22nd International Conference on Types for Proofs and Programs

TYPES 2016, May 23–26, 2016, Novi Sad, Serbia

Edited by

Silvia Ghilezan
Herman Geuvers
Jelena Ivetić

LIPICS

Editors

Herman Geuvers
Eindhoven University of Technology
Radboud University
herman@cs.ru.nl

Silvia Ghilezan
Mathematical Institute SASA, Belgrade
University of Novi Sad
gsilvia@uns.ac.rs

Jelena Ivetić
Faculty of Technical Sciences
University of Novi Sad
jelenaivetic@uns.ac.rs

ACM Classification 2012
Theory of computation → Type theory

ISBN 978-3-95977-065-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-065-1.

Publication date
October, 2018

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

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2016.0

LIPIcs – Leibniz International Proceedings in Informatics

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface
Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić 0:vii

List of Authors
. . . . 0:ix–0:x

Invited Papers

Mechanized Metatheory Revisited: An Extended Abstract
Dale Miller 1:1–1:16

Intersection Types and Denotational Semantics: An Extended Abstract
Simona Ronchi Della Rocca 2:1–2:7

Regular Papers

A Normalizing Computation Rule for Propositional Extensionality in Higher-Order Minimal Logic
Robin Adams, Marc Bezem, and Thierry Coquand 3:1–3:20

On Natural Deduction for Herbrand Constructive Logics II: Curry-Howard Correspondence for Markov's Principle in First-Order Logic and Arithmetic
Federico Aschieri and Matteo Manighetti 4:1–4:17

Design and Implementation of the Andromeda Proof Assistant
Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone 5:1–5:31

Realizability at Work: Separating Two Constructive Notions of Finiteness
Marc Bezem, Thierry Coquand, Keiko Nakata, and Erik Parmann 6:1–6:23

Parametricity, Automorphisms of the Universe, and Excluded Middle
Auke B. Booij, Martín H. Escardó, Peter LeFanu Lumsdaine, and Michael Shulman 7:1–7:14

Coq Support in HAHA
Jacek Chrząszcz, Aleksy Schubert, and Jakub Zakrzewski 8:1–8:26

A Shallow Embedding of Pure Type Systems into First-Order Logic
Łukasz Czajka 9:1–9:39

Permutability in Proof Terms for Intuitionistic Sequent Calculus with Cuts
José Espírito Santo, Maria João Frade, and Luís Pinto 10:1–10:27

Covering Spaces in Homotopy Type Theory
Kuen-Bang Hou (Favonia) and Robert Harper 11:1–11:16

Defining Trace Semantics for CSP-Agda
Bashar Igried and Anton Setzer 12:1–12:23

On Subtyping in Type Theories with Canonical Objects
Georgiana Elena Lungu and Zhaohui Luo 13:1–13:31

22nd International Conference on Types for Proofs and Programs (TYPES 2016).
Editors: Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A Formal Study of Boolean Games with Random Formulas as Payoff Functions
Érik Martin-Dorel and Sergei Soloviev 14:1–14:22

The Completeness of *BCD* for an Operational Semantics
Richard Statman 15:1–15:5

Preface

This issue is the post-proceedings of the 22nd International Conference on Types for Proofs and Programs, TYPES 2016, which was held in Novi Sad, Serbia, May 23-26, 2016.

The TYPES meetings are a forum to present new and on-going work on all aspects of type theory and its applications, especially in formalized and computer assisted reasoning and computer programming. The meetings have started in the late 1980s, from 1990 to 2008 they were the annual workshops of EU funded projects, from 2009 to 2015 TYPES was run as an independent conference. TYPES 2016 was the annual meeting of the COST Action CA15123 EUTypes - The European research network on types for programming and verification. Previous TYPES meetings were held in Antibes (1990), Edinburgh (1991), Baåstad (1992), Nijmegen (1993), Baåstad (1994), Turin (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal (2002), Turin (2003), Jouy-en-Josas (2004), Nottingham (2006), Cividale del Friuli (2007), Turin (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014) and Tallinn (2015).

The TYPES conference is traditionally open for work in all areas of type theory, discussing work in progress and work presented elsewhere or published. The post-proceedings is based on an open call, not restricted to the participants of the meeting.

The program of the conference comprised three invited talks by Dale Miller, Simona Ronchi della Rocca and Simon Gay, and 46 contributed talks. The meeting was attended by over 120 participants. There was a satellite event CLA 2016 - 9th Workshop Computational Logic and Applications.

The Program Committee members of TYPES 2016 were:

Thorsten Altenkirch (University of Nottingham), UK; Zena Ariola (University of Oregon), USA; Andrej Bauer (University of Ljubljana), Slovenia; Marc Bezem (University of Bergen), Norway; Małgorzata Biernacka (University of Wroclaw), Poland; Edwin Brady (University of St Andrews), UK; Thierry Coquand (University of Gothenburg), Sweden; José Espírito Santo (University of Minho), Portugal; Ken-etsu Fujita (Gunma University), Japan; Silvia Ghilezan (University of Novi Sad) (co-chair), Serbia; Hugo Herbelin (INRIA Paris-Rocquencourt), France; Jelena Ivetić (University of Novi Sad) (co-chair), Serbia; Marina Lenisa (University of Udine), Italy; Elaine Pimentel (Federal University of Rio Grande do Norte), Brazil; Andrew Polonsky (University Paris Diderot), France; Jakob Rehof (Technical University of Dortmund), Germany; Claudio Sacerdoti Coen (University of Bologna), Italy; Carsten Schürmann (IT University of Copenhagen), Denmark; Wouter Swierstra (Utrecht University), The Netherlands; Nicolas Tabareau (INRIA), France; Tarmo Uustalu (Tallinn University of Technology), Estonia.

There were 20 submissions to this open post-proceedings, which received at least two reviews by 46 anonymous referees with a second round of reviewing.

We would like to thank the authors and the program committee members for their contribution to this volume. We are grateful to the referees for their expertise which led to improvement of the content. The submissions were handled via the EasyChair platform. We thank Michael Wagner and Schloss Dagstuhl for the final editing and publishing of this volume.

Silvia Ghilezan
Herman Geuvers
Jelena Ivetić
July 2018

List of Authors

Robin Adams
Chalmers tekniska högskola, Data- och informationsteknik
Göteborg, Sweden
robinad@chalmers.se

Federico Aschieri
Institut für Diskrete Mathematik und Geometrie, Technische Universität Wien
Vienna, Austria

Andrej Bauer
University of Ljubljana
Ljubljana, Slovenia

Marc Bezem
Universitetet i Bergen, Institutt for Informatikk
Bergen, Norway
bezem@ii.uib.no

Auke B. Booij
School of Computer Science, University of Birmingham
Birmingham, UK
abb538@cs.bham.ac.uk

Jacek Chrzaszcz
University of Warsaw
Warsaw, Poland
chrzaszc@mimuw.edu.pl

Thierry Coquand
Chalmers tekniska högskola, Data- och informationsteknik
Göteborg, Sweden
coquand@chalmers.se

Lukasz Czajka
DIKU, University of Copenhagen
Copenhagen, Denmark
luta@di.ku.dk

Martín H. Escardó
School of Computer Science, University of Birmingham
Birmingham, UK
m.escardo@cs.bham.ac.uk

