym(s.cache, ma), va, new_pg) ]
    | Error _ ⇒ s end
  | Error _ ⇒
    match s.tlb[va] with
    | Value ma ⇒
      match s.memory[ma] with
      | Value old_pg ⇒
        let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
        s · [ mem := s.memory[ma := new_pg],
              cache := fcache_add(fix_cache_synonym(s.cache, ma), va, new_pg) ]
      | Error _ ⇒ s end
    | Error _ ⇒
      match va_mapped_to_ma_currentPT(s, va) with
      | Value ma ⇒
        match s.memory[ma] with
        | Value old_pg ⇒
          let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
          s · [ mem := s.memory[ma := new_pg],
                cache := fcache_add(fix_cache_synonym(s.cache, ma), va, new_pg),
                tlb := ftlb_add(s.tlb, va, ma) ]
        | Error _ ⇒ s end
      | Error _ ⇒ s end end end.
```

Figure 6 Effect of `write` execution

because it implements a write-through policy. Otherwise, i.e. if the virtual address va is not already in the cache, the machine address associated to va has to be determined in order to write the new value in memory. First, the TLB is inspected to check whether va has already been translated. If there is a translation of va in the TLB, then the machine address is used to update the memory and the new entry $\langle va,\ new\_pg\rangle$ is added to the cache. If there is no translation of va in the TLB, then the corresponding machine address has to be recovered using the current page table of the active guest OS. Once that translation has been found, the memory is updated, the new entry $\langle va,\ new\_pg\rangle$ is added to the cache and the corresponding translation of va is added to the TLB.

3.3.2 Cache and TLB update

In the axiomatic semantics of cache and TLB management the replacement policy has been left abstract. For the execution semantics we have chosen to implement a simple FIFO replacement mechanism. However, this behavior is encapsulated in the definition

```
Definition fcache_add (c : cache_struct) (va : vadd) (pg : page) : cache_struct :=
  if map_valid_index(c, va)
  then map_add(c, va, pg)
  else if is_full_cache(c)
       then fifo_replace(c, va, pg)
       else fifo_add(c, va, pg).
```

Figure 7 Cache update.

of the functions *fcache_add* and *ftlb_add*, which implement cache and TLB replacement, respectively. Therefore, for the implementation of an alternative replacement policy it suffices to modify correspondingly these two functions leaving the rest of the code unchanged. Figure 7 shows the definition of the *fcache_add* function: first, it is checked whether the virtual address *va* is the index of an entry of the cache *c* (*map_valid_index*). If this is the case, it suffices to perform a simple update of *c* with the page *pg* (caches are implemented as bounded maps of virtual addresses to machine addresses). Otherwise, the behaviour of the function depends on whether *c* has room for a new entry or it is full (*is_full_cache*). If *c* is full, the cache update, and entry eviction, is handled using the FIFO replacement algorithm (*fifo_replace*). If there is room left for a new entry, then *c* must be updated following the FIFO replacement algorithm guidelines for adding new entries in the cache (*fifo_add*). The definitions of the replacement and update function for the TLB are analogous.

3.4 Soundness

We proceed now to outline the proof that the executable specification of the hypervisor correctly implements the axiomatic model. It has been formally stated as a soundness theorem and verified using the Coq proof assistant.

▶ **Theorem 2** (Soundness of hypervisor implementation).
$\forall\ (s : State)\ (a : Action),$
$valid\_state(s) \rightarrow\ s \xrightarrow{a/step(s,a).resp} step(s,a).st\,.$

The proof of this theorem follows by, in the first place, performing a case analysis on $Pre(s,a)$ (this predicate is decidable) and then: if $Pre(s,a)$ applying Lemma 3; otherwise applying Lemma 5.

▶ **Lemma 3** (Soundness of valid execution).
$\forall\ (s : State)\ (a : Action),$
$valid\_state(s) \rightarrow\ Pre(s,a)\ \rightarrow$
$s \xrightarrow{a/ok} step(s,a).st\ \wedge\ step(s,a).resp = ok\,.$

The proof of Lemma 3 proceeds by applying functional induction on $step(s,a)$ and then by providing the corresponding proof of soundness of the function that implements the execution of each action. Thus, in the case of the action `write` we have stated and proved Lemma 4. This lemma, in turn, follows by performing a case analysis on the result of applying the function *write_pre* on *s* and the action: if the result is an error code then the thesis follows by contradiction. Otherwise, it follows by the correctness of the function *write_post*.

▶ **Lemma 4** (Correctness of `write` execution).
$\forall\ (s : State)\ (va : vadd)\ (val : value),$
$valid\_state(s) \rightarrow\ Pre(s, (\texttt{write}\ va\ val)) \rightarrow$
$Post(s, (\texttt{write}\ va\ val), write\_post(s, va, val))\,.$

As to Lemma 5, the proof also proceeds by first applying functional induction on $step(s, a)$. Then, for each action a, it is shown that if $\neg Pre(s, a)$ the execution of the function that implements that action yields the values returned by the branch corresponding to the case that the function that validates the precondition of the action a in state s fails, i.e., an error code ec and the (unchanged) state s.

▶ **Lemma 5** (Soundness of error execution).
$\forall\ (s : State)\ (a : Action),$
$valid\_state(s) \rightarrow\ \neg Pre(s, a) \rightarrow\ \exists\ (ec : ErrorCode),$
$step(s, a).st = s\ \wedge\ step(s, a).resp = error(ec)\ \wedge\ ErrorMsg(s, a, ec)\,.$

4 Isolation

Isolation theorems ensure that the virtualization platform protects guest operating systems against each other, in the sense that a malicious operating system cannot gain information about another victim operating system executing on the same platform. In earlier work [5], we adopted ideas from physical cryptography and in particular the idea of leakage function to model possible leaks of information via the cache, and prove that the virtualization platform can guarantee perfect isolation by flushing the cache at every context switch. In this section, we extend the proof of OS isolation from [5], yielding modifications in some key technical definitions and lemmas below, so that it accounts for errors in execution traces.

4.1 OS Isolation

OS isolation is a 2-safety property [32, 12], cast in terms of two executions of the system, and is closely related to the non-influence property studied by Oheimb and co-workers [25, 26]. Unfortunately, the technology for verifying 2-safety properties is not fully mature, making their formal verification on large and complex programs exceedingly challenging.

Informally, OS isolation states that starting from states with the same information for an operating system *osi*, *osi* cannot distinguish between the two traces, as long as it executes the same actions in both. This captures the idea that the execution of *osi* does not depend on the state or behaviour of the other systems, even in the presence of erroneous executions.

Note that there is one particular error (the *out_of_memory* error in [28]) that can in principle influence the execution of an operating system, if during its execution the platform runs out of memory. Since we are specifically interested in modelling observations on states (and the cache, in particular), we treat this error as transparent for the executing operating system, and only make sure it does not modify the state. This is consistent with what usually happens in real implementations, where there are no data leaks from the victims when the platform runs out of memory, and the only information an attacker learns is the total memory consumption of the other operating systems in the platform. Additionally it is possible, in this case, to assign to each guest OS a fixed pool of memory from which to allocate, so whether allocation succeeds or fails for one OS doesn't depend on what any other guest OS does.

To formalize OS isolation we use a notion of state equivalence w.r.t. an operating system *osi*. The definition of *osi-equivalence* ($\equiv_{osi}$), which is stated in Appendix A.3, coincides

with the one used in [4]; in particular, it does not mention the cache and the TLB. However, one can prove that it entails some form of cache equivalence and TLB equivalence on valid states. Formally, we define two valid states s_1 and s_2 to be cache equivalent for *osi*, written $s_1 \equiv_{osi}^{cache} s_2$, iff *osi* is the active OS in both states and the caches hold equal values for all accessible virtual addresses *va* that are in the domain of the cache of both states, i.e. for all virtual address *va* and pages p_1 and p_2

$$s_1.active\_os = s_2.active\_os = osi \;\rightarrow\; os\_accessible(va) \;\rightarrow\; s_1.cache[va] = p_1 \;\rightarrow\; s_2.cache[va] = p_2 \;\rightarrow\; p_1 = p_2\,.$$

Note that we do not require that the domains of both caches coincide, as it would invalidate the following lemma.

▶ **Lemma 6** (Cache equivalence)**.**
$\forall\ (s_1\ s_2 : State)\ (osi : os\_ident),$
$valid\_state(s_1) \;\rightarrow\; valid\_state(s_2) \;\rightarrow\; s_1 \equiv_{osi} s_2 \;\rightarrow\; s_1 \equiv_{osi}^{cache} s_2\,.$

The notion of TLB equivalence is defined in a similar way. We say that two valid states s_1 and s_2 are TLB equivalent for *osi*, written $s_1 \equiv_{osi}^{tlb} s_2$, iff *osi* is the active OS in both states and for all accessible virtual addresses *va* that are in the domain of the TLB of both states, if the machine address $s_1.tlb[va]$ holds a page with RW memory content, then if *va* appears in $s_2.tlb$, it holds the same page, i.e. for all machine addresses ma_1 and ma_2, and page *pg*:

$$s_1.active\_os = s_2.active\_os = osi \;\rightarrow\; s_1.tlb[va] = ma_1 \;\rightarrow\; s_1.memory[ma_1] = pg \;\rightarrow\; \exists\ (val : Value), pg.page\_content = RW\ (Some\ val) \;\rightarrow\; s_2.tlb[va] = ma_2 \;\rightarrow\; s_2.memory[ma_2] = pg$$

and conversely. We have:

▶ **Lemma 7** (Tlb equivalence)**.**
$\forall\ (s_1\ s_2 : State)\ (osi : os\_ident),$
$valid\_state(s_1) \;\rightarrow\; valid\_state(s_2) \;\rightarrow\; s_1 \equiv_{osi} s_2 \;\rightarrow\; s_1 \equiv_{osi}^{tlb} s_2\,.$

We write $s_1 \equiv_{osi}^{cache,tlb} s_2$ as a shorthand for $s_1 \equiv_{osi} s_2 \;\wedge\; s_1 \equiv_{osi}^{cache} s_2 \;\wedge\; s_1 \equiv_{osi}^{tlb} s_2$. We can now generalize the unwinding lemmas of [4]: the first lemma states that equivalence is preserved by the execution of all actions that do not generate errors.

▶ **Lemma 8** (Step-consistent unwinding lemma)**.**
$\forall\ (s_1\ s_1'\ s_2\ s_2' : State)\ (a : Action)\ (osi : os\_ident),$
$s_1 \equiv_{osi} s_2 \;\rightarrow\; os\_action(s_1, a, osi) \;\rightarrow\; os\_action(s_2, a, osi) \;\rightarrow\; s_1 \xrightarrow{a/ok} s_1' \;\rightarrow\; s_2 \xrightarrow{a/ok} s_2' \;\rightarrow\; s_1' \equiv_{osi}^{cache,tlb} s_2'\,.$

where $os\_action(s, a, osi)$ denote that action a is an action successfully executed by the OS *osi* in the state s; in particular, its execution does not cause an error. Note that an execution that fails does not generate a change in the system state.

The second lemma states that execution does not alter the state of non-active OSs, or active OS if it performs an execution that fails.

▶ **Lemma 9** (Locally preserves unwinding lemma)**.**
$\forall\ (s\ s' : State)\ (a : Action)\ (r : Response)\ (osi : os\_ident),$
$\neg\ os\_action(s, a, osi) \;\rightarrow\; s \xrightarrow{a/r} s' \;\rightarrow\; s \equiv_{osi}^{cache,tlb} s'\,.$

4.2 OS isolation in execution traces

The extension to traces of the relation one-step execution with error management is defined as follows: an execution trace is defined as a stream (an infinite list) of states that are related by the transition relation $\xrightarrow{a/r}$, i.e. an object of the form

$$s_0 \xrightarrow{a_0/r_0} s_1 \xrightarrow{a_1/r_1} s_2 \xrightarrow{a_2/r_2} s_3 \ldots$$

In the sequel, we let $t[i]$ denote the i-th state of a trace t and we use $s \xrightarrow{a/r} t$ to denote the trace obtained by prepending the valid execution step $s \xrightarrow{a/r} t[0]$ to a trace t. We let $Trace$ define the type of these traces. Isolation properties are eventually expressed on execution traces, rather than execution steps.

Non-influencing execution (errors)

Using the unwinding lemmas previously presented, one can establish a non-influence result in the style of [25]. We define for each operating system *osi* a predicate *same_os_actions* stating that two traces have the same set of actions w.r.t. *osi*; so that two traces are related iff they perform the same valid *osi*-actions. Then we define two traces t_1 and t_2 to be *osi-equivalent*, written $t_1 \approx_{osi,cache,tlb} t_2$, co-inductively by the following rules:

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad \neg\ os\_action(s, a, osi)}{(s \xrightarrow{a/r} t_1) \approx_{osi,cache,tlb} t_2}$$

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad \neg\ os\_action(s, a, osi)}{t_1 \approx_{osi,cache,tlb} (s \xrightarrow{a/r} t_2)}$$

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad os\_action(s_1, a, osi) \quad os\_action(s_2, a, osi) \quad s_1 \equiv_{osi}^{cache,tlb} s_2}{(s_1 \xrightarrow{a/ok} t_1) \approx_{osi,cache,tlb} (s_2 \xrightarrow{a/ok} t_2)}$$

▶ **Theorem 10** (OS isolation)**.**
$\forall\ (t_1\ t_2 : Trace)\ (osi : os\_ident)$,
$same\_os\_actions(osi, t_1, t_2) \rightarrow (t_1[0] \equiv_{osi} t_2[0]) \rightarrow t_1 \approx_{osi,cache,tlb} t_2$.

OS isolation formally establishes that two traces are *osi*-equivalent if they have the same set of *osi*-actions and if their initial states are *osi*-equivalent. The proof of OS isolation is based on co-induction principles and on the previous unwinding lemmas. Note that the definition of *osi*-equivalent traces conveniently generalizes the notion used in [4] (by allowing related traces to differ in the number of actions executed by other OSs) and extends that presented in [5] considering executions with error management. In particular, Theorem 10 states that the OS isolation property introduced in [5] is also valid in the context of executions that include error handling, considering that an *osi*-action is an action successfully executed by the OS *osi*.

Though it is left as future work, it is interesting to comment on the validity of isolation properties under other policies. On the one hand, the replacement policy for the cache and the TLB is left abstract in our model, so any reasonable algorithm will preserve these properties (as embodied e.g. in the definition of *cache_add* in Section 3.1). On the other hand, we have fixed a *write-through* policy for the main memory: this policy entails that updates to memory pages are done simultaneously to the cache and main memory, and we have used throughout the development the invariant property that cache data is included in the memory. This inclusion property will not hold if we were to use a *write-back* policy,

in which written entries are marked dirty and updates to main memory are done when a page is removed from the cache. We believe that it remains possible to prove strong isolation properties under the *write-back* policy, since page values, even if different in memory, will be equal if we consider the cache and memory together.

Finally, the flushing policy is assumed to be a total flush on switch and local switch execution. An alternative would be to tag cache (and TLB) entries with the virtual spaces allowed to access the entry. This will not have as much impact on the current model as the write policy, though changes will need to be done to the cache definition to include the tags. Isolation properties would still hold, given correct semantics of access control of cache entries.

5 Related work and conclusion

Thanks to recent advances in verification technology, it is now becoming feasible to verify formally realistic specifications and implementations of operating systems. A recent account of existing efforts can be found in the surveys [18, 30]. Many of these works focus on functional correctness of the hypervisor; one notable exception is [24], which proves that the seL4 microkernel guarantees information flow security; this work builds on a proof of integrity [29] and a proof of correctness and culminates a 30+ man-year verification effort. In addition, many of these works do not consider cache, which is a distinctive focus of our work. On the other hand, most of these works focus on implementations, and provide an explicit treatment of errors – that was missing in our earlier work [5].

Moving away from OS verification, many works have addressed the problem of relating inductively defined relations and executable functions, in particular in the context of programming language semantics. For instance, Tollitte *et al* [33] show how to extract a functional implementation from an inductive specification in the Coq proof assistant. Similar approaches exist for Isabelle, see e.g. [8]. Earlier, alternative approaches such as [2, 6] aim to provide reasoning principles for executable specifications.

We have enhanced the idealized model of virtualization considered in [5] with an explicit treatment of errors, and showed that OS isolation is preserved in this setting. Moreover we have implemented an executable specification that realizes the axiomatic semantics used in [5]. The formal development in this paper is about 15 kLOC of Coq, where 8k correspond to the verified executable specification and 7k to the OS isolation proof on the extended model with errors. In [28] we derive two certified hypervisor implementations, using the extraction mechanism of Coq [22, 21], in functional languages Haskell and OCaml.

In future work, we intend to implement alternative executable semantics for different models of cache and policies. Moreover, we plan to use our extended model as a basis for investigating whether error management can lead to side-channels.

Acknowledgements. The authors want to thank TYPES reviewers for helpful feedback on the paper.

The work of Gilles Barthe has been partially funded by European Project FP7 256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10 and Madrid Regional project S2009TIC-1465 PROMETIDOS and the work of Gustavo Betarte, Juan Diego Campo and Carlos Luna by Uruguayan project CSIC-Convocatoria 2012, Proyectos I + D, VirtualCert – Fase II.

References

1 June Andronick. *Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation*. PhD thesis, Université Paris-Sud, 2006.

2 Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000.

3 P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

4 G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *FM 2011*, pages 231–245. Springer-Verlag, 2011.

5 G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *CSF 2012*, pages 186–197, 2012.

6 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.

7 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009.

8 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.

9 Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

10 G. Betarte, E. Giménez, C. Loiseaux, and B. Chetali. FORMAVIE: Formal Modeling and Verification of the Java Card 2.1.1 Security Architecture. In *Proceedings of eSmart'02*, 2002.

11 Boutheina Chetali and Quang-Huy Nguyen. About the world-first smart card certificate with eal7 formal assurances. Slides 9th ICCC, Jeju, Korea, September 2008.

12 M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

13 E. Cohen. Validating the microsoft hypervisor. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM'06*, volume 4085 of *LNCS*, pages 81–81. Springer, 2006.

14 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

15 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

16 J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer and Communications Networking Conference*, 2008.

17 G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

18 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas

Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

19 D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with vcc. In A. Cavalcanti and D. Dams, editors, *FM 2009*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.

20 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

21 P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

22 Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

23 Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *TCC 2004*, pages 278–296, 2004.

24 T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *Proc. of the 2013 IEEE Symp. on Security and Privacy (SP'13)*, pages 415–429, 2013.

25 David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004.

26 David von Oheimb, Volkmar Lotz, and Georg Walter. Analyzing SLE 88 memory management security using Interacting State Machines. *International Journal of Information Security*, 4(3):155–171, 2005.

27 C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In M. Bezem and J. F. Groote, editors, *1st Int. Conf. on Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328–345. Springer-Verlag, 1993.

28 The VirtualCert project. Supporting Coq formalization. See `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

29 Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP 2011*, Nijmegen, The Netherlands, 2011.

30 Zhong Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

31 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2012.

32 T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Proceedings of SAS'05*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, 2005.

33 Pierre-Nicolas Tollitte, David Delahaye, and Catherine 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.

34 Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

A Appendix

A.1 Valid state

We define a notion of valid state that captures essential properties of the platform. Formally, the predicate *valid_state* holds on state s if s satisfies the following properties:

- if the active OS is in *running* mode then no hypercall requested by it is pending;
- if the hypervisor or a trusted OS (respectively untrusted OS) is running the processor must be in supervisor (respectively user) mode;
- the hypervisor maps an OS physical address to a machine address owned by that same OS. This mapping is also injective;
- all page tables of an OS o map virtual addresses to pages owned by o;
- the current page table of any OS is owned by that OS;
- any machine address which is associated to a virtual address in a page table has a corresponding pre-image, which is a physical address, in the hypervisor mapping;
- all cache keys are related in a page table mapping of the memory;
- all cache pages have the same owner and type as those in machine memory;
- if *va* is translated into *ma* according to the TLB, then the machine address *ma* is associated to *va* in the active memory mapping.

All properties have a straightforward interpretation in our model. For example, the first property is captured by the proposition:

$$\forall\ osi : os\_ident,\ trusted\_os(osi) \rightarrow (s.oss[osi]).hcall\ =\ None\,.$$

A.2 Actions

Table 3 summarises the complete set of actions specified in the model, and their effects.

A.3 Observational equivalence of states

We say that two states s_1 and s_2 are *osi-equivalent*, written $s_1 \equiv_{osi} s_2$, iff:

- *osi* is the active OS in both states and the processor mode is the same, or the active OS is different to *osi* in both states;
- *osi* has the same hypercall in both states, or no hypercall in both states;
- the current page tables of *osi* are the same in both states;
- all page table mappings of *osi* that map a virtual address to a RW page in one state, must map that address to a page with the same content in the other;
- the hypervisor mappings of *osi* in both states are such that if a given physical address maps to some RW page, it must map to a page with the same content on the other state.

Note that we cannot require that memory contents be the same in both states for them to be *osi*-equivalent, because on a `page_pin` action, the hypervisor can assign an arbitrary (free) machine address to the OS, so we consider *osi*-equivalence without taking into account the actual value of the machine addresses assigned. In particular, two *osi*-equivalent states can have different page table memory pages, which contain mappings from virtual to arbitrary machine addresses, but such that the content at these machine addresses be the same in both states, if it corresponds to an RW page.

Table 3 Full set of actions.

`read` *va*	A guest OS reads virtual address *va*.
`read_hyper` *va*	The hypervisor reads virtual address *va*.
`write` *va val*	A guest OS writes value *val* in virtual address *va*.
`write_hyper` *va val*	The hypervisor writes value *val* in virtual address *va*.
`new_tr` *va pa*	The virtual address *va* is mapped to the machine address *ma* in the memory mapping of the trusted active OS, where *pa* translates to *ma* for the active OS.
`new_untr` *o va pa*	The hypervisor adds (on behalf of the OS *o*) a new ordered pair (mapping virtual address *va* to the machine address *ma*) to the current memory mapping of the untrusted OS *o*, where *pa* translates to *ma* for *o*.
`new_hyper` *va ma*	The hypervisor adds a new ordered pair to the current memory mapping of the active OS (mapping virtual address *va* to the machine address *ma*) for his own purposes.
`del_tr` *va*	The trusted active OS deletes the ordered pair that maps virtual address *va* from its memory mapping.
`del_untr` *o va*	The hypervisor deletes (on behalf of the *o* OS) the ordered pair that maps virtual address *va* from the current memory mapping of *o*.
`del_hyper` *va*	The hypervisor deletes (for its own purposes) the ordered pair that maps virtual address *va* from the current memory mapping of the active OS.
`switch` *o*	The hypervisor sets *o* to be the active OS.
`lswitch_tr` *pa*	The trusted active OS changes its current memory mapping to be the one located at physical address *pa*. This action corresponds to a traditional context switch by the active OS.
`lswitch_untr` *o pa*	The hypervisor changes the current memory mapping of the untrusted active OS, to be the one located at physical address *pa*.
`silent`	Represents the silent action (the system does not advertise any effects).
`hcall` *c*	An untrusted OS requires privileged service *c* to be executed by the hypervisor.
`ret_ctrl`	Returns the execution control to the hypervisor.
`chmod`	The hypervisor changes the execution mode from supervisor to user mode, if the active OS is untrusted, and gives to it the execution control.
`pin_tr` *pa t*	The memory page that corresponds to physical address *pa* (for the active OS) is registered and classified with type *t*.
`pin_untr` *o pa t*	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.
`unpin_tr` *pa*	The memory page that corresponds to physical address *pa* (for the active OS) is un-registered.
`unpin_untr` *o pa*	The memory page that corresponds to physical address *pa* (for the untrusted OS *o*) is un-registered.

Ramsey Theorem for Pairs As a Classical Principle in Intuitionistic Arithmetic*

Stefano Berardi[1] and Silvia Steila[2]

1 **Dipartimento di Informatica, Università degli studi di Torino**
Corso Svizzera 185 Torino, Italia
`stefano@di.unito.it`

2 **Dipartimento di Informatica, Università degli studi di Torino**
Corso Svizzera 185 Torino, Italia
`steila@di.unito.it`

Abstract

We produce a first order proof of a famous combinatorial result, Ramsey Theorem for pairs and in two colors. Our goal is to find the minimal classical principle that implies the "miniature" version of Ramsey we may express in Heyting Arithmetic HA. We are going to prove that Ramsey Theorem for pairs with recursive assignments of two colors is equivalent in HA to the sub-classical principle Σ^0_3-LLPO. One possible application of our result could be to use classical realization to give constructive proofs of some combinatorial corollaries of Ramsey; another, a formalization of Ramsey in HA, using a proof assistant.

In order to compare Ramsey Theorem with first order classical principles, we express it as a schema in the first order language of arithmetic, instead of using quantification over sets and functions as it is more usual: all sets we deal with are explicitly defined as arithmetical predicates. In particular, we formally define the homogeneous set which is the witness of Ramsey theorem as a Δ^0_3-arithmetical predicate.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases formalization, constructivism, classical logic, Ramsey theorem

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.64

1 Introduction

The purpose of this paper is to study, from the viewpoint of first order arithmetic, Ramsey Theorem [15] for pairs for recursive assignments of two colors, in order to find some principle of classical logic equivalent to it in Intuitionistic Arithmetic HA. Ramsey theorem is not intuitionistically provable, and a priori, it is not evident whether a classical principle expressing Ramsey in intuitionistic arithmetic exists. Our long-time research goal is to study the constructive content of corollaries in first order arithmetic of Ramsey Theorem using interactive realizability, and to this aim we want to find the statement and the proof of Ramsey in first order arithmetic requiring the minimum amount of classical logic. In the PhD thesis of Giovanni Birolo [4] there is an example of a constructive study of a classical proof obtained by interactive realizability. Birolo studied a geometric property that required

* This work was partially supported by the Accademia delle Scienze di Torino, the PRIN 2010 project "Metodi logici per il trattamento dell'informazione", and the Doctoral School of Sciences and Innovative Technologies – Computer Science (Università degli studi di Torino).

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 64–83

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

the law of Excluded Middle of level one (EM_1); for Ramsey, the required principles are higher than EM_1 in the hierarchy of classical principles presented in [1].

Our study of Ramsey Theorem differs from the results in Classical Reverse Mathematics ([5], [6], [13], [8]) in many aspects. We already stressed that we formulate Ramsey in first order arithmetic, replacing set variables with explicit set definitions. Besides, Classical Reverse Mathematics is interested in the necessary set existence axioms needed to proof a theorem and investigates the minimum restriction of the induction schema required in a proof, while they assume the entire Excluded Middle schema. Our work may be considered a kind of Intuitionistic Reverse Mathematics: we assume the entire induction schema, and we investigate the minimum restriction of the Excluded Middle Schema and of some other classical schemas required in a classical proof. Therefore our approach is different from what Ishihara calls Constructive Reverse Mathematics in [9]. Ishihara works in Bishop's Constructive Mathematics which is an informal mathematics using intuitionistic logic and assuming some function existence axioms; instead, he does not study the level of classical principles used in the proof.

As regards the comprehension axiom, instead, there are some links with Classical Reverse Mathematics. Recall that the description axiom says that each arithmetic binary predicate that is fully and uniquely defined is a graph of some function: $\mathbb{N} \to \mathbb{N}$. If we add function variables and we assume the description axiom, the Excluded Middle for an arithmetic predicate and the comprehension axiom for the same predicate are equivalent in HA + functions.

We may stress the difference between the two approaches through an example. Let consider the Infinite Pigeonhole Principle. On the one hand, in reverse mathematics, this principle is equivalent to $B\Sigma^0_2$(the bounding principle for Σ^0_2-formulas, see [16]) which is equivalent to Δ^0_2-induction ([17]). On the other hand, in our setting, it is a consequence of the law of Excluded Middle of level two: EM_2. In [12] Liu considered the base system for reverse mathematics RCA_0, in which we assume the entire Excluded Middle, but only induction for Σ^0_1 formulas and recursive comprehension. Liu proved that Ramsey Theorem for pairs in two colors does not imply WKL_0, Weak König's Lemma for recursive trees, in RCA_0. Instead in [11] Kohlenbach and Kreuzer proved in iRCA^*_0, the intuitionistic system corresponding to RCA^* (Σ^0_0-induction, exponentiation axioms but no excluded middle), that Ramsey Theorem for pairs implies Π^0_2-LEM, which is more than WKL_0. In this work we drop function and set variables, and we consider Heyting Arithmetic HA, in which we have no Excluded Middle Schema but we have the full induction schema. Under these assumptions, we prove that recursive Ramsey Theorem for pairs in two colors is equivalent to Σ^0_3-LLPO (Lesser Limited Principle of Omniscience for Σ^0_3 predicates, a principle weaker than full Excluded Middle, but stronger than WKL_0, which we explain below).

Our study of Ramsey Theorem differs also from the no-counterexample [2], since we do not transform Ramsey Theorem into some weaker and constructively provable statement, but we study the minimum restriction of the Excluded Middle schema required to prove the original result in HA. We differ from the dialectica interpretation ([11], [14]), because it transforms RT^2_2 into a constructively provable, classically equivalent statement and deletes the non-constructive content leaving only the combinatorial core. Moreover the dialectica interpretation requires complex types and variables for each type, while we use the type of natural numbers and of functions over natural numbers only, and no function variable.

At the beginning of this work, in a private communication, Alexander Kreuzer conjectured that Erdős Rado proof of Ramsey Theorem may be formalized in $\mathrm{HA} + \mathrm{EM}_4$, Excluded Middle restricted to Σ^0_4 formulas. We prove that he was right. Moreover, by modifying

Jockusch's proof of Ramsey [10] (that is already a modified version of Erdős Rado proof of the same result) we prove that the classical principle Σ^0_3-LLPO is in fact equivalent to Ramsey Theorem in HA. Σ^0_3-LLPO (see [1]) is a classical principle weaker than Excluded Middle Schema for Σ^0_3 formulas, which may be restated as the conjunction of Excluded Middle for Σ^0_2 formulas and De Morgan Laws for Σ^0_3 formulas. If we add Choice to HA, Σ^0_3-LLPO is equivalent to WKL_3, Weak König's Lemma for Σ^0_2 trees.

We hope to apply, in future works, the method called interactive realizability to understand and explain the computational content of Ramsey Theorem, and to find new constructive proofs for some consequences of it. The interactive realizability is a realizability interpretation for first order classical arithmetic introduced in 2008 by Stefano Berardi and Ugo de' Liguoro [3]. If a corollary of Ramsey Theorem is a consequence of Intuitionistic Ramsey Theorem, an alternative method to prove it constructively could be to use the Coquand's work [7]. However his proof use the Brouwer's thesis, so this method does not guarantee a proof in HA.

This is the plan of the paper. In Section 2 we explain how to state Ramsey Theorem without using functions and set variables; in Section 3 we prove that Ramsey Theorem implies Σ^0_3-LLPO and in Section 4, by modifying Jockusch's proof, we prove the opposite implication. In the conclusions we discuss the interest of the equivalence with Σ^0_3-LLPO.

2 Ramsey Theorem and Classical Principles for Arithmetic

In this section we introduce some notations for Ramsey Theorem and for some classical principles. Any natural number n is identified with the set $\{0, \ldots, n-1\}$. We use $\mathbb{N}$ to denote the least infinite ordinal, which is identified with the set of natural numbers. For any set X and any natural number r,

$$[X]^r = \{Y \subseteq X \mid |Y| = r\}$$

denotes the set of subsets of X of cardinality r. If $r = 1$ then $[\mathbb{N}]^r$ is the set of singleton subsets of $\mathbb{N}$, and just another notation for $\mathbb{N}$. If $r = 2$ then $[\mathbb{N}]^2$ is the complete graph on $\mathbb{N}$: we think of any subset $\{x, y\}$ of $\mathbb{N}$ with $x \neq y$ as an edge of the graph. We will think that each edge $\{x, y\}$ has direction from $\min\{x, y\}$ to $\max\{x, y\}$. Let $n, m \in \mathbb{N}$, then a map $f : [\mathbb{N}]^r \to n$ is called a coloring of $[\mathbb{N}]^r$ with n colors. If $r = 2$ and $f(\{x, y\}) = c < n$, then we say that the edge $\{x, y\}$ has color c. If $f : [\mathbb{N}]^r \to n$ is a map then for all $X \subseteq \mathbb{N}$ we denote with $f''[X]^r$ the set of colors of hyper-edges of X, that is:

$$f''[X]^r = \{k \in \mathbb{N} \mid \exists e \in [X]^r \text{ such that } f(e) = k\}.$$

We say that $X \subseteq [\mathbb{N}]^r$ is homogeneous for f, or f is homogeneous on X, if all hyper-edges of X have the same color, that is, there exists $k < n$ such that $f''[X]^r = \{k\}$. We also say that X is homogeneous for f in color k. If $r = 1$ we can think of the function f as a point coloring map on natural numbers. In this case an homogeneous set X is any set of points of $\mathbb{N}$ which all have the same color. If $r = 2$ we can think of the function f as an edge coloring of a graph that has as its vertices the natural numbers. In this case an homogeneous set X is any set of points of $\mathbb{N}$ whose connecting edges all have the same color.

We denote Heyting Arithmetic, with one symbol and axioms for each primitive recursive map, with HA. We work in the language for Heyting Arithmetic with all primitive recursive maps, extended with the symbols $\{f_0, \ldots, f_n\}$, where n is a natural number and f_i denotes a total recursive function for all $i < n + 1$. These f_i will indicate an arbitrary coloring in the formulation of Ramsey Theorem below. If $P = \forall x_1 \exists x_2 \ldots p(x_1, x_2, \ldots)$, with p arithmetic

atomic formula, and $Q = \exists x_1 \forall x_2 \dots \neg p(x_1, x_2 \dots)$, then we say that P, Q are dual each other and we write $P^\perp = Q$ and $Q^\perp = P$. Dual is defined only for prenex formulas as P, Q. We consider the classical principles as statement schemas as in [1]. A conjunctive schema is a set C of arithmetical formulas, expressing the second order statement "for all A in C, A holds" in a first order language. We prove a conjunctive schema C in HA if we prove any A in C in HA. A conjunctive schema C implies a formula A in HA if $s_1 \wedge \dots \wedge s_n \vdash A$ in HA for some $s_1, \dots, s_n \in C$. The conjunctive schema C implies another conjunctive schema C' in HA if C implies A in HA for any A in C'. In order to express Ramsey Theorem we also have to consider the dual concept of disjunctive schema D, expressing the second order statement "for some A in D, A holds" in a first order language. We prove a disjunctive schema D in HA if we prove $s_1 \vee \dots \vee s_n$ in HA for some $s_1, \dots, s_n \in D$. A disjunctive schema D implies a formula A in HA if $s \vdash A$ in HA for all $s \in D$.

The infinite Ramsey Theorem is a very important result for finite and infinite combinatorics. In this paper we study Ramsey Theorem in two colors, for singletons and for pairs. They are informally stated as follows:

▶ **$\mathbf{RT^1_2(\Sigma^0_n)}$.** For any coloring $c_a : \mathbb{N} \to 2$ of vertices with a parameter a, there exists an infinite subset of $\mathbb{N}$ homogeneous for the given coloring. ($c_a \in \Sigma^0_n$).

▶ **$\mathbf{RT^2_2(\Sigma^0_n)}$.** For any coloring $c_a : [\mathbb{N}]^2 \to 2$ of edges with a parameter a, there exists an infinite subset of $\mathbb{N}$ homogeneous for the given coloring. ($c_a \in \Sigma^0_n$).

$\mathrm{RT}^2_2(\Sigma^0_0)$ (respectively $\mathrm{RT}^1_2(\Sigma^0_0)$) says that given a family $\{c_a \mid a \in \mathbb{N}\}$ of recursive edge (vertex) colorings of a graph with $\mathbb{N}$ vertices, then for any coloring there exists a subgraph with $\mathbb{N}$ vertices such that each edge (vertex) of the subgraph has the same color.

In this work we formalize Ramsey Theorem for two colors, for pairs (respectively, for singletons) and for recursive colorings by the following disjunctive schema which we call Ramsey schema R:

$$R := \{\forall a(B(., c_a) \text{ infin. hom. black } \vee W(., c_a) \text{ infin. hom. white}) \mid B, W \text{ arithm. predic.}\}.$$

Here $c = \{c_a \mid a \in \mathbb{N}\}$ denotes any recursive family of recursive assignment of two colors, black and white. A sufficient condition to prove Ramsey schema is to find at least two predicates B, W and a proof of $\forall a(B(., c_a)$ infinite homogeneous black $\vee$ $W(., c_a)$ infinite homogeneous white) in HA. For short we say that for each recursive family of recursive colorings there is an homogeneous set.

The conjunctive schemata for HA we consider, expressing classical principles and taken from [1], are the followings.

▶ **$\mathbf{\Sigma^0_n}$-LLPO.** Lesser Limited Principle of Omniscience. For any parameter a

$$\forall x, x' \ (P(x, a) \ \vee \ Q(x', a)) \implies \forall x P(x, a) \ \vee \ \forall x Q(x, a). \ (P, \ Q \in \Sigma^0_{n-1})$$

It is a kind of law for prenex formulas and if we assume the Axiom of Choice it is equivalent to Weak König's Lemma for Σ^0_{n-1} trees. We postpone the discussion about this principle at the conclusions of the paper.

▶ **Pigeonhole Principle for $\mathbf{\Pi^0_n}$.** The Pigeonhole Principle states that given a partition of infinitely many natural numbers in two classes, then at least one of these classes has infinitely many elements. For any parameter a

$$\forall x \ \exists z \ [z \geq x \ \wedge \ (P(z, a) \vee Q(z, a))] \implies$$

$$\forall x \ \exists z \ [z \geq x \ \wedge \ P(z, a)] \ \vee \ \forall x \ \exists z \ [z \geq x \ \wedge \ Q(z, a)]. \ (P, Q \in \Pi^0_n)$$

▶ **$\mathbf{EM_n}$.** Excluded Middle for Σ^0_n formulas. For any parameter a

$$\exists x\ P(x,a)\ \vee\ \neg\exists x\ P(x,a).\ (P \in \Pi^0_{n-1})$$

Recall that $P^\perp$ denotes the dual of P for any prenex P. As shown in [1, corollary 2.9] the law of Excluded Middle for Σ^0_n formulas is equivalent in HA to

$$\exists x\ P(x,a)\ \vee\ \forall x\ P(x,a)^\perp.\ (P \in \Pi^0_{n-1})$$

In all our schemata we use parameters. The parameter a is necessary since we need to use in HA statements with a free variable a, like

$$\forall a\ (\forall x\ P(x,a)\ \vee\ \exists x\ \neg P(x,a))$$

in our proof.

3 Ramsey Theorem for pairs and recursive coloring implies the Limited Lesser Principle of Omniscience for Σ^0_3 formulas

In this section we prove $\mathrm{RT}^2_2(\Sigma^0_0) \implies \Sigma^0_3$-LLPO in HA. From now on, all proofs are done in Intuitionistic Arithmetic HA. By definition of disjunctive schema, we have to prove that for each P in Σ^0_3-LLPO, there exist a finite number of recursive families of recursive colorings $c_{a,0},\dots,c_{a,j-1}$ such that, fixed any $W_i(.,c_{a,i})$ and $B_i(.,c_{a,i})$, if we assume

$$\{\forall a(W_i(.,c_{a,i}) \text{ infinite and homogeneous } \vee B_i(.,c_{a,i}) \text{ infinite and homogeneous}) \mid i<j\}$$

then we deduce P.

We say that a sequence is stationary if it is constant from a certain point on. In our proof we need some conjunctive schemata provable in Classical Arithmetic: that, in every primitive recursive family of monotone and bounded above sequences $s:\mathbb{N}\to\mathbb{N}$, each sequence is stationary and that, in every primitive recursive family of recursive sequences $t:\mathbb{N}\to\mathbb{N}$ for which there are at most k values of x such that $t(x)\neq t(x+1)$, each sequence is stationary. In order to obtain these results in HA from $\mathrm{RT}^2_2(\Sigma^0_0)$ we need to prove the EM_1 schema first, as shown by the following lemma (proved in HA, as all lemmas for now on).

▶ **Lemma 1.** 1. $\mathrm{RT}^2_2(\Sigma^0_0)$ *implies* EM_1*;*
2. EM_1 *implies that, for any family* $F=\{s(n,\cdot)\mid n\in\mathbb{N}\}$ *of recursive monotone and bounded above sequences enumerated by a binary primitive recursive function* $s:\mathbb{N}\times\mathbb{N}\to\mathbb{N}$*, each sequence in* F *is stationary;*
3. EM_1 *implies that, for any family* $G=\{t(n,\cdot)\mid n\in\mathbb{N}\}$ *of recursive sequences enumerated by a binary primitive recursive function* $t:\mathbb{N}\times\mathbb{N}\to\mathbb{N}$ *for which there are at most* k *values of* x *such that* $t(n,x)\neq t(n,x+1)$*, each sequence in* G *is stationary.*

Proof. 1. $\mathrm{RT}^2_2(\Sigma^0_0)$ implies $\mathrm{RT}^1_2(\Sigma^0_0)$ that implies the infinite pigeonhole principle which implies EM_1.

a. For the first implication, given a coloring of the points $c_a:\mathbb{N}\to 2$ we consider a coloring of the edges

$$c^*_a:[\mathbb{N}]^2\to 2$$

that depends only on the smallest point of the edge, that is, for every $x<y$, $c^*_a(\{x,y\}):=c_a(x)$. The infinite homogeneous set for c^*_a, whose existence is guaranteed by $\mathrm{RT}^2_2(\Sigma^0_0)$, is such that it is homogeneous also for c_a. Then $\mathrm{RT}^2_2(\Sigma^0_0)$ implies $\mathrm{RT}^1_2(\Sigma^0_0)$.

b. The infinite pigeonhole principle can be stated as follows

$$\forall x\ \exists z\ [z \geq x\ \wedge\ (P(z,a) \vee Q(z,a))]\ \Longrightarrow$$

$$\forall x\ \exists z\ [z \geq x\ \wedge\ P(z,a)]\ \vee\ \forall x\ \exists z\ [z \geq x\ \wedge\ Q(z,a)]\,,$$

with P and Q recursive predicates. Assuming the hypothesis of the principle, we define the following recursive coloring $c_a : \mathbb{N} \to 2$: for each $x \in \mathbb{N}$ $c_a(x) := 0$ if and only if the first witness z_x of

$$\exists z\ [z \geq x\ \wedge\ (P(z,a) \vee Q(z,a))]$$

is such that $P(z_x, a)$ is true. Thanks to $\mathrm{RT}^1_2(\Sigma^0_0)$ we have an infinite homogeneous set. If it is uniform in color 0 then P is true for infinitely many z, otherwise Q is true for infinitely many z.

c. By hypothesis we have the pigeonhole principle:

$$\forall x\ \exists z\ [z \geq x\ \wedge\ (P(z,a) \vee Q(z,a))]\ \Longrightarrow$$

$$\forall x\ \exists z\ [z \geq x\ \wedge\ P(z,a)]\ \vee\ \forall x\ \exists z\ [z \geq x\ \wedge\ Q(z,a)]\,,$$

with P and Q recursive predicates. We want to show that

$$\exists x\ P(x,a)\ \vee\ \forall x\ \neg P(x,a)\,;$$

with P recursive predicate. To prove it, we apply the pigeonhole principle with

$$P^*(z,a) := \exists y \leq z\ P(y,a)$$

$$Q^*(z,a) := \forall y \leq z\ \neg P(y,a)\,.$$

The hypothesis of the pigeonhole principle holds for P^*, Q^* with $z = x$. For the same principle, we deduce that either

$$\forall x\ \exists z\ [z \geq x\ \wedge\ \exists y \leq z\ P(y,a)]$$

is true, from which it follows $\exists x\ P(x,a)$, or

$$\forall x\ \exists z\ [z \geq x\ \wedge\ \forall y \leq z\ \neg P(y,a)]$$

is true, from which it follows $\forall x\ \neg P(x,a)$.

2. Suppose that $n \in \mathbb{N}$ and $s(n,\cdot) \in F$. We assume that s is recursive and there is an $r \in \mathbb{N}$ such that for every $x,\ y \in \mathbb{N}$

$$x \leq y\ \Longrightarrow\ s(n,x) \leq s(n,y) \leq r\,.$$

We prove that there exists m such that for every $y \geq m$ we have $s(n,m) = s(n,y)$. The proof is by induction on r. If $r = 0$ then $s(n,x) = 0$ for each x, hence we choose $m = 0$. Supposing the thesis holds for r, we prove the thesis for $r+1$ using EM_1. For EM_1, either there is m such that $s(n,m) = r+1$, or not. In the first case by monotonicity we have that for every x, $m \leq x$ implies $r+1 = s(n,m) \leq s(n,x) \leq r+1$, then $s(n,x) = r+1$ for every $x \geq m$. In the second case, we have $s(n,x) \leq r$ for each $x \in \mathbb{N}$, we apply the induction hypothesis and deduce the thesis. We need to use only one statement of EM_1

$$\forall n \forall r\ (\exists m(s(n,m) = r+1)\ \vee\ \forall m(s(n,m) \neq r+1))\,,$$

which implies all the formulas in EM_1 used in the proof.

3. Let $s(n,x)$ be the number of $y < x$ such that $t(n,y) \neq t(n,y+1)$. $s(n,\cdot)$ is monotone by construction. Since the number of changes of value of $t(n,\cdot)$ is bounded by some $r \in \mathbb{N}$ then $s(n,\cdot)$ is bounded by the same r. Moreover $\{s(n,\cdot) \mid n \in \mathbb{N}\}$ is enumerated by a primitive recursive function, since G has this property. So $s(n,\cdot)$ is stationary from a certain m onwards thanks to the second part of this Lemma. From the same point m even $t(n,\cdot)$ is stationary.

◀

We may now prove the main result of this section:

▶ **Theorem 2.** $\mathrm{RT}^2_2(\Sigma^0_0)$ *implies* Σ^0_3-LLPO.

Proof. Let a be a parameter, we assume the hypothesis of Σ^0_3-LLPO:

$$\forall x,\ x'\ (H_0(x,a) \vee H_1(x',a))\,,$$

where

$$H_0(x,a) := \exists y\ \forall z\ P_0(x,y,z,a)$$

$$H_1(x,a) := \exists y\ \forall z\ P_1(x,y,z,a)$$

for some $P_0,\ P_1$ primitive recursive predicates. In order to prove

$$\forall x\ H_0(x,a) \vee \forall x\ H_1(x,a)$$

we define a recursive 2-coloring such that:

- if there are infinitely many white (0) edges from x, then

$$H_0(0,a) \wedge \cdots \wedge H_0(x,a)\,;$$

- if there are infinitely many black (1) edges from x, then

$$H_1(0,a) \wedge \cdots \wedge H_1(x,a)\,.$$

Given x and m, where $m > x$, the color of $\{x,m\}$ expresses a conjecture based on a limited study of the predicates $H_i(x,a)$. White represents the hypothesis that $H_0(0,a) \wedge \cdots \wedge H_0(x,a)$ is true, after the analysis of the statements $H_0(0,a),\ \ldots,\ H_0(x,a)$ with quantifiers restricted to the set $[0,m]$. Vice versa, black represents the hypothesis that $H_1(0,a) \wedge \cdots \wedge H_1(x,a)$ is true, after the analysis of the statements $H_1(0,a),\ \ldots,\ H_1(x,a)$ with quantifiers restricted to the set $[0,m]$.

The coloring, and so the current hypothesis, is defined as follows. For every $n \in \mathbb{N}$ we define a primitive recursive function

$$y^a_n(m,c) : \mathbb{N} \times 2 \to m+1$$

that returns the minimum $y \leq m+1$ such that

$$\forall z \leq m\ P_c(n,y,z,a)\,,$$

if such y exists. If such y does not exist then $y^a_n(m,c) = m$.

Note that $y^a_n(m,c)$ is weakly increasing: if $y^a_n(m+1,c) \leq m$, then by definition

$$\forall z \leq m+1\ P_c(n, y^a_n(m+1,c), z, a)\,,$$

thus trivially

$$\forall z \leq m\ P_c(n, y^a_n(m+1, c), z, a)$$

follows and hence by construction $y^a_n(m, c) \leq y^a_n(m+1, c) \leq m$; on the other hand if $y^a_n(m+1, c) = m+1$ we obtain $y^a_n(m+1, c) > m \geq y^a_n(m, c)$.

For all $x \in \mathbb{N}$ define a sequence $C_x : \mathbb{N} \to 2$, where, for all $m > x$, $C^a_x(m)$ will be the color of the edge $\{x, m\}$.

$C^a_x(m) = c$ expresses that, analysing the interval $[0, m]$, $H_c(0, a) \wedge \cdots \wedge H_c(x, a)$ is believed to be true. The definition of $C^a_x(m)$ is given by induction on m.

- $C^a_x(0) = 0$;
- if for all $n \leq x$ $y^a_n(m, C^a_x(m)) = y^a_n(m+1, C^a_x(m))$ then $C^a_x(m+1) = C^a_x(m)$, else $C^a_x(m+1) = 1 - C^a_x(m)$.

We paint the edge $\{x, m\}$ with color $C^a_x(m)$. Now we want to prove that for some m_0 and for all $m \geq m_0$, that $C^a_x(m)$ is stationary, that $y^a_n(m, c)$ is stationary for every $n \leq x$, and that $y = y^a_n(m, c)$ is a witness of

$$H_c(n, a) := \exists y\ \forall z P_c(n, y, z, a)\,.$$

As a matter of fact we supposed:

$$\forall n,\ n' \leq x(H_0(n, a) \vee H_1(n', a))\,.$$

Hence we can constructively prove that witnesses exist either for $H_0(0, a) \wedge \cdots \wedge H_0(x, a)$ or for $H_1(0, a) \wedge \cdots \wedge H_1(x, a)$, so there exist $d_1,\ d_2,\ \ldots,\ d_x$ such that either for all $n = 0, \ldots, x$

$$\forall z\ P_0(n, d_n, z, a)$$

or for $n = 0,\ \ldots,\ x$

$$\forall z\ P_1(n, d_n, z, a).$$

In the first case we have

$$y^a_0(m, 0) \leq d_0,\ \ldots,\ y^a_x(m, 0) \leq d_x$$

for each m, so, thanks to the first and the second part of Lemma 1, the recursive sequences $(y^a_0(m, 0),\ \ldots,\ y^a_x(m, 0))$ are stationary. In the other case we have

$$y^a_0(m, 1) \leq d_0,\ \ldots,\ y^a_x(m, 1) \leq d_x$$

for each m, so, as above, the recursive sequence $(y^a_0(m, 1),\ \ldots,\ y^a_x(m, 1))$ are stationary. Moreover the sequences $(y^a_0(m, c),\ \ldots,\ y^a_x(m, c))_{m \in \mathbb{N}}$ with $c < 2$ increase in at least one component every second change of color. Since one of these is stationary, from a point onwards there could be only one change of color, so the number of change of values of $C^a_x(m)$ is bounded above. Thanks to the first and the third part of Lemma 1 the sequence $C^a_x(m)$ is stationary, for each $x \in \mathbb{N}$.

Now we need to prove that if there exists m_0 such that for all $m \geq m_0$ $C^a_x(m) = c$, then $H_c(0, a) \wedge \cdots \wedge H_c(x, a)$. In this case, by definition of $y^a_n(\cdot, c)$, there exist $e_0,\ \ldots,\ e_x$ such that $y^a_n(m, c) = e_n$ for all $n = 0,\ \ldots,\ x$. It follows that

$$\forall z \leq m\ P_c(n, e_n, z, a)$$

for each $n \leq x$, $m \geq m_0$, hence

$$\forall z \; P_c(n, e_n, z, a)$$

for every $n \leq x$, and thus $H_c(n, a)$ for all $n \leq x$.

Applying $\mathrm{RT}^2_2(\Sigma^0_0)$, there exists an infinite homogeneous set X. Hence if X is homogeneous in color c, and $x \in X$, then by stationarity of $C^a_x(m)$ every edge $\{x, m\}$ is of color c, except for a finite number of cases. Thus $H_c(0, a) \wedge \cdots \wedge H_c(x, a)$ for each $x \in X$ and so for infinitely many x. We obtain

$$\forall x \; H_c(x, a)\,.$$

In order to obtain an implication between schemata, observe that only three finite sets of statements in $\mathrm{RT}^2_2(\Sigma^0_0)$ are required in the proof: the statement that corresponds to the coloring of the edges and finitely many statements which corresponds to the two uses of Lemma 1 in the previous page. ◀

4 The Limited Lesser Principle of Omniscience for Σ^0_3 formulas implies Ramsey Theorem for pairs and recursive coloring

In this section we modify Jockusch's proof of Ramsey Theorem [10] in order to obtain a proof in HA of Σ^0_3-LLPO $\implies$ $\mathrm{RT}^2_2(\Sigma^0_0)$. It is enough to prove that if $\{c_a \mid a \in \mathbb{N}\}$ is a recursive family of recursive colorings, a finite number of statement in Σ^0_3-LLPO imply that there are predicates $W(., c)$ and $B(., c)$ such that,

$$\forall a(W(., c_a) \text{ infinite and homogeneous } \vee B(., c_a) \text{ infinite and homogeneous })\,.$$

We first sketch Jockusch's proof of RT^2_2 (which is itself a modification of Erdős Rado proof of RT^2_2): it consists in defining a suitable infinite binary tree J. We first remark that RT^1_2 (Ramsey Theorem for colors and points of $\mathbb{N}$) is nothing but the Pigeonhole Principle: indeed, if we have a partition of $\mathbb{N}$ into two colors, then one of the two classes is infinite. We informally prove now RT^2_2 from RT^1_2. Fix any coloring $f : [\mathbb{N}]^2 \to 2$ of all edges of the complete graph having support $\mathbb{N}$. If X is any subset of $\mathbb{N}$, we say that X defines a 1-coloring of X if for all $x \in X$, any two edges from x to some y, z in X have the same color. If X is infinite and defines a 1-coloring, then, by applying RT^1_2 to X we produce an infinite subset Y of X whose points all have the same color c, that is, such that all edges from all points of X all have the color c. Thus, a sufficient condition for RT^2_2 is the existence of an infinite set defining a 1-coloring. In fact we need even less. We say that a tree V included in the graph $\mathbb{N}$ defines a 1-coloring w.r.t. V if for all $x \in V$, for any two proper descendants y, z of x in V, the edges x to y, z have the same color. Assume there exists some infinite binary tree V defining a 1-coloring w.r.t. V. Then V has some infinite branch B by König's Lemma. B is a total order in V, therefore B is a complete subgraph of $\mathbb{N}$. Thus, B defines an infinite 1-coloring over the points of B, and proves RT^2_2. Therefore a sufficient condition for RT^2_2 is the existence of an infinite binary tree V defining a 1-coloring w.r.t. V. Erdős Rado proof, Jockusch's proof and our proof differ in the definition of V, even if the general idea is similar.

▶ **Theorem 3.** Σ^0_3-LLPO *implies* $\mathrm{RT}^2_2(\Sigma^0_0)$ *in* HA.

Proof. We consider Jockusch's version of Erdős Rado proof of RT^2_2 and we modify it in order to do not use classical principles stronger than Σ^0_3-LLPO. Erdős and Rado introduce an ordering relation $\prec_E$ on $\mathbb{N}$ which defines the proper ancestor relation of a binary tree E

structure on $\mathbb{N}$. The 2-coloring on edges of $\mathbb{N}$, restricted to the set of pairs $x \prec_E y$, gives the same color to any two edges $x \prec_E y$ and $x \prec_E z$ with the same origin x. This defines a canonical 1-coloring over the nodes of E. Jockusch defines a relativization $\prec_J$ to an infinite set J included in $\mathbb{N}$ of the relation $\prec_E$, that still defines a binary tree and a 1-coloring over the nodes of J. In both proofs, an infinite homogeneous set is obtained from an infinite set of nodes of the same color in an infinite branch of the tree. In Erdős-Rado and Jockusch's proofs, the pigeonhole principle is applied to a Δ^0_3-branch obtained by König's Lemma. To formalize this proof in HA we would have to use the classical principle Σ^0_4-LLPO. Our goal is to prove $\mathrm{RT}^2_2(\Sigma^0_0)$ using the weaker principle Σ^0_3-LLPO. We will define an infinite binary tree T with order relation $\prec_T$ such that T is Π^0_1 and has exactly one infinite branch, the rightmost. T is a variant of J such that we may prove that there are infinitely many nodes of the same color in the infinite branch using only Σ^0_3-LLPO. An infinite set totally ordered for $\prec_T$ and painted on the same color will be the monochromatic set for the original graph. Moreover our proof recursively defines two monochromatic Δ^0_3-sets, one of each color, that can not be both finite, even if we can not decide which of these is the infinite one.

Let V be a subset of $\mathbb{N}$ such that $0 \in V$. Firstly define, for each subset V of $\mathbb{N}$ such that $0 \in V$, a tree structure $\prec_V$ for V, then we choose a certain set for V. More precisely, we define a relation $x \prec_V y$ for each $x \in V$ and $y \in \mathbb{N}$, that restricted to $V \times V$ will define a tree with root 0. The definition of $x \prec_V y$ is given by induction on x: at each step we use only the subset $V \cap (x+1)$ of V.

- $0 \prec_V 1$.
- $x \prec_V y$ if and only if $x \in V$ and $y \in \mathbb{N}$ and $x < y$ and for every z such that $z \prec_V x$: $\{z, x\}$ and $\{z, y\}$ have the same color.

We define a tree T choosing an infinite sequence of points $x_0,\ x_1,\ \ldots$ of $\mathbb{N}$. The Jockusch relation $\prec_J$ restricted from $J \times \mathbb{N}$ to $J \times J$ in general is different from the Erdős Rado relation $\prec_E$ restricted from $\mathbb{N} \times \mathbb{N}$ to $J \times J$, but both relations have the same properties, which hold also for our relation $\prec_V$, no matter what is $V \subseteq \mathbb{N}$. Let us briefly state them.

▶ **Lemma 4.** *Let $V \subseteq \mathbb{N}$ be any predicate of* HA*, $0 \in V$, and $\prec_V$ defined as above.*

1. $\prec_V \subseteq <$.
2. $0 \prec_V x$ *for every* $x \in \mathbb{N} \setminus \{0\}$.
3. *If* $x,\ y \in \mathbb{N}$ *and* $V \cap (x+1) = U \cap (x+1)$ *then*

$$x \prec_V y \iff x \prec_U y\,.$$

4. $\prec_V$ *is transitive.*
5. *If* $x < y \prec_V z$ *and* $x \prec_V z$ *then* $x \prec_V y$.
6. *Let* $z \in \mathbb{N}$. *The relations* $<$ *and* $\prec_V$ *describe the same order on*

$$\mathrm{pd}_V(z) := \{x \in V \mid x \prec_V z\}\,,$$

i.e. for each $x, y \in \mathrm{pd}_V(z)$

$$x < y \iff x \prec_V y\,.$$

Proof. 1. It follows from the definition of $\prec_V$.

2. It follows from definition of $\prec_V$ and from the fact that does not exist a natural number z such that $z \prec_V 0$, since for the first point we should have $z < 0$.

3. Prove by induction on x. For $x = 0$ it follows from the second point. Suppose that it is true for each $z < x$. Prove $\Rightarrow$. Assume $x \prec_V y$, then by definition

$$x \in V \ \wedge \ y \in \mathbb{N} \ \wedge \ \forall z \prec_V x \ c_a(\{z, x\}) = c_a(\{z, y\}) .$$

By hypothesis it follows that $x \in U$, since

$$V \cap (x + 1) = U \cap (x + 1) ,$$

and thus, by induction hypothesis on $z < x$ and by $V \cap (z + 1) = U \cap (z + 1)$, we obtain

$$z \prec_V x \iff z \prec_U x ,$$

hence

$$x \in U \ \wedge \ y \in \mathbb{N} \ \wedge \ \forall z \prec_U x \ c_a(\{z, x\}) = c_a(\{z, y\}) ;$$

i.e. $x \prec_U y$. The proof of the vice versa is analogous.

4. $(x \prec_V y) \wedge (y \prec_V z) \implies x \prec_V z$.
By induction on z. For $z = 0$ it is true since $x, \ y \prec_V 0$ is false. Assume that the transitivity holds for all $z' < z$ and that

$$x \prec_V y \wedge y \prec_V z ,$$

then, by definition and by inductive hypothesis on $y < z$,

$$\forall w \prec_V x \ (w \prec_V y \ \wedge \ c_a(\{w, x\}) = c_a(\{w, y\}) = c_a(\{w, z\})) ,$$

we conclude $x \prec_V z$ by the definition of V.

5. By induction on x. If $x = 0$ it is trivial. Assume that it is true for each $t < x$ and we prove it for x. Observe that $x \in V$, $y \in V$ and $z \in \mathbb{N}$. Since $x \prec_V z$, we have that

$$\forall t \prec_V x \ c_a(\{t, x\}) = c_a(\{t, z\}) ,$$

and since $y \prec_V z$ we obtain

$$\forall t' \prec_V y \ c_a(\{t', y\}) = c_a(\{t', z\}) .$$

Since $x < y$, in order to prove $x \prec_V y$ it suffices to show that

$$\forall t \prec_V x (t \prec_V y) .$$

Let $t \prec_V x$, then $t \prec_V x \prec_V z$ and so, thanks to transitivity, we obtain $t \prec_V z$. Since we have $t < x < y \prec_V z$ and $t \prec_V z$, then $t \prec_V y$ by induction hypothesis. Therefore $x \prec_V y$.

6. ($\Leftarrow$) follows from the first property. ($\Rightarrow$). Let x, y be such that $x, y \prec_V z$ and $x < y$. Then, thanks to point 5 and since $x < y \prec_V z$ and $x \prec_V z$, we have $x \prec_V y$.

◀

By the sixth point of Lemma 4, the relation $\prec_V$ defines a total order on $\mathrm{pd}_V(z)$ for each $z \in V$; by the second point of Lemma 4 we have $0 \in \mathrm{pd}_V(z)$ if $z > 0$. Hence $\prec_V$ defines a tree with root 0 (we say that $\prec_V$ is the father/child relation).

It remains to choose a particular tree T definable by a predicate of HA, to use it in the proof of Ramsey Theorem. Define, by induction on n, the set of the first $n + 1$ nodes of T:

$$T_n := \{x_0, \ \ldots, \ x_n\} .$$

As auxiliary parameter we define a color c_n in $\{0,1\}$ as follows: if $n=0$ then $c_n=0$ and if $n>0$ then $c_n = c(\{\text{Father}(x_n), x_n\})$. The next edge added to T_n, if possible, should come from x_n and have color c_n. The proof of correctness of the definition of T requires the law of Excluded Middle of level 1, which is a consequence of Σ^0_3-LLPO (see [1]). T is a finite conjunction of decidable statements or simply universal statements and so it is Π^0_1.

The next node x_{n+1} of T is the first natural number z which satisfies the predicate we call "First Choice", or, if none exists, the first which satisfies the predicate we call "Second Choice".

- z is a first choice node after T_n if z is greater than x_n in the relation defined by T_n, and the edge from x_n to z has color c_n;

$$\text{FirstChoice}(z,T_n) := z \succ_{T_n} x_n \wedge c(\{z,x_n\}) = c_n\,.$$

  $\text{FirstChoice}(z,T_n)$ is decidable.
- z is a second choice node after T_n if z is the first node greater than some ancestor x_p of x_n in the relation defined by T_n, and for no proper descendant of x_p and ascendant of x_n there is such a z.

$$\begin{aligned}\text{SecondChoice}(z,T_n) := \exists p < n+1\{[z \succ_{T_n} x_p \wedge \forall y < z(y > x_n \Rightarrow y \not\succ_{T_n} x_p)]\\ \wedge\forall h \leq n[(h \geq p+1 \wedge x_h \succ_{T_n} x_p \wedge x_n \succcurlyeq_{T_n} x_h) \Rightarrow \forall w(w > x_n \Rightarrow w \not\succ_{T_n} x_h)]\}.\end{aligned}$$

  $\text{SecondChoice}(z,T_n)$ is Π^0_1.

Formally, z is the chosen node after T_n either if z is the minimal first choice node, or if there are not first choice nodes and z is the unique second choice node;

$$\begin{aligned}\text{Chosen}(\{z,T_n\}) :=&[\text{FirstChoice}(z,T_n) \wedge \forall y < z \neg\,\text{FirstChoice}(y,T_n)]\\ &\vee [\forall y \neg\,\text{FirstChoice}(y,T_n) \wedge \text{SecondChoice}(z,T_n)].\end{aligned}$$

$\text{Chosen}(z,T_n)$ is Π^0_1. We informally define the tree T, then we translate its definition in HA.

▶ **Definition 5** (Informal definition of T)**.** We informally define T_n by induction on n.

- If $n=0$ then $T_0 = x_0 := 0$.
- For $n+1$, if $\text{Chosen}(x_{n+1},T_n)$, then $T_{n+1} = T_n \cup \{x_{n+1}\}$.

  $T = \bigcup_{n\in\mathbb{N}} T_n\,.$

The definition 5 of T (which is not yet a definition in HA) uses EM_1, in other words an oracle for the properties Σ^0_1, hence T is a Δ^0_2 tree. We may represent in HA by some Π^0_1 predicates: "$x_0,\dots,x_n$ are the first n nodes of T" and $x \in T$.

▶ **Definition 6** (Formal definition of T)**.**

- "$x_0,\dots,x_n$ are the first n nodes of T" is the predicate of HA:

$$(x_0 = 0) \wedge \forall i < n\,\text{Chosen}(x_{i+1},\{x_0,\dots,x_i\})$$

- "x is a node of T" is the predicate of HA:

$$\text{Node}(x) := \exists n < x \exists x_0,\dots,x_n < x(\text{Chosen}(x,\{x_0,\dots,x_n\}) \wedge$$

  "$x_0,\dots,x_n$ are the first n nodes of T")$\,;$

Both predicates are Π^0_1. Now, we are going to prove that T of definition 6 satisfies the requirements of definition 5.

▶ **Lemma 7.** *If T is the tree defined by definition 5, every occurrence of the relation $\prec_{T_n}$ in* FirstChoice *and* SecondChoice *can be replaced by an occurrence of the relation $\prec_T$.*

Proof. Just see that the definition guarantees that for each n

$$T_n \cap (x+1) = T \cap (x+1)\,,$$

for each $x \in T_n$. Thus, applying the third point of Lemma 4, for every $x \in T_n$ and for every $y \in \mathbb{N}$

$$x \prec_{T_n} y \iff x \prec_T y.$$ ◀

The fact that T of definition 6 satisfies the requirements of definition 5 is a consequence of the uniqueness of the chosen node.

▶ **Lemma 8.** *For each n there exists a unique z such that* $\mathrm{Chosen}(z, T_n)$.

Proof. The uniqueness follows since we choose either the minimal first choice node, or, if it does not exist, the unique second choice node. The existence is a consequence of the EM_1 statement:

$$\forall z \neg\, \mathrm{FirstChoice}(z, T_n) \vee \exists z \mathrm{FirstChoice}(x, T_n)\,.$$

If there exists z which satisfies $\mathrm{FirstChoice}(z, T_n)$ then z is the chosen node, otherwise we prove that the second choice node exists. As a matter of fact, thanks to Σ^0_3-LLPO, EM_1 holds; and, by EM_1, we may prove in HA that either there is a first z such that $z \succ_T x_n$, a statement we may write as $\phi(x_n)$:

$$\phi(x) := \exists z((z \succ_T x) \wedge \forall y < z(y > x \implies y \not\succ_T x)$$

or for all z, $z \succ_T x_n$ is false, a statement we may write as $\psi(x_n)$, where:

$$\psi(x) := \forall z(z \not\succ_T x)\,.$$

Informally, if $\phi(x_n)$, i.e. if x_n has a first child z greater than x_n, we chose z. On the other hand, if $\psi(x_n)$, i.e., if x_n has no child z greater than x_n, we can decide if the father x_p of x_n has got a child greater than x_n or not, and so on. In the worst case we arrive at the root 0, which has at least the child $x_n + 1$, which is $> x_n$.

Formally, we have to prove the following formula:

$$\exists x \leq x_n(\forall y \leq x_n((y > x \wedge y \prec_{T_n} x_n) \implies \psi(y)) \wedge (x \preceq_{T_n} x_n) \wedge \phi(x));$$

which follows by the maximalization principle applied to the list $0 = x_{n_0}, \ldots, x_{n_p} = x_n$ of ancestors of x_n, and by $\phi(x_{n_0})$ and $\forall x.\phi(x) \vee \psi(x)$. ◀

Observe that the construction of the tree required one instance of two formulas of the EM_1 schema with different parameters. Each formula in EM_1 used in the proof above of Lemma 8 is an instance of one of the following formulas:

$$\forall n \forall \langle x_0, \ldots, x_n, c_n\rangle(\forall x \neg\, \mathrm{FirstChoice}(x, T_n) \vee \exists x \mathrm{FirstChoice}(x, T_n))\,,$$

and

$$\forall x(\exists z((z \succ_T x) \wedge \forall y < z(y > x \implies y \not\succ_T x) \vee \forall z(z \not\succ_T x)\,.$$

So only two statements of Σ^0_3-LLPO (the ones that imply the above formulas in EM_1) are sufficient in order to prove the existence of the tree.

Let r_n the branch in T_n that ends with x_n.

$$r_n = \{x_{i_0}, \dots, x_{i_m}\},$$

where $x_{i_0} = 0$ and $x_{i_m} = x_n$. We describe how r_n grows. If the z which satisfies $\mathrm{Chosen}(z, T_n)$ is such that $\mathrm{FirstChoice}(z, T_n)$ then $r_{n+1} = r_n \cup z$, while if it satisfies $\mathrm{SecondChoice}(z, T_n)$ then there exists $x_p \in T_n$ such that $z \succ_{T_n} x_p$ moreover for every $y > x_n$ and for each $h > p$ such that x_h is in r_n between x_{p+1} and x_n, $y \succ_{T_n} x_h$ does not hold. Observe that since $x_n \not\succ_{T_n} x_p$, we have $x_p \in r_n$. From this characterization of r_n we deduce:

▶ **Lemma 9.** *Let T be the tree defined above, and $x,\ y,\ z \in \mathbb{N}$.*

1. *All nodes of T having descendants after x_n are in r_n: if $x_i \in T_n$, $z > x_n$, and $z \succ_{T_n} x_i$, then $x_i \in r_n$.*
2. *If $x \in T$ has two children $y,\ z \in T$, with $y < z$ then y has no descendants in T which are $> z$.*

Proof. **1.** We prove the statement for all z, i by induction on n. If $n = 0$ it is trivial. Now suppose that the thesis is true for n and prove it for $n + 1$. Let r_{n+1} be the branch of T_{n+1} that ends with x_{n+1}. We have to check that for each $x_k \in T_{n+1} \setminus r_{n+1}$, there are no $y \succ_T x_k$ such that $y > x_{n+1}$. By definition of T, we have $r_{n+1} \cap x_n = \{x_{i_0}, \dots, x_{i_q}\}$, where x_{i_q} is the x_p of the predicate SecondChoice. Thus, if $x_k \in T_{n+1} \setminus r_{n+1}$, there are two possibilities left: either $x_k \in \{x_{i_{q+1}}, \dots, x_{i_m}\}$, or $x_k \in T_n \setminus r_n$. In the first case, by the choice of x_p there is not any $y > x_n$ such that

$$y \succ_T x_{i_m} \ \vee \ \dots \ \vee \ y \succ_T x_{i_{q+1}}.$$

Even more so, there is not any $y > x_{n+1} > x_n$ such that

$$y \succ_T x_{i_m} \ \vee \ \dots \ \vee \ y \succ_T x_{i_{q+1}}.$$

In the second case, by induction hypothesis, for every $x_k \in T_n \setminus r_n$ there do not exist any $y \succ_T x_k$ for which $y > x_n$, hence there are not any $y \succ_T x_k$ for which $y > x_{n+1} > x_n$.

2. Assume $z = x_{n+1}$ is the node chosen by some $T_n = \{x_0,\ \dots,\ x_n\}$. x has a child $y < z$ in T, therefore some child $y \in T_n$, hence $x \neq x_n$ because x_n is a leaf in T_n. z is a child of x in T, therefore, by definition of Chosen, z is a second choice node with $x_p = x$ for some $p < n$. By definition of $\mathrm{SecondChoice}(z, T_n)$ we have

$$y \succ_T x \wedge x_n \succ_T y \Rightarrow \forall w (w > x_n \Rightarrow w \not\succ_T y).$$

Since $z > x_n$ we obtain

$$\forall w (w > z \Rightarrow w \not\succ_T y).$$ ◀

Moreover we need to prove that the tree T is a binary tree: each node has at most two children.

▶ **Lemma 10.** *Let T be the predicate from definition 6, defining a tree.*

1. *The following is a sufficient condition for $x \prec_T y$. If $i,\ x \in T$ and $y \in \mathbb{N}$ are such that x is an immediate successor of i with respect to the relation $\prec_T$, $i \prec_T y$, $x < y$ and $c_a(\{i, x\}) = c_a(\{i, y\})$, then $x \prec_T y$.*
2. *Each node i of T has at most one child x such that $\{i, x\}$ is black, and at most one child y such that $\{i, y\}$ is white.*

Proof. 1. By hypothesis we have that

$$\forall t \prec_T i \; c_a(\{t,i\}) = c_a(\{t,x\})$$

and

$$\forall t \prec_T i \; c_a(\{t,i\}) = c_a(\{t,y\}),$$

so we have

$$\forall t \prec_T i \; c_a(\{t,x\}) = c_a(\{t,y\}). \tag{1}$$

Since x is an immediate successor of i,

$$t \prec_T x \iff t \prec_T i \vee t = i$$

by formula 1 and by the hypothesis $c_a(\{i,x\}) = c_a(\{i,y\})$, we obtain the thesis $x \prec_T y$.

2. Let $i \in T$ and let x and y be two children of i. Then we have that $x \prec_T y$ and $y \prec_T x$ are false, otherwise we should have $i \prec_T x \prec_T y$ and $i \prec_T y \prec_T x$. By point 1 above, since $x < y$ or $y < x$, it follows that $c(\{i,x\}) \neq c(\{i,y\})$. Therefore the number of children must be lesser than the number of colors, i.e. two. ◀

The tree T is infinite by construction and is binary by Lemma 10.2. We are going to prove, using EM_2 (that is a consequence of Σ^0_3-LLPO, [1]), that each node with infinitely many descendants has at least one child with infinitely many descendants, then that each node with infinitely many descendants has exactly one child with infinitely many descendants. This implies that T has exactly one infinite branch, which, to be accurate, is the rightmost branch of T, if we order children according to their integer value.

Observe that, by the definition of the tree, we have that, given a node t with infinitely many descendants, his first child has infinitely many descendants if and only if the first child is also the unique child (see Lemma 9.2). We define the uniqueness of the children of x as follows:

$$\mathrm{Unique}(x) := \forall x \forall z((\mathrm{Child}(x,t) \wedge \mathrm{Child}(z,t)) \implies x = z),$$

where

$$\mathrm{Child}(x,t) := \exists n < t \exists x_0, \ldots, x_n < t$$

("$x_0, \ldots, x_n, t, x$ are the first $n+2$ nodes of T").

This is an assertion Π^0_2, since Child is Π^0_1. Indeed, using EM_1, we can transform the occurrence of $\mathrm{Child}(x,t)$ in $\mathrm{Unique}(x)$ in a Σ^0_1 formula and the whole predicate $\mathrm{Unique}(x)$ in a Π^0_2 formula. If we apply EM_2 to $\mathrm{Unique}(x)$ we deduce that either that t has at most one child, or there exist two different children x and z of t. In the first case the first node x_{n+1} chosen after $t = x_n$ in T is a child of t, otherwise, by definition of T_{n+1}, t would not belong to the rightmost branch r_{n+1} of T_{n+1}, and by Lemma 9.1, t would not have descendants. So the node x is the unique child of t, and the infinitely many descendants of t are descendants of x. In the second case if $x < z$ are two children of t then z is the second child of t. Since we proved that a node has at most two children and by the definition of T, every descendant of t grater of z is descendant also of z, otherwise from a point onward t would not have descendants. Hence the second child of t, z, has infinitely many descendants. Observe that only one statement of Σ^0_3-LLPO is sufficient in order to prove that "t has only one child or

not" for every $t \in T$; as a matter of fact we need the formula in Σ^0_3-LLPO that implies the following formula in EM_2

$$\forall t(\forall x \forall z((\mathrm{Child}(x,t) \wedge \mathrm{Child}(z,t)) \implies x = z) \vee$$

$$\neg(\forall x \forall z((\mathrm{Child}(x,t) \wedge \mathrm{Child}(z,t)) \implies x = z))) .$$

We prove now that the infinite branch exists, is unique and define two monochromatic sets, where at least one is infinite. Now define r as follows; we say that $x \in r$ if and only if

$$\mathrm{InfiniteBranch}(x) \iff \forall y > x(\mathrm{Node}(y) \Rightarrow x \prec_T y).$$

▶ **Lemma 11.** *Let T be the tree defined above.*

1. *T has a unique infinite branch, r, the rightmost branch, which consists of all and only the nodes with infinitely many descendants.*
2. *If T has infinitely many edges with color c, then r has infinitely many edges with color c.*

Proof. 1. Thanks to the second part of Lemma 9, if a node has two children the first child has not got descendants greater than the second one, and therefore each node of T has at most one immediate infinite subtree. Since we have just proved the existence of the infinite subtree, it follows that each node of T that has infinitely many descendants is a root of a infinite subtree that has exactly one infinite subtree. Then the set of nodes with infinite children in T, which includes the root because T is infinite, has exactly one child for each node, and then defines the only infinite branch r of T.

2. Let $r = \{x_{i_0}, \ldots, x_{i_n}, \ldots\}$ be the unique infinite branch of T. Suppose that T has infinitely many edges of color c and prove that r has infinitely many edges of color c. Consider any node x_{i_p} of r, we want to prove that r has an edge of color c below x_{i_p}. If $\{x_{i_p}, x_{i_{p+1}}\}$ has color c we are done. Suppose it has color $1-c$: then $c_{i_{p+1}} = 1 - c$. By hypothesis, there exists n such that $n \geq i_{p+1}$ and there exists $m < n$ such that $\{x_m, x_n\}$ has color c. Since r is infinite, there exists q such that $i_q \geq n + 1 > n \geq i_{p+1}$. We prove that at least one of the edges

   $$\{x_{i_{p+1}}, x_{i_{p+2}}\}, \ \ldots, \ \{x_{i_{q-1}}, x_{i_q}\}$$

   has color c. Suppose by contradiction that they all have color $1-c$ (we are using Excluded Middle over a decidable statement about the colors of finitely many edges). In this case, for every $k \in [p+1, q-1]$ there exists $y > x_{i_{k+1}-1} \geq x_{i_k}$ such that $y \succ_T x_{i_k}$ and $\{y, x_{i_k}\}$ has color $1-c$, since $\{x_{i_k}, x_{i_{k+1}}\}$ has color $1-c$; so there exists a first choice node. Since for each such k there is a first choice node (with color $1-c$), it follows that between i_{p+1} and i_q the tree T grows keeping $c_{i_k} = 1 - c$ and only along the branch r. So we do not add the edge $\{x_m, x_n\}$ of color c between i_{p+1} e i_q, contradiction. ◀

We have still to prove that, indeed, the infinite branch of T has infinitely many pairs $x \prec_T y$ of color c. By Lemma 11.2, it is enough to prove that T has infinitely many pairs $x \prec_T y$ of color c, for some c. $\prec_T$ is a Π^0_1 predicate. Thus, if we apply the infinite pigeonhole principle for Π^0_1 predicates, we deduce that T either has infinite white edges, or has infinitely many black edges. However, the pigeonhole principle for Π^0_1 predicates is a classical principle, therefore we have to derive the particular instance we use from Σ^0_3-LLPO.

▶ **Lemma 12.** Σ^0_3-LLPO *implies the infinite pigeonhole principle for Π^0_1 predicates.*

Proof Lemma 12. The infinite pigeonhole principle for Π^0_1 predicates can be stated as follows:

$$\forall x\ \exists z\ [z \geq x\ \wedge\ (P(z,a) \vee Q(z,a))]$$

$$\implies \forall x\ \exists z\ [z \geq x\ \wedge\ P(z,a)]\ \vee\ \forall x\ \exists z[z \geq x \wedge Q(z,a)]\,,$$

with P and Q Π^0_1 predicates. We prove that the formula above is equivalent in HA to some formula of Σ^0_3-LLPO. Let

$$H(x,a) := \exists z\ [z \geq x \wedge P(z,a)]$$
$$K(x,a) := \exists z\ [z \geq x \wedge Q(z,a)].$$

In fact both H and K are equivalent in HA to Σ^0_2 formulas H', K'. By intuitionistic prenex properties (see [1])

$$\exists z[z \geq x\ \wedge\ (P(z,a) \vee Q(z,a))]$$

is equivalent to

$$\exists z[z \geq x\ \wedge\ P(z,a)]\ \vee\ \exists z[z \geq x\ \wedge\ Q(z,a)]\,.$$

The formula above is equivalent to $H' \vee K'$. Thus, any formula of pigeonhole principle for Π^0_1 with H, K is equivalent to the instance of Σ^0_3-LLPO with H', K'. ◀

Thus, there exist infinitely many edges of r in color c. Their smaller nodes define a monochromatic set for the original graph, since given an infinite branch r and $x \in r$, if there exists $y \in r$ such that $x \prec_T y$ and $\{x,y\}$ has color c, then for every $z \in r$ such that $x \prec_T z$, the edge $\{x,z\}$ has color c. Thus we can devise a coloring on r, given color c to x if $\{x,y\}$ has color c, with y child of x in r. After that, every infinite set of points with the same color in r defines an infinite set with all edges of the same color, and then it proves Ramsey Theorem in HA starting from the assumption of Σ^0_3-LLPO. ◀

Observe that the infinite branch r is Π^0_2. Moreover r can not be Δ^0_2. Here we prove it classically for short. Suppose by contradiction that r is Δ^0_2. In this hypothesis we will prove that for each recursive coloring there exists an infinite homogeneous set Δ^0_2. Indeed, using the fact that all edges from the same point of r to another point of r have the same color, we may describe the homogeneous set of color $c = 0, 1$ as the set of points whose edges to any other point of r all have color c:

$$\text{HomSet}(y) \iff y \in r \wedge \forall z > y(\text{InfiniteBranch}(z) \implies c(\{y,z\}) = c)$$

and also as the set of points having some edge to another point of r of color c:

$$\text{HomSet}(y) \iff y \in r \wedge \exists z > y(\text{InfiniteBranch}(z) \wedge c(\{y,z\}) = c)\,.$$

Therefore, if r is Δ^0_2 then the first formula is Π^0_2 and the second one is Σ^0_2. So for any $c = 0, 1$ the homogeneous set is Δ^0_2. Since at least one of these sets is infinite and since Jockusch proved that exists a coloring of $[\mathbb{N}]^2$ that has no infinite homogeneous set Σ^0_2, we obtain a contradiction. So $r \notin \Delta^0_2$ in general.

In Jockusch's proof he shows that one of the homogeneous sets (the red one in his notation) is Π^0_2, since at the beginning of each step he looks for red edges; while the second one is Δ^0_3. In our proof we can see that both the homogeneous sets are Δ^0_3, since our construction is

symmetric with respect to the two colors. As a matter of fact, since r is Π^0_2, the previous two formulas are respectively Π^0_3 and Σ^0_3. This is enough in order to prove that both the homogeneous sets are Δ^0_3. There always is an infinite homogeneous set Π^0_2, but apparently the proof is purely classical and cannot compute the integer code of such Π^0_2 predicate. Again we refer to Jockusch [10] for details.

5 Conclusions

Σ^0_3-LLPO is a principle of uncommon use, but it is equivalent to König's Lemma, given function variables and choice axiom [1]. The first goal of this section is to present the equivalence between Σ^0_3-LLPO and two more common principles: EM_2 and $\mathrm{DeMorgan}(\Sigma^0_3)$. After that we present some possible future developments.

First of all we want to prove that Σ^0_n-LLPO is equivalent to the union of $\mathrm{DeMorgan}(\Sigma^0_n)$ and EM_{n-1}, where

$$\mathrm{DeMorgan}(\Sigma^0_n) := \neg(P \wedge Q) \implies \neg P \vee \neg Q.\ (P, Q \in \Sigma^0_n)$$

$\mathrm{DeMorgan}(\Sigma^0_3)$ is a principle outside the hierarchy considered in [1] and incomparable with EM_1.

In order to prove the equivalence claimed above we need the following statements; their proof are shown in [1].

▶ **Lemma 13.** *Let* Σ^0_n-LLPO$^* := \neg(P \wedge Q) \implies P^\perp \vee Q^\perp$ *where* $P, Q \in \Sigma^0_n$*, then:*

1. Σ^0_n-LLPO *is equivalent to* Σ^0_n-LLPO**;*
2. Σ^0_n-LLPO *implies* EM_{n-1}.

Now, we can prove the equivalence. This equivalence helps us to analyse the proof of Theorem 3. Observing it, we can see that the most of the proof uses only EM_2 and that $\mathrm{DeMorgan}(\Sigma^0_3)$ (and so Σ^0_3-LLPO) is used only in the last part (Lemma 12).

▶ **Theorem 14.** Σ^0_n-LLPO $\iff \mathrm{DeMorgan}(\Sigma^0_n) + \mathrm{EM}_{n-1}$.

Proof. Denote with P, Q any two Σ^0_3 formulas.

$\Rightarrow$. Thanks to Lemma 13 we have Σ^0_n-LLPO $\implies \mathrm{EM}_{n-1}$. We have to prove $\mathrm{DeMorgan}(\Sigma^0_n)$. By the first part of Lemma 13, it suffices to prove that Σ^0_n-LLPO* implies $\mathrm{DeMorgan}(\Sigma^0_n)$. In HA holds $P^\perp \implies \neg P$, so we obtain

$$\neg(P \wedge Q) \implies P^\perp \vee Q^\perp \implies \neg P \vee \neg Q\,.$$

$\Leftarrow$. Thanks to De Morgan we have $\neg(P \wedge Q) \implies \neg P \vee \neg Q$. Moreover, by EM_{n-1}, we obtain $\neg P \implies P^\perp$ [1, corollary 2.9]. So, it follows Σ^0_n-LLPO* (that is equivalent to Σ^0_n-LLPO):

$$\neg(P \wedge Q) \implies P^\perp \vee Q^\perp\,.$$ ◀

The first question that raises after this work is what is the minimal classical principle that implies $\mathrm{RT}^2_2(\Sigma^0_n)$, Ramsey Theorem for pairs in two colors, but with any Σ^0_n family of colorings. We conjecture that, modifying conveniently the proofs of Theorem 2 and Theorem 3, we should obtain

$$\Sigma^0_{n+3}\text{-LLPO} \iff \mathrm{RT}^2_2(\Sigma^0_n). \tag{2}$$

A first development of this paper might be to check of the equivalence 2, for each $n \in \mathbb{N}$.

We conjecture that the result $\mathrm{RT}^2_2(\Sigma^0_0)$ may be generalized from 2 colors to any finite number of colors, that is, to the theorem $\mathrm{RT}^2_n(\Sigma^0_0)$, for any $n \in \mathbb{N}$. Apparently, however, the proof of Theorem 3 requires non-trivial changes in the case of n colors.

In this paper we consider Ramsey Theorem as schema in order to work with first order statements. Now our idea is to study Ramsey Theorem working in HA + functions + description axiom (that is a conservative extension of HA, see [1]), in order to use only one statement to express Ramsey Theorem for pairs in two colors. It seems to us that this unique statement is still equivalent to Σ^0_3-LLPO.

As we said in the introduction, in the future we hope to apply the interactive realizability [3] in order to study the computational content of Ramsey Theorem, and to find new constructive proofs for some consequences of it. Since the use of EM_n corresponds to n nested limits in this interpretation, thanks to our results, we may state that only three nested limits suffice to formalize this proof.

A further development would be to use this equivalence in order to find the minimal classical principles which imply a given corollary of Ramsey Theorem in HA.

Moreover we may observe that our proofs are semi-formal in HA, so it could be formalized using proof assistant software, like Coq.

Acknowledgements. We want to thank Alexander Kreuzer and Paulo Oliva for their useful comments and suggestions.

References

1 Yohji Akama, Stefano Berardi, Susumu Hayashi, and Ulrich Kohlenbach. An Arithmetical Hierarchy of the Law of Excluded Middle and Related Principles. In *LICS*, pages 192–201. IEEE Computer Society, 2004.

2 G. Bellin. Ramsey interpreted: a parametric version of Ramsey's Theorem. In AMS, editor, *Logic and Computation: Proceedings of a Symposium held at Carnegie-Mellon University*, volume 106, pages 17–37, 1990.

3 Stefano Berardi and Ugo de' Liguoro. A Calculus of Realizers for EM1 Arithmetic. In *Proceedings of CSL'08*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

4 Giovanni Birolo. *Interactive Realizability, Monads and Witness Extraction.* PhD thesis, Doctoral School of Sciences and Innovative Technologies, 2012. PhD Thesis, Università di Torino.

5 Peter A. Cholak, Carl G. Jockusch, and Theodore A. Slaman. On the strength of Ramsey's theorem for pairs. *Journal of Symbolic Logic*, pages 1–55, 2001.

6 C. T. Chong, Theodore A. Slaman, and Yue Yang. The Metamathematics of Stable Ramsey's Theorem for Pairs. *preprint*, 2012.

7 Thierry Coquand. A direct proof of Ramsey's Theorem. Author's website, 2011.

8 Denis R. Hirschfeldt. Slicing the Truth: On the Computability Theoretic and Reverse Mathematical Analysis of Combinatorial Principles. Author's webpage.

9 Hajime Ishihara. Reverse mathematics in Bishop's constructive mathematics. *Philosophia Scientiae, Cahier special*, 6:43–59, 2006.

10 Carl G. Jockusch Jr. Ramsey's Theorem and Recursion Theory. *J. Symb. Log.*, 37(2):268–280, 1972.

11 Ulrich Kohlenbach and Alexander Kreuzer. Term extraction and Ramsey's theorem for pairs. *J. Symb. Log.*, 77(3):853–895, 2012.

12 Jiayi Liu. ${\mathrm{RT}_2}^2$ does not imply WKL_0. *J. Symb. Log.*, 77(2):609–620, 2012.

13 Joseph R. Mileti. Partition theorems and computability theory. Technical Report 3, Department of Mathematics, University of Chicago, 2005.

14 Paulo Oliva and Thomas Powell. A Constructive Interpretation of Ramsey's Theorem via the Product of Selection Functions. *CoRR*, abs/1204.5631, 2012.

15 F. P. Ramsey. On a problem in formal logic. *Proc. London Math. Soc.*, 30:264–286, 1930.

16 Stephen George Simpson. *Subsystems of second order arithmetic*. Perspectives in logic. Association for symbolic logic, New York, 2009. 2nd edition.

17 Theodore A. Slaman. Σ_n-bounding and Δ_n-induction. *Proc. Amer. Math. Soc.*, 132:2449–2456, 2004.

Extracting Imperative Programs from Proofs: In-place Quicksort

Ulrich Berger, Monika Seisenberger, and Gregory J. M. Woods

Swansea University
Swansea, UK
U.Berger@swansea.ac.uk, M.Seisenberger@swansea.ac.uk, csgreg@swansea.ac.uk

Abstract

The process of program extraction is primarily associated with functional programs with less focus on imperative program extraction. In this paper we consider a standard problem for imperative programming: In-place Quicksort. We formalize a proof that every array of natural numbers can be sorted and apply a realizability interpretation to extract a program from the proof. Using monads we are able to exhibit the inherent imperative nature of the extracted program. We see this as a first step towards an automated extraction of imperative programs. The case study is carried out in the interactive proof assistant Minlog.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification, B.5.2 Design Aids, F.4.1 Mathematical Logic

Keywords and phrases program extraction, verification, realizability, imperative programs, in-place quicksort, computational monads, Minlog

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.84

1 Introduction

Program extraction based on the proofs-as-programs paradigm is a powerful method of generating, in one step, programs together with a proof of their correctness. Often, this technique is based on some form of realizability (see e.g. [15, 5]), or a similar method, and usually yields terms denoting computable functionals or elements of a partial combinatory algebra. These terms can be naturally interpreted as functional programs. For this reason, the process of program extraction has long been associated with functional programs. There exist many tools which are able to extract functional programs from proofs (see e.g. Coq [9, 17], Minlog [18, 4], Agda [2, 8], NuPRL [22], Isabelle [13, 6]) whereas relatively little research and tool development has been explored addressing the problem of extracting imperative programs from proofs. Since most programs that are written today are more towards the imperative paradigm (cf TIOBE Index [28]) and imperative programs, in general, are notoriously difficult to verify, it would be highly desirable to have tools that allow the extraction of verified imperative programs.

In this paper we show that imperative program extraction is possible. We start with a case study where we extract an imperative In-place Quicksort algorithm from a proof. First, we informally describe In-place Quicksort, then we present a formalisation of a proof that every array can be sorted. From this proof we extract a first version of Quicksort using the Minlog system. This version of Quicksort, which is still functional, is then translated into a program that uses the well-known state monad and can be directly interpreted as the desired imperative In-place Quicksort algorithm.

An analysis of the *program* obtained from this case study leads us to a restricted functional calculus, called **SIT** (**Si**ngle-**T**hreaded Functional Language), that singles out programs

which use their array argument in a single-threaded way. All well-typed **SIT** programs can be translated into well-typed programs of a monadic language, called **MON**, that directly admits an imperative interpretation. We prove the correctness of this translation by showing that it is the inverse of the natural interpretation of **MON** in **SIT**.

The next step will be to analyse the *proof* from which our Quicksort program was extracted, and develop a proof calculus which only extracts programs in the language **SIT**, or a suitable generalisation of **SIT**, for which a similar automatic translation into imperative code is possible. We leave this, as well as the actual translation of **MON** programs into imperative code, for further work.

1.1 Related work

There are other attempts at extracting imperative programs from proofs. A notable one is carried out in [23] where the authors combine the proofs-as-programs concept with Intuitionistic Hoare Logic. However, this work leans more towards a verification strategy, which builds on the specification of the program.

A different route is Krivine's Classical Realizability ([16]) which is based on an abstract machine model that has imperative features. Krivine's work has further been adapted by Miquel [19]. The precise links between these methods and ours have yet to be investigated.

Quicksort has traditionally been a case study for many verification techniques and as such, there are many examples in the literature. A recent case study of the In-place Quicksort algorithm was formed in [11] using Event-B[1]. The behaviour of the algorithm is specified and then the algorithm is verified through a series of "refinements" where the problem is simplified into abstract machines and the effect of the abstraction makes the verification tasks more simple.

Another interesting verification is undertaken in [26] using ACL2 (see [7]) with an efficient version of the In-place Quicksort algorithm using single-threaded objects. The verification task involves showing that the efficient version of Quicksort is equivalent to a non single-threaded version and then showing that this algorithm satisfies the properties of a sorting algorithm. Once the equivalence is established, the process of verification is in some ways similar to Event-B using a "refinement" process to obtain correctness.

In relation to our work, the Event-B case study has an extraction process, through refinement, of the imperative In-place Quicksort algorithm whereas the ACL2 case study shows correctness of an already written imperative program w.r.t. the behaviour of the original functional Quicksort algorithm. The crucial difference between these works and ours is that we synthesize programs from mathematical proofs that do not require particular representations of data, and do not involve the process of programming. Therefore, our approach is highly modular, language independent, and more accessible to users outside the programming community.

1.2 Minlog

Minlog [18, 3] is an interactive proof system based on a first-order natural deduction calculus. It is not a type-theoretic system, like Coq or Agda, since it keeps formulas and proofs separate from (non-dependent) types and terms. Terms and types have a simple domain-theoretic denotational semantics. The theoretical background of Minlog is explained in [27]. One of the main motivations behind Minlog is to exploit the proofs-as-programs paradigm for program development and program verification. Minlog implements various methods of program extraction (realizability, dialectica interpretation) which also include extraction

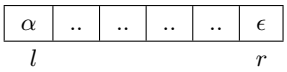
from classical proofs via the Friedman A-translation (see [25] for a comparative case study). Recent work on Minlog has been extending program extraction to simultaneous inductive and coinductive definitions and extraction to Haskell [20]. The system is supported by automatic proof search and normalization by evaluation as an efficient term rewriting device. Minlog is implemented in Scheme, and is an open system which invites users to contribute to its development and explore new methods.

2 Informal description of In-Place Quicksort

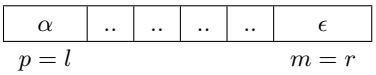
Quicksort, as we consider it in this paper, is a sorting algorithm that takes an array of natural numbers as input and sorts its elements into order of lowest to highest. The Quicksort algorithm was invented by Tony Hoare [12, p11]. Here we focus on an imperative variant of Quicksort, called In-place Quicksort, which sorts an array by repeatedly swapping elements, without using extra memory space (for creating new arrays).

In-place Quicksort works as follows:

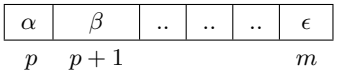
1. We are given an array a and indices l, r. We wish to sort a on the interval $[l, r] = \{i \in \mathbb{N} \mid l \leq i \leq r\}$:



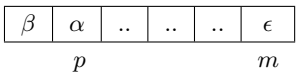
   If $l \geq r$, nothing needs to be done. Hence, in the following we assume that $l < r$.
2. Pick l as the pivot index p, and pick r as the max swap index m:



   We call $a[p]$, the element of the array at p, the *pivot*.
3. Now we compare the pivot with the element at $p + 1$:



   a. If the pivot is greater than or equal to the element at $p + 1$, then we swap the two elements of the array and increment the pivot index by 1:



   b. If the pivot is smaller than the element at $p + 1$, then we swap the element at $p + 1$ with the element at m and decrement the index m by 1:

4. Repeat step 3 until the markers p and m are equal.
5. Repeat steps 1–4 recursively on the array between l and $p - 1$ and on the array between $p + 1$ and r.

We will show that this imperative algorithm can be extracted automatically from a proof that contains no reference to imperative features at all in contrast to [23].

3 A formal proof that every array can be sorted

We sketch a formal proof that every array can be sorted. The proof is organized in such a way that its computational content corresponds to the previously described In-place Quicksort algorithm. We present the proof in some detail, so that the reader can use it as a guide through the Minlog proof script (see [24]).

For simplicity, we assume that the array to be sorted has as index set the whole set of natural numbers and every cell of the array holds a natural number (the array could hold any type of elements with a computable ordering). We view arrays as an abstract data type equipped with operations $a[i]$, for accessing the content of the array a at index i and $a[i := x]$ for changing the value of the array a at index i to x. Hence we assume the following axioms

(RW1) $b = a[i := x] \to b[i] = x\,.$
(RW2) $b = a[i := x] \wedge k \neq i \to b[k] = a[k]\,.$

Our goal is to prove that every array can be permuted into a sorted one between any given index bounds $l \leq r$

▶ **Theorem 1** (Sorting Theorem). $\forall l, r, a.\ l \leq r \to \exists b.\mathtt{Permuted}(a, b, l, r) \wedge \mathtt{Sorted}(b, l, r)$[1]

where

$$\mathtt{Sorted}(a, l, r) := \forall i, j.l \leq i < j \leq r \to a[i] \leq a[j]$$

and the intuitive meaning of the predicate `Permuted` is

$$\mathtt{Permuted}(a, b, l, r) := \exists \text{ permutation } \sigma \text{ of } [l, r].\ \forall i \in [l, r]\ b[i] = a[\sigma(i)] \quad \wedge \quad \forall i \notin [l, r].\ b[i] = a[i]$$

where $[l, r] = \{i \mid l \leq i \leq r\}$.

In the proof it will be convenient to work with a different (but equivalent) *inductive* definition of the predicate `Permuted`, based on the fact that every permutation is a composition of swappings (transpositions). First, we define what it means for two arrays a and b to differ only by swapping the contents at the indices i and j:

$$\mathtt{Swap}(a, b, i, j) := b[i] = a[j] \wedge b[j] = a[i] \wedge \forall k \notin \{i, j\}.\ b[k] = a[k]\,.$$

The predicate `Permuted` is now defined inductively by the clauses:

(P1) $\mathtt{Permuted}(a, a, l, r)\,.$
(P2) $l \leq i, j \leq r \wedge \mathtt{Swap}(a, b, i, j) \to \mathtt{Permuted}(a, b, l, r)\,.$
(P3) $\mathtt{Permuted}(a_1, a_2, l, r) \wedge \mathtt{Permuted}(a_2, a_3, l, r) \to \mathtt{Permuted}(a_1, a_3, l, r)\,.$

This means that `Permuted` is the least predicate satisfying the clauses P1–3.

We first prove we can always swap two elements of an array and that the predicate `Permuted` has the expected properties:

▶ **Lemma 2** (Swap). $\forall a, i, j.\ \exists b.\ \mathtt{Swap}(a, b, i, j)\,.$

[1] We use the dot notation where the dot allows the quantifiers to have the largest possible range. For example $\forall i.A \to B$ is the same as $\forall i(A \to B)$.

▶ **Lemma 3** (Permutation). *For all* a, b, l, r *with* $\texttt{Permuted}(a, b, l, r)$

(a) $\forall i \in [l, r].\exists j \in [l, r].\ b[i] = a[j]$.

(b) $\forall i \notin [l, r].\ b[i] = a[i]$.

(c) $\forall l', r'.\ l' \leq l \wedge r \leq r' \to \texttt{Permuted}(a, b, l', r')$.

The proofs of both lemmas are easy. The proof of the Permutation Lemma proceeds by induction on the definition of $\texttt{Permuted}(a, b, l, r)$. We omit further details.

3.1 Partitioning an array

The crucial idea of Quicksort is the notion of a *partition*. A partition is an array together with a selected pivot element such that every element to the left of the pivot is smaller than or equal to the pivot and every element to the right is greater than the pivot.

$$\begin{aligned}\texttt{Partition}(a, l, p, r) := \forall i\,(\,l \leq i < p \to a[i] \leq a[p]) \wedge \\ \forall i\,(\,p < i \leq r \to a[p] < a[i])\,.\end{aligned}$$

For the proof of the Sorting Theorem (Theorem 1) we need to prove that every array can be permuted into a partition:

▶ **Lemma 4** (Partition).

$$\forall l, r, a\ .\ l \leq r \to \exists p, b\ .\ l \leq p \leq r \wedge \texttt{Permuted}(a, b, l, r) \wedge \texttt{Partition}(b, l, p, r)\,.$$

Proof. We prove a slightly stronger statement stating in addition that $b[p] = a[l]$. The proof is by induction on $k := r - l$. Formally, we prove

$$\forall k, l, r, a\ .\ l + k = r \to \exists p \in [l, r], b\ .\ b[p] = a[l] \wedge \texttt{Permuted}(a, b, l, r) \wedge \texttt{Partition}(b, l, p, r)$$

by induction on k.

Base Case: $k = 0$. Assume $l + 0 = k$. Then $l = k$ and we may take $p = l$ and $b = a$.

Step: $k + 1$. We assume $l + (k + 1) = r$.

- Case 1: $a[l] \geq a[l + 1]$.
  We use the induction hypothesis with $l + 1$, r and some a_1 satisfying $\texttt{Swap}(a, a_1, l, l + 1)$, which by the Swapping Lemma exists. Since $(l + 1) + k = r$ we get $p \in [l + 1, r]$ and b with $b[p] = a_1[l + 1]$, $\texttt{Permuted}(a_1, b, l + 1, r)$ and $\texttt{Partition}(b, l + 1, p, r)$. We take the same p and b. We have $b[p] = a_1[l + 1] = a[l]$. Furthermore, by definition of the predicate Permuted, it follows that $\texttt{Permuted}(a, b, l, r)$. In order to show $\texttt{Partition}(b, l, p, r)$, we assume first $l \leq i < p$ and show $b[i] \leq b[p]$. If $i = l$, then $b[i] = a_1[l]$ by the Permutation Lemma, part (c), and hence $b[i] = a[l + 1] \leq a[l] = a_1[l + 1] = b[p]$. If $i \geq l + 1$, the in-equation $b[i] \leq b[p]$ follows from $\texttt{Partition}(b, l + 1, p, r)$. That for $p < i \leq r$ we have $b[p] < b[r]$ follows immediately from $\texttt{Partition}(b, l + 1, p, r)$.
- Case 2: $a[l] < a[l + 1]$.
  We use the induction hypothesis with l, $r - 1$ and a_1 satisfying $\texttt{Swap}(a, a_1, l + 1, r)$. Since $l + k = r - 1$ we get $p \in [l, r - 1]$ and b with $b[p] = a_1[l]$, $\texttt{Permuted}(a_1, b, l, r - 1)$ and $\texttt{Partition}(b, l, p, r - 1)$. Again, we take the same p and b. We have $b[p] = a_1[l] = a[l]$ since $l \notin \{l + 1, r\}$. Furthermore, by definition of the predicate Permuted, it follows $\texttt{Permuted}(a, b, l, r)$. In order to show $\texttt{Partition}(b, l, p, r)$, we assume first $p < i \leq r - 1$

and show $b[i] > b[p]$. If $i = r$, then $b[i] = a_1[r]$ by the Permutation Lemma, part (c), and hence $b[i] = a[l+1] > a[l] = b[p]$. If $i \leq r-1$, the in-equation $b[i] > b[p]$ follows from $\texttt{Partition}(b, l+1, p, r)$. That for $l \leq i < p$ we have $b[i] \leq b[p]$ follows immediately from $\texttt{Partition}(b, l, p, r-1)$.

This completes the proof of the Partition Lemma. ◀

3.2 Proof of the Sorting Theorem

Now we have everything we need to prove the Sorting Theorem (1). If $l \geq r$, the proof is trivial. Hence, in the following we assume $l < r$.

Proof. We do an induction on the length $r - l$ of the segment to be sorted. Therefore, we have an induction hypothesis for all shorter segments. Our goal is

$$\exists b.\ \texttt{Sorted}(b, l, r) \wedge \texttt{Permuted}(a, b, l, r)\,.$$

By the Partition Lemma (4) we have an array a_1 and a pivot index p that fulfil the partition property

$$\texttt{Permuted}(a, a_1, l, r) \wedge \texttt{Partition}(a_1, l, p, r)\,. \tag{1}$$

- Case 1: Both sides of the partition are non-empty, i.e. $l < p < r$.
  We use the induction hypothesis with the left side of the partition with the array $l, p-1$ and a_1 and get a_2 such that

  $$\texttt{Sorted}(a_2, l, p-1) \wedge \texttt{Permuted}(a_1, a_2, l, p-1)\,. \tag{2}$$

  We now do the same on the right side of the partition using the induction hypothesis with $p+1, r$ and a_2 and get b with:

  $$\texttt{Sorted}(b, p+1, r) \wedge \texttt{Permuted}(a_2, b, p+1, r)\,. \tag{3}$$

  We show that our b satisfies:

  $$\texttt{Sorted}(b, l, r) \wedge \texttt{Permuted}(a, b, l, r)\,.$$

  - $\texttt{Sorted}(b, l, r)$: by (2) we have $\texttt{Sorted}(a_2, l, p-1)$ and by (3), $\texttt{Sorted}(b, p+1, r)$. Furthermore, by the Permutation Lemma (3), part (b), and by using the other conjunct of (2) it follows $\texttt{Sorted}(b, l, p-1)$. Furthermore, from (1) and the Permutation Lemma (3), part (a), it follows $\texttt{Partition}(b, l, p, r)$. From these facts one easily derives $\texttt{Sorted}(b, l, r)$.
  - $\texttt{Permuted}(a, b, l, r)$: this follows easily from the Permutation Lemma, part (c), and the last clause of the inductive definition of the predicate $\texttt{Permuted}$.
- Case 2: One side of the partition is empty.
  If the left side is empty, i.e. $p = l$, then we apply the induction hypothesis to $a_1, l+1, r$. If the right side is empty we use the induction hypothesis for $a_1, l, r-1$. The rest of the proof is similar to the previous Case 1, but simpler. ◀

4 Program Extraction

In this section we address some technical issues of the implementation in Minlog, present the extracted programs and explain why they can be considered as imperative programs. The Minlog source files for this extraction example can be found on the Swansea Minlog Repository web page [24].

4.1 Implementation in Minlog

We highlight some aspects of the implementation in Minlog that are necessary to understand the proof scripts and the extracted programs.

We present the theorems as they appear in Minlog:

Listing 1 Sorting Theorem (1)