José Espírito Santo
Centro de Matemática, Universidade do Minho
Braga, Portugal

Maria João Frade
HASLab/INESC TEC and Universidade do Minho
Braga, Portugal

Gaëtan Gilbert
École Normale Supérieure de Lyon
Lyon, France

Robert Harper
Computer Science Department, Carnegie Mellon University
Pittsburgh, PA, USA
rwh@cs.cmu.edu

Philipp G. Haselwarter
University of Ljubljana
Ljubljana, Slovenia

Kuen-Bang Hou (Favonia)
Department of Computer Science and Engineering, University of Minnesota Twin Cities
Minneapolis, MN, USA
favonia@umn.edu

Bashar Igried
The Hashemite University, Faculty of Prince Al-Hussein Bin Abdallah II for Information Technology
Zarqa, Jordan
bashar.igried@yahoo.com

Peter LeFanu Lumsdaine
Mathematics Department, Stockholm University
Stockholm, Sweden
p.l.lumsdaine@math.su.se

Georgiana Elena Lungu
Department of Computer Science, Royal Holloway, University of London
London, UK
georgiana.lungu.2013@live.rhul.ac.uk

Zhaohui Luo
Department of Computer Science, Royal Holloway, University of London
London, UK
Zhaohui.Luo@rhul.ac.uk

Matteo Manighetti
Institut für Diskrete Mathematik und Geometrie, Technische Universität Wien
Vienna, Austria

Érik Martin-Dorel
Lab. IRIT, Univ. of Toulouse, CNRS
Toulouse, France
erik.martin-dorel@irit.fr

Dale Miller
Inria and LIX, École Polytechnique
Palaiseau, France

Keiko Nakata
SAP Innovation Center Network
Potsdam, Germany

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

Luís Pinto
Centro de Matemática, Universidade do Minho
Braga, Portugal

Matija Pretnar
University of Ljubljana
Ljubljana, Slovenia

Simona Ronchi Della Rocca
Università degli Studi di Torino, dept Computer Science
Torino, Italy
ronchi@di.unito.it

Aleksy Schubert
University of Warsaw
Warsaw, Poland
alx@mimuw.edu.pl

Anton Setzer
Swansea University, Dept. of Computer Science
Swansea, Wales, UK
a.g.setzer@swansea.ac.uk

Michael Shulman
Department of Mathematics, University of San Diego
San Diego, USA
shulman@sandiego.edu

Sergei Soloviev
Lab. IRIT, Univ. of Toulouse, CNRS
Toulouse, France
sergei.soloviev@irit.fr

Richard Statman
Carnegie Mellon University, Department of Mathematical Sciences
Pittsburgh, PA, USA
statman@cs.cmu.edu

Christopher A. Stone
Harvey Mudd College
Claremont, CA, USA

Jakub Zakrzewski
University of Warsaw
Warsaw, Poland
j.zakrzewski@mimuw.edu.pl

Mechanized Metatheory Revisited: An Extended Abstract

Dale Miller
Inria and LIX, École Polytechnique, Palaiseau, France
https://orcid.org/0000-0003-0274-4954

Abstract

Proof assistants and the programming languages that implement them need to deal with a range of linguistic expressions that involve bindings. Since most mature proof assistants do not have built-in methods to treat this aspect of syntax, many of them have been extended with various packages and libraries that allow them to encode bindings using, for example, de Bruijn numerals and nominal logic features. I put forward the argument that bindings are such an intimate aspect of the structure of expressions that they should be accounted for directly in the underlying programming language support for proof assistants and not added later using packages and libraries. One possible approach to designing programming languages and proof assistants that directly supports such an approach to bindings in syntax is presented. The roots of such an approach can be found in the *mobility* of binders between term-level bindings, formula-level bindings (quantifiers), and proof-level bindings (eigenvariables). In particular, by combining Church's approach to terms and formulas (found in his Simple Theory of Types) and Gentzen's approach to sequent calculus proofs, we can learn how bindings can declaratively interact with the full range of logical connectives and quantifiers. I will also illustrate how that framework provides an intimate and semantically clean treatment of computation and reasoning with syntax containing bindings. Some implemented systems, which support this intimate and built-in treatment of bindings, will be briefly described.

2012 ACM Subject Classification Theory of computation → Proof theory, Theory of computation → Automated reasoning, Theory of computation → Higher order logic

Keywords and phrases mechanized metatheory, mobility of binders, lambda-tree syntax, higher-order abstract syntax

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.1

Category Invited Paper

Related Version A full version of this paper, titled "Mechanized metatheory revisited", is available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/pubs.html`.

Funding This work was funded by the ERC Advanced Grant ProofCert.

Foreword This extended abstract is a non-technical look at the mechanization of formalized metatheory. While this paper may be provocative at times, I mainly intend to shine light on a slice of literature that is developing a coherent and maturing approach to mechanizing metatheory.

1 Mechanization of metatheory

A decade ago, the POPLmark challenge suggested that the theorem proving community had tools that were close to being usable by programming language researchers to formally prove properties of their designs and implementations. The authors of the POPLmark challenge looked at existing practices and systems and urged the developers of proof assistants to make improvements to existing systems.

Our conclusion from these experiments is that the relevant technology has developed *almost* to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research – mechanized metatheory for the masses. [5]

In fact, a number of research teams have used proof assistants to formally prove significant properties of programming language related systems. Such properties include type preservation, determinancy of evaluation, and the correctness of an OS microkernel and of various compilers: see, for example, [41, 42, 44, 59].

As noted in [5], the poor support for binders in syntax was one problem that held back proof assistants from achieving even more widespread use by programming language researchers and practitioners. In recent years, a number of enhancements to programming languages and to proof assistants have been developed for treating bindings. These go by names such as locally nameless [12, 76], nominal reasoning [3, 14, 69, 83], and parametric higher-order abstract syntax [15]. Some of these approaches involve extending underlying programming language implementations while the others do not extend the proof assistant or programming language but provide various packages, libraries, and/or abstract datatypes that attempt to orchestrate various issues surrounding the syntax of bindings. In the end, nothing canonical seems to have arisen: see [4, 68] for detailed comparisons.

2 An analogy: concurrency theory

While extending mature proof assistants (such as Coq, HOL, and Isabelle) with facilities to handle bindings is clearly possible, it seems desirable to consider directly the computational principles surrounding the treatment of binding in syntax independent of a given programming language. Developments in programming design has, of course, run into similar situations where there was a choice to be made between accounting for features by extending existing programming languages or by the development of new programming languages. Consider, for example, the following analogous (but more momentous) situation.

Historically speaking, the first high-level, mature, and expressive programming languages to be developed were based on sequential computation. When those languages were forced to deal with concurrency, parallelism, and distributed computing, they were augmented with, say, thread packages and remote procedure calls. Earlier pioneers of computer programming languages and systems – e.g., Dijkstra, Hoare, Milner – saw concurrency and communications not as incremental improvements to existing imperative languages but as a new paradigm deserving a separate study. The concurrency paradigm required a fresh and direct examination and in this respect, we have seen a great number of concurrency frameworks appear: e.g., Petri nets, CSP, CCS, IO-automata, and the π-calculus. Given the theoretical results and understanding that have flowed from work on these and related calculi, it has been possible to find ways for conventional programming languages to make accommodations within the concurrency and distributed computing settings. Such understanding and accommodations were not likely to flow from clever packages added to programming languages: new programming principles from the theory of concurrency and distributed computing were needed.

Before directly addressing some of the computational principles behind bindings in syntax, it seems prudent to critically examine the conventional design of a wide range of proof assistants. (The following section updates a similar argument found in [52].)

3 Dropping mathematics as an intermediate

Almost all ambitious theorem provers in use today follow the following two step approach to reasoning about computation.

Step 1: *Implement mathematics.* This step is achieved by picking a general, well understood formal system. Common choices are first-order logic, set theory, higher-order logic [16, 36], or some foundation for constructive mathematics, such as Martin-Löf type theory [18, 19, 45].

Step 2: *Reduce reasoning about computation to mathematics.* Computation is generally encoded via some model theoretic semantics (such as denotational semantics) or as an inductive definition over an operational semantics.

A key methodological element of this proposal is that we shall drop mathematics as an intermediate and attempt to find more direct and intimate connections between computation, reasoning, and logic. The main problem with having mathematics in the middle seems to be that many aspects of computation are rather "intensional" but a mathematical treatment requires an extensional encoding. The notion of *algorithm* is an example of this kind of distinction: there are many algorithms that can compute the same function (say, the function that sorts lists). In a purely extensional treatment, it is functions that are represented directly and algorithm descriptions that are secondary. If an intensional default can be managed instead, then function values are secondary (usually captured via the specification of evaluators or interpreters).

For a more explicit example, consider whether or not the formula $\forall w_i.\ \lambda x.x \neq \lambda x.w$ is a theorem. In a setting where λ-abstractions denote functions (the usual extensional treatment), we have not provided enough information to answer this question: in particular, this formula is true if and only if the domain type i is not a singleton. If, however, we are in a setting where λ-abstractions denote syntactic expressions, then it is sensible for this formula to be provable since no (capture avoiding) substitution of an expression of type i for the w in $\lambda x.w$ can yield $\lambda x.x$.

For a more significant example, consider the problem of formalizing the metatheory of bisimulation-up-to [56, 72] for the π-calculus [57]. Such a metatheory can be used to allow people working in concurrent systems to write hopefully small certificates (actual bisimulations-up-to) in order to guarantee that bisimulation holds (usually witnessed directly by only infinite sets of pairs of processes). In order to employ the Coq theorem prover, for example, to attack such metatheory, Coq would probably need to be extended with packages in two directions. First, a package that provides flexible methods for doing coinduction following, say, the Knaster-Tarski fixed point theorems, would be necessary. Indeed, such a package has been implemented and used to prove various metatheorems surrounding bisimulation-up-to (including the subtle metatheory surrounding weak bisimulation) [11, 70, 71]. Second, a package for the treatment of bindings and names that are used to describe the operational semantics of the π-calculus would need to be added. Such packages exist (for example, see [6]) and, when combined with treatments of coinduction, may allow one to make progress on the metatheory of the π-calculus. Recently, the Hybrid systems [27] has shown a different way to incorporate both induction, coinduction, and binding into a Coq (and Isabelle) implementation. Such an approach could be seen as one way to implement this metatheory task on top of an established formalization of mathematics.

There is another approach that seeks to return to the most basic elements of logic by reconsidering the notion of terms (allowing them to have binders as primitive features) and the notion of logical inference rules so that coinduction can be seen as, say, the de Morgan (and proof theoretic) dual to induction. In that approach, proof theory principles can be identified in that enriched logic with least and greatest fixed points [7, 47, 58] and with a treatment of bindings [81, 32]. Such a logic has been given a model-checking-style implementation [9] and is the basis of the Abella theorem prover [8, 31]. Using such implementations, the π-calculus has been implemented, formalized, and analyzed in some detail [80, 79] including some of the metatheory of bisimulation-up-to for the π-calculus [13].

I will now present some foundational principles in the treatment of bindings that are important to accommodate directly, even if we cannot immediately see how those principles might fit into existing mature programming languages and proof assistants.

4 How abstract is your syntax?

Two of the earliest formal treatments of the syntax of logical expressions were given by Gödel [35] and Church [16] and, in both of these cases, their formalization involved viewing formulas as strings of characters. Clearly, such a view of logical expressions contains too much information that is not semantically meaningful (e.g., white space, infix/prefix distinctions, parenthesis) and does not contain explicitly semantically relevant information (e.g., the function-argument relationship). For this reason, those working with syntactic expressions generally parse such expressions into *parse trees*: such trees discard much that is meaningless (e.g., the infix/prefix distinction) and records directly more meaningful information (e.g., the child relation denotes the function-argument relation). One form of "concrete nonsense" generally remains in parse trees since they traditionally contain the *names* of bound variables.

One way to get rid of bound variable names is to use de Bruijn's nameless dummy technique [21] in which (non-binding) occurrences of variables are replaced by positive integers that count the number of bindings above the variable occurrence through which one must move in order to find the correct binding site for that variable. While such an encoding makes the check for α-conversion easy, it can greatly complicate other operations that one might want to do on syntax, such as substitution, matching, and unification. While all such operations can be supported and implemented using the nameless dummy encoding [21, 43, 61], the complex operations on indexes that are needed to support those operations clearly suggests that they are best dealt within the implementation of a framework and not in the framework itself.

The following four principles about the treatment of bindings in syntax will guide our further discussions.

Principle 1: The names of bound variables should be treated in the same way we treat white space: they are artifacts of how we write expressions and they have no semantic content.

Of course, the name of variables are important for parsing and printing expressions (just as is white space) but such names should not be part of the meaning of an expression. This first principle simply repeats what we stated earlier. The second principle is a bit more concrete.

Principle 2: There is "one binder to ring them all."[1]

With this principle, we are adopting Church's approach [16] to binding in logic, namely, that one has only λ-abstraction and all other bindings are encoded using that binder. For example, the universally quantified expression $(\forall x.\, B\, x)$ is actually broken into the expression

[1] A scrambling of J. R. R. Tolkien's "One Ring to rule them all, ... and in the darkness bind them."

$(\forall(\lambda x.\, B\, x))$, where $\forall$ is treated as a constant of higher-type. Note that this latter expression is η-equivalent to $(\forall\ B)$ and universal instantiation of that quantified expression is simply the result of using λ-normalization on the expression $(B\ t)$. In this way, many details about quantifiers can be reduced to details about λ-terms.

Principle 3: There is no such thing as a free variable.

This principle is taken from Alan Perlis's epigram 47 [63]. By accepting this principle, we recognize that bindings are never dropped to reveal a free variable: instead, we will ask for bindings to *move*. This possibility suggests the main novelty in this list of principles.

Principle 4: Bindings have *mobility* and the equality theory of expressions must support such mobility [51, 53].

Since the other principles are most likely familiar to the reader, I will now describe this last principle in more detail.

5 Mobility of bindings

Since typing rules are a common operation in metatheory, I illustrate the notion of binding mobility in that setting. In order to specify untyped λ-terms (to which one might attribute a simple type via an inference), we introduce a (syntactic) type *tm* and two constants

$$abs\colon (tm \to tm) \to tm \qquad \text{and} \qquad app\colon tm \to tm \to tm.$$

Untyped λ-terms are encoded as terms of type *tm* using the translation define as

$$\lceil x \rceil = x, \qquad \lceil \lambda x.t \rceil = (abs\ (\lambda x.\lceil t \rceil)), \qquad \text{and} \qquad \lceil (t\ s) \rceil = (app\ \lceil t \rceil\ \lceil s \rceil).$$

The first clause here indicates that bound variables in untyped λ-terms are mapped to bound variables in the encoding. For example, the untyped λ-term $\lambda w.ww$ is encoded as $(abs\ \lambda w.\ app\ w\ w)$. This translation has the property that it maps bijectively α-equivalence classes of untyped λ-terms to $\alpha\beta\eta$-equivalence classes of simply typed λ-terms of type *tm*.