```
(set-goal (pf "all l,r,a . l<= r ->
                           ex b. Permuted a b l r & Sorted b l r"))
```

Listing 2 Partition Lemma (4)

```
(set-goal (pf "all k,l,r,a. l+k=r ->
            (ex a1,p. ((all i (l<=i ->  i<p -> Rd a1 i <= Rd a1 p)) &
                       (all i (p<i -> i<=r  ->  Rd a1 p <  Rd a1 i)) &
                       Permuted a a1 l r &
                       Rd a1 p = Rd a l &
                       l <= p &
                       p <= r))"))
```

Listing 3 Swap Lemma (2)

```
(set-goal (pf "all a,i,j ex b. Swap a b i j"))
```

4.1.1 Arrays

Minlog does not have a built-in data type of arrays. We chose to define arrays as a free algebra with the nullary constructor `Empty`, denoting the constant zero array, and a ternary constructor `Wr` (for "write"):

Listing 4 Minlog Array Definition

```
(add-alg "ar" '("Empty" "ar") '("Wr" "ar=>nat=>nat=>ar"))
```

The earlier used notation $a[i := x]$ was just syntactic sugar for $\texttt{Wr}\, a\, x\, i$.

By structural recursion on the free algebra of arrays we define a reading operation:

Listing 5 Minlog Read Definition

```
    (apc "Rd" (py "ar=>nat=>nat"))
    (add-computation-rule
      (pt "Rd (Wr a n i) j")
      (pt "[if (i=j) n (Rd a j)]"))
```

This means that `Rd` is introduced as a program constant with the computation rule rewriting `Rd (Wr a n i) j` to `[if (i=j) n (Rd a j)]`. We do not use a rewrite rule for `Rd Empty j` in the proof.

Using the notation $a[i]$ for $\texttt{Rd}\, a\, i$ the Axioms (RW1) and (RW2) (cf. Sect. 3) now become easily provable theorems.

4.1.2 Equational reasoning via normalisation

The reader might have noticed that we slightly deviated from Sect. 3 where we said that we consider arrays as an abstract data type. The reason for our choice is purely a matter of convenience since equational reasoning becomes much easier if certain equations are expressed via (strongly normalizing and confluent) term rewriting rules, allowing us to prove equations

by checking syntactic equality of normal forms. One could, of course, replace arrays by a different data structure where accessing an element takes logarithmic instead of linear time. However, this would have no effect on the extracted program.

4.1.3 Realizability

In this paper we work with Minlog's program extraction module based on modified realizability [15] which can be viewed as a typed version of Kleene's realizability for numbers [14].

Realizability provides a very intuitive way of extracting the computational content from a formal constructive proof in the spirit of the Curry-Howard correspondence that associates propositions with types and proofs with programs. It does not establish an isomorphism between proofs and programs, but a highly non-injective homomorphism that eliminates large parts of proofs that are computationally irrelevant. In order to widen the range of applications, Minlog uses an extended form of realizability that allows, for example, to extract algebraic data types from inductive definitions [18].

4.1.4 Induction and recursion

There occur two kinds of induction in our formalization:

1. Ordinary "Zero-successor"-induction on the natural numbers. This is used, for example in the proof of the Partition Lemma. In Minlog this axiom scheme is interpreted computationally (via realizability) as a constant-scheme: `Rec`.
2. The other form of induction we use is "induction with respect to a measure function". This is used in the Sorting Theorem. The realizability interpretation of this form of induction is a constant-scheme: `GRecGuard`.

4.2 The extracted programs

4.2.1 Program extracted from the proof of the Sorting Theorem.

Listing 6 Qsort Extracted Program

```
qsort =
  cGIND
  [n3,n4,g5,a6]
    [let ap7 (cPart(n4--n3)n3 n4 a6)
       [if (n3<right ap7)
           [if (right ap7<n4)
               [let a8 (g5 n3(Pred right ap7)left ap7)
                [let a9 (g5(Succ right ap7)n4 a8)
                 a9]]
               (g5 n3(Pred right ap7)left ap7)]
           [if (right ap7<n4)
               (g5(Succ right ap7)n4 left ap7)
               (left ap7)]]]
```

Note: `ap7` corresponds to (a_1, p) in the proof of the Partition Lemma. `left ap7` and `right ap7` correspond to p and a_1 respectively. `g5` plays the role of the recursive call, `cGIND` is a particular instance of the constant-scheme `GRecGuard` and `cPart` corresponds to the Partition lemma. The letter `c` in `cPart` indicates that it is an automatically generated name for the program from partition lemma `Part`. For better readability, we omit the `c` in the further discussion.

4.2.2 Program extracted from the proof of the Partition Lemma.

Listing 7 Part Extracted Program

```
part =
 [n0]
  (Rec nat=>nat=>nat=>ar=>ar@@nat)
   n0
  ([n4,n5,a6]a6@n4)
  ([n4,rec5,n6,n7,a8]
     [if (Rd a8(Succ n6)<=Rd a8 n6)
         [let a9 (Swap a8(Succ n6)n6) (rec5(Succ n6)n7 a9)]
         [let a9 (Swap a8(Succ n6)n7) (rec5 n6(Pred n7) a9)]])
```

Note: The variable `rec5` has type `nat=>nat=>ar=>ar@@nat` and represents the recursive call where `ar@@nat` is the pairing of an array and a natural number.

4.2.3 The swap function extracted from the proof of the Swapping Lemma

Listing 8 Swap Extracted Program

```
swap =
 [a0,n1,n2]
   [let n3 (Rd a0 n1)
     [let n4 (Rd a0 n2)
       [let a1 (Wr a0 n4 n1) (Wr a1 n3 n2)]]]
```

4.3 The extracted programs explained

Minlog's formalization of recursive definitions via recursion operators is adequate from a technical point of view, but it makes recursively defined functions hard to read. Therefore, we replace the recursion operators by recursive equations, and re-name the (automatically generated) variables so that they match the variable names used in the proofs. Furthermore, we write the variable `ap7` in the let expression of `qsort` as a pair `(a1,p)` which spares us the use of the projection functions `left` and `right`. Also, in the function `part` we omit the first parameter `n0`, which corresponds to k, since it always has the value $r - l$ (`n4-n5`), and replace, for example "$k = 0$" by $l = r$. All these changes are only cosmetic and intended to ease the understanding of the programs. They do not affect their behaviour.

We use Haskell-like syntax in the pseudo-code below. For example, `(N,N,A)` stands for the type $\mathbb{N} \times \mathbb{N} \times A$ where A is the type of arrays. The programs `qsort` and `part` are intended to be used for $l \leq r$ only.

Listing 9 Qsort (Haskell Style)

```
qsort(l,r,a) =
  let (a1,p) = part(l,r,a)
  in if l < p
     then if p < r
          then let {a2 = qsort(l,p-1,a1)} in qsort(p+1,r,a2)
          else qsort(l,r-1,a1)
     else if p < r
          then qsort(l+1,r,a1)
          else a1
```

Listing 10 Part (Haskell Style)

```
part : (N,N,A) -> (A,N)
part(l,r,a) =
  if l = r
  then (a,l)
  else if a[l+1] <= a[l]
       then let {a1 = swap(a,l+1,l)} in part(l+1,r,a1)
       else let {a1 = swap(a,l+1,r)} in part(l,r-1,a1)
```

Listing 11 Swap (Haskell Style)

```
swap : (A,N,N) -> A
swap(a,i,j) = let {x = a[i]; y = a[j]; a1 = a[i:=y]} in a1[j:=x]
```

4.4 How are these extracted programs imperative?

At first sight the extracted programs look clearly functional. Indeed they are, provided the writing operation `a[i:=y]`, which is the only operation where the array is modified, is implemented functionally. However, nothing prevents us from implementing the write operation as a procedure that destructively changes the array (instead of producing a new array). But, are the extracted programs then still correct? Looking at the proof of the Soundness Theorem (see eg [5]) for realizability, which is the source of correctness of our extracted program, we see that the extracted programs are assumed to behave functionally, i.e. have no side effects. In particular, functional programs do not destroy their input, and as a consequence, referential transparency holds, which means that the value of a complex programs only depends on the values of its subprograms and not, for example, on the order in which the subprograms are evaluated. Referential transparency is crucial for the Soundness Theorem, and it does, in general, not hold for imperative code. Therefore, an additional argument is needed in order to show that, in our particular case, the programs stay correct when the writing operation is implemented imperatively. The argument is simple: in the extracted programs, arrays can be viewed as "single-threaded" objects, since, once an array is used as an argument of the update operation, or a program that uses the writing operation such as `swap`, `part` or `qsort`, it is never used again. Hence the correctness of the program is not compromised if the write operation destroys its argument.

4.5 Monadic presentation of the extracted programs

The imperative nature of the programs `qsort` and `part` becomes particularly lucid when they are formulated using the *state monad* where arrays play the role of states. Monads [21] are a popular concept to incorporate imperative code into functional programs in an elegant and clean way. The definitions below are standard in functional programming, but in order to make the paper self-contained we include them.

We define a type operator `M` (the state monad with arrays as states) by

```
M u = A -> (u,A)
```

where `u` is a type variable. A value of type `M u` can be viewed as an *action* that, when executed, produces a result of type `u`, but may, in addition, have a side effect on the state (array in our case).

We have the general monad operators

```
return : u -> M u
return x a = (x,a)
```

```
bind : M u -> (u -> M v) -> M v
bind m f a = let {(x,b) = m a} in f x b
```

and the special operators for this particular monad:

```
get : N -> M N
get i a = (a[i],a)
```

```
put : (N,N) -> M ()
put(i,x) a = ((),a[i:=x])
```

where `()` is a singleton type, We will use the suggestive "do-notation": If `m1 : M u` and `m2 : M v` are expressions where `m2` may depend on a variable `x : u`, then

```
do { x <- m1 ; m2 } := bind m1 (\x-> m2) : M v
```

where `\x->` denotes lambda-abstraction. If `m2` does not depend on `x`, then

```
do { m1 ; m2 } := bind m1 (\_ -> m2) : M v
```

This notation can be extended to more than two actions in the obvious way. Furthermore, expressions of the form `do { ... x <- m ; return x }` can be simplified to `do { ... m }`.

Using canonical isomorphisms such as

$$\mathbb{N} \times \mathbb{N} \times A \to A \quad \simeq \quad \mathbb{N} \times \mathbb{N} \to (A \to () \times A)$$

we can write the programs `qsort`, `part`, and `swap` equivalently as follows, using the new names `mqsort`, `mpart` and `mswap`:

Listing 12 Qsort (Monadic Style)

```
mqsort(l,r) =
  do {
       p <- mpart(l,r) ;
       (if l < p
        then if p < r
             then do { mqsort(l,p-1) ; mqsort(p+1,r) }
             else mqsort(l,r-1)
        else if p < r
             then mqsort(l+1,r)
             else return ()
     }
```

Listing 13 Part (Monadic Style)

```
mpart : (N,N) -> M N
mpart(l,r) =
  if l = r
  then return l
  else do {
            x <- get l ;
            y <- get (l+1) ;
            (if x >= y
             then do { mswap(l,l+1); mpart(l+1,r) }
             then do { mswap(l+1,r); mpart(l,r-1) }
          }
```

Listing 14 Swap (Monadic Style)

```
mswap : (N,N) -> M ()
mswap(i,j) = do { x <- get(i) ; y <- get(j) ; put(i,y) ; put(j,x) }
```

The significance of the monadic notation is that the syntax alone enforces that arrays, once they have been used as input to a program involving the write operation, can no longer be accessed, thus enabling destructive interpretations of these operations. For example, we might want a modified sorting program `qsort1` that outputs the sorted array together with the original array. In the non-monadic style this can be done by simply defining

```
qsort1(l,r,a) = (qsort(l,r,a),a)
```

This would exhibit the desired behaviour if `qsort` is interpreted functionally, but not if it is interpreted imperatively. However, in the monadic style we need a copying operation

```
copy : M A
copy a = (a,a)
```

in order to define

```
mqsort1(l,r) = do { a <- copy ; mqsort(l,r) ; return a }
```

The latter program is correct w.r.t. the functional *and* the imperative interpretation.

5 Automated Monadification

Abstracting from the Quicksort case study we now define a functional language **SIT** modelling a subset of Minlog's term language which captures the idea of single-threadedness and includes the original functional Quicksort program extracted by Minlog. Then we describe a (meaning preserving) translation of this language into a monadic language which admits an imperative interpretation.

5.1 Remark

A different monadification process has been studied by Erwig and Ren [10]. They consider the problem of transforming a program of type $\rho \to \sigma$ into one of type $\rho \to M(\sigma)$ where M is a "runnable" monad (i.e. a monad with a left inverse of the *return* operation) and the resulting program is again functional. On the other hand, our translation transforms programs of type $(\rho, A) \to (\sigma, A)$ into programs of type $\rho \to M(\sigma)$ where M is specifically the state monad, i.e. we deal with programs where the input and output may be state dependent. In addition we are careful that the resulting program has an imperative interpretation.

5.2 The Single-Threaded Functional Language SIT

In the **SIT** language we have two different kinds of types. A distinguished type A of *states* and other "ordinary" types ρ, σ different from A. For the Quicksort example the state is an array and the only other types are natural numbers and Boolean values.

SIT is parametrized by three different kinds of functions which are distinguished by the types of values they access and return:

- Basic Functions – functions that neither access or modify the state:
  - $f^{\mathrm{bas}} : \vec{\rho} \to \sigma$
- Accessor Functions – functions which may require access to the state, but do *not* modify it:
  - $f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma$

- Quasi Side-Effect Functions – functions that may modify the state ("Quasi" because they are **SIT** functions which are side-effect free, but will later be translated into functions with side-effects):
  - $f^{\mathrm{sid}} : (\vec{\rho}, A) \to (\vec{\sigma}, A)$

In $(\rho_1, \ldots, \rho_n, A)$ the number n may be zero in which case we write just (A). We let x, y, z range over an infinite set of variables which will later be given ordinary types. We only need *one* state variable a.

The terms t and expressions e of **SIT** are defined as follows:

- SitTerm $\ni t ::= x \mid f^{\mathrm{bas}}(\vec{t}) \mid f^{\mathrm{acc}}(\vec{t}, a)$
- SitExpr $\ni e ::= (\vec{t}, a) \mid f^{\mathrm{sid}}(\vec{t}, a) \mid$ if t then e_1 else $e_2 \mid$ let $(\vec{x}, a) = e_1$ in e_2

where all variables in $\vec{x}$ are different. In general, $\vec{x}, \vec{y}, \ldots$ will always denote vectors of pairwise different variables.

A **SIT** program is a finite list of equations of the form

$$f_1^{\mathrm{sid}}(\vec{x}_1, a) = e_1$$
$$\ldots$$
$$f_n^{\mathrm{sid}}(\vec{x}_n, a) = e_n\,.$$

The functions f_i^{sid} may occur in the expressions e_j and are considered to be (possibly recursively) defined by the equations. All other functions symbols occurring in the e_j are considered as predefined functions (or parameters of the program).

The "let" is the only construction which binds free variables. In fact, the "let" expression let $(\vec{x}, a) = e_1$ in e_2 is to be understood as the β-redex $(\lambda(\vec{x}, a).e_2)e_1$. This intuition is reflected by the following axiom:

▶ Axiom 1. let $(\vec{x}, a) = (\vec{t}, a)$ in $e \equiv e[\vec{t}/\vec{x}]$ for all $\vec{t} \in$ SitTerm, $e \in$ SitExpr,

where substitution of terms into expressions, $e[\vec{t}/\vec{x}]$, is defined below.

Note that in a "let" expression, let $(\vec{x}, a) = e_1$ in e_2, the equation $(\vec{x}, a) = e_1$ is *not* to be understood as a (potentially) recursive definition of $(\vec{x}, a)$ as it is the case in Haskell. Therefore, our "let" rather corresponds to Scheme's "let" while Haskell's "let" corresponds to Scheme's "letrec".

The free variables for "let" constructs are defined as:

$$\mathrm{FV}(\text{let } (\vec{x}, a) = e_1 \text{ in } e_2) := \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{\vec{x}, a\})\,.$$

The notion of α-equivalence is defined as usual, in particular:

$$\text{let } (\vec{x}, a) = e_1 \text{ in } e_2 \overset{\alpha}{\equiv} \text{let } (\vec{y}, a) = e_1 \text{ in } e_2[\vec{y}/\vec{x}]$$

where $\vec{y}$ are fresh variables. Simultaneous substitution, $e[\vec{t}/\vec{x}]$, of terms $\vec{t}$ for variables $\vec{x}$ in a **SIT** expression e is defined as follows:

$$\begin{aligned}
y[\vec{t}/\vec{x}] &:= \begin{cases} t_i, & \text{if } y \in \{x_1, \ldots, x_n\}, \text{where } y = x_i \\ y, & \text{otherwise} \end{cases} \\
f^{\mathrm{bas}}(\vec{s})[\vec{t}/\vec{x}] &:= f^{\mathrm{bas}}(s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}]) \\
f^{\mathrm{acc}}(\vec{s}, a)[\vec{t}/\vec{x}] &:= f^{\mathrm{acc}}(s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}], a) \\
(\vec{s}, a)[\vec{t}/\vec{x}] &:= (s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}], a) \\
f^{\mathrm{sid}}(\vec{s}, a)[\vec{t}/\vec{x}] &:= f^{\mathrm{sid}}(s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}], a) \\
(\text{if } t \text{ then } e_1 \text{ else } e_2)[\vec{t}/\vec{x}] &:= \text{if } s[\vec{t}/\vec{x}] \text{ then } e_1[\vec{t}/\vec{x}] \text{ else } e_2[\vec{t}/\vec{x}] \\
(\text{let } (\vec{y}, a) = e_1 \text{ in } e_2)[\vec{t}/\vec{x}] &:= \text{let } (\vec{y}, a) = e_1[\vec{t}/\vec{x}] \text{ in } e_2[\vec{t}/\vec{x}]
\end{aligned}$$

where in the "let" case we can assume, possibly after α-renaming, that $\vec{y} \cap (\vec{x} \cup \mathrm{FV}(\vec{t})) = \emptyset$.

5.3 SIT Typing System

In the following we define a typing system which derives judgements of the form $\Gamma \vdash t : \rho$ or $\Gamma \vdash e : (\vec{\rho}, A)$ where the context Γ is a finite set of type declarations $x : \rho$:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{f^{\mathrm{bas}} : \vec{\rho} \to \sigma \qquad \Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{bas}}(\vec{t}) : \sigma}$$

$$\frac{f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma \qquad \Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{acc}}(\vec{t}, a) : \sigma}$$

$$\frac{\Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash (\vec{t}, a) : (\vec{\rho}, A)}$$

$$\frac{f^{\mathrm{sid}} : (\vec{\rho}, A) \to (\vec{\sigma}, A) \qquad \Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{sid}}(\vec{t}, a) : (\vec{\sigma}, A)}$$

$$\frac{\Gamma \vdash t : \mathbb{B} \qquad \Gamma \vdash e_1 : (\vec{\sigma}, A) \qquad \Gamma \vdash e_2 : (\vec{\sigma}, A)}{\Gamma \vdash \text{if } t \text{ then } e_1 \text{ else } e_2 : (\vec{\sigma}, A)}$$

$$\frac{\Gamma \vdash e_1 : (\vec{\rho}, A) \qquad \Gamma, \vec{x} : \vec{\rho} \vdash e_2 : (\vec{\sigma}, A)}{\Gamma \vdash \text{let } (\vec{x}, a) = e_1 \text{ in } e_2 : (\vec{\sigma}, A)}$$

A **SIT** program

$$\begin{aligned} f_1^{\mathrm{sid}}(\vec{x}_1, a) &= e_1 \\ &\cdots \\ f_n^{\mathrm{sid}}(\vec{x}_n, a) &= e_n \end{aligned}$$

is well-typed if for each of the defined functions $f_i^{\mathrm{sid}} : (\vec{\rho}_i, A) \to (\vec{\sigma}_i, A)$ the typing judgement $\vec{x_i} : \vec{\rho}_i \vdash e_i : (\vec{\sigma}_i, A)$ is derivable.

Since the typing rules are syntax directed it is clear that the typing rules are decidable (in linear time) and therefore can be automated.

The single-threadedness of **SIT** is enforced by the fact that there are no functions with result type A. For example, $\mathtt{Wr}$ has result type (A) but not A. Otherwise, one could form well-typed, but non single-threaded terms such as $\mathtt{Rd}(i, \mathtt{Wr}(i, x, a)) + \mathtt{Rd}(i, a)$.

5.4 Quicksort as a SIT program

We briefly demonstrate that the extracted Quicksort program (written as recursive equations) can be written in **SIT**. We have the predefined functions $<, \leq, = : (\mathbb{N}, \mathbb{N}) \to \mathbb{B}$ (basic functions), $\mathtt{Rd} : (\mathbb{N}, A) \to \mathbb{N}$ (accessor function) and $\mathtt{Wr} : (\mathbb{N}, \mathbb{N}, A) \to (A)$ (quasi side-effect function). Semantically (expressed outside the **SIT** language), $\mathtt{Rd}(i, a) = a[i]$ and $\mathtt{Wr}(i, x, a) = (a[i := x])$. The **SIT** program defining Quicksort is as follows:

Listing 15 Qsort (**SIT**)

```
qsort : (N,N,A) -> (A)
qsort(l,r,a) =
  let (p,a) = part(l,r,a) in
     if l < p
     then if p < r
          then let (a) = qsort(l,p-1,a) in qsort(p+1,r,a)
          else qsort(l,r-1,a)
     else if p < r
          then qsort(l+1,r,a)
          else (a)
```

Listing 16 Part (**SIT**)

```
part : (N,N,A) -> (N,A)
part(l,r,a) =
  if l = r
  then (l,a)
  else if Rd(l+1,a) <= Rd(l,a)
       then let (a) = swap(l+1,l,a) in part(l+1,r,a)
       else let (a) = swap(l+1,r,a) in part(l,r-1,a)
```

Listing 17 Swap (**SIT**)