In order to satisfy Principle 3 above, we shall describe a Gentzen-style sequent as a triple $\Sigma : \Delta \vdash B$ where B is the succedent (a formula), Δ is the antecedent (a multiset of formulas), and Σ a signature, that is, a list of variables that are formally bound over the scope of the sequent. Thus all free variables in the formulas in $\Delta \cup \{B\}$ are bound by Σ. Gentzen referred to the variables in Σ as *eigenvariables* (although he did not consider them as binders over sequents).

The following inference rule is a familiar rule.

$$\frac{\Sigma : \Delta, \mathit{typeof}\ x\ (\mathit{int} \to \mathit{int}) \vdash C}{\Sigma : \Delta, \forall\tau(\mathit{typeof}\ x\ (\tau \to \tau)) \vdash C}\ \forall L$$

This rule states (when reading it from conclusions to premise) that if the symbol x can be attributed the type $\tau \to \tau$ for all instances of τ, then it can be assumed to have the type $\mathit{int} \to \mathit{int}$. Thus, bindings can be instantiated (the $\forall\tau$ is removed by instantiation). On the other hand, consider the following inferences.

$$\cfrac{\cfrac{\Sigma, x : \Delta, \mathit{typeof}\ \lceil x \rceil\ \tau \vdash \mathit{typeof}\ \lceil B \rceil\ \beta}{\Sigma : \Delta \vdash \forall x(\mathit{typeof}\ \lceil x \rceil\ \tau \supset \mathit{typeof}\ \lceil B \rceil\ \tau')}\ \forall R}{\Sigma : \Delta \vdash \mathit{typeof}\ \lceil \lambda x.B \rceil\ (\tau \to \tau')}$$

These inferences illustrate how bindings can, instead, *move* during the construction of a proof. In this case, the term-level binding for x in the lower sequent can be seen as moving to the formula-level binding for x in the middle sequent and then to the proof-level binding (as an eigenvariable) for x in the upper sequent. Thus, a binding is not lost or converted to a "free variable": it simply moves.

The mobility of bindings needs to be supported by the equality theory of expressions. Clearly, equality already includes α-conversion by Property 1. We also need a small amount of β-conversion. If we rewrite this last inference rule using the definition of the $\lceil\cdot\rceil$ translation, we have the inference figure.

$$\dfrac{\dfrac{\Sigma, x : \Delta, \mathit{typeof}\ x\ \tau \vdash \mathit{typeof}\ (Bx)\ \tau'}{\Sigma : \Delta \vdash \forall x(\mathit{typeof}\ x\ \tau \supset \mathit{typeof}\ (Bx)\ \tau')}\ \forall R}{\Sigma : \Delta \vdash \mathit{typeof}\ (\mathit{abs}\ B)\ (\tau \to \tau')}$$

Note that here B is a variable of arrow type $tm \to tm$ and that instances of these inference figures will create an instance of $(B\ x)$ that may be a β-redex. As I now argue, that β-redex has a limited form. First, observe that B is a schema variable that is implicitly universally quantified around this inference rule: if one formalizes this approach to type inference in, say, λProlog, one would write a specification similar to the formula

$$\forall B \forall \tau \forall \tau' [\forall x(\mathit{typeof}\ x\ \tau \supset \mathit{typeof}\ (Bx)\ \tau') \supset \mathit{typeof}\ (\mathit{abs}\ B)\ (\tau \to \tau')].$$

Second, any closed instance of $(B\ x)$ that is a β-redex is such that the argument x is not free in the instance of B: this is enforced by the nature of (quantificational) logic since the scope of B is outside the scope of x. Thus, the only form of β-conversion that is needed to support this notion of binding mobility is the so-called β_0-conversion rule [50]: $(\lambda x.t)x = t$ or equivalently (in the presence of α-conversion) $(\lambda y.t)x = t[x/y]$, provided that x is not free in $\lambda y.t$.

Given that β_0-conversion is such a simple operation, it is not surprising that higher-order pattern unification, which simplifies higher-order unification to a setting only needing α, β_0, and η conversion, is decidable and unitary [50]. For this reason, matching and unification can be used to help account for the mobility of binding. Note also that there is an elegant symmetry provided by binding and β_0-reduction: if t is a term over the signature $\Sigma \cup \{x\}$ then $\lambda x.t$ is a term over the signature Σ and, conversely, if $\lambda x.s$ is a term over the signature Σ then the β_0-reduction of $((\lambda x.s)\ y)$ is a term over the signature $\Sigma \cup \{y\}$.

To illustrate how β_0-conversion supports the mobility of binders, consider how one specifies the following rewriting rule: given a conjunction of universally quantified formulas, rewrite it to be the universal quantification of the conjunction of formulas. In this setting, we would write something like:

$$(\forall(\lambda x.A\ x)) \wedge (\forall(\lambda x.B\ x)) \mapsto (\forall(\lambda x.(A\ x \wedge B\ x))).$$

To rewrite an expression such as $(\forall \lambda z(p\ z\ z)) \wedge (\forall \lambda z(q\ a\ z))$ (where p, q, and a are constants) we first need to use β_0-expansion to get the expression

$$(\forall \lambda z((\lambda w.(p\ w\ w))z)) \wedge (\forall \lambda z((\lambda w.(q\ a\ w))z))$$

At this point, the pattern variables A and B in the rewriting rule can now be instantiated by the *closed* terms $\lambda w.(p\ w\ w)$ and $\lambda w.(q\ a\ w)$, respectively, which yields the expression

$$(\forall(\lambda x.((\lambda w.(p\ w\ w))\ x \wedge (\lambda w.(q\ a\ w))\ x))).$$

Finally, a β_0-contraction yields the expected expression $(\forall(\lambda x.(p\ x\ x) \wedge (q\ a\ x)))$. Note that at no time did a bound variable become unbound. Since pattern unification incorporates β_0-conversion, such rewriting can be accommodated simply by calls to such unification.

The analysis of these four principles above do not imply that full β-conversion is needed to support them. Clearly, full β-conversion will implement β_0-conversion and several systems (which we shall speak about more below) that support λ-tree syntax do, in fact, implement β-conversion. Systems that only implement β_0-conversion have only been described in print. For example, the L_λ logic programming language of [50] was restricted so that proof search could be complete while only needing to do β_0-conversion. The π_I-calculus (the π-calculus with internal mobility [74]) can also be seen as a setting where only β_0-conversion is needed [53].

6 Logic programming provides a framework

As the discussion above suggests, quantificational logic using the proof-search model of computation can capture all four principles listed in the previous section. While it might be possible to account for these principles also in, say, a functional programming language (a half-hearted attempt at such a design was made in [49]), the logic programming paradigm supplies an appropriate framework for satisfying all these properties. Such a framework is available using the higher-order hereditary Harrop [54] subset of an intuitionistic variant of Church's Simple Theory of Types [16]: λProlog [53] is a logic programming language based on that logic and implemented by the Teyjus compiler [73] and the ELPI interpreter [24].

The use of logic programming principles in proof assistants pushes against usual practice: since the first LCF prover [37], many (most?) proof assistants have had intimate ties to functional programming. For example, such theorem provers are often implemented using functional programming languages: in fact, the notion of LCF tactics and tacticals was originally designed and illustrated using functional programming principles [37]. Also, such provers frequently view proofs constructively and can output the computational content of proofs as functional programs [10].

I argue here that a framework based on logic programming principles might be more appropriate for mechanizing metatheory than one based on functional programming principles. Note that the arguments below do not lead to the conclusion that first-order logic programming languages, such as Prolog, are appropriate for metalevel reasoning: direct support for λ-abstractions and quantifiers (as well as hypothetical reasoning) are critical and are not supported in first-order logic programming languages. Also, I shall focus on the *specification* of mechanized metatheory tasks and not on their *implementation*: it is completely possible that logic programming principles are used in specifications while a functional programming language is used to implement that specification language (for example, Teyjus and Abella are both implemented in OCaml).

6.1 Expressions versus values

In logic programming, (closed) terms denote themselves and only themselves (in the sense of free algebra). It often surprises people that in Prolog, the goal `?- 3 = 1 + 2` fails, but the expression that is the numeral `3` and the expression `1 + 2` are, of course, different expressions. The fact that they have the same value is a secondary calculation (performed in Prolog using the `is` predicate). Functional programming, however, fundamentally links expressions and values: the value of an expression is the result of applying some evaluation strategy (e.g., call-by-value) to an expression. Thus the value of both `3` and `1 + 2` is `3` and these two expressions are, in fact, equated. Of course, one can easily write datatypes in functional programming languages that denote only expressions: datatypes for parse trees are such an example. However, the global notion that expressions denote values is particularly problematic when expressions denote λ-abstractions. The value of such expressions in functional programming is trivial and immediate: such values simply denote a function (a closure). In the logic programming setting, however, an expression that is a λ-abstraction is just another expression: following the principles stated in Section 4, equality of two such expressions needs to be based on the rather simple set of conversion rules α, β_0, and η. The λ-abstraction-as-expression aspect of logic programming is one of that paradigm's major advantages for the mechanization of metatheory.

6.2 Syntactic types

Given the central role of expressions (and not values), types in logic programming are better thought of as denoting *syntactic categories*. That is, such syntactic types are useful for distinguishing, say, encodings of types from terms from formula from proofs or program expressions from commands from evaluation contexts. For example, the *typeof* specification in Section 5 is a binary relation between the syntactic categories *tm* (for untyped λ-terms) and, say, *ty* (for simple type expression). The logical specification of the *typeof* predicate might attribute integer type or list type to different expressions via clauses such as

$$\forall T\colon tm\ \forall L\colon tm\ \forall\tau\colon ty\ [\mathit{typeof}\ T\ \tau \supset \mathit{typeof}\ L\ (\mathit{list}\ \tau) \supset \mathit{typeof}\ (T :: L)\ (\mathit{list}\ \tau)].$$

Given our discussion above, it seems natural to propose that if τ and τ' are both syntactic categories, then $\tau \to \tau'$ is a new syntactic category that describes objects of category τ' with a variable of category τ abstracted. For example, if o denotes the category of formulas (a la [16]) and *tm* denotes the category of terms, then $tm \to o$ denotes the type of term-level abstractions over formulas. As we have been taught by Church, the quantifiers $\forall$ and $\exists$ can then be seen as constructors that take expressions of syntactic category $tm \to o$ to formulas: that is, these quantifiers are given the syntactic category $(tm \to o) \to o$.

6.3 Substitution lemmas for free

Consider an attempt to prove the sequent

$$\Sigma : \Delta \vdash \mathit{typeof}\ (\mathit{abs}\ R)\ (\tau \to \tau')$$

where the assumptions (the theory) contains only one rule for proving such a statement, such as the clause used in the discussion of Section 5. Since the introduction rules for $\forall$ and $\supset$ are invertible, the sequent above is provable if and only if the sequent

$$\Sigma, x : \Delta, \mathit{typeof}\ x\ \tau \vdash \mathit{typeof}\ (R\ x)\ \tau'$$

is provable. Given that we are committed to using a proper logic (such as higher-order intuitionistic logic), it is the case that modus ponens is valid and that instantiating an eigenvariable in a provable sequent yields a provable sequent. In this case, the sequent

$$\Sigma : \Delta, \mathit{typeof}\ N\ \tau \vdash \mathit{typeof}\ (R\ N)\ \tau'$$

must be provable (for N a term of syntactic type *tm* all of whose free variables are in Σ). Thus, we have just shown, using nothing more than rather minimal assumptions about the specification of *typeof* (and formal properties of logic) that if $\Sigma : \Delta \vdash \mathit{typeof}\ (\mathit{abs}\ B)\ (\tau \to \tau')$

and $\Sigma : \Delta \vdash \mathit{typeof}\ N\ \tau$ then $\Sigma : \Delta \vdash \mathit{typeof}\ (B\ N)\ \tau'$. (Of course, instances of the term $(B\ N)$ are β-redexes and the reduction of such redexes result in the substitution of N into the bound variable of the term that instantiates B.) Such lemmas about substitutions are common and often difficult to prove [85]: in this setting, this lemma is essentially an immediate consequent of using logic and logic programming principles [8, 46]. In this way, Gentzen's cut-elimination theorem (the formal justification of modus ponens) can be seen as the mother of all substitution lemmas. The Abella theorem prover's implementation of the *two-level logic* approach to reasoning about computation [33, 48] makes it possible to employ the cut-elimination theorem in exactly the style illustrated above.

6.4 Dominance of relational specifications

Another reason that logic programming can make a good choice for metatheoretic reasoning systems is that logic programming is based on relations (not functions) and that metatheoretic specifications are often dominated by relations. For example, the typing judgment describe in the Section 5 is a relation. Similarly, both small step (SOS) and big step (natural semantics) approaches to operational semantics describe evaluation, for example, as a relation. Occasionally, specified relations – typing or evaluation – describe a partial function but that is generally a result proved about the relation.

A few logic programming-based systems have been used to illustrate how typing and operational semantic specifications can be animated. The core engine of the Centaur project, called Typol, used Prolog to animate metatheoretic specifications [17] and λProlog has been used to provide convincing and elegant specifications of typing and operational semantics for expressions involving bindings [2, 53].

6.5 Dependent typing

The typing that has been motivated above is rather simple: one takes the notions of syntactic types as syntactic category – e.g., programs, formulas, types, terms, etc – and adds the arrow type constructor to denote abstractions of one syntactic type over another one. Since typing is, of course, an open-ended concept, it is completely possible to consider any number of ways to refine types. For example, instead of saying that a given expression denotes a term (that is, the expression has the syntactic type for terms), one could instead say that such an expression denotes, for example, a function from integers to integers. For example, the typing judgment $t : \mathit{tm}$ ("t denotes a term") can be refined to $t : \mathit{tm}\ (\mathit{int} \to \mathit{int})$ ("t denotes a term of type $\mathit{int} \to \mathit{int}$). Such richer types are supported (and generalized) by the *dependent type* paradigm [20, 38] and given a logic programming implementation in, for example, Twelf [64, 66].

Most dependently typed λ-calculi come with a fixed notion of typing and with a fixed notion of proof (natural deduction proofs encoded as typed λ-terms). The reliance described here on logical connectives and relations is expressive enough to specify dependently typed frameworks [26, 77, 78] but it is not committed to only that notion of typing and proof.