```
swap : (N,N,A) -> (A)
swap(i,j,a) = let (x,y,a) = (Rd(i,a),Rd(j,a),a) in
                let (a) = Wr(i,y,a) in Wr(j,x,a)
```

The above **SIT** code wouldn't run correctly in Haskell because, as mentioned in Sect. 5.2, Haskell would interpret, for example, `let (p,a) = part(l,r,a)` as a recursive definition of `(p,a)`. In order to obtain correct Haskell code (up to minor syntactic details such as capitalised functions) we can simply α-rename the state variables, for example, rename `let (p,a) = part(l,r,a) in` e to `let (p,a1) = part(l,r,a) in` $e[a1/a]$. Then we would end up with essentially the same code as in listings 9–11.

5.5 The Monadic Language MON

We now introduce a monadic language **MON** with programs of type $\vec{\rho} \to M\vec{\sigma}$ where M is a monad. For convenience we let M operate on tuples of types rather than single types. For the moment M is an arbitrary monad, but later in Sect. 6.1 we will interpret **MON** into **SIT** where it will be pinned down as the state monad. In **MON** we have two different kinds of functions:

- The basic functions of the **SIT** language,
$$f^{\mathrm{bas}} : \vec{\rho} \to \sigma\,.$$
- Functions which may access and modify the state,
$$g^{\mathrm{sid}} : \vec{\rho} \to M\vec{\sigma}\,.$$

The terms u and expressions m of **MON** are defined as follows:

- MonTerm $\ni u ::= x \mid f^{\mathrm{bas}}(\vec{u})$
- MonExpr $\ni m ::= g^{\mathrm{sid}}(\vec{u}) \mid \mathrm{return}\ \vec{u} \mid \mathrm{if}\ u\ \mathrm{then}\ m_1\ \mathrm{else}\ m_2 \mid \mathrm{bind}\ m_1\ (\lambda\vec{x}.m_2)$

where in bind m_1 $(\lambda\vec{x}.m_2)$ we assume $\vec{x} \notin \mathrm{FV}(m_1)$.

Notice that MonTerm $\subset$ SitTerm. A **MON** program is a finite list of equations

$$\begin{aligned} g_1^{\mathrm{sid}}(\vec{x}_1) &= m_1 \\ &\dots \\ g_n^{\mathrm{sid}}(\vec{x}_n) &= m_n\,. \end{aligned}$$

As with **SIT**, the functions g_i^{sid} are considered to be (possibly recursively) defined by the equations. All other functions symbols occurring in the m_j are considered as predefined functions.

5.6 MON Typing System

We define a typing system for judgements of the form $\Gamma \vdash u : \sigma$ and $\Gamma \vdash m : M\vec{\sigma}$ where contexts Γ are as before:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{f^{\mathrm{bas}} : \vec{\rho} \to \sigma \qquad \Gamma \vdash \vec{u} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{bas}}(\vec{u}) : \sigma}$$

$$\frac{\Gamma \vdash \vec{u} : \vec{\rho}}{\Gamma \vdash \mathrm{return}\ \vec{u} : M\vec{\rho}}$$

$$\frac{g^{\mathrm{sid}} : \vec{\rho} \to M\vec{\sigma} \qquad \Gamma \vdash \vec{u} : \vec{\rho}}{\Gamma \vdash g^{\mathrm{sid}}(\vec{u}) : M\vec{\sigma}}$$

$$\frac{\Gamma \vdash u : \mathbb{B} \qquad \Gamma \vdash m_1 : M\vec{\sigma} \qquad \Gamma \vdash m_2 : M\vec{\sigma}}{\Gamma \vdash \mathrm{if}\ u\ \mathrm{then}\ m_1\ \mathrm{else}\ m_2 : M\vec{\sigma}}$$

$$\frac{\Gamma \vdash m_1 : M\vec{\rho} \qquad \Gamma, \vec{x} : \vec{\rho} \vdash m_2 : M\vec{\sigma}}{\Gamma \vdash \mathrm{bind}\ m_1\ (\lambda\vec{x}.m_2) : M\vec{\sigma}}$$

5.7 Translation from SIT to MON

The translation of **SIT** into **MON** is parametric in a translation of the predefined function symbols. Hence, we assume that we have assigned in a one-to-one way:

- to every **SIT** accessor function $f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma$ a **MON** side-effect function $\mathbf{F}_0(f^{\mathrm{acc}}) : \vec{\rho} \to M\sigma$, and
- to every **SIT** quasi side-effect function $f^{\mathrm{sid}} : (\vec{\rho}, A) \to (\vec{\sigma}, A)$ a **MON** side-effect function $\mathbf{F}_0(f^{\mathrm{sid}}) : \vec{\rho} \to M\vec{\sigma}$

Recall that basic **SIT** functions $f^{\mathrm{bas}} : \vec{\rho} \to \sigma$ are at the same time basic **MON** functions and are hence translated into themselves. Based on this assignment we define a translation

$$\mathbf{F} : \mathrm{SitTerm} \cup \mathrm{SitExpr} \to \mathrm{MonExpr}$$

Terms

$\mathbf{F}(x)$	:=	return x (where x is a variable)
$\mathbf{F}(f^{\mathrm{bas}}(\vec{t}))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}$.return $f^{\mathrm{bas}}(\vec{x})$
$\mathbf{F}(f^{\mathrm{acc}}(\vec{t}, a))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.\mathbf{F}_0(f^{\mathrm{acc}})(\vec{x})$

Where bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.m$ is defined recursively as:

bind $\mathbf{F}(\emptyset)$ $\lambda\emptyset.m := m$

bind $\mathbf{F}(t, \vec{t})$ $\lambda x.\lambda\vec{x}.m :=$ bind $\mathbf{F}(t)$ $(\lambda x.$bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.m)$

assuming $\vec{x} \cap \mathrm{FV}(\vec{t}) = \emptyset$

Expressions

$\mathbf{F}((\vec{t}, a))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}$.return $\vec{x}$
$\mathbf{F}(f^{\mathrm{sid}}(\vec{t}, a))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.\mathbf{F}_0(f^{\mathrm{sid}})(\vec{x})$
$\mathbf{F}(\mathrm{if}\ t\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2)$	:=	bind $\mathbf{F}(t)$ $(\lambda x.$if x then $\mathbf{F}(e_1)$ else $\mathbf{F}(e_2))$
$\mathbf{F}(\mathrm{let}(\vec{x}, a) = e_1\ \mathrm{in}\ e_2)$	:=	bind $\mathbf{F}(e_1)$ $(\lambda\vec{x}.\mathbf{F}(e_2))$

where all introduced λ-abstractions use fresh variables. This induces a translation of any **SIT** program:

$$f_1^{\mathrm{sid}}(\vec{x}_1, a) = e_1$$
$$\dots$$
$$f_n^{\mathrm{sid}}(\vec{x}_n, a) = e_n$$

into the **MON** program:

$$\mathbf{F}_0(f_1^{\mathrm{sid}})(\vec{x}_1) = \mathbf{F}(e_1)$$
$$\dots$$
$$\mathbf{F}_0(f_n^{\mathrm{sid}})(\vec{x}_n) = \mathbf{F}(e_n)\,.$$

▶ **Lemma 5** (Type preservation of translation **F**).
- *If* $\Gamma \vdash t : \rho$ *then* $\Gamma \vdash \mathbf{F}(t) : M\rho$, *and*
- *If* $\Gamma \vdash e : (\vec{\rho}, A)$ *then* $\Gamma \vdash \mathbf{F}(e) : M\vec{\rho}$

Proof. By induction on the typing derivations. ◀

▶ **Corollary 6.** *If a* **SIT** *program is well-typed then its monadic translation is also well-typed.*

5.8 Translating the SIT code of Quicksort into MON

We apply the translation **F** to the **SIT** code of the Quicksort program shown in Sect. 5.4. Defining the function $\mathbf{F}_0$ as:
- $\mathbf{F}_0(\texttt{Rd}) = \texttt{get}$
- $\mathbf{F}_0(\texttt{Wr}) = \texttt{put}$
- $\mathbf{F}_0(\texttt{qsort}) = \texttt{mqsort}$
- $\mathbf{F}_0(\texttt{part}) = \texttt{mpart}$
- $\mathbf{F}_0(\texttt{swap}) = \texttt{mswap}$

and using the "do-notation" introduced in Sect. 4.5 we obtain exactly the **MON** programs in the same section (listings 12-14).

6 Soundness of the translation

To prove the soundness of the monadic translation **F** we define a new translation:

$$\mathbf{G} : \mathbf{MON} \to \mathbf{SIT}$$

which "demonadifys" the program. We will prove the following theorem (which will be Theorem 10 below):

For all typable **SIT** expressions e, $\mathbf{G}(\mathbf{F}(e)) \equiv e$.

6.1 Translation from MON to SIT

The following translation of **MON** into **SIT** can be viewed as a definition of the monadic constructs in functional terms. In fact, it expresses that M is the state monad. To every **MON** side-effect function $g^{\mathrm{sid}} : \vec{\rho} \to M\vec{\sigma}$, each of which we may assume to be in the image of $\mathbf{F}_0$, we assign a **SIT** quasi side-effect function $\mathbf{G}_0(g^{\mathrm{sid}}) : (\vec{\rho}, A) \to (\vec{\sigma}, A)$ as follows:
- If $g^{\mathrm{sid}} = \mathbf{F}_0(f^{\mathrm{sid}})$, then $\mathbf{G}_0(g^{\mathrm{sid}}) := f^{\mathrm{sid}}$,
- If $g^{\mathrm{sid}} = \mathbf{F}_0(f^{\mathrm{acc}})$, then $\mathbf{G}_0(g^{\mathrm{sid}}) := \widehat{f^{\mathrm{acc}}}$,

where for a **SIT** accessor function $f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma$, $\widehat{f^{\mathrm{acc}}} : (\vec{\rho}, A) \to (\sigma, A)$ is a new **SIT** quasi side-effect function that "behaves" like f^{acc}, that is, we postulate:

▶ Axiom 2. $\widehat{f^{\mathrm{acc}}}(\vec{x}, a) \equiv (f^{\mathrm{acc}}(\vec{x}, a), a)$.

Now we define the function

$$\mathbf{G} : \mathrm{MonExpr} \to \mathrm{SitExpr}\,.$$

Recall that MonTerm $\subset$ SitTerm, and **SIT** has only one state variable a which is used below.

$$\begin{array}{lcl}
\mathbf{G}(g^{\mathrm{sid}}(\vec{u})) & := & \mathbf{G}_0(g^{\mathrm{sid}})(\vec{u}, a) \\
\mathbf{G}(\text{return } \vec{u}) & := & (\vec{u}, a) \\
\mathbf{G}(\text{if } u \text{ then } m_1 \text{ else } m_2) & := & \text{if } u \text{ then } \mathbf{G}(m_1) \text{ else } \mathbf{G}(m_2) \\
\mathbf{G}(\text{bind } m_1\ \lambda\vec{x}.m_2) & := & \text{let } (\vec{x}, a) = \mathbf{G}(m_1) \text{ in } \mathbf{G}(m_2)
\end{array}$$

6.2 Auxiliary Lemmas

We prepare the proof of Theorem 10 by a sequence of lemmas.

▶ **Lemma 7.** *Let $\vec{t} = t_1, \ldots, t_n$. Assume $\Gamma \vdash \vec{t} : \vec{\rho}$, and $\mathbf{G}(\mathbf{F}(t_i)) \equiv (t_i, a)$. Let $\vec{x} = x_1, \ldots, x_n$ be pairwise different variables that are not free in $\vec{t}$. Then:*
(a) $\mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\text{return } f^{bas}(\vec{x})) \equiv (f^{bas}(\vec{t}), a)$
(b) $\mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\mathbf{F}_0(f^{acc})(\vec{x})) \equiv (f^{acc}(\vec{t}, a), a)$

Proof. For part (a) we prove more generally:

$$\mathbf{G}(\text{bind } \mathbf{F}(t_{k+1}, \ldots, t_n)\ \lambda x_{k+1}, \ldots, x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \equiv (f^{\mathrm{bas}}(x_1, \ldots, x_k, t_{k+1}, \ldots, t_n), a)$$

for all $k \in 1, \ldots, n$, by induction on $n - k$.

- Base case $n - k = 0$ $(n = k)$.

$$\begin{array}{ll}
& \mathbf{G}(\text{bind } \mathbf{F}(\emptyset)\ \lambda\emptyset.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \mathbf{G}(\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \qquad \text{(def. of bind)} \\
\equiv & (f^{\mathrm{bas}}(x_1, \ldots, x_n), a) \qquad \text{(def. of } \mathbf{G})
\end{array}$$

- Step $n > 0$ $(k < n)$.

$$\begin{array}{ll}
& \mathbf{G}(\text{bind } \mathbf{F}(t_{k+1}, \ldots, t_n)\ \lambda x_{k+1}, \ldots, x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \mathbf{G}(\text{bind } \mathbf{F}(t_{k+1})\ (\lambda x_{k+1}.\text{bind } \mathbf{F}(t_{k+2}, \ldots, t_n) \qquad \text{(def. of bind)} \\
& \qquad \lambda x_{k+2}, \ldots, \lambda x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n))) \\
\equiv & \text{let}(x_{k+1}, a) = \mathbf{G}(\mathbf{F}(t_{k+1})) \text{ in} \qquad \text{(def. of } \mathbf{G}) \\
& \quad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2}, \ldots, t_n)\ \lambda x_{k+2}, \ldots, \lambda x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \text{let}(x_{k+1}, a) = (t_{k+1}, a) \text{ in} \qquad \text{(assumption)} \\
& \quad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2}, \ldots, t_n)\ \lambda x_{k+2}, \ldots, \lambda x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \text{let } (x_{k+1}, a) = (t_{k+1}, a) \text{ in } (f^{\mathrm{bas}}(x_1, \ldots, x_k, x_{k+1}, t_{k+2}, \ldots, t_n), a) \qquad \text{(I.H.)} \\
\equiv & (f^{\mathrm{bas}}(x_1, \ldots, x_k, x_{k+1}, t_{k+2}, \ldots, t_n), a)[t_{k+1}/x_{k+1}] \qquad \text{(Axiom 1)} \\
\equiv & (f^{\mathrm{bas}}(x_1, \ldots, x_k, t_{k+1}, t_{k+2}, \ldots, t_n), a) \qquad (x_{k+1} \notin \mathrm{FV}(\vec{t}))
\end{array}$$

Similarly for part (b) we prove the following more general statement:

$$\mathbf{G}(\text{bind } \mathbf{F}(t_{k+1},\ldots,t_n)\ \lambda x_{k+1},\ldots,x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n)) \equiv (f^{\text{acc}}(x_1,\ldots,x_k,t_{k+1},\ldots,t_n,a),a)$$

for all $k \in 1,\ldots,n$, by induction on $n-k$.

- Base case $n-k=0$ $(n=k)$.

$$\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(\emptyset)\ \lambda\emptyset.\ \mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \mathbf{G}(\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n)) && \text{(def. of bind)}\\
&\equiv \mathbf{G}_0(\mathbf{F}_0(f^{\text{acc}}))(x_1,\ldots,x_n,a) && \text{(def. of } \mathbf{G})\\
&\equiv \widehat{f^{\text{acc}}}(x_1,\ldots,x_n,a) && \text{(def. of } \mathbf{G}_0)\\
&\equiv (f^{\text{acc}}(x_1,\ldots,x_n,a),a) && \text{(Axiom 2)}
\end{aligned}$$

- Step $n>0$ $(k<n)$.

$$\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(t_{k+1},\ldots,t_n)\ \lambda x_{k+1},\ldots,x_n.\ \mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(t_{k+1})\ (\lambda x_{k+1}.\text{bind } \mathbf{F}(t_{k+2},\ldots,t_n) && \text{(def. of bind)}\\
&\qquad\qquad \lambda x_{k+2},\ldots,\lambda x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n)))\\
&\equiv \text{let}(x_{k+1},a) = \mathbf{G}(\mathbf{F}(t_{k+1})) \text{ in} && \text{(def. of } \mathbf{G})\\
&\qquad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2},\ldots,t_n)\ \lambda x_{k+2},\ldots,\lambda x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \text{let}(x_{k+1},a) = (t_{k+1},a) \text{ in} && \text{(assumption)}\\
&\qquad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2},\ldots,t_n)\ \lambda x_{k+2},\ldots,\lambda x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \text{let } (x_{k+1},a) = (t_{k+1},a) \text{ in } (f^{\text{acc}}(x_1,\ldots,x_k,x_{k+1},t_{k+2},\ldots,t_n,a),a) && \text{(I.H.)}\\
&\equiv (f^{\text{acc}}(x_1,\ldots,x_k,x_{k+1},t_{k+2},\ldots,t_n,a),a)[t_{k+1}/x_{k+1}] && \text{(Axiom 1)}\\
&\equiv (f^{\text{acc}}(x_1,\ldots,x_k,t_{k+1},t_{k+2},\ldots,t_n,a),a) && (x_{k+1} \notin \text{FV}(\vec{t}))
\end{aligned}$$

◀

▶ **Lemma 8** (Soundness for a single term)**.**
If $\Gamma \vdash t : \sigma$ *(i.e.* t *is typable in the* **SIT** *language), then* $\mathbf{G}(\mathbf{F}(t)) = (t,a)$.

Proof. By induction on **SIT** terms.

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(x))\\
&\equiv \mathbf{G}(\text{return } x) && \text{(def. of } \mathbf{F})\\
&\equiv (x,a) && \text{(def. of } \mathbf{G})
\end{aligned}$$

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(f^{\text{bas}}(\vec{s}))\\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{s})\ \lambda\vec{x}.\text{return } f^{\text{bas}}(\vec{x})) && \text{(def. of } \mathbf{F})\\
&\equiv (f^{\text{bas}}(\vec{s}),a) && \text{(Lemma 7)}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(f^{\text{acc}}(\vec{s},a)))\\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{s})\ \lambda\vec{x}.\ \mathbf{F}_0(f^{\text{acc}})(\vec{x})) && \text{(def. of } \mathbf{F})\\
&\equiv (f^{\text{acc}}(\vec{x},a),a) && \text{(Lemma 7)}
\end{aligned}$$

◀

▶ **Lemma 9** (Bind Lemma). *If* $\Gamma \vdash \vec{t} : \vec{\rho}$ *and* $\vec{x} \cap \mathrm{FV}(\vec{t}) = \emptyset$, *then* $\mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.m) \equiv \mathbf{G}(m)[\vec{t}/\vec{x}]$.

Proof. By induction on the length of $\vec{t}$.

- Base $\mathrm{len}(\vec{t}) = 0$ (i.e. $\vec{t} = \emptyset$)

$$
\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(\emptyset)\ \lambda\emptyset.m) \\
&\equiv \mathbf{G}(m) && \text{(def. of bind)}
\end{aligned}
$$

- Step

$$
\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(t,\vec{t})\ \lambda x\lambda\vec{x}.m) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(t)\ (\lambda x.\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.m)) && \text{(def. of bind)} \\
&\equiv \text{let } (x,a) = \mathbf{G}(\mathbf{F}(t)) \text{ in } \mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.m) && \text{(def. of } \mathbf{G}) \\
&\equiv \text{let } (x,a) = \mathbf{G}(\mathbf{F}(t)) \text{ in } (\mathbf{G}(m)[\vec{t}/\vec{x}]) && \text{(I.H.)} \\
&\equiv \text{let } (x,a) = (t,a) \text{ in } (\mathbf{G}(m)[\vec{t}/\vec{x}]) && \text{(Lemma 8)} \\
&\equiv \mathbf{G}(m)[\vec{t}/\vec{x}][t/x] && \text{(Axiom 1)} \\
&\equiv \mathbf{G}(m)[t,\vec{t}/x,\vec{x}] && (x \notin \mathrm{FV}(\vec{t}))
\end{aligned}
$$

◀

6.3 Soundness Proof

Now we show the main Theorem:

▶ **Theorem 10** (Soundness for expressions). *If* $\Gamma \vdash e : (\vec{\rho}, A)$, *then* $\mathbf{G}(\mathbf{F}(e)) \equiv e$

Proof. By induction on **SIT** expressions.

- $e \equiv (\vec{t}, a)$.

$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}((\vec{t},a))) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\text{return } \vec{x}) && \text{(def. of } \mathbf{F}) \\
&\equiv \mathbf{G}(\text{return } \vec{x})[\vec{t}/\vec{x}] && \text{(Lemma 9)} \\
&\equiv (\vec{x},a)[\vec{t}/\vec{x}] && \text{(def. of } \mathbf{G}) \\
&\equiv (\vec{t},a)
\end{aligned}
$$

- $e \equiv f^{\mathrm{sid}}(\vec{t}, a)$.

$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}(f^{\mathrm{sid}}(\vec{t},a))) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\ \mathbf{F}_0(f^{\mathrm{sid}})(\vec{x})) && \text{(def. of } \mathbf{F}) \\
&\equiv \mathbf{G}(\mathbf{F}_0(f^{\mathrm{sid}})(\vec{x}))[\vec{t}/\vec{x}] && \text{(Lemma 9)} \\
&\equiv (\mathbf{G}_0(\mathbf{F}_0(f^{\mathrm{sid}}))(\vec{x},a)[\vec{t}/\vec{x}] && \text{(def. of } \mathbf{G}) \\
&\equiv f^{\mathrm{sid}}(\vec{x},a)[\vec{t}/\vec{x}] && \text{(def. of } \mathbf{G}_0) \\
&\equiv f^{\mathrm{sid}}(\vec{t},a)
\end{aligned}
$$

- $e \equiv \text{if } t \text{ then } e_1 \text{ else } e_2$.

$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}(\text{if } t \text{ then } e_1 \text{ else } e_2)) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(t)\ \lambda x.\ \text{if } x \text{ then } \mathbf{F}(e_1) \text{ else } \mathbf{F}(e_2)) && \text{(def. of } \mathbf{F}) \\
&\equiv \mathbf{G}(\text{if } x \text{ then } \mathbf{F}(e_1) \text{ else } \mathbf{F}(e_2))[t/x] && \text{(Lemma 9)} \\
&\equiv (\text{if } x \text{ then } \mathbf{G}(\mathbf{F}(e_1)) \text{ else } \mathbf{G}(\mathbf{F}(e_2)))[t/x] && \text{(def. of } \mathbf{G}) \\
&\equiv (\text{if } x \text{ then } e_1 \text{ else } e_2)[t/x] && \text{(I.H.)} \\
&\equiv \text{if } t \text{ then } e_1 \text{ else } e_2 && \text{(Since } x \notin \mathrm{FV}(e_1) \text{ and } x \notin \mathrm{FV}(e_2))
\end{aligned}
$$

- $e \equiv \text{let } (\vec{x}, a) = e_1 \text{ in } e_2$.

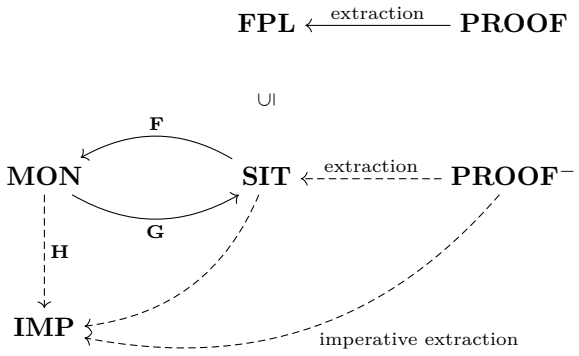
$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}(\text{let } (\vec{x}, a) = e_1 \text{ in } e_2)) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(e_1)\ \lambda \vec{x}.\mathbf{F}(e_2)) && \text{(def. of } \mathbf{F}) \\
&\equiv \text{let } (\vec{x}, a) = \mathbf{G}(\mathbf{F}(e_1)) \text{ in } \mathbf{G}(\mathbf{F}(e_2)) && \text{(def. of } \mathbf{G}) \\
&\equiv \text{let } (\vec{x}, a) = e_1 \text{ in } e_2 && \text{(I.H.)}
\end{aligned}
$$

◀

Since we understand the translation **G** as the definition of the semantics of monadic expressions this theorem states that the transformation **F** is a semantic preserving translation.

7 Conclusion and Further Work

We started by extracting a program from a formal proof of the Quicksort algorithm using the interactive theorem prover Minlog. We then observed, using Monads, that the extracted program behaves imperatively. Using this example for inspiration we defined a restricted functional language **SIT**, a monadic language **MON** and an automatic translation between the two that would allow for fully automatic imperative program extraction. At present it is by chance that the program extracted from the proof behaves imperatively, but we plan to develop a restricted proof calculus which *only* yields imperative programs. The situation is depicted in the diagram below:



The full arrows correspond to the results of this paper whereas the dashed arrows hint at further work:

- a restricted proof calculus, called **PROOF**$^-$, where the extracted programs from proofs are always in the **SIT** language;
- a translation **H** from **MON** to an imperative language **IMP** which would enable, through composition of **F** and **H**, an automatic extraction of imperative programs;
- a direct *imperative extraction* from **PROOF**$^-$ to **H**.

In this paper we considered only first-order constructs, but, ideally, we would like to develop a proof calculus and a program extraction process which combines higher order constructs with imperative features.

References

1 J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

2 Agda. `http://wiki.portal.chalmers.se/agda/`.

3 H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In *Automated Deduction*, volume II of *Applied Logic Series*, pages 41–71. Kluwer, 1998.

4 U. Berger, K. Miyamoto, H. Schwichtenberg, and M. Seisenberger. Minlog – A Tool for Program Extraction Supporting Algebras and Coalgebras. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO 2011*, volume 6859 of *Lecture Notes in Computer Science*, pages 393–399. Springer, 2011.

5 U. Berger and M. Seisenberger. Proofs, Programs, Processes. *Theory of Computing Systems*, 51(3):313–329, 2012.

6 S. Berghofer. Program extraction in simply-typed higher order logic. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2003.

7 B. Brock, M. Kaufmann, and J. S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293. Springer-Verlag, 1996.

8 C. M. Chuang. *Extraction of Programs for Exact Real Number Computation Using Agda*. PhD thesis, Swansea University, Wales, 2011.

9 The Coq Proof Assistant. `http://coq.inria.fr/`.

10 M. Erwig and D. Ren. Monadification of Functional Programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

11 S. Hallerstede and M. Leuschel. Experiments in program verification using Event-B. *Formal Aspects of Computing*, 24:97–125, 2012.

12 C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

13 Isabelle. `http://isabelle.in.tum.de/`.

14 S. C. Kleene. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

15 G. Kreisel. Interpretation of Analysis by means of Constructive Functionals of Finite Types. *Constructivity in Mathematics*, pages 101–128, 1959.

16 J. Krivine. Realizability Algebras: A Program to Well Order $\mathbb{R}$. *Logical Methods in Computer Science*, 7:1–47, 2011.

17 P. Letouzey. A New Extraction for Coq. In *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219, 2003.

18 The Minlog System. `http://www.minlog-system.de`.

19 A. Miquel. Classical Program Extraction in the Calculus of Constructions. In *CSL 2007*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327, 2007.

20 K. Miyamoto, F. Nordvall Forsberg, and H. Schwichtenberg. Program Extraction from Nested Definitions. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 370 – 385. Springer, 2013.

21 E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

22 PRL Project. `http://www.nuprl.org/`.

23 I. Poernomo, J. N. Crossley, and M. Wirsing. *Adapting Proofs-as-Programs: The Curry-Howard Protocol*. Springer, 2005.

24 Swansea Minlog Repository. `http://cs.swan.ac.uk/minlog/`.

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

26 S. Ray and R. Sumners. Verification of an In-place Quicksort in ACL2. In D. Borrione, M. Kaufmann, and J. S. Moore, editors, 3*rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 204–212, Grenoble, 2002.

27 H. Schwichtenberg and S. S. Wainer. *Proofs and Computations*. Perspectives in Logic. Assoc. Symb. Logic and Cambridge Univ. Press, 2012.

28 TIOBE. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

A Model of Type Theory in Cubical Sets

Marc Bezem[1], Thierry Coquand[2], and Simon Huber[2]

1 **Department of Informatics, University of Bergen**
Postboks 7800, N-5020 Bergen, Norway
bezem@ii.uib.no

2 **Department of Computer Science and Engineering, University of Gothenburg**
SE-412 96 Göteborg, Sweden
{thierry.coquand, simon.huber}@cse.gu.se

Abstract

We present a model of type theory with dependent product, sum, and identity, in cubical sets. We describe a universe and explain how to transform an equivalence between two types into an equality. We also explain how to model propositional truncation and the circle. While not expressed internally in type theory, the model is expressed in a constructive metalogic. Thus it is a step towards a computational interpretation of Voevodsky's Univalence Axiom.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.2 Semantics of Programming Languages

Keywords and phrases models of dependent type theory, cubical sets, univalent foundations

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.107

La théorie singulière classique utilise des simplexes*; dans la suite de ce chapitre, nous aurons besoin d'une définition équivalente, mais utilisant des* cubes*; il est en effet évident que ces derniers se prêtent mieux que les simplexes à l'étude des produits directs, et, a fortiori, des espaces fibrés qui en sont la généralisation.*

(J. P. Serre, Thèse, Paris,1951 [21])

1 Introduction

In [16], Voevodsky proposes a new axiom in dependent type theory: the Univalence Axiom. This opens up for many improvements for the encoding of mathematics in type theory in general: function extensionality, identification of isomorphic structures, etc.

In order to preserve the good computational properties of type theory it is crucial that postulated constants have a computational interpretation. Concerning univalence, this is an important open problem. One way of attacking this problem is by constructing a model of the new axiom, in type theory itself, or at least in a constructive metalogic. The computational interpretation can then be obtained through the semantics, for example, by evaluating a term of type N (natural numbers) in the model.

The model of type theory with the Univalence Axiom given by Voevodsky [16] is based on Kan simplicial sets. A problem with a constructive approach to Kan simplicial sets is that degeneracy is in general undecidable [3]. This problem makes it impossible to use the Kan simplicial set model as it is to obtain a computational interpretation of univalence.

We present a model of dependent type theory in cubical sets. This can be seen as a generalization of Bishop's notion of *set* [4]. While not expressed internally in type theory,

this model is expressed in a constructive metalogic. It can be seen as a simplification and a constructive version of the Kan simplicial set model of type theory [16, 1].

The first combinatorial description of homotopy groups by Kan used cubical sets [15]; see [7], [27] for a more recent account. Our presentation of cubical sets amounts to have a formal representation of cubes seen as continuous maps $[0,1]^I \to X$, where I is a finite set of symbols, instead of using only continuous maps $[0,1]^n \to X$. If $I = x_1, \ldots, x_n$ such a continuous map u can be seen as a function of $x_1, \ldots, x_n$ which vary in the unit interval. We can then consider for instance $u(x_i = 0)$, which is the quantity u where we set x_i to be 0, or we can introduce a new symbol y and consider u to be a quantity as a function of $x_1, \ldots, x_n, y$, which is actually independent of y. We formalize this by defining a cubical set to be a covariant presheaf on a suitable base category, where objects are finite sets of symbols and maps are substitution. This opens connections with the theory of nominal sets [20, 19].

Following e.g. [11], we can give a model of type theory where a context is interpreted by a cubical set. Like for the classical model based on simplicial sets where one restricts the model to Kan fibrations, we restrict our model by requiring a certain *Kan structure* on dependent types. Like in Kan's original paper [15], such a Kan structure requires fillers of open boxes. However, in order for this structure to be preserved–in a constructive metalogic–under all type forming operations, in particular Π, a certain uniformity condition is required on the choice of the fillers. This structure is essential for validating the elimination rule of identity types.

The strengthening of the Kan condition is natural given the reformulation of the notion of cubical sets that we present in the first section, and the connection mentioned above with nominal sets.

In this paper we present the semantics of dependent products, sums and identity types. We also show how to interpret the universe, but only sketch one special case how one could define the Kan structure on the universe. We also only describe how to transform an equivalence between two small types into a path between these types. Based on the model described in the first version of this paper (a nominal version of it) C. Cohen, A. Mörtberg and the last two authors have implemented a type checker[1]. This implementation supports computing with the Univalence Axiom and Kan operations for the universe.

The paper is structured as follows. In the next two sections we introduce the category of names and substitutions and we define cubical sets. In Section 4 we explain the presheaf semantics of type theory in the special case of cubical sets. In the next two sections we define the uniform Kan condition and we give examples of cubical sets. In Section 7 we show that Kan cubical sets are a model for dependent types. In the last section we show how identity types can be interpreted in the Kan cubical set model, and describe the universe as a cubical set (and only indicate how Kan fillings can be given), and how to transform an equivalence into an equality of types. Finally, we explain how to represent in our model spaces up to homotopy such as the sphere, and the operation of propositional truncation, giving in particular a new computational interpretation of the axiom of description [26, Introduction].

2 The category of names and substitutions

We start by fixing a countable discrete set of names or symbols, hereafter called the *name space*, such that 0 and 1 are not names.

[1] Available at: `http://github.com/simhu/cubical`

▶ **Definition 1.** The category $\mathcal{C}$ of names and substitutions has as objects all finite decidable subsets of the name space, denoted by $I, J, K, \ldots$. A morphism $f\colon I \to J$ is a map $I \to J \cup \{0,1\}$ such that $f(i) = f(j)$ iff $i = j$ whenever $f(i)$ and $f(j)$ are in J.

Notice that $\{0,1\}$ is disjoint from J since J is a set of names. We say that i is in the *domain* of f, or that $f(i)$ is *defined*, if $f(i)$ is an element of J.[2] So the condition for f being a morphism can be reformulated by saying that f is injective on its domain.

Clearly, $1_I : I \to I$ defined by $1_I(i) = i$ for all $i \in I$ is a morphism. If $f : I \to J$ and $g : J \to K$ are morphisms, we define the composition $g \circ f$ by $(g \circ f)(i) = g(f(i))$ if i is in the domain of f, and $(g \circ f)(i) = f(i)$ if $f(i) = 0, 1$. Clearly, $g \circ f : I \to K$ is a morphism. We shall write fg for the composition $g \circ f$, so first f and then g. It is not difficult to see that composition is associative and that $1_I f = f = f 1_J$. Hence $\mathcal{C}$ is a category. From now on, we may simply write 1 instead of 1_I.

Every $f : I \to J$ has a unique extension to a map $I \cup \{0,1\} \to J \cup \{0,1\}$ that is the identity on $\{0,1\}$, and this canonical extension respects composition. Together with $I \mapsto I \cup \{0,1\}$ we get a functor $\mathcal{C} \to \mathsf{Set}$.

We think of $f : I \to J$ as a substitution with renaming, where the only values we can substitute are 0 and 1. In particular we have for any x in I two substitutions $(x = b) : I \to I - x$, for $b = 0, 1$, defined by $(x = b)(y) = y$ if $y \neq x$ and $(x = b)(x) = b$. These are the *face maps*. Thus there are $2n$ face maps when I has n elements, that is, in dimension n (where simplicial sets have $n + 1$ face maps).

We say that a map $f : I \to J$ is a *degeneracy map* iff all elements in I are in the domain of f. For instance, if $I \subseteq J$ the canonical inclusion $I \to J$ defines a degeneracy map. If x is not in I the inclusion map $I \to I, x$ will be written as ι_x. We have two face maps $(x = 0), (x = 1) : I, x \to I$ and we have $\iota_x(x = 0) = \iota_x(x = 1) = 1_I$, which is one example of a *cubical identity*. There are many more cubical identities, often implicit in the notations. We also have the following result (cf. simplicial sets): every morphism f has a unique decomposition $f = gh$ where g is a composite of face maps and h is a degeneracy map.

If $f : I \to J$ is defined on x, we write $f - x : I - x \to J - f(x)$ for the map defined by $(f - x)(y) = f(y)$ if y is in $I - x$.

If $f : I \to J$ and x is not in I and y is not in J, we can extend f to a map $(f, x = y) : I, x \to J, y$ by sending x to y.

3 Cubical sets

▶ **Definition 2.** A *cubical set* is a covariant functor $\mathcal{C} \to \mathsf{Set}$.

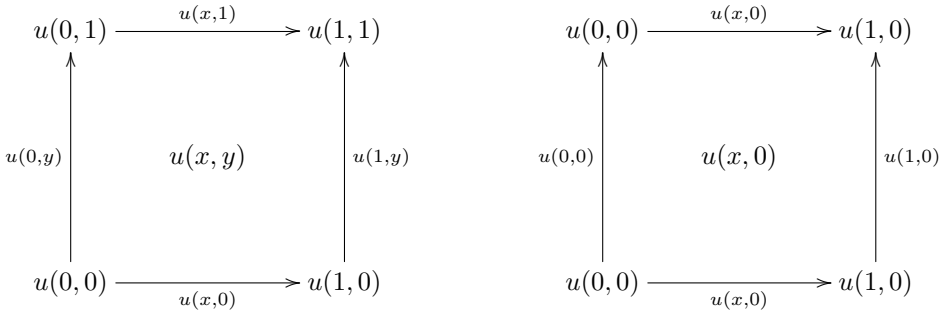
Let X be a cubical set. Then we have sets $X(I)$ and set maps (called *restrictions*) $X(I) \to X(J)$, $u \longmapsto uf$ for any morphism $f : I \to J$, such that $u1 = u$ and $u(fg) = (uf)g$. Another notation for uf would be $X(f)(u)$.

A cubical set X is a presheaf on the category $\mathcal{C}^{op}$. Any finite set of directions I represents by the Yoneda embedding $\mathbf{y}\colon \mathcal{C}^{op} \to \mathsf{Set}^{\mathcal{C}}$ a cubical set $\mathbf{y}I$, which can be thought of as a formal representation of $[0,1]^I$. An element of $X(I)$ can then be seen as a formal representation of a "continuous" map $[0,1]^I \to X$, and it is natural to call an element of $X(I)$ an I-cube.

[2] In a previous attempt, we have been considering the category of finite sets with maps $I \to J + 2$ (i.e. the Kleisli category for the monad $I + 2$). This category appears on pages 47–48 in Pursuing Stacks [10] as "in a sense, the smallest test category".

For finite sets of names we will write commas instead of unions and often omit curly braces; e.g. we write I, x for $I \cup \{x\}$, $I - x$ for $I - \{x\}$, and $X(x_1, \ldots, x_n)$ for $X(\{x_1, \ldots, x_n\})$.

We think of u in $X(I)$ as meaning that u may depend on the names in I, and only on those names; we think of uf in $X(J)$ as the element we obtain by performing the substitution f on u, possibly combined with renaming and/or adding variables. An element of $X()$ represents a point, an element ω of $X(x)$ a line connecting the points $\omega(x = 0)$ and $\omega(x = 1)$ in $X()$. An element in $X(x, y)$ represents a square. We then follow some notations similar to the ones in first-order logic by writing $u = u(x_1, \ldots, x_n)$ when u is in $X(x_1, \ldots, x_n)$. This is similar to saying that u may depend at most on the names $x_1, \ldots, x_n$. In doing so we always implicitly assume that the names $x_1, \ldots, x_n$ are pairwise distinct; the order of the names in $X(x_1, \ldots, x_n)$ does not matter. Applying a face map will now be expressed by actually performing the substitution. For example, we have that $u(x = 0)$ is in $X(y)$ whenever u is in $X(x, y)$:



If v is an $I - x$ cube of X then we can consider $v\iota_x$ which is an I-cube of X (we recall that $\iota_x : I - x \to I$ is the canonical inclusion). The map $v \longmapsto v\iota_x$ is injective (we have $v\iota_x(x = 0) = v$) and it is natural to identify v and $v\iota_x$, thus considering $X(I - x)$ to be a subset of $X(I)$. An example is the degenerate right square above.

If u is in $X(I)$ and x is in I, there may exist a v in $X(I - x)$ such that $u = v\iota_x = v$. Intuitively, this means that x "does not occur" in u, or that u is "independent" of x. One sometimes uses the notation $x\#u$ to express this relation. In general, this relation does not need to be decidable.

If X is a cubical set and a and u are two points ($\emptyset$-cube) of X we can define a new cubical set $\mathsf{Id}_X\ a\ u$ by taking an element in $(\mathsf{Id}_X\ a\ u)(I)$ to be an I, x-cube ω of X where x is a fresh variable (i.e. $x \notin I$), such that $\omega(x = 0) = a$ and $\omega(x = 1) = u$. The name x is "bound" in this operation so that another I, x'-cube ω' is equal to ω iff $\omega'(x' = x) = \omega$. We introduce a new binding operation $\langle x \rangle \omega$ which defines this I-cube of $\mathsf{Id}_X\ a\ u$. One way to make this notion precise is to assume a choice function on the set of names which selects a fresh name for any finite subset and define $\langle x \rangle \omega$ to be $\omega(x = x_I)$ where x_I is the fresh name given by the choice function. (This is the solution suggested in [22].)

The corresponding category with the same objects and morphisms $I \to J \cup \{0\}$ has been already considered as the category of *partial injections*. It has been shown by Staton that the category of covariant presheaves over this category is equivalent to the category of nominal sets with one restriction operation (see [20, exercise 9.7]). Using the same method, we can associate in a canonical way a nominal set to any cubical set. A category equivalent to to the category of cubical sets is presented in [19].

4 Cubical sets as a presheaf model

We will now recall how cubical sets, as does any presheaf category, give rise to a model of dependent type theory. We use Dybjer's notion of *category with families* (CwF) to devise such a model [9, 8, 11]. We stress the fact that such a structure is described by a generalized algebraic theory [5]. To give a CwF is to give:

1. interpretations (as sets) for the sorts of contexts, context morphisms (substitutions), types and terms;
2. operations;
3. checking equations.

This amounts to validate the rules given in Figure 1. Note that we use polymorphic notation to increase readability as in [5, 9]; e.g. without this convention we should have written $\mathsf{p}_{\Gamma,A}$ for the first projection $\mathsf{p}\colon \Gamma.A \to \Gamma$. Also, we leave the type parameters implicit, e.g. $(A\sigma)\delta = A(\sigma\delta)$ tacitly assumes the premises $\sigma\colon \Delta \to \Gamma$, $\delta\colon \Theta \to \Delta$ and $\Gamma \vdash A$. These points are also stressed in [25, Sec. 1] and [9].

We will now describe how cubical sets give rise to such a structure. This construction works for any presheaf category and is described in [11, Sec. 4]. Instead of using contravariant presheaves, we use covariant presheaves and write composition in diagram order.

A context Γ, written $\Gamma \vdash$, is interpreted by a cubical set, and context morphisms $\sigma : \Delta \to \Gamma$ are interpreted as cubical set maps (i.e. natural transformations), that is we have $(\sigma\beta)f = \sigma(\beta f)$ if β is a I-cube of Δ. A dependent type $\Gamma \vdash A$ is given by sets $A\alpha$ for each I-cube α of Γ together with maps (also called *restrictions*) $A\alpha \to A\alpha f$, $u \longmapsto uf$ for each $f\colon I \to J$, satisfying $u1 = u$ and $u(fg) = (uf)g$. Another way to express this is to say that A is a covariant presheaf on the category of elements of Γ, where the category of elements of Γ is given by objects (I, α) with $\alpha \in \Gamma(I)$, and morphisms $f\colon (I,\alpha) \to (J,\beta)$ given by $f\colon I \to J$ in $\mathcal{C}$ such that $\beta = \alpha f$. A section (or term) $\Gamma \vdash a : A$ is defined by giving an element $a\alpha$ in $A\alpha$ for each I-cube α of Γ in such a way that $(a\alpha)f = a(\alpha f)$ for any $f\colon I \to J$. The empty context $()$ is given by the cubical set with exactly one I-cube for each I. Given $\Gamma \vdash A$ and $\sigma\colon \Delta \to \Gamma$ we define $\Delta \vdash A\sigma$ by $(A\sigma)\alpha = A(\sigma\alpha)$ and the induced maps; likewise, substitution in a term $\Gamma \vdash a : A$ is given by $(a\sigma)\alpha = a(\sigma\alpha)$. If $\Gamma \vdash A$, we define the cubical set $\Gamma.A$ by taking as I-cubes of $\Gamma.A$ pairs (α, u) with α an I-cube of Γ and u in $A\alpha$. For $f : I \to J$ we define $(\alpha, u)f = (\alpha f, uf)$. The first projection $\mathsf{p}\colon \Gamma.A \to \Gamma$, $\mathsf{p}(\alpha, u) = \alpha$ becomes thus a context morphism, and the second projection $\mathsf{q}(\alpha, u) = u$ a section $\Gamma.A \vdash \mathsf{q} : A\mathsf{p}$ corresponding to the first de Bruijn index. For $\Gamma \vdash A$, $\sigma\colon \Delta \to \Gamma$ and $\Delta \vdash u : A\sigma$ we give $(\sigma, u)\colon \Delta \to \Gamma.A$ by $(\sigma, u)\beta = (\sigma\beta, u\beta)$. This concludes the description of the CwF without type formers.

We now describe how to interpret Π and Σ. If $\Gamma \vdash A$ and $\Gamma.A \vdash B$, we define the type $\Gamma \vdash \Pi\, A\, B$ as follows. For each I-cube α of Γ, an element w of $(\Pi\, A\, B)\alpha$ is a family (w_f) indexed by $f : I \to J$ such that

$$w_f \in \prod_{u \in A\alpha f} B(\alpha f, u)$$

is a dependent function and $(w_f(u))g = w_{fg}(ug)$ for $g\colon J \to K$ and $u \in A\alpha f$. We define the family wf in $(\Pi\, A\, B)\alpha f$ by putting $(wf)_g = w_{fg}$, which completes the definition of $\Gamma \vdash \Pi\, A\, B$. Given $\Gamma.A \vdash b : B$ we interpret $\Gamma \vdash \lambda\, b : \Pi\, A\, B$ by $((\lambda\, b)\alpha)_f(u) = b(\alpha f, u)$ for u in $A\alpha f$. Application $\Gamma \vdash \mathsf{app}(w, u) : B[u]$ (where $[u] = (1, u)\colon \Gamma \to \Gamma.A$) of $\Gamma \vdash w : \Pi\, A\, B$ to $\Gamma \vdash u : A$ is given by $\mathsf{app}(w, u)\alpha = (w\alpha)_1(u\alpha)$. We get $\mathsf{app}((\lambda\, b), u)\alpha = ((\lambda\, b)\alpha)_1(u\alpha) = b(\alpha, u\alpha) = (b[u])\alpha$.

$$\frac{\Gamma \vdash}{1 : \Gamma \to \Gamma} \qquad \frac{\sigma : \Delta \to \Gamma \quad \delta : \Theta \to \Delta}{\sigma\delta : \Theta \to \Gamma} \qquad \frac{\Gamma \vdash A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash A\sigma} \qquad \frac{\Gamma \vdash t : A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash t\sigma : A\sigma}$$

$$\frac{}{() \vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma.A \vdash} \qquad \frac{\Gamma \vdash A}{\mathsf{p} : \Gamma.A \to \Gamma} \qquad \frac{\Gamma \vdash A}{\Gamma.A \vdash \mathsf{q} : A\mathsf{p}}$$

$$\frac{\sigma : \Delta \to \Gamma \quad \Gamma \vdash A \quad \Delta \vdash u : A\sigma}{(\sigma, u) : \Delta \to \Gamma.A}$$

$$\frac{\Gamma.A \vdash B}{\Gamma \vdash \Pi\ A\ B} \qquad \frac{\Gamma.A \vdash B \quad \Gamma.A \vdash b : B}{\Gamma \vdash \lambda b : \Pi\ A\ B} \qquad \frac{\Gamma \vdash w : \Pi\ A\ B \quad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}(w, u) : B[u]}$$

$$\frac{\Gamma.A \vdash B}{\Gamma \vdash \Sigma\ A\ B} \qquad \frac{\Gamma.A \vdash B \quad \Gamma \vdash u : A \quad \Gamma \vdash v : B[u]}{\Gamma \vdash (u, v) : \Sigma\ A\ B} \qquad \frac{\Gamma \vdash w : \Sigma\ A\ B}{\Gamma \vdash w.1 : A} \qquad \frac{\Gamma \vdash w : \Sigma\ A\ B}{\Gamma \vdash w.2 : B[w.1]}$$

$$1\sigma = \sigma 1 = \sigma \qquad (\sigma\delta)\nu = \sigma(\delta\nu) \qquad [u] = (1, u)$$

$$A1 = A \quad (A\sigma)\delta = A(\sigma\delta) \qquad u1 = u \quad (u\sigma)\delta = u(\sigma\delta)$$

$$(\sigma, u)\delta = (\sigma\delta, u\delta) \quad \mathsf{p}(\sigma, u) = \sigma \quad \mathsf{q}(\sigma, u) = u \quad (\mathsf{p}, \mathsf{q}) = 1$$

$$(\Pi\ A\ B)\sigma = \Pi\ (A\sigma)\ (B(\sigma\mathsf{p}, \mathsf{q})) \qquad (\lambda b)\sigma = \lambda(b(\sigma\mathsf{p}, \mathsf{q}))$$

$$\mathsf{app}(w, u)\delta = \mathsf{app}(w\delta, u\delta) \qquad \mathsf{app}(\lambda b, u) = b[u] \qquad w = \lambda(\mathsf{app}(w\mathsf{p}, \mathsf{q}))$$

$$(\Sigma\ A\ B)\sigma = \Sigma\ (A\sigma)\ (B(\sigma\mathsf{p}, \mathsf{q})) \quad (w.1)\sigma = (w\sigma).1 \quad (w.2)\sigma = (w\sigma).2$$

$$(u, v)\sigma = (u\sigma, v\sigma) \qquad (u, v).1 = u \qquad (u, v).2 = v \qquad (w.1, w.2) = w$$

Figure 1 Rules of MLTT.

The definition of dependent sums is easier: $\Gamma \vdash \Sigma\, A\, B$ for $\Gamma \vdash A$ and $\Gamma.A \vdash B$ is defined by sums in each stage, i.e. for an I-cube α in Γ, $(\Sigma\, A\, B)\alpha$ consists of pairs (u, v) with u in $A\alpha$ and v in $B(\alpha, u)$. Restrictions are defined component-wise: $(u, v)f = (uf, vf)$ where $f \colon I \to J$. If $\Gamma \vdash w : \Sigma\, A\, B$ and $w\alpha = (u, v)$, then $(w.1)\alpha = u$ and $(w.2)\alpha = v$.

We can then verify all the equations of Figure 1.

5 The uniform Kan condition

Using these notations we can formulate the Kan condition (cf. [15]) and our strengthening as follows. Let X be a cubical set. First we define the notion of an *open box* in X, the equivalent of a *horn* in a simplicial set. Let I be a finite set of names and let $J, x \subseteq I$. The variable x must not be in J and will be the direction in which the box is open. For every $y \in J$, the open box will have two faces, one with $y = 0$ and one with $y = 1$. Let $O^+(J, x)$ consist of pairs $(x, 0)$ and (y, b) for $y \in J,\ b = 0, 1$. In the same way we define $O^-(J, x)$, but with $(x, 1)$ instead of $(x, 0)$. The idea for both is that one face in the direction x is missing. We use $O(J, x)$ to denote either $O^+(J, x)$ or $O^-(J, x)$. An *open box*, denoted by $\vec{u}$, is a family of elements (faces) u_{yb} in $X(I - y)$ for each $(y, b) \in O(J, x)$ such that

$$u_{yb}(z = c) = u_{zc}(y = b)$$

for all $(y,b),(z,c) \in O(J,x)$ with $y \neq z$. The latter condition may be phrased as: *the faces of an open box are adjacent-compatible*. If $f : I \to K$ is defined on J, x, we write $\vec{u}f$ for the open box indexed by $O(f(J), f(x))$ with components $(\vec{u}f)_{(fy)b} = u_{yb}(f - y)$ in $X(K - f(y))$.

For X to be a *(constructive) Kan cubical set*, we require to be given operations $X{\uparrow}$ and $X{\downarrow}$ for every $J, x \subseteq I$ such that $X{\uparrow}\vec{u}$ and $X{\downarrow}\vec{u}$ are both in $X(I)$. Here $\vec{u}$ is an open box with u_{x0} and u_{x1} in $X(I - x)$ in the respective cases $X{\uparrow}\vec{u}$ and $X{\downarrow}\vec{u}$. (From now on we will always tacitly assume that the open box $\vec{u}$ is of the right type with respect to $X{\uparrow}, X{\downarrow}$.) The operations $X{\uparrow}, X{\downarrow}$ are to be thought of as a filling their respective open boxes. Therefore we require for all $(y,b) \in O(J,x)$:

$$(X{\uparrow}\vec{u})(y = b) = u_{yb} \qquad (X{\downarrow}\vec{u})(y = b) = u_{yb}$$

The new uniformity condition is: if $f : I \to K$ is defined on J, x, we require:

$$(X{\uparrow}\vec{u})f = X{\uparrow}(\vec{u}f) \qquad (X{\downarrow}\vec{u})f = X{\downarrow}(\vec{u}f)$$

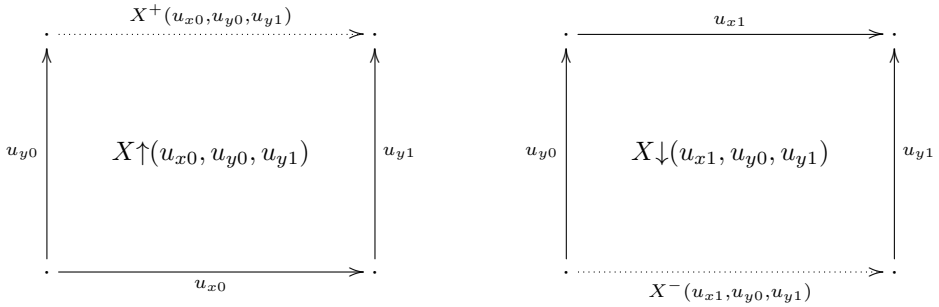
We refer to the combined condition as the *uniform Kan condition for cubical sets*, or the *Kan condition* for short.

If we only consider the case where $I = J, x$, that is, no other variables in I, and without the uniformity conditions, we get back the usual notion of Kan cubical sets [15, Sec. 4] (adapted to our notion of cubical sets). Similar uniformity conditions have been considered for semisimplicial sets in [2]. For a suggestive description of how to define combinatorially $\pi_n(X,u)$ for each point u of X if X satisfies the Kan property, see [27].

If X is a Kan cubical set with operations $X{\uparrow}, X{\downarrow}$, we define new operations (see the figure below)

$$X^+\vec{u} = (X{\uparrow}\vec{u})(x = 1) \qquad X^-\vec{u} = (X{\downarrow}\vec{u})(x = 0)$$

representing *transport* in the open box in the direction in which it is open.



Let Γ be a cubical set (which does not need to satisfy the Kan condition) and $\Gamma \vdash A$ a type over Γ. A *Kan structure* on $\Gamma \vdash A$ is given by operations $A\alpha{\uparrow}$ and $A\alpha{\downarrow}$ for each $\alpha \in \Gamma(I)$ and $J, x \subseteq I$, such that $A\alpha{\uparrow}\vec{u}$ and $A\alpha{\downarrow}\vec{u}$ are both in $A\alpha$ for every open box $\vec{u}$. Here *open box* means that $u_{yb} \in A\alpha(y = b)$ for all $(y,b) \in O(J,x)$, and that these faces are adjacent-compatible. $A\alpha{\uparrow}, A\alpha{\downarrow}$ must satisfy the same Kan conditions as $X{\uparrow}, X{\downarrow}$ above. The usual Kan conditions are obtained by simply substituting $A\alpha$ for X. Since $f : I \to K$ interacts with α, we reformulate the uniformity conditions:

$$(A\alpha{\uparrow}\vec{u})f = A\alpha f{\uparrow}(\vec{u}f) \qquad (A\alpha{\downarrow}\vec{u})f = A\alpha f{\downarrow}(\vec{u}f)$$

If $\Gamma \vdash A$ has a Kan structure with operations $A\alpha{\uparrow}, A\alpha{\downarrow}$, we define as before

$$A\alpha^+\vec{u} = (A\alpha{\uparrow}\vec{u})(x = 1) \qquad A\alpha^-\vec{u} = (A\alpha{\downarrow}\vec{u})(x = 0)$$

Notice that if $\Gamma \vdash A$ has a Kan structure, then the map $\mathsf{p}\colon \Gamma.A \to \Gamma$ is a Kan fibration as in [15, 27].

For $\Gamma \vdash A$ with Kan structure and a line α in $\Gamma(x)$ connecting points ρ_0 to ρ_1 one can define a map of cubical sets $A\rho_0 \to A\rho_1$ as follows. First, consider $A\rho_i$ as a cubical set with set of points $A\rho_i$, set of lines $A\rho_i\iota_x$, and so on. In general, we define $A\rho_i\iota_I$ by taking ι_I to be the unique morphism $\emptyset \to I$; restrictions are induced by $\Gamma \vdash A$. Then, the map $A\rho_0 \to A\rho_1$ is defined by $a \mapsto A\alpha^+ a$. The equivalence $a \mapsto A\alpha^+ a$ works uniformly and does not distinguish cases in which a is degenerate or not. One can show that this map is an equivalence (see Section 8.2 and 8.4). This is in contrast to Kan simplicial sets where classical logic is essential to define such an equivalence [3].

6 Examples of cubical sets

In this section we elaborate the following examples of cubical sets: discrete cubical sets; the unit interval $\mathbb{I}$; polynomial rings; the cubical nerve N of the group Z_2 with two elements; the exponential $N^{\mathbb{I}}$. A noticeable difference between simplicial sets and our cubical sets is that, while N is Kan, $N^{\mathbb{I}}$ is not. This is important motivation for the main result of the next section, implying that B^A is a Kan cubical set if both A and B are.

Every set A gives rise to the *discrete cubical set* $\mathsf{K}\,A$ via the constant presheaf, i.e. $(\mathsf{K}\,A)(I) = A$ for each I and all restrictions are the identity map $A \to A$. Note that in an open box $\vec{u}$ all the components have to be equal, say u, and this u is also the (unique) filler $u = \mathsf{K}\,A{\uparrow}\vec{u}$ making the discrete cubical set trivially into a Kan cubical set.

6.1 Unit interval

Recall the canonical extension of a map $f : J \to K$ in $\mathcal{C}$ to a set map $J \cup \{0,1\} \to K \cup \{0,1\}$ that is the identity on $\{0,1\}$. Together with mapping objects J of $\mathcal{C}$ to $J \cup \{0,1\}$, canonical extension actually forms a functor $\mathcal{C} \to \mathsf{Set}$. This covariant functor is called the *unit interval*, denoted by $\mathbb{I}$. We explore: $\mathbb{I}() = \{0,1\}$ ($\mathbb{I}$ has two points); $\mathbb{I}(x) = \{0,1,x\}$ ($\mathbb{I}$ has three lines, only x is non-degenerate); $\mathbb{I}(x,y) = \{0,1,x,y\}$ ($\mathbb{I}$ has four degenerate squares, see the display below); and so on. The square x varies in direction x, but is constant in direction y, and hence degenerate. Similarly for objects of higher dimension in $\mathbb{I}$. This completes the description of the unit interval as a cubical set. Note that $\mathbb{I} \cong \mathbf{y}\{x\}$ for a name x (where $\mathbf{y}$ denotes the Yoneda embedding) is another way to describe the interval.

$$\mathbb{I}(x,y) : \begin{array}{cccccccc} 0 & 0 & & 1 & 1 & & 1 & 1 \\ 0 & 0 & & 1 & 1 & & 0 & 0 \end{array}$$

6.2 Polynomial rings

A particularly natural example of a cubical set, suggested by Aczel, is based on polynomials over a commutative ring R. We let $R[I]$ as usual denote the ring of polynomials with coefficients in R and variables in (at most) I. For $x \notin I$ and $p \in R[I]$, we define $p\iota_x = p \in R[I,x]$. For $x \in I$ and $p \in R[I]$, we define $p(x=0) \in R[I-x]$ as p with $0 \in R$ substituted for x. Likewise, $p(x=1) \in R[I-x]$ is p with $1 \in R$ substituted for x. It is easily verified that this defines a cubical set, which we denote by $R[\_]$. In the following paragraph we show that $R[\_]$ has Kan structure.

For simplicity, we first take $I = x, J$ and $J = y, z$. After that, the general case can be done by an easy induction on $|J|$. Let $\vec{u}$ be an open box indexed by $O^+(J,x)$. The construction of filling an open box can be described as *iterated orthogonal linear interpolation*, in which we

stepwise approximate the filler p, one direction per step, ending with the direction in which the box is open. Define $p_z = (1-z)u_{z0} + zu_{z1}$. Then $p_z(z=0) = u_{z0}$, $p_z(z=1) = u_{z1}$, so p_z has the right faces in direction z. Now define:

$$p_{yz} = p_z + (1-y)(u_{y0} - p_z(y=0)) + y(u_{y1} - p_z(y=1))$$

This step is typical for the induction case. Per construction, p_{yz} has the right faces in the direction y. We verify that p_{yz} still has the right faces in the direction z. For $b = 0, 1$ we have

$$\begin{aligned} p_{yz}(z=b) &= p_z(z=b) + (1-y)(u_{y0}(z=b) - p_z(z=b)(y=0)) \\ &\quad + y(u_{y1}(z=b) - p_z(z=b)(y=1)) \\ &= u_{zb} + (1-y)(u_{y0}(z=b) - u_{zb}(y=0)) + y(u_{y1}(z=b) - u_{zb}(y=1)) \\ &= u_{zb}. \end{aligned}$$

In the last step above we have used that an open box has adjacent-compatible faces, such that $u_{yc}(z=b) = u_{zb}(y=c)$. It remains to define the filler $p = p_{xyz}$ by

$$p_{xyz} = p_{yz} + (1-x)(u_{x0} - p_{yz}(x=0))$$

and to verify the p has all the same faces as $\vec{u}$. The latter is similar to the verification of p_{yz} and is left to the reader. We note that the construction of the filler p is completely uniform, and hence $R[\_]$ has Kan structure.

We can also fill closed boxes by adding a term $x(u_{x1} - p_{yz}(x=1))$ to p_{xyz} above. Another consequence of linear interpolation is that the cubical set $R[\_]$ is contractible.

6.3 Cubical nerve

Recall that a morphism $f : J \to K$ in $\mathcal{C}$ is a function $f : J \to K \cup \{0,1\}$ such that for every $y \in K$ there exists at most one $x \in J$ with $f(x) = y$. Hence every morphism $f : J \to K$ defines a function $\{0,1\}^K \to \{0,1\}^J$ through precomposition with f. We can view $\{0,1\}^J$ as a product of posets $0 \leq 1$, and hence as a category with unique morphisms. Then every morphism $f : J \to K$ defines a functor $\{0,1\}^K \to \{0,1\}^J$, as the precomposition preserves the order. We denote this functor also by f.

Given a small category $\mathcal{D}$, we define its *cubical nerve* $N\mathcal{D}$ as follows. The sets $N\mathcal{D}(J)$ are functors $\{0,1\}^J \to \mathcal{D}$. For every morphism $f : J \to K$, its function $N\mathcal{D}(J) \to N\mathcal{D}(K)$ is defined by precomposition with the functor f. Note that the unit interval is not the cubical nerve of the poset $0 \leq 1$: they have similar sets of points and lines, but $N(0 \leq 1)$ has two more squares, both non-degenerate in two directions:

$$N(0 \leq 1)(x,y) : \begin{array}{cccccccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}$$

An element of $N\mathcal{D}(J)$ can be viewed as a (hyper)cube with the edges labelled by morphisms of $\mathcal{D}$ and vertices labelled by objects of $\mathcal{D}$, such that all paths commute (or equivalently, all triangles commute). This completes the description of the cubical nerve of a category.

Consider the group of two elements as a category (groupoid) with one object $\star$ and two morphisms $0, 1 : \star \to \star$ where 0 is the identity of $\star$. Let N be the nerve of this groupoid: N has one point and two lines, again denoted by $\star$ and $0, 1$. Note that $\star\iota_x = 0$ and $1 \circ 1 = 1 + 1 = 0$. The squares of N are listed as follows, where we only show the lines:

$$N(x,y): 0\begin{smallmatrix}0\\0\end{smallmatrix}0 \quad 1\begin{smallmatrix}0\\0\end{smallmatrix}1 \quad 0\begin{smallmatrix}1\\1\end{smallmatrix}0 \quad 1\begin{smallmatrix}0\\1\end{smallmatrix}0 \quad 1\begin{smallmatrix}1\\0\end{smallmatrix}0 \quad 0\begin{smallmatrix}1\\0\end{smallmatrix}1 \quad 0\begin{smallmatrix}0\\1\end{smallmatrix}1 \quad 1\begin{smallmatrix}1\\1\end{smallmatrix}1$$

Being the nerve of a groupoid, N is Kan (see the next section).

We now show that $N^{\mathbb{I}}$ is not Kan. By the Yoneda Lemma we have $N^{\mathbb{I}}(J) \cong (\mathbf{y}J \times \mathbb{I}) \to N)$, the latter denoting a set of natural transformations. Defining $p \in N^{\mathbb{I}}(J)$ means defining maps (index K omitted) $p : \mathbf{y}J(K) \to (\mathbb{I}(K) \to N(K))$ for all K, such that $(p_f u)g = p_{fg}(ug)$ for every $f : J \to K$, $g : K \to L$, and $u \in \mathbb{I}(K)$.

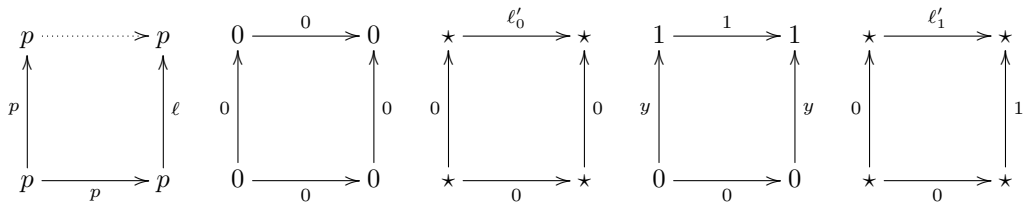
We explore the points of $N^{\mathbb{I}}$ and define $p \in N^{\mathbb{I}}()$ by, first $p_{1_{()}} : \mathbb{I}() \to N() : 0, 1 \mapsto \star$. Then, $p_{\iota_x} : \mathbb{I}(x) \to N(x) : 0, 1 \mapsto \star\iota_x = 0$ is forced by naturality, but for $p_{\iota_x}x$ there is a choice. If we choose 0, we must make the same choice for all names x in the name space. The choice 1 for all names x in the name space would give the only other point. In higher dimensions all arguments are degenerate, determining the function values, and naturality is compatible with each of the two choices above. We now fix p with $p_{\iota_x}x = 0$.

Next we explore lines from p to p in $N^{\mathbb{I}}$, say in direction i, and define $\ell : p \to p$ in $N^{\mathbb{I}}(i)$ by $\ell_{(i=b)g} = p_g$ for all $b = 0, 1$ and $g : \emptyset \to K$. For $\ell_{(i=x)} : \mathbb{I}(x) \to N(x)$ there is a choice. For the moment we put $\ell_{(i=x)}c = \ell_c$ for all $c \in \mathbb{I}(x)$. Note that we must make the same choices ℓ_0, ℓ_1, ℓ_x for all names x in the name space. On the next level, there is no choice left. First, $\ell_{(i=b)g} = p_g$ for $b = 0, 1$ and $g = \iota_x\iota_y$. Moreover, $\ell_{(i=x)\iota_y}, \ell_{(i=y)\iota_x} : \mathbb{I}(x, y) \to N(x, y)$ are completely determined by the choices of ℓ_0, ℓ_1, ℓ_x. Even more so, naturality limits the choice on the lower level. This can be seen by applying $\ell_{(i=x)\iota_y}$ and $\ell_{(i=y)\iota_x}$ to both x and y in $\mathbb{I}(x, y)$. This results in the four squares (NB: $\ell_x = \ell_y$):

$$\begin{array}{ccc} 0 & \overset{\ell_x}{} & 0 \\ & {\scriptstyle \ell_x} & \end{array} \quad \begin{array}{ccc} 0 & \overset{\ell_1}{} & 0 \\ & {\scriptstyle \ell_0} & \end{array} \quad \begin{array}{ccc} \ell_0 & & 0 \\ & & \\ 0 & & \ell_1 \end{array} \quad \begin{array}{ccc} \ell_y & & 0 \\ & & \\ 0 & & \ell_y \end{array}$$

Since the squares have to commute we get $\ell_0 = \ell_1$. In higher dimensions all values are determined by naturality, and naturality is compatible with each of the four possible choices (recall that objects in $\mathbb{I}$ can be non-degenerate in at most one direction). This yields in total four lines from p to p in $N^{\mathbb{I}}$.

In order to show that $N^{\mathbb{I}}$ is not Kan, consider lines $p, \ell : p \to p$, where p is degenerate ($p_0 = p_x = p_1 = 0$) and ℓ is defined by $\ell_0 = \ell_1 = 0,\ \ell_x = 1$. Consider an open box as in the picture below, left:



Assume we could fill the box. Call the closing (dotted) line above ℓ'. Applying the first square to the second results in the third square, yielding $\ell'_0 = 0$. Applying the first square to the fourth results in the last square, yielding $\ell'_1 = 1$. This contradicts $\ell'_0 = \ell'_1$ for any line $p \to p$. Hence the above box has no filler.

6.4 The nerve of a groupoid is Kan

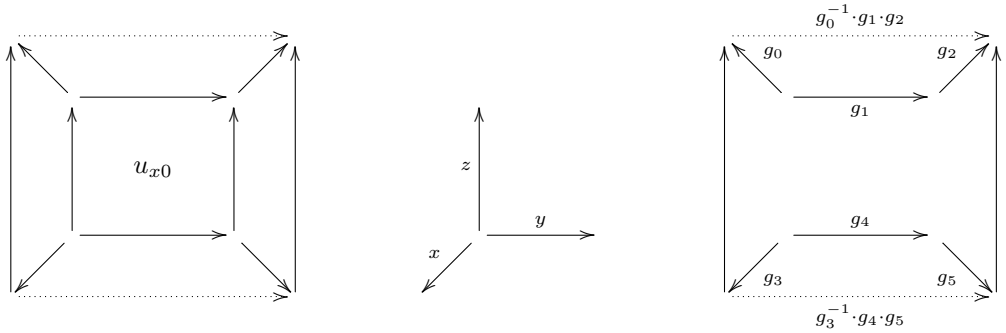
Let G be a groupoid, and N its cubical nerve. We sketch a proof that N is Kan. Take $I = x, J, z$ in $\mathcal{C}$, with $J = y_1, \ldots, y_k$ ($k \geq 0$). Taking one variable z instead of $z_1, \ldots, z_n$ simplifies the presentation, but is otherwise inessential.

Let $\vec{u}$ be an open box indexed by $O(J, x)$, that is, adjacent-compatible faces $u_{x0} \in N(I - x)$ and u_{yb} in $N(I - y)$. We have to define $u \in N(I)$ with faces as given by the open box. For this we define closing faces u_{x1},u_{z0},u_{z1}, such that they are adjacent-compatible with the

open box, and show that all squares commute. This will define u in a unique way. Thereafter we shall verify the uniformity condition.

If $J = \emptyset$ $(k = 0)$, the open 'box' is a degenerate line u_{x0} in direction z. We close by taking $u_{x1} = u_{z0} = u_{z1} = u_{x0}$, and u is the doubly degenerate square. If $J \neq \emptyset$ $(k > 0)$, we observe that all the points of u are already given by the open box, so that we can limit attention to the edges. Moreover, if J consists of more than one variable, all edges are also already present in the open box, which makes the definition of the closing faces particularly simple. This can be seen as follows. For $b = 0, 1$, the faces $u_{y_1 b}$ contain all edges in which $y_1 = b$, and the faces $u_{y_2 b}$ contain all edges in which $y_2 = b$. In particular, the two faces $u_{y_2 b}$ contain all edges in direction y_1. Hence, the four faces $u_{y_1 b}$, $u_{y_2 b}$ together contain all edges. The groupoid structure guarantees that all squares of the closing face commute.

The most interesting case to elaborate is $I = x, y, z,\ J = y$, where we have to define the edges in u_{x1} in direction y. This situation is depicted below, left, with the new edges as defined right. The new edges make essential use of the inverses in the groupoid and are uniquely defined.



The new squares u_{zb} commute as per construction. Moreover, the new square u_{x1} commutes since it can be projected down to the commuting square u_{x0} along edges that are invertible. A similar argument can be used if J contains more variables. This completes the construction of $u \in N(I)$.

Uniformity will be shown to be a consequence of the uniqueness of u constructed above, and the following easy lemma. This lemma can be useful in other places as well.

▶ **Lemma 3.** *For all morphisms $f : I \to K$ in $\mathcal{C}$ defined on x we have (i) $(x = b)(f - x) = f(f(x) = b)$ and (ii) $\iota_x f = (f - x)\iota_{fx}$.*

Now let $u = N\uparrow(u_{x0}, u_{y0}, u_{y1})$ and $u' = N\uparrow(u_{x0}(f - x), u_{y0}(f - y), u_{y1}(f - y))$. We have to show $u' = uf$. By the lemma we have $u_{x0}(f - x) = uf(f(x) = 0)$ and $u_{yb}(f - y) = uf(f(y) = b)$. This means that u' and uf agree on the open box defining u', so they are equal by uniqueness. Again, a similar argument can be used if J contains more variables. This completes the proof sketch that the cubical nerve of a groupoid is Kan.

7 The Kan cubical set model

In this section we will give a refinement of the model given in Section 4. The *Kan cubical set model* is given as follows: contexts and context morphisms are interpreted as in Section 4, i.e. by cubical sets and morphisms between cubical sets; a type is given by a type $\Gamma \vdash A$ in the sense of Section 4 *together* with a Kan structure; terms are given as in Section 4. The Kan structure on types is needed in order to justify the elimination rules for the identity types (cf. Section 8.2).

It is crucial to note that the Kan structure is part of a type in the Kan cubical set model. Two types $\Gamma \vdash A$ and $\Gamma \vdash B$ which have a Kan structure can be equal as cubical sets, but *not* with their Kan structure. Thus we have to check whether the equations between types in Figure 1 are preserved *for their Kan structure*.

The definition of the model is such that it follows the model described in Section 4, but additionally we have to define how the Kan structure is given on the types. This is done in the proofs of the following theorems.

▶ **Theorem 4.** *If $\Gamma \vdash A$ has a Kan structure and $\sigma \colon \Delta \to \Gamma$, then also $\Delta \vdash A\sigma$ has a Kan structure. Moreover the definition is such that $A1 = A$ and $(A\sigma)\tau = A(\sigma\tau)$ as types with Kan structures.*

Proof. For an I-cube α of Δ recall that $(A\sigma)\alpha = A(\sigma\alpha)$ as cubical sets; we define the filling operations in $(A\sigma)\alpha$ to be those in $A(\sigma\alpha)$, i.e. we set $(A\sigma)\alpha{\uparrow}\vec{u} = A(\sigma\alpha){\uparrow}\vec{u}$. With this definition it is clear that $A1$ and A have the same filling operations, and similarly for the other equation. ◀

7.1 Dependent product

▶ **Theorem 5.** *If both $\Gamma \vdash A$ and $\Gamma.A \vdash B$ have Kan structures, then so does $\Gamma \vdash \Pi\, A\, B$. Moreover the definition of the Kan structure is such that $(\Pi\, A\, B)\sigma = \Pi(A\sigma)(B(\sigma \mathsf{p}, \mathsf{q}))$.*

Proof. We present the argument in the case $J = \emptyset$, the general case is not essentially more difficult. Also, as the cases $\uparrow, \downarrow$ are perfectly symmetric, we restrict attention to $\uparrow$. We denote the direction of filling with a subscript to $\uparrow, \downarrow, {}^-, {}^+$. Let $C = \Pi\, A\, B$.

First we will define $C\alpha^+_x w \in C\alpha(x = 1)$ for α an I-cube of Γ, $x \in I$, and w in $C\alpha(x = 0)$. This amounts to define a family of dependent functions $(C\alpha^+_x w)_f$ in $\prod_{u \in A\alpha(x=1)f} B(\alpha(x = 1)f, u)$ for all $f \colon I - x \to K$, such that

$$\big((C\alpha^+_x w)_f(u)\big)g = (C\alpha^+_x w)_{fg}(ug). \tag{1}$$

We will first define $(C\alpha^+_x w)_f$ for $f = 1 \colon I - x \to I - x$. For this let $u \in A\alpha(x = 1)$. We use the Kan fillings to map u down to $A\alpha^-_x u$, apply w (at $1 \colon I - x \to I - x$) and map the result up:

$$(C\alpha^+_x w)_1(u) = B(\alpha, A\alpha{\downarrow_x}u)^+_x(w_1(A\alpha^-_x u)) \tag{2}$$

which is in $B(\alpha(x = 1), u)$ as $(A\alpha{\downarrow_x}u)(x = 1) = u$. So we have defined $(C\alpha^+ w)_1$ for arbitrary α and w.

For general $f \colon I - x \to K$ we let z be fresh w.r.t. K and set:

$$(C\alpha^+_x w)_f = (C\alpha(f, x = z)^+_z wf)_1 \tag{3}$$

By the uniformity conditions, this definition does not depend on the choice of z, and we also get by uniformity and (2)

$$\big((C\alpha^+_x w)_1(u)\big)f = (C\alpha(f, x = z)^+_z wf)_1(uf). \tag{4}$$

Note that (3) suffices to get the uniformity conditions for $C\alpha^+_x w$; (3) together with (4), yields (1) and thus an element in $C\alpha(x = 1)$, concluding the definition of $C\alpha^+_x w$.

Next we define $C\alpha{\uparrow_x}w \in C\alpha$; we do so again by first defining $(C\alpha{\uparrow_x}w)_f$ for $f = 1 \colon I \to I$. Let $\gamma \in A\alpha$, $u_0 = \gamma(x = 0)$ and $u = \gamma(x = 1)$; the definition of $(C\alpha{\uparrow_x}w)_1(\gamma) \in B(\alpha, \gamma)$ has

to satisfy:

$$\begin{array}{ccccc} (C\alpha^+_x w)_1 & : & u & \longmapsto & (C\alpha^+_x w)_1(u) \\ \uparrow & & \uparrow & & \uparrow \\ (C\alpha{\uparrow}_x w)_1 & : & \gamma & \longmapsto & (C\alpha{\uparrow}_x w)_1(\gamma) \\ \uparrow & & & & \uparrow \\ w_1 & : & u_0 & \longmapsto & w_1(u_0) \end{array}$$

Let y be a fresh name; using the uniform Kan filling for $\Gamma \vdash A$ in $A\alpha$ with $J = \{y\}$ (denoted by $A\alpha{\downarrow_{x,y}}$) we construct

$$\theta = A\alpha{\downarrow_{x,y}}(u, \gamma, A\alpha{\downarrow_x} u),$$

resulting in a square:

$$\begin{array}{ccc} u & \xrightarrow{u} & u \\ {\scriptstyle\gamma}\uparrow & \theta & \uparrow{\scriptstyle A\alpha\downarrow_x u} \\ u_0 & \xrightarrow[\theta(x=0)]{} & A\alpha^-_x u \end{array}$$

With $\lambda = B(\alpha, A\alpha{\downarrow_x} u){\uparrow_x}(w_1(A\alpha^-_x u))$ we get an open box in $B(\alpha, \theta)$

$$\begin{array}{ccc} (C\alpha^+_x w)_1(u) & \xrightarrow{(C\alpha^+_x w)_1(u)} & (C\alpha^+_x w)_1(u) \\ & & \uparrow{\scriptstyle\lambda} \\ w_1 u_0 & \xrightarrow[w_{\iota_y}(\theta(x=0))]{} & w_1(A\alpha^-_x u) \end{array}$$

where the line on the right hand side is by the defining equation (2). Using the Kan structure of $\Gamma.A \vdash B$ for $J = \{x\}$ we define

$$(C\alpha{\uparrow_x} w)_1(\gamma) = B(\alpha, \theta)^-_{y,x}\big(\lambda, w_{\iota_y}(\theta(x = 0)), (C\alpha^+_x w)_1(u)\big)$$

with λ as above. Using the uniformity conditions for $\Gamma \vdash A$ and $\Gamma.A \vdash B$, this definition is such that

$$\big((C\alpha{\uparrow_x} w)_1(\gamma)\big) f = (C\alpha f{\uparrow_{fx}} w(f - x))_1(\gamma f)$$

for $f \colon I \to K$ defined on x.

Now, if $f : I \to K$ is defined on x, we define $(C\alpha{\uparrow} w)_f = (C\alpha f{\uparrow_{fx}} w(f - x))_1$. If f is not defined on x, we can write $f = (x = b)f'$ for some $f' : I - x \to K$. Then we can simply define $(C\alpha{\uparrow_x} w)_f = w_{f'}$ for $b = 0$, and $(C\alpha{\uparrow_x} w)_f = (C\alpha^+_x w)_{f'}$ for $b = 1$. This defines the element $C\alpha{\uparrow} w$ in $C\alpha$ which satisfies the uniformity conditions.

To verify that the Kan structure of $\Pi(A\sigma)(B(\sigma\mathsf{p}, \mathsf{q}))$ (as defined above) is equal to the Kan structure for $(\Pi\, A\, B)\sigma$ (as defined in the proof of the preceding theorem), assume that above $\alpha = \sigma\beta$ for $\sigma \colon \Delta \to \Gamma$; then $C\alpha = ((\Pi\, A\, B)\sigma)\beta$ and in equation (2) we have

$$B(\sigma\beta, A(\sigma\beta){\downarrow_x} u)^+_x(w_1(A(\sigma\beta)^-_x u)) = (B(\sigma\mathsf{p}, \mathsf{q}))(\beta, (A\sigma)\beta{\downarrow_x} u)^+_x(w_1((A\sigma)\beta^-_x u))$$

and the right hand side is the definition of $\big(\Pi(A\sigma)(B(\sigma\mathsf{p}, \mathsf{q}))^+_x w\big)_1(u)$. Similarly for the other parts of the definition. ◀

Notice that we make essential use of the uniformity conditions in the above proof in order to verify that the fillers we define are indeed elements in the dependent product. Moreover, in the general case the fillings used from $\Gamma \vdash A$ are only with J such that $|J| \leq 1$.

7.2 Sum type

▶ **Theorem 6.** *If $\Gamma \vdash A$ and $\Gamma.A \vdash B$ have Kan structures, then so does $\Gamma \vdash \Sigma\, A\, B$. Moreover the definition of the Kan structure is such that $(\Sigma\, A\, B)\sigma = \Sigma(A\sigma)(B(\sigma\mathsf{p}, \mathsf{q}))$.*

Proof. Given an open box $\vec{p}$ in $(\Sigma\, A\, B)\alpha$ with $p_{yb} = (u_{yb}, v_{yb})$ for any $(y, b) \in O^+(J, x)$ we first fill $u = A\alpha\uparrow\vec{u}$ in $A\alpha$, and then set

$$(\Sigma\, A\, B)\alpha\uparrow\vec{p} = (u, B(\alpha, u)\uparrow\vec{v}).$$

This clearly satisfies the uniformity condition as they are satisfied for $\Gamma \vdash A$ and $\Gamma.A \vdash B$.

Moreover, if $\alpha = \sigma\beta$ for $\sigma \colon \Delta \to \Gamma$, we get $u = (A\sigma)\beta\uparrow\vec{u}$ and

$$B(\sigma\beta, u)\uparrow\vec{v} = (B(\sigma\mathsf{p}, \mathsf{q}))(\beta, u)\uparrow\vec{v},$$

yielding $(\Sigma\, A\, B)\sigma = \Sigma(A\sigma)(B(\sigma\mathsf{p}, \mathsf{q}))$. ◀

8 Extensions

8.1 Inductive types

We can interpret inductive types by adding the corresponding constructors in each dimension. In the case of inductive definitions without parameters this results in a discrete Kan cubical sets (see Section 6). E.g. the booleans $\Gamma \vdash N_2$ are defined by $N_2\alpha = \{\mathsf{true}, \mathsf{false}\}$ for each $\alpha \in \Gamma(I)$, and restrictions being the identity map. As in Section 6 one defines a Kan structure. We interpret the constants $\Gamma \vdash \mathsf{true} : N_2$ by $\mathsf{true}\alpha = \mathsf{true}$, and similar for $\Gamma \vdash \mathsf{false} : N_2$. To interpret the elimination principle

$$\frac{\Gamma.N_2 \vdash C \quad \Gamma \vdash d_0 : C[\mathsf{true}] \quad \Gamma \vdash d_1 : C[\mathsf{false}] \quad \Gamma \vdash b : N_2}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1 : C[b]}$$

we define $(\mathsf{if}\ b\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1)\alpha = d_0\alpha$ for $b\alpha = \mathsf{true}$, and $(\mathsf{if}\ b\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1)\alpha = d_1\alpha$ for $b\alpha = \mathsf{false}$.

8.2 Identity type

We describe the interpretation of $\Gamma \vdash \mathsf{Id}_A\ a\ b$ given $\Gamma \vdash A$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$. Given an I-cube α in Γ we define $(\mathsf{Id}_A\ a\ b)\alpha$ to be the set of elements $\langle x\rangle\omega$ where ω is in $A\alpha\iota_x$ and x is a fresh variable not in I, such that $\omega(x = 0) = a\alpha$ and $\omega(x = 1) = b\alpha$. The latter situation is conveniently described by $\omega : a\alpha \to_x b\alpha$. We recall that ι_x denotes the canonical injection $I \to I, x$. The element $\langle x\rangle\omega$ is the equivalence class of I, x-cubes of $A\alpha\iota_x$, x not in I, where ω is identified with $\omega(x = x')$ if x' is not in I. This operation $\langle x\rangle\omega$ binds the name x. (One could define $\langle x\rangle\omega$ to be $\omega(x = x_I)$ where x_I is a name not in I obtained by a choice function.) If f is a substitution $I \to K$ we choose a variable y not in K, extend f to $(f, x = y) : I, x \to K, y$ and define $(\langle x\rangle\omega)f$ to be $\langle y\rangle\omega(f, x = y)$, preserving equivalence.

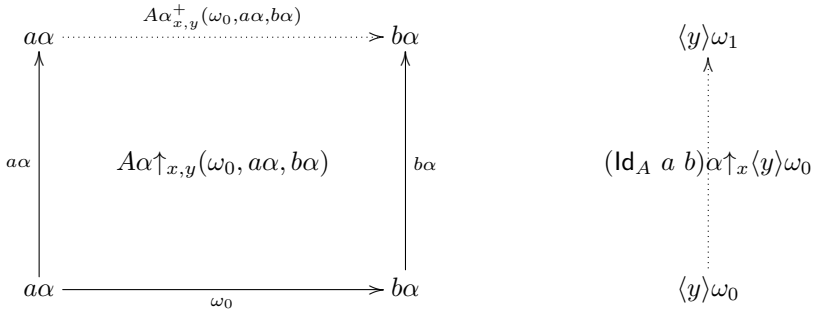
▶ **Theorem 7.** *If $\Gamma \vdash A$ has a Kan structure, then so does $\Gamma \vdash \mathsf{Id}_A\ a\ b$ whenever we have $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$. Moreover the definition is such that $(\mathsf{Id}_A\ a\ b)\sigma = \mathsf{Id}_{A\sigma}\ a\sigma\ b\sigma$ as types with Kan structures.*

Proof. Let α be an I-cube of Γ and $J, x \subseteq I$. After a suitable renaming, we can conveniently denote an open box for $(\mathsf{Id}_A\ a\ b)\alpha$ by a vector $\langle y\rangle\vec{\omega}$ with components $\langle y\rangle\omega_{zc}$ in $(\mathsf{Id}_A\ a\ b)\alpha(z = c)$, for all $(z, c) \in O(J, x)$.

We define, with $a\alpha, b\alpha$ the faces in the direction y, omitting subscripts J,

$$(\mathsf{Id}_A\ a\ b)\alpha\uparrow_x\langle y\rangle\vec{\omega} = \langle y\rangle(A\alpha\uparrow_{x,y}(\vec{\omega}, a\alpha, b\alpha))$$

which shows that $\Gamma \vdash \mathsf{Id}_A\ a\ b$ satisfies the Kan condition for J, x if $\Gamma \vdash A$ satisfies the Kan condition for $(J, y), x$. The situation in case $J = \emptyset$ is depicted below. The uniformity condition follows from the uniformity of $\Gamma \vdash A$. ◀



We give the interpretation of $\Gamma \vdash \mathsf{Ref}\ a : \mathsf{Id}_A\ a\ a$ given $\Gamma \vdash a : A$. For any set of directions I, and any I-cube ρ, we have to give a line $a\rho \to a\rho$. For this, we choose a direction x not in I and we define $(\mathsf{Ref}\ a)\rho = \langle x\rangle a\rho\iota_x$, which can also simply be written $(\mathsf{Ref}\ a)\rho = \langle x\rangle a\rho$.

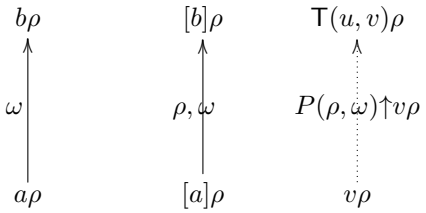
Next we show that we can interpret an elimination operator for the identity type. Suppose $\Gamma \vdash a : A$, $\Gamma \vdash b : A$, $\Gamma \vdash u : \mathsf{Id}_A\ a\ b$ and $\Gamma.A \vdash P$ and $\Gamma \vdash v : P[a]$. We will define an operator

$$\Gamma \vdash \mathsf{T}(u, v) : P[b].$$

Let ρ be some I-cube of Γ. Then $u\rho$ is of the form $\langle x\rangle\omega$ for some path $\omega : a\rho \to_x b\rho$, x not in I, $\omega \in A\rho$. The I, x-cube (ρ, ω) in $\Gamma.A$ is then a path $[a]\rho \to_x [b]\rho$ and we define (see the picture below)

$$\mathsf{T}(u, v)\rho = P(\rho, \omega)^+ v\rho \quad \text{where} \quad \langle x\rangle\omega = u\rho$$

The condition $(\mathsf{T}(u, v)\rho)f = \mathsf{T}(u, v)(\rho f)$ follows from the uniformity condition on the Kan filling operations.



We have that $P(\rho, \omega)\uparrow v\rho$ is a line connecting $v\rho$ and $\mathsf{T}(u, v)\rho$. In particular for $u = \mathsf{Ref}\ a$, this gives an interpretation of an operator

$$\Gamma \vdash \mathsf{H}(v) : \mathsf{Id}_{P[a]}\ v\ \ \mathsf{T}(\mathsf{Ref}\ a, v)$$

by taking $\mathsf{H}(v)\rho = \langle x\rangle P(\rho\iota_x, a\rho)\uparrow v\rho$. The computation rule for identity is thus only validated by a path to v via $\mathsf{H}(v)$[3].

[3] The validity of the computation rule for identity corresponds to considering only fibrations that are *regular* in the sense of Hurewicz [14].

$$\frac{\Gamma \vdash A \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash \mathsf{Id}_A\ a\ b} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{Ref}\, a : \mathsf{Id}_A\ a\ a}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash u : \mathsf{Id}_A\ a\ b \quad \Gamma.A \vdash P \quad \Gamma \vdash v : P[a]}{\Gamma \vdash \mathsf{T}(u, v) : P[b]}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma.A \vdash P \quad \Gamma \vdash v : P[a]}{\Gamma \vdash \mathsf{H}(v) : \mathsf{Id}_{P[a]}\ v\ \ \mathsf{T}(\mathsf{Ref}\, a, v)}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{center}\,(a, \mathsf{Ref}\, a) : \Pi\, T(\mathsf{Id}_{T\mathsf{p}}(a, \mathsf{Ref}\, a)\ \mathsf{q})} \quad \text{where } T = \Sigma\, A\, (\mathsf{Id}_{A\mathsf{p}} a \mathsf{p}\ \mathsf{q})$$

Figure 2 Rules for Id-types.

We finally show that, given $\Gamma \vdash a : A$, the type $\Gamma \vdash T = \Sigma\ A\ (\mathsf{Id}_{A\mathsf{p}}\ a\mathsf{p}\ \mathsf{q})$ is contractible. For this we have to find a center of T and a path to this center for any element of T. That is, we have to find two sections $\Gamma \vdash t : T$ and $\Gamma.T \vdash u : \mathsf{Id}_{T\mathsf{p}}\ t\mathsf{p}\ \mathsf{q}$. We define $t = (a, \mathsf{Ref}\, a)$. Let ρ be some I-cube of Γ and let $(v, \langle x \rangle \alpha)$ be some element of $T\rho$. So v is an element of $A\rho$ and α is a line connecting $a\rho$ and v in some direction x not in I. We introduce a direction y not in I, x and define:

$$u(\rho, (v, \langle x \rangle \alpha)) = \langle y \rangle (A\rho^{+}_{x,y}(a\rho, a\rho, \alpha), \langle x \rangle A\rho\uparrow_{x,y}(a\rho, a\rho, \alpha))$$

The fact that the filling operations commute with substitution ensures that this defines a section $\Gamma.T \vdash u : \mathsf{Id}_{T\mathsf{p}}\ t\mathsf{p}\ \mathsf{q}$.

We summarize the rules we interpret in the Kan cubical set model in Figure 2, where we left out the equations that the operations commute with substitutions, e.g. $(\mathsf{Id}_A\ a\ b)\sigma = \mathsf{Id}_{A\sigma}\ a\sigma\ b\sigma$.

N.A. Danielsson has checked formally in Agda that these properties are enough to develop all basic propositions of univalent mathematics; this Agda development[4] is accompanying the paper [6].

Let us define the more common elimination operator of C. Paulin-Mohring [18] from the above operations—with the difference that its computation rule only holds propositionally, and not as usual definitionally. In order not to make the notation too heavy we'll use informal reasoning in type theory; note that the definition can be given internally in type theory and we don't refer to the model; this definition follows N.A. Danielsson's Agda development (loc. cit.). First note that using the transport operation T one can define composition $p \circ q : \mathsf{Id}_A a\ c$ of two identity proofs $p : \mathsf{Id}_A a\ b$, $q : \mathsf{Id}_A b\ c$, as well as inverses $p^{-1} : \mathsf{Id}_A b\ a$. With H one can derive $\mathsf{Id}_{\mathsf{Id}_{A\,a}\ a}(p^{-1} \circ p)\ (\mathsf{Ref}\, a)$.

Let A be a type, $a : A$, and $C(b, p)$ a type given $b : A$, $p : \mathsf{Id}_A a\ b$, such that $v : C(a, \mathsf{Ref}\, a)$; for $b : A$ and $p : \mathsf{Id}_A a\ b$ we define $\mathsf{J}(a, v, b, p) : C(b, u)$. We can consider C as a dependent type over $T = (\Sigma x : A)\mathsf{Id}_A a\ x$ via $C(w.1, w.2)$ for $w : T$. As we showed in the last paragraph, T is contractible with center $(a, \mathsf{Ref}\, a)$, and thus we get a witness $\mathsf{app}(h, (b, p)) : \mathsf{Id}_T(a, \mathsf{Ref}\, a)\ (b, p)$ for $h = \lambda u$, u as in the above paragraph; now with T (w.r.t. the type $C(w.1, w.2)$ for $w : T$) we can define

$$\mathsf{J}(a, v, b, p) = \mathsf{T}(\mathsf{app}(h, (a, \mathsf{Ref}\, a))^{-1} \circ \mathsf{app}(h, (b, p)), v).$$

[4] Available at: `http://www.cse.chalmers.se/~nad/`

Now if $p = \mathsf{Ref}\, a$, we get that $\mathsf{app}(h, (a, \mathsf{Ref}\, a))^{-1} \circ \mathsf{app}(h, (b, p))$ is propositionally equal to $\mathsf{Ref}(\mathsf{Ref}\, a)$, and thus using T and H again one gets a witness of $\mathsf{Id}_{C(a,\mathsf{Ref}\, a)}\ v\ \ \mathsf{J}(a, v, a, \mathsf{Ref}\, a)$.

Even though J doesn't satisfy the judgmental equality, the model validates a new operation $\mathsf{mapOnPaths}$ which behaves well w.r.t. judgmental equality. Its rule given $\Gamma \vdash A$, $\Gamma \vdash B$, $\Gamma \vdash u : A$ and $\Gamma \vdash v : A$ is

$$\frac{\Gamma \vdash \varphi : A \to B \quad \Gamma \vdash p : \mathsf{Id}_A\ u\ v}{\Gamma \vdash \mathsf{mapOnPaths}(\varphi, p) : \mathsf{Id}_B\ (\mathsf{app}(\varphi, u))\ (\mathsf{app}(\varphi, v))}$$

where $A \to B$ is the non-dependent function space $\Pi A(B\mathsf{p})$. Given ρ in $\Gamma(I)$ we define $\mathsf{mapOnPaths}(\varphi, p)\rho = \langle x\rangle\, (\varphi\rho)_1\omega$ for $p\rho = \langle x\rangle\omega$. This satisfies the equations

$$\begin{aligned}
\mathsf{mapOnPaths}(\mathrm{id}, p) &= p \\
\mathsf{mapOnPaths}(\varphi \circ \psi, p) &= \mathsf{mapOnPaths}\big(\varphi, \mathsf{mapOnPaths}(\psi, p)\big) \\
\mathsf{mapOnPaths}(\varphi, \mathsf{Ref}\, a) &= \mathsf{Ref}(\mathsf{app}(\varphi, a)) \\
\mathsf{mapOnPaths}(\lambda(b\mathsf{p}), p) &= \mathsf{Ref}\, b
\end{aligned}$$

where now $\varphi \circ \psi$ denotes ordinary function composition and $\lambda(b\mathsf{p})$ is constant.

Notice that some of these equations do *not* hold if the identity type is defined as an inductive family, as in [17].

This interpretation of identity satisfies function extensionality (left to the reader).

8.3 Description of a universe

We now describe the interpretation of U as a universe of Kan cubical sets. We give U only as a cubical set (following [12, 23]) and only indicate how an operation similar to the Kan fillings can be given. The full proof that U has a Kan structure will be presented in the forthcoming [13].

Recall that the Yoneda embedding is denoted by $\mathbf{y}$. An element A of $U(I)$ is a type $\mathbf{y}I \vdash A$ with Kan structure such that for each $f\colon I \to J$ the set A_f is small (we use subscripts to keep the notation separate from the restrictions). Given such a $\mathbf{y}I \vdash A$ and $f\colon I \to J$ the restriction Af of A by f is defined to be $\mathbf{y}J \vdash A(\mathbf{y}f)$, where $\mathbf{y}f\colon \mathbf{y}J \to \mathbf{y}I$ is the substitution induced by f; thus $(Af)_g = A_{fg}$. This defines U as a cubical set.

Note that the points of U are simply the (small) uniform Kan cubical sets. More precisely, since $\emptyset$ is initial in $\mathcal{C}$, any A in $U(\emptyset)$ becomes a cubical set when we define $A(I)$ as A_f for the unique $f : \emptyset \to I$. A line in U between points A and B can be seen as a "heterogeneous" notion of lines, cubes, ... $a \to b$ where a is an I-cube of A and b an I-cube of B.

As a first step towards proving that this cubical set satisfies the Kan condition we show how to compose an A and B in $U(I)$ with $x \in I$ assuming $A(x = 1) = B(x = 0)$; we define $C = \mathsf{comp}(A, B) \in U(I)$ such that $C(x = 0) = A(x = 0)$, $C(x = 1) = B(x = 1)$, and for $f\colon I \to J$ defined on x, $Cf = \mathsf{comp}(Af, Bf)$. (Compare this to the composition of relations.)

We define the sets C_f, $f\colon I \to J$ by case distinction on $f(x)$; in case $f(x) = 0$, we can write $f = (x = 0)f'$ and we have to set $C_f = A_f$ as we have to satisfy $C_f = (C(x = 0))_{f'} = (A(x = 0))_{f'} = A_f$; similarly, if $f(x) = 1$, we set $C_f = B_f$. In case, f is defined on x, an element of C_f is any pair (a, b) such that $a \in A_f$ and $b \in B_f$ with $a(x = 1) = b(x = 0)$ in $A_{f(x=1)} = A(x = 1)_{(f-x)} = B(x = 0)_{(f-x)} = B_{f(x=0)}$.

We still have to define the restrictions $C_f \to C_{fg}$ for $g\colon J \to K$; in the first two cases from above, the restrictions are induced by A_f and B_f respectively. In case f is defined on x, we look at $g(f(x))$: if $g(f(x)) = 0$, we set $(a, b)g = ag$; if $g(f(x)) = 1$, we set $(a, b)g = bg$; and if g is defined at $f(x)$, we define $(a, b)g = (ag, bg)$.

It remains to define the Kan fillings for C; it suffices to give them for C_1 as C_f is either determined by A_f, B_f, or $\mathsf{comp}(Af, Bf)_1$; so let $J, x' \subseteq I$ with $x' \notin J$, and $\vec{u}$ be a open box in C_1, i.e. $u_{yc} \in C_{(y=c)}$ for $(y, c) \in O^+(J, x')$ with $u_{yc}(z = d) = u_{zd}(y = c)$. Note that for $y \neq x$, $u_{yc} = (a_{yc}, b_{yc})$ with $a_{yc} \in A_1$ and $b_{yc} \in B_1$ with $a_{yc}(x = 1) = b_{yc}(x = 0)$. We want to define $u = C_1{\uparrow}\vec{u}$. There are three cases. First, in case $x = x'$, we set $a_{x0} = u_{x0} \in C_{(x=0)} = A_{(x=0)}$; this yields an open box $\vec{a}$ in A_1 which we can fill to $a = A_1{\uparrow}\vec{a} \in A_1$. Now setting $b_{x0} = a(x = 1)$ yields an open box $\vec{b}$ in B_1 which we can fill to get $b = B_1{\uparrow}\vec{b} \in B_1$. Note that $b(x = 0) = a(x = 1)$ and thus we can set $u = (a, b)$.

Second, in case $x \neq x'$ with $x \in J$, we construct an element $v \in A_{(x=1)} = B_{(x=0)}$ first. For $(y, c) \in O^+(J - x, x')$ define $v_{yc} = a_{yc}(x = 1)$ (which is also equal to $b_{yc}(x = 0)$). It is readily checked that this defines an open box in $A_{(x=1)} = B_{(x=0)}$ and thus we get $v = A_{(x=1)}{\uparrow}\vec{v}$. Now set $a_{x1} = b_{x0} = v$; this yields open boxes $\vec{a}$ and $\vec{b}$ in A_1 and B_1, respectively. Thus we can take $u = (A_1{\uparrow}\vec{a}, B_1{\uparrow}\vec{b})$.

Finally, in case $x \notin J$, we directly have open boxes $\vec{a}$ and $\vec{b}$ in A_1 and B_1, respectively. Setting $u = (A_1{\uparrow}\vec{a}, B_1{\uparrow}\vec{b})$ gives an element in C_1 since

$$(A_1{\uparrow}\vec{a})(x = 1) = A_{(x=1)}{\uparrow}(\vec{a}(x = 1)) = B_{(x=0)}{\uparrow}(\vec{b}(x = 0)) = (B_1{\uparrow}\vec{b})(x = 0).$$

This concludes the definition of $C = \mathsf{comp}(A, B)$.

8.4 Equivalence and equality of types

We explain in this section how to transform any equivalence $\sigma\colon A \to B$ between two small Kan cubical sets to a path $A \to B$ in U, as defined in the previous section. Let us recall the notion of equivalence between types (cf. [24, Definition 4.4.1]) using informal notation. For a type A we define the proposition of being contractible $\mathsf{isContr}\, A$ to be $(\Sigma a : A)(\Pi x : A)\, \mathsf{Id}_A\ a\ x$. The *fiber* $\mathsf{fib}_\sigma\, b$ of a map $\sigma\colon A \to B$ over $b : B$ is defined as $(\Sigma x : A)\, \mathsf{Id}_B\ \mathsf{app}(\sigma, x)\ b$. A map $\sigma\colon A \to B$ is an *equivalence* if all its fibers are contractible, i.e. if

$$(\Pi b : B)\ \mathsf{isContr}(\mathsf{fib}_\sigma\, b).$$

This amounts to give $\varphi : (\Pi b : B)(\Sigma x : A)\, \mathsf{Id}_B\ \mathsf{app}(\sigma, x)\ b$ and $\psi : (\Pi b : A)(\Pi u : \mathsf{fib}_\sigma\, b)\, \mathsf{Id}_{\mathsf{fib}_\sigma\, b}\ \mathsf{app}(\varphi, b)\ u$. If we now assume that A and B are Kan cubical sets (which corresponds to types in the empty context), this definition unfolds to the following data: a map $\sigma : A \to B$ is an equivalence if there is a map $\delta : B \to A$ and a map assigning to b a line $b'\colon \sigma\delta b \to b$, and a transformation of any equality $\omega : \sigma a \to b$, where a (resp. b) is an I-cube of A (resp. B) to a "square" (really a pair of an I, x-cube of A and an I, x, y-cube of B)

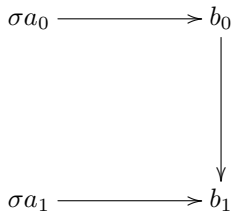
$$\begin{array}{ccc} a & \xrightarrow{\omega^*} & \delta b \end{array}$$

$$\begin{array}{ccc} \sigma a & \xrightarrow{\sigma\omega^*} & \sigma\delta b \\ {\scriptstyle\omega}\downarrow & & \downarrow{\scriptstyle b'} \\ b & \xrightarrow[b]{} & b \end{array}$$

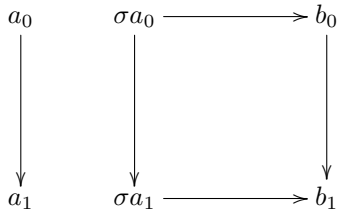
We define from this a path C between A and B in the direction x. For any substitution $f : \{x\} \to I$ we have to define a set C_f together with substitution maps $C_f \to C_{fg}$. If

$f(x) = 0$ we take $C_f = A(I)$ and if $f(x) = 1$ we take $C_f = B(I)$. If $f(x) = y$ then we define C_f to be the set of pairs (a, b) where a is an $(I - y)$-cube of A and b is an I-cube of B and $b(y = 0) = \sigma a$. It can be then be checked in an elementary way that if σ is an equivalence, then this "heterogeneous" notion of cube has the uniform Kan property.

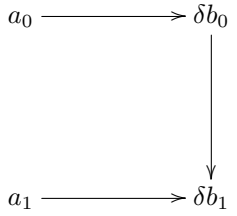
In pictures, the main difficult case is to complete an open box



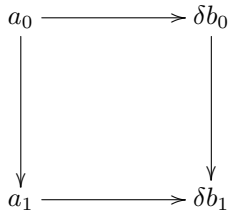
to a square



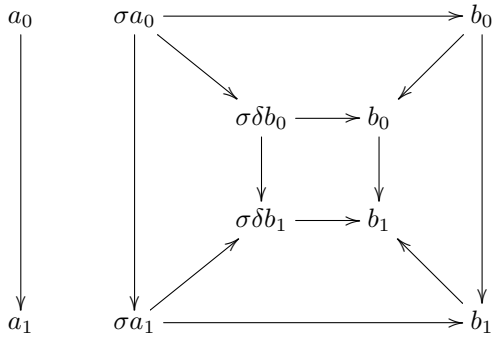
For this, using the fact that σ is an equivalence, we transform the open box in an open box in A



and since A is Kan, it can be filled to a box



and we can then fill the box in B



Since our model is constructive, this gives a way to effectively transport properties and structures on a Kan cubical set to one which is equivalent. In particular we can effectively transport properties and structures of a groupoid to one which is categorically equivalent.

We have only described here a weak corollary of the Axiom of Univalence, but the complete Axiom can be validated in this model as well[5] and will be presented in a forthcoming publication.

8.5 Propositional reflection

We can describe the operation of Kan "completion". Given a cubical set X we add operations X^+, $X\uparrow$, X^-, $X\downarrow$ in a *free* way, i.e. considering these operations as *constructors*. At the same time one defines the restrictions of the added operations, resulting in an inductive-recursive definition. The uniformity condition determine what the restrictions of these elements should. In this way we get a new cubical set Y, satisfying by definition the Kan extension property, with a map $X \to Y$. Furthermore, if Z is Kan, and we have a map $\sigma : X \to Z$ there is a map $Y \to Z$ extending σ. This map is furthermore unique if we impose it to commute with the Kan operations. In general however, the maps of Kan cubical sets do not need to commute with the Kan operations.

The same idea can be used to define $\mathsf{inh}\, X$, the *proposition* stating that X is inhabited. Besides adding constructors $(\mathsf{inh}\, X)^+$, $(\mathsf{inh}\, X)\uparrow$, $(\mathsf{inh}\, X)^-$ and $(\mathsf{inh}\, X)\downarrow$, we also add a constructor $\alpha_x(u_0, u_1)$ connecting formally along the dimension x any two I-cubes u_0 and u_1 (with x not in I) and constructors for the Kan filling and composition operations. Thus each I-cube u in $\mathsf{inh}\, X$ is of one of the forms: either u an I-cube of X; a formal Kan filling, e.g. $(\mathsf{inh}\, X)\uparrow\vec{u}$ with $\vec{u}$ an open box in $\mathsf{inh}\, X$; or of the form $\alpha_x(u_0, u_1)$ with u_i in $(\mathsf{inh}\, X)(I - x)$. At the same time we define the restrictions

$$\alpha_x(u_0, u_1)(x = 0) = u_0 \qquad \alpha_x(u_0, u_1)(x = 1) = u_1$$

and, if f is defined on x with $y = f(x)$,

$$\alpha_x(u_0, u_1)f = \alpha_y(u_0(f - x), u_1(f - x)).$$

This satisfies the required induction principle of $\mathsf{inh}\, X$: if we have a map $\varphi : X \to Y$, we can extend this to a map $\tilde{\varphi} : \mathsf{prop}\, Y \times \mathsf{inh}\, X \to Y$ where $\mathsf{prop}\, Y$ is $(\Pi y_0\ y_1 : Y)\, \mathsf{Id}_Y\ y_0\ y_1$. For $p \in (\mathsf{prop}\, Y)(I)$ and $u \in (\mathsf{inh}\, X)(I)$ we define $\tilde{\varphi}(p, u)$ in $Y(I)$ by induction on u. The difficult case is when u is $\alpha_x(u_0, u_1)$ with $x \in I$ and $u_i \in (\mathsf{inh}\, X)(I - x)$. By induction hypothesis, we already defined $v_i = \tilde{\varphi}(p(x = i), u_i) \in Y(I - x)$. Applying $p(x = 0)$ to both v_0 and v_1 gives a path $\langle x\rangle\omega$, where $\omega \in Y(I)$ connecting v_0 to v_1 along x, and we set $\tilde{\varphi}(p, u) = \omega$. Note that the choice of $p(x = 0) \in (\mathsf{prop}\, Y)(I - x)$ above is *not* canonical.

We can also define the spheres. For instance S^1 will be the Kan completion of the cubical set generated by a point base and a loop loop.

We can then define $\exists\ A\ B$ to be $\mathsf{inh}(\Sigma\ A\ B)$. If $\Sigma\ A\ B$ is a proposition we have an inhabitant of $\exists\ A\ B \to \Sigma\ A\ B$ and this can be seen as a generalization of the *axiom of description* since if A set, B proposition and B is satisfied by at most one element of A then $\Sigma\ A\ B$ is a proposition.

[5] The algorithms can be found in the implementation available at `http://github.com/simhu/cubical`.

Acknowledgments. The research for this paper has been started while the first two authors were members of the Institute for Advanced Study in Princeton, as part of the program *Univalent Foundations of Mathematics*. We are grateful for the generous support by the IAS and the Fund for Math.

The last two authors acknowledge financial support from the ERC: The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nr. 247219.

The authors wish to thank Jean-Philippe Bernardy, Cyril Cohen, Andy Pitts and Michael Shulman for stimulating discussions on the topic of this paper. The clear presentation of [27] provided an important help.

References

1 Steve Awodey and Michael Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146:45–55, 2009.

2 Bruno Barras, Thierry Coquand, and Simon Huber. A generalization of Takeuti-Gandy interpretation. To appear in Mathematical Structures in Computer Science, 2013.

3 Marc Bezem and Thierry Coquand. A Kripke model for simplicial sets. Preprint, 2013.

4 Erret Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York, 1967.

5 John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

6 Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, 2013.

7 Sjoerd Crans. *On combinatorial models for higher dimensional homotopies*. PhD thesis, Universiteit Utrecht, 1995.

8 Pierre-Louis Curien. Substitutions up to isomorphisms. *Fundamenta Informaticae*, 19:51–85, 1993.

9 Peter Dybjer. Internal type theory. In *Types for Programs and Proofs*, pages 120–134. Lecture Notes in Computer Science, Springer, 1996.

10 Alexander Grothendieck. Pursuing stacks. Manuscript, 1983.

11 Martin Hofmann. Syntax and semantics of dependent types. In A.M. Pitts and P. Dybjer, editors, *Semantics and logics of computation*, volume 14 of *Publ. Newton Inst.*, pages 79–130. Cambridge University Press, Cambridge, 1997.

12 Martin Hofmann and Thomas Streicher. Lifting Grothendieck universes. Unpublished Note.

13 Simon Huber. *A model of type theory in cubical sets*. Licentiate thesis, University of Gothenburg, 2014.

14 Witold Hurewicz. On the concept of fiber space. *Proc. Nat. Acad. Sci. U. S. A.*, 41:956–961, 1955.

15 Daniel M. Kan. Abstract homotopy. I. *Proc. Nat. Acad. Sci. U. S. A.*, 41:1092–1096, 1955.

16 Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. Preprint, `http://arxiv.org/abs/1211.2851`, 2012.

17 Per Martin-Löf. An intiutionistic theory of types: Predicative part. In H. E. Rose and J. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North–Holland, Amsterdam, 1975.

18 Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In Marc Bezem and Jan Friso Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993.

19 Andrew M. Pitts. An equivalent presentation of the Bezem-Coquand-Huber category of cubical sets. Manuscript, `http://arxiv.org/abs/1401.7807`, September 2013.

20 Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

21 Jean-Pierre Serre. *Homologie simgulière des espaces fibrés. Applications.* Thèse, Paris, 1951.

22 Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

23 Ross Street. Cosmoi of internal categories. *Trans. Amer. Math. Soc.*, 258:271–318, 1980.

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

25 Vladimir Voevodsky. The equivalence axiom and univalent models of type theory. Talk at CMU, February 2010.

26 Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 2nd edition, 1925.

27 Richard Williamson. Combinatorial homotopy theory. Preprint, 2012.

Isomorphism of "Functional" Intersection Types*

Mario Coppo, Mariangiola Dezani-Ciancaglini, Ines Margaria, and Maddalena Zacchi

Dipartimento di Informatica, Università di Torino
corso Svizzera 185, 10149 Torino, Italy
{coppo, dezani, ines, zacchi}@di.unito.it

Abstract

Type isomorphism for intersection types is quite odd, since it is not a congruence and it does not extend type equality in the standard interpretation of types. The lack of congruence is due to the proof theoretic nature of the intersection introduction rule, which requires the same term to be the subject of both premises. A partial congruence can be recovered by introducing a suitable notion of type similarity. Type equality in standard models becomes included in type isomorphism whenever atomic types have "functional" interpretations, i.e. they are equivalent to arrow types. This paper characterises type isomorphism for a type system in which the equivalence between atomic types and arrow types is induced by the initial projections of the Scott D_∞ model via the correspondence between inverse limit models and filter λ-models.

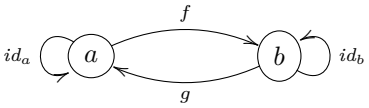
1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.3 Studies of Program Constructs, D.1.1 Applicative (Functional) Programming

Keywords and phrases type isomorphism, lambda calculus, intersection types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.129

1 Introduction

The notion of *type isomorphism* is a particularisation of the general notion of isomorphism as defined in category theory. Two objects a and b are *isomorphic* if there exist two morphisms $f : a \to b$ and $g : b \to a$ such that $f \circ g = id_b$ and $g \circ f = id_a$:



Analogously, two *types* σ and τ in some typed λ-calculus, are *isomorphic* if there are two λ-terms f and g of types $\sigma \to \tau$ and $\tau \to \sigma$, respectively, such that $f \circ g$ is $\beta\eta$ equal to the identity at type τ and $g \circ f$ is $\beta\eta$ equal to the identity at type σ.

In a recent paper [15], isomorphic types are identified. So λ-terms getting a type σ have also all types isomorphic to σ. This is useful both in looking for proofs of formulas through the Curry-Howard correspondence and in searching functions by type in program libraries.

Bruce and Longo proved in [5] that only one equation, namely the **swap** equation:

$$\sigma \to \tau \to \rho \;\approx\; \tau \to \sigma \to \rho$$

is needed for characterising isomorphism in the simply typed λ-calculus.

* This work was partially supported by EU Collaborative project ASCENS 257414, ICT COST Action IC1201 BETTY, MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.

Later, the study has been directed toward richer λ-calculi, obtained from the simply typed λ-calculus in an incremental way, by adding some other type constructors (like product [22, 4, 23]) or by allowing higher-order types (System F [5, 13]). Di Cosmo summarised in [14] the equations characterising type isomorphisms in different type systems. The set of equations grows incrementally in the sense that the set of equations for a typed λ-calculus, obtained by adding a primitive to a given typed λ-calculus, is an extension of the set of equations of the λ-calculus without that primitive.

In the presence of intersection, this incremental approach does not work, as pointed out in [12]; the isomorphism is no longer a congruence and type equality in the standard models of intersection types does not entail type isomorphism. Notice that both features hold also in the very tricky case of the sum types [17].

The lack of congruence can be shown considering, for instance, the types $\varphi_1 \to \varphi_2 \to \sigma$ and $\varphi_2 \to \varphi_1 \to \sigma$. They are isomorphic (by argument swapping), while their intersections with the same type $(\varphi_3 \to \varphi_4 \to \tau)$, i.e.

$$(\varphi_1 \to \varphi_2 \to \sigma) \wedge (\varphi_3 \to \varphi_4 \to \tau) \text{ and } (\varphi_2 \to \varphi_1 \to \sigma) \wedge (\varphi_3 \to \varphi_4 \to \tau),$$

are not. It is interesting to note that the lack of congruence prevents to give a finitary axiomatisation of the type isomorphism studied in this paper.

The standard models of intersection types map types to subsets of any domain that is a model of the untyped λ-calculus, with the conditions that the arrow is interpreted as the function space constructor and the intersection as the set-theoretic intersection [2]. For example, $\sigma \wedge \tau \to \rho$ is isomorphic (and equal in all standard models) to $\tau \wedge \sigma \to \rho$, but they are no longer isomorphic when intersected with an atomic type φ, i.e. $(\sigma \wedge \tau \to \rho) \wedge \varphi$ is not isomorphic to $(\tau \wedge \sigma \to \rho) \wedge \varphi$ (although their interpretations remain equal).

In place of congruence one can use a suitable notion of type similarity, as done in [12] for characterising isomorphism. Instead, the existence of non-isomorphic types, which are equal in all standard interpretations, reveals that the type assignment system considered in [12] can be improved. The problem is caused by the absence of a functional behaviour for atomic types. This is quite odd for the pure λ-calculus, where everything is a function.

The present paper proposes a type system whose isomorphisms contain type equality in standard intersection models. This is achieved by assuming that each atomic type is equivalent to a functional type. In particular the type system is sound for a type interpretation in which each atomic type is interpreted as the set of constant functions returning values belonging to the set itself. Notably, this choice takes inspiration from the properties of the standard projections of Scott's D_∞ λ-model [21] and from the relations between inverse limit models and filter models [6]. As proved in [6], in fact, D_∞ is isomorphic to a filter λ-model built from a set of atomic types which correspond to elements of the initial domain D_0. In this model a type interpretation in which all types have a functional character is obtained in a natural way by taking the open sets in the Scott topology.

A strongly related paper is [9], where the functional interpretation of atomic types is considered in a type system with also union types. Type similarity is extended to union types and proved to be sound for type isomorphism, while its completeness is only conjectured.

Summary. Section 2 presents the type assignment system with its properties, notably Subject Reduction and Subject Expansion. Section 3 discusses some basic isomorphisms which entail equality in standard models (Theorem 18). The main result of this paper is the characterisation of type isomorphism (Theorem 37) given in Section 5, using the type normalisation presented in Section 4. As a consequence type isomorphism turns out to be decidable (Theorem 39). Section 6 concludes with some directions for further studies.

2 Type Assignment System

Let $\mathtt{A}$ be a denumerable set of atomic types ranged over by φ, ψ and ω an atom not in $\mathtt{A}$. The syntax of types is given by:

$$\sigma \quad ::= \quad \varphi \quad | \quad \omega \quad | \quad \sigma \to \sigma \quad | \quad \sigma \wedge \sigma \,.$$

As usual, parentheses are omitted according to the precedence rule " $\wedge$ over $\to$" and $\to$ associates to the right. It is useful to distinguish between different kinds of types. So in the following:

- σ, τ, ρ, θ range over arbitrary types;
- α, β, γ range over atomic and arrow types, defined as $\alpha ::= \varphi \mid \omega \mid \sigma \to \sigma$.

The following equivalence asserts the functional character of atomic types, by equating them to arrow types. It also agrees with the interpretation of type ω as the whole domain of elements (see Definition 17).

▶ **Definition 1** (Semantic type equivalence). The *semantic equivalence relation* $\cong$ *on types* is defined as the minimal congruence such that:

$$\varphi \cong \omega \to \varphi \qquad \omega \cong \omega \to \omega \qquad \sigma \cong \sigma \wedge \omega \qquad \sigma \cong \omega \wedge \sigma.$$

The congruence allows one to state that $\sigma \cong \sigma'$ and $\tau \cong \tau'$ imply $\sigma \wedge \tau \cong \sigma' \wedge \tau'$. Moreover $\sigma \to \tau \cong \sigma' \to \tau'$ iff $\sigma \cong \sigma'$ and $\tau \cong \tau'$. Note that no other equivalence is assumed between types, for instance $\sigma \wedge \tau$ is different from $\tau \wedge \sigma$.

The equivalence of Definition 1 is dubbed semantic since it is derived by the relation between D_∞ λ-models and filter λ-models, see [1] and [3] (Section 16.3). Briefly, each inverse limit model built from an ω-algebraic lattice D_0 with order $\sqsubseteq$ is isomorphic to a filter λ-model with subtyping $\leq_\infty$ when:

- the intersections of atomic types are in one-to-one correspondence γ with the compact elements of D_0 (ω corresponds to $\bot$);
- each type corresponds to a compact element of D_∞;
- each arrow type corresponds to a step function between compact elements of D_∞;
- each intersection type corresponds to the join between compact elements of D_∞;
- the subtype relation $\leq_\infty$ mimics
  - the (reverse) partial order on the compact elements of D_0, i.e. $d, d' \in D_0$ and $d \sqsubseteq d'$ imply $\gamma^{-1}(d') \leq_\infty \gamma^{-1}(d)$, and
  - the initial projection from the compact elements of D_0 to the set of continuous functions mapping D_0 in D_0, i.e. if $d \in D_0$ is mapped to the step function $d_1 \Rightarrow d_2$, then
$$\gamma^{-1}(d) \leq_\infty \gamma^{-1}(d_1) \to \gamma^{-1}(d_2) \leq_\infty \gamma^{-1}(d).$$

The standard initial projection ι of Scott's model [21] maps each element of D_0 in the constant function returning that element, i.e. $\iota(d)$ is equal to the step function $\bot \Rightarrow d$ for all $d \in D_0$ (including $d = \bot$). It is then easy to verify that the first two equivalences of Definition 1 are induced by associating $\bot$ with type ω, by taking as D_0 the lattice obtained by join completion of a domain with a denumerable set of incomparable elements (corresponding to the types in $\mathtt{A}$) and by using the standard initial projection. The last two equivalences of Definition 1 agree with the facts that $\bot$ is the least element of D_∞ and that the intersection corresponds to the join.

In the type assignment system considered in this paper, types can be assigned only to linear λ-terms. A λ-term is *linear* if each free or bound variable occurs exactly once in it.

$$(Ax)\quad x:\sigma \vdash x:\sigma \qquad (\cong)\quad \frac{\Gamma \vdash M:\sigma \quad \sigma \cong \tau}{\Gamma \vdash M:\tau}$$

$$(\to I)\quad \frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x.M:\sigma \to \tau} \qquad (\to E)\quad \frac{\Gamma_1 \vdash M:\sigma \to \tau \quad \Gamma_2 \vdash N:\sigma}{\Gamma_1,\Gamma_2 \vdash MN:\tau}$$

$$(\wedge I)\quad \frac{\Gamma \vdash M:\sigma \quad \Gamma \vdash M:\tau}{\Gamma \vdash M:\sigma \wedge \tau} \qquad (\wedge E)\quad \frac{\Gamma \vdash M:\sigma \wedge \tau}{\Gamma \vdash M:\sigma} \qquad \frac{\Gamma \vdash M:\sigma \wedge \tau}{\Gamma \vdash M:\tau}$$

Figure 1 Typing rules.

This is justified by the observation that type isomorphisms are realised by finite hereditarily permutators which are linear λ-terms (see Definitions 10 and 12). This is not restrictive since it is easy to prove that the full system without linearity restriction [6] is conservative over the present one. Therefore the types that can be derived for the finite hereditarily permutators are the same in the two systems, so the present study of type isomorphism holds for the full system too.

Figure 1 gives the typing rules. As usual, *environments* associate variables to types and contain at most one type for each variable. The environments are relevant, i.e. they contain only the used premises. The domain of the environment Γ is denoted by $dom(\Gamma)$. When writing Γ_1,Γ_2 one convenes that $dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$. It is easy to verify that $\Gamma \vdash M:\sigma$ implies $dom(\Gamma) = FV(M)$ ($FV(M)$ denotes the set of free variables of M). An example of derivation is shown in Figure 2.

Some useful admissible rules are:

$$(L)\quad \frac{x:\sigma \vdash x:\tau \quad \Gamma, x:\tau \vdash M:\rho}{\Gamma, x:\sigma \vdash M:\rho} \qquad (\omega)\quad \frac{dom(\Gamma) = FV(M)}{\Gamma \vdash M:\omega}$$

In order to state and prove the Inversion Lemma (Lemma 4) it is handy to introduce a pre-order on types (Definition 2), which is induced by the typing rules (Lemma 3(2)).

▶ **Definition 2** (Identity pre-order on types). **1.** *The set $\mathcal{A}$ of atomic and arrow types of a type σ* (notation $\mathcal{A}(\sigma)$) is inductively defined by:

$$\mathcal{A}(\alpha) = \{\alpha, \omega\} \qquad \mathcal{A}(\sigma \wedge \tau) = \mathcal{A}(\sigma) \cup \mathcal{A}(\tau)$$

2. *The identity pre-order relation* $\precsim$ on types is defined by:

$$\sigma \precsim \tau \text{ if for all } \alpha \in \mathcal{A}(\tau) \text{ there is } \beta \in \mathcal{A}(\sigma) \text{ such that } \beta \cong \alpha.$$

It is easy to verify that $\sigma \precsim \omega$ and $\sigma \precsim \omega \to \omega$ for all types σ. Clearly, whereas $\sigma \cong \tau$ implies $\sigma \precsim \tau$, the inverse does not hold since for example $\varphi \wedge \psi \precsim \omega \to \varphi$, but $\varphi \wedge \psi \not\cong \omega \to \varphi$.

▶ **Lemma 3.** **1.** $\Gamma \vdash M:\sigma$ *iff* $\Gamma \vdash M:\alpha$ *for all* $\alpha \in \mathcal{A}(\sigma)$.
2. *If* $\Gamma \vdash M:\sigma$ *and* $\sigma \precsim \tau$, *then* $\Gamma \vdash M:\tau$.

Proof. (1). By structural induction on σ. If $\sigma = \alpha$ and $\Gamma \vdash M:\sigma$, the rule (ω) derives $\Gamma \vdash M:\omega$. Let $\sigma = \sigma_1 \wedge \sigma_2$. By rules $(\wedge I)$ and $(\wedge E)$ $\Gamma \vdash M:\sigma$ iff $\Gamma \vdash M:\sigma_1$ and $\Gamma \vdash M:\sigma_2$, so the induction hypothesis applies.
(2). By definition for all $\alpha \in \mathcal{A}(\tau)$ there is $\beta \in \mathcal{A}(\sigma)$ such that $\beta \cong \alpha$. Point (1) implies that $\Gamma \vdash M:\beta$ for all $\beta \in \mathcal{A}(\sigma)$. Then by rule $(\cong)$ $\Gamma \vdash M:\alpha$ for all $\alpha \in \mathcal{A}(\tau)$, so again by point (1) $\Gamma \vdash M:\tau$. ◀

In the following, $\bigwedge_{i\in\{1,\dots,n\}} \tau_i$ is used to denote any type obtained by multiple applications of the intersection type constructor to the types $\tau_1,\dots,\tau_n$.

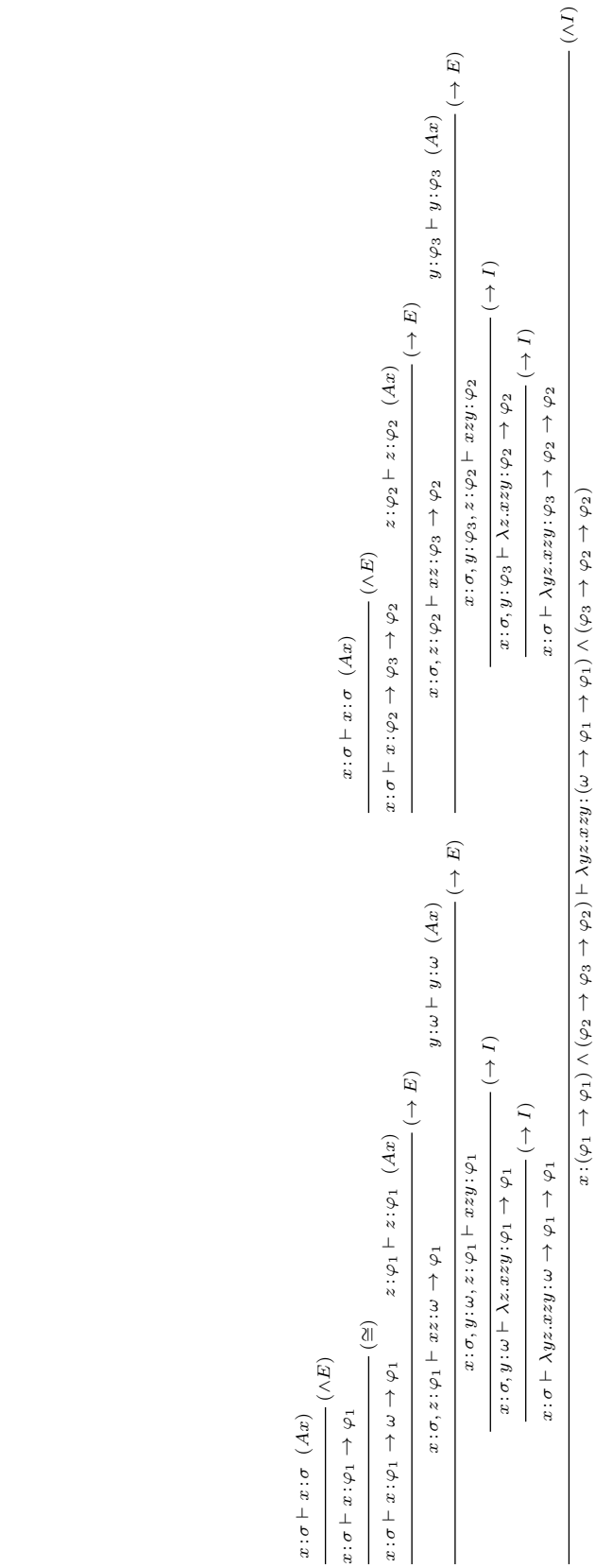


Figure 2 Derivation of $\vdash \lambda xyz.xzy:(\varphi_1 \to \varphi_1) \wedge (\varphi_2 \to \varphi_3 \to \varphi_2) \to (\omega \to \varphi_1 \to \varphi_1) \wedge (\varphi_3 \to \varphi_2 \to \varphi_2)$, where $\sigma = (\varphi_1 \to \varphi_1) \wedge (\varphi_2 \to \varphi_3 \to \varphi_2)$.

▶ **Lemma 4** (Inversion Lemma). **1.** *If* $x:\sigma \vdash x:\tau$, *then* $\sigma \precsim \tau$.
2. *If* $\Gamma \vdash \lambda x.M:\tau$ *and* $\tau \precsim \rho \to \sigma$, *then* $\Gamma, x:\rho \vdash M:\sigma$.
3. *If* $\Gamma \vdash MN:\tau$, *then there are* $\Gamma_1, \Gamma_2, \sigma_i, \tau_i \ \ (1 \leq i \leq n)$ *such that* $\Gamma = \Gamma_1, \Gamma_2$ *and* $\Gamma_1 \vdash M:\sigma_i \to \tau_i$, *and* $\Gamma_2 \vdash N:\sigma_i$ *for* $1 \leq i \leq n$ *and* $\bigwedge_{i\in\{1,\ldots,n\}} \tau_i \precsim \tau$.
4. *If* $\Gamma \vdash MN:\alpha$, *then there are* $\Gamma_1, \Gamma_2, \sigma, \tau$ *such that* $\Gamma = \Gamma_1, \Gamma_2$ *and* $\Gamma_1 \vdash M:\sigma \to \tau$, $\Gamma_2 \vdash N:\sigma$ *and* $\tau \precsim \alpha$.

Proof. Points (1), (2) and (3) are proved by induction on derivations. Only the non-standard cases are presented.
For point (1), if the last applied rule is $(\cong)$, observe that $\sigma \precsim \tau'$ and $\tau' \cong \tau$ imply $\sigma \precsim \tau$. If the last applied rule is $(\wedge I)$ or $(\wedge E)$, observe that $\sigma \precsim \tau_1$ and $\sigma \precsim \tau_2$ iff $\sigma \precsim \tau_1 \wedge \tau_2$.
For point (2), if the last applied rule is $(\cong)$, observe that $\tau' \cong \tau$ and $\tau \precsim \sigma \to \rho$ imply $\tau' \precsim \sigma \to \rho$. If the last applied rule is $(\wedge I)$ or $(\wedge E)$, observe that $\tau_1 \wedge \tau_2 \precsim \sigma \to \rho$ iff $\tau_1 \precsim \sigma \to \rho$ or $\tau_2 \precsim \sigma \to \rho$.
The proof of point (3), if the last applied rule is $(\cong)$ or $(\wedge E)$, is the same as that of point (1). If the last applied rule is $(\wedge I)$ by the induction hypothesis one has $\bigwedge_{i\in\{1,\ldots,n\}} \tau_i^{(1)} \precsim \tau_1$ and $\bigwedge_{i\in\{1,\ldots,m\}} \tau_i^{(2)} \precsim \tau_2$, which imply $(\bigwedge_{i\in\{1,\ldots,n\}} \tau_i^{(1)}) \wedge (\bigwedge_{i\in\{1,\ldots,m\}} \tau_i^{(2)}) \precsim \tau_1 \wedge \tau_2$.
Point (4) follows from point (3) and the definition of $\precsim$. In fact point (3) gives $\Gamma = \Gamma_1, \Gamma_2$ such that $\Gamma_1 \vdash M:\sigma_i \to \tau_i$, $\Gamma_2 \vdash N:\sigma_i$ and $\bigwedge_{i\in\{1,\ldots,n\}} \tau_i \precsim \tau$, for some $\Gamma_1, \Gamma_2, \sigma_i, \tau_i$ $(1 \leq i \leq n)$. In this case $\tau = \alpha$ and $\bigwedge_{i\in\{1,\ldots,n\}} \tau_i \precsim \alpha$ implies that there is $\beta \in \mathcal{A}(\bigwedge_{i\in\{1,\ldots,n\}} \tau_i) = \bigcup_{i\in\{1,\ldots,n\}} \mathcal{A}(\tau_i)$ such that $\beta \cong \alpha$. So there is an i_0 $(1 \leq i_0 \leq n)$ such that $\beta \in \mathcal{A}(\tau_{i_0})$ and $\beta \cong \alpha$, that give $\tau_{i_0} \precsim \alpha$. One can then choose $\sigma = \sigma_{i_0}$ and $\tau = \tau_{i_0}$. ◀

The following characterisation of the arrow types of the identity $\lambda x.x$ justifies the name of the pre-order relation in Definition 2.

▶ **Corollary 5.** $\vdash \lambda x.x:\sigma \to \tau$ *iff* $\sigma \precsim \tau$.

Proof. Easy from Lemmas 4(2), 4(1) and 3(2). ◀

The Inversion Lemma allows one to show some useful properties of arrow types derivable for λ-abstractions.

▶ **Lemma 6.** **1.** *If* $\Gamma \vdash \lambda x.M:\sigma \to \tau$ *and* $\Gamma \vdash \lambda x.M:\rho \to \theta$, *then* $\Gamma \vdash \lambda x.M:\sigma \wedge \rho \to \tau \wedge \theta$.
2. *If* $\Gamma \vdash \lambda x.M:\sigma \to \tau$ *and* $\Gamma \vdash \lambda x.M:\sigma \to \rho$, *then* $\Gamma \vdash \lambda x.M:\sigma \to \tau \wedge \rho$.

Proof. (1). By Lemma 4(2) $\Gamma, x:\sigma \vdash M:\tau$ and $\Gamma, x:\rho \vdash M:\theta$, which imply $\Gamma, x:\sigma \wedge \rho \vdash M:\tau$ and $\Gamma, x:\sigma \wedge \rho \vdash M:\theta$ by rules $(\wedge E)$ and (L). Rule $(\wedge I)$ derives $\Gamma, x:\sigma \wedge \rho \vdash M:\tau \wedge \theta$. Rule $(\to I)$ concludes the proof.
(2). By Lemma 4(2) and rules $(\wedge I)$, $(\to I)$. ◀

This section ends with the proofs of Subject Reduction (Theorem 8) and Subject Expansion (Theorem 9). As usual a Substitution Lemma is required.

▶ **Lemma 7** (Substitution Lemma). *If* $\Gamma, x:\sigma \vdash M:\tau$ *and* $\Gamma' \vdash N:\sigma$ *and* $dom(\Gamma) \cap dom(\Gamma') = \emptyset$, *then* $\Gamma, \Gamma' \vdash M[N/x]:\tau$.

Proof. The proof is by structural induction on M. ◀

▶ **Theorem 8** (Subject Reduction). *If* $\Gamma \vdash M:\tau$ *and* $M \longrightarrow_\beta^* N$, *then* $\Gamma \vdash N:\tau$.

Proof. It is enough to show that $\Gamma \vdash (\lambda x.M)N : \tau$ implies $\Gamma \vdash M[N/x] : \tau$. By Lemma 4(3) there are $\Gamma_1, \Gamma_2, \sigma_i, \tau_i$ $(1 \leq i \leq n)$ such that $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_1 \vdash \lambda x.M : \sigma_i \to \tau_i$, $\Gamma_2 \vdash N : \sigma_i$ for $1 \leq i \leq n$ and $\bigwedge_{i\in\{1,\ldots,n\}} \tau_i \precsim \tau$. By Lemma 4(2) $\Gamma_1, x : \sigma_i \vdash M : \tau_i$, which implies $\Gamma_1, \Gamma_2 \vdash M[N/x] : \tau_i$ by Lemma 7 for $1 \leq i \leq n$. By applications of rule $(\wedge I)$ one has $\Gamma \vdash M[N/x] : \bigwedge_{i\in\{1,\ldots,n\}} \tau_i$ and, by Lemma 3(2), one obtains $\Gamma \vdash M[N/x] : \tau$. ◀

Types are not preserved by η-reduction, for example $x : \varphi \to \varphi \vdash \lambda y.xy : \varphi \wedge \psi \to \varphi$, while $x : \varphi \to \varphi \not\vdash x : \varphi \wedge \psi \to \varphi$.

Subject expansion holds for both β and η-expansions.

▶ **Theorem 9** (Subject Expansion). *If M is a linear λ-term and $M \longrightarrow^*_{\beta\eta} N$ and $\Gamma \vdash N : \tau$, then $\Gamma \vdash M : \tau$.*

Proof. For β-expansion it is enough to show that $\Gamma \vdash M[N/x] : \tau$ implies $\Gamma \vdash (\lambda x.M)N : \tau$. The proof is by structural induction on M, observing that the linearity condition implies that there is exactly one occurrence of x in M.
For η-expansion let $\Gamma \vdash M : \tau$ and $\alpha \in \mathcal{A}(\tau)$. By Lemma 3(1) it is enough to show that $\Gamma \vdash \lambda x.Mx : \alpha$, where x is fresh. Let $\alpha \cong \sigma \to \rho$. By Lemma 3(1) and rule $(\cong)$ $\Gamma \vdash M : \sigma \to \rho$. By rules $(\to E)$ and $(\to I)$ one has $\Gamma \vdash \lambda x.Mx : \sigma \to \rho$. Rule $(\cong)$ implies $\Gamma \vdash \lambda x.Mx : \alpha$. Lemma 3(1) concludes. ◀

3 Isomorphism and Equality in Models

The study of type isomorphism in λ-calculus is based on the characterisation of λ-term invertibility. A λ-term P is *invertible* if there exists a λ-term P^{-1} such that $P \circ P^{-1} =_{\beta\eta} P^{-1} \circ P =_{\beta\eta} \lambda x.x$. The paper [11] completely characterises the invertible λ-terms in $\lambda\beta\eta$-calculus: the invertible terms are all and only the *finite hereditary permutators*.

▶ **Definition 10** (Finite Hereditary Permutator). A *finite hereditary permutator* (FHP for short) is a λ-term of the form (modulo β-conversion)

$$\lambda x y_1 \ldots y_n . x (P_1 y_{\pi(1)}) \ldots (P_n y_{\pi(n)}) \quad (n \geq 0)$$

where π is a permutation of $1, \ldots, n$, and $P_1, \ldots, P_n$ are FHPs.

Note that the identity is trivially an FHP (take $n = 0$). Another example of an FHP is

$$\lambda x y_1 y_2 . x\, y_2\, y_1 \overset{*}{\underset{\beta}{\longleftarrow}} \lambda x y_1 y_2 . x\, ((\lambda z.z)\, y_2)\, ((\lambda z.z)\, y_1),$$

which proves the swap equation. It is easy to show that FHPs are closed on composition.

▶ **Theorem 11.** *A λ -term is invertible iff it is a finite hereditary permutator.*

This result, obtained in the framework of the untyped λ-calculus, has been the basis for studying type isomorphism in different type systems for the λ-calculus. Note that every FHP has, modulo $\beta\eta$-conversion, a unique inverse P^{-1}. Even if in the type free λ-calculus FHPs are defined modulo $\beta\eta$-conversion [11], in this paper each FHP is considered only modulo β-conversion, because types are not invariant under η-reduction. Taking into account these properties, the definition of type isomorphism can be stated as follows:

▶ **Definition 12** (Type isomorphism). Two types σ and τ are isomorphic $(\sigma \approx \tau)$ if there exists a pair $< P, P^{-1} >$ of FHPs, inverse of each other, such that $\vdash P : \sigma \to \tau$ and $\vdash P^{-1} : \tau \to \sigma$. The pair $< P, P^{-1} >$ *proves* the isomorphism.

When $P = P^{-1}$ one can simply write "P proves the isomorphism".

It is immediate to verify that type isomorphism is an equivalence relation.

Clearly semantic type equivalence implies type isomorphism, i.e.

$$\sigma \cong \tau \text{ implies } \sigma \approx \tau$$

The inverse does not hold, for example $\lambda xyz.xzy$ proves $\omega \to \varphi \to \varphi \approx \varphi \to \varphi$ (note that $\varphi \to \varphi \cong \varphi \to \omega \to \varphi$), but $\omega \to \varphi \to \varphi \ncong \varphi \to \varphi$.

It is useful to consider some basic isomorphisms, which are directly related to set theoretic properties of intersection and to standard properties of functional types. It is interesting to remark that all these isomorphisms are provable equalities in the system $\mathbf{B}_+$ of relevant logic [20].

idem.	$\sigma \wedge \sigma \approx \sigma$
comm.	$\sigma \wedge \tau \approx \tau \wedge \sigma$
assoc.	$(\sigma \wedge \tau) \wedge \rho \approx \sigma \wedge (\tau \wedge \rho)$
split.	$\sigma \to \tau \wedge \rho \approx (\sigma \to \tau) \wedge (\sigma \to \rho)$

The identity $\lambda x.x$ proves the first three isomorphisms, and its η-expansion $\lambda xy.xy$ proves the fourth one.

An intersection $\sigma \wedge \tau$ is set-theoretically equal to σ if σ is included in τ. So, it is handy to introduce a pre-order on types which formalises set-theoretic inclusion taking into account the meaning of the arrow type constructor and the semantic type equivalence given in Definition 1. This pre-order is dubbed normalisation pre-order being used in the next section to define normalisation rules (Definition 19).

▶ **Definition 13** (Normalisation pre-order on types)**.** The *normalisation pre-order* $\leq$ is the pre-order relation on types defined by:

$$\sigma \leq \omega \qquad \sigma \wedge \tau \leq \sigma \quad \sigma \wedge \tau \leq \tau$$
$$\varphi \leq \sigma \to \varphi \qquad \omega \leq \sigma \to \omega$$
$$\sigma \leq \tau,\ \sigma \leq \rho \Rightarrow \sigma \leq \tau \wedge \rho \qquad \sigma' \leq \sigma,\ \tau \leq \tau' \Rightarrow \sigma \to \tau \leq \sigma' \to \tau'$$

Notice that $\sigma \leq \omega$ derives from $\sigma \wedge \omega \cong \sigma$. Moreover $\varphi \leq \sigma \to \varphi$ and $\omega \leq \sigma \to \omega$ are justified by $\varphi \cong \omega \to \varphi$, $\omega \cong \omega \to \omega$ and the contra-variance of $\leq$ for arrow types.

The identity pre-order and the normalisation pre-order are incomparable, for example $\omega \to \varphi \not\leq \varphi$, $\omega \to \varphi \not\leq \varphi$ and $\sigma \to \tau \leq \sigma \wedge \rho \to \tau$, $\sigma \to \tau \not\leq \sigma \wedge \rho \to \tau$.

The soundness of the normalisation pre-order follows from the following lemma, which shows the expected isomorphisms. This lemma uses particular forms of FHPs defined as follows.

▶ **Definition 14** (Finite Hereditary Identity)**.** A *finite hereditary identity* (FHI) is a β-normal form obtained from $\lambda x.x$ through a finite (possibly zero) number of η-expansions.

It is easy to verify that, for each FHI different from the identity, one gets

$$\mathsf{Id} \xleftarrow{*}_{\beta} \lambda xy.\mathsf{Id}_1(x(\mathsf{Id}_2 y))$$

for unique FHIs $\mathsf{Id}_1, \mathsf{Id}_2$. For example, for $\mathsf{Id} = \lambda xy_1y_2y_3.x(\lambda t.y_1t)y_2(\lambda u_1u_2.y_3u_1u_2)$ one has $\mathsf{Id}_1 = \lambda xy_2y_3.xy_2(\lambda u_1u_2.y_3u_1u_2)$ and $\mathsf{Id}_2 = \lambda xt.xt$.

▶ **Lemma 15.** 1. *Let* Id *be an FHI, then* $\vdash \mathsf{Id} : \sigma \to \sigma$ *for every type* σ.
2. *If* $\sigma \leq \tau$*, then there is an FHI* Id *such that* $\vdash \mathsf{Id} : \sigma \to \tau$.
3. *If* $\sigma \leq \tau$*, then* $\sigma \wedge \tau \approx \sigma$.

Proof. (1). The proof is trivial observing that the identity $\lambda x.x$ has type $\sigma \to \sigma$ for all σ and types are preserved by η-expansions (Theorem 9).
(2). The proof is by induction on the definition of $\leq$. Only interesting cases are considered.
If $\sigma \leq \rho$ and $\rho \leq \tau$ imply $\sigma \leq \tau$, then by the induction hypothesis there are FHIs Id_1, Id_2 such that $\vdash \mathsf{Id}_1 : \sigma \to \rho$ and $\vdash \mathsf{Id}_2 : \rho \to \tau$. This implies $\vdash \lambda x.\mathsf{Id}_2(\mathsf{Id}_1 x) : \sigma \to \tau$. It is easy to verify that $\lambda x.\mathsf{Id}_2(\mathsf{Id}_1 x)$ β-reduces to an FHI.
If $\sigma \leq \tau$ and $\sigma \leq \rho$ imply $\sigma \leq \tau \wedge \rho$, then by the induction hypothesis there are FHIs Id_1, Id_2 such that $\vdash \mathsf{Id}_1 : \sigma \to \tau$ and $\vdash \mathsf{Id}_2 : \sigma \to \rho$. By definition of FHI there is an FHI Id such that $\mathsf{Id} \longrightarrow_\eta^* \mathsf{Id}_1$ and $\mathsf{Id} \longrightarrow_\eta^* \mathsf{Id}_2$. By Subject Expansion (Theorem 9) $\vdash \mathsf{Id} : \sigma \to \tau$ and $\vdash \mathsf{Id} : \sigma \to \rho$, which imply $\vdash \mathsf{Id} : \sigma \to \tau \wedge \rho$ by Lemma 6(2).
If $\varphi \leq \sigma \to \varphi$ one can derive $y : \sigma \vdash y : \omega$ by rule (ω), and $x : \varphi \vdash x : \omega \to \varphi$ by rule $(\cong)$. Then $\vdash \lambda xy.xy : \varphi \to \sigma \to \varphi$ holds by rules $(\to E)$ and $(\to I)$.
If $\sigma' \leq \sigma$ and $\tau \leq \tau'$ imply $\sigma \to \tau \leq \sigma' \to \tau'$, then by the induction hypothesis there are FHIs Id_1, Id_2 such that $\vdash \mathsf{Id}_2 : \sigma' \to \sigma$ and $\vdash \mathsf{Id}_1 : \tau \to \tau'$. This implies

$$\vdash \lambda xy.\mathsf{Id}_1(x(\mathsf{Id}_2 y)) : (\sigma \to \tau) \to \sigma' \to \tau'$$

and $\lambda xy.\mathsf{Id}_1(x(\mathsf{Id}_2 y))$ β-reduces to an FHI.
(3). By point (2) there is an FHI Id such that $\vdash \mathsf{Id} : \sigma \to \tau$. By point (1) one has $\vdash \mathsf{Id} : \sigma \to \sigma$. Lemma 6(2) gives $\vdash \mathsf{Id} : \sigma \to \sigma \wedge \tau$. Lastly $\vdash \lambda x.x : \sigma \wedge \tau \to \sigma$. ◀

For example $\lambda xyz.xyz$ has type $(\varphi \to \varphi) \to (\varphi \to \varphi) \wedge (\varphi \to \psi \to \varphi)$. Notice that $\varphi \to \varphi \cong \varphi \to \omega \to \varphi \leq \varphi \to \psi \to \varphi$.

Lemma 15 proves the validity of the basic isomorphism:

$$\textbf{erase}. \qquad \text{if } \sigma \leq \tau \text{ then} \qquad \sigma \wedge \tau \approx \sigma$$

The following lemma assures that one can consider types modulo idempotence, commutativity, associativity, splitting and erasure in every type context $C[]$. A *type context* is defined as usual:

$$\mathcal{C}[\,] ::= [\,] \mid \mathcal{C}[\,] \to \sigma \mid \sigma \to \mathcal{C}[\,] \mid \sigma \wedge \mathcal{C}[\,] \mid \mathcal{C}[\,] \wedge \sigma$$

▶ **Lemma 16.** *If $\sigma \approx \tau$ is proved by reflexive and transitive application of the basic isomorphisms (***idem***), (***comm***), (***assoc***), (***split***), and (***erase***), then $\mathcal{C}[\sigma] \approx \mathcal{C}[\tau]$.*

Proof. As the isomorphism is reflexive and transitive, it is enough to consider the case in which $\sigma \approx \tau$ is proved by one application of (**idem**), (**comm**), (**assoc**), (**split**), and (**erase**). The proof is by structural induction on type contexts. For any context $\mathcal{C}[\,]$, an FHI $\mathsf{Id}_{\mathcal{C}[\,]}$ that proves the isomorhism $\mathcal{C}[\sigma] \approx \mathcal{C}[\tau]$ is provided.

- $\mathsf{Id}_{[\,]} = \lambda x.x$ for (**idem**), (**comm**), (**assoc**); $\mathsf{Id}_{[\,]} = \lambda xy.xy$ for (**split**); $\mathsf{Id}_{[\,]}$ is given by Lemma 15(3) for (**erase**).
- $\mathsf{Id}_{\mathcal{C}[\,]\to\rho} \overset{\beta}{\longleftarrow} \lambda xy.x(\mathsf{Id}_{\mathcal{C}[\,]} y)$.
- $\mathsf{Id}_{\rho\to\mathcal{C}[\,]} \overset{\beta}{\longleftarrow} \lambda xy.\mathsf{Id}_{\mathcal{C}[\,]}(xy)$.
- $\mathsf{Id}_{\rho\wedge\mathcal{C}[\,]} = \mathsf{Id}_{\mathcal{C}[\,]\wedge\rho} = \mathsf{Id}_{\mathcal{C}[\,]}$. ◀

For example $\lambda xy.x(\lambda zt.yzt)$ proves $(\varphi \to \varphi) \to \psi \approx (\varphi \to \varphi) \wedge (\varphi \to \psi \to \varphi) \to \psi$. In fact, $\lambda wzt.wzt$, having the type $(\varphi \to \varphi) \to (\varphi \to \varphi) \wedge (\varphi \to \psi \to \varphi)$, proves the isomorphism by erasure (Lemma 15(3)). Moreover $\mathsf{Id}_{[\,]\to\varphi} \overset{\beta}{\longleftarrow}{}^* \lambda xy.x(\mathsf{Id}_{[\,]} y) = \lambda xy.x((\lambda wzt.wzt)y)$ since the isomorphism used in the empty context is the (**erase**).

Lemma 16 justifies the notation $\bigwedge_{i\in I} \sigma_i$ with finite I, where a single atomic or arrow type is seen as an intersection (in this case I is a singleton).