7 λ-tree syntax

The term *higher-order abstract syntax* (HOAS) was originally defined as an approach to syntax that used "a simply typed λ-calculus enriched with products and polymorphism" [65]. A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the meta-language" [66]. The term HOAS is problematic for a number of reasons. First, it seems that few, if any, researchers use this term in a setting that includes products and polymorphism (although simple and dependently typed λ-calculus are often used). Second, since the metalanguage (often the programming language) can vary a great deal, the resulting notion of HOAS can vary similarly, including the case where HOAS is a representation of syntax that incorporates *function spaces* on expressions [22, 39]. Third, the adjective higher-order seems inappropriate here: in particular, the equality (and unification) of terms discussed in Section 5 is completely valid without reference to typing. If there are no types, what exactly is "higher-order"? For these reasons, the term "λ-tree syntax" [8, 53], with its obvious parallel to the term "parse tree syntax," has been introduced as a more appropriate term for the approach to syntactic representation described here.

While λ-tree syntax can be seen as a kind of HOAS (using the broad definition of HOAS given in [66]), there is little connections between λ-tree syntax and the problematic aspects of HOAS that arise when the latter uses function spaces to encode abstractions. For example, there are frequent claims that structural induction and structural recursive definitions are either difficult, impossible, or semantically problematic for HOAS: see, for example, [29, 39, 40]. When we consider specifically λ-tree syntax, however, induction (and coinduction) and structural recursion in the λ-tree setting have been given proof theoretic treatments and implementations.

8 Reasoning with λ-tree syntax

Proof search (logic programming) style implementations of specifications can provide simple forms of metatheory reasoning. For example, given the specification of typing, both type checking and type inference are possible to automate using unification and backtracking search. Similarly, a specification of, say, big step evaluation can be used to provide a symbolic evaluator for at least simple expressions [17].

There is, however, much more to mechanizing metatheory than performing unification and doing logic programming-style search. One must also deal with negations (difficult for straightforward logic programming engines): for example, one wants to prove that certain terms do not have simple types: for example,

$$\vdash \neg\, \exists \tau : ty.\ typeof\ (abs\ \lambda x\ (app\ x\ x))\ \tau.$$

Proving that a certain relation actually describes a (partial or total) function has proved to be an important kind of metatheorem to prove: the Twelf system [66] is able to automatically prove many of the simpler forms of such metatheorems. Additionally, one should also deal with induction and coinduction and be able to reason directly about, say, bisimulation of π-calculus expressions as well as confluence of λ-conversion.

In recent years, several researchers have developed two extensions to logic and proof theory that have made it possible to reason in rich and natural ways about expressions containing bindings. One of these extensions involved a proof theory for least and greatest fixed points: results from [47, 82] have made it possible to build automated and interactive inductive and coinductive theorem provers in a simple, relational setting. Another extension [32, 55] introduced the ∇-quantifier which allows logic to reason in a rich and natural way with bindings: in terms of mobility of bindings, the ∇-quantifier provides an additional formula-level and proof-level binder, thereby enriching the expressiveness of quantificational logic.

Given these developments in proof theory, it has been possible to build both an interactive theorem prover, called Abella [8, 30], and an automatic theorem prover, called Bedwyr [9], that unfolds fixed points in a style similar to a model checker. These systems have successfully been able to prove a range of metatheoretic properties about the λ-calculus and the π-calculus [1, 8, 81]. The directness and naturalness of the encoding for the π-calculus bisimulation is evident in the fact that simply adding the excluded middle on name equality changes the interpretation of that one definition from open bisimulation to late bisimulation [81].

Besides the Abella, Bedwyr, and Twelf system mentioned above, there are a number of other implemented systems that support some or all aspects of λ-tree syntax: these include Beluga [67], Hybrid [27], Isabelle [62], Minlog [75], and Teyjus [60]. See [28] for a survey and comparison of several of these systems.

The shift from conventional proof assistants based on functional programming principles to assistants based on logic programming principles does disrupt a number of aspects of proof assistants. For example, when computations are naturally considered as functional, it seems that there is a lost of expressiveness and effectiveness if one must write those specifications using relations. Recent work shows, however, that when a relation actually encodes a function, it is possible to use the proof search framework to actually compute that function [34]. A popular feature of many proof assistants is the use of tactics and tacticals, which have been implemented using functional programs since their introduction [37]. There are good arguments, however, that those operators can be given elegant and natural implementations using (higher-order) logic programs [23, 25, 53]. The disruptions that result from such a shift seem well worth exploring.

9 Conclusions

I have argued that parsing concrete syntax into parse trees does not yield a sufficiently abstract representation of expressions: the treatment of bindings should be made more abstract. I have also described and motivated the λ-tree syntax approach to such a more abstract framework. For a programming language or proof assistant to support this level of abstraction in syntax, equality of syntax must be based on α and β_0 (at least) and must allow for the mobility of binders from within terms to within formulas (i.e., quantifiers) to within proofs (i.e., eigenvariables). I have also argued that the logic programming paradigm – broadly interpreted – provides an elegant and high-level framework for specifying both computation and deduction involving syntax containing bindings. This framework is offered up as an alternative to the more conventional approaches to mechanizing metatheory using formalizations based on more conventional mathematical concepts. While the POPLmark challenge was based on the assumption that increments to existing provers will solve the problems surrounding the mechanization of metatheory, I have argued and illustrated here that we need to make a significant shift in the underlying paradigm that has been built into today's most mature proof assistants.

References

1 Beniamino Accattoli. Proof pearl: Abella formalization of lambda calculus cube property. In Chris Hawblitzel and Dale Miller, editors, *Second International Conference on Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2012.

2 Andrew W. Appel and Amy P. Felty. Polymorphic lemmas and definitions in λProlog and Twelf. *Theory and Practice of Logic Programming*, 4(1-2):1–39, 2004. doi:10.1017/S1471068403001698.

3 Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP)*, pages 69–77, Seattle, WA, USA, 2006.

4 Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. Technical Report MS-CIS-09-06, University of Pennsylvania, 2009.

5 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

6 Brian E. Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq: (extended abstract). *Electr. Notes Theor. Comput. Sci*, 174(5):69–77, 2007. doi:10.1016/j.entcs.2007.01.028.

7 David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), 2012. doi:10.1145/2071368.2071370.

8 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014. doi:10.6092/issn.1972-5787/4650.

9 David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, number 4603 in Lecture Notes in Artificial Intelligence, pages 391–397, New York, 2007. Springer. doi:10.1007/978-3-540-73595-3_28.

10 Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006. doi:10.1007/s11225-006-6604-5.

11 Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 457–468. ACM, 2013.

12 Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011. doi:10.1007/s10817-011-9225-2.

13 Kaustuv Chaudhuri, Matteo Cimini, and Dale Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, pages 157–166, Mumbai, India, 2015. ACM. doi:10.1145/2676724.2693170.

14 James Cheney and Christian Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5):1–47, 2008. doi:10.1145/1387673.1387675.

15 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi:10.1145/1411204.1411226.

16 Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940. doi:10.2307/2266170.

17 Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoët, and Gilles Kahn. Natural semantics on the computer. Research Report 416, INRIA, Rocquencourt, France, 1985.

18 Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

19 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, / 1988.

20 N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, New York, 1980.

21 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

22 Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, apr 1995.

23 Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing HOL in an higher order logic programming language. In Gilles Dowek, Daniel R. Licata, and Sandra Alves, editors, *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2016, Porto, Portugal, June 23, 2016*, pages 4:1–4:10. ACM, 2016. `doi:10.1145/2966268.2966272`.

24 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015. `doi:10.1007/978-3-662-48899-7_32`.

25 Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 61–80, Argonne, IL, may 1988. Springer.

26 Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 221–235. Springer, 1990.

27 Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *J. of Automated Reasoning*, 48:43–105, 2012.

28 Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2–A survey. *J. of Automated Reasoning*, 55(4):307–372, 2015.

29 M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Symp. on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.

30 Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008. URL: `http://arxiv.org/abs/0803.2305`.

31 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in Electronic Notes in Theoretical Computer Science, pages 85–100, 2008. `doi:10.1016/j.entcs.2008.12.118`.

32 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011. `doi:10.1016/j.ic.2010.09.004`.

33 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012. `doi:10.1007/s10817-011-9218-1`.

34 Ulysse Gérard and Dale Miller. Separating functional computation from relations. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *LIPIcs*, pages 23:1–23:17, 2017. `doi:10.4230/LIPIcs.CSL.2017.23`.

35 Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte der Mathematischen Physik*, 38:173–198, 1931. English Version in [84].

36 M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

37 Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

38 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

39 Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symp. on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.

40 Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.

41 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 *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)*, pages 207–220, Big Sky, MT, 2009. ACM SIGOPS.

42 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. `doi:10.1145/1538788.1538814`.

43 Chuck Liang, Gopalan Nadathur, and Xiaochu Qi. Choices in representing and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33:89–132, 2005.

44 Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.

45 Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli, 1984.

46 Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, 1997. IEEE Computer Society Press.

47 Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000. `doi:10.1016/S0304-3975(99)00171-1`.

48 Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

49 Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, pages 323–335, Antibes, France, jun 1990. Available as UPenn CIS technical report MS-CIS-90-59. URL: `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/mll.pdf`.

50 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

51 Dale Miller. Bindings, mobility of bindings, and the ∇-quantifier. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Conference on Computer Science Logic (CSL) 2004*, volume 3210 of *Lecture Notes in Computer Science*, page 24, 2004.

52 Dale Miller. Finding unity in computational logic. In *Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference*, ACM-BCS '10, pages 3:1–3:13. British Computer Society, apr 2010. URL: `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/unity2010.pdf`.

53 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, jun 2012. `doi:10.1017/CBO9781139021326`.

54 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

55 Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, oct 2005. `doi:10.1145/1094622.1094628`.

56 Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

57 Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, New York, NY, USA, 1999.

58 Alberto Momigliano and Alwen Tiu. Induction and co-induction in sequent calculus. In Mario Coppo, Stefano Berardi, and Ferruccio Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in Lecture Notes in Computer Science, pages 293–308, jan 2003.

59 J. Strother Moore. A mechanically verified language implementation. *J. of Automated Reasoning*, 5(4):461–492, 1989.

60 Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291, Trento, 1999. Springer.

61 Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

62 Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999. URL: `http://www.jucs.org/jucs_5_3/a_generic_tableau_prover`.

63 Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, 1982. `doi:10.1145/947955.1083808`.

64 Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

65 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, jun 1988.

66 Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in Lecture Notes in Artificial Intelligence, pages 202–206, Trento, 1999. Springer. `doi:10.1007/3-540-48660-7\_14`.

67 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in Lecture Notes in Computer Science, pages 15–21, 2010.

68 The POPLmark Challenge webpage. `http://www.seas.upenn.edu/~plclub/poplmark/`, 2015.

69 François Pottier. An overview of Cαml. In *ACM Workshop on ML*, Electronic Notes in Theoretical Computer Science, pages 27–51, sep 2005.

70 Damien Pous. Weak bisimulation upto elaboration. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 390–405. Springer, 2006.

71 Damien Pous. Complete lattices and upto techniques. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 351–366, Singapore, 2007. Springer.

72 Damien Pous and Davide Sangiorgi. Enhancements of the bisimulation proof method. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, pages 233–289. Cambridge University Press, 2011. `doi:10.1017/CBO9780511792588.007`.

73 Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, 2015. URL: `http://teyjus.cs.umn.edu/`.

74 Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

75 Helmut Schwichtenberg. Minlog. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 151–157. Springer, 2006. `doi:10.1007/11542384\_19`.

76 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(01):71–122, 2010.

77 Zachary Snow, David Baelde, and Gopalan Nadathur. A meta-programming approach to realizing dependently typed logic programming. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.

78 Mary Southern and Kaustuv Chaudhuri. A two-level logic approach to reasoning about typed specification languages. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 557–569, New Delhi, India, 2014. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.FSTTCS.2014.557`.

79 Alwen Tiu. Model checking for π-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of CONCUR'05*, volume 3653 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

80 Alwen Tiu and Dale Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, 2005. `doi:10.1016/j.entcs.2005.05.006`.

81 Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. on Computational Logic*, 11(2), 2010. `doi:10.1145/1656242.1656248`.

82 Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic*, 10(4):330–367, 2012. `doi:10.1016/j.jal.2012.07.007`.

83 Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

84 Jean van Heijenoort. *From Frege to Gödel: A Source Book in Mathematics, 1879-1931.* Source books in the history of the sciences series. Harvard Univ. Press, Cambridge, MA, 3rd printing, 1997 edition, 1967.

85 Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, University of Pennsylvania, may 1996.

Intersection Types and Denotational Semantics: An Extended Abstract

Simona Ronchi Della Rocca
Università degli Studi di Torino, dept Computer Science, Torino, Italy
ronchi@di.unito.it

Abstract

This is a short survey of the use of intersection types for reasoning in a finitary way about terms interpretations in various models of lambda-calculus.

2012 ACM Subject Classification Theory of computation → Denotational semantics

Keywords and phrases Lambda-calculus, Lambda-models, Intersection types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.2

Category Invited Paper

1 Introduction

Intersection types have been introduced by Coppo and Dezani [6], with the aim of enforcing the typability power of simple types, but they quite immediately turned out to be a very powerful tool to reason about the semantics properties of programming languages.

In the general framework of denotational semantics of λ-calculus, intersection types supply a logical description of various kind of λ-models. In particular they allow for a finitary description of the interpretation of terms, through type assignment systems, assigning types to terms starting from a context (finite assignment of types to free variables). Terms are interpreted by sets either of types or of pairs of the shape (context, type), so reasoning about the interpretation of a term can be done via type inference; in fact, in order to prove the equivalence between two terms, it is sufficient to show that they share the same type derivations. Although the type inference is usually undecidable, such a logical description of models supplies concrete tools to reason in a finitary way about the interpretation of terms, since a derivation grasps a finite piece of the semantic-interpretations.

The aim of this brief survey is to illustrate how this technique can be applied to three different classes of models, in the general settings of continuous, stable and relational semantics. I recall that a λ-model is a reflexive object in a cartesian closed category, i.e., a space $\mathtt{D}$ such that the set of morphisms from $\mathtt{D}$ to $\mathtt{D}$ is a retract of $\mathtt{D}$ (in a more concise way $\mathtt{D} \rhd [\mathtt{D} \to \mathtt{D}]$) [2].