The standard models of intersection types map types to subsets of any domain that is a model of the untyped λ-calculus, with the condition that the arrow is interpreted as the function space constructor and the intersection as the set-theoretic intersection. More formally using $\mathcal{P}$ to denote the power-set:

▶ **Definition 17.** Let $\mathcal{D}$ be the domain of a λ-model and $\mathcal{V} : \mathtt{A} \to \mathcal{P}(\mathcal{D})$ a mapping from atomic types to subsets of $\mathcal{D}$. The *standard interpretation of types* is given by:

$$\begin{array}{ll} [\![\varphi]\!]_{\mathcal{V}} = \mathcal{V}(\varphi) & [\![\omega]\!]_{\mathcal{V}} = \mathcal{D} \\ [\![\sigma \to \tau]\!]_{\mathcal{V}} = \{d \in \mathcal{D} \mid \forall d' \in [\![\sigma]\!]_{\mathcal{V}} :\ d \cdot d' \in [\![\tau]\!]_{\mathcal{V}}\} & [\![\sigma \wedge \tau]\!]_{\mathcal{V}} = [\![\sigma]\!]_{\mathcal{V}} \cap [\![\tau]\!]_{\mathcal{V}} \end{array}$$

The equalities corresponding to the contextual closure of the basic type isomorphisms (**idem**), (**comm**), (**assoc**), (**split**), and (**erase**), include the ones of [2], which are proved to be the equalities valid in all standard models. Therefore all types equal in all standard models are isomorphic in the system of Figure 1.

▶ **Theorem 18.** *Type* equality *in the standard models of intersection types entails type isomorphism.*

Instead, the standard type interpretation does not validate a pre-order which includes the clause $\varphi \leq \omega \to \varphi$ or $\omega \to \varphi \leq \varphi$ or both, unless the mapping from atomic types to subsets of $\mathcal{D}$ enjoys particular properties. In particular a mapping $\mathcal{V}_0$ from atomic types to subsets of $\mathcal{D}$ such that

$$\mathcal{V}_0(\varphi) = \{d \in \mathcal{D} \mid \forall d' \in \mathcal{D} :\ d \cdot d' \in \mathcal{V}_0(\varphi)\}$$

validates the semantic type equivalence given in Definition 1, i.e. it gives the same interpretation to equivalent types. If $\mathcal{D}$ is the domain of the inverse limit model discussed after Definition 1, then a valid interpretation is that of taking as $\mathcal{V}_0(\varphi)$ the set of all the elements of $\mathcal{D}$ greater than or equal to the finite element corresponding to φ. Notably $\mathcal{V}_0(\varphi)$ is an open set in the Scott topology over $\mathcal{D}$. More generally, the interpretation of each type σ is the open set of all elements of $\mathcal{D}$ greater than or equal to the finite element corresponding to σ, when mapping arrow types in step functions and intersection types in joins.

4 Normalisation

To investigate type isomorphism, following a common approach [4, 14, 12, 7, 8], a notion of *normal form* of types is introduced. *Normal type* is short for type in normal form. The notion of normal form is effective, since an algorithm to find the normal form of an arbitrary type is given.

Type normalisation rules are introduced together with the proof of their soundness.

▶ **Definition 19** (Type normalisation rules)**.** The type normalisation rules are:

$$\begin{array}{llll} (\varphi \Rightarrow) & \omega \to \varphi \Longrightarrow \varphi & (\omega \Rightarrow) & \omega \leq \sigma \text{ and } \sigma \neq \omega \text{ imply } \sigma \Longrightarrow \omega \\ (\wedge \Rightarrow) & \sigma \to \tau \wedge \rho \Longrightarrow (\sigma \to \tau) \wedge (\sigma \to \rho) & (\leq\!\Rightarrow) & \sigma \leq \tau \text{ implies } \sigma \wedge \tau \Longrightarrow \sigma \\ & & (\text{ctx} \Rightarrow) & \sigma \Longrightarrow \tau \text{ implies } \mathcal{C}[\sigma] \Longrightarrow \mathcal{C}[\tau] \end{array}$$

The first two rules follow immediately from semantic type equivalence, the following two rules correspond to the **split** and **erase** basic isomorphisms, respectively. Since $\omega \leq \sigma \to \omega$, an admissible rule is $\sigma \to \omega \Longrightarrow \omega$.

For example by rules $(\wedge \Rightarrow)$ and $(\leq\!\Rightarrow)$, taking into account that $\wedge$ is considered modulo commutativity:

$$(\varphi_1 \to \varphi_2 \wedge \varphi_3) \wedge \varphi_3 \Longrightarrow (\varphi_1 \to \varphi_2) \wedge (\varphi_1 \to \varphi_3) \wedge \varphi_3 \Longrightarrow (\varphi_1 \to \varphi_2) \wedge \varphi_3$$

since $\varphi_3 \leq \varphi_1 \to \varphi_3$.

A *normal type* ξ is either ω or a normal intersection type. A *normal intersection type* ζ is either a normal singleton type or an intersection of normal intersection types, which cannot be reduced by rule $(\leq\Rightarrow)$. A *normal singleton type* ν is either an atomic type different from ω or an arrow type from a normal intersection type to a normal singleton type, which cannot be reduced by rule $(\varphi\Rightarrow)$. Formally:

$$\xi ::= \omega \mid \zeta \qquad \zeta ::= \nu \mid \zeta \wedge \zeta \qquad \nu ::= \varphi \mid \xi \to \nu$$

where an intersection is allowed only if rule $(\leq\Rightarrow)$ cannot be applied at top level and an arrow is allowed only if rule $(\varphi\Rightarrow)$ cannot be applied at top level. So a normal type is either ω or $\bigwedge_{i\in I}\nu_i$ for some I and ν_i with $i \in I$.
For example $(\varphi \to \varphi) \wedge \psi$ is a normal type, but not a normal singleton type, while $\varphi \to \varphi$ is a normal singleton type.
The type $(\omega \to \varphi \to \varphi) \wedge \psi \to \psi$ is a normal singleton type, because $(\omega \to \varphi \to \varphi) \wedge \psi$ is a normal intersection type, being $\omega \to \varphi \to \varphi$ a normal singleton type. On the contrary $(\varphi \to \omega \to \varphi) \wedge \psi \to \psi$ is not a normal singleton type, because $\varphi \to \omega \to \varphi$ is not so.

▶ **Theorem 20** (Soundness of the normalisation rules). *If $\sigma \Longrightarrow \tau$, then there are FHIs* Id, Id' *such that* $\vdash \mathsf{Id} : \sigma \to \tau$, $\vdash \mathsf{Id}' : \tau \to \sigma$.

Proof. Rule $(\varphi\Rightarrow)$ is obtained by orienting the equivalence relation between types, so it is sound since equivalent types are shown isomorphic by the identity. Rule $(\omega\Rightarrow)$ is sound because, by Lemma 15(2), there is an FHI Id such that $\vdash \mathsf{Id} : \omega \to \sigma$, and obviously $\vdash \mathsf{Id} : \sigma \to \omega$. Rule $(\wedge\Rightarrow)$ is sound by the isomorphism (**split**). Lemma 15(3) implies the soundness of rule $(\leq\Rightarrow)$. Lemma 16 implies the soundness of rule (ctx $\Rightarrow$). ◀

For example $(\varphi_1 \to \varphi_2 \wedge \varphi_3) \wedge \varphi_3 \Longrightarrow^* (\varphi_1 \to \varphi_2) \wedge \varphi_3$ as shown before, and $\lambda xy.xy$ proves

$$(\varphi_1 \to \varphi_2 \wedge \varphi_3) \wedge \varphi_3 \;\approx\; (\varphi_1 \to \varphi_2) \wedge \varphi_3.$$

In fact, both

$$x : (\varphi_1 \to \varphi_2 \wedge \varphi_3) \wedge \varphi_3 \vdash \lambda y.xy : (\varphi_1 \to \varphi_2) \wedge \varphi_3$$

and

$$x : (\varphi_1 \to \varphi_2) \wedge \varphi_3 \vdash \lambda y.xy : (\varphi_1 \to \varphi_2 \wedge \varphi_3) \wedge \varphi_3$$

are derivable.

The following theorem shows the existence and uniqueness of the normal forms, i.e. that the normalisation rules are terminating and confluent.

▶ **Theorem 21** (Uniqueness of normal form). *The normalisation rules of Definition 19 are terminating and confluent.*

Proof. The *termination* follows from an easy adaptation of the recursive path ordering method [10]. The partial order on operators is defined by: $\to \succ \wedge$. Notice that the induced recursive path ordering $\succ^*$ has the subterm property. This solves the case of all rules but $(\wedge\Rightarrow)$. For rule $(\wedge\Rightarrow)$, since $\to \succ \wedge$, it is enough to observe that $\sigma \to \tau \wedge \rho \;\succ^*\; \sigma \to \tau$ and $\sigma \to \tau \wedge \rho \;\succ^*\; \sigma \to \rho$.
For *confluence*, thanks to the Newman Lemma [18], it is sufficient to prove the convergence of the critical pairs. Figure 3 shows the diamonds for the only three interesting cases, where $\sigma \to \tau \wedge \rho \leq \theta$, $\omega \to \varphi \leq \sigma$, $\omega \leq \sigma$, respectively. ◀

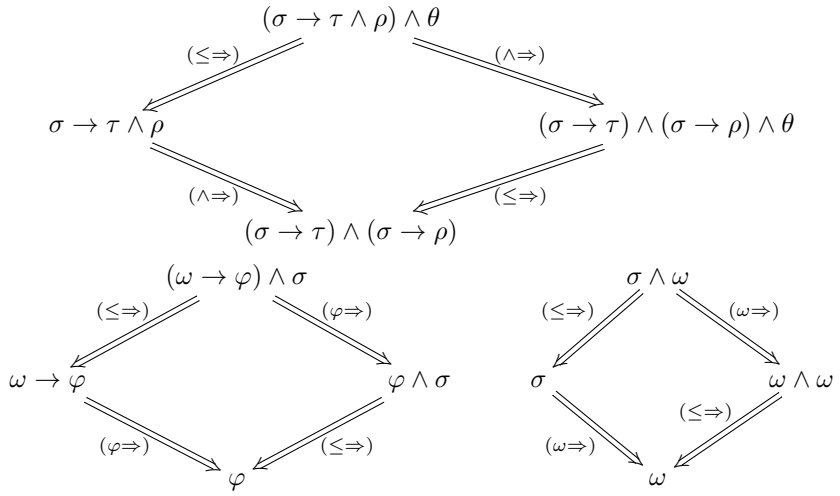


Figure 3 Critical pairs and their diamonds.

The unique (modulo idempotence, commutativity and associativity of $\wedge$) normal form of σ is denoted by $\sigma{\downarrow}$. The soundness of the normalisation rules (Theorem 20) implies that each type is isomorphic to its normal form.

▶ **Corollary 22.** $\sigma \approx \sigma{\downarrow}$.

As expected, semantic equivalent types have the same normal form. Clearly the inverse is false, since $(\sigma \to \tau \wedge \rho){\downarrow} = (\sigma \to \tau) \wedge (\sigma \to \rho)$, but $\sigma \to \tau \wedge \rho \not\cong (\sigma \to \tau) \wedge (\sigma \to \rho)$.

▶ **Lemma 23.** *If $\sigma \cong \tau$, then $\sigma{\downarrow} = \tau{\downarrow}$.*

Proof. The proof is by cases on Definition 1. For the equivalences $\varphi \cong \omega \to \varphi$ and $\omega \cong \omega \to \omega$, rules $(\varphi \Rightarrow)$ and $(\omega \Rightarrow)$ give $(\omega \to \varphi){\downarrow} = \varphi$ and $(\omega \to \omega){\downarrow} = \omega$, respectively. For the equivalences $\sigma \cong \omega \wedge \sigma$ and $\sigma \cong \sigma \wedge \omega$, rule $(\wedge \Rightarrow)$ with $\sigma \leq \omega$ gives $(\omega \wedge \sigma){\downarrow} = (\sigma \wedge \omega){\downarrow} = \sigma$. The congruence follows from the applicability of the normalisation rules in any type context. ◀

This section ends showing some properties of normal types for FHPs. The main result is that isomorphic normal types different from ω are intersections with the same number of normal singleton types, which are pairwise isomorphic (Theorem 27). Lemmas 24, 25 and 26 show preliminary results. In the following ξ, χ *range over normal types* and ν, μ, λ *range over normal singleton types.*

▶ **Lemma 24.** 1. *If $\omega \precsim \sigma \to \tau$, then $\omega \cong \tau$.*
2. *If $\mu \precsim \sigma \to \tau$ and $\tau \not\cong \omega$, then $\mu \cong \sigma \to \nu$ and $\tau \cong \nu$ for some ν.*

Proof. (1). Immediate by definition of $\precsim$ (Definition 2).
(2). By definition of $\precsim$ and of $\cong$ (Definition 1). ◀

▶ **Lemma 25.** *Let $\lambda x y_1 \dots y_n . x Q_1 \dots Q_n$ be an FHP.*
1. *If $x : \bigwedge_{i \in I} \mu_i \vdash \lambda y_1 \dots y_n . x Q_1 \dots Q_n : \bigwedge_{j \in J} \nu_j$, then for every $j \in J$ there is a $i_j \in I$ such that $x : \mu_{i_j} \vdash \lambda y_1 \dots y_n . x Q_1 \dots Q_n : \nu_j$.*
2. *If $x : \omega \vdash \lambda y_1 \dots y_n . x Q_1 \dots Q_n : \xi$, then $\xi = \omega$.*

Proof. (1). Take an arbitrary $j \in J$. Without loss of generality assume

$$\nu_j \cong \xi_1 \to \cdots \to \xi_n \to \nu.$$

This is not a restriction since $\varphi \cong \underbrace{\omega \to \cdots \to \omega}_{m} \to \varphi$ for all m. By rules $(\wedge E)$ and $(\cong)$

$$x : \bigwedge_{i\in I} \mu_i \vdash \lambda y_1 \ldots y_n.xQ_1 \ldots Q_n : \bigwedge_{j\in J} \nu_j$$

implies $x : \bigwedge_{i\in I} \mu_i \vdash \lambda y_1 \ldots y_n.xQ_1 \ldots Q_n : \xi_1 \to \cdots \to \xi_n \to \nu$. Then by Lemma 4(2) it follows

$$x : \bigwedge_{i\in I} \mu_i, y_1 : \xi_1, \ldots, y_n : \xi_n \vdash xQ_1 \ldots Q_n : \nu.$$

By repeated applications of Lemma 4(4) there are $\sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_n$ such that

$$x : \bigwedge_{i\in I} \mu_i, y_{\pi(1)} : \xi_{\pi(1)}, \ldots, y_{\pi(h-1)} : \xi_{\pi(h-1)} \vdash xQ_1 \ldots Q_{h-1} : \sigma_h \to \tau_h \text{ and}$$

$$y_{\pi(h)} : \xi_{\pi(h)} \vdash Q_h : \sigma_h,$$

where $y_{\pi(h)}$ is the head variable of Q_h for $1 \le h \le n$. Moreover $\tau_k \precsim \sigma_{k+1} \to \tau_{k+1}$ for $1 \le k \le n-1$ and $\tau_n \precsim \nu$. By Lemma 4(1) $x : \bigwedge_{i\in I} \mu_i \vdash x : \sigma_1 \to \tau_1$ implies $\bigwedge_{i\in I} \mu_i \precsim \sigma_1 \to \tau_1$. Then there is $i_j \in I$ such that $\mu_{i_j} \precsim \sigma_1 \to \tau_1$ by definition of $\precsim$. Lemma 24(2) applied to $\mu_{i_j} \precsim \sigma_1 \to \tau_1$ gives $\mu_{i_j} \cong \sigma_1 \to \nu_1$ and $\tau_1 \cong \nu_1$ for some ν_1. This together with $\tau_1 \precsim \sigma_2 \to \tau_2$ implies $\nu_1 \precsim \sigma_2 \to \tau_2$. Again by Lemma 24(2) one has $\nu_1 \cong \sigma_2 \to \nu_2$ and $\tau_2 \cong \nu_2$ for some ν_2. By iterating one gets $\nu_k \cong \sigma_{k+1} \to \nu_{k+1}$ and $\tau_{k+1} \cong \nu_{k+1}$ for some ν_{k+1} $(1 \le k \le n-1)$. Lastly $\tau_n \cong \nu_n$ and $\tau_n \precsim \nu$ imply $\nu_n \cong \nu$. Taking into account that $\nu_k \cong \sigma_{k+1} \to \nu_{k+1}$ and $\nu_{k+1} \cong \sigma_{k+2} \to \nu_{k+2}$ imply $\nu_k \cong \sigma_{k+1} \to \sigma_{k+2} \to \nu_{k+2}$, $(1 \le k \le n-2)$, one can conclude $\mu_{i_j} \cong \sigma_1 \to \cdots \to \sigma_n \to \nu$. Notice that this implies $\mu_{i_j} \not\cong \omega$ whenever $\nu_j \not\cong \omega$.
Rules $(\to E)$ and $(\to I)$ applied to $x : \mu_{i_j} \vdash x : \sigma_1 \to \cdots \to \sigma_n \to \nu$ and $y_{\pi(h)} : \xi_{\pi(h)} \vdash Q_h : \sigma_h$ for $1 \le h \le n$ derive $x : \mu_{i_j} \vdash \lambda y_1 \ldots y_n.xQ_1 \ldots Q_n : \nu_j$.
(2). Toward a contradiction assume $\xi = \bigwedge_{j\in J} \nu_j$. Let $\nu_j \cong \xi_1 \to \cdots \to \xi_n \to \nu$ for an arbitrary $j \in J$, as in the proof of point (1). One gets $\omega \precsim \sigma_1 \to \tau_1$ and

$$\tau_k \precsim \sigma_{k+1} \to \tau_{k+1} \text{ for } 1 \le k \le n-1,\ \tau_n \precsim \nu.$$

Lemma 24(1) implies $\omega \cong \nu$, which is impossible. ◀

▶ **Lemma 26.** *If* $\vdash \mathsf{Id} : \xi \to \chi$, *then* $\xi \le \chi$.

Proof. If $\chi = \omega$ the proof is trivial. If $\xi = \omega$, then $\chi = \omega$ by Lemma 25(2). Let $\xi = \bigwedge_{i\in I} \mu_i$, $\chi = \bigwedge_{j\in J} \nu_i$. By Lemma 25(1) for all $j \in J$ there is $i_j \in I$ such that $\vdash \mathsf{Id} : \mu_{i_j} \to \nu_j$. Then it is enough to show $\mu_{i_j} \le \nu_j$. The proof is by structural induction on Id. If $\mathsf{Id} = \lambda x.x$ by Lemma 4(1) $\mu_{i_j} \precsim \nu_j$; since both these types are normal singleton types, $\mu_{i_j} \cong \nu_j$, and Lemma 23 implies $\mu_{i_j} = \nu_j$. Otherwise let $\mathsf{Id} \overset{*}{\underset{\beta}{\longleftarrow}} \lambda xy.\mathsf{Id}_1(x(\mathsf{Id}_2 y))$ and $\mu_{i_j} \cong \xi' \to \mu$, $\nu_i \cong \chi' \to \nu$. By Lemma 4 $\vdash \mathsf{Id}_1 : \mu \to \nu$ and $\vdash \mathsf{Id}_2 : \chi' \to \xi'$. By the induction hypothesis $\mu \le \nu$ and $\chi' \le \xi'$, which imply $\mu_{i_j} \le \nu_j$. ◀

One can use the previous lemmas to prove that if an FHP P has the type $\bigwedge_{i\in I} \mu_i \to \bigwedge_{j\in J} \nu_j$ and its inverse P^{-1} has the type $\bigwedge_{j\in J} \nu_j \to \bigwedge_{i\in I} \mu_i$, then not only for every $j \in J$ there is a $i_j \in I$ such that $\vdash P : \mu_{i_j} \to \nu_j$, but P^{-1} precisely maps each component ν_j of the target intersection to its corresponding μ_{i_j} in the source intersection.

▶ **Theorem 27.** *If* $\bigwedge_{i\in I} \mu_i \approx \bigwedge_{j\in J} \nu_j$ *and* $< P, P^{-1} >$ *proves this isomorphism, then there is a permutation* π *between* I *and* J *such that* $< P, P^{-1} >$ *proves* $\mu_i \approx \nu_{\pi(i)}$ *for all* $i \in I$.

Proof. By Lemma 25(1), for all $j \in J$ there is $i_j \in I$ such that $\vdash P : \mu_{i_j} \to \nu_j$. Again by Lemma 25(1) there is $j' \in J$ such that $\vdash P^{-1} : \nu_{j'} \to \mu_{i_j}$. Let us suppose $j' \neq j$ towards a contradiction. One gets $x : \nu_{j'} \vdash P(P^{-1}x) : \nu_j$ and by rule $(\to I) \vdash \lambda x.P(P^{-1}x) : \nu_{j'} \to \nu_j$,

which implies that $\nu_{j'} \leq \nu_j$ by Lemma 26, since $\lambda x.P(P^{-1}x)$ β-reduces to an FHI. So $\bigwedge_{j\in J}\nu_j$ would not be a normal type, since rule $(\leq\Rightarrow)$ could be applied. ◀

5 Characterisation of Isomorphism

This section shows the main result of the paper, i.e. that two types are isomorphic iff their normal forms are "similar" (Definition 28). The basic aim of the similarity relation is that of formalising isomorphism determined by argument permutations (as in the swap equation). This relation has to take into account the fact that, for two types to be isomorphic, it is not sufficient that they coincide modulo permutations of types in the arrow sequences, as in the case of cartesian products. Indeed the same permutation must be applicable to all types in the corresponding intersections. The key notion of similarity exactly expresses such a condition.

▶ **Definition 28** (Similarity). The *similarity* relation between two sequences of normal types $\langle\xi_1,\ldots,\xi_m\rangle$ and $\langle\chi_1,\ldots,\chi_m\rangle$, written $\langle\xi_1,\ldots,\xi_m\rangle \sim \langle\chi_1,\ldots,\chi_m\rangle$, is the smallest equivalence relation such that:

1. $\langle\xi_1,\ldots,\xi_m\rangle \sim \langle\xi_1,\ldots,\xi_m\rangle$;
2. if $\langle\xi_1,\ldots,\xi_i,\xi_{i+1},\ldots,\xi_m\rangle \sim \langle\chi_1,\ldots,\chi_i,\chi_{i+1},\ldots,\chi_m\rangle$, then
$$\langle\xi_1,\ldots,(\xi_i\wedge\xi_{i+1}){\downarrow},\ldots,\xi_m\rangle \sim \langle\chi_1,\ldots,(\chi_i\wedge\chi_{i+1}){\downarrow},\ldots,\chi_m\rangle;$$
3. if $\langle\xi_i^{(1)},\ldots,\xi_i^{(m)}\rangle \sim \langle\chi_i^{(1)},\ldots,\chi_i^{(m)}\rangle$ for $1\leq i\leq n$, then
$$\langle(\xi_1^{(1)}\to\ldots\to\xi_n^{(1)}\to\nu_1){\downarrow},\ldots,(\xi_1^{(m)}\to\ldots\to\xi_n^{(m)}\to\nu_m){\downarrow}\rangle \sim$$
$$\langle(\chi_{\pi(1)}^{(1)}\to\ldots\to\chi_{\pi(n)}^{(1)}\to\nu_1){\downarrow},\ldots,(\chi_{\pi(1)}^{(m)}\to\ldots\to\chi_{\pi(n)}^{(m)}\to\nu_m){\downarrow}\rangle,$$
where π is a permutation of $1,\ldots,n$.

Similarity between normal types is trivially defined as similarity between unary sequences:
$$\xi\sim\chi \text{ if } \langle\xi\rangle\sim\langle\chi\rangle.$$

For example, from $\langle\omega\rangle\sim\langle\omega\rangle$ and $\langle\varphi\rangle\sim\langle\varphi\rangle$ one obtains, by Definition 28(3),
$$\langle(\omega\to\varphi\to\varphi){\downarrow}\rangle \sim \langle(\varphi\to\omega\to\varphi){\downarrow}\rangle,$$
that is $\omega\to\varphi\to\varphi\sim\varphi\to\varphi$. Moreover $\langle\psi,\omega\to\varphi\to\varphi\rangle\sim\langle\psi,\varphi\to\varphi\rangle$ gives
$$\psi\wedge(\omega\to\varphi\to\varphi)\sim\psi\wedge(\varphi\to\varphi).$$

The soundness of similarity can be shown without difficulties.

▶ **Theorem 29** (Soundness). *If $\langle\xi_1,\ldots,\xi_m\rangle\sim\langle\chi_1,\ldots,\chi_m\rangle$, then there is a pair of FHPs that proves $\xi_j\approx\chi_j$, for $1\leq j\leq m$.*

Proof. By induction on the definition of $\sim$ (Definition 28).
(1). $\langle\xi_1,\ldots,\xi_m\rangle\sim\langle\xi_1,\ldots,\xi_m\rangle$. The identity proves the isomorphism.
(2). $\langle\xi_1,\ldots,\xi_i,(\xi_i\wedge\xi_{i+1}){\downarrow},\ldots,\xi_m\rangle\sim\langle\chi_1,\ldots,\chi_i,(\chi_i\wedge\chi_{i+1}){\downarrow},\ldots,\chi_m\rangle$ since
$$\langle\xi_1,\ldots,\xi_i,\xi_{i+1},\ldots,\xi_m\rangle\sim\langle\chi_1,\ldots,\chi_i,\chi_{i+1},\ldots,\chi_m\rangle.$$
By the induction hypothesis there is a pair $<P,P^{-1}>$ that proves $\xi_j\approx\chi_j$, for $1\leq j\leq m$. By Lemma 6(1), the same pair proves $\xi_i\wedge\xi_{i+1}\approx\chi_i\wedge\chi_{i+1}$. By Theorem 20 there are FHIs $\mathsf{Id}_1,\mathsf{Id}_2,\mathsf{Id}'_1,\mathsf{Id}'_2$ such that $<\mathsf{Id}_1,\mathsf{Id}_2>$ proves $\xi_i\wedge\xi_{i+1}\approx(\xi_i\wedge\xi_{i+1}){\downarrow}$ and $<\mathsf{Id}'_1,\mathsf{Id}'_2>$ proves $\chi_i\wedge\chi_{i+1}\approx(\chi_i\wedge\chi_{i+1}){\downarrow}$. By Lemma 15(1) $\vdash\mathsf{Id}_\ell:\xi_j\to\xi_j$ and $\vdash\mathsf{Id}'_\ell:\chi_j\to\chi_j$ for $1\leq j\leq m$ and $1\leq\ell\leq 2$. Then the pair $<\lambda x.\mathsf{Id}'_1(P(\mathsf{Id}_2x)),\lambda x.\mathsf{Id}_1(P^{-1}(\mathsf{Id}'_2x))>$ proves the required isomorphism.

(3). $$\begin{array}{l}\langle(\xi_1^{(1)} \to \ldots \to \xi_n^{(1)} \to \nu_1)\downarrow, \ldots, (\xi_1^{(m)} \to \ldots \to \xi_n^{(m)} \to \nu_m)\downarrow\rangle \sim \\ \langle(\chi_{\pi(1)}^{(1)} \to \ldots \to \chi_{\pi(n)}^{(1)} \to \nu_1)\downarrow, \ldots, (\chi_{\pi(1)}^{(m)} \to \ldots \to \chi_{\pi(n)}^{(m)} \to \nu_m)\downarrow\rangle\end{array}$$

since $\langle \xi_i^{(1)}, \ldots, \xi_i^{(m)}\rangle \sim \langle \chi_i^{(1)}, \ldots, \chi_i^{(m)}\rangle$ for $1 \le i \le n$. By the induction hypothesis, there are pairs $< P_i, P_i^{-1} >$ proving $\xi_i^{(j)} \approx \chi_i^{(j)}$ for $1 \le j \le m$. Let

$$\begin{array}{rcl} P & = & \lambda x y_1 \ldots y_n . x (P_1^{-1} y_{\pi^{-1}(1)}) \ldots (P_n^{-1} y_{\pi^{-1}(n)}) \\ P^{-1} & = & \lambda x y_1 \ldots y_n . x (P_{\pi(1)} y_{\pi(1)}) \ldots (P_{\pi(n)} y_{\pi(n)}) \end{array}$$

It is easy to verify that

$$\begin{array}{c} \vdash P : (\xi_1^{(j)} \to \ldots \to \xi_n^{(j)} \to \mu_j) \to \chi_{\pi(1)}^{(j)} \to \ldots \to \chi_{\pi(n)}^{(j)} \to \nu_j \\ \vdash P^{-1} : (\chi_{\pi(1)}^{(j)} \to \ldots \to \chi_{\pi(n)}^{(j)} \to \nu_j) \to \xi_1^{(j)} \to \ldots \to \xi_n^{(j)} \to \mu_j \end{array}$$

for $1 \le j \le m$. Notice that

$$(\xi_1 \to \ldots \to \xi_h \to \mu)\downarrow = \begin{cases} \xi_1 \to \ldots \to \xi_k \to \mu & \text{if } \xi_{k+1} = \ldots = \xi_h = \omega \\ & \text{and } \mu \text{ is an atomic type,} \\ \xi_1 \to \ldots \to \xi_h \to \mu & \text{otherwise} \end{cases}$$

since $\xi_1, \ldots, \xi_h$ are normal types and μ is a normal singleton type. Then

$$\xi_1 \to \ldots \to \xi_h \to \mu \cong (\xi_1 \to \ldots \to \xi_h \to \mu)\downarrow,$$

and, by the typing rule ($\cong$):

$$\begin{array}{c} \vdash P : (\xi_1^{(j)} \to \ldots \to \xi_n^{(j)} \to \mu_j)\downarrow \to (\chi_{\pi(1)}^{(j)} \to \ldots \to \chi_{\pi(n)}^{(j)} \to \nu_j)\downarrow \\ \vdash P^{-1} : (\chi_{\pi(1)}^{(j)} \to \ldots \to \chi_{\pi(n)}^{(j)} \to \nu_j)\downarrow \to (\xi_1^{(j)} \to \ldots \to \xi_n^{(j)} \to \mu_j)\downarrow \end{array}$$

for $1 \le j \le m$. So $< P, P^{-1} >$ is the required pair. ◀

An immediate implication of the Soundness Theorem is that two similar types are isomorphic.

▶ **Corollary 30.** *If $\xi \sim \chi$, then $\xi \approx \chi$.*

As an example, by $\langle \omega, \varphi_1, \omega\rangle \sim \langle \omega, \varphi_1, \omega\rangle$, $\langle \varphi_2, \varphi_3, \omega\rangle \sim \langle \varphi_2, \varphi_3, \omega\rangle$, $\langle \omega, \varphi_4, \varphi_5\rangle \sim \langle \omega, \varphi_4, \varphi_5\rangle$ and the permutation $< 3, 2, 1 >$, one has:

$$\begin{array}{l} \langle \omega \to \varphi_2 \to \psi_1,\ \varphi_1 \to \varphi_3 \to \varphi_4 \to \psi_2,\ \omega \to \omega \to \varphi_5 \to \psi_3\rangle \sim \\ \langle \omega \to \varphi_2 \to \psi_1,\ \varphi_4 \to \varphi_3 \to \varphi_1 \to \psi_2,\ \varphi_5 \to \psi_3\rangle. \end{array}$$

The isomorphism between the corresponding elements of the two sequences is proved by the FHP $\lambda x y_1 y_2 y_3 . x y_3 y_2 y_1$.

As another example, by $\langle \varphi_1\rangle \sim \langle \varphi_1\rangle$, $\langle \varphi_2\rangle \sim \langle \varphi_2\rangle$, $\langle \varphi_3\rangle \sim \langle \varphi_3\rangle$, $\langle \omega\rangle \sim \langle \omega\rangle$, using the permutation $< 4, 1, 3, 2 >$, one has

$$\varphi_1 \to \varphi_2 \to \varphi_3 \to \psi \ \sim\ \omega \to \varphi_1 \to \varphi_3 \to \varphi_2 \to \psi.$$

The isomorphism is proved by the pair $< \lambda x y_1 y_2 y_3 y_4 . x y_2 y_4 y_3 y_1,\ \lambda x y_1 y_2 y_3 y_4 . x y_4 y_1 y_3 y_2 >$.

The proof of the similarity completeness, i.e. that isomorphic types have similar normal forms (Theorem 36), is based on the isomorphism characterisation given in [12]. The type system of [12] has all the rules of Figure 1, but rule ($\cong$). The isomorphism of [12] is called here *weak isomorphism* and it is denoted by $\approx^w$. The pre-order on types of [12] (*weak pre-order*) is a restriction of the present normalisation pre-order, since in [12] no equivalence between types is assumed. For example φ and $\sigma \to \varphi$ are unrelated in the weak pre-order. The rules for normalising types in [12] are the rules ($\wedge \Rightarrow$), ($\le \Rightarrow$), and (ctx $\Rightarrow$), but the application of rule (ctx $\Rightarrow$) is subject to some conditions. For example $(\sigma \to \tau \wedge \rho) \wedge \varphi$ is a normal form in [12]. The normal form of [12] is called here *weak normal form* and denoted by $\downarrow^w$.

The paper [12] defines a similarity between types, here dubbed weak similarity ($\sim^w$) that differs from similarity ($\sim$) since the semantic type equivalence $\cong$, introduced in the current type system, makes necessary to identify semantic equivalent types. The definition of similarity pays heed to that.

▶ **Definition 31** (Weak Similarity)**.** The *weak similarity* relation between two sequences of types $\langle\sigma_1,\ldots,\sigma_m\rangle$ and $\langle\tau_1,\ldots,\tau_m\rangle$, written $\langle\sigma_1,\ldots,\sigma_m\rangle \sim^w \langle\tau_1,\ldots,\tau_m\rangle$, is the smallest equivalence relation such that:

1. $\langle\sigma_1,\ldots,\sigma_m\rangle \sim^w \langle\sigma_1,\ldots,\sigma_m\rangle$;
2. if $\langle\sigma_1,\ldots,\sigma_i,\sigma_{i+1},\ldots,\sigma_m\rangle \sim^w \langle\tau_1,\ldots,\tau_i,\tau_{i+1},\ldots,\tau_m\rangle$, then
$$\langle\sigma_1,\ldots,(\sigma_i\wedge\sigma_{i+1}),\ldots,\sigma_m\rangle \sim^w \langle\tau_1,\ldots,(\tau_i\wedge\tau_{i+1}),\ldots,\tau_m\rangle;$$
3. if $\langle\sigma_i^{(1)},\ldots,\sigma_i^{(m)}\rangle \sim^w \langle\tau_i^{(1)},\ldots,\tau_i^{(m)}\rangle$ for $1\le i\le n$, then
$$\langle\sigma_1^{(1)}\to\ldots\to\sigma_n^{(1)}\to\rho_1,\ldots,\sigma_1^{(m)}\to\ldots\to\sigma_n^{(m)}\to\rho_m\rangle \sim^w$$
$$\langle\tau_{\pi(1)}^{(1)}\to\ldots\to\tau_{\pi(n)}^{(1)}\to\rho_1,\ldots,\tau_{\pi(1)}^{(m)}\to\ldots\to\tau_{\pi(n)}^{(m)}\to\rho_m\rangle,$$
where π is a permutation of $1,\ldots,n$.

Weak similarity between types is trivially defined as weak similarity between unary sequences:
$$\sigma\sim^w\tau \text{ if } \langle\sigma\rangle\sim^w\langle\tau\rangle.$$

The main difference between similarity (Definition 28) and weak similarity (Definition 31) is that the first one only relates types in normal form. As a matter of fact, similarity and weak similarity are incomparable, for example $\varphi\to\varphi\sim\omega\to\varphi\to\varphi$, $\varphi\to\varphi\not\sim^w\omega\to\varphi\to\varphi$, and
$$\varphi\to\omega\to\varphi\sim^w\omega\to\varphi\to\varphi,\ \varphi\to\omega\to\varphi\not\sim\omega\to\varphi\to\varphi,$$
since $\varphi\to\omega\to\varphi$ is not a normal type.

The characterisation of type isomorphism given in [12] can be written using the present notation as:

▶ **Theorem 32.** *$\sigma\ \approx^w\ \tau$ iff $\sigma{\downarrow}^w\sim^w\tau{\downarrow}^w$.*

In order to use this result for showing completeness (Theorem 36) it is handy to compare $\sim$ with $\sim^w$ (Lemma 34) and $\approx$ with $\approx^w$ (Lemma 35). The following auxiliary lemma can be shown by induction on the definition of $\sim$.

▶ **Lemma 33.** *If $\langle\xi_1,\ldots,\xi_i,\xi_{i+1},\ldots,\xi_m\rangle\sim\langle\chi_1,\ldots,\chi_i,\chi_{i+1},\ldots,\chi_m\rangle$, then*

1. *$\langle\xi_1,\ldots,\xi_i,\xi_i,\xi_{i+1},\ldots,\xi_m\rangle\sim\langle\chi_1,\ldots,\chi_i,\chi_i,\chi_{i+1},\ldots,\chi_m\rangle$;*
2. *$\langle\xi_1,\ldots,\xi_i,\omega,\xi_{i+1},\ldots,\xi_m\rangle\sim\langle\chi_1,\ldots,\chi_i,\omega,\chi_{i+1},\ldots,\chi_m\rangle$.*

Proof. The proof is by induction over the derivation of similarity. The only interesting case is when similarity is obtained using case 3 of Definition 28. Let
$\langle(\xi_1^{(1)}\to\ldots\to\xi_n^{(1)}\to\nu_1){\downarrow},\ldots,(\xi_1^{(i)}\to\ldots\to\xi_n^{(i)}\to\nu_i){\downarrow},\ldots,(\xi_1^{(m)}\to\ldots\to\xi_n^{(m)}\to\nu_m){\downarrow}\rangle\sim$
$\langle(\chi_{\pi(1)}^{(1)}\to\ldots\to\chi_{\pi(n)}^{(1)}\to\nu_1){\downarrow},\ldots,(\chi_{\pi(1)}^{(i)}\to\ldots\to\chi_{\pi(n)}^{(i)}\to\nu_i){\downarrow},\ldots,(\chi_{\pi(1)}^{(m)}\to\ldots\to\chi_{\pi(n)}^{(m)}\to\nu_m){\downarrow}\rangle$
since $\langle\xi_j^{(1)},\ldots,\xi_j^{(i)},\ldots,\xi_j^{(m)}\rangle\sim\langle\chi_j^{(1)},\ldots,\chi_j^{(i)},\ldots,\chi_j^{(m)}\rangle$ for $1\le j\le n$.
By the induction hypothesis $\langle\xi_j^{(1)},\ldots,\xi_j^{(i)},\xi_j^{(i)},\ldots,\xi_j^{(m)}\rangle\sim\langle\chi_j^{(1)},\ldots,\chi_j^{(i)},\chi_j^{(i)},\ldots,\chi_j^{(m)}\rangle$ for $1\le j\le n$, which imply by the same clause:
$$\langle(\xi_1^{(1)}\to\ldots\to\xi_n^{(1)}\to\nu_1){\downarrow},\ldots,\mu,\mu,\ldots,(\xi_1^{(m)}\to\ldots\to\xi_n^{(m)}\to\nu_m){\downarrow}\rangle\sim$$
$$\langle(\chi_{\pi(1)}^{(1)}\to\ldots\to\chi_{\pi(n)}^{(1)}\to\nu_1){\downarrow},\ldots,\mu',\mu',\ldots,(\chi_{\pi(1)}^{(m)}\to\ldots\to\chi_{\pi(n)}^{(m)}\to\nu_m){\downarrow}\rangle$$
and

$\langle(\xi_1^{(1)} \to \ldots \to \xi_n^{(1)} \to \nu_1)\downarrow, \ldots, \mu, (\xi_1^{(i)} \to \ldots \to \xi_n^{(i)} \to \omega)\downarrow, \ldots, (\xi_1^{(m)} \to \ldots \to \xi_n^{(m)} \to \nu_m)\downarrow\rangle \sim$
$\langle(\chi_{\pi(1)}^{(1)} \to \ldots \to \chi_{\pi(n)}^{(1)} \to \nu_1)\downarrow, \ldots, \mu', (\chi_{\pi(1)}^{(i)} \to \ldots \to \chi_{\pi(n)}^{(i)} \to \omega)\downarrow, \ldots, (\chi_{\pi(1)}^{(m)} \to \ldots \to \chi_{\pi(n)}^{(m)} \to \nu_m)\downarrow\rangle$

where $\mu = (\xi_1^{(i)} \to \ldots \to \xi_n^{(i)} \to \nu_i)\downarrow$, $\mu' = (\chi_{\pi(1)}^{(i)} \to \ldots \to \chi_{\pi(n)}^{(i)} \to \nu_i)\downarrow$. This concludes the proof by observing that $(\xi_1^{(i)} \to \ldots \to \xi_n^{(i)} \to \omega)\downarrow = (\chi_{\pi(1)}^{(i)} \to \ldots \to \chi_{\pi(n)}^{(i)} \to \omega)\downarrow = \omega$. ◀

▶ **Lemma 34.** *$\sigma \sim^w \tau$ implies $\sigma\downarrow \sim \tau\downarrow$.*

Proof. One needs to show that $\langle\sigma_1, \ldots, \sigma_m\rangle \sim^w \langle\tau_1, \ldots, \tau_m\rangle$ implies

$$\langle\sigma_1\downarrow, \ldots, \sigma_m\downarrow\rangle \sim \langle\tau_1\downarrow, \ldots, \tau_m\downarrow\rangle.$$

The proof is by induction on the definition of weak similarity.
(1). $\langle\sigma_1, \ldots, \sigma_m\rangle \sim^w \langle\sigma_1, \ldots, \sigma_m\rangle$. By case 1 of Definition 28 $\langle\sigma_1\downarrow, \ldots, \sigma_m\downarrow\rangle \sim \langle\sigma_1\downarrow, \ldots, \sigma_m\downarrow\rangle$.
(2). $\langle\sigma_1, \ldots, (\sigma_i \wedge \sigma_{i+1}), \ldots, \sigma_m\rangle \sim^w \langle\tau_1, \ldots, (\tau_i \wedge \tau_{i+1}), \ldots, \tau_m\rangle$ since
$\langle\sigma_1, \ldots, \sigma_i, \sigma_{i+1}, \ldots, \sigma_m\rangle \sim^w \langle\tau_1, \ldots, \tau_i, \tau_{i+1}, \ldots, \tau_m\rangle$. By the induction hypothesis

$$\langle\sigma_1\downarrow, \ldots, \sigma_i\downarrow, \sigma_{i+1}\downarrow, \ldots, \sigma_m\downarrow\rangle \sim \langle\tau_1\downarrow, \ldots, \tau_i\downarrow, \tau_{i+1}\downarrow, \ldots, \tau_m\downarrow\rangle.$$

This implies, by case 2 of Definition 28,

$$\langle\sigma_1\downarrow, \ldots, (\sigma_i\downarrow \wedge \sigma_{i+1}\downarrow)\downarrow, \ldots, \sigma_m\downarrow\rangle \sim \langle\tau_1\downarrow, \ldots, (\tau_i\downarrow \wedge \tau_{i+1}\downarrow)\downarrow, \ldots, \tau_m\downarrow\rangle,$$

which concludes the proof since $(\rho\downarrow \wedge \theta\downarrow)\downarrow = (\rho \wedge \theta)\downarrow$ for all ρ, θ.
(3).
$$\langle\sigma_1^{(1)} \to \ldots \to \sigma_n^{(1)} \to \rho_1, \ldots, \sigma_1^{(m)} \to \ldots \to \sigma_n^{(m)} \to \rho_m\rangle \sim^w$$
$$\langle\tau_{\pi(1)}^{(1)} \to \ldots \to \tau_{\pi(n)}^{(1)} \to \rho_1, \ldots, \tau_{\pi(1)}^{(m)} \to \ldots \to \tau_{\pi(n)}^{(m)} \to \rho_m\rangle$$

since $\langle\sigma_i^{(1)}, \ldots, \sigma_i^{(m)}\rangle \sim^w \langle\tau_i^{(1)}, \ldots, \tau_i^{(m)}\rangle$ for $1 \le i \le n$, where π is a permutation of $1, \ldots, n$.
By the induction hypothesis $\langle\sigma_i^{(1)}\downarrow, \ldots, \sigma_i^{(m)}\downarrow\rangle \sim \langle\tau_i^{(1)}\downarrow, \ldots, \tau_i^{(m)}\downarrow\rangle$ for $1 \le i \le n$.
Let $\sharp(\rho_j) = p_j$ for $1 \le j \le m$, where $\sharp(\theta) = \begin{cases} p & \text{if } \theta\downarrow = \bigwedge_{\ell\in\{1,\ldots,p\}} \nu_\ell, \\ 1 & \text{if } \theta\downarrow = \omega. \end{cases}$
Then $\rho_j\downarrow = \bigwedge_{\ell\in\{1,\ldots,p_j\}} \lambda_\ell^{(j)}$ for some $\lambda_\ell^{(j)}$ $(1 \le \ell \le p_j)$ $(1 \le j \le m)$.
By Lemma 33(1)

$$\langle\underbrace{\sigma_i^{(1)}\downarrow, \ldots, \sigma_i^{(1)}\downarrow}_{p_1}, \ldots, \underbrace{\sigma_i^{(m)}\downarrow, \ldots, \sigma_i^{(m)}\downarrow}_{p_m}\rangle \sim \langle\underbrace{\tau_i^{(1)}\downarrow, \ldots, \tau_i^{(1)}\downarrow}_{p_1}, \ldots, \underbrace{\tau_i^{(m)}\downarrow, \ldots, \tau_i^{(m)}\downarrow}_{p_m}\rangle.$$

This implies by case 3 of Definition 28

$$\langle\mu_1^{(1)}\downarrow, \ldots, \mu_{p_1}^{(1)}\downarrow, \ldots, \mu_1^{(m)}\downarrow, \ldots, \mu_{p_m}^{(m)}\downarrow\rangle \sim \langle\nu_1^{(1)}\downarrow, \ldots, \nu_{p_1}^{(1)}\downarrow, \ldots, \nu_1^{(m)}\downarrow, \ldots, \nu_{p_m}^{(m)}\downarrow\rangle$$

where $\mu_\ell^{(j)} = \sigma_1^{(j)}\downarrow \to \ldots \to \sigma_n^{(j)}\downarrow \to \lambda_\ell^{(j)}$ for $1 \le \ell \le p_j$, $\nu_\ell^{(j)} = \tau_{\pi(1)}^{(j)}\downarrow \to \ldots \to \tau_{\pi(n)}^{(j)}\downarrow \to \lambda_\ell^{(j)}$ for $1 \le \ell \le p_j$. By repeated applications of case 2 of Definition 28

$$\langle\textstyle\bigwedge_{\ell\in\{1,\ldots,p_1\}} \mu_\ell^{(1)}\downarrow, \ldots, \bigwedge_{\ell\in\{1,\ldots,p_m\}} \mu_\ell^{(m)}\downarrow\rangle \sim \langle\bigwedge_{\ell\in\{1,\ldots,p_1\}} \nu_\ell^{(1)}\downarrow, \ldots, \bigwedge_{\ell\in\{1,\ldots,p_m\}} \nu_\ell^{(m)}\downarrow\rangle$$

Notice that $\bigwedge_{\ell\in\{1,\ldots,p_j\}} \mu_\ell^{(j)}\downarrow$ and $\bigwedge_{\ell\in\{1,\ldots,p_j\}} \nu_\ell^{(j)}\downarrow$ for $1 \le j \le m$ are normal types by construction. This concludes the proof, since it is easy to verify that $(\sigma_1^{(j)} \to \ldots \to \sigma_n^{(j)} \to \rho_j)\downarrow = \bigwedge_{\ell\in\{1,\ldots,p_j\}} \mu_\ell^{(j)}\downarrow$ and $(\tau_{\pi(1)}^{(1)} \to \ldots \to \tau_{\pi(n)}^{(1)} \to \rho_1)\downarrow = \bigwedge_{\ell\in\{1,\ldots,p_j\}} \nu_\ell^{(j)}\downarrow$ for $1 \le j \le m$. ◀

▶ **Lemma 35.** *If $\xi \approx \chi$, then there are $\sigma \cong \xi$, $\tau \cong \chi$ such that $\sigma \approx^w \tau$.*

Proof. By induction on the abstraction nesting in the normal forms of P,P^{-1}, where the pair $< P, P^{-1} >$ proves the isomorphism $\xi \approx \chi$. By Lemma 25(2) and Theorem 27 either $\xi = \chi = \omega$ or $\xi = \bigwedge_{i\in I} \mu_i$, $\chi = \bigwedge_{i\in I} \nu_i$ and $\mu_i \approx \nu_i$ for all $i \in I$ (note that, since $\wedge$ is commutative, one can consider the identity permutation in Theorem 27). In the first case the proof is trivial. In the second case it is enough to show that there are $\mu_i' \cong \mu_i$, $\nu_i' \cong \nu_i$ such that $< P, P^{-1} >$ proves the isomorphism $\mu_i' \approx^w \nu_i'$ for all $i \in I$. One can assume that P, P^{-1} have the same number of initial abstractions, possibly by η-expanding (Theorem 9). Let $P = \lambda x y_1 \ldots y_n.x Q_1 \ldots Q_n$ and $P^{-1} = \lambda z t_1 \ldots t_n.z R_1 \ldots R_n$. It is easy

to verify that if $y_{\pi(j)}$ is the head variable of Q_j, then t_j is the head variable of $R_{\pi(j)}$ and $\lambda y_{\pi(j)}.Q_j$ is inverse of $\lambda t_j.R_{\pi(j)}$ for $1 \leq j \leq n$. Let $\mu_i \cong \xi_1 \to \ldots \to \xi_n \to \mu$ and $\nu_i \cong \chi_1 \to \ldots \to \chi_n \to \nu$. By Lemma 4(2) $x : \mu_i, y_1 : \chi_1, \ldots, y_n : \chi_n \vdash xQ_1 \ldots Q_n : \nu$ and $z : \nu_i, t_1 : \xi_1, \ldots, t_n : \xi_n \vdash zR_1 \ldots R_n : \mu$. By means of an argument similar to that one used in the proof of Lemma 25(1) there are $\sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_n$ such that $\mu_i \cong \sigma_1 \to \ldots \to \sigma_n \to \nu$, $\nu_i \cong \tau_1 \to \ldots \to \tau_n \to \mu$. Then $\xi_j \cong \sigma_j$, $\chi_j \cong \tau_j$ for $1 \leq j \leq n$ and $\mu \cong \nu$. Moreover $y_{\pi(j)} : \chi_{\pi(j)} \vdash Q_j : \sigma_j$ and $t_j : \xi_j \vdash R_{\pi(j)} : \tau_{\pi(j)}$ for $1 \leq j \leq n$. Therefore $y_{\pi(j)} : \chi_{\pi(j)} \vdash Q_j : \xi_j$ and $t_j : \xi_j \vdash R_{\pi(j)} : \chi_{\pi(j)}$ for $1 \leq j \leq n$. This implies $\xi_j \approx \chi_{\pi(j)}$, and then by the induction hypothesis there are $\xi'_j \cong \xi_j$, $\chi'_j \cong \chi_j$ such that $\xi'_j \approx^w \chi'_{\pi(j)}$ for $1 \leq j \leq n$. One can then choose $\mu'_i \cong \xi'_1 \to \ldots \to \xi'_n \to \mu$, $\nu'_i \cong \chi'_1 \to \ldots \to \chi'_n \to \mu$, and $\sigma = \bigwedge_{i \in I} \mu'_i$, $\tau = \bigwedge_{i \in I} \nu'_i$. ◀

▶ **Theorem 36** (Completeness). *If $\sigma \approx \tau$, then $\sigma{\downarrow} \sim \tau{\downarrow}$.*

Proof. By Corollary 22, $\sigma \approx \tau$ implies $\sigma{\downarrow} \approx \tau{\downarrow}$. So, Lemma 35 assures that there are σ', τ' such that $\sigma' \cong \sigma{\downarrow}$, $\tau' \cong \tau{\downarrow}$, and $\sigma' \approx^w \tau'$. By Theorem 32 $\sigma' \approx^w \tau'$ implies $\sigma'{\downarrow}^w \sim^w \tau'{\downarrow}^w$. Lemma 34 gives $\sigma'{\downarrow} \sim \tau'{\downarrow}$, since $(\rho{\downarrow}^w){\downarrow} = \rho{\downarrow}$ for all types ρ. Lemma 23 concludes $\sigma{\downarrow} \sim \tau{\downarrow}$. ◀

The result of the present paper is summarised in the following theorem.

▶ **Theorem 37** (Main). *Two types are isomorphic iff their normal forms are similar.*

A consequence of the Main Theorem is the decidability of type isomorphism. A last lemma shows the inverse of the Soundness Theorem.

▶ **Lemma 38.** *If there is a pair of FHPs that proves $\xi_j \approx \chi_j$ for $1 \leq j \leq m$, then*

$$\langle \xi_1, \ldots, \xi_m \rangle \sim \langle \chi_1, \ldots, \chi_m \rangle.$$

Proof. By induction on the abstraction nesting in the normal forms of P,P^{-1}, where the pair $< P, P^{-1} >$ proves the isomorphisms $\xi_j \approx \chi_j$ for $1 \leq j \leq m$. As in the proof of Lemma 35, one gets $P = \lambda x y_1 \ldots y_n . x Q_1 \ldots Q_n$ and $P^{-1} = \lambda z t_1 \ldots t_n . z R_1 \ldots R_n$, where $\lambda y_{\pi(i)}.Q_i$ is inverse of $\lambda t_i.R_{\pi(i)}$ for $1 \leq i \leq n$. By Lemmas 25(2) and 33(2) one can assume that all ξ_j, χ_j are different from ω for $1 \leq j \leq m$. By Theorem 27 and case 2 of Definition 28 one can consider that all ξ_j, χ_j are singleton types for $1 \leq j \leq m$. Let $\xi_j \cong \xi_1^{(j)} \to \ldots \xi_n^{(j)} \to \mu_j$ and $\chi_j \cong \chi_1^{(j)} \to \ldots \chi_n^{(j)} \to \nu_j$ for $1 \leq j \leq m$. As in the proof of Lemma 35 one gets $y_{\pi(i)} : \chi_{\pi(i)}^{(j)} \vdash Q_i : \xi_i^{(j)}$ and $t_i : \xi_i^{(j)} \vdash R_{\pi(i)} : \chi_{\pi(i)}^{(j)}$ for $1 \leq j \leq m$ and $1 \leq i \leq n$. By the induction hypothesis $\langle \xi_i^{(1)}, \ldots, \xi_i^{(m)} \rangle \sim \langle \chi_{\pi(i)}^{(1)}, \ldots, \chi_{\pi(i)}^{(m)} \rangle$ for $1 \leq i \leq n$, so case 3 of Definition 28 concludes the proof. ◀

▶ **Theorem 39.** *Type isomorphism is decidable.*

Proof. By Theorem 37, for deciding if two types are isomorphic it is sufficient to check if their normal forms are similar. Normal forms can be computed owing to the fact that the normalisation rules are terminating and confluent. By Definition 28, two types are similar when the unary sequences built by these types are similar, then it enough to show that similarity of type sequences is decidable. This is done by induction on the total number of symbols in the types which occur in the two sequences. Let the sequences be $\langle \xi_1, \ldots, \xi_m \rangle$ and $\langle \chi_1, \ldots, \chi_m \rangle$. Theorem 29 implies that there is a pair of FHPs that proves $\xi_i \approx \chi_i$, for $1 \leq i \leq m$. There are the following cases (leaving out the symmetric ones):

1. If one of the ξ_i is ω, then by $\xi_i \approx \chi_i$ and Lemma 25(2) χ_i must be ω and the two sequences
$$\langle \xi_1, \ldots, \xi_{i-1}, \xi_{i+1}, \ldots, \xi_m \rangle \text{ and } \langle \chi_1, \ldots, \chi_{i-1}, \chi_{i+1}, \ldots, \chi_m \rangle$$
must be similar.

2. If one of the ξ_i is an intersection $\bigwedge_{j\in\{1,\dots,n\}}\mu_j$, then by Theorems 29 and 27 χ_i must be an intersection $\bigwedge_{j\in\{1,\dots,n\}}\nu_j$, and there are a pair of FHPs and a permutation π of $\{1,\dots,n\}$ such that the pair proves $\xi_i \approx \chi_i$, for $1\le i\le m$, and $\mu_j \approx \nu_{\pi(j)}$, for $1\le j\le n$. Lemma 38 implies that the two sequences
$\langle \xi_1,\dots,\xi_{i-1},\mu_1,\dots,\mu_n,\xi_{i+1},\dots,\xi_m\rangle$ and $\langle \chi_1,\dots,\chi_{i-1},\nu_{\pi(1)},\dots,\nu_{\pi(n)},\chi_{i+1},\dots,\chi_m\rangle$
are similar. Note that the number of permutations is finite and all sequences to be checked have types with lower numbers of symbols.
3. If all types in the sequences are singleton types, let for $1\le i\le m$: $\xi_i = \xi_1^{(i)}\to\dots\xi_{p_i}^{(i)}\to\varphi_i$ and $\chi_i = \chi_1^{(i)}\to\dots\chi_{q_i}^{(i)}\to\psi_i$, and $n=\max\{p_1,\dots,p_m,q_1,\dots,q_m\}$. Let the similarity in question be obtained by cases 1 or 3 of Definition 28. Both cases prescribe $\varphi_i=\psi_i$ for $1\le i\le m$ and that there must exist a permutation π of $\{1,\dots,n\}$ such that the following similarities hold:
$$\langle\hat\xi_j^{(1)},\dots,\hat\xi_j^{(m)}\rangle \sim \langle\hat\chi_{\pi(j)}^{(1)},\dots,\hat\chi_{\pi(j)}^{(m)}\rangle \text{ for } 1\le j\le n,$$
where $\hat\xi_j^{(i)} = \begin{cases}\xi_j^{(i)} & \text{if } j\le p_i,\\ \omega & \text{otherwise.}\end{cases}$ $\hat\chi_j^{(i)} = \begin{cases}\chi_j^{(i)} & \text{if } j\le q_i,\\ \omega & \text{otherwise.}\end{cases}$
It is easy to check that any pair of the so obtained sequences has a number of symbols less than the one of the original sequence.
If instead the similarity in question is obtained by case 2 of Definition 28, one has
$$\langle\xi_1,\dots,\xi_i,\xi,\xi_{i+1},\dots,\xi_m\rangle \sim \langle\chi_1,\dots,\chi_i,\chi,\chi_{i+1},\dots,\chi_m\rangle$$
and $(\xi_i\wedge\xi)\!\downarrow = \xi_i$, $(\chi_i\wedge\chi)\!\downarrow = \chi_i$. Then one starts from the sequences obtained by removing ξ,χ and iterate this process until the similarity is obtained by cases 1 or 3 of Definition 28. ◀

Note that in the system of [12], in which only intersection types are considered, decidability is a rather immediate consequence of the decidability of type assignment for normal forms proved in [19]. This result does not seem easily extensible to the present type assignment system.

6 Conclusion

In this paper type isomorphism is studied in the setting of an intersection type system in which all types have a functional character. An equivalence relation is introduced that equates any atomic type φ to an arrow type from a distinguished atom ω to φ itself. In the derived type system all type isomorphisms related to the set theoretic properties of intersection, in particular idempotence, commutativity and associativity, are realised by λ-terms of proper type. These isomorphisms, together with other two isomorphisms that express properties of functional interpretation and inclusion of types, are preserved by every context. It follows that semantic type equality in all standard models of intersection types entails type isomorphism.

The type equivalence defined in this paper can be validated in the model D_∞ [21] by an interpretation in which each type denotes an open set in the Scott topology. One could then use the present type system to investigate the isomorphisms between open sets in D_∞. The problem of finding a model which validates all and only the type isomorphisms studied in this paper remains open.

We plan to study type isomorphism in other theories of intersection and union types, in particular in the theories providing models of the call-by-value λ-calculus. An interesting observation is that, with the typing rules given in [16] for the type constant ν, all intersections of arrow types ending by ν are isomorphic to ν. In fact the rule $\Gamma\vdash\lambda x.M:\nu$ allows one to

derive both

$$x:\nu \vdash \lambda y_1 \dots y_m.xy_1 \dots y_m : \sigma_1 \to \dots \to \sigma_n \to \nu$$
$$\text{and } x:\sigma_1 \to \dots \to \sigma_n \to \nu \vdash \lambda y.xy : \nu,$$

for any $n \leq m$ and arbitrary $\sigma_1, \dots, \sigma_n$. Notably, the type theory of [16] gives a model of the call-by-value λ-calculus.

Acknowledgements. We gratefully acknowledge the anonymous referees for their careful reading of our paper and their many useful remarks and suggestions. In particular Lemma 35 strongly improved. We thank Alejandro Díaz-Caro for his useful advices.

References

1 Fabio Alessi, Mariangiola Dezani-Ciancaglini, and Furio Honsell. Inverse limit models as filter models. In D. Kesner, F. van Raamsdonk, and J. Wells, editors, *Higher-Order Rewriting*, pages 3–25. RWTH Aachen, 2004.

2 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.

3 Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge, 2013.

4 Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

5 Kim Bruce and Giuseppe Longo. Provable isomorphisms and domain equations in models of typed languages. In R. Sedgewick, editor, *Symposium on the Theory of Computing*, pages 263–272. ACM Press, 1985.

6 Mario Coppo, Mariangiola Dezani-Ciancaglini, Furio Honsell, and Giuseppe Longo. Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium*, pages 241–262. North-Holland, 1984.

7 Mario Coppo, Mariangiola Dezani-Ciancaglini, Ines Margaria, and Maddalena Zacchi. Towards isomorphism of intersection and union types. In S. Graham-Lengrand and L. Paolini, editors, *Intersection Types and Related Systems*, volume 121 of *EPTCS*, pages 58–80, 2013.

8 Mario Coppo, Mariangiola Dezani-Ciancaglini, Ines Margaria, and Maddalena Zacchi. Isomorphism of intersection and union types. *Mathematical Structures in Computer Science*, 2014. To appear.

9 Mario Coppo, Mariangiola Dezani-Ciancaglini, Ines Margaria, and Maddalena Zacchi. On isomorphism of "functional" intersection and union types. In J. Rehof, editor, *Intersection Types and Related Systems*, EPTCS, 2014. To appear.

10 Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

11 Mariangiola Dezani-Ciancaglini. Characterisation of normal forms possessing an inverse in the $\lambda\beta\eta$-calculus. *Theoretical Computer Science*, 2(3):323–337, 1976.

12 Mariangiola Dezani-Ciancaglini, Roberto Di Cosmo, Elio Giovannetti, and Makoto Tatsuta. On isomorphisms of intersection types. *ACM Transactions on Computational Logic*, 11(4):1–22, 2010.

13 Roberto Di Cosmo. Second order isomorphic types. A proof theoretic study on second order λ-calculus with surjective pairing and terminal object. *Information and Computation*, 119(2):176–201, 1995.

14 Roberto Di Cosmo. A short survey of isomorphisms of types. *Mathematical Structures in Computer Science*, 15:825–838, 2005.

15 Alejandro Díaz-Caro and Gilles Dowek. Simply typed lambda-calculus modulo type isomorphisms. `https://who.rocq.inria.fr/Alejandro.Diaz-Caro/stmti.pdf`, 2014.

16 Lavinia Egidi, Furio Honsell, and Simona Ronchi Della Rocca. Operational, Denotational and Logical Descriptions: a Case Study. *Fundamenta Informaticae*, 16(1):149–169, 1992.

17 Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1–2):35–50, 2006.

18 Maxwell H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

19 Simona Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59(1–2):1–29, 1988.

20 Richard Routley and Robert K. Meyer. The semantics of entailment III. *Journal of Philosophical Logic*, 1:192–208, 1972.

21 Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 97–136. Springer-Verlag, 1972.

22 Sergei Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983. English translation of the original paper in Russian published in Zapiski Nauchnych Seminarov LOMI, v.105, 1981.

23 Sergei Soloviev. A complete axiom system for isomorphism of types in closed categories. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 360–371. Springer-Verlag, 1993.

A Hybrid Linear Logic for Constrained Transition Systems

Joëlle Despeyroux[1] and Kaustuv Chaudhuri[2]

1 **INRIA and CNRS, I3S, Sophia-Antipolis, France**
joelle.despeyroux@inria.fr
2 **INRIA, France**
kaustuv.chaudhuri@inria.fr

Abstract

Linear implication can represent state transitions, but real transition systems operate under temporal, stochastic or probabilistic constraints that are not directly representable in ordinary linear logic. We propose a general modal extension of intuitionistic linear logic where logical truth is indexed by constraints and hybrid connectives combine constraint reasoning with logical reasoning. The logic has a focused cut-free sequent calculus that can be used to internalize the rules of particular constrained transition systems; we illustrate this with an adequate encoding of the synchronous stochastic pi-calculus.

1998 ACM Subject Classification F.4.1. Mathematical Logic, F.1.2. Modes of Computation

Keywords and phrases linear logic, hybrid logic, stochastic pi-calculus, focusing, adequacy

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.150

1 Introduction

To reason about state transition systems, we need a logic of state. Linear logic [21] is such a logic and has been successfully used to model such diverse systems as process calculi [25], references and concurrency in programming languages [38], and formal security [7, 8], to give a few examples. Linear logic achieves this versatility by representing propositions as *resources* that are combined using $\otimes$, which can then be transformed using the linear implication ($\multimap$). However, linear implication is timeless: there is no way to correlate two concurrent transitions. If resources have lifetimes and state changes have temporal, probabilistic or stochastic *constraints*, then the logic will allow inferences that may not be realizable in the system being modelled. The need for formal reasoning in such constrained systems has led to the creation of specialized formalisms such as Computation Tree Logic (*CTL*)[18], Continuous Stochastic Logic (*CSL*) [2] or Probabilistic CTL (*PCTL*) [22]. These approaches pay a considerable encoding overhead for the states and transitions in exchange for the constraint reasoning not provided by linear logic. A prominent alternative to the logical approach is to use a suitably enriched process algebra such as the stochastic and probabilistic π-calculi or the κ-calculus [14]. Processes are animated by means of simulation and then compared with the observations. Process calculi do not however completely fill the need for *formal reasoning for constrained transition systems.*

We propose a simple yet general method to add constraint reasoning to linear logic. It is an old idea—*labelled deduction* [37] with *hybrid* connectives [6]—applied to a new domain. Precisely, we parameterize ordinary logical truth on a *constraint domain*: $A@w$ stands for the truth of A under constraint w. Only a basic monoidal structure is assumed about the constraints from a proof-theoretic standpoint. We then use the hybrid connectives of

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 150–168

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

satisfaction (`at`) and *localization* ($\downarrow$) to perform generic symbolic reasoning on the constraints at the propositional level. We call the result *hybrid linear logic* (HyLL); it has a generic cut-free (but cut admitting) sequent calculus that can be strengthened with a focusing restriction [1] to obtain a normal form for proofs. Any instance of HyLL that gives a semantic interpretation to the constraints enjoys these proof-theoretic properties.

Focusing allows us to treat HyLL as a *logical framework* for constrained transition systems. Logical frameworks with hybrid connectives have been considered before; hybrid *LF* (*HLF*), for example, is a generic mechanism to add many different kinds of resource-awareness, including linearity, to ordinary *LF* [33]. *HLF* follows the usual *LF* methodology of keeping the logic of the framework minimal: its proof objects are β-normal η-long natural deduction terms, but the equational theory of such terms is sensitive to permutative equivalences [39]. With a focused sequent calculus, we have more direct access to a canonical representation of proofs, so we can enrich the framework with any connectives that obey the focusing discipline. The representational adequacy of an encoding in terms of (partial) focused sequent derivations tends to be more straightforward than in a natural deduction formulation. We illustrate this by encoding the synchronous stochastic π-calculus ($S\pi$) in HyLL using rate functions as constraints.

In addition to the novel stochastic component, our encoding of $S\pi$ is a conceptual improvement over other encodings of π-calculi in linear logic. In particular, we perform a full propositional reflection of processes as in [25], but our encoding is first-order and adequate as in [9]. HyLL does not itself prescribe an operational semantics for the encoding of processes; thus, bisimilarity in continuous time Markov chains (*CTMC*) is not the same as logical equivalence in stochastic HyLL, unlike in *CSL* [16]. This is not a deficiency; rather, the *combination* of focused HyLL proofs and a proof search strategy tailored to a particular encoding is necessary to produce faithful symbolic executions. This exactly mirrors $S\pi$ where it is the simulation rather than the transitions in the process calculus that is shown to be faithful to the *CTMC* semantics [29].

This work has the following main contributions. First is the logic HyLL itself and its associated proof-theory, which has a very standard and well understood design. Second, we show how to obtain many different instances of HyLL for particular constraint domains because we only assume a basic monoidal structure for constraints. Third, we illustrate the use of focused sequent derivations to obtain adequate encodings by giving a novel adequate encoding of $S\pi$. Our encoding is, in fact, *fully adequate*, *i.e.*, partial focused proofs are in bijection with traces. The ability to encode $S\pi$ gives an indication of the versatility of HyLL.

This paper is organized as follows: in Sec. 2, we present the inference (sequent calculus) rules for HyLL and describe the two main semantic instances: temporal and probabilistic constraints. In Sec. 3 we sketch the general focusing restriction on HyLL sequent proofs. In Sec. 4 we give the encoding of $S\pi$ in probabilistic HyLL, and show that the encoding is representationally adequate for focused proofs (theorems 16 and 18). We end with an overview of related (Sec. 5) and future work (Sec. 6). The full version of this paper is available as a technical report [12].

2 Hybrid Linear Logic

In this section we define HyLL, a conservative extension of intuitionistic first-order linear logic (ILL) [21] where the truth judgements are labelled by worlds representing constraints. Like in ILL, propositions are interpreted as *resources* which may be composed into a *state* using the usual linear connectives, and the linear implication ($\multimap$) denotes a transition

between states. The world label of a judgement represents a constraint on states and state transitions; particular choices for the worlds produce particular instances of HyLL. The common component in all the instances of HyLL is the proof theory, which we fix once and for all. We impose the following minimal requirement on the kinds of constraints that HyLL can deal with.

▶ **Definition 1.** A *constraint domain* $\mathcal{W}$ is a monoid structure $\langle W, \cdot, \iota \rangle$. The elements of W are called *worlds*, and the partial order $\preceq : W \times W$—defined as $u \preceq w$ if there exists $v \in W$ such that $u \cdot v = w$—is the *reachability relation* in $\mathcal{W}$.

The identity world ι is $\preceq$-initial and is intended to represent the lack of any constraints. Thus, the ordinary ILL is embeddable into any instance of HyLL by setting all world labels to the identity. When needed to disambiguate, the instance of HyLL for the constraint domain $\mathcal{W}$ will be written HyLL($\mathcal{W}$). Two design choices are important to note. First, we only require the worlds to be monoids, not lattices, because this lets us give a sufficiently general system that it can be instantiated with rate functions as the constraint domain. Second, we do not assume that the monoid is commutative so that we can still choose to use lattices for the constraint domain.

Atomic propositions are written using lowercase letters $(a, b, \ldots)$ applied to a sequence of *terms* $(s, t, \ldots)$, which are drawn from an untyped term language containing term variables $(x, y, \ldots)$ and function symbols $(f, g, \ldots)$ applied to a list of terms. Non-atomic propositions are constructed from the connectives of first-order intuitionistic linear logic and the two hybrid connectives *satisfaction* (`at`), which states that a proposition is true at a given world $(w, u, v, \ldots)$, and *localization* $(\downarrow)$, which binds a name for the (current) world the proposition is true at. The following grammar summarizes the syntax of HyLL propositions.

$$\begin{aligned} A, B, \ldots ::= {} & a\ \vec{t} \mid A \otimes B \mid \mathbf{1} \mid A \multimap B \mid A \mathbin{\&} B \mid \top \mid A \oplus B \mid \mathbf{0} \mid !\, A \mid \forall x.\, A \mid \exists x.\, A \\ \mid {} & (A\ \mathtt{at}\ w) \mid \downarrow u.\, A \mid \forall u.\, A \mid \exists u.\, A \end{aligned}$$

Note that in the propositions $\downarrow u.\, A$, $\forall u.\, A$ and $\exists u.\, A$, world u is bound in A. World variables cannot be used in terms, and neither can term variables occur in worlds; this restriction is important for the modular design of HyLL because it keeps purely logical truth separate from constraint truth. We let α range over variables of either kind. As we shall prove later (Theorem 5), the $\downarrow$ connective commutes with every propositional connective, including itself. That is, $\downarrow u.\, (A * B)$ is equivalent to $(\downarrow u.\, A) * (\downarrow u.\, B)$ for all binary connectives $*$, and $\downarrow u.\, * A$ is equivalent to $*(\downarrow u.\, A)$ for every unary connective $*$, assuming the commutation will not cause an unsound capture of u. It is purely a matter of taste where to place the $\downarrow$, and repetitions are harmless.

The unrestricted connectives $\wedge$, $\vee$, $\supset$, *etc.* of intuitionistic (non-linear) logic can also be defined in terms of the linear connectives and the exponential ! using any of the available embeddings of intuitionistic logic into linear logic, such as Girard's embedding [21]. For example, $A \supset B$ can be defined as $!\, A \multimap B$.

2.1 Sequent Calculus for HyLL

In this section, we give a sequent calculus presentation of HyLL and prove a cut-admissibility theorem. The sequent formulation in turn will lead to an analysis of the polarities of the connectives in order to get a focused sequent calculus that can be used to compile a logical theory into a system of derived inference rules with nice properties (Sec. 3). For instance, if a given theory defines a transition system, then the derived rules of the focused calculus will

exactly exhibit the same transitions. This is key to obtain the necessary representational adequacy theorems, as we shall see for the $S\pi$-calculus example chosen in this paper (Sec. 4.1).

We start with the judgements from linear logic [21] and enrich them with a modal situated truth. We present the syntax of hybrid linear logic in a sequent calculus style, using Martin-Löf's principle of separating judgements and logical connectives. Instead of the ordinary mathematical judgement "A is true", for a proposition A, judgements of HyLL are of the form "A is true at world w", abbreviated as $A@w$. We use sequents of the form $\Gamma\ ;\ \Delta \implies C@w$ where Γ and Δ are sets of judgements of the form $A@w$, with Δ being moreover a *multiset*. Γ is called the *unrestricted context*: its hypotheses can be consumed any number of times. Δ is a *linear context*: every hypothesis in it must be consumed singly in the proof. Note that in a judgement $A@w$ (as in a proposition $A\ \mathtt{at}\ w$), w can be any expression in $\mathcal{W}$, not only a variable. The notation $A[\tau/\alpha]$ stands for the replacement of all free occurrences of the variable α in A with the expression τ, avoiding capture. Note that the expressions in the rules are to be read up to alpha-conversion.

The full collection of rules of the HyLL sequent calculus is in Fig. 1. The rules for the linear connectives are borrowed from [11] where they are discussed at length, so we omit a more thorough discussion here. The rules for the first-order quantifiers are completely standard. A brief discussion of the hybrid rules follows. To introduce the *satisfaction* proposition $(A\ \mathtt{at}\ u)$ (at any world v') on the right, the proposition A must be true in the world u. The proposition $(A\ \mathtt{at}\ u)$ itself is then true at any world, not just in the world u. In other words, $(A\ \mathtt{at}\ u)$ carries with it the world at which it is true. Therefore, suppose we know that $(A\ \mathtt{at}\ u)$ is true (at any world v); then, we also know that $A@u$. The other hybrid connective of *localisation*, $\downarrow$, is intended to be able to name the current world. That is, if $\downarrow u.\, A$ is true at world w, then the variable u stands for w in the body A. This interpretation is reflected in its introduction rule on the right $\downarrow R$. For left introduction, suppose we have a proof of $\downarrow u.\, A@v$ for some world v. Then, we also know $[v/u]A@v$.

There are only two structural rules: the init rule infers an atomic initial sequent, and the copy rule introduces a contracted copy of an unrestricted assumption into the linear context (reading from conclusion to premise). Weakening and contraction are admissible rules:

▶ **Theorem 2** (Structural Properties).
1. *If* $\Gamma\ ;\ \Delta \Longrightarrow C@w$, *then* $\Gamma, \Gamma'\ ;\ \Delta \Longrightarrow C@w$. *(weakening)*
2. *If* $\Gamma, A@u, A@u\ ;\ \Delta \Longrightarrow C@w$, *then* $\Gamma, A@u\ ;\ \Delta \Longrightarrow C@w$. *(contraction)*

Proof. By straightforward structural induction on the given derivations. ◀

The most important structural properties are the admissibility of the identity and the cut principles. The identity theorem is the general case of the init rule and serves as a global syntactic completeness theorem for the logic. Dually, the cut theorem below establishes the syntactic soundness of the calculus; moreover there is no cut-free derivation of $\cdot\ ;\ \cdot \Longrightarrow \mathbf{0}@w$, so the logic is also globally consistent.

▶ **Theorem 3** (Identity). $\Gamma\ ;\ A@w \Longrightarrow A@w$.

Proof. By induction on the structure of A (see [12]). ◀

▶ **Theorem 4** (Cut).
1. *If* $\Gamma\ ;\ \Delta \Longrightarrow A@u$ *and* $\Gamma\ ;\ \Delta', A@u \Longrightarrow C@w$, *then* $\Gamma\ ;\ \Delta, \Delta' \Longrightarrow C@w$.
2. *If* $\Gamma\ ;\ \cdot \Longrightarrow A@u$ *and* $\Gamma, A@u\ ;\ \Delta \Longrightarrow C@w$, *then* $\Gamma\ ;\ \Delta \Longrightarrow C@w$.

Judgemental rules

$$\frac{}{\Gamma\ ;\ a\ \vec{t}\ @u \Longrightarrow a\ \vec{t}\ @u}\ \text{init} \qquad \frac{\Gamma, A@u\ ;\ \Delta, A@u \Longrightarrow C@w}{\Gamma, A@u\ ;\ \Delta \Longrightarrow C@w}\ \text{copy}$$

Multiplicatives

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow A@w \quad \Gamma\ ;\ \Delta' \Longrightarrow B@w}{\Gamma\ ;\ \Delta, \Delta' \Longrightarrow A \otimes B@w}\ \otimes R \qquad \frac{\Gamma\ ;\ \Delta, A@u, B@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, A \otimes B@u \Longrightarrow C@w}\ \otimes L$$

$$\frac{}{\Gamma\ ;\ \cdot \Longrightarrow \mathbf{1}@w}\ \mathbf{1}R \qquad \frac{\Gamma\ ;\ \Delta \Longrightarrow C@w}{\Gamma\ ;\ \Delta, \mathbf{1}@u \Longrightarrow C@w}\ \mathbf{1}L$$

$$\frac{\Gamma\ ;\ \Delta, A@w \Longrightarrow B@w}{\Gamma\ ;\ \Delta \Longrightarrow A \multimap B@w}\ \multimap R \qquad \frac{\Gamma\ ;\ \Delta \Longrightarrow A@u \quad \Gamma\ ;\ \Delta', B@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, \Delta', A \multimap B@u \Longrightarrow C@w}\ \multimap L$$

Additives

$$\frac{}{\Gamma\ ;\ \Delta \Longrightarrow \top@w}\ \top R \qquad \frac{}{\Gamma\ ;\ \Delta, \mathbf{0}@u \Longrightarrow C@w}\ \mathbf{0}L$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow A@w \quad \Gamma\ ;\ \Delta \Longrightarrow B@w}{\Gamma\ ;\ \Delta \Longrightarrow A\ \&\ B@w}\ \&R \qquad \frac{\Gamma\ ;\ \Delta, A_i@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, A_1\ \&\ A_2@u \Longrightarrow C@w}\ \&L_i$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow A_i@w}{\Gamma\ ;\ \Delta \Longrightarrow A_1 \oplus A_2@w}\ \oplus R_i \qquad \frac{\Gamma\ ;\ \Delta, A@u \Longrightarrow C@w \quad \Gamma\ ;\ \Delta, B@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, A \oplus B@u \Longrightarrow C@w}\ \oplus L$$

Quantifiers

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow A@w}{\Gamma\ ;\ \Delta \Longrightarrow \forall\alpha.\ A@w}\ \forall R^{\alpha} \qquad \frac{\Gamma\ ;\ \Delta, [\tau/\alpha]A@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, \forall\alpha.\ A@u \Longrightarrow C@w}\ \forall L$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow [\tau/\alpha]A@w}{\Gamma\ ;\ \Delta \Longrightarrow \exists\alpha.\ A@w}\ \exists R \qquad \frac{\Gamma\ ;\ \Delta, A@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, \exists\alpha.\ A@u \Longrightarrow C@w}\ \exists L^{\alpha}$$

For $\forall R^{\alpha}$ and $\exists L^{\alpha}$, α is assumed to be fresh with respect to the conclusion. For $\exists R$ and $\forall L$, τ stands for a term or world, as appropriate.

Exponentials

$$\frac{\Gamma\ ;\ \cdot \Longrightarrow A@w}{\Gamma\ ;\ \cdot \Longrightarrow\ !A@w}\ !R \qquad \frac{\Gamma, A@u\ ;\ \Delta \Longrightarrow C@w}{\Gamma\ ;\ \Delta,\ !A@u \Longrightarrow C@w}\ !L$$

Hybrid connectives

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow A@u}{\Gamma\ ;\ \Delta \Longrightarrow (A\ \mathtt{at}\ u)@v}\ \mathtt{at}R \qquad \frac{\Gamma\ ;\ \Delta, A@u \Longrightarrow C@w}{\Gamma\ ;\ \Delta, (A\ \mathtt{at}\ u)@v \Longrightarrow C@w}\ \mathtt{at}L$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow [w/u]A@w}{\Gamma\ ;\ \Delta \Longrightarrow \downarrow u.\ A@w}\ \downarrow R \qquad \frac{\Gamma\ ;\ \Delta, [v/u]A@v \Longrightarrow C@w}{\Gamma\ ;\ \Delta, \downarrow u.\ A@v \Longrightarrow C@w}\ \downarrow L$$

Figure 1 The sequent calculus for HyLL.

Proof. By lexicographic structural induction on the given derivations, with cuts of kind 2 additionally allowed to justify cuts of kind 1. The style of proof sometimes goes by the name of *structural cut-elimination* [11]. See [12] for the details. ◀

We can use the admissible cut rules to show that the following rules are invertible: $\otimes L$, $\mathbf{1}L$, $\oplus L$, $\mathbf{0}L$, $\exists L$, $\multimap R$, $\& R$, $\top R$, and $\forall R$. In addition, the four hybrid rules, $\mathtt{at}R$, $\mathtt{at}L$, $\downarrow R$ and $\downarrow L$ are invertible. In fact, $\downarrow$ and $\mathtt{at}$ commute freely with all non-hybrid connectives:

▶ **Theorem 5** (Invertibility). *The following rules are invertible:*
1. *On the right:* $\& R$, $\top R$, $\multimap R$, $\forall R$, $\downarrow R$ *and* $\mathtt{at}R$;
2. *On the left:* $\otimes L$, $\mathbf{1}L$, $\oplus L$, $\mathbf{0}L$, $\exists L$, $!L$, $\downarrow L$ *and* $\mathtt{at}L$. ◀

▶ **Corollary 6** (Consistency). *There is no proof of* $\cdot\ ;\ \cdot \Longrightarrow \mathbf{0}\,@w$. ◀

Proof. A straightforward consequence of Thm. 4. ◀

HyLL is conservative with respect to ordinary intuitionistic linear logic: as long as no hybrid connectives are used, the proofs in HyLL are identical to those in ILL [11]. The proof (omitted) is by simple structural induction.

▶ **Theorem 7** (Conservativity). *Call a proposition or multiset of propositions* pure *if it contains no instance of the hybrid connectives and no instance of quantification over a world variable, and let* Γ, Δ *and* A *be pure. Then, If* $\Gamma\ ;\ \Delta \Longrightarrow_{HyLL} C\,@w$ *is derivable, then so is* $\Gamma\ ;\ \Delta \Longrightarrow_{ILL} C$. ◀

In the rest of this paper we use the following derived connectives:

▶ **Definition 8** (Modal Connectives).

$$\Box\, A \triangleq \downarrow u.\, \forall w.\, (A\ \mathtt{at}\ u \cdot w) \qquad \Diamond\, A \triangleq \downarrow u.\, \exists w.\, (A\ \mathtt{at}\ u \cdot w)$$
$$\rho_v\, A \triangleq \downarrow u.\, (A\ \mathtt{at}\ u \cdot v) \qquad \dagger\, A \triangleq \forall u.\, (A\ \mathtt{at}\ u)$$

The connective ρ represents a form of delay. Note its derived right rule:

$$\frac{\Gamma\ ;\ \Delta \vdash A@w \cdot v}{\Gamma\ ;\ \Delta \vdash \rho_v\, A@w}\ \rho R\,.$$

The proposition $\rho_v\, A$ thus stands for an *intermediate state* in a transition to A. Informally it can be thought to be "v before A"; thus, $\Box\, A = \forall v.\ \rho_v\, A$ represents *all* intermediate states in the path to A, and $\Diamond\, A = \exists v.\ \rho_v\, A$ represents *some* such state. The modally unrestricted proposition $\dagger\, A$ represents a resource that is consumable in any world; it is intended to be used to make the transition rules available everywhere.

It is worth remarking that the reachability relation in HyLL is trivial: every world that can be defined is reachable from every other. To illustrate, the (linear form of the) axioms of the S5 modal logic are derivable in HyLL; in particular, the sequent $\cdot\ ;\ \Diamond\, A@w \Longrightarrow \Box \Diamond\, A@w$, which represents the 5 axiom, is provable. HyLL is, in fact, more expressive than S5 as it allows direct manipulation of the worlds using the hybrid connectives: for example, the ρ connective is not definable in S5.

2.2 Temporal Constraints

As a pedagogical example, consider the constraint domain $\mathcal{T} = \langle \mathbf{R}^+, +, 0 \rangle$ representing instants of time. This domain can be used to define the lifetime of resources, such as keys, sessions, or delegations of authority. Delay (Defn. 8) in HyLL($\mathcal{T}$) represents intervals of

time; $\rho_d\, A$ means "A will become available after delay d", similar to metric tense logic [32]. This domain is very permissive because addition is commutative, resulting in the equivalence of $\rho_u\, \rho_v\, A$ and $\rho_v\, \rho_u\, A$. The "forward-looking" connectives G (always in the future) and F (sometimes in the future) of ordinary tense logic are precisely $\Box$ and $\Diamond$ of Defn. 8.

In addition to the future connectives, the domain $\mathcal{T}$ also admits past connectives if we add saturating subtraction (*i.e.*, $a - b = 0$ if $b \geq a$) to the language of worlds. We can then define the duals H (historically) and O (once) of G and F as:

$$\mathrm{H}\ \mathrm{A} \triangleq \downarrow u.\, \forall w.\, (A\ \mathtt{at}\ u - w) \qquad \mathrm{O}\ \mathrm{A} \triangleq \downarrow u.\, \exists w.\, (A\ \mathtt{at}\ u - w)$$

While this domain does not have any branching structure like CTL, it is expressive enough for many common idioms because of the branching structure of derivations involving $\oplus$. CTL reachability ("in some path in some future"), for instance, is the same as our $\Diamond$; similarly CTL steadiness ("in some path for all futures") is the same as $\Box$. CTL stability ("in all paths in all futures"), however, has no direct correspondance in HyLL (see however [15] for a correspondance in particular cases). Note that model checking cannot cope with temporal expresssions involving the "in all paths" notion anyway. [1]

On the other hand, the availability of linear reasoning, enriched with modalities, makes certain kinds of reasoning in HyLL much more natural than in ordinary temporal logics. One important example is of *oscillation* between states in systems with kinetic feedback. In a temporal specification language such as BIOCHAM [10], only aperiodic oscillations are representable, while in HyLL an oscillation between A and B with delay d is represented by the rule $\dagger(A \multimap \rho_d\, B)\ \&\ (B \multimap \rho_d\, A)$ (or $\dagger(A \multimap \Diamond\, B)\ \&\ (B \multimap \Diamond\, A)$ if the oscillation is aperiodic). If HyLL($\mathcal{T}$) were extended with constrained implication and conjunction in the style of CILL [36] or η [17], then we can define localized versions of $\Box$ and $\Diamond$, such as "A is true everywhere/somewhere in an interval".

For examples of applications of HyLL with temporal constraints, the interested reader can see [15], which gives an encoding of a simple biological system and its temporal properties in HyLL($\mathcal{T}'$), where $\mathcal{T}' = \langle \mathbb{N}, +, 0 \rangle$ represents discrete instants of time. We will, instead, use a version of HyLL dedicated to continuous time Markov Chains with exponential distribution, as used in Sπ. We introduce this type of constraints below.

2.3 Probabilistic Constraints

Transitions in practice rarely have precise delays. Phenomenological and experimental evidence is used to construct a probabilistic model of the transition system where the delays are specified as probability distributions of continuous (or discrete) variables. A number of variations of monoids representing probabilistic and stochastic constraints are presented in [12], both for the general case and for the special case of Markov processes.

One of the standard models of stochastic transition systems is continuous time Markov chains (CTMCs) where the delays of transitions between states are distributed according to the Markov assumption of memorylessness (Markov processes) with the further condition that their state-spaces are countable [35]. In the synchronous stochastic π-calculus (Sπ), the probability of a reaction with *rate* r is given by continuous time Markov chains with exponential distribution of parameter r (See [31]). To describe such processes, we shall take $\mathbf{R}^+$ to represent the *rates* of their exponential distribution. To encode the Sπ calculus in a suitable instanciation of HyLL, we only need a *symbolic* operation on the rates. This abstract

[1] at least in their full generality, involving an infinite number of states.

treatment can be made fully precise, but this would require a detour into measure theory that is beyond the scope of this paper; see [12] for the details.

▶ **Definition 9.** The *rates domain* $\mathcal{R}$ is the monoid $\mathcal{R} = \langle \mathbf{R}^{+*}, \cdot, [] \rangle$ of lists of positive reals, where $\cdot$ is concatenation of lists, and [] is the empty list.

Worlds $r \in \mathcal{R}$ represent the (rates of the) *sequence* of actions that have led to the current world from a given fixed initial world. We might equivalently have chosen $\mathcal{T} = \langle \mathbf{R}^{+}, +, 0 \rangle$ to represent average time delays, which would be the sum of the reciprocals of the rates in the list of rates. We choose to use lists of rates because they are more informative than average time delays. Note that since our rate functions are assumed to be memoryless, the order of the list of rates is immaterial, so we can easily relax it to a multi-set of rates; this change could not substantially alter the development of this paper.

3 Focusing

As HyLL is intended to represent transition systems adequately, it is crucial that HyLL derivations in the image of an encoding have corresponding transitions. However, transition systems are generally specified as rewrite algebras over an underlying congruence relation. These congruences have to be encoded propositionally in HyLL, so a HyLL derivation will generally require several inference rules to implement a single transition; moreover, several trivially different reorderings of these "micro" inferences would correspond to the same transition. It is therefore futile to attempt to define an operational semantics directly on HyLL inferences.

We restrict the syntax to focused derivations [1], which ignores many irrelevant rule permutations in a sequent proof and divides the proof into clear *phases* that define the grain of atomicity. The logical connectives are divided into two classes, *negative* and *positive*, and rule permutations for connectives of like polarity are confined to *phases*. A *focused derivation* is one in which the positive and negative rules are applied in alternate maximal phases in the following way: in the *active* phase, all negative rules are applied (in irrelevant order) until no further negative rule can apply; the phase then switches and one positive proposition is selected for *focus*; this focused proposition is decomposed under focus (*i.e.*, the focus persists to its sub-formulas) until it becomes negative, and the phase switches again.

As noted before, the logical rules of the hybrid connectives $\mathtt{at}$ and $\downarrow$ are invertible, so they can be considered to have both polarities. It would be valid to decide a polarity for each occurrence of each hybrid connective independently; however, as they are mainly intended for book-keeping during logical reasoning, we define the polarity of these connectives in the following *parasitic* form: if its immediate subformula is positive (resp. negative) connective, then it is itself positive (resp. negative). These connectives therefore become invisible to focusing. This choice of polarity can be seen as a particular instance of a general scheme that divides the $\downarrow$ and $\mathtt{at}$ connectives into two polarized forms each. To complete the picture, we also assign a polarity for the atomic propositions; this restricts the shape of focusing phases further [13]. The full syntax of positive ($P, Q, \ldots$) and negative ($M, N, \ldots$) propositions is as follows:

$$\begin{array}{lll} P, Q, \ldots & ::= & p\ \vec{t} \mid P \otimes Q \mid \mathbf{1} \mid P \oplus Q \mid \mathbf{0} \mid !\, N \mid \exists \alpha.\ P \mid \downarrow u.\ P \mid (P\ \mathtt{at}\ w) \mid \Downarrow N \\ N, M, \ldots & ::= & n\ \vec{t} \mid N\ \&\ N \mid \top \mid P \multimap N \mid \forall \alpha.\ N \mid \downarrow u.\ N \mid (N\ \mathtt{at}\ w) \mid \Uparrow P \end{array}$$

The two syntactic classes refer to each other via the new *shift* connectives $\Uparrow$ and $\Downarrow$. Sequents in the focusing calculus are of the following forms.

Focused logical rules

$$\frac{}{\Gamma\ ;\ \left[n\ \vec{t}@w\right] \Longrightarrow \Downarrow n\ \vec{t}@w}\ \mathrm{li} \qquad \frac{\Gamma\ ;\ \Delta\ ;\ P@u \Longrightarrow \cdot\ ;\ Q@w}{\Gamma\ ;\ \Delta\ ;\ \left[\Uparrow P@u\right] \Longrightarrow Q@w}\ \Uparrow L \qquad \frac{\Gamma\ ;\ \Delta\ ;\ \cdot \Longrightarrow N@w\ ;\ \cdot}{\Gamma\ ;\ \Delta \Longrightarrow \left[\Downarrow N@w\right]}\ \Downarrow R$$

$$\frac{\Gamma\ ;\ \Delta\ ;\ \left[N_i@u\right] \Longrightarrow Q@w}{\Gamma\ ;\ \Delta\ ;\ \left[N_1\ \&\ N_2@u\right] \Longrightarrow Q@w}\ \&L_i \qquad \frac{\Gamma\ ;\ \Delta \Longrightarrow \left[P@u\right] \quad \Gamma\ ;\ \Xi\ ;\ \left[N@u\right] \Longrightarrow Q@w}{\Gamma\ ;\ \Delta, \Xi\ ;\ \left[P \multimap N@u\right] \Longrightarrow Q@w}\ \multimap L$$

$$\frac{\Gamma\ ;\ \Delta\ ;\ \left[[\tau/\alpha]N@u\right] \Longrightarrow Q@w}{\Gamma\ ;\ \Delta\ ;\ \left[\forall\alpha.\ N@u\right] \Longrightarrow Q@w}\ \forall L \qquad \frac{\Gamma\ ;\ \Delta\ ;\ \left[[v/u]N@v\right] \Longrightarrow Q@w}{\Gamma\ ;\ \Delta\ ;\ \left[\downarrow u.\ N@v\right] \Longrightarrow Q@w}\ \downarrow LF$$

$$\frac{\Gamma\ ;\ \Delta\ ;\ \left[N@u\right] \Longrightarrow Q@w}{\Gamma\ ;\ \Delta\ ;\ \left[(N\ \mathtt{at}\ u)@v\right] \Longrightarrow Q@w}\ \mathtt{at}LF \qquad \frac{}{\Gamma\ ;\ \Uparrow p\ \vec{t}@w \Longrightarrow \left[p\ \vec{t}@w\right]}\ \mathrm{ri}$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow \left[P@w\right] \quad \Gamma\ ;\ \Xi \Longrightarrow \left[Q@w\right]}{\Gamma\ ;\ \Delta, \Xi \Longrightarrow \left[P \otimes Q@w\right]}\ \otimes R \qquad \frac{\Gamma\ ;\ \Delta \Longrightarrow \left[P_i@w\right]}{\Gamma\ ;\ \Delta \Longrightarrow \left[P_1 \oplus P_2@w\right]}\ \oplus R_i$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow \left[[\tau/\alpha]P@w\right]}{\Gamma\ ;\ \Delta \Longrightarrow \left[\exists\alpha.\ P@w\right]}\ \exists R \qquad \frac{\Gamma\ ;\ \cdot\ ;\ \cdot \Longrightarrow N@w\ ;\ \cdot}{\Gamma\ ;\ \cdot \Longrightarrow \left[!N\right]@w}\ !R \qquad \frac{\Gamma\ ;\ \Delta \Longrightarrow \left[[w/u]P@w\right]}{\Gamma\ ;\ \Delta \Longrightarrow \left[\downarrow u.\ P@w\right]}\ \downarrow RF$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow \left[P@u\right]}{\Gamma\ ;\ \Delta \Longrightarrow \left[(P\ \mathtt{at}\ u)@w\right]}\ \mathtt{at}RF \qquad \frac{}{\Gamma\ ;\ \cdot \Longrightarrow \left[\mathbf{1}@w\right]}\ \mathbf{1}R$$

Active logical rules

(R of the form $\cdot\ ;\ Q@w$ or $N@w\ ;\ \cdot$, and L of the form $\Gamma\ ;\ \Delta\ ;\ \Omega$)

$$\frac{\mathsf{L}, P@u, Q@u \Longrightarrow \mathsf{R}}{\mathsf{L}, P \otimes Q@u \Longrightarrow \mathsf{R}}\ \otimes L \qquad \frac{\mathsf{L} \Longrightarrow \mathsf{R}}{\mathsf{L}, \mathbf{1}@u \Longrightarrow \mathsf{R}}\ \mathbf{1}L \qquad \frac{\mathsf{L}, P@u \Longrightarrow \mathsf{R} \quad \mathsf{L}, Q@u \Longrightarrow \mathsf{R}}{\mathsf{L}, P \oplus Q@u \Longrightarrow \mathsf{R}}\ \oplus L$$

$$\frac{\mathsf{L}, [v/u]P@v \Longrightarrow \mathsf{R}}{\mathsf{L}, \downarrow u.\ P@v \Longrightarrow \mathsf{R}}\ \downarrow LA \qquad \frac{\mathsf{L}, P@u \Longrightarrow \mathsf{R}}{\mathsf{L}, (P\ \mathtt{at}\ u)@v \Longrightarrow \mathsf{R}}\ \mathtt{at}LA \qquad \frac{\mathsf{L}, P@u \Longrightarrow \mathsf{R}}{\mathsf{L}, \exists\alpha.\ P@u \Longrightarrow \mathsf{R}}\ \exists L^{\alpha}$$

$$\frac{\Gamma, N@u\ ;\ \Delta\ ;\ \Omega \Longrightarrow \mathsf{R}}{\Gamma\ ;\ \Delta\ ;\ \Omega, !N@u \Longrightarrow \mathsf{R}}\ !L \qquad \frac{\Gamma\ ;\ \Delta, N@w\ ;\ \Omega \Longrightarrow \mathsf{R}}{\Gamma\ ;\ \Delta\ ;\ \Omega, \Downarrow N@w \Longrightarrow \mathsf{R}}\ \Downarrow L \qquad \frac{\Gamma\ ;\ \Delta, \Uparrow p\ \vec{t}\ ;\ \Omega \Longrightarrow \mathsf{R}}{\Gamma\ ;\ \Delta\ ;\ \Omega, p\ \vec{t}@w \Longrightarrow \mathsf{R}}\ \mathrm{lp}$$

$$\frac{\mathsf{L} \Longrightarrow M@w\ ;\ \cdot \quad \mathsf{L} \Longrightarrow N@w\ ;\ \cdot}{\mathsf{L} \Longrightarrow M\ \&\ N@w\ ;\ \cdot}\ \&R \qquad \frac{}{\mathsf{L} \Longrightarrow \top@w\ ;\ \cdot}\ \top R \qquad \frac{\mathsf{L}, P@w \Longrightarrow N@w\ ;\ \cdot}{\mathsf{L} \Longrightarrow P \multimap N@w\ ;\ \cdot}\ \multimap R$$

$$\frac{\mathsf{L} \Longrightarrow [w/u]N@w\ ;\ \cdot}{\mathsf{L} \Longrightarrow \downarrow u.\ N@w\ ;\ \cdot}\ \downarrow RA \qquad \frac{\mathsf{L} \Longrightarrow N@u}{\mathsf{L} \Longrightarrow (N\ \mathtt{at}\ u)@w}\ \mathtt{at}RA \qquad \frac{\mathsf{L} \Longrightarrow N@u\ ;\ \cdot}{\mathsf{L} \Longrightarrow \forall\alpha.\ N@u\ ;\ \cdot}\ \forall R^{\alpha}$$

$$\frac{\mathsf{L} \Longrightarrow \cdot\ ;\ P@w}{\mathsf{L} \Longrightarrow \Uparrow P@w\ ;\ \cdot}\ \Uparrow R \qquad \frac{\mathsf{L} \Longrightarrow \cdot\ ;\ \Downarrow n\ \vec{t}@w}{\mathsf{L} \Longrightarrow n\ \vec{t}@w\ ;\ \cdot}\ \mathrm{rp} \qquad \frac{}{\mathsf{L}, \mathbf{0}@u \Longrightarrow \mathsf{R}}\ \mathbf{0}L$$

Focusing decisions

$$\frac{\Gamma\ ;\ \Delta\ ;\ \left[N@u\right] \Longrightarrow Q@w \quad N \text{ not } \Uparrow p\ \vec{t}}{\Gamma\ ;\ \Delta, N@u\ ;\ \cdot \Longrightarrow \cdot\ ;\ Q@w}\ \mathrm{lf} \qquad \frac{\Gamma, N@u\ ;\ \Delta\ ;\ \left[N@u\right] \Longrightarrow Q@w}{\Gamma, N@u\ ;\ \Delta\ ;\ \cdot \Longrightarrow \cdot\ ;\ Q@w}\ \mathrm{cplf}$$

$$\frac{\Gamma\ ;\ \Delta \Longrightarrow \left[P@w\right] \quad P \text{ not } \Downarrow n\ \vec{t}}{\Gamma\ ;\ \Delta\ ;\ \cdot \Longrightarrow \cdot\ ;\ P@w}\ \mathrm{rf}$$

Figure 2 Focusing rules for HyLL.

$$\left.\begin{array}{l}\Gamma\ ;\ \Delta\ ;\ \Omega \Longrightarrow \cdot\ ;\ P @ w \\ \Gamma\ ;\ \Delta\ ;\ \Omega \Longrightarrow N @ w\ ;\ \cdot\end{array}\right\}\ \text{active} \qquad \left.\begin{array}{l}\Gamma\ ;\ \Delta\ ;\ [N @ u] \Longrightarrow P @ w \\ \Gamma\ ;\ \Delta \Longrightarrow [P @ w]\end{array}\right\}\ \text{focused}$$

In each case, Γ and Δ contain only negative propositions (*i.e.*, of the form $N @ u$) and Ω only positive propositions (*i.e.*, of the form $P @ u$). The full collection of inference rules are in Fig. 2. The sequent form $\Gamma\ ;\ \Delta\ ;\ \cdot \Longrightarrow \cdot\ ;\ P @ w$ is called a *neutral sequent*; from such a sequent, a left or right focused sequent is produced with the rules lf, cplf or rf. Focused logical rules are applied (non-deterministically) and focus persists unto the subformulas of the focused proposition as long as they are of the same polarity; when the polarity switches, the result is an active sequent, where the propositions in the innermost zones are decomposed in an irrelevant order until once again a neutral sequent results.

Soundness of the focusing calculus with respect to the ordinary sequent calculus is immediate by simple structural induction. In each case, if we forget the additional structure in the focused derivations, then we obtain simply an unfocused proof. We omit the obvious theorem. Completeness, on the other hand, is a hard result. We omit the proof because focusing is by now well known for linear logic, with a number of distinct proofs via focused cut-elimination (see *e.g.* the detailed proof in [13]). The hybrid connectives pose no problems because they allow all cut-permutations.

▶ **Theorem 10** (Focusing Completeness). *Let Γ^- and $C^- @ w$ be negative polarizations of Γ and $C @ w$ (that is, adding $\Uparrow$ and $\Downarrow$ to make C and each proposition in Γ negative) and Δ^+ be a positive polarization of Δ. If $\Gamma\ ;\ \Delta \Longrightarrow C @ w$, then $\cdot\ ;\ \cdot\ ;\ !\Gamma^-, \Delta^+ \Longrightarrow C^- @ w\ ;\ \cdot$.*

4 Encoding the Synchronous Stochastic π-calculus

In this section, we shall illustrate the use of HyLL($\mathcal{R}$) (Definiition 9) as a logical framework for constrained transition systems by encoding the syntax and the operational semantics of the synchronous stochastic π-calculus ($S\pi$), which extends the ordinary π-calculus by assigning to every channel and internal action an *inherent* rate of synchronization. In $S\pi$, each rate characterises an exponential distribution such that the probability of a reaction with rate r occuring within time t is given by $1 - e^{-rt}$ [31], where the *rate* r is a parameter.

We shall encode $S\pi$ in HyLL($\mathcal{R}$): a $S\pi$ reaction with rate r will be encoded by a transition of a probability described by a random variable with exponential distribution of parameter r; Worlds r in $\mathcal{R}$ will represent the list of the *rates* of the transitions performed so far.

HyLL($\mathcal{R}$) can therefore be seen as a formal language for expressing $S\pi$ executions (traces). For the rest of this section we shall use $r, s, t, \ldots$ instead of $u, v, w, \ldots$ to highlight the fact that the worlds represent (lists of) rates (overloading single elements and the list of single elements). We do not directly use rates because the syntax and transitions of $S\pi$ are given generically for a π-calculus with labelled actions, and it is only the interpretation of the labels that involves probabilities.

We first summarize the syntax of $S\pi$, which is a minor variant of a number of similar presentations such as [31]. For hygienic reasons we divide entities into the syntactic categories of *processes* ($P, Q, \ldots$) and *sums* ($M, N, \ldots$), defined as follows. We also include environments of recursive definitions (E) for constants.

$$\begin{array}{lll} \textit{(Processes)} & P, Q, \ldots & ::= \nu_r\ P \mid P \mid Q \mid 0 \mid X_n\, x_1 \cdots x_n \mid M \\ \textit{(Sums)} & M, N, \ldots & ::= \overline{x}\langle y\rangle.\ P \mid x.\ P \mid \tau_r.\ P \mid M + N \\ \textit{(Environments)} & E & ::= E, X_n \triangleq P \mid \cdot \end{array}$$

$P \mid Q$ is the parallel composition of P and Q, with unit 0. The restriction $\nu_r\ P$ abstracts over a free channel x in the process $P\,x$. We write the process using higher-order abstract

Interactions

$$\frac{}{\overline{x}\langle y\rangle.\ P + M \mid x.\ Q + M' \xrightarrow{\mathrm{rate}(x)} P \mid Q\ y}\ \text{SYN} \qquad \frac{}{\tau_r.\ P \xrightarrow{r} P}\ \text{INT}$$

$$\frac{P \xrightarrow{r} P'}{P \mid Q \xrightarrow{r} P' \mid Q}\ \text{PAR} \qquad \frac{\forall x_s.\left(P\ x \xrightarrow{r} Q\ x\right)}{\nu_s\ P \xrightarrow{r} \nu_s\ Q}\ \text{RES} \qquad \frac{P \xrightarrow{r} Q \quad P \equiv P' \quad Q \equiv Q'}{P' \xrightarrow{r} Q'}\ \text{CONG}$$

Congruence

$$\frac{}{P \mid 0 \equiv P} \quad \frac{}{P \mid Q \equiv Q \mid P} \quad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \quad \frac{}{\nu_r\ 0 \equiv 0} \quad \frac{X_n \triangleq P \in E}{E \vdash X_n\ x_1 \cdots x_n \equiv P\ x_1 \cdots x_n}$$

$$\frac{}{\nu_r(\lambda x.\ \nu_s(\lambda y.\ P)) \equiv \nu_s(\lambda y.\ \nu_r(\lambda x.\ P))} \quad \frac{\forall x_r.\ (P\ x \equiv Q\ x)}{\nu_r\ P \equiv \nu_r\ Q} \quad \frac{}{\nu_r(\lambda x.\ P \mid Q(x)) \equiv P \mid \nu_r\ Q}$$

$$\frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \quad \frac{P \equiv P'}{\overline{x}\langle m\rangle.\ P \equiv \overline{x}\langle m\rangle.\ P'} \quad \frac{\forall n.\ (P\ n \equiv Q\ n)}{x.\ P \equiv x.\ Q} \quad \frac{P \equiv P'}{\tau_r.\ P \equiv \tau_r.\ P'} \quad \frac{}{M + N \equiv N + M}$$

$$\frac{}{M + (N + K) \equiv (M + N) + K} \quad \frac{M \equiv M'}{M + N \equiv M' + N} \quad \frac{M \equiv N}{M + N \equiv M}$$

Figure 3 Interactions and congruence in $S\pi$. The environment E is elided in most rules.

syntax [28], *i.e.*, P in $\nu_r\ P$ is (syntactically) a function from channels to processes. This style lets us avoid cumbersome binding rules in the interactions because we reuse the well-understood binding structure of the λ-calculus. A similar approach was taken in the earliest encoding of the (ordinary) π-calculus in (unfocused) linear logic [25], and is also present in the encoding in CLF [9].

A sum is a non-empty choice ($+$) over terms with *action prefixes*: the output action $\overline{x}\langle y\rangle$ sends y along channel x, the input action x reads a value from x (which is applied to its continuation process), and the internal action τ_r has no observable I/O behaviour. Replication of processes happens via guarded recursive definitions [26]; in [34] it is argued that they are more practical for programming than the replication operator !. In a definition $X_n \triangleq P$, X_n denotes a (higher-order) defined constant of arity n; given channels $x_1, \ldots, x_n$, the process $X_n\ x_1 \cdots x_n$ is synonymous with $P\ x_1 \cdots x_n$. The constant X_n may occur on the right hand side of any definition in E, including in its body P, as long as it is prefixed by an action; this prevents infinite recursion without progress.

Interactions are of the form $E \vdash P \xrightarrow{r} Q$ denoting a transition from the process P to the process Q, in a global environment E, by performing an action at rate r. Each channel x is associated with an inherent rate specific to the channel, and internal actions τ_r have rate r. The restriction $\nu_r\ P$ defines the rate of the abstracted channel as r.

The full set of interactions and congruences are in fig. 3. We generally omit the global environment E in the rules as it never changes. It is possible to use the congruences to compute a normal form for processes that are a parallel composition of sums and each reaction selects two suitable sums to synchronise on a channel until there are no further reactions possible; this refinement of the operational semantics is used in $S\pi$ simulators such as SPiM [30].

▶ **Definition 11** (Syntax Encoding).

1. The encoding of the process P as a positive proposition, written $[\![P]\!]_{\mathrm{p}}$, is as follows (`sel` is a positive atom and `rt` a negative atom).

$$[\![P \mid Q]\!]_{\mathrm{p}} = [\![P]\!]_{\mathrm{p}} \otimes [\![Q]\!]_{\mathrm{p}} \qquad [\![\nu_r\ P]\!]_{\mathrm{p}} = \exists x.\ !(\mathtt{rt}\ x\ \mathtt{at}\ r) \otimes [\![P\,x]\!]_{\mathrm{p}}$$
$$[\![0]\!]_{\mathrm{p}} = \mathbf{1} \qquad [\![X_n\,x_1 \cdots x_n]\!]_{\mathrm{p}} = X_n\,x_1 \cdots x_n$$
$$[\![M]\!]_{\mathrm{p}} = \Downarrow(\mathtt{sel} \multimap [\![M]\!]_{\mathrm{s}})$$

2. The encoding of the sum M as a negative proposition, written $[\![M]\!]_{\mathrm{s}}$, is as follows (`out`, `in` and `tau` are positive atoms).

$$[\![M + N]\!]_{\mathrm{s}} = [\![M]\!]_{\mathrm{s}}\ \&\ [\![N]\!]_{\mathrm{s}} \qquad [\![\overline{x}\langle m\rangle.\ P]\!]_{\mathrm{s}} = \Uparrow(\mathtt{out}\ x\ m \otimes [\![P]\!]_{\mathrm{p}})$$
$$[\![x.\ P]\!]_{\mathrm{s}} = \forall n.\Uparrow(\mathtt{in}\ x\ n \otimes [\![P\,n]\!]_{\mathrm{p}}) \qquad [\![\tau_r.\ P]\!]_{\mathrm{s}} = \Uparrow(\mathtt{tau}\ r \otimes [\![P]\!]_{\mathrm{p}})$$

3. The encoding of the definitions E as a context, written $[\![E]\!]_{\mathrm{e}}$, is as follows.

$$\left[\!\!\left[E, X_n \triangleq P\right]\!\!\right]_{\mathrm{e}} = [\![E]\!]_{\mathrm{e}}, \dagger\forall x_1, \ldots, x_n.\ X_n\,x_1 \cdots x_n \equiv [\![P\,x_1 \cdots x_n]\!]_{\mathrm{p}}$$
$$[\![\cdot]\!]_{\mathrm{e}} = \cdot$$

where $P \equiv Q$ is defined as $(P \multimap \Uparrow Q)\ \&\ (Q \multimap \Uparrow P)$.

The encoding of processes is positive, so they will be decomposed in the active phase when they occur on the left of the sequent arrow, leaving a collection of sums. The encoding of restrictions will introduce a fresh unrestricted assumption about the rate of the restricted channel. Each sum encoded as a processes undergoes a polarity switch because $\multimap$ is negative; the antecedent of this implication is a *guard* `sel`. This pattern of guarded switching of polarities prevents unsound congruences such as $\overline{x}\langle m\rangle.\ \overline{y}\langle n\rangle.\ P \equiv \overline{y}\langle n\rangle.\ \overline{x}\langle m\rangle.\ P$ that do not hold for the synchronous π calculus. To see this, note that $[\![\overline{x}\langle m\rangle.\ \overline{y}\langle n\rangle.\ P]\!]_{\mathrm{p}}$ has the form $X \multimap (A \otimes (X \multimap B \otimes C))$ (eliding the polarity shifts) which is not provably equivalent to $X \multimap (B \otimes (X \multimap A \otimes C))$ in both linear logic and HyLL. Thus, even though we use a commutative connective $\otimes$ in $[\![\overline{x}\langle m\rangle.\ P]\!]_{\mathrm{s}}$, output actions are still sequential and synchronous.

The guard `sel` also *locks* the sums in the context: the $S\pi$ interaction rules INT and SYN discard the non-interacting terms of the sum, so the environment will contain the requisite number of `sel`s only when an interaction is in progress. The action prefixes themselves are also synchronous, which causes another polarity switch. Each action releases a token of its respective kind—`out`, `in` or `tau`—into the context. These tokens must be consumed by the interaction before the next interaction occurs. For each action, the (encoding of the) continuation process is also released into the context.

The proof of the following congruence lemma is omitted. Because the encoding is (essentially) a $\otimes$/& structure, there are no distributive laws in linear logic that would break the process/sum structure.

▶ **Theorem 12** (Congruence). *$E \vdash P \equiv Q$ iff both $[\![E]\!]_{\mathrm{e}}\,@\iota\ ;\ \cdot\ ;\ [\![P]\!]_{\mathrm{p}}\,@\iota \Longrightarrow \cdot\ ;\ [\![Q]\!]_{\mathrm{p}}\,@\iota$ and $[\![E]\!]_{\mathrm{e}}\,@\iota\ ;\ \cdot\ ;\ [\![Q]\!]_{\mathrm{p}}\,@\iota \Longrightarrow \cdot\ ;\ [\![P]\!]_{\mathrm{p}}\,@\iota$.*

Now we encode the interactions. Because processes were lifted into propositions, we can be parsimonious with our encoding of interactions by limiting ourselves to the atomic interactions SYN and INT (below); the PAR, RES and CONG interactions will be ambiently implemented by the logic. Because there are no concurrent interactions—only one interaction can trigger at a time in a trace—the interaction rules must obey a locking discipline. We

represent this lock as the proposition $\mathtt{act}$ that is consumed at the start of an interaction and produced again at the end. This lock also carries the net rate of the prefix of the trace so far: that is, an interaction $P \xrightarrow{r} Q$ will update the lock from $\mathtt{act}@s$ to $\mathtt{act}@s \cdot r$. The encoding of individual atomic interactions must also remove the $\mathtt{in}$, $\mathtt{out}$ and $\mathtt{tau}$ tokens introduced in context by the interacting processes.

▶ **Definition 13** (Interaction)**.**
Let $\mathtt{inter} \triangleq \dagger(\mathtt{act} \multimap \Uparrow\mathtt{int} \,\&\, \Uparrow\mathtt{syn})$ where $\mathtt{act}$ is a positive atom and $\mathtt{int}$ and $\mathtt{syn}$ are as follows:

$$\mathtt{int} \triangleq (\mathtt{sel}\,\mathtt{at}\,\iota) \otimes \Downarrow\forall r.\, \Big((\mathtt{tau}\, r\,\mathtt{at}\,\iota) \multimap \rho_r \Uparrow\mathtt{act}\Big)$$

$$\mathtt{syn} \triangleq (\mathtt{sel} \otimes \mathtt{sel}\,\mathtt{at}\,\iota) \otimes \Downarrow\forall x, r, m.\, \Big((\mathtt{out}\, x\; m \otimes \mathtt{in}\, x\; m\,\mathtt{at}\,\iota) \multimap \Downarrow(\mathtt{rt}\, x\,\mathtt{at}\, r) \multimap \rho_r \Uparrow\mathtt{act}\Big).$$

The number of interactions that are allowed depends on the number of instances of $\mathtt{inter}$ in the linear context: each focus on $\mathtt{inter}$ implements a single interaction. If we are interested in all finite traces, we will add $\mathtt{inter}$ to the unrestricted context so it may be reused as many times as needed.

4.1 Representational Adequacy.

Adequacy consists of two components: completeness and soundness. Completeness is the property that every $S\pi$ execution is obtainable as a HyLL derivation using this encoding, and is the comparatively simpler direction (see Thm. 16). Soundness is the reverse property, and is false for unfocused HyLL as such. However, it *does* hold for focused proofs (see Thm. 18). In both cases, we reason about the following canonical sequents of HyLL.

▶ **Definition 14.** The *canonical context of* P, written $\lfloor P \rfloor$, is given by:

$$\lfloor X_n\, x_1 \cdots x_n \rfloor = \Uparrow X_n\, x_1 \cdots x_n \qquad \lfloor P \mid Q \rfloor = \lfloor P \rfloor, \lfloor Q \rfloor \qquad \lfloor 0 \rfloor = \cdot \qquad \lfloor \nu_r\; P \rfloor = \lfloor P\, a \rfloor$$
$$\lfloor M \rfloor = \mathtt{sel} \multimap [\![M]\!]_{\mathrm{s}}$$

For $\lfloor \nu_r\; P \rfloor$, the right hand side uses a *fresh* channel a that is not free in the rest of the sequent it occurs in.

As an illustration, take $P \triangleq \overline{x}\langle a \rangle.\, Q \mid x.\, R$. We have:

$$\lfloor P \rfloor = \;\; \mathtt{sel} \multimap \Uparrow(\mathtt{out}\, x\; a \otimes [\![Q]\!]_{\mathrm{p}}), \mathtt{sel} \multimap \forall y. \Uparrow(\mathtt{in}\, x\; y \otimes [\![R\, y]\!]_{\mathrm{p}})$$

Obviously, the canonical context is what would be emitted to the linear zone at the end of the active phase if $[\![P]\!]_{\mathrm{p}}$ were to be present in the left active zone.

▶ **Definition 15.** A neutral sequent is *canonical* iff it has the shape

$$[\![E]\!]_{\mathrm{e}}, \mathtt{rates}, \mathtt{inter}@\iota \;;\; \Uparrow\mathtt{act}@s, \lfloor P_1 \mid \cdots \mid P_k \rfloor @\iota \;;\; \cdot \Longrightarrow \cdot \;;\; ([\![Q]\!]_{\mathrm{p}}\,\mathtt{at}\,\iota) \otimes \mathtt{act}@t$$

where $\mathtt{rates}$ contains elements of the form $\mathtt{rt}\, x @ r$ defining the rate of the channel x as r, and all free channels in $[\![E]\!]_{\mathrm{e}}, \lfloor P_1 \mid \cdots \mid P_k \mid Q \rfloor$ have a single such entry in $\mathtt{rates}$.

Figure 4 contains an example of a derivation for a canonical sequent involving P. Focusing on any (encoding of a) sum in $\lfloor P \rfloor @\iota$ will fail because there is no $\mathtt{sel}$ in the context, so only $\mathtt{inter}$ can be given focus; this will consume the $\mathtt{act}$ and release two copies of $(\mathtt{sel}\,\mathtt{at}\,\iota)$ and the continuation into the context. Focusing on the latter will fail now (because $\mathtt{out}\, x\; m$ and $\mathtt{in}\, x\; m$ (for some m) are not yet available), so the only applicable foci are the two

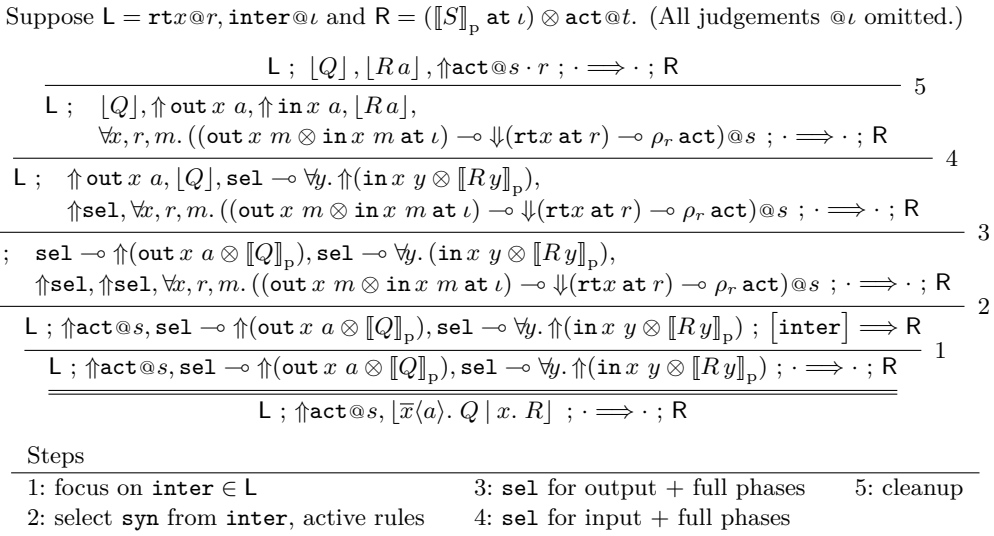


Figure 4 Example interaction in the $S\pi$-encoding.

sums that can now be "unlocked" using the $\mathtt{sel}$s. The output and input can be unlocked in an irrelevant order, producing two tokens $\mathtt{in}\,x\ a$ and $\mathtt{out}\,x\ a$. Note in particular that the witness a was chosen for the universal quantifier in the encoding of $x.\,Q$ because the subsequent consumption of these two tokens requires the messages to be identical. (Any other choice will not lead to a successful proof.) After both tokens are consumed, we get the final form $\mathtt{act}@s \cdot r$, where r is the inherent rate of x (found from the $\mathtt{rates}$ component of the unrestricted zone). This sequent is canonical and contains $\lfloor Q \mid R\, a \rfloor$.

Our encoding therefore represents every $S\pi$ action in terms of "micro" actions in the following rigid order: one micro action to determine what kind of action (internal or synchronization), one micro action per sum to select the term(s) that will interact, and finally one micro action to establish the contract of the action. Thus we see that focusing is crucial to maintain the semantic interpretation of (neutral) sequents. In an unfocused calculus, several of these steps could have partial overlaps, making such a semantic interpretation inordinately complicated. We do not know of any encoding of the π calculus that can provide such interpretations in unfocused sequents without changing the underlying logic. In CLF [9] the logic is extended with explicit monadic staging, and this enables a form of adequacy [9]; however, the encoding is considerably more complex because processes and sums cannot be fully lifted and must instead be specified in terms of a lifting computation. Adequacy is then obtained via a permutative equivalence over the lifting operation. Other encodings of π calculi in linear logic, such as [20] and [3], concentrate on the easier asynchronous fragment and lack adequacy proofs anyhow.

▶ **Theorem 16** (Completeness). *If $E \vdash P \xrightarrow{r} Q$, then the following canonical sequent is derivable.*

$$\llbracket E \rrbracket_{\mathrm{e}}, \mathtt{rates}, \mathtt{inter}@\iota\ ;\ \Uparrow \mathtt{act}@s, \lfloor P \rfloor@\iota\ ;\ \cdot \Longrightarrow \cdot\ ;\ (\llbracket Q \rrbracket_{\mathrm{p}}\ \mathtt{at}\ \iota) \otimes \mathtt{act}@s \cdot r.$$

Proof. By structural induction of the derivation of $E \vdash P \xrightarrow{r} Q$. Every interaction rule of $S\pi$ is implementable as an admissible inference rule for canonical sequents. For CONG, we appeal to Thm. 12. ◀

Completeness is a testament to the expressivity of the logic – all executions of $S\pi$ are also expressible in HyLL. However, we also require the opposite (soundness) direction: that every canonical sequent encodes a possible $S\pi$ trace. The proof hinges on the following canonicity lemma.

▶ **Lemma 17** (Canonical Derivations). *In a derivation for a canonical sequent, the derived inference rules for* `inter` *are of one of the two following forms (conclusions and premises canonical).*

$$\frac{}{[\![E]\!]_{\mathrm{e}}, \mathtt{rates}, \mathtt{inter}\,@\iota \;;\; \Uparrow \mathtt{act}\,@s, \lfloor P \rfloor\,@\iota \;;\; \cdot \Longrightarrow \cdot \;;\; ([\![P]\!]_{\mathrm{p}}\;\mathtt{at}\;\iota) \otimes \mathtt{act}\,@s}$$

$$\frac{[\![E]\!]_{\mathrm{e}}, \mathtt{rates}, \mathtt{inter}\,@\iota \;;\; \Uparrow \mathtt{act}\,@s \cdot r, \lfloor Q \rfloor\,@\iota \;;\; \cdot \Longrightarrow \cdot \;;\; ([\![R]\!]_{\mathrm{p}}\;\mathtt{at}\;\iota) \otimes \mathtt{act}\,@t}{[\![E]\!]_{\mathrm{e}}, \mathtt{rates}, \mathtt{inter}\,@\iota \;;\; \Uparrow \mathtt{act}\,@s, \lfloor P \rfloor\,@\iota \;;\; \cdot \Longrightarrow \cdot \;;\; ([\![R]\!]_{\mathrm{p}}\;\mathtt{at}\;\iota) \otimes \mathtt{act}\,@t}$$

where: either $E \vdash P \equiv Q$ *with* $r = \iota$ *or* $E \vdash P \xrightarrow{r} Q$.

Proof. This is a formal statement of the phenomenon observed earlier in the example (Fig. 4): $[\![R]\!]_{\mathrm{p}} \otimes \mathtt{act}$ cannot be focused on the right unless $P \equiv R$, in which case the derivation ends with no more foci on `inter`. If not, the only elements available for focus are `inter` and one of the congruence rules $[\![E]\!]_{\mathrm{e}}$ in the unrestricted context. In the former case, the definition of a top level X_n in $\lfloor P \rfloor$ is (un)folded (without advancing the world). In the latter case, the derived rule consumes the $\Uparrow \mathtt{act}\,@s$, and by the time `act` is produced again, its world has advanced to $s \cdot r$. The proof proceeds by induction on the structure of P. ◀

Lemma 17 is a strong statement about HyLL derivations using this encoding: every partial derivation using the derived inference rules represents a prefix of an $S\pi$ trace. This is sometimes referred to as *full adequacy*, to distinguish it from adequacy proofs that require complete derivations [27]. The structure of focused derivations is crucial because it allows us to close branches early (using init). It is impossible to perform a similar analysis on unfocused proofs for this encoding; both the encoding and the framework will need further features to implement a form of staging [9, Chapter 3].

▶ **Corollary 18.** *If* $[\![E]\!]_{\mathrm{e}}, \mathtt{rates}, \mathtt{inter}\,@\iota \;;\; \Uparrow \mathtt{act}\,@\iota, \lfloor P \rfloor\,@\iota \;;\; \cdot \Longrightarrow \cdot \;;\; ([\![Q]\!]_{\mathrm{p}}\;\mathtt{at}\;\iota) \otimes \mathtt{act}\,@r$ *is derivable, then* $E \vdash P \xrightarrow{r}{}^{*} Q$.

Proof. Directly from Lem. 17. ◀

4.2 Stochastic Correctness with respect to simulation

So far the HyLL($\mathcal{R}$) encoding of $S\pi$ represents any $S\pi$ trace symbolically. However, not every symbolic trace of an $S\pi$ process can be produced according to the operational semantics of $S\pi$, which is traditionally given by a simulator. This is the main difference between HyLL (and $S\pi$) and the approach of CSL [2], where the truth of a proposition is evaluated against a CTMC, which is why equivalence in CSL is identical to CTMC bisimulation [16]. In this section we sketch how the execution could be used directly on the canonical sequents to produce only correct traces (proofs). The proposal in this section should be seen by analogy to the execution model of $S\pi$ simulators such as SPiM [29], although we do not use the Gillespie algorithm.

The main problem of simulation is determining which of several competing enabled actions in a canonical sequent to select as the "next" action from the *race condition* of the actions enabled in the sequent. Because of the focusing restriction, these enabled actions are

easy to compute. Each element of $\lfloor P \rfloor$ is of the form $\texttt{sel} \multimap [\![M]\!]_s$, so the enabled actions in that element are given precisely by the topmost occurrences of $\Uparrow$ in $[\![M]\!]_s$. Because none of the sums can have any restricted channels (they have all been removed in the active decomposition of the process earlier), the rates of all the channels will be found in the `rates` component of the canonical sequent.

The effective rate of a channel x is related to its inherent rate by scaling by a factor proportional to the *activity* on the channel, as defined in [29]. Note that this definition is on the *rate constants* of exponential distributions, not the rates themselves. The distribution of the minimum of a list of random variables with exponential distribution is itself an exponential distribution whose rate constant is the sum of those of the individual variables. Each individual transition on a channel is then weighted by the contribution of its rate to this sum. The choice of the transition to select is just the ordinary logical non-determinism. Note that the rounds of the algorithm do not have an associated *delay* element as in [29]; instead, we compute (symbolically) a distribution over the delays of a sequence of actions.

Because stochastic correctness is not necessary for the main adequacy result in the previous subsection, we leave the details of simulation to future work.

5 Related Work

Logically, the HyLL sequent calculus is a variant of labelled deduction [37], a very broad topic not elaborated on here. The combination of linear logic with labelled deduction isn't new to this work. In the η-logic [17] the constraint domain is intervals of time, and the rules of the logic generate constraint inequalities as a side-effect; however its sole aim is the representation of proof-carrying authentication, and it does not deal with the full generality of constraint domains or with focusing. The main feature of η not in HyLL is a separate constraint context that gives new constrained propositions. HyLL is also related to the Hybrid Logical Framework (HLF) [33] which captures linear logic itself as a labelled form of intuitionistic logic. Encoding constrained π calculi directly in HLF would be an interesting exercise: we would combine the encoding of linear logic with the constraints of the process calculus. Because HLF is a very weak logic with a proof theory based on natural deduction, it is not clear whether (and in what forms) an adequacy result in HyLL can be transferred to HLF.

Temporal logics such as CSL and PCTL [22] are popular for logical reasoning on temporal properties of transition systems with probabilities. In such logics, truth is defined in terms of correctness with respect to a constrained forcing relation on the constraint algebra. In CSL and PCTL states are formal entities (names) labeled with atomic propositions. Formulae are interpreted on algebraic structures that are discrete (in PCTL) or continuous (in CSL) time Markov chains. Transitions between states are viewed as pairs of states labeled with a probability (the probability of the transition), which is defined as a function from $S \times S$ into $[0, 1]$, where S is the set of states. While such logics have been very successful in practice with efficient tools, the proof theory of these logics is very complex. Indeed, such modal logics generally cannot be formulated in the sequent calculus, and therefore lack cut-elimination and focusing. In contrast, HyLL has a very traditional proof theoretic pedigree, but lacks such a close correspondence between logical and algebraic equivalence. Probably the most well known and relevant stochastic formalism not already discussed is that of stochastic Petri-nets [24], which have a number of sophisticated model checking tools, including the PRISM framework [23]. Recent advances in proof theory suggest that the benefits of model checking can be obtained without sacrificing proofs and proof search [4].

6 Conclusion and Future Work

We have presented HyLL, a hybrid extension of intuitionistic linear logic with a simple notion of situated truth, a traditional sequent calculus with cut-elimination and focusing, and a modular and instantiable constraint system (set of worlds) that can be directly manipulated using hybrid connectives. We have proposed two instances of HyLL (i.e two particular instances of the set of worlds): one modelling temporal constraints and the others modelling stochastic (continuous time Markov processes) constraints. We have shown how to obtain representationally adequate encodings of constrained transition systems, such as the synchronous stochastic π-calculus in a suitable instance of HyLL.

Several instantiations of HyLL besides the ones in this paper seem interesting. For example, we can already use disjunction ($\oplus$) to explain disjunctive states, but it is also possible to obtain a more extensional branching by treating the worlds as points in an arbitrary partially-ordered set instead of a monoid. Another possibility is to consider lists of worlds instead of individual worlds – this would allow defining periodic availability of a resource, such as one being produced by an oscillating process. The most interesting domain is that of discrete probabilities: here the underlying semantics is given by discrete time Markov chains instead of CTMCs, which are often better suited for symbolic evaluation [40].

The logic we have provided so far is a logical framework well suited *to represent* constrained transition systems. The design of a logical framework *for* (i.e. to reason about) constrained transition systems is left for future work -and might be envisioned by using a two-levels logical framework such as the Abella system [19]. The work presented in [15] provides a first step in this direction in the area of systems biology (where biological systems are viewed as transition systems), using the Coq proof assistant [5] with HyLL($\mathcal{T}'$) (with discrete instants of time) as an object logic. This work can be seen as a first possible implementation of HyLL with temporal constraints.

An important open question is whether a general logic such as HyLL can serve as a framework for specialized logics such as CSL and PCTL. A related question is what benefit linearity truly provides for such logics – linearity is obviously crucial for encoding process calculi that are inherently stateful, but CSL requires no such notion of single consumption of resources.

In the κ-calculus, reactions in a biological system are modeled as reductions on graphs with certain state annotations. It appears (though this has not been formalized) that the κ-calculus can be embedded in HyLL even more naturally than $S\pi$, because a solution—a multiset of chemical products—is simply a tensor of all the internal states of the binding sites together with the formed bonds. One important innovation of κ is the ability to extract semantically meaningful "stories" from simulations. We believe that HyLL provides a natural formal language for such stories.

We became interested in the problem of encoding stochastic reasoning in a resource aware logic because we were looking for the logical essence of biochemical reactions. What we envision for the domain of "biological computation" is a resource-aware stochastic or probabilistic λ-calculus that has HyLL propositions as (behavioral) types.

Acknowledgements. This work was partially supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and by the European TYPES project. We thank François Fages, Sylvain Soliman, Alessandra Carbone, Vincent Danos and Jean Krivine for fruitful discussions on various preliminary versions of the work presented here. Thanks also

go to Nicolas Champagnat, Luc Pronzato and André Hirschowitz who helped us understand the algebraic nature of stochastic constraints.

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

2 A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

3 David Baelde. Logique linéaire et algèbre de processus. Technical report, INRIA Futurs, LIX and ENS, 2005.

4 David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, ed., *21th Conf. on Automated Deduction (CADE)*, number 4603 in LNAI, pp. 391–397, Springer, 2007.

5 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

6 Torben Braüner and Valeria de Paiva. Intuitionistic hybrid logic. *Journal of Applied Logic*, 4:231–255, 2006.

7 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, eds., *Proc. of the 21th Int'l Conf. on Concurrency Theory (CONCUR)*, vol. 6269 of *Lecture Notes in Computer Science*, pp. 222–236. Springer, 2010.

8 Iliano Cervesato. Typed multiset rewriting specifications of security protocols. *Electronic Notes on Theoretical Computer Sciience*, 40:8–51, 2000.

9 Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University, 2003. Revised, May 2003.

10 Nathalie Chabrier-Rivier, François Fages, and Sylvain Soliman. The biochemical abstract machine BIOCHAM. In *International Workshop on Computational Methods in Systems Biology (CMSB-2)*, LNCS. Springer, 2004.

11 Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, Dec. 2003.

12 Kaustuv Chaudhuri and Joëlle Despeyroux. A hybrid linear logic for constrained transition systems with applications to molecular biology. Technical Report hal-00402942, INRIA-HAL, October 2013.

13 Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. of Automated Reasoning*, 40(2-3):133–177, March 2008.

14 Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.

15 Elisabetta de Maria, Joëlle Despeyroux, and Amy Felty. A logical framework for systems biology. Technical Report hal-00981409, INRIA-HAL, April 2014.

16 Josée Desharmais and Prakash Panangaden. Continuous stochastic logic characterizes bisimulation of continuous-time Markov processes. *Journal of Logic and Algebraic Programming*, 56:99–115, 2003.

17 Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Computer Security Foundations Symp. (CSF-21)*, pp. 133–145. IEEE CS, 2008.

18 E. Allen Emerson. Temporal and modal logic. In *TCS*, pages 995–1072. Elsevier, 1995.

19 Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, September 2009.

20 Deepak Garg and Frank Pfenning. Type-directed concurrency. In Martín Abadi and Luca de Alfaro, editors, *16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *LNCS*, pages 6–20. Springer, 2005.

21 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

22 H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6, 1994.

23 M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking using PRISM: a hybrid approach. *International Journal of Software Tools for Technology Transfer*, 6(2), 2004.

24 M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Francenschinis. *Modelling with Generalised Stochastic Petri Nets*. Wiley Series in Parallel Computing. Wiley and Sons, 1995.

25 Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *3rd Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265, Bologna, Italy, 1993. Springer.

26 Robin Milner. *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, New York, NY, USA, 1999.

27 Vivek Nigam and Dale Miller. Focusing in linear meta-logic. In *Proc. of IJCAR: Int'l Joint Conf. on Automated Reasoning*, volume 5195 of *LNAI*, pages 507–522. Springer, 2008.

28 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.

29 Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. *Concurrent Models in Molecular Biology*, August 2004.

30 Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. In *Proceedings of BioConcur'04*, ENTCS, 2004.

31 Andrew Phillips, Luca Cardelli, and Giuseppe Castagna. A graphical representation for biological processes in the stochastic pi-calculus. *Transactions on Computational Systems Biology VII*, pages 123–152, 2006.

32 Arthur N. Prior. *Time and Modality*. Oxford: Clarendon Press, 1957.

33 Jason Reed. Hybridizing a logical framework. In *International Workshop on Hybrid Logic (HyLo)*, Seattle, USA, August 2006.

34 A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the π-calculus and process algebra. In L. Hunter, R. B. Altman, A. K. Dunker, and T. E. Klein, editors, *Pacific Symposium on Biocomputing*, volume 6, pages 459–470, Singapore, 2001. World Scientific Press.

35 L. C. G. Rogers and D. Williams. *Diffusions, Markov Processes and Martingales*, volume 1: Foundations. Cambridge Mathematical Library, 2nd edition, 2000.

36 Uluç Saranli and Frank Pfenning. Using constrained intuitionistic linear logic for hybrid robotic planning problems. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3705–3710. IEEE, 2007.

37 Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

38 Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

39 Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2003. Revised, May 2003.

40 Peng Wu, Catuscia Palamidessi, and Huimin Lin. Symbolic bisimulations for probabilistic systems. In *QEST'07*, pages 179–188. IEEE Computer Society, 2007.

The Rooster and the Syntactic Bracket*

Hugo Herbelin[1] and Arnaud Spiwack[2]

1 **Inria Paris-Rocquencourt**
Paris, France
hugo.herbelin@inria.fr
2 **Inria Paris-Rocquencourt**
Paris, France
arnaud@spiwack.net

Abstract

We propose an extension of pure type systems with an algebraic presentation of inductive and co-inductive type families with proper indices. This type theory supports coercions toward from smaller sorts to bigger sorts via explicit type construction, as well as impredicative sorts. Type families in impredicative sorts are constructed with a bracketing operation. The necessary restrictions of pattern-matching from impredicative sorts to types are confined to the bracketing construct. This type theory gives an alternative presentation to the calculus of inductive constructions on which the Coq proof assistant is an implementation.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs

Keywords and phrases Coq, calculus of inductive constructions, impredicativity, strictly positive type families, inductive type families

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.169

1 Introduction

In the Coq proof assistant [14] inductive types are treated as toplevel definitions. If it makes sense from a convenience or an efficiency point of view, the monolithic nature of the definitions make it hard to describe what they precisely mean. As a matter of fact, inductive definitions mean different things depending on the type they are defined in: specifically, some types are interpreted differently in impredicative sorts like **Prop** or the impredicative variant of **Set**.

In this article, we present a calculus of inductive and co-inductive constructions where inductive and co-inductive types are presented algebraically. The algebraic presentation is an extension of a PTS [3] with inductive and co-inductive type families. Thanks to its modularity, it is meant to serve as a description which is simpler to expose and more mathematically amenable than the monolithic scheme which is found in a practical system such as Coq. For the sake of clarity, the system is given with a single universe and explicit subtyping, although Coq has an unbounded cumulative hierarchy of universes and implicit subtyping. Apart from these technicalities, it is believed that our calculus of algebraic inductive and co-inductive constructions expresses all the features of the Set-impredicative Calculus of Inductive Constructions that Coq implements, *e.g.* in its version 8.4 when launched with option -impredicative-set.

* This research has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD

$$\frac{\Gamma \vdash A : s \quad x \text{ is fresh in } \Gamma}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma \vdash u : A \quad \Gamma \vdash B : s \quad x \text{ is fresh in } \Gamma}{\Gamma, x{:}B \vdash u : A}$$

$$\frac{\Gamma \vdash \prod_{x:A} B : s \quad \Gamma, x{:}A \vdash u : B}{\Gamma \vdash \lambda x^A.\, u : \prod_{x:A} B} \qquad \frac{\Gamma \vdash u : \prod_{x:A} B \quad \Gamma \vdash v : A}{\Gamma \vdash u\, v : B[\, x \setminus v\,]}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash B : s \quad A \equiv B}{\Gamma \vdash u : B} \qquad \left(\lambda x^A.\, u\right) v \rightsquigarrow u[\, x \setminus v\,]$$

Figure 1 Generic rules of PTS.

This work draws most of its inspiration from Morris & al [9, 10] for the algebraic presentation of inductive type families in a predicative sort, and Awodey & Bauer [2] for the treatment of impredicative sorts.

We use examples from Coq to illustrate the algebraic presentation. To differentiate expressions in Coq from expressions in the algebraic presentation, the former are typeset in a sans-serif font and the latter in a roman font.

2 Pure type systems

To model the type system of Coq, we start with the classic presentation of pure type systems (PTS) of Barendregt [3], which we will then extend to model type families. A PTS is characterised by a *single* syntactic category of terms which are used both as λ-terms and as types. It has a single form of typing judgment $\Gamma \vdash u : A$, where u and A are terms, and Γ a context assigning terms to variables. A PTS has a set of *sorts*, which we shall denote schematically by the symbol s. Every sort is an atomic term. A PTS has a conversion relation $u \equiv v$. Here we diverge from the presentation of [3] which always uses β-conversion. Coq, on the other hand, uses $\beta\eta$-conversion on the fragment described in this section. In this article we will take the conversion rule as abstract, not even requiring it to be decidable. We will only require that it contains all the reduction rules which are given in the form $u \rightsquigarrow v$ (in this section, we only have β-reduction).

The typing rules of a PTS comprise of a set of generic rules given in Figure 1, together with a number of rules of the form

$$\frac{}{\vdash s_1 : s_2}$$

called axioms, and rules of the form

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \prod_{x:A} B : s_3}$$

of product formation rules. As usual we write $A \to B$ for $\prod_{x:A} B$ when x does not bind a variable in B.

As a starting point of the algebraic presentation, we shall use a PTS with two sorts, Type and $\Box$, together with the following axiom:

$$\frac{}{\Gamma \vdash \mathrm{Type} : \Box}$$

and the following product formation rule:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \prod_{x:A} B : \max s_1\ s_2}$$

where $\max s\ s = s$ and $\max \mathrm{Type}\ \Box = \max \Box\ \mathrm{Type} = \Box$.

The sorts Type and $\Box$ are predicative. The sort $\Box$ plays a technical role in allowing type variable and the formation of type-level functions; it cannot, however, be referenced in terms. In the following sections, $\Box$ will also be used to be able to define types by pattern-matching (*strong elimination*).

To model the entire Coq system, Type and $\Box$ would be replaced with a hierarchy of predicative sorts Type_i, such that

$$\frac{}{\Gamma \vdash \mathrm{Type}_i : \mathrm{Type}_{i+1}}$$

are axioms. Adapting the presentation to a hierarchy of sorts is straightforward, but in the interest of keeping to the heart of the matter we give a presentation with two sorts.

3 Inductive type families

We shall now extend the algebraic presentation with a notion of *inductive type families* to model (dependent) datatypes. In this section we will stay in the predicative fragment of Coq.

Contrary to the inductive types of Coq, where inductive definitions must be *named* at *toplevel*, like in:

```
Inductive Even : Type :=
| eo : Even
| es : Odd → Even
with Odd : Type :=
| os : Even → Odd.
```

the presentation given here is essentially anonymous, and inductive families need not be defined at toplevel prior to use. Mutually inductive types such as Even and Odd are not modelled directly, rather they are encoded using an index:

```
Inductive EvenOdd : bool → Type :=
| eo : EvenOdd true
| es : EvenOdd false → EvenOdd true
| os : EvenOdd true → EvenOdd false.
Definition Even := EvenOdd true.
Definition Odd := EvenOdd false.
```

This encoding works as long as all the mutual definitions are all in the same sort. A variant for mutual definition involving **Type** and **Prop** is demonstrated in Section 4.3. When the types being defined are in different predicative sorts, however, we have to resort to another encoding which involves nested datatypes [4, Section 8.6].

We will not explore the latter kind of mutual definition. However, nested datatypes – where recursive calls occur as arguments of another type – such as:

```
Inductive List (A:Type) : Type :=
| nil : List A
| cons : A → List A → List A.
Inductive Tree : Type :=
| node : List Tree → Tree.
```

are indeed modelled in the algebraic presentation.

3.1 Regular types

To be able to traverse terms of inductive type, the core PTS constructions is extended with a *recursive fixed point* on functions:

$$\frac{\Gamma \vdash \prod_{x_1:A_1,\ldots,x_{n-1}:A_{n-1},x_n:A_n} B : s \qquad \text{guarded}\, f\, x_1 \ldots x_{n-1}\, x_n\, u \qquad \Gamma, f{:}\prod_{x_1:A_1,\ldots,x_{n-1}:A_{n-1},x_n:A_n} B, x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1}, x_n{:}A_n \vdash u : B}{\Gamma \vdash \text{fix}\ f\ x_1{:}A_1\ \ldots\ x_{n-1}{:}A_{n-1}\ x_n{:}A_n \Rightarrow u : \prod_{x_1:A_1,\ldots,x_{n-1}:A_{n-1},x_n:A_n} B}\,.$$

Recursive fixed points are unfolded when fully applied

$$(\text{fix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u)\, v_1 \ldots v_n \rightsquigarrow u[\, x_i \setminus v_i\,]\,.$$

To ensure strong normalisation, this reduction rule is limited, and a guard condition is imposed on the recursive calls to f. It is not, however, the object of this article to discuss these restriction or the guard condition. Briefly, Coq currently relies on a single *structural* argument in the block $x_1, \ldots, x_n$: fixed points are not unfolded until their structural argument starts with a constructor, and the guard condition ensures that each recursive call is performed on a subterm of said structural argument, for a relaxed notion of subterm. Other possibilities exist: Agda2 [11] uses any number of arguments as structural, and tries to find a lexicographic ordering. Yet another possibilities is to use sized types [1]. We shall simply assume that an adequate guard condition is given.

We now extend the grammar of type constructors. The presentation of this article is largely inspired by the synthetic definition of *strictly positive families* by Morris *&* al [9, 10], but is adapted to *intensional type theory*. The presentation of [9, 10] is designed for generic programming inside a type theory, they give codes for strictly positive families which are then decoded into an actual type of the ambient theory. No elimination principle needs to be given for the strictly positive families, as they are implicit in their decoding. Here, we are defining the syntax of inductive type families, including their elimination rules.

The regular type constructors, whose typing rules are given in Figure 2, are the empty type 0, the unit type 1, and the sum of two types. The elimination rules are given in the form of *dependent pattern-matching* with a syntax made to remind of that of Coq. We shall often omit the typing predicate when it is clear from the context, especially when it does not depend on the branch. With this material we can define a first example type, namely the booleans:

$$\begin{cases} \mathbb{B} = 1 + 1 \\ \text{true} = \text{inl}\,() \\ \text{false} = \text{inr}\,()\,. \end{cases}$$

3.2 Inductive type families

Inductive families differ from regular inductive types in that they are parametrised by *indices*, that is they are functions $F : A \to \text{Type}$ for some A. An inductive family of the form $\lambda x^A.\, R$, is said to be *uniformly parametrised* by A. In general, inductive families are not uniformly parametrised: the value of the index is allowed to vary in recursive calls, and constructors may build values of $F\, x$ for certain x only. Remember, for instance, the EvenOdd family:

```
Inductive EvenOdd : bool → Type :=
| eo : EvenOdd true
| es : EvenOdd false → EvenOdd true
| os : EvenOdd true → EvenOdd false.
```

Sum type

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma \vdash B : \mathrm{Type}}{\Gamma \vdash A + B : \mathrm{Type}}$$

$$\frac{\Gamma \vdash A + B : s \quad \Gamma \vdash t : A}{\Gamma \vdash \mathrm{inl}\, t : A + B} \qquad \frac{\Gamma \vdash A + B : s \quad \Gamma \vdash u : B}{\Gamma \vdash \mathrm{inr}\, u : A + B}$$

$$\frac{\Gamma \vdash u : A + B \quad \Gamma, x{:}A + B \vdash P : s \quad \Gamma, y{:}A \vdash v : P[x \setminus \mathrm{inl}\, y] \quad \Gamma, z{:}A \vdash w : P[x \setminus \mathrm{inr}\, z]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{inl}\, y \Rightarrow v \\ \mid\ \mathrm{inr}\, z \Rightarrow w\end{array} : P[x \setminus u]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{inl}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{inl}\, y \Rightarrow v \\ \mid\ \mathrm{inr}\, z \Rightarrow w\end{array} \rightsquigarrow v[y \setminus u]$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{inr}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{inl}\, y \Rightarrow v \\ \mid\ \mathrm{inr}\, z \Rightarrow w\end{array} \rightsquigarrow w[z \setminus u]$$

Unit type

$$\frac{}{\Gamma \vdash 1 : \mathrm{Type}} \qquad \frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma \vdash u : 1 \quad \Gamma, x{:}1 \vdash P : s \quad \Gamma \vdash v : P[x \setminus ()]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ () \Rightarrow v\end{array} : P[x \setminus u]}$$

$$\begin{array}{l}\mathrm{match}\ ()\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ () \Rightarrow v\end{array} \rightsquigarrow v$$

Empty type

$$\frac{}{\Gamma \vdash 0 : \mathrm{Type}} \qquad \frac{\Gamma \vdash u : 0 \quad \Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash \mathrm{match}\ u\ \mathrm{return}\ A\ \mathrm{with}\ \cdot : A}$$

Figure 2 Regular type constructors.

Inductive fixed point

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Type} \vdash F : A \to \mathrm{Type} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \mu X^{A \to \mathrm{Type}}.\, F : A \to \mathrm{Type}}$$

$$\mu X^{A \to s}.\, F \equiv F\big[\, X \setminus \mu X^{A \to s}.\, F \,\big]$$

Proper indices

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma \vdash B : \mathrm{Type} \quad \Gamma, x{:}A \vdash T : \mathrm{Type} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Type}}$$

$$\frac{\Gamma \vdash \sum^{f}_{x:A} T : B \to s \quad \Gamma \vdash u : A \quad \Gamma \vdash v : T[\, x \setminus u \,]}{\Gamma \vdash (u, v)^{f}_{x:A.T} : \left(\sum^{f}_{x:A} T\right)(f[\, x \setminus u \,])}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : \left(\sum^{f}_{x:A} T\right) b \qquad \Gamma, y{:}B, z{:}\left(\sum^{f}_{x:A} T\right) y \vdash P : s \\ \Gamma, i{:}A, j{:}T\, i \vdash v : P\Big[\, y \setminus f\, i\,,\ z \setminus (i, j)^{f}_{x:A.T} \,\Big]\end{array}}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ z\ \mathrm{in}\ y\ \mathrm{return}\ P\ \mathrm{with} \\ \big|\ (i, j)^{f}_{x:A.T} \Rightarrow v\end{array} : P[\, y \setminus b\,,\ z \setminus u \,]}$$

$$\begin{array}{l}\mathrm{match}\ (u, v)^{f}_{x:A.T}\ \mathrm{as}\ z\ \mathrm{in}\ y\ \mathrm{return}\ P\ \mathrm{with} \\ \big|\ (i, j)^{f}_{x:A.T} \Rightarrow w\end{array} \rightsquigarrow w[\, i \setminus u\,,\ j \setminus v \,]$$

Figure 3 Inductive type families.

The inductive family constructors, presented in Figure 3, warrant individual discussion. First, notice that as a simplifying hypothesis, inductive families have exactly one index. This is, of course, not a limitation in expressive power as multiple indices can be encoded as a dependent sum, and the unit type allows us to write families without an index.

The construction $\mu X^{A \to \mathrm{Type}}.\, F$ constructs the *inductive fixed point* of F. It acts on type families, because indices vary through recursive calls to X. To be able to form an inductive fixed point, occurrences of X must be strictly positive in F, rules for strict positivity are given in Figure 4. The rules of Figure 4 are a simple set which suits the needs of this article, however in practice, we may want to consider strict positivity rules involving elimination rules and a finer treatment of application. Strict positivity ensures that no non-terminating term can be written without recursive fixed points, so that the guard condition suffices to enforce termination. Paradoxes which can be derived from non-positive or non-strictly positive inductive fixed points can be found in [13, Chapter 4, Section 4.2][8, Chapter 3][4, Chapter 8]. To avoid clutter, we give a presentation where inductive fixed points can be freely rolled and unrolled thanks to the conversion. An alternative can be to give an explicit term constructor for fixed points, see Section 3.4.

We will also use an inductive fixed point on nullary families, defined as:

$$\mu X^{\mathrm{Type}}.\, F = \left(\mu Y^{1 \to \mathrm{Type}}.\, F[\, X \setminus Y\, ()\,]\right)\, ()$$

from which we have that $\mu X^{\mathrm{Type}}.\, F$ can be freely rolled from or unrolled to $F\big[\, X \setminus \mu X^{\mathrm{Type}}.\, F \,\big]$.

$$\frac{}{\mathsf{sp}_X\, y} \qquad \frac{}{\mathsf{sp}_X\, 0} \qquad \frac{}{\mathsf{sp}_X\, 1}$$

$$\frac{X \text{ is fresh in } A \quad \mathsf{sp}_X\, B}{\mathsf{sp}_X\, (\prod_{x:A} B)} \qquad \frac{\mathsf{sp}_X\, A \quad \mathsf{sp}_X\, B}{\mathsf{sp}_X\, (A + B)} \qquad \frac{X \text{ is fresh in } A \quad \mathsf{sp}_X\, F}{\mathsf{sp}_X\, (\mu Y^A.\, F)}$$

$$\frac{X \text{ is fresh in } A \quad \mathsf{sp}_X\, T}{\mathsf{sp}_X\, (\lambda x^A.\, T)} \qquad \frac{\mathsf{sp}_X\, U \quad X \text{ is fresh in } t}{\mathsf{sp}_X\, (U\, t)} \qquad \frac{\mathsf{sp}_X\, B \quad B \equiv A}{\mathsf{sp}_X\, A}$$

$$\frac{X \text{ is fresh in } f \quad \mathsf{sp}_X\, A \quad \mathsf{sp}_X\, T}{\mathsf{sp}_X\, \left(\sum^f_{x:A} T\right)}$$

Figure 4 Strict positivity condition.

With inductive fixed points, we can, for instance, define the accessibility predicate. In Coq:

```
Inductive Acc (A:Type) (R:A→A→Type) (x:A) : Type :=
| acc_intro : (forall y:A, R y x → Acc A R y) → Acc A R x.
```

This type represents the generic form of termination proofs: any terminating recursive fixed point can be made structural over a proof of accessibility. In the algebraic presentation, it is defined as:

$$\begin{cases} \mathrm{Acc} = \lambda A^{\mathrm{Type}}\, R^{A\to A\to \mathrm{Type}}.\, \mu\mathrm{Acc}^{A\to\mathrm{Type}}.\, \lambda x^A.\, \prod\limits_{y:A} R\, y\, x \to \mathrm{Acc}\, y \\ \mathrm{acc}_{\mathrm{intro}} = \lambda A^{\mathrm{Type}}\, R^{A\to A\to \mathrm{Type}}\, x^A\, f^{\prod_{y:A} R\, y\, x\to \mathrm{Acc}\, A\, R\, y}.\, f \end{cases}$$

Because inductive fixed points are treated transparently, the constructor is rather trivial. However, notice how, in the definition of Acc, the parameter x is treated differently from A and R. The reason is that A and R are *uniform parameters*, in that they do not vary through recursive calls, whereas x does: it is a *non-uniform parameter*. The parameter x is, hence, represented as an index. However, such an index is not sufficient to encode types like EvenOdd.

Representing proper indices requires a new type construction, which we write $\sum^f_{x:A} T$. This construction comes from [9, 10], where it is inspired by a categorical point of view: in a sufficiently extensional setting, $\sum^f_{x:A} T$ is the right adjoint to a pullback functor. The similarity with the usual notation of dependent sum is not fortuitous, indeed we can define dependent sum as a special case of proper indexing:

$$\begin{cases} \sum\limits_{x:A} B = \left(\sum^{()}_{x:A} B\right)() \\ (u,v)_{x:A.B} = (u,v)^{()}_{x:A.B} \end{cases}$$

We also write $A \times B$ and (u,v) for $\sum_{x:A} B$ and $(u,v)_{x:A.B}$, respectively, when x is not free in B.

In the case of dependent sums, the index is trivial. When it is not, however, the pattern matching return clause P is allowed to depend on the value of the index. This is the purpose

of Coq's in-pattern. With the algebraic presentation, the in-pattern has the pleasant property of being confined to the proper indexing construction, hopefully making its meaning more explicit. The syntax differs a little from that of Coq, however: Coq renders the in clause as a pattern with the type name at the head:

```
match n as n' in EvenOdd b return P n' b with
...
end.
```

In the algebraic presentation, types not having a name, the in clause simply consists of a name for the index.

The prototype of proper indexing is the identity type, which we name Eq. In Coq:

```
Inductive Eq (A:Type) (x:A) : A → Type :=
| eq_refl : Eq A x x
```

in the algebraic presentation:

$$\begin{cases} \mathrm{Eq} = \lambda A^{\mathrm{Type}}\, x^{A}.\, \sum_{\_:1}^{x} 1 \\ \mathrm{eq}_{\mathrm{refl}} = \lambda A^{\mathrm{Type}}\, x^{A}.\, ((),())^{x}_{\_:1.1} \end{cases}$$

In fact, dependent sums and identity types are sufficient to define proper indexing. Indeed $\sum^{f}_{x:A} T$ can be redefined as:

$$\begin{cases} \sum^{f}_{x:A} T = \lambda y^{B}.\, \sum\limits_{x:A} (\mathrm{Eq}\, B\, y\, f) \times T \\ (u,v)^{f}_{x:A.T} = (u, (\mathrm{eq}_{\mathrm{refl}}\, B\, f[\, x \setminus u\,], v))_{x:A.(\mathrm{Eq}\, B\, (f[\, x \setminus u\,])\, f)\times T} \end{cases}$$

It is closer to the spirit of Coq, but in no way essential, to take a proper indexing construction rather than equality as primitive. In Morris & al [9, 10], the dependent sum and equality of the ambient type theory is used to define $\sum^{f}_{x:A} T$ which is then taken as primitive.

An other choice lies in the use of $A + B$ as primitive. It is the only type construction which allows to define a type with distinct elements. However, a common alternative is to take $\mathbb{B}$ as primitive, in which case we can define $A + B$ as:

$$\begin{cases} A + B = \sum\limits_{b:\mathbb{B}} \left| \begin{array}{l} \text{match } b \text{ with} \\ \text{true} \Rightarrow A \\ \text{false} \Rightarrow B \end{array} \right. \\ \mathrm{inl} = \lambda x^{A}.\, (\mathrm{true}, x)_{b:\mathbb{B}.\left| \begin{array}{l} \text{match } b \text{ with} \\ \text{true} \Rightarrow A \\ \text{false} \Rightarrow B \end{array} \right.} \\ \mathrm{inr} = \lambda y^{B}.\, (\mathrm{false}, y)_{b:\mathbb{B}.\left| \begin{array}{l} \text{match } b \text{ with} \\ \text{true} \Rightarrow A \\ \text{false} \Rightarrow B \end{array} \right.} \end{cases}$$

3.3 Examples

The previous section concludes the description of the predicative fragment of the algebraic presentation. We can now give definitions of the remaining inductive families we have encountered in terms of the algebraic presentation. Starting with EvenOdd:

Inductive EvenOdd : bool → **Type** :=
| eo : EvenOdd true
| es : EvenOdd false → EvenOdd true
| os : EvenOdd true → EvenOdd false.

translates, in the algebraic presentation, to:

$$\begin{cases} \mathrm{EvenOdd} = \mu\mathrm{EvenOdd}^{\mathbb{B}\to\mathrm{Type}}.\ \left(\sum_{\_:1}^{\mathrm{true}} 1 + \mathrm{EvenOdd}\,\mathrm{false}\right) + \left(\sum_{\_:1}^{\mathrm{false}} \mathrm{EvenOdd}\,\mathrm{true}\right) \\ \quad \mathrm{eo} = \mathrm{inl}\,((),\mathrm{inl}\,())_{\_:1.1}^{\mathrm{true}} \\ \quad \mathrm{es} = \lambda x^{\mathrm{EvenOdd}\,\mathrm{false}}.\,\mathrm{inl}\,((),\mathrm{inr}\,x)_{\_:1.1}^{\mathrm{true}} \\ \quad \mathrm{os} = \lambda x^{\mathrm{EvenOdd}\,\mathrm{true}}.\,\mathrm{inr}\,((),x)_{\_:1.1}^{\mathrm{false}} \end{cases}$$

Here is the definition of List in Coq:

Inductive List (A:**Type**) : **Type** :=
| nil : List A
| cons : A → List A → List A.

and in the algebraic presentation:

$$\begin{cases} \mathrm{List} = \lambda A^{\mathrm{Type}}.\,\mu\mathrm{List}^{\mathrm{Type}}.\,1 + A \times \mathrm{List} \\ \mathrm{nil} = \lambda A^{\mathrm{Type}}.\,\mathrm{inl}\,() \\ \mathrm{cons} = \lambda A^{\mathrm{Type}}\,x^{A}\,l^{\mathrm{List}\,A}.\,\mathrm{inr}\,(x,l) \end{cases}$$

Finally the definition of Tree:

Inductive Tree : **Type** :=
| node : List Tree → Tree.

translates to:

$$\begin{cases} \mathrm{Tree} = \mu\mathrm{Tree}^{\mathrm{Type}}.\,\mathrm{List}\,\mathrm{Tree} \\ \mathrm{node} = \lambda f^{\mathrm{List}\,\mathrm{Tree}}.\,f \end{cases}$$

Note that in the definition of Tree, we must β-reduce List Tree for the recursive definition to be strictly positive.

A more complex example is given by the type of lists indexed by their length, often called *vectors*:

Inductive Nat : **Type** :=
| o : Nat
| s : Nat → Nat.
Inductive Vector (A:**Type**) : Nat → **Type** :=
| vnil : Vector A o
| vcons : **forall** n, A → Vector A n → Vector A (s n).

It is encoded in the algebraic presentation as:

$$\begin{cases} \mathrm{Nat} = \mu\mathrm{Nat}^{\mathrm{Type}}.\, 1 + \mathrm{Nat} \\ \quad o = \mathrm{inl}\,() \\ \quad s = \lambda n^{\mathrm{Nat}}.\, \mathrm{inr}\, n \\ \mathrm{Vector} = \lambda A^{\mathrm{Type}}.\, \mu V^{\mathrm{Nat}\to\mathrm{Type}}.\, \lambda n^{\mathrm{Nat}}.\, \left(\sum\nolimits^{o}_{\_:1} 1\right) n + \left(\sum\nolimits^{s\, n'}_{n':\mathrm{Nat}} A \times V\, n'\right) n \\ \quad \mathrm{vnil} = \lambda A^{\mathrm{Type}}.\, \mathrm{inl}\,((),())^{o}_{\_:1.1} \\ \mathrm{vcons} = \lambda A^{\mathrm{Type}}\, n^{\mathrm{Nat}}\, a^{A}\, v^{\mathrm{Vector}\, n\, a}.\, \mathrm{inr}\,(n,(a,v))^{s\, n'}_{n':\mathrm{Nat}.A\times V\, n'} \end{cases}$$

Contrary to proper indices, the types of non-uniform parameters are allowed to be in $\Box$, this allows the definition of types such as the binary lists [12]:

```
Inductive BList (A:Type) : Type :=
| one : A → BList A
| twice : BList (A*A) → BList A
| stwice : A → BList (A*A) → BList A
```

which are rendered in the algebraic presentation as:

$$\begin{cases} \mathrm{BList} = \mu\mathrm{BList}^{\mathrm{Type}\to\mathrm{Type}}.\, \lambda A^{\mathrm{Type}}.\, A + (\mathrm{BList}\,(A\times A) + A \times (\mathrm{BList}\,(A\times A))) \\ \quad \mathrm{one} = \lambda A^{\mathrm{Type}}\, x^{A}.\, \mathrm{inl}\, x \\ \quad \mathrm{twice} = \lambda A^{\mathrm{Type}}\, l^{\mathrm{BList}\,(A\times A)}.\, \mathrm{inr}\,(\mathrm{inl}\, l) \\ \mathrm{stwice} = \lambda A^{\mathrm{Type}}\, x^{A}\, l^{\mathrm{BList}\,(A\times A)}.\, \mathrm{inr}\,(\mathrm{inr}\,(a,l)) \end{cases}$$

In Coq, where there is a hierarchy of universe, types of proper indices can be in any sort. However, a proper index whose type is in Type_i constrains the final type to be in Type_{i+1} or higher. Uniform parameters, of any type, do not constrain the type they parametrise.

Inductive types are consumed by recursive fixed points. Using the implicit unfolding of inductive fixed points, we can pattern-match over the top constructor directly. The Coq function

```
Fixpoint add (x y:Nat) : Nat :=
match y with
| o ⇒ x
| s y' ⇒ s (add x y')
end
```

is rendered in the algebraic presentation as

$$\mathrm{add} = \mathrm{fix\ add}\ x{:}\mathrm{Nat}\ y{:}\mathrm{Nat} \Rightarrow \left|\begin{array}{l} \mathrm{match}\ y\ \mathrm{as}\ \_\ \mathrm{return\ Nat\ with} \\ \mid \mathrm{inl}\,\_ \Rightarrow x \\ \mid \mathrm{inr}\, y' \Rightarrow s\,(\mathrm{add}\, x\, y') \end{array}\right.$$

3.4 Co-induction

In addition to inductive fixed points, Coq also has support for co-inductive fixed points. Co-inductive fixed points are required to be strictly positive, like inductive fixed points. We choose in this section, a presentation of co-inductive data where fixed points are explicitly introduced with a constructor. Below we will use this explicit presentation to give a variation on Coq's co-inductive fixed points.

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Type} \vdash F : A \to \mathrm{Type} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \nu X^{A\to\mathrm{Type}}.\, F : A \to \mathrm{Type}}$$

$$\frac{\Gamma \vdash \nu X^{A\to s}.\, F : A \to s \quad \Gamma \vdash i : A \quad \Gamma \vdash u : F\big[\, X \setminus \nu X^{A\to s}.\, F \,\big]\, i}{\Gamma \vdash \mathrm{forced}\, u : \big(\nu X^{A\to s}.\, F\big)\, i}$$

$$\frac{\begin{array}{lll}\Gamma \vdash \nu X^{A\to s}.\, F : A \to s & \Gamma \vdash i : A & \Gamma \vdash u : \big(\nu X^{A\to s}.\, F\big)\, i \\ \Gamma, x{:}\big(\nu X^{A\to s}.\, F\big)\, i \vdash P : s' & \Gamma, y{:}F\big[\, X \setminus \nu X^{A\to s}.\, F \,\big]\, i \vdash v : P[\, x \setminus \mathrm{forced}\, y \,] & \end{array}}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v\end{array} : P[\, x \setminus u \,]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{forced}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v\end{array} \rightsquigarrow v[\, y \setminus u \,]$$

Just like inductive data is *destructed* by a recursive fixed point operation, co-inductive data is *constructed* by a co-recursive fixed point operation, allowing co-inductive data to be infinite. The guard condition on co-recursive fixed points ensures that a finite number of unfolding will eventually produce a forced value.

$$\frac{\begin{array}{ll}\Gamma \vdash \prod_{x_1:A_1,\ldots,x_n:A_n} \big(\nu X^{A\to s}.\, F\big)\, i : s & \mathrm{coguarded}\, f\, x_1 \ldots\, x_n\, u \\ \Gamma, f{:}\prod_{x_1:A_1,\ldots,x_n:A_n} \big(\nu X^{A\to s}.\, F\big)\, i, x_1{:}A_1, \ldots, x_n{:}A_n \vdash u : \big(\nu X^{A\to s}.\, F\big)\, i & \end{array}}{\Gamma \vdash \mathrm{cofix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u : \prod_{x_1:A_1,\ldots,x_n:A_n} \big(\nu X^{A\to s}.\, F\big)\, i}$$

Co-recursive fixed-point are meant to represent infinite data: they cannot be unfolded eagerly, lest they would fail to terminate. They are unfolded only when they appear at the head of a pattern-matching expression:

$$\begin{array}{rl} & \begin{array}{l}\mathrm{match}\ (\mathrm{cofix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u)\, v_1 \ldots\, v_n\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v\end{array} \\ \rightsquigarrow & \begin{array}{l}\mathrm{match}\ u[\, f \setminus (\mathrm{cofix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u)\ ,\ x_i \setminus v_i \,]\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v\end{array}\end{array}$$

The dependent elimination rule for co-inductive fixed points asserts, in essence, that every co-inductive data is of the form $\mathrm{forced}\, u$. Even though it would be fine for inductive fixed points – this is why we could leave the unrolling to the conversion – this does not reflect well the computational aspects of co-inductive data: suspended co-recursive fixed points are values, and won't be evaluated until the context demands it. The fact that the elimination for co-inductive data claims that all values are forced gives rise to undesirable behaviour.

Take for instance the following simple co-inductive type, and data:

$$\begin{cases} T = \nu X.\, X \\ i = \mathrm{cofix}\ i \Rightarrow \mathrm{forced}\, i \end{cases}$$

So that i is effectively an infinite sequence of forced. Using the elimination principle above, it is possible to give a *closed* proof that $\mathrm{Eq}\, T\, i\, (\mathrm{forced}\, i)$:

$$\begin{array}{l}\mathrm{match}\ i\ \mathrm{as}\ x\ \mathrm{return}\ \mathrm{Eq}\, T\, x\left(\mathrm{forced}\left(\begin{array}{l}\mathrm{match}\ x\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow y\end{array}\right)\right)\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow \mathrm{eq}_{\mathrm{refl}}\, T\, y\end{array}$$

However, i and $(\text{forced}\, i)$ are not convertible, yet, as every closed proof of equality does, this proof reduces to eq_{refl}, hence should relate convertible terms. The dependent elimination rule of co-inductive fixed points compromises the type safety of the logic.

Coq uses the above dependent elimination rule for co-inductive fixed points. It was a deliberate decision made for practical purposes. Nonetheless, one may want to weaken it to avoid the incompatibility between equality and conversion. To do so, it suffices to erase the dependency of the return predicate over the matched term:

$$\frac{\Gamma \vdash \nu X^{A\to s}.\, F : A \to s \quad \Gamma \vdash i : A \quad \Gamma \vdash u : \left(\nu X^{A\to s}.\, F\right) i \quad \Gamma \vdash P : s' \quad \Gamma \vdash v : P}{\Gamma \vdash \begin{array}{l}\text{match } u \text{ return } P \text{ with} \\ \mid\ \text{forced}\, y \Rightarrow v\end{array} : P}$$

4 Prop

With all the common baggage for predicative sorts set in place, we can add impredicative sorts to the algebraic presentation. The main such sort in Coq is the sort **Prop** of propositions. The design of **Prop** is guided by *proof irrelevance*: even if it is not actually provable in Coq, different proofs of a proposition are thought of as being equal. This property is useful for program extraction: only the *computationally relevant* parts of a program need to be executed to get the final result. In other words: propositions are considered as *static* data. It is why, with disjunction and existential defined as:

```
Inductive Or (A B:Prop) : Prop :=
| or_introl : A → A ∨ B
| or_intror : B → A ∨ B.
Inductive Ex (A:Type) (P:A→Prop) : Prop :=
| ex_intro : forall x:A, P x → Ex A P.
```

the following terms are refused by type-checking:

```
match x with
| or_introl _ ⇒ true
| or_intror _ ⇒ false
end.
```

and

```
match x with
| ex_intro x _ ⇒ x
end.
```

On the other hand, it is not the case of every inductive type defined in **Prop**, that they cannot be eliminated into **Type**. Conjunction and falsity are two counter-examples:

```
Inductive False : Prop := .
Inductive And (A B:Prop) : Prop :=
| conj : A → B → A ∧ B.
```

Coq allows elimination over these two propositions into **Type**, and both following terms are well-typed:

```
match x return Bool with end.
```

and

```
match x with
| conj _ _ ⇒ true
end.
```

The object of this section is to make syntactically explicit what happens when an inductive type of Coq is declared to be of sort **Prop**. The description elaborated in this section has strong similarities with the system of bracket-types proposed by Awodey & Bauer [2]. They describe the propositions as the subset of types with at most one element, and introduce a left adjoint, written as brackets, to the inclusion of propositions into types. We will reuse their notation, even though, in our intensional setting, T:**Prop** does not enforce that T has a most one element, and the bracketing operation does not properly form an adjunction with the inclusion from **Prop** to **Type**.

4.1 Impredicativity

Let us start by introducing the new sort Prop in the algebraic presentation:

$$\frac{}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}}$$

As in [2], propositions form a subset of types. Coq has a subtyping rule (also known as *cumulativity*) to make the inclusion transparent. We will, however, render it with a syntactic construct:

$$\frac{\Gamma \vdash A : \mathrm{Prop}}{\Gamma \vdash \{A\} : \mathrm{Type}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \mathrm{prf}\, u : \{A\}}$$

$$\frac{\Gamma \vdash u : \{A\} \quad \Gamma, x{:}\{A\} \vdash P : s \quad \Gamma, y{:}A \vdash v : P[\,x \setminus \mathrm{prf}\, y\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with}\\ \mid\ \mathrm{prf}\, y \Rightarrow v\end{array} : P[\,x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{prf}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with}\\ \mid\ \mathrm{prf}\, y \Rightarrow v\end{array} \rightsquigarrow v[\,y \setminus u\,]$$

This definition simply makes $\{A\}$ a synonym of A, except of sort Type. It is strictly positive in A:

$$\frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, \{A\}}$$

The fact that Prop is impredicative – *i.e.* supports the following product formation rules:

$$\frac{\Gamma \vdash A : s \quad \Gamma, x{:}A \vdash B : \mathrm{Prop}}{\Gamma \vdash \prod_{x:A} B : \mathrm{Prop}}$$

is easily understood in terms of proof irrelevance. Indeed, if for all x, B has at most one element, so has the product over x. Even though it uses functional extensionality, which is not provable.

$$\frac{}{\Gamma \vdash 0 : \mathrm{Prop}} \qquad \frac{}{\Gamma \vdash 1 : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash A : \mathrm{Prop} \quad \Gamma \vdash B : \mathrm{Prop} \quad \Gamma, x{:}A \vdash T : \mathrm{Prop} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Prop}}$$

Figure 5 Singleton rules.

4.2 Singleton rules

The types which (ideally) preserve the proof irrelevance property are sometimes called singleton types in the setting of Coq. In our algebraic presentation, they correspond to inductive type family constructors with extra formation rules to make them preserve propositions. The rules are shown in Figure 5.

This elucidates why Coq allows elimination over False and And into arbitrary type: False is implemented as 0 and And A B and $A \times B$. The elimination rules being unchanged, pattern-matching over proofs of False and And are unrestricted. Because restricted pattern-matching is often seen as the default, singleton types are said to enjoy *singleton elimination*.

Remark that, proofs of propositions being uninformative, there is essentially nothing to be gained from depending on, or being indexed over a proposition. In consequence, the type formation rule for proper indexing in Figure 5 is only useful, in practice, for the subcase of cartesian product.

Coq actually implements two other singleton rules. The first one is for inductive fixed points. In our algebraic presentation:

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Prop} \vdash F : A \to \mathrm{Prop} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \mu X^{A \to \mathrm{Prop}}.\, F : A \to \mathrm{Prop}}$$

It allows to type the accessibility predicate Acc in Prop. This rule is sound in that fixed points indeed preserve proof irrelevance in presence of functional extensionality. It is also very useful for extraction: structural recursion over Acc allows the definition of functions whose termination cannot be proved automatically by the guard condition. However, the proof is no longer needed to ensure termination in the target languages of extraction. In this sense, at least, it is static data.

The last singleton rule allows properly indexed families in Prop (not how it is stronger than the rule dependent sum of Figure 5):

$$\frac{\Gamma \vdash A : \mathrm{Prop} \quad \Gamma \vdash B : \mathrm{Type} \quad \Gamma, x{:}A \vdash T : \mathrm{Prop} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Prop}}$$

It turns the identity type Eq into a proposition. It is known to be sound to accept that Eq is proof irrelevant [6]. It is also useful for extraction, as equal types, in a closed environment, are extracted to the same type. Hence a program may safely eliminate over Eq knowing that it will not affect the performances of the extracted code. In Coq, the index B in the rule above can be of any sort Type_i, however, this wisdom has been challenged in recent years with the formulation of the *univalence principle* [15], of which a simple consequence is that Eq is *not* proof irrelevant at every type. Indeed, some extracted Coq programs written assuming the univalence principle crash.

To correct for the univalence principle, the singleton rule for proper indices can be simply dropped; but it can also be restricted to the lowest sort: $B : \mathrm{Type}_0$. More precisely the

conjunction of the univalence principle and the proof irrelevance principle is consistent as long as the singleton rule of proper indices is restricted to sorts s such that there is no sort s' other than Prop such that $s' : s$. Because, if such a sort s' exists, $\mathbb{B} : s'$ and by univalence, $\mathrm{Eq}\,\mathbb{B}$ has two distinct elements contradicting proof irrelevance.

For types which do not enjoy singleton elimination, turning them into propositions means restricting their elimination. We achieve this effect by adding a single type construction coercing from Type to Prop:

$$\frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash [A] : \mathrm{Prop}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \langle u \rangle : [A]}$$

$$\frac{\Gamma \vdash u : [A] \qquad \Gamma, x{:}[A] \vdash P : \mathrm{Prop} \qquad \Gamma, y{:}A \vdash v : P[\, x \setminus \langle y \rangle\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle \Rightarrow v\end{array} : P[\, x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \langle u \rangle\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle \Rightarrow v\end{array} \rightsquigarrow v[\, y \setminus u\,]$$

The important rule is the elimination rule, where the return clause is limited to be of sort Prop, whereas every other type construction can be eliminated to any sort. Apart from this restriction $[A]$ is a synonym of A, except in Prop. In [2], the type theory is extensional, in that the identity type and the conversion relation coincide. The elimination rules for bracket is much finer and reflects precisely the fact that propositions are proof-irrelevant. In an intensional type theory, restricting with respect to sorts approximates this behaviour: even if we constrained propositions to be proof-irrelevant, not every proof irrelevant type will have type Prop. The bracketing construction is also strictly positive:

$$\frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, [A]}$$

It is actually possible, using only the impredicative dependent product to define a bracketing operation: $\prod_{P:\mathrm{Prop}} (A \to P) \to P$. Like $[A]$ it behaves as A except it can only be used to form a proposition. However, the impredicative encoding is positive but not strictly, which motivates the introduction of the extra construction.

4.3 Examples

The logical connectives can be defined as follows:

$$\begin{cases} \mathrm{False} = 0 \\ \mathrm{And} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}.\, A \times B \\ \mathrm{pair} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}\, x^A\, y^B.\, (x, y) \\ \mathrm{Or} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}.\, [A + B] \\ \mathrm{or}_{\mathrm{introl}} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}\, x^A.\, \langle \mathrm{inl}\, x \rangle \\ \mathrm{or}_{\mathrm{intror}} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}\, y^B.\, \langle \mathrm{inr}\, y \rangle \\ \mathrm{Ex} = \lambda A^{\mathrm{Type}}\, P^{A \to \mathrm{Prop}}.\, \left[\sum_{x:A} P\, x \right] \\ \mathrm{ex}_{\mathrm{intro}} = \lambda A^{\mathrm{Type}}\, P^{A \to \mathrm{Prop}}\, x^A\, p^{P\, x}.\, \langle (x, p)_{x:A.P} \rangle \end{cases}$$

Note how, because of the brackets, existentials and disjunctions are prohibited from being eliminated to non-propositional types. Thanks to the singleton rules, however, conjunction and falsity do not require brackets.

As a final example, consider the type Ascending n p of ascending sequences of integers between p and n defined by mutual recursion with the proposition Ge m p which stands from m is greater than or equal to p:

```
Inductive Ascending : Nat → Nat → Type :=
| top : forall n, Ascending n n
| up : forall n p m, Ge m (s p) → Ascending n m → Ascending n p
with Ge : Nat → Nat → Prop :=
| ascend : forall m p, Ascending m p → Ge m p.
```

As Ascending has type **Type**, whereas Ge has type **Prop**, the translation to a single inductive type is not as straightforward as Even and Odd. The translation requires the use of brackets around the recursive calls:

$$\begin{cases} \text{AscendingGe} = & \mu X^{(\mathrm{Nat}\times\mathrm{Nat})+(\mathrm{Nat}\times\mathrm{Nat})\to\mathrm{Type}}.\, \lambda i^{(\mathrm{Nat}\times\mathrm{Nat})+(\mathrm{Nat}\times\mathrm{Nat})}. \\ & \quad \left(\sum_{n:\mathrm{Nat}}^{\mathrm{inl}\,(n,n)} 1\right) i \\ & + \quad \left(\sum_{j:\mathrm{Nat}\times\mathrm{Nat}}^{\mathrm{inl}\,j} \sum_{m:\mathrm{Nat}} \left[X\left(\mathrm{inr}\,(m, s\,(\pi_2\,j))\right)\right] \times X\left(\mathrm{inl}\,(\pi_1\,j, m)\right)\right) i \\ & + \quad \left(\sum_{j:\mathrm{Nat}\times\mathrm{Nat}}^{\mathrm{inr}\,j} X\left(\mathrm{inl}\,j\right)\right) i \\ \text{Ascending} = & \lambda n\, p.\, \text{AscendingGe}\left(\mathrm{inl}\,(n,p)\right) \\ \text{Ge} = & \lambda m\, p.\, \left[\text{AscendingGe}\left(\mathrm{inr}\,(m,p)\right)\right] \end{cases}$$

5 Impredicative Set

In addition to the impredicative sort Prop, Coq has a sort Set which is predicative by default but can be turned impredicative with a flag. Where Prop is meant to be used in the context of separating static and dynamic information, the spirit of the impredicative sort Set is to be as powerful as possible without being inconsistent. In the algebraic presentation, that means being stable by every construction except dependent sums with the first projection in an arbitrary sort (*strong sums*).

To mirror the optional nature of the impredicativity of Set, the rules for a predicative sort Set are given in Figure 6; to turn impredicativity on, the rules of Figure 7 must be used *in addition* to those of predicative Set. This presentation makes immediately apparent that impredicative Set is an extension of predicative Set, in that every program of the latter typechecks in the former.

The rules of Set are the same as those of Prop, with the exception of $A + B$ which is in Set when both A and B are – even with predicative Set. Hence, there are types in Set with several elements – *e.g.* $\mathbb{B}$. As a consequence, the bracketing operation which coerces types in Type to Set does not enjoy an explanation in terms of proof irrelevance, as was the case in Prop. As a matter of fact, there is no clear set-theoretical description at all. A close cousin of Set bracketing, however, can be found in homotopy type theory [15], where, roughly, groupoids are *truncated* to sets through a quotient of their homsets by the total relation.

$$\frac{}{\Gamma \vdash \mathrm{Set} : \mathrm{Type}} \qquad \frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma, x{:}A \vdash B : \mathrm{Set}}{\Gamma \vdash \prod_{x:A} B : \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma, x{:}A \vdash P : \mathrm{Set}}{\Gamma \vdash \sum_{x:A} P : \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma \vdash B : \mathrm{Set}}{\Gamma \vdash A + B : \mathrm{Set}} \qquad \frac{}{\Gamma \vdash 1 : \mathrm{Set}} \qquad \frac{}{\Gamma \vdash 0 : \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Set} \vdash F : A \to \mathrm{Set} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \mu X^{A\to\mathrm{Set}}.\, F : A \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma \vdash B : \mathrm{Set} \quad \Gamma, x{:}A \vdash T : \mathrm{Set} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Set} \vdash F : A \to \mathrm{Set} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \nu X^{A\to\mathrm{Set}}.\, F : A \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set}}{\Gamma \vdash \{A\}_{\mathrm{Set}} : \mathrm{Type}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \mathrm{elt}\, u : \{A\}_{\mathrm{Set}}}$$

$$\frac{\Gamma \vdash u : \{A\}_{\mathrm{Set}} \quad \Gamma, x{:}\{A\}_{\mathrm{Set}} \vdash P : s \quad \Gamma, y{:}A \vdash v : P[\,x \setminus \mathrm{elt}\, y\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with}\\ \mid\ \mathrm{elt}\, y \Rightarrow v\end{array} : P[\,x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{elt}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with}\\ \mid\ \mathrm{elt}\, y \Rightarrow v\end{array} \rightsquigarrow v[\,y \setminus u\,]$$

$$\frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, \{A\}_{\mathrm{Set}}}$$

Figure 6 Rules for predicative Set.

$$\frac{\Gamma \vdash A : s \quad \Gamma, x{:}A \vdash B : \mathrm{Set}}{\Gamma \vdash \prod_{x:A} B : \mathrm{Set}} \qquad \frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, [A]_{\mathrm{Set}}}$$

$$\frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash [A]_{\mathrm{Set}} : \mathrm{Set}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \langle u \rangle_{\mathrm{Set}} : [A]_{\mathrm{Set}}}$$

$$\frac{\Gamma \vdash u : [A]_{\mathrm{Set}} \quad \Gamma, x{:}[A]_{\mathrm{Set}} \vdash P : \mathrm{Set} \quad \Gamma, y{:}A \vdash v : P[\, x \setminus \langle y \rangle_{\mathrm{Set}}\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle_{\mathrm{Set}} \Rightarrow v\end{array} \ : P[\, x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \langle u \rangle_{\mathrm{Set}}\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle_{\mathrm{Set}} \Rightarrow v\end{array} \ \rightsquigarrow v[\, y \setminus u\,]$$

Figure 7 Rules for impredicative Set.

6 Conclusion

The algebraic presentation of Coq makes the conversion between sorts explicit. The toplevel inductive definitions of Coq can be understood as implicitly inserting canonical bracketing operations when an inductive type is declared inside an impredicative sort but should be of a different sort due to its form; and inserting type coercion from a smaller sort to a bigger sort when applying a cumulativity rule.

Monolithic type definitions like in Coq have a number of advantages over the algebraic presentation, they boil down to better type errors due to naming, better type inference and better memory representation due to n-ary sums and products. However, the value of the implicit coercions between sorts is less clear. In particular, the bracketing operation to impredicative sorts is probably a better guide for program extraction than the current method of figuring whether or not a given type is a proposition, which interacts badly with universe polymorphism [7]. Explicit coercions for extraction are also in the spirit of [5].

All of the algebraic type constructors can actually be defined in Coq, except the two fixed-points because there is no way to abstract over strictly positive type families. So is it clear that expressions of the algebraic presentation which do not use inductive or co-inductive fixed points can be translated into Coq. Occurrences of the fixed points in a type must be λ-lifted and given a toplevel name. Some care must be given to avoiding the duplication of such definitions otherwise types which must be convertible for the expression to typecheck, might be seen as different in the Coq translation. Apart from this technicality, translation from the algebraic presentation to Coq is straightforward. We claim that, at least if we extend the algebraic presentation to a hierarchy of universes and the strict positivity condition is made a bit more fine-grained, Coq terms can be, conversely, translated into the algebraic presentation.

References

1 Andreas Abel. *A polymorphic lambda-calculus with sized higher-order types.* PhD thesis, 2006.

2 Steve Awodey and Andrej Bauer. Propositions as [Types]. *Journal of Logic and Computation*, 14(4):447–471, August 2004.

3 Henk Barendregt. Lambda calculus with types. *Handbook of logic in computer science*, 1992.

4 Bruno Barras. Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families. *Thèse d'Habilitation*, 2013.

5 Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. *Foundations of Software Science and Computational Structures*, 4962:365–379, 2008.

6 Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. 2011.

7 Pierre Letouzey and Bas Spitters. Implicit and noncomputational arguments using monads. pages 1–16, 2005.

8 Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, 2007.

9 Peter Morris and Thorsten Altenkirch. Constructing strictly positive families. *CATS '07 Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 111–121, 2007.

10 Peter Morris, Thorsten Altenkirch, and Neil Ghani. A universe of strictly positive families. *IInternational journal of foundations of computer science*, pages 83–107, 2009.

11 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

12 Chris Okasaki. *Purely functional data structures*. 1999.

13 Christine Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. PhD thesis, Université Claude Bernard-Lyon I, 1996.

14 The Coq development team. The Coq Proof Assistant.

15 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*.

A Direct Version of Veldman's Proof of Open Induction on Cantor Space via Delimited Control Operators*†

Danko Ilik[1] **and Keiko Nakata**[2]

1 **Research Center for Computer Science and Information Technologies**
Macedonian Academy of Sciences and Arts
Skopje, Macedonia
danko.ilik@gmail.com

2 **Institute of Cybernetics**
Tallinn University of Technology
Tallinn, Estonia
keiko@cs.ioc.ee

Abstract

First, we reconstruct Wim Veldman's result that Open Induction on Cantor space can be derived from Double-negation Shift and Markov's Principle. In doing this, we notice that one has to use a countable choice axiom in the proof and that Markov's Principle is replaceable by slightly strengthening the Double-negation Shift schema. We show that this strengthened version of Double-negation Shift can nonetheless be derived in a constructive intermediate logic based on delimited control operators, extended with axioms for higher-type Heyting Arithmetic. We formalize the argument and thus obtain a proof term that directly derives Open Induction on Cantor space by the shift and reset delimited control operators of Danvy and Filinski.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.3 Studies of Program Constructs

Keywords and phrases open induction, axiom of choice, double negation shift, Markov's principle, delimited control operators

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.188

1 Introduction

Let X be a set with an equality relation $=_X$ and a binary relation $<_X$. We denote by X^ω and X^* the set of infinite sequences, or *streams*, over X and the set of finite sequences over X, respectively. Let elements of X^ω be denoted by Greek letters α, β, γ, let natural numbers be denoted by n, k, l, m, and let $\overline{\alpha}n$ denote the finite sequence $\langle \alpha(0), \alpha(1), \ldots, \alpha(n-1) \rangle$, i.e., the initial segment of length n of the sequence α.

The lexicographic extension $<_{X^\omega}$ of $<_X$ is a binary relation on streams, defined by

$$\alpha <_{X^\omega} \beta \text{ iff } \exists n(\overline{\alpha}n =_{X^*} \overline{\beta}n \wedge \alpha(n) <_X \beta(n)),$$

where $=_{X^*}$ denotes the equality relation induced from $=_X$ by element-wise comparison, i.e., $p =_{X^*} q$ iff p and q are of the same length and element-wise equal with respect to $=_X$.

* D. Ilik's work is covered by a Kurt Gödel Research Prize Fellowship 2011.
† K. Nakata acknowledges the ERDF funded EXCS project, the Estonian Ministry of Education and Research research theme no. 0140007s12, and the Estonian Science Foundation grant no. 9398.

A non-empty subset U of X^ω is called *open* if there is an enumeration $\pi : \mathbb{N} \to X^*$ which can approximate U, in the sense that membership in U can be defined[1] by

$\alpha \in U$ iff $\exists n \exists k(\overline{\alpha}n =_{X^*} \pi(k))$.

The *Principle of Open Induction on* X^ω (equipped with $<_X$ and $=_X$) is the following statement, for U open:

$$\forall \alpha\, (\forall \beta <_{X^\omega} \alpha\, (\beta \in U) \to \alpha \in U) \to \forall \alpha (\alpha \in U). \tag{OI-X}$$

One immediately sees that OI-X has the form of a well-founded induction principle. However, one should note that, even for the simple choice of $X = \{0, 1\}$ equipped with the usual decidable order and equality relation, an open set U is generally uncountable, and the lexicographic ordering $<_{X^\omega}$ is not well-founded!

The utility of this principle has been recognized by Raoult [15] who gave, using OI-X, a new version of Nash-Williams' proof of Kruskal's theorem that does not explicitly use the Axiom of Dependent Choice[2].

OI-X was introduced in the context of Constructive Mathematics by Coquand [4]. He proved OI-X by relativized Bar Induction, and also first considered separately the version for X^ω being the Cantor space [5].

Berger [3] showed that OI-X in higher-type Arithmetic, where X can be any type ρ, is classically equivalent to the Axiom of Dependent Choice (DC) for the type ρ. He also gave a modified realizability interpretation of OI-X by a schema of Open Recursion, and showed that, unlike DC, OI-X is closed under double-negation- and A-translation – this means that there is a simple way to extract open-recursive programs from classical proofs of Π^0_2-statements that use DC or OI-X.

In the context of Constructive Reverse Mathematics, in a series of lectures [18], Veldman showed that Open Induction for Cantor space is equivalent to Double-negation Shift,

$$\forall n \neg\neg A(n) \to \neg\neg \forall n A(n) \quad \text{(for any formula } A(n)\text{)}, \tag{DNS}$$

in presence of Markov's Principle,

$$\neg\neg \exists n A_0(n) \to \exists n A_0(n) \quad \text{(for a decidable } A_0(n)\text{)}. \tag{MP}$$

Given that it is possible to obtain proofs for both MP [9] and DNS [11] using constructive logical systems based on delimited control operators, it is a natural next step to attempt to provide a direct constructive proof of OI for Cantor space based on delimited control operators. This is what we do in this paper.

The remainder of the paper is organized as follows. In Section 2, we reconstruct in detail Veldman's argument that proves OI on Cantor space from DNS and MP via the principle EnDec. In Section 3, we recall the logical system $\mathrm{MQC}_+(S)$ from [11] that is able to prove a strengthened version DNS_S of DNS using delimited control operators. DNS_S allows us to prove (a minimal logic version of) EnDec without explicitly using MP. In Section 4, we give a formalized proof term for OI on Cantor space in a variant of HA^ω based on the logical system $\mathrm{MQC}_+(S)$. In the concluding Section 5, we explain the current limitation of our approach for extracting proofs from programs and we mention directly related works.

[1] For simplicity, we exclude the possibility of $U = \emptyset$, so that we may take *total* enumerations π, rather than partial enumerations, sending $\mathbb{N}$ to $\mathrm{option}(X^*)$.

[2] Raoult proves OI-X using Zorn's Lemma.

2 From DNS and MP to Open Induction for Cantor Space

We will consider the case $X = \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ with $0 <_{\mathbb{B}} 1$ and $0 =_{\mathbb{B}} 0$, $1 =_{\mathbb{B}} 1$, that is, Open Induction *on Cantor space*, OI-$\mathbb{B}$. We will show that OI-$\mathbb{B}$ is provable from DNS, MP, and AC!$^{0,\mathbb{B}}$, where

$$\forall x^{\mathbb{N}} \exists! y^{\mathbb{B}} A(x, y) \to \exists f^{\mathbb{N}\to\mathbb{B}} \forall x^{\mathbb{N}} A(x, f(x)) \tag{AC!$^{0,\mathbb{B}}$}$$

is a restriction of the Axiom of Unique Countable Choice (also known as Countable Comprehension). All the arguments of this section take place in plain intuitionistic logic; if a principle that is not intuitionistically derivable is used, that is explicitly noted.

In addition to the already introduced notational conventions, let p, q, r, s denote finite binary sequences (bit-strings), $\mathbb{B}^*$, and let $p * q$ denote the concatenation of p and q. For a natural number k, $\mathbb{B}^k$ denotes the set of bit-strings of length k. Concrete bit-strings are constructed using the notation $\langle\cdot\rangle$, e.g. $\langle\rangle$ denotes an empty sequence, $\langle 0\rangle$ the bit-string of length 1 that contains a 0, $\langle 1, 1, 1, 1\rangle$ the bit-string that contains four 1's, etc. Thus $p * \langle 0\rangle$ means that a zero bit is appended at the end of p. The function $\mathrm{len}(p)$ computes the length of p. Analogously to the initial segment function $\overline{\alpha}n$ on infinite sequences, we denote by $\overline{p}n$ the initial segment function on finite sequences, with default value $\overline{p}n := p$ when $n > \mathrm{len}(p)$. Instead of writing $<_{\mathbb{B}^\omega}$ and $=_{\mathbb{B}^*}$, we simply write $<$ and $=$. We abbreviate $(S_1 \to S_2) \wedge (S_2 \to S_1)$ to $(S_1 \leftrightarrow S_2)$. We may write $n \notin A$ to mean $\neg(n \in A)$.

By a Σ-*formula*, we mean a formula built only from existential quantifiers (over the set $\mathbb{N}$), disjunction, conjunction, and the equality symbol "=" for $\mathbb{N}$. This definition is equivalent to the usual definition of Σ^0_1-formula if the language has all the primitive recursive symbols, as is the case for the system from Section 4.

We say that a set $B \subseteq \mathbb{N}$ is *enumerable* when the membership in B is a Σ-formula, i.e., $n \in B$ is defined as $S(n)$ for a Σ-formula S. Equivalently[3], B is enumerable when B is given by a function $f : \mathbb{N} \to \mathbb{N}$ such that $n \in B$ is a notation for $\exists m(f(m) = n + 1)$. A set $B \subseteq \mathbb{N}$ is *decidable* when we have that $\forall n(n \in B \vee n \notin B)$[4].

Veldman introduced the following principle.

▶ **Axiom 1** (EnDec)**.** Assume $B \subseteq \mathbb{N}$ is enumerable. Let, for any decidable $C \subseteq B$, we have that, if $\exists m(m \notin C)$, then $\exists m(m \notin C \wedge m \in B)$. Then $\mathbb{N} \subseteq B$ (and hence B is decidable).

Note that EnDec holds classically, since classically any B is decidable, so we may set $C := B$ to obtain $\mathbb{N} \subseteq B$. Our interest in EnDec here is because it is a stepping stone to proving OI-$\mathbb{B}$.

▶ **Theorem 1.** *Assuming AC!$^{0,\mathbb{B}}$, EnDec implies Open Induction on Cantor space.*

Proof. Let A be a non-empty open subset of Cantor space[5] i.e., there exists $\pi : \mathbb{N} \to \mathbb{B}^*$ such that "$\alpha \in A$" is a notation for $\exists l, m(\overline{\alpha}l = \pi(m))$. Let also A be *progressive*, that is,

$$\forall \alpha(\forall \beta < \alpha(\beta \in A) \to \alpha \in A).$$

We want to show that $\forall \alpha(\alpha \in A)$. Define $B \subseteq \mathbb{B}^*$ as

$$p \in B \text{ iff } \exists k \forall q \in \mathbb{B}^k \exists l, m(\overline{p * q}\, l = \pi(m))$$

[3] "Equivalent" in the system from Section 4.
[4] In some literature, our "decidable" is called "detachable".
[5] The progressiveness on Cantor space in fact ensures that A is non-empty.

such that p is in B if p is "uniformly barred" by π. That is, $p \in B$ if there exists k such that any extension of p by a finite bit-string of length k is covered by $\pi(m)$ for some m[6].

It suffices to show $\langle\rangle \in B$ for the empty bit-string $\langle\rangle$, since we then know that π covers the entire Cantor space. We show that B is actually equal to $\mathbb{B}^*$, using EnDec. Notice that $\mathbb{B}^*$ is bijective to $\mathbb{N}$ by primitive recursive functions and B is enumerable[7], hence we may transport EnDec from $\mathbb{N}$ to $\mathbb{B}^*$. It is left to show that, for any decidable subset $C \subseteq B$, if $\exists q(q \notin C)$, then $\exists r(r \notin C \wedge r \in B)$.

Suppose that such C and q are given. If $\langle\rangle \in C \subseteq B$, then we have that $q \in B$. So we are done. We assume $\langle\rangle \notin C$. Since C is decidable, we can construct α, using AC!$^{0,\mathbb{B}}$, such that

$$\alpha(n) := \begin{cases} 0 & \text{, if } \overline{\alpha}n * \langle 0\rangle \notin C \\ 1 & \text{, if } \overline{\alpha}n * \langle 0\rangle \in C \text{ and } \overline{\alpha}n * \langle 1\rangle \notin C \\ 0 & \text{, if } \overline{\alpha}n * \langle 0\rangle \in C \text{ and } \overline{\alpha}n * \langle 1\rangle \in C \end{cases}$$

The sequence α tries to stay outside of C for as long as possible and tries to be minimal. It first tries to "turn left" (value 0). If it was not possible, i.e., $\overline{\alpha}n * \langle 0\rangle \in C$, then it tries to "turn right" (value 1). If neither was possible, then it defaults to "turning left". One may notice that if α fails to stay outside of C at $n+1$, i.e., $\overline{\alpha}n * \langle 0\rangle \in C$ and $\overline{\alpha}n * \langle 1\rangle \in C$, then we have $\overline{\alpha}n \in B$. This fact, a manifestation of the compactness of Cantor space, will be used later in the proof.

Now, we can find a prefix of α that is in B but not in C, by following α up to the first point where it enters B. Let us first prove that α is in A, which guarantees that α has a prefix in B, hence that α will enter B. We use progressiveness of A. Let $\beta < \alpha$ i.e., $\exists n(\overline{\beta}n = \overline{\alpha}n \wedge \beta(n) = 0 \wedge \alpha(n) = 1)$. We have to show $\beta \in A$. By construction of α, $\alpha(n) = 1$ is only possible if $\overline{\alpha}n * \langle 0\rangle \in C$ and $\overline{\alpha}n * \langle 1\rangle \notin C$. Noticing that $\overline{\beta}(n+1) = \overline{\beta}n * \langle 0\rangle = \overline{\alpha}n * \langle 0\rangle$, this yields $\overline{\beta}(n+1) \in C \subseteq B$. We conclude that $\beta \in A$, which was to be shown.

From $\alpha \in A$, we obtain l, m such that $\overline{\alpha}l = \pi(m)$. We finish the proof by proving the following more general statement by induction

$$\forall n \leq l\,(\overline{\alpha}(l-n) \notin C \to \exists l'(\overline{\alpha}l' \notin C \wedge \overline{\alpha}l' \in B))\,.$$

Indeed, since we have $\langle\rangle \notin C$, by instantiating the above statement with $n := l$, we obtain p such that $p \notin C$ and $p \in B$.

In the base case, $n = 0$, we have that $\overline{\alpha}l \notin C$ by the hypothesis and that $\overline{\alpha}l \in B$ (from $\alpha \in A$); so we set $l' := l$. In the induction case for $n+1$ we consider three possibilities:

1. if $\overline{\alpha}(l-(n+1)) * \langle 0\rangle \notin C$, then $\overline{\alpha}(l-n) = \overline{\alpha}(l-(n+1)+1) = \overline{\alpha}(l-(n+1)) * \langle 0\rangle \notin C$ and we close the case by induction hypothesis;
2. similarly, if $\overline{\alpha}(l-(n+1)) * \langle 0\rangle \in C$ and $\overline{\alpha}(l-(n+1)) * \langle 1\rangle \notin C$, then $\overline{\alpha}(l-n) = \overline{\alpha}(l-(n+1)+1) = \overline{\alpha}(l-(n+1)) * \langle 1\rangle \notin C$, and we close the case by induction hypothesis;
3. if $\overline{\alpha}(l-(n+1)) * \langle 0\rangle \in C$ and $\overline{\alpha}(l-(n+1)) * \langle 1\rangle \in C$, then we get that $\overline{\alpha}(l-(n+1)) \in B$ as we noted earlier. Recalling that we also have $\overline{\alpha}(l-(n+1)) \notin C$ by hypothesis, we can set $l' := l-(n+1)$.

The first two cases could be merged into one, verifying only whether $\overline{\alpha}(l-(n+1)+1) \notin C$. ◀

[6] A bit-string p is *covered* by q if, as a bit-string, q is a prefix of p, or the open set given by p is covered by the open set given by q.

[7] B is enumerable because it is defined by a Σ-formula: the bounded universal quantifier "$\forall q \in \mathbb{B}^k$" does not pose a problem, since it could be interpreted as a bounded minimization operator, for example like in §3.5 of [12].

▶ Remark. In the previous proof, we used $\mathrm{AC!}^{0,\mathbb{B}}$ when constructing the sequence α by course-of-values recursion using the choice function extracted from the decidability of C. Since the principle EnDec is classically valid, not using a choice axiom would mean that one can reduce OI-$\mathbb{B}$ (and, using Berger's results [3], also Dependent Choice for $\mathbb{B}$) to plain classical logic without choice[8].

We now consider the principle of Double-negation Shift (DNS), which is independently important because it allows to interpret the double-negation translation of the Axiom of Countable Choice [16]. Following Veldman, we find it useful to consider the following variant of DNS.

▶ **Axiom 2** ($\mathsf{DNS}^{\mathsf{V}}$). $\neg\neg\forall n(A(n) \vee \neg A(n))$, for any formula $A(n)$.

▶ Remark. The proof of equivalence between DNS and $\mathrm{DNS}^{\mathrm{V}}$ is analogous to the proof of equivalence between the law of double-negation elimination (DNE) and the law of excluded middle (EM). In minimal logic, which is intuitionistic logic without the rule of $\bot$-elimination (*ex falso quodlibet*), EM is weaker than DNE [1]. We expect a similar result for DNS, i.e., that $\mathrm{DNS}^{\mathrm{V}}$ is weaker than DNS in minimal logic.

When quantifier-free formulas and decidable formulas coincide, as in Arithmetic, we may state Markov's Principle using Σ-formulas.

▶ **Axiom 3** (MP). For any Σ-formula S, we have that $\neg\neg S \to S$.

We can now prove EnDec from $\mathrm{DNS}^{\mathrm{V}}$and MP.

▶ **Theorem 2.** *$DNS^V$and MP together imply EnDec.*

Proof. Let the premises of EnDec hold. Given $n \in \mathbb{N}$, we have to prove $n \in B$, which is a Σ-formula. We are entitled to apply MP. Now, we have to show that $\neg\neg(n \in B)$. Suppose $\neg(n \in B)$. Thanks to $\mathrm{DNS}^{\mathrm{V}}$, it suffices to prove $\bot$ assuming moreover that B is decidable, i.e., $\forall n(n \in B \vee \neg(n \in B))$. We use the premise of EnDec by taking $C := B$ and recalling that we have $\neg(n \in B)$. This gives us $\exists m(m \in B \wedge \neg(m \in B))$, from which we derive $\bot$. ◀

3 A Constructive Logic Proving EnDec

In this section, we recall the logical system $\mathrm{MQC}_+(S)$ from [11], and show that EnDec is provable in $\mathrm{MQC}_+(S)$ (with a suitably instantiated parameter S), without an explicit use of MP, thanks to the slightly stronger form of DNS that $\mathrm{MQC}_+(S)$ proves.

$\mathrm{MQC}_+(S)$ is a pure predicate logic system, parameterized over a closed Σ-formula S, that, in addition to the usual rules of minimal intuitionistic predicate logic, adds two rules for proving the Σ-formula S [9]. The rule "reset",

$$\frac{\Gamma \vdash_S S}{\Gamma \vdash_\diamond S}\ \#\ (\text{"reset"}),$$

sets a marker (under the turnstile) meaning that one wants to prove S. Once the marker is set, one can use the "shift" rule,

[8] Classically $\mathrm{AC!}^{0,\mathbb{B}}$ is equivalent to Dependent Choice for $\mathbb{B}$ (in Berger's formulation), hence that we only use $\mathrm{AC!}^{0,\mathbb{B}}$ is not a concern.

[9] In the context of $\mathrm{MQC}_+(S)$, Σ-formulas coincide with formulas without $\forall$ and $\to$.

$$\frac{\Gamma, A \Rightarrow S \vdash_S S}{\Gamma \vdash_S A}\,\mathcal{S}\ (\text{“shift”}),$$

to prove by a principle related to double-negation elimination from classical logic. The idea is to internalize in the formal system the fact, known from Friedman-Dragalin's A-translation, that a classical proof of a Σ^0_1-formula can be translated to an intuitionistic proof of the same formula, showing that classical proofs of such formulas are in fact constructive. The first system built around this internalization idea was Herbelin's [9] with the power to derive Markov's Principle. It satisfies, like $\mathrm{MQC}_+(S)$, the disjunction and existence properties, characteristic of plain intuitionistic logic.

The names “shift” and “reset” come from the computational intention behind the normalization of these proof rules, Danvy and Filinski's delimited control operators [6, 7, 8]. These operators were developed in the theory of programming languages with the aim of enabling to write continuation-passing style (CPS) programs in so-called *direct style*. Since CPS transformations are known to be one and the same thing as double-negation translations [14], one can think of shift/reset in Logic as enabling to prove *directly* theorems whose double-negation translation is intuitionistically provable. In order for this facility to remain constructive, we allow its use only for proving Σ-formulas.

The natural deduction system for $\mathrm{MQC}_+(S)$ is given in Table 1 with proof term annotations. The diamond in the subscript of $\vdash$ is a wild-card: $\vdash_\diamond$ denotes either $\vdash$ or $\vdash_S$, where in the latter the subscript S is the same formula as the parameter S. We mark $\vdash$ with the parameter to record that a reset has been set. The rules should be read bottom-up, so that the marker is propagated from below to above the line. The usual intuitionistic rules neither “read” nor “write” this marker, hence $\diamond$ denotes the same below and above the line. The reset rule is the one that sets the marker (if it is not already set). If the marker has been already set, then the marker is simply kept. This kind of use of reset would have no logical purpose, but it would affect the course of normalization, hence the computational behavior of the proof term. The rule shift can only be applied when the marker is set, hence it is assured that we are ultimately proving the Σ-formula S.

The following theorem shows a utility of proving with shift and reset.

▶ **Theorem 3.** *Let S be a closed Σ-formula and $A(x)$ an arbitrary formula. The following version of DNS^V,*

$$\Big(\big(\forall x\big(A(x) \vee (A(x) \to S)\big)\big) \to S\Big) \to S, \tag{$\mathrm{DNS}^{\mathrm{V}}_S$}$$

is provable in $\mathrm{MQC}_+(S)$.

Proof. Using the proof term $\lambda h.\#h\Big(\tilde{\lambda}x.\mathcal{S}k.k\Big(\iota_2\big(\lambda a.k(\iota_1 a)\big)\Big)\Big)$. ◀

$\mathrm{DNS}^{\mathrm{V}}_S$ is a version of $\mathrm{DNS}^{\mathrm{V}}$, in which $\bot$ is generalized to a closed Σ-formula S. $\mathrm{DNS}^{\mathrm{V}}_S$ already has some form of MP built in, as can be seen from the proof of Theorem 4 below.

We now state a version of EnDec which is suitable for use in minimal logic, where $\bot$-elimination is absent.

▶ **Axiom 4** (A minimal-logic version of Axiom 1)**.** Assume that $B \subseteq \mathbb{N}$ is enumerable and $n \in \mathbb{N}$. Let, for any $s \in \mathbb{N}$ and any $C \subseteq B$, such that

$$\forall x\,(x \in C \vee (x \in C \to s \in B))\,,$$

Table 1 Natural deduction system for $\mathrm{MQC}_+(S)$, parameterized over a closed Σ-formula S, with proof terms annotating the rules.

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash_\diamond a : A}\ \mathrm{Ax}$$

$$\frac{\Gamma \vdash_\diamond p : A_1 \qquad \Gamma \vdash_\diamond q : A_2}{\Gamma \vdash_\diamond (p, q) : A_1 \wedge A_2}\ \wedge_I \qquad \frac{\Gamma \vdash_\diamond p : A_1 \wedge A_2}{\Gamma \vdash_\diamond \pi_i\, p : A_i}\ \wedge^i_E$$

$$\frac{\Gamma \vdash_\diamond p : A_i}{\Gamma \vdash_\diamond \iota_i\, p : A_1 \vee A_2}\ \vee^i_I$$

$$\frac{\Gamma \vdash_\diamond p : A_1 \vee A_2 \qquad \Gamma, a_1 : A_1 \vdash_\diamond q_1 : C \qquad \Gamma, a_2 : A_2 \vdash_\diamond q_2 : C}{\Gamma \vdash_\diamond \mathsf{case}\ p\ \mathsf{of}\ \ (a_1.q_1 \| a_2.q_2) : C}\ \vee_E$$

$$\frac{\Gamma, a : A_1 \vdash_\diamond p : A_2}{\Gamma \vdash_\diamond \lambda a.p : A_1 \to A_2}\ \to_I \qquad \frac{\Gamma \vdash_\diamond p : A_1 \to A_2 \qquad \Gamma \vdash_\diamond q : A_1}{\Gamma \vdash_\diamond p\, q : A_2}\ \to_E$$

$$\frac{\Gamma \vdash_\diamond p : A(x) \qquad x\ \text{fresh}}{\Gamma \vdash_\diamond \tilde{\lambda} x.p : \forall x A(x)}\ \forall_I \qquad \frac{\Gamma \vdash_\diamond p : \forall x A(x)}{\Gamma \vdash_\diamond p\, t : A(t)}\ \forall_E$$

$$\frac{\Gamma \vdash_\diamond p : A(t)}{\Gamma \vdash_\diamond (t, p) : \exists x.A(x)}\ \exists_I$$

$$\frac{\Gamma \vdash_\diamond p : \exists x.A(x) \qquad \Gamma, a : A(x) \vdash_\diamond q : C \qquad x\ \text{fresh}}{\Gamma \vdash_\diamond \mathsf{dest}\ p\ \mathsf{as}\ (x.a)\ \mathsf{in}\ q : C}\ \exists_E$$

$$\frac{\Gamma \vdash_S p : S}{\Gamma \vdash_\diamond \#p : S}\ \#\ (\text{“reset”}) \qquad \frac{\Gamma, k : A \to S \vdash_S p : S}{\Gamma \vdash_S \mathcal{S}k.p : A}\ \mathcal{S}\ (\text{“shift”})$$

we have that, if

$$\exists m(m \in C \to s \in B),$$

then

$$\exists m((m \in C \to s \in B) \wedge m \in B).$$

Then, $n \in B$.

The following result is the minimal-logic analogue of Theorem 2, showing that an instance of Axiom 4 is derivable in $\mathrm{MQC}_+(S)$.

▶ **Theorem 4.** *Assume that $B \subseteq \mathbb{N}$ is enumerable and $n \in \mathbb{N}$. The instance of Axiom 4 with conclusion $n \in B$ is derivable in the system $\mathrm{MQC}_+(n \in B)$.*

Proof. Let the premises of Axiom 4 hold. To show that $n \in B$, which is a Σ-formula, we use $\mathrm{DNS}^{\mathrm{V}}_S$ for $A(x) := x \in B$ and $S := n \in B$. Now, given $\forall x(x \in B \vee (x \in B \to n \in B))$, we have to show $n \in B$. We use the premise of Axiom 4 for $s := n$ and $C := B$, and, using the trivial proof of $\exists m(m \in B \to n \in B)$ for $m := n$, the premise gives us a proof of $\exists m(m \in B \wedge (m \in B \to n \in B))$, from which we derive $n \in B$. ◀

4 A Proof Term for Open Induction

In this section, we give a proof term for OI on Cantor space in the system $\mathrm{HA}^{\omega}_{+}(S)$ (by suitably instantiating the parameter S), which is the system of axioms HA^{ω} (from §§1.6.15 of [17]) and $\mathrm{AC!}^{0,\mathbb{B}}$ added on top of the predicate logic $\mathrm{MQC}_{+}(S)$ — the need of $\mathrm{AC!}^{0,\mathbb{B}}$ is justified by Remark 2. Basic ingredients to construct the proof term are at hand: Theorem 1 and Theorem 4. We are to interpret them in $\mathrm{HA}^{\omega}_{+}(S)$ and combine the thus obtained proof terms for Theorem 1 and Theorem 4.

4.1 The system $\mathrm{HA}^{\omega}_{+}(S)$

Let S be a closed Σ-formula. First, we take a multi-sorted version of $\mathrm{MQC}_{+}(S)$, that is, given different sorts (denoted by $\sigma, \rho, \tau, \delta$), the language is extended with individual variables (denoted by x, y, z) of any sort, and quantifiers for all sorts. We will not annotate quantifiers with their sorts, since those will be clear from the context; we may annotate variables by their sorts when we want to avoid ambiguity.

The sorts are built inductively, according to the following rules: there is a sort named 0; if ρ and σ are sorts, then there is a sort named $\rho \to \sigma$. The intended interpretation is that the sort 0 stands for $\mathbb{N}$, the sort $0 \to 0$ stands for functions $\mathbb{N} \to \mathbb{N}$, the sort $((0 \to 0) \to 0)$ for functionals $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, etc. We will employ the word 'type' instead of sort, henceforth, and we abbreviate the type $0 \to 0$ by 1.

Now, we add to the language a binary predicate symbol $=$ for individual terms of type 0, intended to be interpreted as (the decidable) equality on $\mathbb{N}$. We emphasize that we only have decidable equality. The individual terms will be built from the function symbols 0^0 (zero), $(\cdot{+}1)^1$ (successor), $\Pi^{\rho\to\tau\to\rho}$ and $\Sigma^{(\delta\to\rho\to\tau)\to(\delta\to\rho)\to\delta\to\tau}$ (combinators), and $\mathrm{R}^{0\to\rho\to(\rho\to0\to\rho)\to\rho}$ (recursor of type ρ). There is also the function symbol of juxtaposition which is not explicitly denoted: for terms $t^{\sigma\to\tau}$ and s^{σ}, $t\,s$ is a term of type τ.

The axioms defining these symbols are (the universal closures of each of):

$$x = x, \qquad x = y \to y = x, \qquad x = y \to y = z \to x = z, \qquad x = y \to x + 1 = y + 1,$$

$$x = y \to t[x/z] = t[y/z] \qquad \text{where } t[x/z] \text{ is the simultaneous substitution of } x \text{ for } z \text{ in } t$$

$$\begin{aligned} t[\Pi xy/u] &= t[x/u] \\ t[\Sigma xyz/u] &= t[xz(yz)/u] \\ t[\mathrm{R}0yz/u] &= t[y/u] \\ t[\mathrm{R}(x+1)yz/u] &= t[z(\mathrm{R}xyz)x/u] \end{aligned}$$

We also add the axiom schema of induction, for arbitrary formula $A(x)$, but only for variables x of type 0:

$$A(0) \to \forall x^0(A(x) \to A(x+1)) \to \forall x^0(A(x)) \tag{IA}$$

Since "=" is the only predicate symbol, all atomic (prime) formulas are of form $t = s$. This allows us to show that $x = y \to A(x) \to A(y)$, by induction on the complexity of formula A.

It is known that using the combinators one may define an individual term for lambda abstraction, denoted $\dot{\lambda}x.t$, of type 1, which satisfies the usual β-reduction axiom,

$$(\dot{\lambda}x^0.s^0)t^0 = s[t/x].$$

Using this and the recursor R, one can easily define all the usual primitive recursive functions. Using the thus defined predecessor function, and the induction axiom, one can derive the remaining Peano axioms, $x + 1 = y + 1 \to x = y$, and $(x + 1 = 0) \to 1 = 0$, where we took $1 = 0$ instead of $\bot$ because we are in minimal logic. In fact, in the presence of arithmetic, one can prove, again by induction, that the rule of $\bot$-elimination (with $\bot$ replaced by $1 = 0$) is derivable, although we will not need it.

Some notational conventions follow. We shall need to speak of bits, finite sequences of bits (bit-strings), and infinite sequences of bits (bit-streams). Bits and bit-strings can be encoded by natural numbers, but, instead of using the type 0 for terms of that kind, to be more pragmatic, we will write bool (intended to interpret $\mathbb{B}$) and bool^* (intended to interpret $\mathbb{B}^*$). Bitstreams are represented by terms of type $0 \to 0$, but we will write $0 \to \mathsf{bool}$ instead. We will need the operations for concatenation and initial segments of both bit-strings and bit-streams, that we already introduced. In addition, the operator $\mathrm{head}(p)$ returns the first bit of p, while $\mathrm{tail}(p)$ returns the string that follows the first bit of p. Although p is not a function, we will use the notation $p(n)$ to extract the $(n + 1)$-th bit of p[10]. We will also use the fact that one can define by primitive recursion a term $\mathsf{if}\ \cdots\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots$ of type $\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$, such that the following equations hold:

$$\begin{aligned} \mathsf{if}\ 0\ \mathsf{then}\ y\ \mathsf{else}\ z &= z \\ \mathsf{if}\ x+1\ \mathsf{then}\ y\ \mathsf{else}\ z &= y \end{aligned}$$

We will also need the usual operation $\min : 0 \to 0 \to 0$ on numbers. All the mentioned operations can be defined by a restricted amount of primitive recursion at higher types, level 3 of the Grzegorcyk hierarchy would suffice. Hence we could work in a corresponding subsystem of HA^ω, like for example $\mathrm{G_3A}^\omega_i$ from §3.5 of [12].

Finally, we shall also need the following choice axiom, a restriction of the usual Axiom of Countable Choice ($\mathrm{AC}^{0,0}$):

$$\forall x^0 \exists! y^{\mathsf{bool}} A(x, y) \to \exists \phi^{0 \to \mathsf{bool}} \forall x^0 A(x, \phi\, x) \qquad (\mathrm{AC!}^{0,\mathbb{B}})$$

Neither $\mathrm{AC}^{0,0}$ nor $\mathrm{AC!}^{0,\mathbb{B}}$ is provable in HA^ω. For arithmetical formulas, $\mathrm{AC}^{0,0}$ (and hence $\mathrm{AC!}^{0,\mathbb{B}}$) is an admissible rule for HA^ω [2].

4.2 Proof term for OI-$\mathbb{B}$

We now formalize the concepts involved in the proof of OI-$\mathbb{B}$. An open set A in Cantor space is given, as a parameter to the logical system, by a term π of type $0 \to \mathsf{bool}^*$, an enumeration of basic opens. Each bit-string $\pi(n)$ is a basic open and the union of them

[10] $\mathrm{head}\, p$ (resp. $p(n)$) returns an arbitrary default value when p is an empty sequence (resp. $\mathrm{len}(p) < n+1$). However, we will use these operations only in a well-defined way.

makes A. Membership in A, $\alpha \in A$, means that α is covered by some basic open from the enumeration. Formally, we define

$$\alpha \in A \text{ iff } \exists l^0 \exists m^0 (\overline{\alpha}\, l = \pi(m)),$$

and we see that membership in A is a closed Σ-formula. (Recall that π is a parameter of the logical system.) The relation $<$ on bit-streams is formalized as

$$\beta < \alpha \text{ iff } \exists n^0 \left(\overline{\beta} n = \overline{\alpha} n \wedge (\beta(n) = 0 \wedge \alpha(n) = 1)\right).$$

We use an instance of Axiom 4 for the enumerable set B given by a Σ-formula $B(x)$, to be defined below, and n given by the natural number encoding an empty sequence. We define

$$B(x) := \exists k^0 \forall q^{\mathsf{bool}^k} \exists l^0 \exists m^0 (\overline{x * q}\, l = \pi(m)),$$

where $\forall q^{\mathsf{bool}^k}$ denotes a *bounded* universal quantification over bit-strings of length k. Bounded quantification can be encoded away using primitive recursive symbols, hence $B(x)$ is still a Σ-formula. We define $p \in B$ by $B(p)$. We have that, for any α, $\exists n(\overline{\alpha} n \in B)$ iff $\alpha \in A$. We instantiate the parameter S of $\mathrm{HA}^\omega_+(S)$ by $\langle\rangle \in B$.

Next, we give an interpretation of the instance of Axiom 4 in $\mathrm{HA}^\omega_+(\langle\rangle \in B)$. We cannot literally formalize Axiom 4 in $\mathrm{HA}^\omega_+(S)$, since $\mathrm{HA}^\omega_+(S)$ does not have higher-order quantification (but only quantification over higher types), hence we cannot quantify over subsets. We therefore "interpret" (the instance of) Axiom 4:

$$\forall s^{\mathsf{bool}^*} \Big(\forall \chi_C^{\mathsf{bool}^* \to \mathsf{bool}} \Big(\forall x^{\mathsf{bool}^*} (\chi_C(x) = 1 \to B(x)) \to \\ \exists q^{\mathsf{bool}^*} (\chi_C(q) = 1 \to B(s)) \to \ \exists r^{\mathsf{bool}^*} ((\chi_C(r) = 1 \to B(s)) \wedge B(r))\Big)\Big) \to B(\langle\rangle).$$

The enumerable set B is represented by the Σ-formula $B(x)$, the decidable subset C by a characteristic function $\chi_C^{\mathsf{bool}^* \to \mathsf{bool}}$, replacing the premise $\forall x\, (x \in C \vee (x \in C \to s \in B))$. The characteristic function should intuitively read as $\chi_C(p) = 1$ iff "$p \in C$", but we take $B(s)$ for $\bot$.

The proof term for OI-$\mathbb{B}$ is shown in Figure 1. We obtained it by formalizing the proofs of Theorems 1 and 4 in $\mathrm{HA}^\omega_+(\langle\rangle \in B)$, and then by normalizing and (hand-)optimizing the formalized proof term, to obtain a compact and direct program proving OI-$\mathbb{B}$.

To ease the presentation, at certain places, we have put after a semicolon the type annotations for individual terms, and the formulas for proof terms. Some parts, being too long, have been put below the main proof term. We suppress the use of equality axioms, to keep the proof term simple without equality-rewriting terms. It is known that equality proofs have no computational content when extracting programs, as they are realized by singleton data types.

We now explain the behavior of the proof term. Given a proof h that A is progressive, it has to show that $\alpha' \in A$ for any α'. As in the proof of Theorem 1, it proves $\langle\rangle \in B$ (lines 3-10), from which we obtain k' such that $h^5 : \forall q^{\mathsf{bool}^{k'}} \exists l^0 \exists m^0 (\overline{q}\, l = \pi(m))$ (line 10). Then $h^5(\overline{\alpha'} k')$ gives us j' such that $h^6 : \exists m^0 (\overline{\overline{\alpha'} k' j'} = \pi(m))$ (line 11), so that $(\min(k', j'), h^6)$ proves $\exists l^0 \exists m^0 (\overline{\alpha'} l = \pi(m))$ (line 12). (An explicit proof of the equality $\overline{\overline{\alpha'} k' j'} = \overline{\alpha'}(\min(k', j'))$ would need an explicit definition of the min function and induction).

To show $\langle\rangle \in B$, which is the parameter of the system, it applies a reset # (line 3), and now it has to show the same formula, but classical logic in the form of the shift rule

$$
\begin{array}{ll}
1: & \lambda h : \forall\alpha(\forall\beta < \alpha(\beta \in A) \to \alpha \in A).\tilde{\lambda}\alpha'. \\
2: & \mathsf{dest} \\
3: & \quad \Big(\#\mathsf{dest}\ a_C(\tilde{\lambda}x.\mathcal{S}k.k(\iota_2(\lambda a.k(\iota_1 a))))\ \mathsf{as}\ (\chi.b)\ \mathsf{in} \\
4: & \qquad \mathsf{dest}\ \Big(h\alpha\big(\tilde{\lambda}\beta.\lambda h' : \beta < \alpha. \\
5: & \qquad\quad \mathsf{dest}\ (h' : \beta < \alpha)\ \mathsf{as}\ (n.h'')\ \mathsf{in} \\
6: & \qquad\quad \mathsf{dest}\ (a_1(\pi_2\pi_2 h'') : \overline{\beta}(n+1) \in B)\ \mathsf{as}\ (k.h''')\ \mathsf{in} \\
7: & \qquad\quad \mathsf{dest}\ (h'''(\langle\beta(n+1)\rangle * \cdots * \langle\beta(n+k)\rangle) : \overline{\beta}(n+k+1) \in A)\ \mathsf{as}\ (j.h^4)\ \mathsf{in} \\
8: & \qquad\quad (\min(n+k+1, j), h^4)\big) : \alpha \in A\Big)\ \mathsf{as}\ (l.c)\ \mathsf{in} \\
9: & \qquad \mathsf{dest}\ (c : \exists m(\overline{\alpha}l = \pi(m)))\ \mathsf{as}\ (m.d)\ \mathsf{in} \\
10: & \qquad a_I\,(\lambda h.h)\,a_3\,l\,(0, \tilde{\lambda}q.(l,(m,d))) : \langle\rangle \in B\Big)\ \mathsf{as}\ (k'.h^5)\ \mathsf{in} \\
11: & \quad \mathsf{dest}\ (h^5\,(\overline{\alpha'}k') : \overline{\alpha'}k' \in A)\ \mathsf{as}\ (j'.h^6)\ \mathsf{in} \\
12: & \quad (\min(k', j'), h^6)
\end{array}
$$

$$
\begin{array}{l}
\alpha := \dot{\lambda}n. \\
\quad \mathrm{R}(n+1, \langle\rangle, (\dot{\lambda}z.\dot{\lambda}n'.z * \langle \mathsf{if}\ \chi(z*\langle 0\rangle)\ \mathsf{then}\ (\mathsf{if}\ \chi(z*\langle 1\rangle)\ \mathsf{then}\ 0\ \mathsf{else}\ 1)\ \mathsf{else}\ 0\rangle))(n) \\
a_1 : \alpha(n) = 1 \to \overline{\beta}(n+1) \in B := \lambda h.\mathsf{case}\ a_B(\chi(\overline{\beta}(n+1)))\ \mathsf{of} \\
\quad \big(h_1.(\pi_1(b(\overline{\beta}(n+1))))\,h_1 \| h_2.(\pi_1(b(\overline{\beta}(n+1))))\,h_2\big) \\
a_3 := \tilde{\lambda}n.\lambda h_I : \overline{\alpha}n \in B \to \langle\rangle \in B.\lambda h : \overline{\alpha}(n+1) \in B. \\
\quad \mathsf{case}\ a_B(\chi(\overline{\alpha}n * \langle 0\rangle))\ \mathsf{of}\ (h_1.(\pi_2(b(\overline{\alpha}(n+1))))\,h_1\,h \\
\qquad \| h_2.\mathsf{case}\ (a_B(\chi(\overline{\alpha}n * \langle 1\rangle)))\ \mathsf{of}\ \ (h_{21}.(\pi_2(b(\overline{\alpha}(n+1))))\,h_{21}\,h \| h_{22}.h_I\,a_4)) \\
a_4 : \overline{\alpha}n \in B := \\
\quad \mathsf{dest}\ ((\pi_1(b(\overline{\alpha}n * \langle 0\rangle)))\,h_2 : \overline{\alpha}n * \langle 0\rangle \in B) \\
\qquad \mathsf{as}\ (k_0.f_0 : \forall q : \mathsf{bool}^{k_0}.\exists l, m(\overline{\overline{\alpha}n * \langle 0\rangle * q}\ l = \pi(m)))\ \mathsf{in} \\
\quad \mathsf{dest}\ ((\pi_1(b(\overline{\alpha}n * \langle 1\rangle)))\,h_{22} : \overline{\alpha}n * \langle 1\rangle \in B) \\
\qquad \mathsf{as}\ (k_1.f_1 : \forall q : \mathsf{bool}^{k_1}.\exists l, m(\overline{\overline{\alpha}n * \langle 1\rangle * q}\ l = \pi(m)))\ \mathsf{in} \\
\quad (\min(k_0, k_1) + 1, \lambda q : \mathsf{bool}^{\min(k_0,k_1)+1}.\mathsf{if}\ \ \mathrm{head}(q)\ \mathsf{then}\ f_1(\overline{\mathrm{tail}(q)}k_1)\ \mathsf{else}\ f_0(\overline{\mathrm{tail}(q)}k_0))
\end{array}
$$

Figure 1 Proof term for OI-$\mathbb{B}$ of type $((\forall\alpha(\forall\beta < \alpha(\beta \in A) \to \alpha \in A)) \to \forall\alpha'(\alpha' \in A))$ in $\mathrm{HA}^{\omega}_{+}(\langle\rangle \in B)$.

can be used. Indeed, the proof term $\tilde{\lambda}x.\mathcal{S}k.k(\iota_2(\lambda a.k(\iota_1 a)))$ proves the "decidability" of B: $\forall x^{\mathsf{bool}^*}(x \in B \vee (x \in B \to \langle\rangle \in B))$. Using the proof term a_C for the formula

$$
\begin{array}{l}
\forall x^{\mathsf{bool}^*}(x \in B \vee (x \in B \to \langle\rangle \in B)) \to \\
\qquad \exists \chi^{\mathsf{bool}^* \to \mathsf{bool}} \forall x^{\mathsf{bool}^*}((\chi(x) = 1 \to x \in B) \wedge (\chi(x) = 0 \to (x \in B \to \langle\rangle \in B))),
\end{array}
$$

we obtain from the decidability, a characteristic function $\chi^{\mathsf{bool}^* \to \mathsf{bool}}$ for B. The proof term a_C is constructed by combining $\mathrm{AC!}^{0,\mathbb{B}}$ together with a proof term that eliminates disjunction in presence of arithmetic[11]. The proof term b proves the characteristic property of χ, namely, $\forall x((\chi(x) = 1 \to x \in B) \wedge (\chi(x) = 0 \to (x \in B \to \langle\rangle \in B)))$.

[11] For the proof of this statement, $(A \vee B) \leftrightarrow \exists x((x = 1 \to A) \wedge (x = 0 \to B))$, see for example §§1.3.7 of [17].

Now, using this χ, the bit-stream α that we saw in the proof of Theorem 1 can be constructed using R and $\mathsf{if}\ \cdots\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots$ by (encoded) course-of-values recursion.

Next one needs to show that $\alpha \in A$ (lines 4-8). One uses progressiveness h: from β and a proof h' of $\beta < \alpha$, one extracts n and a proof h'' of

$$\overline{\beta}n = \overline{\alpha}n \wedge (\beta(n) = 0 \wedge \alpha(n) = 1).$$

Then, $\pi_2\pi_2 h''$ shows $\alpha(n) = 1$, and it is for a_1 to show that $\overline{\alpha}n * \langle 0\rangle = \overline{\beta}(n+1)$ is in B, which in turn shows, with the help of h''', that $\overline{\beta}(n+k+1) \in A$, i.e., $\exists j \exists i(\overline{\beta}(n+k+1)j = \pi(i))$[12]. Now, one concludes $\beta \in A$ with $(\min(n+k+1, j), h^4)$ by appropriately choosing the witness $\min(n+k+1, j)$ so that $\overline{\beta}(n+k+1)j = \overline{\beta}(\min(n+k+1, j))$ holds. (Again, we suppress the proof term for this equality.)

The proof term a_1 derives $\overline{\beta}(n+1) \in B$ from $\alpha(n) = 1$ by making a case distinction. To generate the disjunction needed for the case analysis, one uses a proof term a_B for $\forall x^{\mathsf{bool}}(x = 0 \vee x = 1)$. For the first case in which $\chi(\overline{\beta}(n+1)) = 0$, we have an absurdity $1 = 0$, by definition of α, since $\alpha(n) = 1$. Hence, by equality-rewriting we may use the proof term h_1 at type $\chi(\overline{\beta}(n+1)) = 1$. Now, both the two cases are closed by applying $\pi_1(b(\overline{\beta}(n+1)))$, which proves $\chi(\overline{\beta}(n+1)) = 1 \to \overline{\beta}(n+1) \in B$, to h_1 and h_2, respectively.

From $\alpha \in A$, one obtains the length l and the index m such that $\overline{\alpha}l$ is covered by the basic open $\pi(m)$ (the proof term d in line 9), and then one can show that $\overline{\alpha}0 = \langle\rangle$ is in B. This last fact is derived by the proof term

$$a_I\,(\lambda h.h)\,a_3\,l\,(0, \tilde{\lambda}q.(l, (m, d))),$$

where a_I is a proof term behind an instance of the induction axiom showing $\forall l^0(\overline{\alpha}l \in B \to \langle\rangle \in B)$. The proof term a_I uses the proof term a_3 which derives

$$\forall n((\overline{\alpha}\,n \in B \to \langle\rangle \in B) \to \overline{\alpha}\,(n+1) \in B \to \langle\rangle \in B).$$

It is proved by case analysis, considering the possibilities for the pair $(\chi(\overline{\alpha}n * \langle 0\rangle), \chi(\overline{\alpha}n * \langle 1\rangle))$. If either $\chi(\overline{\alpha}n * \langle 0\rangle) = 0$ or $\chi(\overline{\alpha}n * \langle 1\rangle) = 0$ holds, we close the case by the characteristic property of χ together with the hypothesis h. Otherwise, i.e. both $\chi(\overline{\alpha}n * \langle 0\rangle) = 1$ and $\chi(\overline{\alpha}n * \langle 1\rangle) = 1$ holds, we can deduce $\overline{\alpha}n \in B$ (the proof term a_4), from which the case follows by the induction hypothesis.

5 Conclusion

We gave a direct proof for OI-$\mathbb{B}$ in a constructive predicate logic incorporating delimited control operators. While computational interpretation of $\mathrm{MQC}_+(S)$ is available, namely the standard call-by-value weak-head reduction semantics for lambda calculus with shift and reset, we cannot directly analyze the computational behavior of the proof term for OI-$\mathbb{B}$ because, at the moment, we do not have a proof term for $\mathrm{AC}!^{0,\mathbb{B}}$ used in the proof term for OI-$\mathbb{B}$. The best way to overcome this limitation would be to extend $\mathrm{MQC}_+(S)$ so that it can derive $\mathrm{AC}!^{0,\mathbb{B}}$ as it is done in Martin-Löf Type Theory or constructive versions of Hilbert's epsilon calculus.

Another way to overcome the limitation would be to use a realizability or functional interpretation that extracts programs from constructive proofs even in presence of choice

[12] The proof term $a_1(\pi_2\pi_2 h'')$ proves $\overline{\alpha}n * \langle 0\rangle \in B$, from which $\overline{\beta}(n+1) \in B$ follows using equality axioms. As remarked earlier, equality-rewriting is implicit in the proof term.

axioms. For example, by using Spector's extension of Gödel's functional interpretation with bar recursion, we could extract a program from our proof. However, to replace bar recursion is the point of using delimited control operators in the first place.

If and when our future work is successful, it would allow, at least for the case of the compact Cantor space, to replace Berger's general-recursive computation schema of *open recursion* by a terminating computation schema based on control operators.

The work of Krivine on Classical Realizability gives an interpretation of the Axiom of Dependent Choice [13] using control operators for classical logic. Herbelin recently gave a more direct version of that work [10], using classical control operators and coinduction.

Finally, we would like to mention Veldman's recent work in Constructive Reverse Mathematics [19, 20] that has served as inspiration for our work. An article of Veldman on the equivalence of Open Induction with a number of other axioms is in preparation. In our paper, we showed one direction of this equivalence for the topology of Cantor space seen as the infinite binary tree rather than as the subset of the real line.

Acknowledgments. We would like to thank Wim Veldman for explaining us some of his results, and Ralph Matthes and Hugo Herbelin for valuable comments on the draft.

References

1 Zena M. Ariola and Hugo Herbelin. Minimal classical logic and control operators. In *Thirtieth International Colloquium on Automata, Languages and Programming, ICALP'03, Eindhoven, The Netherlands, June 30 to July 4, 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 871–885. Springer, 2003.

2 Michael Beeson. Goodman's theorem and beyond. *Pacific Journal of Mathematics*, 84:1–16, 1979.

3 Ulrich Berger. A computational interpretation of open induction. In F. Titsworth, editor, *Proceedings of the Nineteenth Annual IEEE Symposium on Logic in Computer Science*, pages 326–334. IEEE Computer Society, 2004.

4 Thierry Coquand. Constructive topology and combinatorics. In J. Myers and M. O'Donnell, editors, *Constructivity in Computer Science*, volume 613 of *Lecture Notes in Computer Science*, pages 159–164. Springer Berlin / Heidelberg, 1992. DOI: 10.1007/BFb0021089.

5 Thierry Coquand. A note on the open induction principle, 1997.

6 Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical report, Computer Science Department, University of Copenhagen, 1989. DIKU Rapport 89/12.

7 Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

8 Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

9 Hugo Herbelin. An intuitionistic logic that proves Markov's principle. In *Proceedings, 25th Annual IEEE Symposium on Logic in Computer Science (LICS'10), Edinburgh, UK, 11–14 July 2010*, page N/A. IEEE Computer Society Press, 2010.

10 Hugo Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, 25–28 June 2012, Dubrovnik, Croatia*, pages 365–374. IEEE Computer Society, 2012.

11 Danko Ilik. Delimited control operators prove double-negation shift. *Annals of Pure and Applied Logic*, 163(11):1549–1559, 2012.

12 Ulrich Kohlenbach. *Applied proof theory: proof interpretations and their use in mathematics*. Springer Monographs in Mathematics. Springer-Verlag, Berlin, 2008.

13 Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theor. Comput. Sci.*, 308(1–3):259–276, 2003.

14 Chetan Murthy. *Extracting Classical Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, 1990.

15 Jean-Claude Raoult. Proving open properties by induction. *Information Processing Letters*, 29:19–23, 1988.

16 Clifford Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics. In *Proc. Sympos. Pure Math., Vol. V*, pages 1–27. American Mathematical Society, Providence, R.I., 1962.

17 Anne S. Troelstra, editor. *Metamathematical Investigations of Intuitionistic Arithmetic and analysis*. Lecture Notes in Mathematics 344. Springer-Verlag, 1973.

18 Wim Veldman. The principle of open induction on the unit interval [0,1] and some of its equivalents. Slides from presentation, May 2010.

19 Wim Veldman. Brouwer's Fan Theorem as an axiom and as a contrast to Kleene's Alternative. *ArXiv e-prints*, June 2011.

20 Wim Veldman. Some further equivalents of Brouwer's Fan Theorem and of Kleene's Alternative. *ArXiv e-prints*, November 2013.

The Montagovian Generative Lexicon ΛTy_n: a Type Theoretical Framework for Natural Language Semantics*

Christian Retoré

LaBRI, Université de Bordeaux / IRIT-CNRS, Toulouse
33405 Talence cedex, France
`christian.retore@labri.fr`

Abstract

We present a framework, named the Montagovian generative lexicon, for computing the semantics of natural language sentences, expressed in many-sorted higher order logic. Word meaning is described by several lambda terms of second order lambda calculus (Girard's system F): the principal lambda term encodes the argument structure, while the other lambda terms implement meaning transfers. The base types include a type for propositions and many types for sorts of a many-sorted logic for expressing restriction of selection. This framework is able to integrate a proper treatment of lexical phenomena into a Montagovian compositional semantics, like the (im)possible arguments of a predicate, and the adaptation of a word meaning to some contexts. Among these adaptations of a word meaning to contexts, ontological inclusions are handled by coercive subtyping, an extension of system F introduced in the present paper. The benefits of this framework for lexical semantics and pragmatics are illustrated on meaning transfers and coercions, on possible and impossible copredication over different senses, on deverbal ambiguities, and on "fictive motion". Next we show that the compositional treatment of determiners, quantifiers, plurals, and other semantic phenomena is richer in our framework. We then conclude with the linguistic, logical and computational perspectives opened by the Montagovian generative lexicon.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.7 Natural Language Processing, I.2.4 Knowledge Representation Formalisms and Methods, D.1.1 Applicative (Functional) Programming

Keywords and phrases type theory, computational linguistics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.202

1 Introduction: word meaning and compositional semantics

The study of natural language semantics and its automated analysis, known as computational semantics, is usually divided into *formal semantics*, usually *compositional*, which has strong connections with logic and with philosophy of language, and *lexical semantics* which rather concerns word meaning and their interrelations, derivational morphology and knowledge representation. Roughly speaking, given an utterance, formal semantics tries to determine *who does what* according to this utterance, while lexical semantics analyses the concepts under discussions and their interplay i.e. *what it speaks about*.

* This work supported by the projects ANR LOCI and POLYMNIE, and was done during my CNRS-sabbatical at IRIT.

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 202–229

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

(1) a. *A sentence:* Some club defeated Leeds.

b. *Its formal semantics:* $\exists x : \mathtt{e}\ (\mathtt{club}(x)\ \&\ \mathtt{defeated}(x, \mathtt{Leeds}))$

(2) *Lexical semantics as found in a dictionary:* `defeat:`

a. overcome in a contest, election, battle, etc.; prevail over; vanquish

b. to frustrate; thwart.

c. to eliminate or deprive of something expected

Although applications in computational linguistics require both aspects of semantics, some applications rather focus on formal and compositional semantics, e.g. man machine dialogue, non statistical translation, text generation while other applications like information retrieval, classification, statistical translation rather stress lexical semantics.

Herein we refine compositional semantics with a treatment of some of lexical semantics issues, in particular for selecting the right word meaning in a given context. Of course any sensible analyser, including human beings, or Moot's Grail parser [49] combines both the predicate argument structures and the relations between lexical meanings to build a semantic representation and to understand an utterance.

We define a framework named *the Montagovian generative lexicon*, written $\Lambda\mathsf{Ty}_n$as it extends Ty_n of Muskens [55] with the second order Λ operator and the corresponding quantified Π types. It is based on Montague view of formal and compositional semantics [46], but we provide a faithful and computable account of some phenomena of lexical semantics, which have been addressed in particular by Pustejovsky and Asher [62, 5, 6]: correctness, polysemy, adaptation of word meaning to the context, copredication over different senses of a given expression. Our framework $\Lambda\mathsf{Ty}_n$ also suggests a finer grained analysis of some formal and compositional semantic issues such as determiners, quantification, or plurals.

Compositional semantics is usually described within simply typed lambda calculus: therefore its implementation is rather straightforward in any typed functional programming language like ML, CaML or Haskell. The computational framework for natural language semantics that we present in this paper, as well as the precise description of some semantics constructions, is defined in a subsystem of system F (second order lambda calculus), which does not go beyond the type systems of the afore mentioned functional programming languages. Hence our proposals can easily be implemented in such a language, for instance in Haskell along the lines of the good and recent book by van Eijck and Unger on *computational semantics with functional programming* [73].

1.1 The syntax of compositional semantics

As opposed to many contributions to the domain of linguistics known as "formal semantics" the present paper neither deals with reference nor with truth in a given situation: we only build a logical formula (first order or higher order, single sorted or many-sorted) that can be thereafter interpreted as one wants, if he wishes to. Hence we are not committed to any particular kind of interpretation like truth values, possible worlds, game semantics,...

In the traditional Montagovian view, the process of semantic interpretation of a sentence, consists in computing from syntax and word meanings, a logical formula (which possibly includes logical modalities and intensional operators), and in interpreting this formula in possible world semantics. Although Montague thought that intermediate representations, including the logical formulae, should be regarded as unimportant, and should be wiped off just after computing truth values and references, in this paper we precisely focus on the intermediate representations, in particular on the logical formulae, which can be called the

logical forms of sentences, with particular attention to the way they are computed – for the time being, we leave out the interpretation of these formulae. A reason for doing so is that we can encompass subtle questions, like vague predicates, generalised and vague quantifiers, for which standard notions of truth and references are inadequate: possibly some interactive interpretation would be better suited, as that proposed by Lecomte and Quatrini [33] or by Abrusci and Retoré [1].

1.2 A brief reminder on Montague semantics

Let us briefly remind the reader how one computes the logical forms according to the Montagovian view. Assume for simplicity that a syntactic analysis is a tree specifying how subtrees apply one to the other – the one that is applied is called the function while the other is called its argument. A semantic lexicon provides a simply typed λ-term $[w]$ for each word w. The semantics of a leaf (hence a word) w is $[w]$ and the semantic $[t]$ of a sub syntactic tree $t = (t_1, t_2)$ is recursively defined as $[t] = ([t_1]\ [t_2])$ that is $[t_1]$ *applied to* $[t_2]$, in case $[t_1]$ is the function and $[t_2]$ the argument – and as $[t] = ([t_2]\ [t_1])$ otherwise, i.e. when $[t_2]$ is the function and $[t_1]$ the argument. In addition to these functional applications, the tree could possibly include some λ-expressions, for instance if the syntactic structure is computed with a categorial grammar that includes hypothetical reasoning like Lambek calculus and its extensions, see e.g. [53, Chapter 3].

The typed λ-terms from the lexicon are given in such a way that the function always has a semantic type of the shape $\mathsf{a} \to \mathsf{b}$ that matches the type a of the argument, and the semantics associated with the whole tree has the semantic type t, that is the type of propositions. This correspondence between syntactical categories and semantic types, which extends to a correspondence between parse structures and logical forms is crystal clear in categorial grammars, see e.g. [53, Chapter 3]. Typed λ-terms usually are defined out of two base types, e for individuals (also known as **e**ntities) and t for propositions (which have a **t**ruth value). Logical formulae can be defined in this typed λ-calculus as first observed by Church a long time ago. This early use of lambda calculus, where formulae are viewed as typed lambda terms, cannot be merged with the more familiar view of typed lambda terms as proofs. The proof such a typed lambda term corresponds to is simply the proof that the formula is well formed, e.g. that a two-place predicate is properly applied to *two* individual terms of type e and not to more or fewer objects, nor to objects of a different type etc. This initial vision of lambda calculus was designed for a proper handling of substitution in deductive systems à la Hilbert. One needs constants for the logical quantifiers and connectives:

Quantifier	Constant	Type
there exists	$\exists$	$(\mathsf{e} \to \mathsf{t}) \to \mathsf{t}$
for all	$\forall$	$(\mathsf{e} \to \mathsf{t}) \to \mathsf{t}$

Connective	Constant	Type
and	$\&$	$\mathsf{t} \to \mathsf{t} \to \mathsf{t}$
or	$\vee$	$\mathsf{t} \to \mathsf{t} \to \mathsf{t}$
implies	$\supset$	$\mathsf{t} \to \mathsf{t} \to \mathsf{t}$

Constant	Type
`defeated`	$\mathsf{e} \to \mathsf{e} \to \mathsf{t}$
`won, voted`	$\mathsf{e} \to \mathsf{t}$
`Liverpool, Leeds`	e
...	...

as well as predicates for the precise language to be described – a binary predicate like `won` has the type $\mathsf{e} \to \mathsf{e} \to \mathsf{t}$ – as usual the type $\mathsf{a} \to \mathsf{b} \to \mathsf{c} \to \mathsf{u}$ stands for $\mathsf{a} \to (\mathsf{b} \to (\mathsf{c} \to \mathsf{u}))$ and the term $h\ t\ s\ r$ stands for $(((h\ t)\ s)\ r)$ (h being a function of arity at least 3).

word	***semantic type*** u^* ***semantics : λ-term of type*** u^* x^v *the variable or constant* x *is of type* v
some	$(\mathsf{e} \to \mathsf{t}) \to (\mathsf{e} \to \mathsf{t}) \to \mathsf{t}$ $\lambda P^{\mathsf{e}\to\mathsf{t}}.\ \lambda Q^{\mathsf{e}\to\mathsf{t}}.\ (\exists^{(\mathsf{e}\to\mathsf{t})\to t}\ (\lambda x^{\mathsf{e}}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}(P\ x)(Q\ x))))$
club	$\mathsf{e} \to \mathsf{t}$ $\lambda x^{\mathsf{e}}.\ (\mathtt{club}^{\mathsf{e}\to\mathsf{t}}\ x)$
defeated	$\mathsf{e} \to \mathsf{e} \to \mathsf{t}$ $\lambda y^{\mathsf{e}}.\ \lambda x^{\mathsf{e}}.\ ((\mathtt{defeated}^{\mathsf{e}\to\mathsf{e}\to\mathsf{t}}\ x)y)$
Leeds	e $\mathtt{Leeds}$

Figure 1 A simple semantic lexicon.

A small example goes as follows. Assume the syntax says that the structure of the sentence "*Some club defeated Leeds.*" is

(some (club)) (defeated Leeds)

where the function is always the term on the left. If the semantic terms are as in the lexicon in Figure 1, placing the semantical terms in place of the words yields a large λ-term that can be reduced:

$$\Big(\big(\lambda P^{\mathsf{e}\to\mathsf{t}}.\ \lambda Q^{\mathsf{e}\to\mathsf{t}}.\ (\exists^{(\mathsf{e}\to\mathsf{t})\to\mathsf{t}}\ (\lambda x^{\mathsf{e}}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}(P\ x)(Q\ x)))))\big(\lambda x^{\mathsf{e}}.\ (\mathtt{club}^{\mathsf{e}\to\mathsf{t}}\ x)\big)\Big)$$
$$\Big(\big(\lambda y^{\mathsf{e}}.\ \lambda x^{\mathsf{e}}.\ ((\mathtt{defeated}^{\mathsf{e}\to\mathsf{e}\to\mathsf{t}}\ x)y)\big)\ \mathtt{Leeds}^{\mathsf{e}}\Big)$$
$$\downarrow \beta$$
$$\big(\lambda Q^{\mathsf{e}\to\mathsf{t}}.\ (\exists^{(\mathsf{e}\to t)\to\mathsf{t}}\ (\lambda x^{\mathsf{e}}(\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}(\mathtt{club}^{\mathsf{e}\to\mathsf{t}}\ x)(Q\ x)))))$$
$$\big(\lambda x^{\mathsf{e}}.\ ((\mathtt{defeated}^{\mathsf{e}\to\mathsf{e}\to\mathsf{t}}\ x)\mathtt{Leeds}^{\mathsf{e}})\big)$$
$$\downarrow \beta$$
$$\big(\exists^{(\mathsf{e}\to\mathsf{t})\to\mathsf{t}}\ (\lambda x^{\mathsf{e}}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}(\mathtt{club}^{\mathsf{e}\to\mathsf{t}}\ x)((\mathtt{defeated}^{\mathsf{e}\to\mathsf{e}\to\mathsf{t}}\ x)\mathtt{Leeds}^{\mathsf{e}}))))\big)$$

This λ-term of type t can be called the *logical form* of the sentence. It represents the following formula of predicate calculus (admittedly more pleasant to read):

$$\exists x : \mathsf{e}\ (\mathtt{club}(x)\ \&\ \mathtt{defeated}(x, \mathtt{Leeds}))$$

The above described procedure is quite general: starting with a properly defined semantic lexicon whose terms only contain the logical constants and the predicates of the given language one always obtains a logical formula. Indeed, such λ-terms always reduce to a unique normal form and any normal λ-term of type t (preferably η long, see e.g. [53, Chapter 3]) corresponds to a logical formula.

If we closely look at the Montagovian setting described above, we observe that it is weaving two different "logics":

Logic/calculus for meaning assembly (a.k.a glue logic, metalogic,...) In our example, this is simply typed λ-calculus with two base types e and t – these terms are the proof in intuitionistic propositional logic.

Logic/language for semantic representations In our example, that is higher-order predicate logic.[1]

[1] It can be first-order logic if reification is used, but this may induce unnatural structure and exclude some readings.

The framework we present in this paper mainly concerns the extension of the metalogic and the reorganisation of the lexicon in order to incorporate some phenomena of lexical semantics, first of all restrictions of selection. Indeed, in the standard type system above nothing prevents a mismatch between the real nature of the argument and its expected nature. Consider the following sentences:[2]

(3) a. * A chair barks.
b. * Jim ate a departure
c. ? The five is fast

Although they can be syntactically analysed, they should not receive a semantical analysis. Indeed, "*barks*" requires a "*dog*" or at least an "*animate*" subject while a "*chair*" is neither of them; "*departure*" is an event, which cannot be an "*inanimate*" object that could be eaten; finally a "*number*" like "*five*" cannot do anything fast – but there are particular contexts in which such an utterance makes sense and we shall also handle these meaning transfers.

1.3 The need of integrating lexical semantics in formal semantics

In order to block the interpretation of the semantically ill formed sentences above, it is quite natural to use *types*, where the word *type* should be understood both in its intuitive and in its formal meaning. The type of the subject of barks should be "*dog*", the type of "*fast*" objects should be "*animate*", and the type of the object of "*ate*" should be "*inanimate*". Clearly, having, on the formal side a unique type `e` for all entities is not sufficient.

The traditional view with a single type `e` for entities has another related drawback. It is unable to relate predicates whose meanings are actually related, although a usual dictionary does. A common noun like "*book*" is usually viewed as a unary predicate "$\texttt{book:e} \to \texttt{t}$" while a transitive verb like "*read*" is viewed as a binary predicate "$\texttt{read: e} \to \texttt{e} \to \texttt{t}$" This gives the proper argument structure of *Mary reads a book.* as $(\exists x : \texttt{ebook}(x)\ \&\ \texttt{reads}(Mary, x))$ but this traditional setting cannot relate the predicates `book` and `read` – while any dictionary does. With several types, as we shall have later on, we could stipulate that the object of "*read*" ought to be something that one can "*read*", and a "*book*" can be declared as something that one can "*read*", "*write*", "*print*", "*bind*", etc. Connections between a predicate like "*book*" and predicates like "*write*", "*read*", etc. allow to interpret sentences like "*I finished my book*" which usually means "*I finished to read my book*" and sometimes "*I finished to write my book*", the other possible senses being even rarer.

Hence we need a more sophisticated type theory than the one initially used by Montague to filter semantically invalid sentences. But in many cases some flexibility is needed to accept and analyse sentences in which a word type is coerced into another type. In sentence (3c), in the context of a football match, the noun "*five*" can be considered as a player i.e. a "*person*" who plays the match with the number 5 jersey, who can "*run*".

There is a vast literature on such lexical meaning transfers and coercions, starting from 1980 [11, 12, 21, 57] – see also [32, 13] for more recent surveys of some lexical theories. In those pioneering studies, the objective is mainly to classify these phenomena, to find the rules that govern them. The quest of a computational formalisation that can be incorporated into an automated semantic analyser appears with Pustejovsky's generative lexicon in 1991

[2] We use the standard linguistic notation: a "*" in front of a sentence indicates that the sentence is incorrect, a "?" indicates that the correctness can be discussed and the absence of any symbol in front means that the sentence is correct.

[61, 62]. The integration of lexical issues into compositional semantics à la Montague and type theories appears with the work by Nicholas Asher [5, 6] which led to the book [3], and differently in some works of Robin Cooper with an intensive use of records from type theory to recover frame semantics with features and attributes inside type-theoretical compositional semantics [19, 20]

1.4 Type theories for integrating lexical semantics

As the aforementioned contributions suggest, richer type systems are quite a natural framework for formal semantics à la Montague and for selectional restriction and coercions. Such a model must extend the usual ones into two directions:

1. Montague's original type system and metalogic should be enriched to encompass lexical issues (selectional restriction and coercions), and
2. the usual phenomena studied by formal semantics (quantifiers, plurals, generics) should be extended to this richer type system and so far only Cooper and us did so [19, 20, 16, 52, 41, 35, 64]

At the end of this paper, we shall provide a comparison of the current approaches, which mainly focuses on (1). Let us list right now what the current approaches are:

- The system works with type based coercions and relies on some *Modern Type Theory (MTT)* [3] – this corresponds to the work of Zhaohui Luo [38, 39, 77, 16].
- The system works with type based coercions and relies on usual typed λ-calculus extended with some categorical logic rules – this approach by Asher [5, 6] culminated in his book [3]
- The system works with term based coercions and relies on second order λ-calculus – this is our approach, first introduced with Bassac, Mery, and further developed with Mery, Moot, Prévot, Real-Coelho. [8, 51, 50, 52, 41, 34, 35, 64, 63].

In fact our approach differs from the concurrent ones mainly because of the organisation of the lexicon and of the respective rôles of types and terms. It can be said to be word-driven, as it accounts for the (numerous) idiosyncrasies of natural language in particular the different behaviour of words of the same type is coded by assigning them different terms, while others derive everything from the types.

The precise type system we use, namely system F, could be replaced by some other type theory. However, as far as the presentation of the system is concerned, it is the *simplest* of all systems, because it only contains four term building operations (two of them being the standard λ-calculus rules, the two others being their second order counterparts) and two reduction rules (one of them being the usual beta reduction and the other one being its second order counterpart). Dependent types, which are types defined from terms, are a priori not included although they could be added if necessary.

[3] This name *Modern Type Theory (MTT)* covers several variants of modern type theories, including Martin-Löf type theory, the Predicative Calculus of (Co)Inductive Constructions (pCic), the Unifying Theory of dependent Types (UTT), ... – the latter one being the closest to the system used by Zhaohui Luo

2 A Montagovian generative lexicon for compositional semantic and lexical pragmatics

We now present our solution for introducing some lexical issues in a compositional framework à la Montague.

2.1 Guidelines for a semantic lexicon

We should keep in mind that whatever the precise solution presented, the following questions must be addressed in order to obtain a computational model, so here are the guidelines of our model:

- What is the logic for semantic representation?
  We use many-sorted *higher order predicate calculus. As usual, the higher order can be reified in first order logic, so it can be first order logic, but in any case the logic has to be many-sorted. Asher [3] is quite similar on this point, while Luo use Type Theory [39].*
- What are the sorts?
  The sorts are the base types. As discussed later on these sorts may vary from a small set of ontological kinds to any formula of one variable. We recently proposed that they correspond to classifiers *in language with classifiers: this give sorts a linguistically and cognitively motivated basis [43].*
- What is the metalogic (glue logic) for meaning assembly?
  We use second order λ-calculus (Girard system F) in order to factor operations that apply uniformly to a family of types. For specific coercions, like ontological inclusions we use subtyping introduced in the present paper. Asher [3] uses simply typed λ-terms with additional categorical rules, while Luo also use Type Theory with coercive subtyping [39].
- What kind of information is associated with a word in the lexicon?
  Here it will be a finite set of λ-terms, one of them being called the principal λ-term while the other ones are said to be optional. Other approaches make use of more specific terms and rules.
- How does one compose words and constituents for a compositional semantics?
  We simply apply one λ-term to the other, following the syntactic analysis, perform some transformations corresponding to coercions and presupposition, and reduce the compound by β-reduction.
- How is the semantic incompatibility of two components rendered?
  By type mismatch, between a function of type $A \to X$ and an argument of type $B \neq A$. Most works that insert lexical considerations into compositional semantics model incompatibility by type mismatch.
- How does one allow an *a priori* impossible composition?
  By using the optional λ-terms, which change the type of at least one of the two terms being composed, the function and argument. Both the function and the argument may provide some optional lambda terms. Other approaches rather use type-driven rules.
- How does one allow or block felicitous and infelicitous copredications on various aspects of the same word?
  An aspect can be explicitly declared as incompatible with any other aspect. More recently we saw that linear types (linear system F) can account for compatibility between arbitrary subsets of the possible aspects. [42]

Each word in the lexicon is given a principal term, as well as a finite number, possibly nought, of optional terms that licence type change and implement coercions. They may be

inferred from an ordinary dictionary, electronic or not. Terms combine almost as usual except that there might be type clashes, which account for infringements of selectional restriction: in this case optional terms may be used to solve the type mismatch. In case they lead to different results these results should be considered as different possible readings – just as the different readings with different quantifier scopes are considered by formal semantics as different possible readings of a sentence.

Let us first present the type and terms and thereafter we shall come back to the composition modes.

2.2 Remarks on the type system for semantics

We use a type system that resembles Muskens Ty_n [55] where the usual type of individuals, e is replaced with a finite but large set of base types $\mathsf{e}_1, \ldots, \mathsf{e}_n$ for individuals, for instance *objects*, *concepts*, *events*,... These base types are the sorts of the many-sorted logic whose formulae express semantic representations. The set of base types as well as their interrelations can express some ontological relations as Ben Avi and Francez thought ten years ago [10].

For instance, assume we have a many-sorted logic with a sort ζ for animals, a sort ϕ for physical objects and a predicate *eat* whose arguments are of respective sort ϕ and ζ the many-sorted formula $\forall z : \zeta\ \exists x : \phi\ eat(z, x)$ is rendered in type theory by the λ-term: $\forall^{\zeta}(\lambda z^{\zeta}.\ (\exists^{\phi}(\lambda x^{\phi}.\ ((eat\ x)z))))$ with *eat* a constant of type $\phi \to \zeta \to \mathsf{t}$. Observe that the type theoretic formulation requires a quantifier for each sort α of objects, that is a constant $\forall^{\alpha}$ of type $(\alpha \to \mathsf{t}) \to \mathsf{t}$.[4]

What are the base types? We have a tentative answer, but we cannot be too sure of this answer. Indeed, this is a subtle question depending on one's philosophical convictions, and also on the expected precision of the semantic representations,[5] but it does not really interfere with the formal and computational model we present here. Let us mention some natural sets of base types that have been proposed so far, from the smallest to the largest:

1. *A single base type* e *for all entities* (but as seen above it cannot account for lexical semantics).
2. *A very simple ontology* defines the base types: events, physical objects, living entities, concepts, ... (this resembles Asher's position in [3]).
3. *Classifiers.* Many Asian languages (Chinese, Japanese, Korean, Malay, Burmese, Nepali, ...) and all Sign Languages, have *classifiers* that are pronouns specific to classes of nouns (100–400) especially detailed for physical objects that can be handled, and for animals.There are almost no classifiers in European languages. Nevertheless a word like "*head*" in "*Three heads of cattle.*" can be considered as a classifier. Hence classifiers are a rather natural set of base types, or the importation of the classifiers of a language in one that does not have any [43]. But we do not claim that this is the definitive answer. For instance, for a specific task, some other set of base types may be better.
4. *A base type per common noun* (thousands of base types) as proposed by Luo in [39]).
5. *A type for every formula* with a single free variable.

Our opinion is that types should be cognitively natural classes and rich enough to express selectional restrictions. Whatever types are, there is a relation between types and properties.

[4] We do not speak about interpretations, but if one wishes to, we do not necessarily ask for the usual requirement that sorts are disjoint: this is coherent with the fact that in type theory, nothing prevents a pure term from having several types.

[5] For instance, a dictionary says that pregnant can be said of a "*woman or female animal*", but can it be said of a "*grandma*" or of a "*heifer*"?

With base types as in (5), the correspondence seems quite clear, but, because types can be used to express new many-sorted formulae, the set of types is in this case defined as a least fixed point. For other sets of base types, e.g. (4) or (2) for each type T there should be a corresponding predicate which recognises T entities among entities of a larger type. For instance, if there is a type *dog* there should be a predicate $\widehat{dog} : \alpha \to \mathsf{t}$ but what should be the type α of its argument? Should it be "*animal*", "*animate*",... the simplest solution is to assume a type of all individuals, that is Montague's e, and to say that corresponding to any base type a, there is a predicate, namely $\widehat{\mathsf{a}}$ of type $\mathsf{e} \to \mathsf{t}$.[6]

Let us make here a remark on the predicate constants in the language. If a predicate constant, say Q, has the type $\mathsf{u} \to \mathsf{t}$ with $\mathsf{u} \subsetneq \mathsf{e}$ – sometimes another type u is more natural than the standard e – then there is a canonical extension Q_{e} of Q to e which should be interpreted as false for any object that cannot be viewed as a u-object. Predicates constants from the first or higher order logical language do also have restrictions. Given a type u that is smaller than the domain q of Q one can define $Q|_{\mathsf{u}}$ which is defined as Q on $\mathsf{q} \cap \mathsf{u}$ and as false elsewhere.

2.3 $\Lambda\mathsf{Ty}_n$: many-sorted formulae in second order lambda calculus

Since we have many base types, and many compound types as well, it is quite convenient and almost necessary to define operations over family of similar terms with different types, to have some flexibility in the typing, and to have terms that act upon families of terms and types. Hence we shall extend further Ty_n into $\Lambda\mathsf{Ty}_n$ by using Girard's system F as the type system [25, 24]. System F involves quantified types whose terms can be specialised to any type.

The types of $\Lambda\mathsf{Ty}_n$are defined as follows:

- Constant types e_i and t are (base) types.
- Type variables $\alpha, \beta, \ldots$ are types.
- Whenever T and α respectively are a type and a type-variable, the expression $\Pi\alpha.\ T$ is a type. The type variable may or may not occur in the type T.
- Whenever T_1 and T_2 are types, $T_1 \to T_2$ is a type as well.

The terms of $\Lambda\mathsf{Ty}_n$, which encode proofs of quantified propositional intuitionistic logic, are defined as follows:

- A variable of type T i.e. $x : T$ or x^T is a *term*, and there are countably many variables of each type.
- In each type, there can be a countable set of constants of this type, and a constant of type T is a term of type T. Such constants are needed for logical operations and for the logical language (predicates, individuals, etc.).
- $(f\ t)$ is a term of type U whenever $t : T$ and $f : T \to U$.
- $\lambda x^T.\ t$ is a term of type $T \to U$ whenever $x : T$ and $t : U$.
- $t\{U\}$ is a term of type $T[U/\alpha]$ whenever $t : \Lambda\alpha.\ T$ and U is a type.
- $\Lambda\alpha.\ t$ is a term of type $\Pi\alpha.\ T$ whenever α is a type variable and $t : T$ is a term without any free occurrence of the type variable α in the type of a free variable of t.

The later restriction is the usual one on the proof rule for quantification in propositional logic: one should not conclude that $F[p]$ holds for any proposition p when assuming that a property $G[p]$ of p holds – i.e. when having a free hypothesis of type $G[p]$.

[6] An alternative solution, used by us and others [64, 17] would be $\Pi\alpha.\ \alpha \to \mathsf{t}$, using quantification over types to be defined in the next section.

The reduction of the terms in system F or its specialised version $\Lambda\mathsf{Ty}_n$is defined by the two following reduction schemes that resemble each other:

- $(\lambda x^{\phi}.\ t)u^{\phi}$ reduces to $t[u/x]$ (usual β reduction).
- $(\Lambda\alpha.\ t)\{U\}$ reduces to $t[U/\alpha]$ (remember that α and U are types).

As an example, we earlier said that in Ty_n we needed a first order quantifier per sort (i.e. per base type). In $\Lambda\mathsf{Ty}_n$it is sufficient to have a single quantifier $\forall$, that is a constant of type $\Pi\alpha.\ (\alpha \to \mathsf{t}) \to \mathsf{t}$. Indeed, this quantifier can be specialised to specific types, for instance to the base type ζ, yielding $\forall\{\zeta\} : (\zeta \to \mathsf{t}) \to \mathsf{t}$, or even to properties of ζ objects, which are of type $\zeta \to \mathsf{t}$, yielding $\forall\{\zeta \to \mathsf{t}\} : ((\zeta \to \mathsf{t}) \to \mathsf{t}) \to \mathsf{t}$. We actually do quantify over higher types, for instance in the examples below we respectively quantify over propositions with a human subject (Example 4), and over all propositions (Example 5):

(4) He did everything he could to stop them.

(5) And he believes whatever is politically correct and sounds good.

As Girard showed [25, 24] reduction is strongly normalising and confluent *every term of every type admits a unique normal form which is reached no matter how one proceeds.*[7] The normal forms, which can be asked to be η-long without loss of generality, can be characterised as follows (for a reference see e.g. [28]):

▶ Proposition 1. A normal Λ-term $\mathcal{N}$ of system F, β normal and η long to be precise, has the following structure:[8]

$$\mathcal{N} = \overbrace{(\ \lambda {x_i}^{X_i} \mid \Lambda X_j\ )^*}^{\text{sequence of } \lambda \text{ and } \Lambda \text{ abstractions}} \overbrace{(\cdots(\ h^{(\Pi X_k \mid X_l \to)^* Z}}^{\text{head variable}} \overbrace{(\{W_k\} \mid {t_l}^{X_l})^*\ )\cdots)}^{\text{sequence of } \{\cdots\} \text{ and } (\cdots) \text{ applications to types } W_k \text{ and normal terms } {t_l}^{X_l}}$$

This has a good consequence for computational semantics, see e.g. [53, Chapter 3]:

▶ Property 1 ($\Lambda\mathsf{Ty}_n$terms as formulae of a many-sorted logic). If the predicates, the constants and the logical connectives and quantifiers are the ones from a many-sorted logic of order n (possibly $n = \omega$) then the normal terms of $\Lambda\mathsf{Ty}_n$ of type t unambiguously correspond to many-sorted formulae of order n.

Let us illustrate how F factors uniform behaviours. Given types α, β, two predicates $P^{\alpha\to\mathsf{t}}$, $Q^{\beta\to\mathsf{t}}$, over entities of respective kinds α and β for any ξ with two morphisms from ξ to α and to β (see Figure 2), F contains a term that can coordinate the properties P, Q of (the two images of) an entity of type ξ, every time we are in a situation to do so – i.e. when the lexicon provides the morphisms.

▶ Term 1. [Polymorphic AND] is defined as

$$\&^{\Pi} = \Lambda\alpha.\ \Lambda\beta.\ \lambda P^{\alpha\to\mathsf{t}}.\ \lambda Q^{\beta\to\mathsf{t}}.\ \Lambda\xi.\ \lambda x^{\xi}.\ \lambda f^{\xi\to\alpha}.\ \lambda g^{\xi\to\beta}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}\ (P\ (f\ x))(Q\ (g\ x)))\,.$$

Such a term is straightforwardly implemented in Haskell along the lines of [73]:

[7] This is one way to be convinced of the soundness of F, which defines types depending on other types including themselves: as it is easily observed that there are no normal closed terms of type $\Pi X.\ X \equiv \bot$, the system is necessarily coherent. Another way is to construct a concrete model, for instance coherence spaces, where types are interpreted as countable sets with a binary relation (coherence spaces), and terms up to normalisation are interpreted as structure preserving functions (stable functions) [25].

[8] This structure resembles the structure of (weak) head normal form, in functional programming, but the terms inside the structure are also asked to be normal.

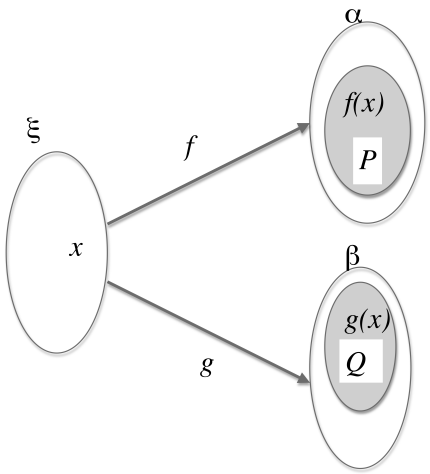


Figure 2 Polymorphic conjunction: $P(f(x))\&Q(g(x))$ with $x : \xi$, $f : \xi \to \alpha$, $g : \xi \to \beta$.

```
andPOLY :: (a -> Bool) -> (b -> Bool) -> c -> (c -> a) -> (c -> b) -> Bool
andPOLY = \ p q x f g -> p (f x) && q (g x)
```

This can apply to say, a "*book*", that can be "*heavy*" as a "*physical object*", and "*interesting*" as an "*informational content*" – the limitation of possible over-generation, that is the production or recognition of incorrect phrases or sentences, is handled by the *rigid* use of possible transformations, to be defined thereafter.

2.4 Organisation of the lexicon and rules for meaning assembly

The lexicon associates each word w with a *principal* λ*-term* $[w]$ which basically is the Montague term reminded earlier, except that the types appearing in $[w]$ belong to a much richer typed system. In particular, the numerous base types can impose some selectional restriction. In addition to this principal term, there can be *optional* λ*-terms* also called *modifiers* or *transformations* to allow, in some cases, compositions that were initially ruled out by selectional restriction.

There are two ways to solve a type conflict using those modifiers. *Flexible modifiers* can be used without any restriction. *Rigid modifiers* turn the type, or the sense of a word, into another one which is *incompatible* with other types or senses. For a technical reason, the identity, which is always a licit modifier, is also specified to be flexible or rigid. In the lalter rigid case, it means that the original sense is incompatible with any other sense, although two other senses may be compatible. Consequently, every modifier, i.e. optional λ-term is declared, in the lexicon, to be either a rigid modifier, noted (R) or a flexible one, noted (F). More subtle compatibility relations between senses can be represented by using the linear version of system F as we did in [42].

The reader may be surprised that we repeat the morphisms in the lexical entries, rather than having general rules. For instance, one could also consider morphisms that are not anchored in a particular entry: in particular, they could implement the ontology at work in [62] as the type-driven approach of Asher does [3]. For instance, a place (type Pl) could be viewed as a physical object (type ϕ) with a general morphism $Pl2\phi$ turning places into physical objects that can be "*spread out*". We are not fully enthusiastic about a general use of

word	principal λ-term	optional λ-terms	rigid/flexible
book	$\widehat{B} : \mathtt{e} \to \mathtt{t}$	$Id_B : B \to B$	(F)
		$b_1 : B \to \phi$	(F)
		$b_2 : B \to I$	(F)
town	$\widehat{T} : \mathtt{e} \to \mathtt{t}$	$Id_T : T \to T$	(F)
		$t_1 : T \to F$	(R)
		$t_2 : T \to P$	(F)
		$t_3 : T \to Pl$	(F)
Liverpool	$\mathtt{Lpl}^T$	$Id_T : T \to T$	(F)
		$t_1 : T \to F$	(R)
		$t_2 : T \to P$	(F)
		$t_3 : T \to Pl$	(F)
spreadout	$\mathtt{spread\_out} : Pl \to \mathtt{t}$		
voted	$\mathtt{voted} : P \to \mathtt{t}$		
won	$\mathtt{won} : F \to \mathtt{t}$		

where the base types are defined as follows:

B	book	T	town
ϕ	physical objects	Pl	place
I	information	P	people
		F	football team

Figure 3 A sample lexicon.

such rules since it is hard to tell whether they are flexible or rigid. As they can be composed they might lead to incorrect copredications, while their repetition inside each entry offers a better control of incorrect and correct copredications. One can think that some meaning transfer differs although the words have the same type. An example of such a situation in French is provided by the words "*classe*" and "*promotion*", which both refer to groups of pupils. The first word "*classe*" (English: "*class*") can be coerced into the room where the pupils are taught, (the "*classroom*"), while the second, "*promotion*" (English: "*class*" or "*promotion*") cannot.

Consequently, we in general prefer word-driven coercions, i.e. modifiers that are anchored in a word. An exception are ontological inclusions that are better represented by type-driven rules: "*cars*" are "*vehicles*" which are "*artefacts*" etc. This is the reason why we also allow optional terms that are available for all words of the same type. This is done by *subtyping* and more precisely by the notion of coercive subtyping that is introduced in Section 3.4.

3 A proper account of meaning transfers

In this section we shall see that the lexicon we propose, provides a proper account of the lexical phenomena that motivated its definition: ill typed readings are rejected, coerced readings are handled, felicitous copredications are analysed while infelicitous ones are rejected. Some particular case of coerced readings are given a finer analysis as the polysemy of deverbals (nouns derived from verbs, like "*construction*"), or fictive motion. Finally we introduce coercive subtyping for system F which handles general coercions corresponding to ontological inclusion.

3.1 Coercions and copredication

One can foresee what is going to happen, using the lexicon given in Figure 3 with sentences like:

(6) a. Liverpool is spread out.
b. Liverpool voted.
c. Liverpool won.

(7) Liverpool is spread out and voted (last Sunday).

(8) # Liverpool voted and won (last Sunday).

Our purpose is not to discuss whether this or that sentence is correct, nor whether this or that copredication is felicitous, but to provide a formal and computational model which given sentences that are assumed to be correct, derives the correct readings, and which given sentences that are said to be incorrect, fails to provide a reading.

Ex. (6a) This sentence leads to a type mismatch $\texttt{spread\_out}^{Pl\to\mathsf{t}}\texttt{Lpl}^{T}$, since "*spread out*" applies to "*places*" (type Pl) and not to "*towns*" as "*Liverpool*". It is solved using the optional term ${t_3}^{T\to Pl}$ provided by the entry for "*Liverpool*", which turns a town (T) into a place (Pl) $\texttt{spread\_out}^{Pl\to\mathsf{t}}({t_3}^{T\to Pl}\texttt{Lpl}^{T})$ – a single optional term is used, the (F) / (R) difference is useless.

Ex. (6b) and (6c) are treated as the previous one, using the appropriate optional terms.

Ex. 7 In this example, the fact that "*Liverpool*" is "*spread out*" is derived as previously, and the fact that "*Liverpool voted*" is obtained from the transformation of the town into people, who can vote. The two can be conjoined by the polymorphic "*and*" defined above as term 1 ($\&^{\Pi}$) because these transformations are flexible: one can use one and the other. We can make this precise using only the rules of second order typed lambda calculus. The syntax yields the predicate $(\&^{\Pi}(\texttt{spread\_out})^{Pl\to\mathsf{t}}\texttt{voted}^{P\to\mathsf{t}})$ and consequently the type variables should be instantiated by $\alpha := Pl$ and $\beta := P$ and the exact term is $\&^{\Pi}\{Pl\}\{P\}\texttt{spread\_out}^{Pl\to\mathsf{t}}\texttt{voted}^{P\to\mathsf{t}}$ which reduces to:
$\Lambda\xi.\ \lambda x^{\xi}.\ \lambda f^{\xi\to\alpha}.\ \lambda g^{\xi\to\beta}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}\ (\texttt{spread\_out}^{Pl\to\mathsf{t}}\ (f\ x))(\texttt{voted}\ (g\ x)))$.
Syntax says that this term is applied to "*Liverpool*". Consequently, the instantiation $\xi := T$ happens and the term corresponding to the sentence is, after some reduction steps,
$\lambda f^{T\to Pl}.\ \lambda g^{T\to P}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}(\texttt{spread\_out}^{Pl\to\mathsf{t}}\ (f\ \texttt{Lpl}^{T}))(\texttt{voted}^{P\to\mathsf{t}}\ (g\ \texttt{Lpl}^{T})))$.
Fortunately the optional λ-terms $t_2 : T \to P$ and $t_3 : T \to Pl$ are provided by the lexicon, and they can both be used, since none of them is rigid. Thus we obtain, as expected
$(\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}\ (\texttt{spread\_out}Pl \to \mathsf{t}\ ({t_3}^{T\to Pl}\ \texttt{Lpl}^{T}))(\texttt{voted}^{Pl\to\mathsf{t}}\ ({t_2}^{T\to P}\ \texttt{Lpl}^{T})))$.

Ex. 8 The last example is rejected as expected. Indeed, the transformation of the town into a football club prevents any other transformation (even the identity) to be used in the polymorphic "*and*" that we defined above. We obtain the same term as above, with `won` instead of `spread_out`. The corresponding term is:
$\lambda f^{T\to Pl}.\ \lambda g^{T\to P}.\ (\&^{\mathsf{t}\to\mathsf{t}\to\mathsf{t}}\ (\texttt{won}^{F\to\mathsf{t}}\ (f\ \texttt{Lpl}^{T}))(\texttt{voted}^{P\to\mathsf{t}}\ (g\ \texttt{Lpl}^{T})))$
and the lexicon provides the two morphisms that would solve the type conflict, but the one turning the Town into its football club is rigid, i.e. we can solely use this one. Consequently the sentence is semantically invalid.

3.2 Fictive motion

A rather innovative extension is to apply this technique to what Talmy called *fictive motion* [72]. Under certain circumstances, a path may introduce a virtual traveller following the

path, as in sentences like:

(9) Path GR3 descends for two hours.

Because of the duration complement "*two hours*", one cannot consider that descends means that the altitude decreases as the curvilinear abscissa goes along the path. One ought to consider someone who follows the road. We model this by one morphism associated with the "*Path GR3*" and one with "*descends*". The first coercion turns the "*Path GR3*" from an immobile object into an object of type "*path*" that can be followed and the second one coerces "*descends*" into a verb that acts upon a "*path*" object and introduces an individual following the path downwards – this individual, which does not need to exist, is quantified, yielding a proposition that can be paraphrased as "*any individual following the path goes downwards for two hours*". [51, 50]

3.3 Deverbals

Deverbals are nouns that correspond to action verbs, as "*construction*" or "*signature*". Usually they are ambiguous between result and process. We showed that our idiosyncratic model is well adapted since their possible senses vary from one deverbal to another, even if the verbs are similar and the suffix is the same.

(10) a. The construction took three months.
b. The construction is of traditional style.
c. * The building that took three months was painted white.

(11) a. The signature was illegible.
b. The signature took three months.
c. * Although it took three months the signature was illegible.
d. Although it took one minute, the signature was illegible.

We showed that a systematical treatment of deverbal meaning as the one proposed by the type-driven approach does not properly account for the data. Indeed, the possible meanings of a deverbal are more diverse than result and event, and there are no known rules to make sure the deverbal refers to the event. Consequently, word entries in the lexicon must include lexical information such as the possible meanings of the deverbal. These meanings can be derived from the event expressed by the verb: meanings usually include the event itself (but not always), the result (but not always), and other meanings as well like the place where the event happens (e.g. English noun "*pasture*"). This lexical information can be encoded in our framework, with one principal meaning and optional terms for accessing other senses and the flexibility or rigidity of these optional terms – they are usually rigid, and copredication on the different senses of a deverbal is generally infelicitous. We successfully applied our framework and treatment to the semantic of deverbals, to the restrictions of selection (both for the deverbal and for the predicate that may apply to the deverbal), to meaning transfers, and to the felicity of copredications on different senses of a deverbal [63].

3.4 Coercive subtyping and ontological inclusions

As we said earlier on, ontological inclusions like "*Human beings are animals.*", would be better modelled by optional terms that are available for any word of the type, instead of anchoring them in words and repeating these terms for every word of this type. The model we described can take these subtyping inclusions into account as standard coercions, by

transitivity

$$\frac{A < B \qquad B < C}{A < C}$$

covariance and contravariance of implication

(if identity coercions are allowed only the left most rule is needed)

$$\frac{A < B \qquad C < D}{D \to A < C \to B} \qquad \frac{A < B}{T \to A < T \to B} \qquad \frac{A < B}{B \to T < A \to T}$$

quantification over types

$$\frac{U < T[X]}{U < \Pi X.\ T[X]}\ X \textit{ not free in } U \qquad \frac{U < \Pi X.\ T[X]}{U < T[W]}$$

Figure 4 Rules for coercive subtyping in system F.

specifying that a word like "*human being*" introduces a transformation into an "*animal*". But this is somehow heavy, since one should also say that "*human beings*" are "*living beings*" etc. Any predicate, that applies to a class, also applies to an ontologically smaller class. For instance, "*run*" that applies to "*animals*" also applies to "*human beings*", because "*human*" is a subtype of "*animals*". These subtype coercions look type-driven, and, consequently, would be more faithfully modelled with a proper notion of subtyping.

Coercive subtyping, introduced by Luo and Soloviev [40, 70] for variants of Martin-Löf type theory, corresponds quite well to these particular transformations. One starts with *a transitive and acyclic set of coercions between base types, with at most one coercion between any two base types*, and from these given coercions rules derive coercions between complex types, preserving the property that there is at most one coercion between any two types.

This kind of subtyping seems adequate for modelling ontological inclusions. Indeed, such ontological inclusions when viewed as functions always are the identity on objects, hence there cannot be two different manners to map them in the larger type. Furthermore, other notions of subtyping that have been studied for higher order type theories are very complicated with tricky restrictions on the subtyping rules. [15, 37]

Coercive subtyping, noted $A_0 < A$, can be viewed as a short hand for allowing a predicate or a function which applies to A-objects to apply to an argument whose type A_0 is not the expected type A but a subtype A_0 of A. Hence coercive application is exactly what we were looking for:

coercive application

$$\frac{f : A \to B \quad u : A_0 \quad A_0 < A}{(f\ u) : B}$$

The subtyping judgements, which have the structure of categorical combinators, are derived with very natural rules given in Figure 4. These rules simply encode transitivity, covariance and contravariance of implicative types (arrow types), and quantification over type variables.

It should be observed that, given constants $\mathsf{c}_{i\to j}$ representing the coercions from a base type e_i to a base type e_j, any derivable coercion $T < U$ can be depicted by a linear Λ-term

$m : U$ of system F or $\Lambda\mathsf{Ty}_n$with a single occurrence of a free variable $x : T$ and occurrences of the constants $\mathsf{c}_{i\to j}$. The construction of the term according to the derivation rules is defined as follows:

- transitivity
$$\frac{x : A < t : B \qquad y : B < u : C}{x : A < u[y := t] : C}$$
- covariance and contravariance of implication
  - $$\frac{x : A < t : B \qquad z : C < u : D}{f : D \to A < \lambda z^C.\ t[x := f(u)] : C \to B}$$
  - $$\frac{x : A < t : B}{f : T \to A < \lambda w^T.\ t[x := f(w)] : T \to B}$$
  - $$\frac{x : A < t : B}{g : B \to T < \lambda x^A.\ g(t) : A \to T}$$
- quantification over types
  - $$\frac{u : U < t : T[X]}{u : U < \Lambda X.\ t : \Pi X.\ T[X]}\ X \textit{ not free in } U$$
  - $$\frac{u : U < t : \Pi X.\ T[X]}{u : U < t\{W\} : T[W]}$$

As easy induction shows that:

▶ Proposition 2. All terms derived in this system are linear, with a single occurrence of a single free variable (whose type is on the left of "<").

From this one easily concludes that:

▶ Proposition 3. Not all Λ-terms of system F can be derived in the subtyping system.

Any derivation c of $\mathsf{e}_i < \mathsf{e}_j$ for base types e_i, e_j is equivalent to a coercion $\mathsf{c}_{i\to j}$, i.e. our derivation system does not introduce new coercions between base types. This kind of result is similar to coherence in categories: given a compositional graph G, the free cartesian category over G does not contain any extra morphism between objects from the initial compositional graph. Here is the precise formulation of this coherence result:

▶ Proposition 4. Given a derivation of $\mathsf{e}_i < \mathsf{e}_j$ for base types e_i, e_j whose associated Λ-term is $\tilde{\mathcal{C}}$, the normal form $\mathcal{C}$ of $\tilde{\mathcal{C}}$ is a compound of $\mathsf{c}_{h\to k}$ applied to $x : \mathsf{e}_i$, which, because of the assumptions on coercions, must be $\mathsf{c}_{i\to j}$.

Proof. As seen above, a deduction of $T < U$ clearly corresponds to a linear Λ-term of system F, whose only free variable is $x : T$ with the $\mathsf{c}_{i\to j}$ as constants. Hence it has a normal from which also has a single free variable $x : T$ and $\mathsf{c}_{i\to j}$ as constants.

Let us show that any normal Λ-term $\mathcal{C}$ of type e_j with a single free variable $x : \mathsf{e}_i$ and constants $\mathsf{c}_{i\to j} : \mathsf{e}_i \to \mathsf{e}_j$ is a compound of $\mathsf{c}_{i\to j}$ applied to x^{e_i}, i.e. is a term of C_i defined by:

- $x^{\mathsf{e}_i} \in C_i$
- if $c^{\mathsf{e}_j} \in C_i$ then $(\mathsf{c}_{j\to k}(c))^{\mathsf{e}_k} \in C_i$

We proceed by induction on the number of occurrences of variables and constants in the normal term $\mathcal{C}$, whose form is, as said in Proposition 1:

$$\mathcal{C} \;=\; \overbrace{(\;\lambda {x_i}^{X_i} \mid \Lambda X_j\;)^*}^{\substack{\text{sequence of}\\ \lambda \text{ and } \Lambda \text{ abstractions}}} \overbrace{(\cdots(\;h^{(\Pi X_k | X_l \to)^* Z}}^{\substack{\text{head}\\ \text{variable}}} \overbrace{(\{W_k\} \mid {t_l}^{X_l})^*\;)\cdots)}^{\substack{\text{sequence of } \{\cdots\} \text{ and } (\cdots) \text{ applications}\\ \text{to types } W_k \text{ and normal terms } {t_l}^{X_l}}}$$

If the term $\mathcal{C}$ corresponds to a proof of $\mathsf{e}_i < \mathsf{e}_j$ there is no $(\;\lambda {x_i}^{X_i} \mid \Lambda X_j\;)$ in front, because e_j is neither of the form $U \to V$ nor of the form $\Pi X.\, T[X]$. What may be the head variable? It is either the only free variable of this term, namely x^{e_i}, or a constant i.e. some coercion $\mathsf{c}_{k\to l}$.

- If the head variable is x^{e_i} then, because of its type, the $(\;\{W_k\} \mid {t_l}^{X_l}\;)^*$ part of the term contains no application to a type nor to a term, hence $\mathsf{e}_i = \mathsf{e}_j$ and the normal form is x^{e_i}, which is in C_i
- If the head variable is is some $\mathsf{c}_{k\to l}$, which because of its type, may only be applied to a normal term ${t_l}^{X_l}$ of type e_k. This normal term is a normal term of type e_k with x^{e_i} as its single free variable and the constants $\mathsf{c}_{j\to l}$. As ${t_l}^{X_l}$ has one symbol less than $\mathcal{C}$, we can conclude that ${t_l}^{X_l}$ is in C_i hence $\mathcal{C} \in C_i$.

Hence in any case the normal form $\mathcal{C} : \mathsf{e}_j$ of the term $\tilde{\mathcal{C}} : \mathsf{e}_j$ is in C_i.

Now, given that the coercions $\mathsf{c}_{i\to j}$ enjoy $\mathsf{c}_{k\to j} \circ \mathsf{c}_{i\to j} = \mathsf{c}_{i\to k}$ (as part of our condition on base coercions) it is easily seen that the only term of type e_j in C_i is $\mathsf{c}_{i\to j}$. ◀

We think that this coherence result can be improved by showing that there is at most one normal term corresponding to a derivation $S < T$, although the proof is likely to use some variant of reducibility candidates [24, 25].

An alternative presentation. The rules for coercive subtyping given above follow a natural deduction style, as lambda terms of system F. Nevertheless, an alternative formulation of the quantifier elimination rule is possible. It requires identity axioms (whose term is identity) to derive obvious subtyping relations.

alternative quantifier elimination rule (sequent calculus style)

$$\frac{s : S[T] < t : U}{\dot{s} : \Pi X.\; S[X] < t[s := \dot{s}\{T\}]}$$

4 Compositional semantics issues: determiners, quantifiers, plurals

So far we have focused on phenomena in *lexical semantics* that are usually left out of standard models but properly accounted for by our model. However, we must also have a look at compositional semantics, that is the logical structure of a sentence, to see whether our model still properly analyses what standard compositional models do, and possibly, provide a better analysis. Fortunately, sentence structures are correctly analysed but furthermore our extended setting is quite appealing for some classical issues in *formal semantics* like determiners and quantification, or plurals, as we show in this section.

4.1 Determiners and quantifiers

The examples presented so far only involved proper names because we chose to extend the treatment of definite descriptions with the ι operator of some authors [68, 22, 75, 76], to indefinite articles and quantifiers. This slightly differs from the usual Montagovian setting, the one we used for "*some*" in subsection 1.2. This standard treatment of quantification can be adapted to many-sorted logic provided the two predicates, the common noun and the verb phrase, apply to the same type, or that the conjunction and implication respectively involved in existential and universal quantification allow some coercions, in the style of the polymorphic $\&^{\Pi}$.

We adopt the view of quantified, definite, and indefinite noun phrases as *individual terms* by using generic elements (or choice functions) [66, 65, 67] as initiated by Russell [68] and formalised by Hilbert [26], Ackerman [2], before Hilbert and Bernays provided a thorough presentation and discussion in [27]. This view of quantification has been adapted to linguistics by researchers like von Heusinger see e.g. [22, 75, 76], and is not that far from Steedman treatment of existential quantifiers by choice functions [71] although there are some differences that we shall not discuss here.

How do we tune our model, in particular the types, if instead of the proper name "*Liverpool*", the examples contain "*the town*", "*a town*", "*all towns*", or "*most towns*"? Indefinite determiners, quantifiers, generalised quantifiers,... are usually viewed as functions from two predicates to propositions, one expressing the restriction and the other the main predicate see e.g. [59]

As we said, and this is especially true in a categorial setting such as the one Moot implemented [49], the syntactic structure usually closely corresponds to the semantic structure. But the usual treatment of quantification that we saw in subsection 1.2 infringes this correspondence, since the semantic term "$(\lambda x.$ *Keith played* $x)$" in the semantic representation (12c) of example (12a) has no corresponding constituent in its syntactical structure (12b):

(12) a. *sentence:* Keith played some Beatles song.

b. *syntactical structure:* (Keith (played (some (Beatles song))))

c. *semantical structure:* `(some (Beatles song))` $(\lambda x.$ `Keith played` $x)$

Another criticism that applies to the usual treatment of quantifiers is the symmetry that it wrongly introduces between the main predicate and the class over which one quantifies. For instance, the two sentences below (13a,13b) usually have the same logical form (13c):

(13) a. Some politicians are crooks.

b. ? Some crooks are politicians.

c. $\exists x.$ `politician`(x) & `crook`(x)

Hence, in accordance with syntax, we prefer to consider that a quantified noun phrase is by itself some individual – a generic one which does not refer to a precise individual nor to a collection of individuals. As [75] we use η for indefinite determiners (whose interpretation picks up a new element) and ι for definite noun phrases[9] (whose interpretation picks up the most salient element). Regarding the deductive rules for handling these operators ι and η, both correspond to Hilbert's ϵ: only their interpretations in the discursive context differ.

Given a first order language $\mathcal{L}$, epsilon terms and formulae are defined by mutual recursion:

[9] Actually [75] writes ϵ instead of ι. We do not follow his notation because we also use Hilbert's ϵ with its traditional meaning.

- Any constant or variable from $\mathcal{L}$ is a term.
- $f(t_1, \ldots, t_p)$ is a term provided that each t_i is a term and f is a function symbol of arity p.
- $\epsilon_x A$ and $\tau_x A$ are terms if A is a formula, x is a variable — any free occurrence of x in A is bound by ϵ_x or $\tau_x A$.
- $s = t$ is a formula whenever s and t are terms.
- $R(t_1, \ldots, t_n)$ is a formula provided each t_i is a term and R is a relation symbol of arity n.
- $A\&B$, $A \vee B$, $A \Rightarrow B$ are formulae if A and B are formulae.
- $\neg A$ is formula if A is a formula.

As the example below shows, a formula of first order logic can be recursively translated into a formula of the epsilon calculus, without surprise:

$$\forall x \exists y P(x,y) = \exists y P(\tau_x P(x,y), y) = P(\tau_x P(x, \epsilon_y P(\tau_x P(x,y), y)), \epsilon_y P(\tau_x P(x,y), y)) \quad (14)$$

Admittedly the epsilon translations of usual formulae may look quite complicated – at least we are not used to them.

The deduction rules for τ and ϵ are the usual rules for quantification:

- From $A(x)$ with x generic in the proof (no free occurrence of x in any hypothesis), infer $A(\tau_x.\ A(x))$.
- From $B(c)$ infer $B(\epsilon_x B(x))$.

The other rules can be found by duality:

- From $A(x)$ with x generic in the proof (no free occurrence of x in any hypothesis), infer $A(\epsilon_x \neg A(x))$.
- From $B(c)$ infer $B(\tau_x \neg B(x))$.

Hence we have $F(\tau_x F(x)) \equiv \forall x.F(x)$ and $F(\epsilon_x F(x)) \equiv \exists x.\ F(x)$ and because of negation, one only of these operators is needed, usually the ϵ operator is used, and the resulting logic is known as the *epsilon calculus*: $\epsilon_x A(x) = \tau_x \neg A(x)$

Hilbert in [27] turned these symbols into a mathematically satisfying deductive system that properly describes quantification. The first and second epsilon theorem basically say that this is an alternative formulation of first order logic.

First epsilon theorem When inferring a quantifier free formula C without ϵ from quantifier free formulae Γ without ϵ, the derivation can be done within quantifier free predicate calculus.

Second epsilon theorem When inferring a formula C without ϵ symbol from formulae Γ not involving the ϵ symbol the derivation can be done within usual predicate calculus.

The epsilon calculus, restricted to the translations of usual formulae with the help of the two epsilon theorems, provided the first correct proof of Herbrand theorem (much before mistakes where found and solved by Goldfarb) and a way to prove, during the same period as Gentzen worked on the same question of the consistence of Peano arithmetic with the epsilon substitution method [27]. Later, Asser [7] and Leisenring [36] worked on the epsilon calculus more specifically in order to have models and completeness. Nevertheless, as one can read on Zentralblatt (see e.g. [14, 44]) these results are misleading as well as the posterior corrections. Only the proof theoretical aspects of the epsilon calculus seem to have been further investigated with some success by Mints[10] [45] or by Moser and Zach [54].

[10] We are sorry to learn that the great logician Grigori Mints just passed away on May 29, 2014.

In a typed model, a predicate that applies to α-objects is of type $\alpha \to \mathsf{t}$. Consequently the semantic constant ι corresponding to "*the*" introducing definite descriptions, should be of type: $(\alpha \to \mathsf{t}) \to \alpha$, and, in order to have a single constant ι, its type should be $\Pi\alpha.\ (\alpha \to \mathsf{t}) \to \alpha$.[11] Therefore, if we have a predicate dog that applies to entities of type animate the term $\iota(\mathsf{dog})$ (written $\iota x.\ \mathsf{dog}(x)$ in untyped models), i.e. the semantics of "*the dog*" is of type animate.... but we would like this term to enjoy the property dog! How could we say so, since the predicate dog does not appear in ι, but only its type. Indeed, only "*animate*" entities appear in ι as an instantiation of α. We solve this by adding a presupposition[12] $P(\iota(P))$ for any P of type $\alpha \to \mathsf{t}$, as soon as some entity enjoying P is uttered.[13]

As advocated by von Heusinger and others, indefinite descriptions that are in fact existentially quantified noun phrases are processed similarly using Hilbert's ϵ instead of ι: both ι and ϵ are constants of type $\Pi\alpha.\ (\alpha \to \mathsf{t}) \to \alpha$. Determiners are modelled in our framework by such typed constants, see [66, 65, 67]. This solution avoids the problems evoked in examples (12a) and (13b). For instance, regarding the unwanted asymmetry in the semantics of (13b) the formulae $P(\epsilon_x Q(x))$ and $Q(\epsilon_x P(x))$ are not equivalent – and neither of them is equivalent to a first order formula, but $Q(\epsilon_x P(x))$, with $P(\epsilon_x P(x))$ which is added as a presupposition, entails $P\&Q(\epsilon_x P\&Q(x)) \equiv \exists x.\ P(x)\&Q(x)$.

It should be observed that generics introduced by Hilbert's operators fit better into our typed and many-sorted semantic representations. Indeed, intuitively it is easier to think of a generic "*politician*" or "*song*" than it is to think of a generic "*entity*" or "*individual*".

One can even introduce constants that model generalised quantification. They are typed just the same way, and this construct can be applied to compute the logical form of statement including the "*most*" quantifier, as exposed in [64]. It does not mean that we have the sound and complete proof rules nor a model theoretical interpretation: we simply are able to automatically compute logical forms from sentences involving generalised and vague quantifiers such as "*most*", "*many*", "*few*".

4.2 Individuals, plurals and sets in a type-theoretical framework

The organisation of the types also allows us to handle simple facts about plurals, as shown in [52, 41] – which resembles some of Partee's ideas [58]. Here are some classical examples involving plurals, exemplifying some typical readings for plurals:

(15) a. *Keith met.
b. Keith and John met. (unambiguous).

(16) a. *The student met.
b. The students met. (unambiguous, one meeting)

(17) a. The committee met. (unambiguous, one meeting)
b. The committees met. (ambiguous: one big meeting, one meeting per committee, several meetings invoking several committees)

[11] An alternative type working with any predicate $\widehat{\alpha}$ that corresponds to a type α, would be $\Pi\alpha.\ \alpha$.
[12] A presupposition is a proposition which is not explicitly stated but which is assumed by the uttered proposition and by its negation as well: "*Keith stopped smoking.*" and "*Keith did not stop smoking*" both presuppose "*Keith used to smoke.*". Observe that a typing judgement $t : \mathsf{a}$ is not easy to refute, as a presupposition: indeed after "*The dog is sleeping on the sofa.*" one can hardly answer "*It is not an animal.*" or "*It is not a dog.*" although one can say "*It is not sleeping*".
[13] If the predicate P corresponds to a type τ i.e. $P = \widehat{\tau}$, this presupposition is better written as $\iota(\widehat{\tau}) : \tau$.

q $\quad \Lambda\alpha.\ \lambda x^{\alpha}.\ \lambda y^{\alpha}.\ x = y$
$*$ $\quad \Lambda\alpha\lambda P^{\alpha\to\mathsf{t}}.\ \lambda Q^{\alpha\to\mathsf{t}}.\ \forall x^{\alpha}.\ Q(x) \Rightarrow P(x)$
$\#$ $\quad \Lambda\alpha\lambda R^{(\alpha\to\mathsf{t})\to\mathsf{t}}.\ \lambda S^{\alpha\to\mathsf{t}\to\mathsf{t}}.\ \forall P^{\alpha\to\mathsf{t}} S(P) \Rightarrow R(P)$
c $\quad \Lambda\alpha.\ \lambda R^{(\alpha\to\mathsf{t})\to\mathsf{t}}.\ \lambda P^{\alpha\to\mathsf{t}}.\ \forall x^{\alpha}.\ P(x) \Rightarrow \exists Q^{\alpha\to\mathsf{t}} Q(x) \wedge (\forall y^{\alpha} Q(y) \Rightarrow P(y)) \wedge R(Q)$

Figure 5 Some operators for plurals.

(18) a. The students wrote a paper. (unambiguous)
b. The students wrote three papers. (covering)

Such readings are derivable in our model because one can define in F operators for handling plurals. Firstly, one can add, as a constant, a cardinality operator for predicates $||\_|| : \Pi\alpha.\ (\alpha \to \mathsf{t}) \to \mathbb{N}$ where $\mathbb{N}$ are the internal integers of system F, namely $\mathbb{N} = \Pi X.\ (X \to X) \to (X \to X)$, or a predefined integer type as in Gödel system T – this might be problematic if infinitely many objects satisfied the predicate, but syntax and restriction of selection can make sure it is only applied when it makes sense. Secondly, as shown in Figure 5, we can have operators for handling plurals: q (turning an individual into a property/set, a curried version of equality), $*$ (distributivity), $\#$ (restricted distributivity from sets of sets to its constituent subsets), c (for coverings), etc. The important fact is that the computation of such readings uses exactly the same mechanisms as lexical coercion. Some combinations are blocked by their types, but optional terms coming either from the predicate or from the plural noun may allow an a priori prohibited reading. To be precise we also provide specific tools for handling groups that are singular nouns, each of which denoting a set. All these functions are easily implemented in a typed functional programming language like Haskell, in the style of [73].

5 Comparison with related work and conclusion

5.1 Variants and implementation

Some variation is possible in the above definition of the Montagovian generative lexicon without changing its general organisation. For instance, as suggested in the beginning of section 2 the set of base types can be discussed. We proposed to use classifiers as base types of a language with classifiers, because classifiers are linguistically and cognitively motivated classes of words and entities. But it is fairly possible that other sets of base types are better suited in particular for specific applications [43].

In relation to this issue, the inclusion between base types, which in our model are morphisms, can be introduced with words or as general axioms. We prefer the first solution that allows idiosyncratic behaviours, dependent on words as explained in subsection 2.4 with "*classe*" and "*promotion*". Nevertheless when dealing with ontological inclusions, or other very general coercions, we think a subtyping approach is possible and reduces the size of the lexicon, this is why we are presently exploring coercive subtyping.

The type we gave for predicates can also vary: it could be systematically $\mathsf{e} \to \mathsf{t}$, but as explained in paragraph 4, types $\mathsf{u} \to \mathsf{t}$ are possible as well – but transition from one form to another is not complicated.

An important variant is to define the very same ideas within a compositional model like λ-DRT [56] the compositional view of Discourse Representation Theory [29] which can, as its name suggests, handle discursive phenomena. Thus one can integrate the semantical and lexical issues presented here into a broader perspective. This can be done, and in fact

several applications of the model presented here are already included in the Grail parser by Richard Moot, in particular for French [49]. The *grammar* was automatically extracted from annotated corpora, but unfortunately the refined semantic terms we need can only be typed by hand. Consequently we only tested the semantic analyses described herein on a small specific lexicon. For instance, our treatment of fictive motion (cf. subsection 3.2) has been tested with a detailed lexicon for spatial semantics, but with λ-DRT [50] rather than plain lambda calculus [51]. The Grail parser is written with Prolog and as far as semantics is concerned, a functional programming language like CaML or Haskell would be better suited, as van Eijck and Unger show in [73].

5.2 Comparison with related work

There are many similarities with the contemporary work by Asher and Luo [4, 39, 16].

A first difference is the type system. Our type system, F, is quite powerful but simple: four-term building operations, and two reduction rules. Luo makes use of a version of Modern Type Theories (MTT), closed to the Unifying Theory of dependent Types (UTT), whose expressive power and computational complexity is difficult to compare: it is predicative but it includes dependent types. Hence it is not clear whether MTT better characterises the logic needed for meaning assembly. Quantification over type variables is quite comparable and admits $\forall \alpha : CN$ (CN are common nouns) which is quite convenient although it can certainly be encoded within system F using the fact that finite sums can be defined in system F, hence $x : \alpha, \alpha : CN$ can be rephrased if there are finitely many CN. This is both a positive and negative feature of system F: it can encode many things, but encodings are often dull. A possible solution, similar to [69], is to introduce predefined types F with specific reduction schemes – e.g. adding integers as in Gödel's system T.

Regarding coercions, Luo [38] makes an extensive use of coercive subtyping, which he introduced with Soloviev [70]: as said in their paper this kind of subtyping may also work well with system F. So we can say that the system of Luo is very similar. Dependent types and predicative quantification may be closer to what we wish to model, but the formal diversity of the numerous employed rules may result in an obscure formalisation. The typed system at work in Asher's view [3] is a simple type theory extended with type constructors and operations imported from category theory. The theory extends cartesian closed category with a few of the many operations that one finds in topos theory, like being a subobject. This approach is difficult to compare with the two above, since it does not belong to the same family: morphisms do not represent (quotiented) proofs of some logic, they are closer to a set theoretic interpretation.

Another ingredient of our models are base types. Asher leaves the set of base types open, but rather small (say a dozen) : e, t, physical objects, etc., with a linguistically motivated subtyping relation $\sqsubset$ defined over these types. Luo, especially in his later article [39], wants to equate base types with common nouns (also with coercions between them), and this is a possible compromise between any formula and the minimal base type system which makes it difficult to express some selectional restrictions with types. However it seems that there are too many of them, since not every common noun appears as a restriction of selection for another word in a dictionary. Dealing with classifiers as base types is a recent proposal of ours which seems cognitively and linguistically motivated. It is worth exploring this hypothesis empirically in tests over corpora.

The subtyping relation between base types are language independent in these two models, i.e. they are not triggered by words, but simply by types. We opted for a compromise in which only ontological inclusions are type-driven, using coercive subtyping, while other coercions are word-driven.

Regarding the general organisation of the lexicon and its composition modes, the same difference applies. While according to Asher and Luo, types determine the coercions, in our approach the coercions are provided by the terms in the lexicon, i.e. by the words themselves and not by their types, with an exception for ontological inclusions. The recent claim by Luo that base type should be common nouns (that are words) partly blurs the differences between on the one hand the type-driven approaches of himself and Asher and, on the other hand, ours which is more idiosyncratic being based on words and terms that are known to be arbitrary.

Finally one may wonder whether we finally derive similar logical forms. They actually are quite similar: we derive higher order many-sorted logical formulae, Asher derives formulae in a category that can be seen as an intuitionnistic set theory, which works with sorts, and Luo derives formulae of type theory. All these are more or less the same: higher order is possible, although not extensively used in examples, and there are sorts or types.

A possible difference may lie in the distance between syntax and semantics. Indeed, the effective computability of the semantic representations requires a specific treatment of the common structures in compositional semantics like determiners, quantifiers, plurals,... and to be integrated in a general analysis that also includes phenomena like time or aspect. For the time being we did more on such issues than the others, but I am pretty sure that a similar treatment is possible within the approach developed by Asher and Luo.

5.3 Perspectives

Apart from fixing up the optimal variant among the possible variants of our model, to study and develop the convergence with related work, or to pursue the implementation, there are some questions both on type theory and on linguistic modelling, both theoretical and practical, that deserve to be further studied.

The *acquisition* of the semantic lexicon has both theoretical and practical aspects. In particular, how could one acquire the optional lambda terms that represent coercions? Syntactic information on words can be automatically extracted: indeed, Moot's parser that we used to experiment our type theoretical semantic analyses was automatically acquired [48, 47]. By now there are some techniques [78] to extract the usual lambda terms of Montague semantics of subsection 1.2 that represent the argument structure of words. Machine learning (see e.g. [23]) and serious games (that are games with an outcome besides entertainment, and in particular collaborative games with a purpose [74] that have been shown to be quite efficient for constructing linguistic ressources) are also able to learn relations between words see e.g. [18, 31]. However, up to now there are no learning algorithms for acquiring a set of base type, nor for determining given a set of base type, the optional lambda terms, and our experiments with Moot parser were performed using a hand typed semantic lexicon.

On the logical side there are many intriguing questions.

- One is the relation in a type system with sorts between the (higher order) predicate calculus and the type system, exemplified by the relation between type judgements $x : T$ that, as linguistic presuppositions, cannot be denied and predicates $\widehat{T}(x)$ that can be denied.
- The Hilbert operator ϵ, which looks more natural in this typed system, deserves to be further studied. Since most of the results are false but Hilbert's original results, the study of both the deductive system and the interpretation of those operators is appealing. In particular, we are puzzled by formulae with Hilbert operators that have no corresponding formula in usual logic.

- The coercive subtyping we introduced in this paper should also be further explored, e.g. by proving that there is at most one coercion between any two types.
- It is quite clear that we do not need the full power of system F: we chose this system of variable types and quantified types for its simplicity and elegance. Nevertheless one may wonder whether there is a simple restriction that would be sufficient. Linear versions of system F both have a lower complexity [30] and allow a finer grained treatment of the constraints on sense compatibility [42].

Regarding computational linguistics, and applications to natural language processing, the way the discourse context is handled is important. In particular, the permanence and the propagation of constraints (e.g. on sense compatibilities) through linguistic structure deserves to be further studied. Observe that:

(19) a. This salmon was living nearby Scottish coast. It was delicious.
b. ? This salmon that was living nearby Scottish coast was delicious.
c. * This salmon was living nearby Scottish coast and was delicious.

We believe that the type theoretical and many-sorted view presented in this paper may shed new light on classical challenges of natural language semantics. A known difficult example is the semantics of mass nouns, like *wine*, which can be quantified:

(20) a. He drank some wine.
b. He drank all the wine.

Thanks. Special thanks to Sergeï Soloviev for his explanations on coercive subtyping during my CNRS sabbatical at IRIT.

Many thanks to Christian Bassac who introduced me to the topic, to my coauthors Bruno Mery (Bordeaux), Richard Moot (Bordeaux), Michele Abrusci (Roma), Laurent Prévot (Aix), Livy Real (Curitiba). I also thanks for helpful discussions Nicholas Asher, Zhaohui Luo, Marta Abrusan, Claire Beyssade, Heather Burnett, Sarah-Jane Conrad, Francis Corblin, Fabio Del Prete, Alda Mari, Hazel Pearson.

Finally let me thank the anonymous reviewers for providing unusually insightful comments, and the editors for their patience.

References

1 Vito Michele Abrusci and Christian Retoré. Quantification in ordinary language: from a critic of set-theoretic approaches to a proof-theoretic proposal. In Peter Schröder-Heister, editor, *14th Congress of Logic, Methodology and Philosophy of Sciences*, 2011.
2 W. Ackermann. Begründung des "tertium non datur" mittels der Hilbertschen Theorie der Widerspruchsfreiheit. *Mathematische Annalen*, 93:1–36, 1924.
3 Nicholas Asher. *Lexical Meaning in context – a web of words*. Cambridge University press, 2011.
4 Nicholas Asher and Zhaohui Luo. Formalization of coercions in lexical semantics. In Emmanuel Chemla, Vincent Homer, and Grégoire Winterstein, editors, *Sinn und Bedeutung 17*, pages 63–80, 2012. `http://semanticsarchive.net/sub2012/`.
5 Nicholas Asher and James Pustejovsky. The metaphysics of words in contexts, 2000.
6 Nicolas Asher. A type driven theory of predication with complex types. *Fundamenta Informaticae*, 84(2):151–183, 2008.
7 Gunter Asser. Theorie der logischen auswahlfunktionen. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 1957.

8 Christian Bassac, Bruno Mery, and Christian Retoré. Towards a Type-Theoretical Account of Lexical Semantics. *Journal of Logic Language and Information*, 19(2):229–245, April 2010.

9 Denis Béchet and Alexander Ja. Dikovsky, editors. *Logical Aspects of Computational Linguistics – 7th International Conference, LACL 2012, Nantes, France, July 2–4, 2012. Proceedings*, volume 7351 of *Lecture Notes in Computer Science*. Springer, 2012.

10 Gilad Ben-Avi and Nissim Francez. Categorial grammars with ontology-refined types. In *Categorial grammars – an efficient tool for natural language processing*, pages 99–113, Montpellier, June 2004. C.N.R.S.

11 Manfred Bierwisch. Wörtliche bedeutung - eine pragmatische gretchenfrage. In G. Grewendorf, editor, *Sprechakttheorie und Semantik*, pages 119–148. Surkamp, Frankfurt, 1979.

12 Manfred Bierwisch. Semantische und konzeptuelle repräsentation lexikalischer einheiten. In R. Rùžička and W. Motsch, editors, *Untersuchungen zur Semantik*, pages 61–99. Akademie-Verlag, Berlin, 1983.

13 Reinhard Blutner. Lexical semantics and pragmatics. In Fritz Hamm and Thomas Ede Zimmermann, editors, *Semantics*, volume 10 (Sonderheft), pages 27–58, Hamburg, 2002. Buske.

14 J. T. Canty. Zbl0327.02013 : review of "on an extension of Hilbert's second ϵ-theorem" by T. B. Flanagan (journal of symbolic logic, 1975). *Zentralblatt Math.*

15 Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1/2):4–56, 1994.

16 Stergios Chatzikyriakidis and Zhaohui Luo. An account of natural language coordination in type theory with coercive subtyping. In Denys Duchier and Yannick Parmentier, editors, *7th International Workshop on Constraint Solving and Language Processing (CSLP'12). Selected and Revised Papers*, number 8114 in Lecture Notes in Computer Science. Springer, 2013.

17 Stergios Chatzikyriakidis and Zhaohui Luo. Adjectives in a modern type-theoretical setting. In Glyn Morrill and Mark-Jan Nederhof, editors, *FG*, volume 8036 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2013.

18 Philipp Cimiano and Johanna Wenderoth. Automatic acquisition of ranked qualia structures from the web. In John A. Carroll, Antal van den Bosch, and Annie Zaenen, editors, *ACL*. The Association for Computational Linguistics, 2007.

19 Robin Cooper. Copredication, dynamic generalized quantification and lexical innovation by coercion. In *Fourth International Workshop on Generative Approaches to the Lexicon*. Université de Genève, 2007.

20 Robin Cooper. Copredication, quantification and frames. In Pogodalla and Prost [60], pages 64–79.

21 D.A. Cruse. *Lexical semantics*. Cambridge textbooks in linguistics. Cambridge University Press, 1986.

22 Urs Egli and Klaus von Heusinger. The epsilon operator and E-type pronouns. In Urs Egli, Peter E. Pause, Christoph Schwarze, Arnim von Stechow, and Götz Wienold, editors, *Lexical Knowledge in the Organization of Language*, pages 121–141. Benjamins, 1995.

23 Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, New York, NY, USA, 2012.

24 Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse et son application: l'élimination des coupures dans l'analyse et la théorie des types. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92, Amsterdam, 1971. North Holland.

25 Jean-Yves Girard. *The blind spot – lectures on logic*. European Mathematical Society, 2011.

26 David Hilbert. Die logischen grundlagen der mathematik. *Mathematische Annalen*, 88:151–165, 1922.

27 David Hilbert and Paul Bernays. *Grundlagen der Mathematik. Bd. 2.* Springer, 1939. Traduction française de F. Gaillard, E. Guillaume et M. Guillaume, L'Harmattan, 2001.

28 Gérard P. Huet. *Résolution d'équations dans des langages d'ordre 1,2,...,ω.* Thèse de doctorat d'état, Université Paris VII, 1976.

29 Hans Kamp and Uwe Reyle. *From Discourse to Logic*. D. Reidel, Dordrecht, 1993.

30 Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1–2):163 – 180, 2004.

31 Mathieu Lafourcade and Alain Joubert. Computing trees of named word usages from a crowdsourced lexical network. In *IMCSIT*, volume Computational Linguistics – Applications (CLA'10), pages 439–446, 2010.

32 Sven Lauer. A comparative study of current theories of polysemy in formal semantics. Master's thesis, Cognitive science Osnabrück - Computational Linguistics, 2004.

33 Alain Lecomte and Myriam Quatrini. Figures of dialogue: a view from ludics. *Synthese*, 183:59–85, 2011.

34 Anaïs Lefeuvre, Richard Moot, and Christian Retoré. Traitement automatique d'un corpus de récits de voyages pyrénéens : analyse syntaxique, sémantique et pragmatique dans le cadre de la théorie des types. In SHS Web of Conferences, editor, *Congrès mondial de linguistique française*, pages 2485–2497, 2012.

35 Anaïs Lefeuvre, Richard Moot, Christian Retoré, and Noémie-Fleur Sandillon-Rezer. Traitement automatique sur corpus de récits de voyages pyrénéens : Une analyse syntaxique, sémantique et temporelle. In *Traitement Automatique du Langage Naturel, TALN'2012*, volume 2, pages 43–56, 2012.

36 Albert C. Leisenring. *Mathematical logic and Hilbert's ϵ symbol*. University Mathematical Series. Mac Donald & Co., 1967.

37 Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. Coherence and transitivity of subtyping as entailment. *Journal of Logic and Computation*, 10(4):493–526, 2000.

38 Zhaohui Luo. Contextual analysis of word meanings in type-theoretical semantics. In Pogodalla and Prost [60], pages 159–174.

39 Zhaohui Luo. Common nouns as types. In Béchet and Dikovsky [9], pages 173–185.

40 Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Inf. Comput.*, 223:18–42, 2013.

41 Bruno Mery, Richard Moot, and Christian Retoré. Plurals: individuals and sets in a richly typed semantics. In Shunsuke Yatabe, editor, *Logic and Engineering of Natural Language Semantics 10 (LENLS 10)*, pages 143–156. Keio University, 2013. ISBN 978-4-915905-57-5.

42 Bruno Mery and Christian Retoré. Advances in the logical representation of lexical semantics. In Valeria de Paiva and Larry Moss, editors, *Natural Language and Computer Science (LICS 2013 satellite workshop)*, New-Orleans, 2013.

43 Bruno Mery and Christian Retoré. Semantic types, lexical sorts and classifiers. In B. Sharp and M. Zock, editors, *10th International Workshop on Natural Language Processing and Cognitive Science*, Marseilles, September 2013.

44 G. Mints. Zbl0381.03042: review of "cut elimination in a Gentzen-style ϵ-calculus without identity" by Linda Wessels (Z. math Logik Grundl. Math., 1977). *Zentralblatt Math.*

45 Grigori Mints. Cut elimination for a simple formulation of epsilon calculus. *Ann. Pure Appl. Logic*, 152(1-3):148–160, 2008.

46 Richard Montague. English as a formal language. In Bruno Visentini, editor, *Linguaggi nella Societa e nella Tecnica*, pages 189–224. Edizioni di Communità, Milan, Italy, 1970. (Reprinted in R. Thomason (ed) *The collected papers of Richard Montague* Yale University Press, 1974.).

47 Richard Moot. Automated extraction of type-logical supertags from the spoken dutch corpus. In Srinivas Bangalore and Aravind Joshi, editors, *The Complexity of Lexical Descriptions and its Relevance to Natural Language Processing: A Supertagging Approach*. MIT Press, 2007.

48 Richard Moot. Semi-automated extraction of a wide-coverage type-logical grammar for French. In *Proceedings of Traitement Automatique des Langues Naturelles (TALN)*, Montreal, 2010.

49 Richard Moot. Wide-coverage French syntax and semantics using Grail. In *Proceedings of Traitement Automatique des Langues Naturelles (TALN)*, Montreal, 2010.

50 Richard Moot, Laurent Prévot, and Christian Retoré. A discursive analysis of itineraries in an historical and regional corpus of travels. In *Constraints in discourse*, Ayay-roches-rouges, France, September 2011. `http://passage.inria.fr/cid2011/doku.php`.

51 Richard Moot, Laurent Prévot, and Christian Retoré. Un calcul de termes typés pour la pragmatique lexicale – chemins et voyageurs fictifs dans un corpus de récits de voyages. In *Traitement Automatique du Langage Naturel, TALN 2011*, pages 161–166, Montpellier, France, June 2011.

52 Richard Moot and Christian Retoré. Second order lambda calculus for meaning assembly: on the logical syntax of plurals. In Reinhard Muskens, editor, *Coconat: Conference on Computing Natural Reasoning*. University of Tilburg, December 2011. `http://hal.inria.fr/hal-00650644`.

53 Richard Moot and Christian Retoré. *The logic of categorial grammars: a deductive account of natural language syntax and semantics*, volume 6850 of *LNCS*. Springer, 2012.

54 Georg Moser and Richard Zach. The epsilon calculus and herbrand complexity. *Studia Logica*, 82(1):133–155, 2006.

55 Reinhard Muskens. Anaphora and the logic of change. In Jan van Eijck, editor, *JELIA*, volume 478 of *Lecture Notes in Computer Science*, pages 412–427. Springer, 1990.

56 Reinhard Muskens. Combining Montague Semantics and Discourse Representation. *Linguistics and Philosophy*, 19:143–186, 1996.

57 Geoffrey Nunberg. Transfers of meaning. *Journal of semantics*, 12(2):109–132, 1995.

58 Barbara Partee. Noun phrase interpretation and type shifting principles. In B.H. Partee and P.H. Portner, editors, *Formal Semantics: The Essential Readings*, pages 357–381. Wiley, 2008.

59 Stanley Peters and Dag Westerståhl. *Quantifiers in Language and Logic*. Clarendon Press, 2006.

60 Sylvain Pogodalla and Jean-Philippe Prost, editors. *Logical Aspects of Computational Linguistics – 6th International Conference, LACL 2011, Montpellier, France, June 29 to July 1, 2011. Proceedings*, volume 6736 of *LNCS*. Springer, 2011.

61 James Pustejovsky. The generative lexicon. *Computational Linguistics*, 17(4):409–441, 1991.

62 James Pustejovsky. *The generative lexicon*. M.I.T. Press, 1995.

63 Livy Real and Christian Retoré. Deverbal semantics and the Montagovian generative lexicon ΛTy_n. *Journal of Logic Language and Information*, 2014. 10.1007/s10849-014-9187-y.

64 Christian Retoré. Variable types for meaning assembly: a logical syntax for generic noun phrases introduced by "most". *Recherches Linguistiques de Vincennes*, 41:83–102, 2012.

65 Christian Retoré. A natural framework for natural language semantics: many sorted logic and Hilbert operators in type theory. In Mário Edmundo and Boban Velickovic, editors, *Logic colloquium*, Evora, 2013.

66 Christian Retoré. Sémantique des déterminants dans un cadre richement typé. In Emmanuel Morin and Yannick Estève, editors, *Traitement Automatique du Langage Naturel, TALN RECITAL 2013*, volume 1, pages 367–380. ACL Anthology, 2013.

67 Christian Retoré. Typed hilbert epsilon operators and the semantics of determiner phrases (invited lecture). In Glyn Morrill, Reinhard Muskens, Rainer Osswald, and Frank Richter, editors, *Proceedings of Formal Grammar 2014*, number 8612 in LNCS/FoLLI, pages 15–33. Springer, 2014. Invited lecture.

68 Bertrand Russell. On denoting. *Mind*, 56(14):479–493, 1905.

69 Sergei Soloviev and David Chemouil. Some Algebraic Structures in Lambda-Calculus with Inductive Types. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 2003.

70 Sergei Soloviev and Zhaohui Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 1-3(113):297–322, 2000.

71 Mark Steedman. *Taking Scope: The Natural Semantics of Quantifiers*. MIT Press, 2012.

72 Leonard Talmy. Fictive motion in language and "ception". In Paul Bloom, Mary A. Peterson, Lynn Nadel, and Merrill F. Garrett, editors, *Language and Space*, pages 211–276. MIT Press, 1999.

73 Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.

74 Luis von Ahn. Games with a purpose. *Computer*, 39(6):92–94, 2006.

75 Klaus von Heusinger. Definite descriptions and choice functions. In S. Akama, editor, *Logic, Language and Computation*, pages 61–91. Kluwer, 1997.

76 Klaus von Heusinger. Choice functions and the anaphoric semantics of definite nps. *Research on Language and Computation*, 2:309–329, 2004.

77 Tao Xue and Zhaohui Luo. Dot-types and their implementation. In Béchet and Dikovsky [9], pages 234–249.

78 Luke S. Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In Keh-Yih Su, Jian Su, and Janyce Wiebe, editors, *ACL/IJCNLP*, pages 976–984. The Association for Computer Linguistics, 2009.

A Certified Extension of the Krivine Machine for a Call-by-Name Higher-Order Imperative Language

Leonardo Rodríguez, Daniel Fridlender, and Miguel Pagano

Universidad Nacional de Córdoba, FaMAF
Córdoba, Argentina
{lrodrig2,fridlend,pagano}@famaf.unc.edu.ar

Abstract

In this paper we present a compiler that translates programs from an imperative higher-order language into a sequence of instructions for an abstract machine. We consider an extension of the Krivine machine for the call-by-name lambda calculus, which includes strict operators and imperative features. We show that the compiler is correct with respect to the big-step semantics of our language, both for convergent and divergent programs.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases abstract machines, compiler correctness, big-step semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.230

1 Introduction

Important advancements have been made during the last decade in the field of compiler certification; the project CompCert [19, 20] being the most significant achievement, since it deals with a realistic compiler for the C language. It is still an active research topic open for new techniques and experimentation. In this work we report our experiments in the use of known techniques to prove the correctness of a compiler for a call-by-name lambda calculus extended with strict operators and imperative features.

Most compilers are multi-pass, meaning that the process of translation usually involves successive symbolic manipulations from the source program to the target program. The compilation of a source program is often carried as a sequence of translations through several intermediate languages, each one closer to assembly code than the previous one. One common intermediate language consists of the instructions for some abstract machine; they are useful because they hide low-level details of concrete hardware, but also permit step-by-step execution of programs. At this level one can discover possible sources of optimization in the compilation.

Historically, several abstract machines have been developed and studied. Perhaps the best known ones are the SECD [15] machine and the CAM [8] machine, both for the call-by-value lambda calculus, and the Krivine machine [14] together with the G-machine [23], for call-by-name. We refer to Diehl *et al.* [11] for a, slightly dated, bibliographical review about different abstract machines. In this article we use the Krivine machine as the target of our compiler.

The Krivine machine has a very strong property: each transition rule of the machine corresponds directly to a reduction rule in the small-step semantics of the lambda calculus. This property is very useful to prove the correctness of the machine, since there is a relation of simulation between the machine and the calculus. This correspondence is, however, very difficult to maintain when one extends the source language, for example, by including

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 230–250

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

imperative features. Indeed, the conventional small step semantics of the calculus and the usual transitions of the machine might not correspond, as there can be transition sequences which do not necessarily simulate reduction steps in the source language. However, when the goal is to get a proof of correctness of the compiler, simulation is a property stronger than the one we actually need.

A different way of proving the correctness of a compiler is by using big-step semantics [22]; in this setting one proves that if a term t evaluates to a value v, then, when the machine executes the code obtained in the compilation of t, it must stop in a state related to v, for some appropriate definition of the relation between the value and the final state. One benefit of this approach is that it can be adapted to handle the correctness of divergent programs by a coinductive definition of the big-step semantics and of the transition relation of the machine.

In this paper we present a compiler which translates programs from an imperative higher-order language into a sequence of instructions for an abstract machine. We consider an extension of the Krivine machine for the call-by-name lambda calculus, which includes strict operators and imperative features. We show that the compiler is correct with respect to the big-step semantics of our language, both for convergent and divergent programs. We formalized the compiler and the proof of its correctness in the Coq proof assistant; the source code is available at `http://cs.famaf.unc.edu.ar/~leorodriguez/compilercorrectness/`.

Except for the absence of the type system, the programming language we consider in this paper has all of the features that Reynolds [26] described as the essence of Algol-like languages: deallocation is automatically done at the end of a block (stack discipline), only the imperative fragment of the language can have side effects, and it includes a call-by-name lambda calculus as a procedure mechanism. We consider this work to be a step towards proving the correctness for a compiler designed by Reynolds [25] for Algol-like languages.

The paper is organized as follows: in Sec. 2 we analyze the calculus of closures and the Krivine machine; and we revisit the proof of correctness of the compiler with respects to the small-step semantics of the calculus. In the following sections, we gradually extend the source language and the machine to cope with the extensions: first we add, in Sec. 3, strict operators and then, in Sec. 4, imperative features. We also prove the compiler correctness with respects to the big-step semantics of those languages.

2 Call-by-name lambda calculus

In this section we revisit the call-by-name lambda calculus, as a calculus of closures, and the Krivine abstract machine. In this simple setting we briefly explain the methodology used to prove the correctness of the compilation function. The proof exploits the fact that the transitions of the machine simulate the small-step semantics of the calculus.

2.1 Calculus of closures

Our high-level language is the lambda calculus with de Bruijn indices, but the operational semantics is given for an extension, proposed by Biernacka and Danvy [4], of Curien's calculus of closures [9]. This calculus is an early version of the lambda calculus with explicit substitutions. The main difference of this calculus with respect to the usual presentation of the lambda calculus is the way in which substitution are treated. In the latter, substitution is a meta-level operation, which in zero steps solves a β-redex. In contrast, in the calculus of closures substitutions are represented as terms equipped with environments – these pairs are

called closures – and new rules are added to perform the reduction of a simple redex. A closed lambda term t can be seen as a closure just by pairing it with the empty environment: $t\,[[]]$.

▶ **Definition 1** (Terms and closures).

Terms	$\Lambda \ni t,\ t'$	$::=$	$\lambda\, t$	Abstraction
		$\mid$	$t\ t'$	Application
		$\mid$	$\overline{n}$	Variables
Closures	$C \ni c,\ c'$	$::=$	$t\,[e] \mid c\ c'$	
Environments	$E \ni e$	$::=$	$[] \mid c :: e$	

Terms are the usual ones of the lambda calculus with de Bruijn indices representing variables. We will use some notational convention for meta-variables, besides those used in the grammar we use $n \in \mathbb{N}$ and the overline is just to see them as variables. In Curien's work only the first production for closures is present; the second one is Biernacka and Danvy's extension and allows to define a small-step operational semantics. An environment is a sequence of closures, and represents a substitution for all of the free variables of a term. The reduction rules for the calculus of closures are given by the following rewriting system. Notice that the side condition in (Var) can be removed if one sticks to closed closures.

▶ **Definition 2** (Reduction rules).

$$
\begin{array}{lrcll}
(\beta) & (\lambda\, t)\,[e]\ c & \to & t\,[c :: e] & \\
(\textsc{App}) & (t\ t')\,[e] & \to & t\,[e]\ t'\,[e] & \\
(\textsc{Var}) & \overline{n}\,[e] & \to & e.n & \text{if } n < |e|
\end{array}
\qquad (\nu)\quad \frac{c_1 \to c_1'}{c_1\ c_2 \to c_1'\ c_2}
$$

These reduction rules evaluate a closed term up to a weak head normal form; i.e. values are closures of the form $\lambda\, t\,[e]$. The (β) rule associates the argument of the redex with the first free variable of the body of the abstraction. The rule (App) just propagates the environment inside the term. Finally, the rule (Var) performs a lookup inside the environment, and reduces to the closure associated with the variable position. The (ν) rule allows the reduction of the left part of an application until getting an abstraction.

It can be easily proved that the semantics is deterministic:

▶ **Lemma 3** (Determinism). *If $c \to c_1$ and $c \to c_2$, then $c_1 = c_2$, for all closures c, c_1 and c_2.*

2.2 The Krivine machine

Now we turn to the target of our compiler: the Krivine abstract machine [14]. This machine has three components: the code, the environment, and the stack. We also have machine-level closures $\gamma = (i, \eta)$, which are pairs of code i together with an environment η.

▶ **Definition 4** (Abstract Machine). A configuration w is a triple $i \mid \eta \mid s$, where

Code:	$I \ni i, i'$	$::=$	$\mathsf{Grab} \triangleright i$
		$\mid$	$\mathsf{Push}\ i \triangleright i'$
		$\mid$	$\mathsf{Access}\ n$
Environments:	$H \ni \eta, \eta'$	$::=$	$[] \mid (i, \eta') :: \eta$
Stacks:	$S \ni s$	$::=$	$[] \mid (i, \eta) :: s$

The environment and the stack are just a list of machine-level closures. We use the following notation for lists: [] denotes the empty list, append is $\_ :: \_$ as in ML tradition;

the length of the list xs is $|xs|$ and if $n < |xs|$ then projecting the nth-element from xs is written $xs.n$. There are only three instructions, whose action is defined by the following three transition rules.

▶ **Definition 5** (Transition of the machine).

$$\begin{array}{rlll} \mathsf{Grab} \triangleright i \mid \eta \mid (i',\eta') :: s & \longmapsto & i \mid (i',\eta') :: \eta \mid s & \\ \mathsf{Push}\ i' \triangleright i \mid \eta \mid s & \longmapsto & i \mid \eta \mid (i',\eta) :: s & \\ \mathsf{Access}\ n \mid \eta \mid s & \longmapsto & i' \mid \eta' \mid s & \text{if } n < |\eta| \text{ and } \eta.n = (i',\eta') \end{array}$$

The instruction $\mathsf{Grab} \triangleright i$ takes a closure from the top of the stack, puts it in the environment, and then continues with the execution of i. The instruction $\mathsf{Push}\ i' \triangleright i$ pushes a closure (i', η) (where η is the current environment) in the top of the stack and continues with the execution of i. Finally, the instruction $\mathsf{Access}\ n$ starts executing the closure associated with the position n inside the environment.

2.3 Compilation and correctness

The next step is the definition of the translation of terms into code. The compiler function is denoted with $[\![\_]\!]$ and it is easily defined by induction on terms. We also define a decompilation function denoted $\{\!| \_ |\!\}$ which is clearly the inverse of the compilation. This function is useful to describe some properties of the machine which in turn are helpful to prove the correctness of the compiler.

▶ **Definition 6** (Compilation and decompilation of terms).

$$\begin{array}{rlrl} [\![\_]\!] : \Lambda \to I & & \{\!| \_ |\!\} : I \to \Lambda & \\ [\![\lambda\, t]\!] = \mathsf{Grab} \triangleright [\![t]\!] & & \{\!| \mathsf{Grab} \triangleright i |\!\} = \lambda\, \{\!| i |\!\} & \\ [\![t\ t']\!] = \mathsf{Push}\ [\![t']\!] \triangleright [\![t]\!] & & \{\!| \mathsf{Push}\ i' \triangleright i |\!\} = \{\!| i |\!\}\ \{\!| i' |\!\} & \\ [\![\overline{n}]\!] = \mathsf{Access}\ n & & \{\!| \mathsf{Access}\ n |\!\} = \overline{n} & \end{array}$$

We homomorphically extend the definition of the decompilation function to machine-level closures and environments:

▶ **Definition 7** (Decompilation of closures and environments).

$$\begin{array}{c} \{\!| \_ |\!\}^c : I \times H \to C \\ \{\!| (i,\eta) |\!\}^c = \{\!| i |\!\}\,[\{\!| \eta |\!\}^e] \end{array} \qquad \begin{array}{c} \{\!| \_ |\!\}^e : H \to E \\ \{\!| [] |\!\}^e = [] \\ \{\!| (i',\eta') :: \eta |\!\}^e = \{\!| (i',\eta') |\!\}^c :: \{\!| \eta |\!\}^e \end{array}$$

We also need to decompilate configurations of the machine into source-level closures. To decompile a configuration $(i \mid \eta \mid s)$ we successively apply the decompilation of the current closure (i, η) to the decompilation of every closure in s.

▶ **Definition 8** (Decompilation of configurations). Let $s = \gamma_1, \ldots, \gamma_n$, then

$$\{\!| (i \mid \eta \mid s) |\!\} = (\ldots (\{\!| (i,\eta) |\!\}^c\, \{\!| \gamma_1 |\!\}) \ldots \{\!| \gamma_n |\!\})$$

We now continue by presenting some well-known lemmas about the behaviour of the machine with respect to the small-step semantics of the calculus. First, we state that every transition of the machine simulates a reduction step in the calculus:

▶ **Lemma 9** (Simulation). *If* $w \longmapsto w'$, *then* $\{\!| w |\!\} \to \{\!| w' |\!\}$.

There is another useful property of the machine: if a configuration w decompiles to the closure c, and c can reduce, then the machine does not stop but makes a transition from w.

▶ **Lemma 10** (Progress). *If $\{\!|\, w \,|\!\} \to c'$, then there exists a configuration w' such that $w \longmapsto w'$.*

We can use Lemma 9 to obtain a stronger version of the previous lemma that better characterizes the configuration to which the machine makes the transition:

▶ **Lemma 11** (Progress and simulate). *If $\{\!|\, w \,|\!\} \to c'$, then there exists a configuration w' such that $w \longmapsto w'$ and $\{\!|\, w' \,|\!\} = c'$.*

Proof. By the progress lemma we state the existence of w', then by the simulation lemma we know that $\{\!|\, w \,|\!\} \to \{\!|\, w' \,|\!\}$ and then we conclude $\{\!|\, w' \,|\!\} = c'$ using the fact that the semantics is deterministic. ◀

The Lemma 11 can be easily extended to the reflexive-transitive closure of the small-step reduction and the machine transitions:

▶ **Lemma 12.** *If $\{\!|\, w \,|\!\} \to^* c'$, then there exists a configuration w' such that $w \longmapsto^* w'$ and $\{\!|\, w' \,|\!\} = c'$.*

We are particularly interested in the case in which the reduction sequence of the previous lemma reaches an irreducible closure c'. In this case, we expect the machine to stop in an irreducible configuration w' which decompiles to c'. We say that a configuration is irreducible if the machine can not perform any transition from it.

▶ **Lemma 13.** *If $\{\!|\, w \,|\!\} \to^* c'$ and c' is irreducible, then there exists a configuration w' such that $w \longmapsto^* w'$, $\{\!|\, w' \,|\!\} = c'$ and w' is irreducible.*

Proof. It is a consequence of the Lemma 12 and the simulation lemma. It is important to note that the proof of this lemma can be done constructively since the property of being irreducible is decidable. ◀

Now we can use these results to prove the correctness of our compiler. The following lemma states the correctness of the compilation of a closed term whose reduction sequence reaches an irreducible closure:

▶ **Lemma 14** (Correctness for convergent closed terms). *If $t\,[[]] \to^* c'$ and c' is irreducible, then there exists a configuration w' such that $([\![\, t \,]\!] \mid [] \mid []) \longmapsto^* w'$, $\{\!|\, w' \,|\!\} = c'$ and w' is irreducible.*

Proof. This lemma is an instance of Lemma 13 since we have $\{\!|\, ([\![\, t \,]\!] \mid [] \mid []) \,|\!\} = t\,[[]]$. ◀

If the reduction sequence of a closed term does not terminate (it does not reach an irreducible closure), then the execution of the compiled code must diverge. We can capture the notion of divergence for both reduction and execution with the following coinductive rules, the double line indicates that the rules are to be interpreted coinductively:

▶ **Definition 15** (Divergence of reduction and execution).

$$\frac{c \to c' \qquad c' \overset{\infty}{\to}}{c \overset{\infty}{\to}} \qquad \frac{w \longmapsto w' \qquad w' \longmapsto^{\infty}}{w \longmapsto^{\infty}}$$

The following lemma states that the divergence of the reduction sequence forces the machine to diverge:

▶ **Lemma 16** (Progress forever). *If* $\{\!|\, w \,|\!\} \to^\infty$*, then* $w \longmapsto^\infty$.

Proof. The proof is obtained by coinduction and using Lemma 11. ◀

Finally, the correctness of the compilation of divergent closed terms can be stated as follows:

▶ **Lemma 17** (Correctness for divergent closed terms). *If* $t\,[[]] \to^\infty$*, then* $(\llbracket t \rrbracket \mid [] \mid []) \longmapsto^\infty$.

In general, obtaining a proof of compiler correctness with respect to the small-step semantics of the source language is a very complicated task. In this section, we avoided some of those complications due to the simplicity of the language, for example, we did not have to define a bisimilarity relation as in [12, 27], but instead we used a decompilation *function*.

For more sophisticated languages, the big-step semantics leads often to simpler proofs of compiler correctness [22]. In the following sections, we use big-step semantics to prove the correctness of the compilation of two languages: a call-by-name lambda calculus with strict operators and an imperative higher-order language. We follow an approach inspired in the work of Leroy [22] (a proof of compiler correctness for the call-by-value lambda calculus).

3 Call-by-name lambda calculus with strict operators

In this section we extend the source language with constants and a strict binary operator; the language is specified by a big-step semantics. Then we present the abstract machine and the corresponding compiler. The correctness of the compiler for convergent terms is a ternary relation involving terms, their values, and the execution of the abstract machine. Leroy defined this relation by compiling values and proving that the execution of the compilation of a term leads to the compilation of the value. Following the same path for our language would impose an artificial set of transition rules for the machine; we avoid this by defining a binary relation between values and configurations.

3.1 The calculus

We now extend the source language with integer constants and the addition operator. Everything in this section can be straightforwardly extended to a language with several strict binary operators, but for the sake of concreteness we restrict our exposition to addition.

▶ **Definition 18** (Terms and closures).

Terms	$\Lambda \ni t,\ t'$	$::=$	$\lambda\, t$	Abstraction
		$\mid$	$t\ t'$	Application
		$\mid$	$\overline{n}$	Variables
		$\mid$	$\underline{k}$	Constants
		$\mid$	$t + t'$	Addition
Closures	$C \ni c$	$::=$	$t\,[e]$	
Environments	$E \ni e$	$::=$	$[] \mid c :: e$	
Values	$V \ni v$	$::=$	$(\lambda\, t)\,[e] \mid k$	

The new terms are constants $\underline{k}$, for $k \in \mathbb{N}$, and addition. Notice that there is no application of closures, this is a consequence of passing from small-step reductions to a big-step semantics, where intermediate computations steps cannot be observed. Values are the canonical forms which are the result of the evaluation of a term: an abstraction with its environment, and a constant. We define now the big-step semantics of the language. The evaluation of a term t in the environment e to the value v is denoted by $e \vdash t \Rightarrow v$.

▶ **Definition 19** (Big-step semantics)**.**

$$(\textsc{Abs})\ \frac{}{e \vdash \lambda\, t \Rightarrow (\lambda\, t)\,[e]} \qquad (\textsc{Const})\ \frac{}{e \vdash \underline{k} \Rightarrow k}$$

$$(\textsc{App})\ \frac{e \vdash t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash t \Rightarrow v}{e \vdash t_1\ t_2 \Rightarrow v} \qquad (\textsc{Var})\ \frac{e' \vdash t' \Rightarrow v}{e \vdash \overline{n} \Rightarrow v}\ e.n = t'\,[e']$$

$$(\textsc{Add})\ \frac{e \vdash t_1 \Rightarrow k \qquad e \vdash t_2 \Rightarrow k'}{e \vdash t_1\ +\ t_2 \Rightarrow k\ +\ k'}$$

The rules for abstractions and constants are trivial, since canonical forms evaluate to themselves. Notice that in the rule of the application the argument is not evaluated, but it is used to extend the environment during the evaluation of the body of the abstraction. In order to evaluate a variable one must do a lookup operation inside the environment, and start the evaluation of the corresponding closure. The rule for addition is quite conventional: one must first evaluate the two arguments and then obtain the final value by performing the addition of the two constants.

Now we show two simple examples of evaluation of terms:

▶ **Example 20.** A term that evaluates to an abstraction (partial application).

$$\frac{e \vdash (\lambda\,\lambda\, t) \Rightarrow (\lambda\,\lambda\, t)\,[e] \qquad t'\,[e] :: e \vdash \lambda\, t \Rightarrow (\lambda\, t)\,[t'\,[e] :: e]}{e \vdash (\lambda\,\lambda\, t)\ t' \Rightarrow (\lambda\, t)\,[t'\,[e] :: e]}$$

▶ **Example 21.** A term that evaluates to a constant.

$$\dfrac{\dfrac{}{e \vdash \lambda\,(\overline{0}\ +\ \underline{3}) \Rightarrow \lambda\,(\overline{0}\ +\ \underline{3})\,[e]} \qquad \dfrac{\dfrac{e \vdash \underline{2} \Rightarrow 2}{\underline{2}\,[e] :: e \vdash \overline{0} \Rightarrow 2} \qquad \dfrac{}{\underline{2}\,[e] :: e \vdash \underline{3} \Rightarrow 3}}{\underline{2}\,[e] :: e \vdash \overline{0}\ +\ \underline{3} \Rightarrow 5}}{e \vdash (\lambda\,(\overline{0}\ +\ \underline{3}))\ \underline{2} \Rightarrow 5}$$

3.2 A call-by-name machine with strict operations

The Krivine machine follows the call-by-name strategy, this implies that the argument of an application is evaluated only when it is needed. But if we want to incorporate some strict operation, like addition, we need a way to force the evaluation of the arguments before computing the operation. A known solution, cf. [28], to this issue is a data structure called *frame*, which is intended to store the code needed to compute the arguments along with the temporal values generated in the computation. The different components of the machine are defined as follows:

▶ **Definition 22** (Abstract machine).

$$
\begin{array}{lllll}
\text{Code:} & I \ni i, i' & ::= & \mathsf{Grab} \triangleright i \\
 & & \mid & \mathsf{Push}\ i \triangleright i' \\
 & & \mid & \mathsf{Access}\ n \\
 & & \mid & \mathsf{Const}\ k \\
 & & \mid & \mathsf{Add} \\
\text{Closures:} & \Gamma \ni \gamma & ::= & (i, \eta) \\
\text{Environments:} & H \ni \eta & ::= & [] \mid \gamma :: \eta \\
\text{Stack values:} & M \ni \mu & ::= & \gamma \mid [+ \bullet \gamma] \mid [+\ k\ \bullet] \\
\text{Stacks:} & S \ni s & ::= & [] \mid \mu :: s \\
\text{Configurations:} & W \ni w & ::= & (\gamma,\ s)
\end{array}
$$

As in the previous section, a closure is composed by a code together with its environment. The environment is a list of closures and a stack is a list of *stack values* which may be closures or frames. The frame $[+ \bullet \gamma]$ stores the code needed to compute the second argument of the addition, this closure remains stored in the stack while the first argument is being computed. On the other hand, the frame $[+\ k\ \bullet]$ stores the computed value of the first argument while the second argument is being computed. In the next section we generalize frames to support n-ary operations.

The following are the transitions of the machine; they are the same from the previous section and the new rules for operators and constants.

▶ **Definition 23** (Machine transitions).

$$
\begin{array}{rlll}
(\mathsf{Grab} \triangleright i, \eta) \mid \gamma :: s & \longmapsto (i, \gamma :: \eta) & \mid s & \\
(\mathsf{Push}\ i \triangleright i', \eta) \mid s & \longmapsto (i', \eta) & \mid (i, \eta) :: s & \\
(\mathsf{Access}\ n, \eta) \mid s & \longmapsto \eta.n & \mid s & \text{if } n < |\eta| \\
(\mathsf{Add}, \eta) \mid \gamma_1 :: \gamma_2 :: s & \longmapsto \gamma_1 & \mid [+ \bullet \gamma_2] :: s & \\
(\mathsf{Const}\ k, \eta) \mid [+ \bullet \gamma] :: s & \longmapsto \gamma & \mid [+\ k\ \bullet] :: s & \\
(\mathsf{Const}\ k, \eta) \mid [+\ k'\ \bullet] :: s & \longmapsto (\mathsf{Const}\ (k + k'), \eta) & \mid s &
\end{array}
$$

The instruction Add expects in the top of the stack one closure for each of the arguments of the addition. It pushes in the stack a new frame with the code of the second argument, and starts executing the code of the first one. For the case of the instruction Const k there are two transition rules, arising from two scenarios: k is the value of the first argument of an addition, and k is the value of the second argument. In the first case, it executes the code γ stored in the frame, and updates the frame with the constant k. In the second case, we can take the value of the first argument k' from the frame and execute Const $(k + k')$.

3.3 Compilation and its correctness

The compiler is defined by induction on the structure of the term, it maps source terms into a sequence of machine instructions:

▶ **Definition 24** (Compilation of terms)**.**

$$\begin{aligned}
[\![\,]\!] &: \Lambda \to I \\
[\![\lambda\, t]\!] &= \mathsf{Grab} \triangleright [\![t]\!] \\
[\![t\ t']\!] &= \mathsf{Push}\ [\![t']\!] \triangleright [\![t]\!] \\
[\![\overline{n}]\!] &= \mathsf{Access}\ n \\
[\![\underline{k}]\!] &= \mathsf{Const}\ k \\
[\![t_1 + t_2]\!] &= \mathsf{Push}\ [\![t_2]\!] \triangleright (\mathsf{Push}\ [\![t_1]\!] \triangleright \mathsf{Add})
\end{aligned}$$

The compilation of a term $\underline{k}$ is just the instruction $\mathsf{Const}\ k$. The code for an addition starts with a Push instruction for each argument, and continues with the Add instruction. By the time when the instruction Add is executed, the code for each argument is already stored in the stack, ready to be inserted inside a frame. We now extend the definition of the compiler for closures and environments:

▶ **Definition 25** (Compilation of closures and environments)**.**

$$\begin{aligned}
[\![_]\!]^{\mathrm{C}} &: C \to \Gamma & [\![_]\!]^{\mathrm{E}} &: E \to H \\
[\![t\,[e]]\!]^{\mathrm{C}} &= ([\![t]\!], [\![e]\!]^{\mathrm{E}}) & [\![[]]\!]^{\mathrm{E}} &= [] \\
& & [\![c :: e]\!]^{\mathrm{E}} &= [\![c]\!]^{\mathrm{C}} :: [\![e]\!]^{\mathrm{E}}
\end{aligned}$$

Here the functions $[\![\_]\!]^{\mathrm{C}}$ and $[\![\_]\!]^{\mathrm{E}}$ are mutually recursive. The compilation of a source-level closure is a machine-level closure which couples the code of the term and the code of its environment. On the other hand, the compilation of an environment is obtained by compiling each closure inside it.

In order to illustrate how the machine works, we take the same terms of the above examples and show the step-by-step execution of the corresponding code:

▶ **Example 26.** Execution of the code $[\![(\lambda\,\lambda\, t)\ t']\!]$.

$$[\![(\lambda\,\lambda\, t)\ t']\!] = \mathsf{Push}\ [\![t']\!] \triangleright \mathsf{Grab} \triangleright \mathsf{Grab} \triangleright [\![t]\!]$$

$$\begin{aligned}
&(\mathsf{Push}\ [\![t']\!] \triangleright \mathsf{Grab} \triangleright \mathsf{Grab} \triangleright [\![t]\!], \eta) \mid s \\
&\longmapsto (\mathsf{Grab} \triangleright \mathsf{Grab} \triangleright [\![t]\!], \eta) \mid ([\![t']\!], \eta) :: s \\
&\longmapsto (\mathsf{Grab} \triangleright [\![t]\!], ([\![t']\!], \eta) :: \eta) \mid s
\end{aligned}$$

▶ **Example 27.** Execution of the code $[\![(\lambda\,(\overline{0} + \underline{3}))\ \underline{2}]\!]$.

$$[\![(\lambda\,(\overline{0} + \underline{3}))\ \underline{2}]\!] = \mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}$$

$$\begin{aligned}
&(\mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, \eta) \mid s \\
&\longmapsto (\mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, \eta) \mid (\mathsf{Const}\ 2, \eta) :: s \\
&\longmapsto (\mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, (\mathsf{Const}\ 2, \eta) :: \eta) \mid s \\
&\longmapsto (\mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, \eta') \mid (\mathsf{Const}\ 3, \eta') :: s \quad \text{where } \eta' = (\mathsf{Const}\ 2, \eta) :: \eta \\
&\longmapsto (\mathsf{Add}, \eta') \mid (\mathsf{Access}\ 0, \eta') :: (\mathsf{Const}\ 3, \eta') :: s \\
&\longmapsto (\mathsf{Access}\ 0, \eta') \mid [+ \bullet\ (\mathsf{Const}\ 3, \eta')] :: s \\
&\longmapsto (\mathsf{Const}\ 2, \eta) \mid [+ \bullet\ (\mathsf{Const}\ 3, \eta')] :: s \\
&\longmapsto (\mathsf{Const}\ 3, \eta') \mid [+\ 2\ \bullet] :: s \\
&\longmapsto (\mathsf{Const}\ 5, \eta') \mid s
\end{aligned}$$

Example 27 is, in fact, an instance of a more general behaviour of the machine: if a term t evaluates to a constant k in an environment e, then, the execution of the code $[\![t]\!]$ in the environment $[\![e]\!]^{\mathrm{E}}$ and an initial stack s leads to the configuration $(\mathsf{Const}\ k, \eta') \mid s$ for some environment η'. In a similar way, Example 26 can be generalized as follows: if a term t evaluates to a closure $(\lambda\, t')\,[e']$ in an environment e, then, the execution of $[\![t]\!]$ in the environment $[\![e]\!]^{\mathrm{E}}$ leads to the configuration $(\mathsf{Grab} \triangleright [\![t']\!], [\![e']\!]^{\mathrm{E}}) \mid s$. These facts can be taken as evidence of the correctness of the compiler:

▶ **Theorem 28** (Compiler Correctness). *For any $e \in E$, $t \in \Lambda$ and $v \in V$, if $e \vdash t \Rightarrow v$ then for all $s \in S$,*

- *if $v = k$ for some constant k, then $[\![t\,[e]]\!]^{\mathrm{C}} \mid s \longmapsto^* (\mathsf{Const}\ k, \eta') \mid s$ for some $\eta' \in H$,*
- *if $v = (\lambda\, t')\,[e']$ for some $t' \in \Lambda$ and $e' \in E$, then $[\![t\,[e]]\!]^{\mathrm{C}} \mid s \longmapsto^* (\mathsf{Grab} \triangleright [\![t']\!], [\![e]\!]) \mid s$.*

The statement of this theorem can significantly shortened by defining the relation $\rightarrowtail \subseteq W \times V$:

▶ **Definition 29.**

$$\begin{array}{lll} \gamma \mid s \rightarrowtail k & \text{iff} & \gamma \mid s \longmapsto^* (\mathsf{Const}\ k, \eta') \mid s \text{ for some } \eta' \in H \\ \gamma \mid s \rightarrowtail (\lambda\, t)\,[e] & \text{iff} & \gamma \mid s \longmapsto^* (\mathsf{Grab} \triangleright [\![t]\!], [\![e]\!]^{\mathrm{E}}) \mid s. \end{array}$$

The following property about this relation is expected and self-evident:

▶ **Lemma 30.** *For any $\gamma, \gamma' \in \Gamma$, $s \in S$ and $v \in V$, if $\gamma \mid s \longmapsto^* \gamma' \mid s$ and $\gamma' \mid s \rightarrowtail v$, then $\gamma \mid s \rightarrowtail v$.*

The relation $\rightarrowtail$ leads to simpler proofs (both in paper and in Coq) and can be generalized in the presence of more values, keeping the statement of correctness unchanged. Theorem 28 can be stated as follows:

▶ **Theorem 31** (Compiler Correctness). *For any $e \in E$, $t \in \Lambda$ and $v \in V$, if $e \vdash t \Rightarrow v$ then for all $s \in S$, $[\![t\,[e]]\!]^{\mathrm{C}} \mid s \rightarrowtail v$.*

Proof. This theorem can be proved by induction on the derivation of $e \vdash t \Rightarrow v$. We illustrate the proof with two cases: for rules (Const) and (App).

In the case of the rule $e \vdash \underline{k} \Rightarrow k$, we have

$$[\![\underline{k}\,[e]]\!]^{\mathrm{C}} \mid s = (\mathsf{Const}\ k, [\![e]\!]^{\mathrm{E}}) \mid s \longmapsto^* (\mathsf{Const}\ k, [\![e]\!]^{\mathrm{E}}) \mid s\ ,$$

for any $s \in S$, and therefore $[\![\underline{k}\,[e]]\!]^{\mathrm{C}} \mid s \rightarrowtail k$.

Now we turn to application; let us recall the rule (App):

$$(\textsc{App})\ \frac{e \vdash t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash t \Rightarrow v}{e \vdash t_1\ t_2 \Rightarrow v}$$

We have one inductive hypothesis for each premise in the rule. In this case we have:

(i) for all $s' \in S$, $[\![t_1\,[e]]\!]^{\mathrm{C}} \mid s' \rightarrowtail (\lambda\, t)\,[e']$
(ii) for all $s' \in S$, $[\![t\,[t_2\,[e] :: e']]\!]^{\mathrm{C}} \mid s' \rightarrowtail v$.

Thus, by definition of $\rightarrowtail$ and (i), we get:

(iii) for all $s' \in S$, $[\![t_1\,[e]]\!]^{\mathrm{C}} \mid s' \longmapsto^* (\mathsf{Grab} \triangleright [\![t]\!], [\![e']\!]^{\mathrm{E}}) \mid s'$.

Using Lemma 30, we can now start with the configuration $[\![\, t_1\ t_2\,[e]\,]\!]^{\mathrm{C}} \mid s$ and try to reach the configuration $[\![\, t\,[t_2\,[e]::e']\,]\!]^{\mathrm{C}} \mid s$, which we know by (ii) is related with v by the $\rightarrowtail$ relation:

$$
\begin{array}{rcll}
[\![\, t_1\ t_2\,[e]\,]\!]^{\mathrm{C}} \mid s & = & ([\![\, t_1\ t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
& = & (\mathsf{Push}\ [\![\, t_2\,]\!] \triangleright [\![\, t_1\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!] \\
& \longmapsto & ([\![\, t_1\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by the } \mathsf{Push} \text{ rule} \\
& = & [\![\, t_1\,[e]\,]\!]^{\mathrm{C}} \mid ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
& \longmapsto^* & (\mathsf{Grab} \triangleright [\![\, t\,]\!], [\![\, e'\,]\!]^{\mathrm{E}}) \mid ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by (iii)} \\
& \longmapsto & ([\![\, t\,]\!], ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: [\![\, e'\,]\!]^{\mathrm{E}}) \mid s & \text{by the } \mathsf{Grab} \text{ rule} \\
& = & ([\![\, t\,]\!], [\![\, t_2\,[e]\,]\!]^{\mathrm{C}} :: [\![\, e'\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
& = & ([\![\, t\,]\!], [\![\, t_2\,[e]::e'\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{E}} \\
& = & [\![\, t\,[t_2\,[e]::e']\,]\!]^{\mathrm{C}} \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}}.
\end{array}
$$

And that finishes the proof for (App). The remaining cases are similar. ◀

There is a third way to state the theorem of correctness that is a bit more intuitive and closer as how Leroy [22] stated it: one defines a compilation for values and then proves that the compilation of a term executes to the compilation of its value.

▶ **Definition 32** (Compilation of values)**.**

$$
\begin{aligned}
[\![\_]\!]^{\mathrm{V}} &: V \to \Gamma \\
[\![\,(\lambda\, t)\,[e]\,]\!]^{\mathrm{V}} &= (\mathsf{Grab} \triangleright [\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \\
[\![\, k\,]\!]^{\mathrm{V}} &= (\mathsf{Const}\ k, [])
\end{aligned}
$$

The alternative version of the correctness theorem can be formally stated as follows:

▶ **Theorem 33** (Compiler Correctness, alternative)**.** *For any $e \in E$, $t \in \Lambda$ and $v \in V$, if $e \vdash t \Rightarrow v$, then, for all $s \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid s \longmapsto^* [\![\, v\,]\!]^{\mathrm{V}} \mid s$.*

The proof also proceeds by induction on derivations of the evaluation. Notice however that the compilation of a constant value pairs the constant with the empty environment, but the compilation of the constant (as a term) is paired with the compilation of the corresponding environment. So one needs to add a rule to discard the environment:

$$(\mathsf{Const}\ k, \gamma :: \eta) \mid s \longmapsto (\mathsf{Const}\ k, []) \mid s$$

But this change introduces some non-determinism in the machine, so one is forced to change the two rules for constants in Def. 23: those would require the environment to be empty. At first sight, it could seem possible to avoid this issue by generalizing the function $[\![\_]\!]^{\mathrm{V}}$ by taking an extra argument for the top-level environment; however, this also fails in the proof of the theorem for the case (Var).

3.3.1 Correctness for divergent terms

To complete the proof of correctness of the compiler, we need to ensure that, when the evaluation of a term t diverges, the execution of $[\![\, t\,]\!]$ will never terminate. We use the approach proposed by Leroy [22] of defining a coinductive big-step semantics to capture the notion of divergence of a term. We write $e \vdash t \Rightarrow \infty$ to denote the divergence of a term t in an environment e. The following are the rules of divergence for the high-level language of this section:

▶ **Definition 34** (Coinductive semantics).

$$(\textsc{App1})\ \frac{e \vdash t_1 \Rightarrow \infty}{e \vdash t_1\ t_2 \Rightarrow \infty} \qquad (\textsc{App2})\ \frac{e \vdash t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash t \Rightarrow \infty}{e \vdash t_1\ t_2 \Rightarrow \infty}$$

$$(\textsc{Var})\ \frac{e' \vdash t' \Rightarrow \infty}{e \vdash \overline{n} \Rightarrow \infty}\ e.n = t'\,[e']$$

$$(\textsc{Add1})\ \frac{e \vdash t_1 \Rightarrow \infty}{e \vdash t_1\ +\ t_2 \Rightarrow \infty} \qquad (\textsc{Add2})\ \frac{e \vdash t_1 \Rightarrow k \qquad e \vdash t_2 \Rightarrow \infty}{e \vdash t_1\ +\ t_2 \Rightarrow \infty}$$

There are two possible reasons for an application $(t_1\ t_2)$ to diverge. The first possibility is that the function term t_1 diverges. The second one is that, when the term t_1 evaluates to an abstraction $(\lambda\, t)\,[e']$, then the evaluation of the body t diverges. Note that, since we are in a call-by-name setting, we do not make any claim about the evaluation of the argument t_2.

The next step is to capture divergence in the abstract machine; again, we use coinductive semantics. The following rule captures the notion of an infinite sequence of machine transitions:

▶ **Definition 35** (Divergence of execution).

$$\frac{w \longmapsto w' \qquad w' \longmapsto^{\infty}}{w \longmapsto^{\infty}}$$

Now we are able to state the following lemma that establishes that if a term t diverges, then the machine makes infinitely many transitions.

▶ **Theorem 36** (Correctness for divergent programs). *If* $e \vdash t \Rightarrow \infty$, *then* $[\![\, t\,[e] \,]\!]^{\mathrm{c}} \mid s \longmapsto^{\infty}$.

4 Higher-order imperative language

In this section we extend the language of the previous section with imperative features, namely we add the possibility to allocate, modify, and access memory locations. We adapt the abstract machine to reflect this extension of the source language and prove the correctness of the compiler with respect to a big-step semantics. For simplicity, the imperative variables of our language can only contain integer values, and we only consider memory locations storing integers.

4.1 The language

The imperative fragment of the language includes locations (natural numbers representing positions in the store), a dereferencing operator, variable declarations, composition and assignments. As in the previous section, we use de Bruijn indices to represent variables.

▶ **Definition 37** (Higher-order imperative language).

Terms	$\Lambda \ni t,\ t'$	$::=$	$\lambda\, t$	Abstraction
		$\mid$	$t\ t'$	Application
		$\mid$	$\overline{n}$	Variables
		$\mid$	$\underline{k}$	Constants
		$\mid$	$t \oplus t'$	Binary operators
		$\mid$	$\underline{\ell}$	Locations
		$\mid$	$!\, t$	Dereferencing
		$\mid$	**newvar** t	Variable declaration
		$\mid$	$t\,;\, t'$	Composition
		$\mid$	$t := t'$	Assignments
		$\mid$	**skip**	Skip command

Closures	$C \ni c$	$::=$	$t\,[e]$
Environments	$E \ni e$	$::=$	$[] \mid c :: e$
Stores	$\Sigma \ni \sigma$	$::=$	$[] \mid \sigma \triangleright k$
Values	$V \ni v$	$::=$	$(\lambda\, t)\,[e] \mid k \mid \ell \mid \sigma$

The initial configurations of the big-step semantics are triples (e, t, σ) and final configurations are values; we write $e \vdash^{\sigma} t \Rightarrow v$ to denote that (e, t, σ) evaluates to v. Of course, the values of commands are stores; moreover, since only commands can have side effects, the rules of the big-step semantics of the language of the previous sections remain unchanged, except that they propagate the state to each premise.

▶ **Definition 38** (Big-step semantics).

$$\text{(Abs)}\ \frac{}{e \vdash^{\sigma} \lambda\, t \Rightarrow (\lambda\, t)\,[e]} \qquad \text{(Var)}\ \frac{e' \vdash^{\sigma} t' \Rightarrow v}{e \vdash^{\sigma} \overline{n} \Rightarrow v}\ e.n = t'\,[e']$$

$$\text{(App)}\ \frac{e \vdash^{\sigma} t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash^{\sigma} t \Rightarrow v}{e \vdash^{\sigma} t_1\ t_2 \Rightarrow v}$$

$$\text{(Const)}\ \frac{}{e \vdash^{\sigma} \underline{k} \Rightarrow k} \qquad \text{(Bop)}\ \frac{e \vdash^{\sigma} t_1 \Rightarrow k \qquad e \vdash^{\sigma} t_2 \Rightarrow k'}{e \vdash^{\sigma} t_1 \oplus t_2 \Rightarrow k \oplus k'}$$

$$\text{(Loc)}\ \frac{}{e \vdash^{\sigma} \underline{\ell} \Rightarrow \ell}\ \ell < |\sigma| \qquad \text{(Deref)}\ \frac{e \vdash^{\sigma} t \Rightarrow \ell}{e \vdash^{\sigma} !\, t \Rightarrow \sigma(\ell)}$$

$$\text{(Skip)}\ \frac{}{e \vdash^{\sigma} \textbf{skip} \Rightarrow \sigma} \qquad \text{(Newvar)}\ \frac{\underline{\ell}\,[e] :: e \vdash^{\sigma \triangleright 0} t \Rightarrow \sigma' \triangleright k}{e \vdash^{\sigma} \textbf{newvar}\ t \Rightarrow \sigma'}\ \ell = |\sigma|$$

$$\text{(Assign)}\ \frac{e \vdash^{\sigma} t \Rightarrow \ell \qquad e \vdash^{\sigma} t' \Rightarrow k}{e \vdash^{\sigma} t := t' \Rightarrow \sigma[\ell \mapsto k]} \qquad \text{(Comp)}\ \frac{e \vdash^{\sigma} t \Rightarrow \sigma' \qquad e \vdash^{\sigma'} t' \Rightarrow \sigma''}{e \vdash^{\sigma} t\,;\, t' \Rightarrow \sigma''}$$

Here we use some conventional notations to access and modify stores. We consider locations to be a position (an index) of the store. Thus, the store $\sigma[\ell \mapsto k]$ contains the same values as σ except perhaps in the position ℓ, in which the value is k. We denote by $\sigma(\ell)$ the

constant allocated in the position ℓ of the store. Extension of the store σ with value k in the new location is written $\sigma \triangleright k$.

In the rule of the term **newvar** we observe that, in order to evaluate the inner command t, we need to extend the store and the environment. The store is extended with the value 0, which is the default value we chose for newly created locations. The environment is extended with the location $\ell = |\sigma|$ which points to the new (and last) position of the extended store.

It is worth noting that, since the access to a location could be done exclusively through the use of a variable bound by **newvar**, we can "hide" to the user the existence of explicit locations as terms. In other words, the user does not need to know that he can write explicit locations, since all of them are created by **newvar** and bound to variables. This kind of explicit locations have been used before, for example in [13, page 3].

Another observation to make is that it is impossible for a location created by **newvar** to leave its lexical scope. In the assignment command $t_1 := t_2$ the term t_2 must evaluate to a constant, and not to a location. The store is also restricted to contain integer constants only.

A distinct feature of Algol-like languages is that the execution of a command should not leave inaccessible locations in the store; as the following lemmas show, our semantics respects that condition.

▶ **Lemma 39** (Store size preservation). *If* $e \vdash^{\sigma} t \Rightarrow \sigma'$ *then* $|\sigma| = |\sigma'|$.

▶ **Lemma 40** (Safe locations). *If* $e \vdash^{\sigma} t \Rightarrow \ell$ *then* $\ell < |\sigma|$.

4.2 A Krivine abstract machine with store

In order to cope with the extensions in the source language we need to make some changes to the abstract machine of Sec. 3. We generalize the treatment of the binary operator of the previous section so as to capture at once addition, dereferencing, and assignment; in order to do so, some instructions carry the arity of the operator. Besides that generalization, we need two instructions for allocating and deallocating memory cells; and yet another one to signal the end of the execution of the current command.

▶ **Definition 41** (Abstract machine).

$$
\begin{array}{lrcl}
\text{Code:} & I \ni i, i' & ::= & \mathsf{Grab} \triangleright i \\
& & \mid & \mathsf{Push}\ i \triangleright i' \\
& & \mid & \mathsf{Access}\ n \\
& & \mid & \mathsf{Const}\ V \\
& & \mid & \mathsf{Op}\ \ominus^{\mathsf{n}} \\
& & \mid & \mathsf{Frame}\ \ominus^{\mathsf{n}} \\
& & \mid & \mathsf{Alloc} \triangleright i \\
& & \mid & \mathsf{Dealloc} \\
& & \mid & \mathsf{Cont}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Closures:} & \Gamma \ni \gamma & ::= & (i, \eta) \\
\text{Environments:} & H \ni \eta & ::= & [] \mid \gamma :: \eta \\
\text{Operators:} & \mathit{Ops} \ni \ominus^{\mathsf{n}} & ::= & \oplus^2 \mid !^1 \mid :=^2 \\
\text{Operator Arguments:} & N \ni \nu & ::= & k \mid \ell \\
\text{Stack values:} & M \ni \mu & ::= & \gamma \mid [\ominus^{\mathsf{n}}\ \overline{\nu} \bullet \overline{\gamma}] \\
\text{Stacks:} & S \ni s & ::= & [] \mid \mu :: s \\
\text{Stores:} & \Sigma \ni \sigma & ::= & [] \mid \sigma \triangleright k \\
\text{Configurations:} & W \ni w & ::= & (\gamma,\ \sigma,\ s)
\end{array}
$$

Here, a frame $[\ominus^{\mathsf{n}}\ \overline{\nu}\ \bullet\ \overline{\gamma}]$ is a data structure that contains: (a) an operator $\ominus^{\mathsf{n}}$, which is always associated with an operation supported by the machine, (b) a list $\overline{\nu}$ with the arguments of the operation which have been already computed, and (c) a list $\overline{\gamma}$ with the code required to compute the rest of the arguments.

The transitions of the machine are given in the following definition. Notice that the execution of code corresponding to expressions will, eventually, finish with a numeric value in the closure part; while the execution of an imperative command will finish with Cont.

▶ **Definition 42.**

$$
\begin{array}{llllll}
(\mathsf{Grab} \triangleright i, \eta) & \mid \sigma \mid \gamma :: s & \longmapsto & (i, \gamma :: \eta) \mid \sigma \mid s & & \\
(\mathsf{Push}\ i \triangleright i', \eta) & \mid \sigma \mid s & \longmapsto & (i', \eta) \mid \sigma \mid (i, \eta) :: s & & \\
(\mathsf{Access}\ n, \eta) & \mid \sigma \mid s & \longmapsto & \eta.n \mid \sigma \mid s & & \text{if } n < |\eta| \\
(\mathsf{Frame}\ \ominus^{\mathsf{n}}, \eta) & \mid \sigma \mid \gamma_1 :: \overline{\gamma} :: s & \longmapsto & \gamma_1 \mid \sigma \mid [\ominus^{\mathsf{n}}\ \bullet\ \overline{\gamma}] :: s & & \text{if } |\overline{\gamma}| < n \\
(\mathsf{Op} \oplus, \eta) & \mid \sigma \mid [\oplus\ k, k' \bullet] :: s & \longmapsto & (\mathsf{Const}\ \hat{k}, \eta) \mid \sigma \mid s & & \text{where } \hat{k} = k \oplus k' \\
(\mathsf{Op}\ :=, \eta) & \mid \sigma \mid [:=\ \ell, k \bullet] :: s & \longmapsto & (\mathsf{Cont}, \eta) \mid \sigma' \mid s & & \text{where } \sigma' = \sigma[\ell \mapsto k] \\
(\mathsf{Op}\ !, \eta) & \mid \sigma \mid [!\ \ell \bullet] :: s & \longmapsto & (\mathsf{Const}\ k, \eta) \mid \sigma \mid s & & \text{where } k = \sigma(\ell) \\
(\mathsf{Alloc} \triangleright i, \eta) & \mid \sigma \mid s & \longmapsto & (i, \gamma :: \eta) \mid \sigma \triangleright 0 \mid s & & \text{where } \gamma = (\mathsf{Const}\ |\sigma|, \eta) \\
(\mathsf{Dealloc}, \eta) & \mid \sigma \triangleright k \mid s & \longmapsto & (\mathsf{Cont}, \eta) \mid \sigma \mid s & & \\
(\mathsf{Cont}, \eta) & \mid \sigma \mid \gamma :: s & \longmapsto & \gamma \mid \sigma \mid s & & \\
(\mathsf{Const}\ \nu, \eta) & \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu}\ \bullet\ \gamma_1, \overline{\gamma}] :: s & \longmapsto & \gamma_1 \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu}, \nu\ \bullet\ \overline{\gamma}] :: s & & \\
(\mathsf{Const}\ \nu, \eta) & \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu} \bullet] :: s & \longmapsto & (\mathsf{Op}\ \ominus^{\mathsf{n}}, \eta) \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu}, \nu \bullet] :: s & &
\end{array}
$$

The instruction $\mathsf{Frame}\ \ominus^{\mathsf{n}}$ expects n closures in the top of the stack. Then it executes the code of the first argument, and creates a frame containing the rest of them.

As in the previous section, the instruction $\mathsf{Const}\ k$ updates the frame with the constant k – which is the value of an argument – and executes the next closure stored in the frame, if there is any. When all the arguments have been computed, the instruction $\mathsf{Op}\ \ominus^{\mathsf{n}}$ is executed. This instruction expects a frame with all the arguments computed and applies the built-in operation associated with $\ominus^{\mathsf{n}}$. For example, if $\ominus^{\mathsf{n}}$ is the assignment operator $(:=)$, then $\mathsf{Op}\ (:=)$ expects a frame $[:=\ \ell, k \bullet]$ and then updates the store in the location ℓ with the constant value k.

4.3 Compilation and correctness

The compilation of the applicative part of the language remains unchanged with respect to the previous section. The translation of an n-ary operator $\ominus$ consists in compiling all its operands and putting the instruction $\mathsf{Frame}\ \ominus^{\mathsf{n}}$ after their code. Notice, however, that the code for the operands will be executed after constructing the appropriate frame in the stack. To compile the allocation of a new variable, we prepare the deallocation of the new location –to be executed after the body of the block–, then we generate an allocation instruction followed by the code of the body.

▶ **Definition 43** (Compilation of terms)**.**

$$
\begin{array}{rl}
[\![\_]\!] \colon \Lambda \to I & [\![t_1 \oplus t_2]\!] = \mathsf{Push}\ [\![t_2]\!] \triangleright \mathsf{Push}\ [\![t_1]\!] \triangleright \mathsf{Frame}\ (\oplus) \\
[\![\lambda\, t]\!] = \mathsf{Grab} \triangleright [\![t]\!] & [\![!\, t]\!] = \mathsf{Push}\ [\![t]\!] \triangleright \mathsf{Frame}\ (!) \\
[\![t\ t']\!] = \mathsf{Push}\ [\![t']\!] \triangleright [\![t]\!] & [\![\mathbf{newvar}\ t]\!] = \mathsf{Push}\ (\mathsf{Dealloc}) \triangleright \mathsf{Alloc} \triangleright [\![t]\!] \\
[\![\overline{n}]\!] = \mathsf{Access}\ n & [\![t_1\,;\, t_2]\!] = \mathsf{Push}\ [\![t_2]\!] \triangleright [\![t_1]\!] \\
[\![\underline{k}]\!] = \mathsf{Const}\ k & [\![t_1 := t_2]\!] = \mathsf{Push}\ [\![t_2]\!] \triangleright \mathsf{Push}\ [\![t_1]\!] \triangleright \mathsf{Frame}\ (:=) \\
[\![\underline{\ell}]\!] = \mathsf{Const}\ \ell & [\![\mathbf{skip}]\!] = \mathsf{Cont}
\end{array}
$$

The following is the definition of compilation functions for closures and environments. Note that these functions are mutually recursive:

▶ **Definition 44** (Compilation of closures and environments)**.**

$$\begin{array}{ll} [\![\_]\!]^{\mathrm{C}} \colon C \to \Gamma & [\![\_]\!]^{\mathrm{E}} \colon E \to H \\ [\![\, t\,[e]\,]\!]^{\mathrm{C}} = ([\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) & [\![\,[]\,]\!]^{\mathrm{E}} = [] \\ & [\![\, c :: e\,]\!]^{\mathrm{E}} = [\![\, c\,]\!]^{\mathrm{C}} :: [\![\, e\,]\!]^{\mathrm{E}} \end{array}$$

As in the previous section, we use a relation $\rightarrowtail$ between configurations and values to state the correctness theorem. We extend Definition 29 as follows:

▶ **Definition 45** ($\rightarrowtail \subseteq W \times V$)**.**

$$\begin{array}{lll} \gamma \mid \sigma \mid s \rightarrowtail k & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Const}\ k, \eta') \mid \sigma \mid s \ \text{ for some } \eta' \in H \\ \gamma \mid \sigma \mid s \rightarrowtail (\lambda\, t)\,[e] & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Grab} \triangleright [\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid s \\ \gamma \mid \sigma \mid s \rightarrowtail \ell & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Const}\ \ell, \eta') \mid \sigma \mid s \ \text{ for some } \eta' \in H \\ \gamma \mid \sigma \mid s \rightarrowtail \sigma' & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Cont}, \eta') \mid \sigma' \mid s \ \text{ for some } \eta' \in H. \end{array}$$

Now we can state the theorem of correctness for convergent terms:

▶ **Theorem 46** (Correctness for convergent terms)**.** *For any $e \in E$, $t \in \Lambda$, $\sigma \in \Sigma$, $v \in V$, if $e \vdash^{\sigma} t \Rightarrow v$ then, for all $s \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s \rightarrowtail v$.*

Proof. The proof is by induction in the derivation of $e \vdash^{\sigma} t \Rightarrow v$. We illustrate the proof for the case of the assignment command. Let us recall the rule for assignment:

$$(\textsc{Assign})\ \frac{e \vdash^{\sigma} t \Rightarrow \ell \qquad e \vdash^{\sigma} t' \Rightarrow k}{e \vdash^{\sigma} t := t' \Rightarrow \sigma[\ell \mapsto k]}$$

We have an inductive hypothesis for each premise of the rule:

(i) for all $s' \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \rightarrowtail \ell$

(ii) for all $s' \in S$, $[\![\, t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \rightarrowtail k$.

Therefore, by definition of $\rightarrowtail$, we get:

(iii) for all $s' \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \longmapsto^* (\mathsf{Const}\ \ell, \eta_1) \mid \sigma \mid s'$ for some $\eta_1 \in H$

(iv) for all $s' \in S$, $[\![\, t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \longmapsto^* (\mathsf{Const}\ k, \eta_2) \mid \sigma \mid s'$ for some $\eta_2 \in H$.

Now we can make the following sequence of transitions:

$$
\begin{array}{lr}
[\![\, t := t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s & \\
= ([\![\, t := t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
\longmapsto (\mathsf{Push}\ [\![\, t'\,]\!] \triangleright \mathsf{Push}\ [\![\, t\,]\!] \triangleright \mathsf{Frame}\ (:=), [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid s & \text{by definition of } [\![\_]\!] \\
\longmapsto (\mathsf{Push}\ [\![\, t\,]\!] \triangleright \mathsf{Frame}\ (:=), [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by the Push rule} \\
\longmapsto (\mathsf{Frame}\ (:=), [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid ([\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by the Push rule} \\
\longmapsto ([\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid [:= \ \bullet\, ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}})] :: s & \text{by the Frame rule} \\
= \ [\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid [:= \ \bullet\, ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}})] :: s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
\longmapsto^{*} (\mathsf{Const}\ \ell, \eta_1) \mid \sigma \mid [:= \ \bullet\, ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}})] :: s & \text{by (iii)} \\
\longmapsto ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid [:= \ \ell \bullet\,] :: s & \text{by a Const rule} \\
= [\![\, t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid [:= \ \ell \bullet\,] :: s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
\longmapsto^{*} (\mathsf{Const}\ k, \eta_2) \mid \sigma \mid [:= \ \ell \bullet\,] :: s & \text{by (iv)} \\
\longmapsto (\mathsf{Op}\ :=, \eta_2) \mid \sigma \mid [:= \ \ell,\, k \bullet\,] :: s & \text{by a Const rule} \\
\longmapsto (\mathsf{Cont}, \eta_2) \mid \sigma[\ell \mapsto k] \mid s & \text{by the Op } (:=) \text{ rule .}
\end{array}
$$

Thus, we have proved $[\![\, t := t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s \rightarrowtail \sigma[\ell \mapsto k]$. The remaining cases are similar. ◀

4.3.1 Correctness for divergent terms

We continue with the definition of the coinductive big-step semantics. In the following definition we present the rules for the imperative fragment of our language, since the rules for the other terms are similar to those in Definition 34, except for the propagation of the store through the premises:

▶ **Definition 47** (Coinductive semantics).

$$
(\textsc{Deref})\ \frac{e \vdash^{\sigma} t \Rightarrow \infty}{e \vdash^{\sigma} !\, t \Rightarrow \infty} \qquad (\textsc{Newvar})\ \frac{\underline{\ell}\,[e] :: e \vdash^{\sigma \triangleright 0} t \Rightarrow \infty}{e \vdash^{\sigma} \mathbf{newvar}\ \ t \Rightarrow \infty}
$$

$$
(\textsc{Comp1})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \infty}{e \vdash^{\sigma} t_1 \,;\, t_2 \Rightarrow \infty} \qquad (\textsc{Comp1})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \sigma' \qquad e \vdash^{\sigma'} t_2 \Rightarrow \infty}{e \vdash^{\sigma} t_1 \,;\, t_2 \Rightarrow \infty}
$$

$$
(\textsc{Assign1})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \infty}{e \vdash^{\sigma} t_1 := t_2 \Rightarrow \infty} \qquad (\textsc{Assign2})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \ell \qquad e \vdash^{\sigma} t_2 \Rightarrow \infty}{e \vdash^{\sigma} t_1 := t_2 \Rightarrow \infty}
$$

We can prove, by coinduction, that if the machine executes the code of a divergent term, then it never stops:

▶ **Theorem 48** (Correctness for divergent terms). *If $e \vdash^{\sigma} t \Rightarrow \infty$, then $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s \longmapsto^{\infty}$ for all $s \in S$.*

4.4 About the formalization

In Coq, most of the definitions above are represented using inductive types. For example, the following is the definition of the evaluation rules for abstractions, applications and variables:

```
Inductive eval (e : env) (q : store) : term → value → Prop :=
| eval_abs : forall t, eval e q (term_abs t) (value_abs t e)
| eval_app : forall t1 t2 t e' v,
               eval e q t1 (value_abs t e') →
               eval (t2 [e] :: e') q t v →
               eval e q (term_app t1 t2) v
| eval_var : forall n t' e' v,
               lookup e n = Some (t' [e']) →
               eval e' q t' v →
               eval e  q (term_var n) v
[...]
```

Here, each constructor corresponds to one of the rules of evaluation. For example, the constructor `eval_abs` corresponds to the rule (Abs) of Definition 38.

We rely on an important feature of Coq that is its built-in support for coinductive definitions and proofs, which allowed us to handle proof involving infinite sequences of transitions or coinductive evaluation in a simple manner. For example, the following is the definition of the coinductive evaluation rules for the case of the application:

```
CoInductive diverges (e : env) (q : store) : term → Prop :=
| diverges_app_fst :
    forall t1 t2,
      diverges e q t1 →
      diverges e q (term_app t1 t2)
| diverges_app_snd :
    forall t1 t t2 e' ,
      eval e q t1 (value_abs t e') →
      diverges (t2[e] :: e') q t →
      diverges e q (term_app t1 t2)
[...]
```

The constructor `diverges_app_fst` covers the case when the application diverges due to its operator (the term `t1`), and `diverges_app_snd` covers the case when the operator evaluates to an abstraction but the divergence occurs after the contraction of the redex. The correctness lemma for divergent terms is proved using the `cofix` tactic that permits proofs by coinduction:

```
Lemma correctness_for_divergent :
  forall e q t,
    diverges e q t →
    forall s,
      let g := closure_code (compile t) (compile_env e) in
      infseq (plus trans) (g, q, s).
Proof.
  cofix.
  [...]
Qed.
```

We have used Coq's `Ltac` tactic language to define tactics useful to automate some of the proofs of the formalization. For example, the following tactic is used in the proof of compiler correctness for convergent terms to make as many machine transitions as possible:

```
Ltac progress_until_possible :=
  repeat
    match goal with
```

```
    | [|- star trans _ _ ] ⇒
      first [
          eassumption
        | apply star_refl
        | eapply star_step ; [econstructor | eauto]
        | eapply star_trans ; [eassumption | eauto]
        ]
    | [|- _ ] ⇒ simpl ; progress eauto
  end
```

Here, the inductive type `star trans` represents a sequence of machine transitions. This tactic tries to prove a goal where the conclusion has the form `star trans _ _`. First, it tries to use an assumption to prove the goal, but if it is not possible, it will try to make zero, one or more steps (in that order) to reach the desired configuration.

We have measured the size of the formalization using the tool `coqwc` that prints the number of lines of code designated to specifications or proofs. The next table shows the results for the formalization of each of the three languages we considered in the paper:

Language	Specifications	Proofs
Call-by-name lambda calculus	345	531
Call-by-name lambda calculus with strict operators	336	199
Imperative higher-order language	468	272

The formalization of the first section has larger proofs scripts than the others. This is due to the fact that the use of small-step semantics requires to prove more results to capture the notion of correctness of the compiler and to a less extensive use of the `Ltac` mechanism.

5 Conclusion

In this paper we used well-known techniques [21, 22, 2] to mechanize in Coq the correctness of a compiler for a higher-order imperative language to a variant of the Krivine abstract machine. As far as we know, this is the first proof of correctness of a compiler combining call-by-name lambda calculus extended with a store and strict operators.

This formalization is also one of our first steps towards proving the correctness of a compiler for an Algol-like language [25]. Our next steps towards that goal involve (i) to add booleans with non-strict binary operations, (ii) to impose a type system on the source language, and (iii) to add a recursion operator.

Most of those changes planned for the language also entail modifications in the design of the compiler or the machine. For example, if we impose a type system in the language, the compiler might be designed to compile typing derivations instead of raw terms, as we do in this paper. The type system should also enable us to eliminate the need for a dereferencing operator, since we can detect during type-checking the different roles of the occurrences of a variable.

We plan to make some improvements in the abstract machine and also consider the use of the refocusing technique [10] to derive an abstract machine for the imperative language. One downside of our machine is the overhead incurred by the use of *frames* to implement strict operators; one possible remedy for this is the use of *stack markers* as in the ZAM machine [18]. Since we are using call-by-name evaluation, we could get some improvements in the execution by also considering sharing as in [17, 16].

Related work. Leroy [22] defined an abstract machine for a call-by-value lambda calculus, and used coinductive big-step semantics to describe the behavior of divergent programs. He also used Coq to prove the correctness of the compiler and some additional semantic properties like evaluation determinism and progress. A similar approach has been used by Leroy [21] and Bertot [3] for the simple imperative language.

Danvy and Nielsen [10] introduced the *refocusing* technique, that allows to systematically derive abstract machines from reduction semantics, by applying successive program transformations. Sieczkowski *et al.* [29] formalized in Coq and proved correct the technique for some applicative languages. We have taken Sieckzowski's formalization and adapted it for the language in Sec. 3; the resulting formalization is longer than our original mechanization. This happens because that method requires to prove several technical lemmas for each language; it could be interesting to investigate the possibility of stating refocusing more abstractly in order to prove some of those lemmas in a more general setting. It is not immediate if this technique can be applied to imperative languages, like the one in Section 4.

Chlipala [5, 6, 7] and Benton [1] used denotational semantics and logical relations to structure the proof of correctness of compilers for several programming languages, including typed lambda calculus and impure functional languages. Peter Selinger [28] derived extensions of the Krivine machine from the CPS translations of the $\lambda\mu$-calculus. Piróg *et al.* [24] derived a lazy abstract machine for an applicative language and formalized that derivation in Coq.

Acknowledgements. We would like to thank three anonymous reviewers for their comments and suggestions on an earlier version of this article.

References

1 Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.*, 44(9):97–108, August 2009.

2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, September 1998.

3 Yves Bertot. Theorem proving support in programming language semantics. *CoRR*, abs/0707.0926, 2007.

4 Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1), 2007.

5 Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, June 2007.

6 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, September 2008.

7 Adam Chlipala. A verified compiler for an impure functional language. In *POPL*, pages 93–106, 2010.

8 G. Cousineau and P.-L. Curien. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, April 1987.

9 Pierre-Louis Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82(2):389–402, 1991.

10 Olivier Danvy and Lasse Nielsen. Refocusing in reduction semantics. Research report BRICS RS-04-26, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004.

11 Stephan Diehl and Peter Sestoft. Abstract machines for programming language implementation. *Future Gener. Comput. Syst.*, 16(7):739–751, May 2000.

12 Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, March 1998.

13 Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 141–152. ACM, 2006.

14 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007.

15 P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

16 John Launchbury. Lazy imperative programming. In *ACM Sigplan Workshop on State in Programming Languages*, pages 46–56, 1993. (available as YALEU/DCS/RR968, Yale University).

17 John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

18 Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

19 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

20 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

21 Xavier Leroy. Mechanized semantics – with applications to program proof and compiler verification. In *Logics and Languages for Reliability and Security*, pages 195–224. IOS Press BV, 2010.

22 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, February 2009.

23 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

24 Maciej Pirog and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq. *SIGPLAN Not.*, 45(11):25–36, September 2010.

25 John C. Reynolds. Using functor categories to generate intermediate code. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'95, pages 25–36, New York, NY, USA, 1995. ACM.

26 John C. Reynolds. The essence of Algol. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

27 M. Rittri and Institutionen för informationsbehandling (Göteborg). *Proving the Correctness of a Virtual Machine by a Bisimulation*. Department of computer sciences, 1988.

28 Peter Selinger. From continuation passing style to Krivine's abstract machine. Manuscript, 2003. Available in Peter Selinger's web site.

29 Filip Sieczkowski, Małlgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 72–88, Berlin, Heidelberg, 2011. Springer-Verlag.

Definitional Extension in Type Theory

Tao Xue

School of Computer Science, McGill University, Montreal, Canada
xuet.cn@hotmail.com

Abstract

When we extend a type system, the relation between the original system and its extension is an important issue we want to understand. Conservative extension is a traditional relation we study with. But in some cases, like coercive subtyping, it is not strong enough to capture all the properties, more powerful relation between the systems is required. We bring the idea definitional extension from mathematical logic into type theory. In this paper, we study the notion of definitional extension for type theories and explicate its use, both informally and formally, in the context of coercive subtyping.

1998 ACM Subject Classification F.4 Mathematical Logic and Formal Language

Keywords and phrases conservative extension, definitional extension, subtype, coercive subtyping

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.251

1 Introduction

In the studies of type theory, sometimes we extend a type system with some notions and rules. We are interested in what power the extension systems can bring to us, and we also want to know the relations between the systems. Understanding the relations between the systems tells us some of the properties the new system should hold. The most common property we always think of is conservativity, or put in anther way, whether the extension is a *conservative extension*. For example, Hofmann showed the conservativity of extensional type theory over intensional type theory with extensional concepts added [4]. Informally, conservativity means that the new system maybe more convenient than the original system but it cannot prove any new theorem within the old language. It requires that all the theorems in the old language, which are provable in new system, are also provable in the old system.

Subtypes are introduced into type theory and studied in many works [1, 2, 14, 15]. Coercive subtyping [7] is one approach of studying subtype in type theory. Unlike the traditional way of dealing subtype with subsumption rule

$$\frac{a : A \qquad A \leq B}{a : B}$$

which is very common in the study of functional programming languages [13], coercive subtyping is an abbreviation mechanism. We consider a unique coercion c between two types A and B, written as $A <_c B$. Intuitively, for every place we require a term of type B, we can use a term a of type A instead, and it is just an abbreviation of using the term $c(a)$. This simple mechanism is quite powerful, one recent use is in the study of linguistic semantics [9, 19].

Since we take coercive subtyping as an abbreviation mechanism, we don't want it to increase any power of the existing system. Soloviev and Luo [17] studied the relationship between a type system and its coercive subtyping extension and called it "conservativity". In

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 251–269

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

fact, the relationship is not quite the same as the traditional notion of conservative extension and it turns out that it can better be characterized as an *definitional extension* in a more general sense. In this paper, we will give a definition of this notion of definitional extension and explicate it, both informally with a simple example and formally for coercive subtyping.

Soloviev and Luo's previous work [17] was based on a notion of basic subtyping rules which turns out to be unnecessarily general. It does not exclude certain "bad" subtyping rules which cannot be used normally but can be applied once we introduce coercive subtyping. This would destroy the consistency of the whole system. Recently, we fix the problem by considering coercion sets rather than coercion rules and, furthermore, the latter can be captured by the former [11, 18]. In this paper, our treatment of coercive subtyping is based on this new framework.

The paper goes in the following way. We give the motivation of introducing definitional extension in Section 2 by showing that coercion mechanism cannot be expected as a conservative extension. In Section 3, we present a definition of conservative extension and definitional extension in type theory. We use a simplified example to demonstrate the relation between a system and its coercion extension in Section 4 and give a sketch on the full study of the relation in Section 5.

2 Motivation: coercive subtyping

Coercive subtyping [7] is an approach to introducing subtypes into type theory and it considers subtyping by means of *abbreviations*.

The basic idea of coercive subtyping is that, when we consider A as a subtype of B, we choose a unique function c from A to B, and declare c to be a *coercion*, written as $A <_c B$. Intuitively, the idea means anywhere we need to write an object of type B, we can use an object a of type A instead. Actually in the context, the object a is to be seen as an *abbreviation* for the object $c(a) : B$. More precisely, if we have f from B to C, then f can be applied to any object a of type A to form $f(a)$ of type C, which is definitionally equal to $f(c(a))$. We can consider $f(a)$ to be an abbreviation for $f(c(a))$, with coercion c being inserted to fill the *gap* between f and a. The idea above could be captured by means of the following formal rules:

$$\frac{f\colon B \to C \quad a\colon A \quad A <_c B}{f(a)\colon C} \qquad \frac{f\colon B \to C \quad a\colon A \quad A <_c B}{f(a) = f(c(a))\colon C}$$

As an extension of a type theory, coercive subtyping is based on the idea that subtyping is abbreviation. On the one hand, it should not increase any expressive power of the original system. On the other hand, coercions can always be correctly inserted to obtain the abbreviated expressions as long as the basic coercions are coherent [1].

In the study coercive subtyping, Soloviev and Luo tried to think it be a conservative extension [17]. But we find that conservativity is not accurate to capture the relation. In the expressions of coercive subtyping, there are "gaps" introduced by the coercions. Given $f : B \to C$ and $a : A$, although $f(a) : C$ is well-formed with coercive subtyping $A <_c B$, we can still image that there is a "gap" in $f(a)$ between f and a. As mentioned above, we want to show that all the "gaps" in the expressions caused by coercions can be correctly inserted.

[1] Informally, coherence in coercive subtyping means there is unique coercion between two different types, further details are discussed in Section 5.

For example, let's consider types $Nat = 0|succ(Nat)$, $Bool = true|false$ and coercion $Bool <_c Nat$. With coercive subtyping, we can have terms:

$$succ(true), succ(false), succ(succ(true)), succ(succ(false)), \ldots$$

As we have emphasised that coercive subtyping is just abbreviation, these terms should actually be equivalent to the following terms:

$$succ(c(true)), succ(c(false)), succ(succ(c(true))), succ(succ(c(false))), \ldots$$

Such equivalence is the most important property of the extension with coercive subtyping. We want to show that all the judgements or derivations in the system with coercive subtyping can be translated into the equivalent ones in the original type system. "conservative extension" is not enough for our use, it only talks about whether the derivable judgements in new system are still derivable in the original one, it doesn't ask for such equivalence connection. We find the idea of "definitional extension" in first-order logic theories contains a translation between the formulas of the theories. Hence, we think that such notion of definitional extension is a suitable option to describe the relation between a type system and its coercive subtyping extension.

3 Conservative extension and definitional extension

To build a definition for the definitional extension, we should give definitions for the equivalence between judgements and equivalence between derivations first. Such definitions depend on the forms of judgements. In this paper, we will consider the type systems formalised in Luo's logical framework[2] [6]. For other cases, we should be able to consider them in a similar way.

3.1 Luo's logical framework

Luo's logical framework [6] is a typed version of Martin-Löf's logical framework [15]. In Luo's logical framework, the functional abstractions of the form $(x)k$ in Martin-Löf's logical framework are replaced by the typed form $[x : K]k$. We will simply call it LF in the rest part of this paper.

LF is a type system with terms of the following forms:

$$\mathbf{Type},\ El(A),\ (x : K)K',\ [x : K]k',\ f(k)$$

The kind **Type** denotes the conceptual universe of types; $El(A)$ denotes the kind of objects of type A; $(x\colon K)K'$ denotes a dependent product; $[x\colon K]k'$ denotes an abstraction; and $f(k)$ denotes an application. The free occurrences of the variable x in K' and k' are bound by the binding operators $(x : K)$ and $[x : K]$. There are five forms of judgements in LF:

- $\Gamma \vdash$ **valid**, which asserts that Γ is a valid context.
- $\Gamma \vdash K$ **kind**, which asserts that K is a valid kind.
- $\Gamma \vdash k : K$, which asserts that k is an object of kind K.
- $\Gamma \vdash k = k' : K$, which asserts that k and k' are equal objects of kind K.
- $\Gamma \vdash K = K'$, which asserts that K and K' are two equal kinds.

Figure 7 shows the LF rules. It contains the rules for context validity and assumptions, the general equality rules, the type equality rules, the substitution rules, the rules for kind Type and the rules for dependent product kinds.

[2] It is different from the Edinburgh Logical Framework [3].

3.2 Conservative extension

In mathematical logic, when we say a logical theory S_2 is an *extension* of a theory S_1, it means that the syntax of S_2 includes all the syntax of S_1 and every theorem of S_1 is a theorem of S_2. We call S_2 a *conservative extension* of S_1, if S_2 is an extension of S_1 and we have a further condition that any theorem of S_2 in the language of S_1 is a theorem in S_1.

When we talk about such extensions, it is important to point out that the syntax of S_2 contains all the syntax of S_1. We can have two labels of the theorems, one is *proposable*, another is *provable*. Proposable means the theorem can be written down in the language, not necessary be proved. Provable means the theorem can not only be written down but also be proved. In conservative extension, we don't care those theorems which are proposable in S_2 but not proposable in S_1. However, we will see later that in definitional extension we need to think of them.

We can consider the idea similarly in type theory. Instead of thinking of the theorems, we would like to think of the judgements. If a judgement can be derived through the rules in the system, we call it a *derivable judgement*. We say type system T_2 which includes all the syntax of system T_1 is a *conservative extension* of T_1 , if for any proposable sequent (judgement) t of the system T_1, t is derivable in T_2 implies that t is derivable in T_1. If a sequent is not proposable in T_1 (only proposable in T_2), its derivability does not matter.

More precisely, let's use $\vdash_T$ for the derivable judgements in system T. T_2 is an extension of T_1 requires that, T_2 includes all the syntax of T_1 and for any judgement $\Gamma \vdash \Sigma$ in T_1(it may not be derivable):

$$\Gamma \vdash_{T_1} \Sigma \quad \Rightarrow \quad \Gamma \vdash_{T_2} \Sigma$$

For such an extension to be conservative, we also require:

$$\Gamma \vdash_{T_2} \Sigma \quad \Rightarrow \quad \Gamma \vdash_{T_1} \Sigma$$

▶ **Definition 1** (conservative extension)**.** Type theory T_2 is a conservative extension of T_1, if T_2 includes all the syntax of T_1 and for any proposable judgement J in T_1, there's a derivation of J in T_1 if and only if there's a derivation of J in T_2.

3.3 Definitional extension

Sometimes, conservative extension is not powerful enough to describe the relation between the systems. In some cases, like the study of coercive subtyping [11], we not only want to show the conservativity, but also want the systems to keep a stronger relation. We want the formulas, judgements or derivations in one system could be translated to corresponding ones in another system. Definitional extension describes such kind of relation.

Traditionally, the notion of *definitional extension* was formulated for first-order logical theories [5]: a first-order theory is a definitional extension of another if the former is a conservative extension of the latter and any formula in the former is logically equivalent to its translation in the latter. More precisely, a definitional extension S' of a first-order theory S is obtained by successive introductions of relations(or functions) in such a way that, for example, for an n-ary relation R, the following *defining axiom* of R is added:

$$\forall x_1 ... \forall x_n.\ R(x_1, ..., x_n) \iff \phi(x_1, ..., x_n),$$

where $\phi(x_1, ..., x_n)$ is a formula in S.

For such a definitional extension S', we have:

- for any formula ψ in S', $\psi \iff \psi^*$ is provable in S', where ψ^* is the formula in S obtained from ψ by replacing any occurrence of $R(t_1, ..., t_n)$ by $\phi(t_1, ..., t_n)$ (with necessary changes of bound variables).
- S' is a conservative extension of S.

Taking the idea of definitional extension, especially the translation between formulas, we are going to consider a similar relation in type theory. The notion of definitonal extension in first-order logic is characterised in terms of translation on *formulas*. In our type theory, we have at least two options to present the translation on: *judgements* and *derivations*. Intuitively, derivable judgements and derivations are very close related to each other. In analogy to the formulas in logic, it sounds even more natural to use judgements in type theory. However, we will choose derivations to formalise our definition. Let's consider the type systems with coercive subtyping. Translating a judgement with coercive subtyping into a judgement without coercive subtyping requires us to point out all the "gaps" introduced by coercion in the judgement. They are not simply marked in the syntax, and due to the congruence rules of subtyping, the insertion might not be syntactically unique. We have to look up the derivations to find the coercions out. More generally, in intensional type theories, the non-syntax-directed use of the conversion rule makes the connection between the judgement and derivation non-structural. When we have the mechanisms like coercion, the choice of rules by which to refine a judgement becomes no more free. Based on these reasons, it is necessary to give the definition in term of derivations.

Before giving a formal definition of definitional extension, we need to define the equivalence between derivations first. The equivalence between the derivations can be defined by the equivalence between derivable judgements and the equivalence between the judgements intuitively means that the corresponding parts of two judgements are equal formulas. In LF, the judgements are of form:

$$\Gamma \vdash \textbf{valid},\ \Gamma \vdash K\ \textbf{kind},\ \Gamma \vdash k : K,\ \Gamma \vdash k_1 = k_2 : K\ \ \textit{and}\ \ \Gamma \vdash K_1 = K_2$$

Hence, we can define the equivalence between the judgements in the following way:

▶ **Notation 2.** In a type system S specified in LF, let Γ_1 and Γ_2 be

$$\Gamma_1 \equiv x_1 : K_1, x_2 : K_2, \cdots, x_n : K_n$$

$$\Gamma_2 \equiv x_1 : M_1, x_2 : M_2, \cdots, x_n : M_n$$

The equality $\Gamma \vdash \Gamma_1 = \Gamma_2$ is an abbreviation for the following list of n judgements:

$$\begin{aligned}
\Gamma &\vdash K_1 = M_1;\\
\Gamma,\ x_1 : K_1 &\vdash K_2 = M_2;\\
&\vdots\\
\Gamma, x_1 : K_1, \cdots, x_{n-1} : K_{n-1} &\vdash K_n = M_n.
\end{aligned}$$

With the LF rules, we can proof the following propositions of our equality abbreviation in type system S specified in LF. Then, we can use them to define equality between judgements and between derivations in S.

▶ **Proposition 3.** *In a type system S specified in LF.*

1. *If Γ_1 is a valid context, then $\vdash \Gamma_1 = \Gamma_1$.*
2. *If $\Gamma \vdash \Gamma_1 = \Gamma_2$, then $\Gamma \vdash \Gamma_2 = \Gamma_1$.*

3. *If* $\Gamma \vdash \Gamma_1 = \Gamma_2$ *and* $\Gamma \vdash \Gamma_2 = \Gamma_3$, *then* $\Gamma \vdash \Gamma_1 = \Gamma_3$.
4. *If* $\Gamma, \Gamma_1 \vdash J$ *and* $\Gamma \vdash \Gamma_1 = \Gamma_2$ *then* $\Gamma, \Gamma_2 \vdash J$. *(*$J$ *is of form* **valid**, K **kind**, $k\colon K$, $k_1 = k_2\colon K$ *or* $K_1 = K_2$*)*

Proof. See appendix B. ◀

▶ **Definition 4.** (equality between judgements) Let S be a type theory specified in LF. The notion of equality between judgements of the same form in S, notation $J_1 =_s J_2$, is inductively defined as follows:

1. $(\Gamma_1 \vdash \textbf{valid}) =_s (\Gamma_2 \vdash \textbf{valid})$ iff $\vdash \Gamma_1 = \Gamma_2$ is derivable in S.
2. $(\Gamma_1 \vdash K_1\ \textbf{kind}) =_s (\Gamma_2 \vdash K_2\ \textbf{kind})$ iff $\vdash \Gamma_1 = \Gamma_2$ and $\Gamma_1 \vdash K_1 = K_2$ are derivable in S.
3. $(\Gamma_1 \vdash k_1\colon K_1) =_s (\Gamma_2 \vdash k_2\colon K_2)$ iff $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$ and $\Gamma_1 \vdash k_1 = k_2\colon K_1$ are derivable in S.
4. $(\Gamma_1 \vdash K_1 = K_1') =_s (\Gamma_2 \vdash K_2 = K_2')$ iff $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$ and $\Gamma_1 \vdash K_1' = K_2'$ are derivable in S.
5. $(\Gamma_1 \vdash k_1 = k_1'\colon K_1) =_s (\Gamma_2 \vdash k_2 = k_2'\colon K_2)$ iff $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$, $\Gamma_1 \vdash k_1 = k_2\colon K_1$ and $\Gamma_1 \vdash k_1' = k_2'\colon K_1$ are derivable in S.

The equivalence between the derivations can be given as follows:

▶ **Definition 5.** (equality between derivations) Suppose d is a derivation in type system S, let $conc(d)$ denote the conclusion of derivation d. Given two derivations d_1 and d_2, we call d_1 and d_2 equivalent derivations in S and write $d_1 \sim_s d_2$ iff $conc(d_1) =_s conc(d_2)$ in S.

▶ **Theorem 6.** *Let* S *be a type theory specified in LF,* $=_s$ *and* $\sim_s$ *are equivalence relations.*

Proof. Straight with Proposition 3 and LF rules in Figure 7. ◀

When no confusion may occur, We will omit S and simply write $=$ and $\sim$ for the equivalence between judgements and derivations in system S.

▶ **Definition 7.** (definitional extension) We call T_2 is a definitional extension of T_1, if we have:

- for any derivation d in T_2, we can translate d into a corresponding derivation d' in T_1, d and d' are equivalent derivations in T_2.
- T_2 is a conservative extension of T_1.

4 A simple example

In Section 2, we have proposed our motivation of introducing definitional extension: conservative extension is not enough to capture the properties when we extend a system with coercive subtyping. However, we find that coercive subtyping is not a definitional extension either. The reason is that terms like $succ(true)$ are proposable but not derivable in the original system. With the help of coercive subtyping, they are derivable. It doesn't satisfy the definition of conservative extension, hence not definitional extension. To figure out what exactly the relation is, we have to employ some intermediate systems to help us.

The complete description of the relations between a type system, its coercive subtyping extension and intermediate systems is complex and includes some tedious proofs [18]. We will give a sketch of it in the next section. In this section, we try to give an example with coercive subtyping to tell such story in a simple and informal way. Through this trivial looking example, we would like to show the following points: 1) why definitional extension is still not enough (or why we introduce a intermediate system); 2) how to introduce a proper intermediate system; 3) the relations between the systems.

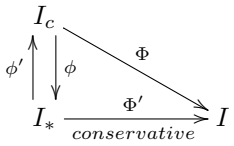


Figure 1 Relations between I_c, I_* and I.

We will consider three systems in the example, a type system I and two of its extensions. I is a very simple type system with only two constant types *Nat* and *Bool*. We extend it into system I_c with one coercion $Bool <_c Nat$. We also introduce system I_* as extension of I with $*$ calculus, such $*$ plays a role of gap holder when we apply the coercion. Through the relations between the judgements of these systems, we can draw a picture for the links between these systems as Figure 1 (definitions of Φ, Φ', ϕ and ϕ' are shown in the later parts of this section).

We will use some informal notions in this example section for the purpose of a simple description. We only have subtyping in the example, while in LF we shift them into subkinding. We will omit the contexts of judgements and use judgements to formalise translations for definitional extension. It is worth pointing out that using judgements for the translations doesn't violate our previous settings with derivations in this example. Because in the syntax of judgements, the applications of *succ* on *true* or *false* indicate the use of coercion application rule in the derivation clearly.

4.1 System I

In I, we only have two basic types *Nat* and *Bool* with their constructors, and a term c of type $Bool \to Nat$:

$Nat\colon Type,\quad 0\colon Nat,\quad succ\colon Nat \to Nat,$
$Bool\colon Type,\quad true\colon Bool,\quad false\colon Bool,\quad c\colon Bool \to Nat$

And, we have the following rules:

$$\frac{f\colon M \to N \quad a\colon M}{f(a)\colon N} \qquad \frac{a\colon M}{a = a\colon M} \qquad \frac{a_1 = a_2\colon M}{a_2 = a_1\colon M} \qquad \frac{a_1 = a_2\colon M \quad a_2 = a_3\colon M}{a_1 = a_3\colon M}$$

In this system, the judgements are of form:

$$a\colon M \quad and \quad a_1 = a_2\colon M$$

We can easily list out all the derivable judgements in I, they can only be of the following cases:

$0\colon Nat,\quad succ\colon Nat \to Nat,\quad succ(...succ(0))\colon Nat,$
$true\colon Bool,\quad false\colon Bool,\quad c\colon Bool \to Nat,\quad c(true)\colon Nat,\quad c(false)\colon Nat,$
$succ(...succ(c(true)))\colon Nat,\quad succ(...succ(c(false))\colon Nat,$
$0 = 0\colon Nat,\quad succ(...succ(0)) = succ(...succ(0))\colon Nat,$
$true = true\colon Bool,\quad false = false\colon Bool,$
$succ = succ\colon Nat \to Nat,\quad c = c\colon Bool \to Nat,$
$succ(...succ(c(true))) = succ(...succ(c(true)))\colon Nat,$
$succ(...succ(c(false))) = succ(...succ(c(false)))\colon Nat$

▶ Remark. For the judgements like $succ(...succ(c(true))) = succ(...succ(c(true))) : Nat$, the left and right term of the equal mark have the same number of *succ*. It's the same case for the other similar judgements in rest of this section.

4.2 System I_c

Let's enrich the system I with coercive subtyping. We extend I into system I_c with coercion $Bool <_c Nat$ and coercion application rules:

$$\frac{f : B \to C \quad a : A \quad A <_c B}{f(a) : C} \qquad \frac{f : B \to C \quad a : A \quad A <_c B}{f(a) = f(c(a)) : C}$$

The judgements in system I_c are of form[3]:

$$a : M \quad and \quad a_1 = a_2 : M$$

We can get all the derivable judgements in system I_c. Besides all those we have in system I, we can derive the following judgements:

$$\begin{aligned}
&succ(...succ(true)) : Nat,\ succ(...succ(false)) : Nat,\\
&succ(...succ(true)) = succ(...succ(true)) : Nat,\\
&succ(...succ(false)) = succ(...succ(false)) : Nat,\\
&succ(...succ(c(true))) = succ(...succ(true)) : Nat,\\
&succ(...succ(c(false))) = succ(...succ(false)) : Nat,\\
&succ(...succ(true)) = succ(...succ(c(true))) : Nat,\\
&succ(...succ(false)) = succ(...succ(c(false))) : Nat
\end{aligned}$$

4.3 Relation between I and I_c

Now, let's consider the relation between I and I_c. We want to show that every derivable judgement in I_c is equivalent to a corresponding derivable judgement in I. To achieve this goal, we define a translation Φ from every derivable judgements in system I_c to derivable judgements in I. Φ inserts all the gaps caused by coercion with term c (since we only have one subtyping relation). The definition of Φ is as follows :

1. $\Phi(t) \equiv t$, if the t is the judgement in I,
2. $\Phi(t) \equiv succ(...succ(c(b)) : Nat$, if $t \equiv succ(...succ(b)) : Nat$, b is either *true* or *false*,
3. $\Phi(t) \equiv succ(...succ(c(b))) = succ(...succ(c(b))) : Nat$,
   if $t \equiv succ(...succ(b)) = succ(...succ(b)) : Nat$, b is either *true* or *false*,
4. $\Phi(t) \equiv succ(...succ(c(b) = succ(...succ(c(b))) : Nat$,
   if $t \equiv succ(...succ(b)) = succ(...succ(c(b))) : Nat$, b is either *true* or *false*,
5. $\Phi(t) \equiv succ(...succ(c(b)) = succ(...succ(c(b))) : Nat$,
   if $t \equiv succ(...succ(c(b))) = succ(...succ(b)) : Nat$, b is either *true* or *false*.

It is easy to prove that Φ is total. In order to show the equality between the judgements in I_c and their translations in I, we can prove Φ is holding the following property.

▶ **Proposition 8.** *For any derivable judgement t in system I_c, $\Phi(t)$ and t are equivalent judgements in system I_c*

[3] We do not consider subtyping relation as a judgement in this example section. But in full study of coercive subtyping in LF, we will think them as judgements. See the discussion in Section 5.

Although we have shown certain relation between system I and I_c, it just satisfies the first condition of definitional extension. We cannot say I_c is a definitional extension of I, because definitional extension requires conservativity. Unfortunately, I_c is not a conservative extension of I_c. A simple counter example is that, $succ(true) : Nat$ is a judgement but not derivable in I, however it is derivable in I_c. It doesn't satisfy the definition of conservative extension.

The reason for this problem is that the abbreviation with "gaps" mechanism of coercive subtyping makes such non-well-formed sequences to be well-formed. If we consider an intermediate system with an extra place holder for the "gaps", we may get rid of the problem.

4.4 System I_*

To make a more specific study for the relations, we will introduce another system I_*. Intuitively, I_* means that for any place we want to use a coercion, we insert a symbol $*$ to fill the gap, it equals to the term where the coercion applied. Similarly like I_c, I_* extends system I with the following rules:

$$\frac{f\colon B \to C \quad a\colon A \quad A <_c B}{f * a\colon C} \qquad \frac{f\colon B \to C \quad a\colon A \quad A <_c B}{f * a = f(c(a))\colon C}$$

In system I_*, the judgements are also of form:

$$a : M \quad and \quad a_1 = a_2 : M$$

We can list all the derivable judgements in system I_* as follows, besides all those in system I:

$$succ(...succ * true) : Nat, \quad succ(...succ * false) : Nat,$$
$$succ(...succ * true) = succ(...succ * true) : Nat,$$
$$succ(...succ * false) = succ(...succ * false) : Nat,$$
$$succ(...succ(c(true)) = succ(...succ * true) : Nat,$$
$$succ(...succ(c(false))) = succ(...succ * true) : Nat,$$
$$succ(...succ * true) = succ(...succ(c(true))) : Nat,$$
$$succ(...succ * true) = succ(...succ(c(false))) : Nat$$

4.5 Relation between I and I_*

It is trivial to show that I_* is a conservative extension of I. Since judgements like $succ * true : Nat$ are not judgements in I, we don't need to consider them, all the other derivable judgements in I_* are exactly the same judgements in I.

▶ **Proposition 9.** *System I_* is a conservative extension of system I.*

Like what we have done for the relation between I_c and I. We can introduce a total translation Φ' from judgements of system I_* to judgements of system I. Intuitively, it substitutes all the appearance of $*$ with our only coercion c:

1. $\Phi'(t) \equiv t$, if the t is a derivable judgement in I,
2. $\Phi'(t) \equiv succ(...succ(c(b)) : Nat$, if $t \equiv succ(...succ * b) : Nat$, b is either $true$ or $false$,
3. $\Phi'(t) \equiv succ(...succ(c(b)) = succ(...succ(c(b))) : Nat$,
   if $t \equiv succ(...succ * b) = succ(...succ * b) : Nat$, b is either $true$ or $false$,
4. $\Phi'(t) \equiv succ(...succ(c(b) = succ(...succ(c(b))) : Nat$,
   if $t \equiv succ(...succ * b) = succ(...succ(c(b))) : Nat$, b is either $true$ or $false$,
5. $\Phi'(t) \equiv succ(...succ(c(b)) = succ(...succ(c(b))) : Nat$,
   if $t \equiv succ(...succ(c(b))) = succ(...succ * b) : Nat$, b is either $true$ or $false$.

Now we have a total translation Φ' from system I_* to system I. Again, it is easy to prove that for every derivable judgement t in system I_*, $\Phi'(t)$ and t are equal judgements in I_*. Together with the conservative property, we can conclude that I_* is a definitional extension of I.

- For any derivable judgement t in I_*, $\Phi'(t)$ is a derivable judgement in I, $\Phi'(t)$ and t are equivalent judgements in I_*.
- I_* is a conservative extension of I.

4.6 Relation between I_c and I_*

Now, let's think of the relation between, I_c and I_*. The rules and judgements are almost the same, only different in symbols. Intuitively, they should be equivalent systems. We can show their equality by introducing two more translations between the systems: ϕ from the judgement of I_c to the judgement of I_*, ϕ' from the judgement of I_* to the judgement of I_c.

- ϕ changes every place of $succ(true)$ or $succ(false)$ in system I_c into term $succ * true$ or $succ * false$.
- ϕ' simply removes every occurrence of $*$ in system I_*.

It's trivial to show that ϕ and ϕ' are total, and easy to prove that I_c and I_* are two equivalent systems by means of :

▶ **Proposition 10.**

- *For every judgement t in I_c, $\phi'(\phi(t)) \equiv t$.*
- *For every judgement t' in I_* $\phi(\phi'(t')) \equiv t'$.*

We can also show that Φ is a composition of Φ' and ϕ:

▶ **Proposition 11.** *For any derivable judgement t in I_c, $\Phi(t) \equiv \Phi'(\phi(t))$*

Finally, we can reach the conclusion for the relations between all these systems: I_c is equivalent to a system I_* which is a definitional extension of I, as shown in the graph previously (Figure 1):

- I_c is an equivalent system of I_*
- I_* is a definitional extension of I:

5 Coercive subtyping in LF

Luo formulated coercive subtyping [7] in his LF [6]. Later we find that the extension took a too general set of coercion rules which may ruin the consistency of the extension system. We solve the problem by reformulating it with some restriction [11, 18]. In this section, we give a sketch of reformulated system and proofs to show the definitional extension, further details could be found in the author's thesis [18].

We will mainly consider the following systems: an original type system T; an extension of system T with coercive subtyping ($T[\mathcal{C}]$); an extension of system T with coercive subtyping and place holder $*$ ($T[\mathcal{C}]^*$); an intermediate system without coercion application rules ($T[\mathcal{C}]_{0K}$).

We introduce coercive subtyping in type level (rules in Figure 2) and then move them into kind level (rules in Figure 3). The symbol $*$ is introduced as a place holder to fill the gaps left by the coercions. We call it $*$-calculus. Following the idea in Section 4, we should be able to show that $T[\mathcal{C}]^*$ is a definitional extension over T. Unfortunately, we can not reach this conclusion yet, because we need to consider the derivations of subtyping and subkinding judgements ($\Gamma \vdash A <_c B : \mathbf{Type}$ or $\Gamma \vdash K <_c K'$). We didn't consider them in the simplified

Base Coercion

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \in \mathcal{C}}{\Gamma \vdash A <_c B : \mathbf{Type}}$$

Congruence

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A' : \mathbf{Type} \quad \Gamma \vdash B = B' : \mathbf{Type} \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A' <_{c'} B' : \mathbf{Type}}$$

Transitivity

$$\frac{\Gamma \vdash A <_{c_1} B : \mathbf{Type} \quad \Gamma \vdash B <_{c_2} C : \mathbf{Type}}{\Gamma \vdash A <_{c_2 \circ c_1} C : \mathbf{Type}}$$

Substitution

$$\frac{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type} \qquad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]A <_{[k/x]c} [k/x]B : \mathbf{Type}}$$

Weakening

$$\frac{\Gamma, \Gamma' \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash K \ \mathbf{kind} \quad x \notin FV(\Gamma) \cup FV(\Gamma')}{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type}}$$

Context Retyping

$$\frac{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash K = K'}{\Gamma, x : K', \Gamma' \vdash A <_c B : \mathbf{Type}}$$

Figure 2 The structural subtyping rules of $T[\mathcal{C}]_0$.

example in the previous section, there was only one coercion taken as axiom. In a complete description in LF, we have derivations of these subtyping and subkinding judgements, we can hardly match them to any equivalent derivations in T. To fill this gap, we have to involve the intermediate system $T[\mathcal{C}]_{0K}$ into the relations between T, $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$. $T[\mathcal{C}]_{0K}$ extends T as $T[\mathcal{C}]$ but without the coercion application rules (rules in Figure 4). We will show that $T[\mathcal{C}]^*$ is a definitional extension of $T[\mathcal{C}]_{0K}$, $T[\mathcal{C}]_{0k}$ is a conservative extension of T and $T[\mathcal{C}]$ is an equivalent system of $T[\mathcal{C}]^*$.

5.1 System $T[\mathcal{C}]$

Let T be a type system specified in LF such as Martin-Löf's type theory [12] or UTT [6]. With a set $\mathcal{C}$ of coercive subtyping judgements (judgements of form $\Gamma \vdash A <_c B : \mathbf{Type}$), the following basic coercion rules in Figure 2, 3 and coercion application rules in Figure 4, we can extend T into a type system $T[\mathcal{C}]$ with coercive subtyping[4].

5.2 Coherence

Coherence is an important issue in coercive subtyping. Informally, it means there's a unique coercion between two types. To give a formal definition in our structure, we need to introduce an intermediate system $T[\mathcal{C}]_0$.

[4] Rules in Figures 2, 3, 4 and 5 are only the subtyping and subkinding rules. Figure 7 contains the rest LF rules.

Basic subkinding rule

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$$

Subkinding for dependent product kinds

$$\frac{\Gamma \vdash K_1' <_{c_1} K_1 \qquad \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 = K_2' \qquad \Gamma, x : K_1 \vdash K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']f(c_1(x'))$;

$$\frac{\Gamma \vdash K_1' = K_1 \qquad \Gamma, x' : K_1' \vdash K_2 <_{c_2} K_2' \qquad \Gamma, x : K_1 \vdash K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']c_2 f(x')$;

$$\frac{\Gamma \vdash K_1' <_{c_1} K_1 \qquad \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 <_{c_2} K_2' \qquad \Gamma, x : K_1 \vdash K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']c_2(f(c_1(x')))$.

Congruence for subkinding

$$\frac{\Gamma \vdash K_1 <_c K_2 \quad \Gamma \vdash K_1 = K_1' \quad \Gamma \vdash K_2 = K_2' \quad \Gamma \vdash c = c' : (K_1)K_2}{\Gamma \vdash K_1' <_c K_2'}$$

Transitivity for subkinding

$$\frac{\Gamma \vdash K <_c K' \quad \Gamma \vdash K' <_{c'} K''}{\Gamma \vdash K <_{c' \circ c} K''}$$

Substitution for subkinding

$$\frac{\Gamma, x : K, \Gamma' \vdash K_1 <_c K_2 \qquad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K_1 <_{[k/x]c} [k/x]K_2}$$

Weakening for subkinding

$$\frac{\Gamma, \Gamma' \vdash K_1 <_c K_2 \quad \Gamma \vdash K \ \mathbf{kind} \quad x \notin FV(\Gamma) \cup FV(\Gamma')}{\Gamma, x : K, \Gamma' \vdash K_1 <_c K_2}$$

Context Retyping for subkinding

$$\frac{\Gamma, x : K, \Gamma' \vdash K_1 <_c K_2 \quad \Gamma \vdash K = K'}{\Gamma, x : K', \Gamma' \vdash K_1 <_c K_2}$$

Figure 3 The subkinding rules of $T[\mathcal{C}]_{0K}$.

Coercive application rule

$$(CA1)\frac{\Gamma \vdash f : (x : K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) : [c(k_0)/x]K'}$$

$$(CA2)\frac{\Gamma \vdash f = f' : (x : K)K' \qquad \Gamma \vdash k_0 = k_0' : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f'(k_0') : [c(k_0)/x]K'}$$

Coercive definition rule

$$(CD)\frac{\Gamma \vdash f : (x : K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

Figure 4 The coercive application and definition rules of $T[\mathcal{C}]$.

$T[\mathcal{C}]_0$ is a system extending T with set $\mathcal{C}$ of coercion subtyping judgements, subtyping judgements $\Gamma \vdash A <_c B : \mathbf{Type}$ and basic subtyping rules (Figure 2).

▶ **Definition 12.** (coherence) $\mathcal{C}$ is called a coherent set of coercive subtyping judgement, if in $T[\mathcal{C}]_0$ we have:

1. $\Gamma \vdash A <_c B : \mathbf{Type}$ implies $\Gamma \vdash A : \mathbf{Type}$, $\Gamma \vdash B : \mathbf{Type}$, $\Gamma \vdash c : (A)B$ are derivable in T.
2. We cannot derive $\Gamma \vdash A <_c A : \mathbf{Type}$, for any Γ, A, c.
3. $\Gamma \vdash A <_{c_1} B : \mathbf{Type}$ and $\Gamma \vdash A <_{c_2} B : \mathbf{Type}$ imply that $\Gamma \vdash c_1 = c_2 : (A)B$ is derivable in T.

In fact, we can prove that any two coercions between two given kinds are equal in $T[\mathcal{C}]$. Let c and c' be two different coercion from K to K', $K <_c K'$ and $K <_{c'} K'$:

$$\begin{aligned}
\Gamma \vdash c &= [x : K](c(x)) && (\eta \text{ rule}) \\
&= [x : K]([y : K']y)c(x) && (\beta \text{ rule}) \\
&= [x : K]([y : K']y)(x) && (\xi \text{ and coercive definition}) \\
&= [x : K]([y : K']y)(c'(x)) && (\xi \text{ and coercive definition}) \\
&= [x : K](c'(x)) && (\beta \text{ rule}) \\
&= c' : (K)K' && (\eta \text{ rule})
\end{aligned}$$

This fact implies that without the coherence condition, in $T[\mathcal{C}]$ we can prove some result that we can't get in T. That's the reason why we define coherence before introducing the coercion application rule. And we have to use a coherent set of $\mathcal{C}$, otherwise the conservativity cannot hold.

5.3 Relation between $T[\mathcal{C}]$ and T

Now, we would like to consider the relation between system $T[\mathcal{C}]$ and T. The example in Section 4 gives us the basic idea of dealing their relation. However, it is more complicated in LF, there are several extra things we need to consider.

We need to extend the form of judgements. As we have rules of subtyping and subkinding and derivations of them, we consider the subtyping and subkinding as judgements as well.

Coercive application rule

$$(CA^*1)\frac{\Gamma \vdash f : (x : K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 : [c(k_0)/x]K'}$$

$$(CA^*2)\frac{\Gamma \vdash f = f' : (x : K)K' \qquad \Gamma \vdash k_0 = k_0' : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 = f' * k_0' : [c(k_0)/x]K'}$$

Coercive definition rule

$$(CD^*)\frac{\Gamma \vdash f : (x : K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 = f(c(k_0)) : [c(k_0)/x]K'}$$

Figure 5 The coercive application and definition rules of $T[\mathcal{C}]^*$.

So we introduce two new forms of judgements:

$$\Gamma \vdash A <_c B : \mathbf{Type} \quad \textit{and} \quad \Gamma \vdash K_1 <_c K_2$$

Since we have two new forms of judgements, we need to consider the equivalence between these judgements as well. We can extend the Definition 4 with the following two cases:

▶ **Definition 13.** (equality between the subtyping and subkinding judgements) Let S be a type theory specified in LF:

1. $(\Gamma_1 \vdash A_1 <_{c_1} B_1 : \mathbf{Type}) =_s (\Gamma_2 \vdash A_2 <_{c_2} B_2 : \mathbf{Type})$ iff $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash A_1 = A_2 : \mathbf{Type}$, $\Gamma_1 \vdash B_1 = B_2 : \mathbf{Type}$, $\Gamma \vdash c_1 = c_2 : (A_1)B_1$ are derivable in S.
2. $(\Gamma_1 \vdash K_1 <_{c_1} K_1') =_s (\Gamma_2 \vdash K_2 <_{c_2} K_2')$ iff $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$, $\Gamma_1 \vdash K_1' = K_2'$ and $\Gamma \vdash c_1 = c_2 : (K_1)K_1'$ are derivable in S.

It is straight to show the relation $=_s$ and $\sim_s$ are still equivalence relations in coercive subtyping extensions.

▶ **Theorem 14.** *Let S be a type theory with coercive subtyping specified in LF, $=_s$ and $\sim_s$ are equivalence relations.*

5.3.1 System $T[\mathcal{C}]_{0K}$

The system $T[\mathcal{C}]_{0K}$ is an intermediate system which extends T with subtyping and subkinding rules but no coercion application and definition rules. It is obtained from T by adding the new judgement form $\Gamma \vdash A <_c B : \mathbf{Type}$, $\Gamma \vdash K <_c K'$ and the inference rules in Figure 2 and 3. Since we don't have any coercion application rule in $T[\mathcal{C}]_{0K}$, the coercion judgements cannot be applied, $T[\mathcal{C}]_{0K}$ can be trivially proved as a conservative extension of T.

▶ **Proposition 15.** *System $T[\mathcal{C}]_{0K}$ is a conservative extension of system T.*

5.3.2 System $T[\mathcal{C}]^*$

We can think $T[\mathcal{C}]$ as a system obtained from $T[\mathcal{C}]_{0K}$ by adding the *coercive application* and *coercive definition* rules in Figure 4. We will extend $T[\mathcal{C}]_{0K}$ into another system $T[\mathcal{C}]^*$ with $*$ as gap holder when we apply coercive subtyping.

$T[\mathcal{C}]^*$ is the system obtained from $T[\mathcal{C}]_{0K}$ by adding the *coercive application* and *coercive definition* rules in Figure 5.

It is easy to find out that all the judgements with $*$ are not judgements in $T[\mathcal{C}]_{0K}$. It means that $T[\mathcal{C}]^*$ is conservative over $T[\mathcal{C}]_{0K}$

▶ **Proposition 16.** *$T[\mathcal{C}]^*$ is a conservative extension of $T[\mathcal{C}]_{0K}$.*

5.3.3 Relation between the systems

To describe the relation between the type system $T[\mathcal{C}]$, $T[\mathcal{C}]^*$ and $T[\mathcal{C}]_{0K}$, we introduce four algorithms Θ, Θ^*, θ_1 and θ_2 between the systems.

For two type systems T_1 and T_2, we write

$$f : T_1 \to T_2$$

if f is a function from the T_1-derivations to T_2-derivations.

We describe four algorithms, which are such functions:

$$\begin{array}{rcl} \Theta & : & T[\mathcal{C}] \to T[\mathcal{C}]_{0K} \\ \Theta^* & : & T[\mathcal{C}]^* \to T[\mathcal{C}]_{0K} \\ \theta_1 & : & T[\mathcal{C}] \to T[\mathcal{C}]^* \\ \theta_2 & : & T[\mathcal{C}]^* \to T[\mathcal{C}] \end{array}$$

The algorithms behave in the following way:

- The algorithm Θ replaces the derivations of $\Gamma \vdash K_1 <_c K_2$ in the premises of coercive rules (CA1)(CA2)(CD) by derivations of $\Gamma \vdash c : (K_1)K_2$ and replaces the coercive applications by several ordinary applications.
- The algorithm Θ^* replaces the derivations of $\Gamma \vdash K_1 <_c K_2$ in the premises of coercive rules (CA*1)(CA*2)(CD*) by derivations of $\Gamma \vdash c : (K_1)K_2$ and replaces the coercive applications by several ordinary applications.
- The algorithm θ_1 replaces coercive applications in $T[\mathcal{C}]$ derivations by coercive applications in $T[\mathcal{C}]^*$, by inserting $*$ into appropriate places.
- The algorithm θ_2 replaces coercive applications of the form $f * a$ in $T[\mathcal{C}]^*$ by coercive applications $f(a)$ in $T[\mathcal{C}]$.

We need to show that our algorithms behave in the right way, they insert the coercions into where they should be. The following property guarantees that all the coercions are inserted correctly by the algorithms:

▶ **Proposition 17.**

1. *For any derivation t in $T[\mathcal{C}]$, t and $\Theta(t)$ are equivalent derivations in $T[\mathcal{C}]$.*
2. *For any derivation t' in $T[\mathcal{C}]^*$, t' and $\Theta^*(t')$ are equivalent derivations in $T[\mathcal{C}]^*$.*

With the proposition below, we can show that $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent systems.

▶ **Proposition 18.**

1. *For any derivation t in $T[\mathcal{C}]$, t and $\theta_2(\theta_1(t))$ are equivalent derivations in $T[\mathcal{C}]$.*
2. *For any derivation t' in $T[\mathcal{C}]^*$, t' and $\theta_1(\theta_2(t'))$ are equivalent derivations in $T[\mathcal{C}]^*$.*

Finally, with the propositions above we can conclude the relations between our systems and intermediate systems. Their relations can be drawn as Figure 6.

- $T[\mathcal{C}]$ is a equivalent system of $T[\mathcal{C}]^*$.
- $T[\mathcal{C}]^*$ is a definitional extension of $T[\mathcal{C}]_{0K}$.
- $T[\mathcal{C}]_{0K}$ is a conservative extension of T.

$T[\mathcal{C}]^*$ is a definitional extension of $T[\mathcal{C}]_{0K}$ and $T[\mathcal{C}]_{0K}$ is a conservative extension of T, we would like to call that $T[\mathcal{C}]^*$ is a *D-conservative extension*[5] of T.

[5] There is a notion of *D-conservativity* in Luo's note [8], we have a different meaning with that.

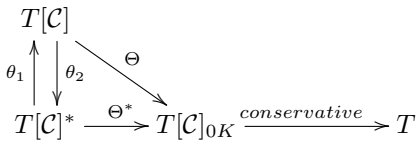


Figure 6 Relations between $T[\mathcal{C}]$, $T[\mathcal{C}]^*$, $T[\mathcal{C}]_{0K}$ and T.

▶ Remark. Although we have shown that $T[\mathcal{C}]^*$ with $*$-calculus has a more nature relationship with T, we still use $T[\mathcal{C}]$ as for description of coercive subtyping. $T[\mathcal{C}]$ itself is directly connected to important themes in the study of subtyping: *implicit coercions* and *subtyping as abbreviation*.

6 Conclusion and discussion

During the study of coercive subtyping, we find that conservativity is not enough to capture the relation between the systems. We borrow the idea of definitional extension from mathematical logic to describe the relation and formulate it in type theory. With a simple example, we demonstrate the relations and properties between a type system and its coercive subtyping extension. Although the example only consists of two basic types and one coercion, it's a nice shot containing the idea and key elements of the whole coercive subtyping extension story. We also give a sketch of the study on coercive subtyping in LF.

We hope this work presents a clear description of extending a type system with coercive subtyping and wish the notion of *definitional extension* can help with studies on other extensions in type theory. For example, *implicit syntax* of Pollack [16] is a good candidate. It starts from LEGO [10] and widely used on today's systems. We write terms with implicit arguments omitted and they are not well-typed in the system until the missing arguments have been inserted. It is not a conservative extension and we wish our notion could help to figure the exact relation out. More broadly, we can think of *elaboration*. An elaboration process maps surface language features to underlying constructions. We would like to see if elaboration is definitional extension or something more.

Acknowledgments. I would like to thank Zhaohui Luo for discussions regarding this topic and also thank to the anonymous reviewers for their valuable comments.

References

1 David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2):273–309, 2001.

2 Gilles Barthe and Maria João Frade. Constructor subtyping. In S. Doaitse Swierstra, editor, *Proceedings of Programming Languages and Systems, 8 conf. (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1999.

3 Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. In *Proceedings of Symposium on Logic in Computer Science 1987*, pages 194–204. IEEE Computer Society, 1987.

4 Martin Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

5 Stephen Kleene. *Introduction to Metamathematics*. North Holland, 1952.

6 Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
7 Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.
8 Zhaohui Luo. D-conservativity. Notes, January 2012.
9 Zhaohui Luo. Formal semantics in modern type theories with coercive subtyping. *Linguistics and Philosophye*, 35(6):491–513, 2012.
10 Zhaohui Luo and Robert Pollack. Lego proof development system: User manual, 1992.
11 Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, February 2013.
12 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
13 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer Systems and Sciences*, 17:348–375, 1978.
14 John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.
15 Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, Oxford, 1990.
16 Robert Pollack. Implicit syntax. In the preliminary Proceedings of the 1st Workshop on Logical Frameworks, 1990.
17 Sergei Soloviev and Zhaohui Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113(1–3):297–322, 2002.
18 Tao Xue. *Theory and Implementation of Coercive Subtyping*. PhD thesis, Royal Holloway, University of London, 2013.
19 Tao Xue and Zhaohui Luo. Dot-types and their implementaion. *Logical Aspects of Computational Linguistics (LACL'12). LNCS*, 7351:234–249, 2012.

A LF inference rules

Contexts and assumptions

$$\frac{}{<>\vdash \textbf{valid}} \qquad \frac{\Gamma\vdash K\ \textbf{kind} \quad x\notin FV(\Gamma)}{\Gamma, x:K\ \vdash \textbf{valid}} \qquad \frac{\Gamma, x:K, \Gamma'\vdash \textbf{valid}}{\Gamma, x:K, \Gamma'\vdash x:K}$$

$$\frac{\Gamma, \Gamma'\vdash J \quad \Gamma\vdash K\ \textbf{kind} \quad x\notin FV(\Gamma)\cup FV(\Gamma')}{\Gamma, x:K, \Gamma'\vdash J}$$

General equality rules

$$\frac{\Gamma\vdash K\ \textbf{kind}}{\Gamma\vdash K=K} \qquad \frac{\Gamma\vdash K=K'}{\Gamma\vdash K'=K} \qquad \frac{\Gamma\vdash K=K' \quad \Gamma\vdash K'=K''}{\Gamma\vdash K=K''}$$

$$\frac{\Gamma\vdash k:K}{\Gamma\vdash k=k:K} \qquad \frac{\Gamma\vdash k=k':K}{\Gamma\vdash k'=k:K} \qquad \frac{\Gamma\vdash k=k':K \quad \Gamma\vdash k'=k'':K}{\Gamma\vdash k=k'':K}$$

Equality typing rules

$$\frac{\Gamma\vdash k:K \quad \Gamma\vdash K=K'}{\Gamma\vdash k:K'} \qquad \frac{\Gamma\vdash k=k':K \quad \Gamma\vdash K=K'}{\Gamma\vdash k=k':K'}$$

$$\frac{\Gamma, x:K, \Gamma'\vdash J \quad \Gamma\vdash K=K'}{\Gamma, x:K', \Gamma'\vdash J}$$

where J is of form: $\textbf{valid}$, $K_0\ \textbf{kind}$, $k:K_0$, $K_1=K_2$ *or* $k_1=k_2:K_0$

Substitution rules

$$\frac{\Gamma, x:K, \Gamma'\ \vdash \textbf{valid} \quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\ \vdash \textbf{valid}}$$

$$\frac{\Gamma, x:K, \Gamma'\vdash K'\ \textbf{kind} \quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]K'\ \textbf{kind}} \qquad \frac{\Gamma, x:K, \Gamma'\vdash k':K' \quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]k':[k/x]K'}$$

$$\frac{\Gamma, x:K, \Gamma'\vdash K'=K'' \quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]K'=[k/x]K''} \qquad \frac{\Gamma, x:K, \Gamma'\vdash k'=k'':K' \quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]k'=[k/x]k'':[k/x]K'}$$

$$\frac{\Gamma, x:K, \Gamma'\vdash K'\ \textbf{kind} \quad \Gamma\vdash k=k':K}{\Gamma, [k/x]\Gamma'\vdash [k/x]K'=[k'/x]K'} \qquad \frac{\Gamma, x:K, \Gamma'\vdash k':K' \quad \Gamma\vdash k_1=k_2:K}{\Gamma, [k_1/x]\Gamma'\vdash [k_1/x]k'=[k_2/x]k':[k_1/x]K'}$$

The kind Type

$$\frac{\Gamma\ \vdash \textbf{valid}}{\Gamma\vdash \textbf{Type}\ \textbf{kind}} \qquad \frac{\Gamma\vdash A:\textbf{Type}}{\Gamma\vdash El(A)\ \textbf{kind}} \qquad \frac{\Gamma\vdash A=B:\textbf{Type}}{\Gamma\vdash El(A)=El(B)}$$

Dependent product kinds

$$\frac{\Gamma\vdash K\ \textbf{kind} \quad \Gamma, x:K\vdash K'\ \textbf{kind}}{\Gamma\vdash (x:K)K'\ \textbf{kind}} \qquad \frac{\Gamma\vdash K_1=K_2 \quad \Gamma, x:K_1\vdash K_1'=K_2'}{\Gamma\vdash (x:K_1)K_1'=(x:K_2)K_2'}$$

$$\frac{\Gamma, x:K\vdash k:K'}{\Gamma\vdash [x:K]k:(x:K)K'} \qquad (\eta)\frac{\Gamma\vdash K_1=K_2 \quad \Gamma, x:K_1\vdash k_1=k_2:K}{\Gamma\vdash [x:K_1]k_1=[x:K_2]k_2:(x:K_1)K}$$

$$\frac{\Gamma\vdash f:(x:K)K' \quad \Gamma\vdash k:K}{\Gamma\vdash f(k):[k/x]K'} \qquad \frac{\Gamma\vdash f=f':(x:K)K' \quad \Gamma\vdash k_1=k_2:K}{\Gamma\vdash f(k_1)=f'(k_2):[k_1/x]K'}$$

$$(\beta)\frac{\Gamma, x:K\vdash k':K' \quad \Gamma\vdash k:K}{\Gamma\vdash ([x:K]k')(k)=[k/x]k':[k/x]K'} \qquad (\xi)\frac{\Gamma\vdash f:(x:K)K' \quad x\notin FV(f)}{\Gamma\vdash [x:K]f(x)=f:(x:K)K'}$$

Figure 7 The inference rules of LF.

B Proof of Proposition 3

In a type system S specified in LF.

1. If Γ_1 is a valid context, $\vdash \Gamma_1 = \Gamma_1$
2. If $\Gamma \vdash \Gamma_1 = \Gamma_2$, then $\Gamma \vdash \Gamma_2 = \Gamma_1$.
3. If $\Gamma \vdash \Gamma_1 = \Gamma_2$ and $\Gamma \vdash \Gamma_2 = \Gamma_3$, then $\Gamma \vdash \Gamma_1 = \Gamma_3$.
4. If $\Gamma, \Gamma_1 \vdash J$ and $\Gamma \vdash \Gamma_1 = \Gamma_2$ then $\Gamma, \Gamma_2 \vdash J$. ($J$ is of form **valid**, K **kind**, $k\colon K$, $k_1 = k_2 \colon K$ or $K_1 = K_2$)

Proof. Suppose

$$\Gamma_1 \equiv x_1 : K_1, x_2 : K_2, \cdots, x_n : K_n$$
$$\Gamma_2 \equiv x_1 : M_1, x_2 : M_2, \cdots, x_n : M_n$$
$$\Gamma_3 \equiv x_1 : N_1, x_2 : N_2, \cdots, x_n : N_n$$

1. Straight by definition.
2. Since $\Gamma \vdash \Gamma_1 = \Gamma_2$, by definition we have:

$$\begin{aligned} \Gamma &\vdash K_1 = M_1; \\ \Gamma, x_1 \colon K_1, \cdots, x_{i-1} \colon K_{i-1} &\vdash K_i = M_i \quad (i = 2, \cdots, n) \end{aligned}$$

   For any $1 < i \leq n$:

$$\dfrac{\dfrac{\dfrac{\Gamma, x_1 \colon K_1, \cdots, x_{i-2} \colon K_{i-2}, x_{i-1} \colon K_{i-1} \vdash K_i = M_i \quad \Gamma, x_1 \colon K_1, \cdots, x_{i-2} \colon K_{i-2} \vdash K_{i-1} = M_{i-1}}{\Gamma, x_1 \colon K_1, \cdots, x_{i-2} \colon K_{i-2}, x_{i-1} \colon M_{i-1} \vdash K_i = M_i}}{\vdots}}{\dfrac{\Gamma, x_1 : K_1, x_2 : M_2, \cdots, x_{i-1} : M_{i-1} \vdash K_i = M_i \qquad \Gamma \vdash K_1 = M_1}{\Gamma, x_1 : M_1, x_2 : M_2, \cdots, x_{i-1} : M_{i-1} \vdash K_i = M_i}}}{\Gamma, x_1 : M_1, x_2 : M_2, \cdots, x_{i-1} : M_{i-1} \vdash M_i = K_i}$$

   and $i = 1$ is trivial with $\dfrac{\Gamma \vdash K_1 = M_1}{\Gamma \vdash M_1 = K_1}$. Hence, we have $\Gamma \vdash \Gamma_2 = \Gamma_1$ by definition.
3. Since $\Gamma \vdash \Gamma_2 = \Gamma_3$, by definition we have:

$$\begin{aligned} \Gamma &\vdash M_1 = N_1 \\ \Gamma, x_1 \colon M_1, \cdots, x_{i-1} \colon M_{i-1} &\vdash M_i = N_i \quad (i = 2, \cdots, n) \end{aligned}$$

   We have $\Gamma \vdash K_1 = M_1$, and from case 2:

$$\Gamma, x_1 \colon M_1, \cdots, x_{i-1} \colon M_{i-1} \vdash K_i = M_i \quad (i = 2, \cdots, n)$$

   In the LF, we have transitivity rules for equal kinds, so we can get:

$$\begin{aligned} \Gamma &\vdash K_1 = N_1 \\ \Gamma, x_1 \colon M_1, \cdots, x_{i-1} \colon M_{i-1} &\vdash K_i = N_i \quad (i = 2, \cdots, n) \end{aligned}$$

   For any $1 < i \leq n$:

$$\dfrac{\dfrac{\dfrac{\Gamma, x_1 \colon M_1, \cdots, x_{i-2} \colon M_{i-2}, x_{i-1} \colon M_{i-1} \vdash K_i = N_i \quad \Gamma, x_1 \colon M_1, \cdots, x_{i-2} \colon M_{i-2} \vdash M_{i-1} = K_{i-1}}{\Gamma, x_1 \colon M_1, \cdots, x_{i-2} \colon M_{i-2}, x_{i-1} \colon K_{i-1} \vdash K_i = N_i}}{\vdots}}{\dfrac{\Gamma, x_1 : M_1, x_2 : K_2, \cdots, x_{i-1} : K_{i-1} \vdash K_i = N_i \qquad \Gamma \vdash K_1 = M_1}{\Gamma, x_1 : K_1, x_2 : K_2, \cdots, x_{i-1} : K_{i-1} \vdash K_i = N_i}}$$

   We have $\Gamma \vdash \Gamma_1 = \Gamma_3$ by definition.
4.

$$\dfrac{\dfrac{\dfrac{\Gamma, x_1 \colon K_1, \cdots, x_{n-1} \colon K_{n-1}, x_n \colon K_n \vdash J \quad \Gamma, x_1 \colon K_1, \cdots, x_{n-1} \colon K_{n-1} \vdash K_n = M_n}{\Gamma, x_1 \colon K_1, \cdots, x_{n-1} \colon K_{n-1}, x_n \colon M_n \vdash J}}{\vdots}}{\dfrac{\Gamma, x_1 : K_1, x_x : M_2, \cdots, x_n : M_n \vdash J \qquad \Gamma \vdash K_1 = M_1}{\Gamma, x_1 : M_1, x_x : M_2, \cdots, x_n : M_n \vdash J}}$$

   Hence we have $\Gamma, \Gamma_2 \vdash J$.