2 Continuous Semantics

The first local description of a continuous λ-model through intersection types is in [3]. Then various instances of continuous models have been studied (see, between others, [7, 8, 13, 1]). A general correspondence between intersection types and continuous λ-models has been described in [18].

$$\frac{}{\mathtt{x}:\{\mathtt{A}\}\vdash_{\mathtt{C},\nabla}\mathtt{x}:\mathtt{A}}\,ax \qquad \frac{\Gamma\vdash_{\mathtt{C},\nabla}\mathtt{M}:\mathtt{A}\quad \mathtt{A}\le_\nabla\mathtt{B}}{\Gamma\vdash_{\mathtt{C},\nabla}\mathtt{M}:\mathtt{B}}\,\le_\nabla \qquad \frac{\Gamma,\mathtt{x}:\sigma\vdash_{\mathtt{C},\nabla}\mathtt{M}:\mathtt{A}}{\Gamma,\mathtt{x}:\sigma\cup\tau\vdash_{\mathtt{C},\nabla}\mathtt{M}:\mathtt{A}}\,w$$

$$\frac{\Gamma,\mathtt{x}:\sigma\vdash_{\mathtt{C},\nabla}\mathtt{M}:\mathtt{A}}{\Gamma\vdash_{\mathtt{C},\nabla}\lambda\mathtt{x}.\mathtt{M}:\sigma\to\mathtt{A}}\to I \qquad \frac{\Gamma\vdash_{\mathtt{C},\nabla}\mathtt{M}:\sigma\to\mathtt{A}\quad(\Delta_\mathtt{B}\vdash_{\mathtt{C},\nabla}\mathtt{N}:\mathtt{B})_{\mathtt{B}\in\sigma}}{\Gamma\bigcup_{\mathtt{B}\in\sigma}\Delta_\mathtt{B}\vdash_{\mathtt{C},\nabla}\mathtt{MN}:\mathtt{A}}\to E$$

Figure 1 The continuous parametric type assignment system.

▶ **Definition 1.**

- Types ($\mathcal{T}_\mathtt{C}$) are defined (starting from a countable set $\mathtt{C}$ of constants) as:

$$\mathtt{A},\mathtt{B} ::= \mathtt{a} \mid \sigma\to\mathtt{A}$$

  where $\mathtt{a}\in\mathtt{C}$, and σ is a finite set of types.
- Let ∇ be any pre-order on $\mathcal{T}_\mathtt{C}$, extended to sets in the following way:

$$\begin{array}{lll}\sigma\subseteq\tau & \Rightarrow & \tau\le\sigma\\ \sigma\le\tau,\mathtt{A}\le\mathtt{B} & \Rightarrow & \sigma\cup\{\mathtt{A}\}\le\tau\cup\{\mathtt{B}\}\end{array}$$

  and closed under:

$$\sigma'\le\sigma,\mathtt{A}\le\mathtt{B}\quad\Leftrightarrow\quad\sigma\to\mathtt{A}\le\sigma'\to\mathtt{B}$$

- Let $\simeq_\nabla$ be the congruence induced by ∇, and $\le_\nabla$ the partial order on $\mathcal{T}_\mathtt{C}/\simeq_\nabla$.

▶ Remark. In order to describe different approaches in a uniform manner, I do not use explicit the intersection connective. A set of types $\{\mathtt{A}_1,...,\mathtt{A}_n\}$ corresponds to the more standard notation $\mathtt{A}_1\wedge...\wedge\mathtt{A}_n$, where the intersection connective $\wedge$ is considered modulo idempotency, associativity and commutativity.

The continuous type assignment system, parametric with respect to C,∇, assigning to λ-terms types in $\mathcal{T}_C$ is defined in Figure 1, where Γ,Δ (contexts) are functions from variables to finite subsets of $\mathcal{T}_C$, such that $\Gamma(\mathtt{x})\neq\emptyset$ for a finite number of variables.

Every type assignment system of this kind can be seen as a finitary description of the interpretation of terms in a continuous λ-model $\mathcal{D}_{\mathtt{C},\nabla}$. In fact, for every set of types $\mathcal{T}_\mathtt{C}$ and every partial order $\le_\nabla$, the set of subsets of $\mathcal{T}_\mathtt{C}/\simeq_\nabla$ equipped with the partial order defined as: ($\sigma\sqsubseteq_{\mathtt{C},\nabla}\tau$ if and only if $\tau\le_\nabla\sigma$) is a prime-algebraic lattice $\mathtt{D}_{\mathtt{C},\nabla}$, whose prime elements are the singleton over $\mathcal{T}_\mathtt{C}/\simeq_\nabla$.

Moreover, types are notations for step functions, interpreting $\sigma\to\mathtt{A}$ as a step function approximating f, for every continuous function f such that $\mathtt{A}\in f(\tau)$, for each $\tau\sqsupseteq_{\mathtt{C},\nabla}\sigma$. Under this interpretation $\mathtt{D}_{\mathtt{C},\nabla}$ is a linear solution of the domain equation $D\triangleright[D\Rightarrow_c D]$, where $[.\Rightarrow_c .]$ denotes the space of continuous functions ordered pointwise, and the linearity of the solution means that both the immersion-projection functions map prime elements into prime elements. Let us call *linear continuous models* the models of this class. So $\mathtt{D}_{\mathtt{C},\nabla}$ gives rise to a linear continuous λ-model $\mathcal{D}_{\mathtt{C},\nabla}$. On the other direction, every linear continuous model can be described through a set of types equipped by a suitable intersection relation. Let $[\![.]\!]^{\mathcal{D}_{\mathtt{C},\nabla}}$ be the interpretation function in $\mathcal{D}_{\mathtt{C},\nabla}$, defined as usual: the following theorem holds.

▶ **Theorem 2.** $\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A}$ *if and only if* $\mathtt{A} \in [\![\mathtt{M}]\!]_{\rho}^{\mathcal{D}_{\mathtt{C},\nabla}}$*, for all* ρ *such that* $\Gamma(\mathtt{x}) \subseteq \rho(\mathtt{x})$*.*

So the interpretation of a term in $\mathcal{M}_{\mathtt{C},\nabla}$ is simply the set of types that can be assigned to it.

▶ Remark. If we define a filter to be a set of types closed under $\leq_\nabla$, it turns out that the set of types derivable for a given term is a filter. This justifies the fact that models of this kind have been called "filter models" in the literature.

▶ **Example 3.**
1. The filter model in [3] (where $\mathtt{C}$ is an infinite set of type constants and ∇ is the preorder induced by the empty set of rules) supplies a solution of the domain equation $\mathtt{D} = \mathtt{C} \times [\mathtt{D} \Rightarrow \mathtt{D}]$, through a coding where the type constant $\mathtt{a}$ codes the prime element $(\mathtt{a}, \bot)$, the type $\sigma \to \mathtt{A}$ codes the prime element $(\bot, \sigma \to \mathtt{A})$, and the equation is solved by the bijection $(\bot, \sigma \to \mathtt{A}) \mapsto \sigma \to \mathtt{A}$.
2. The $\mathcal{D}_\infty$ model of Scott [19] is described by $\mathtt{C} = \{\mathtt{a}\}$ and ∇ is induced by the rules $\{\mathtt{a} \leq \emptyset \to \mathtt{a}, \emptyset \to \mathtt{a} \leq \mathtt{a}\}$ ([18]).
3. The Park model [17] is described by $\mathtt{C} = \{\mathtt{a}\}$, and ∇ is induced by the rules $\{\mathtt{a} \leq \{\mathtt{a}\} \to \mathtt{a}, \{\mathtt{a}\} \to \mathtt{a} \leq \mathtt{a}\}$ ([13]).

3 Stable semantics

The stable semantics is based on the notion of stable functions, introduced by Berry [4]. Here I will consider a particular class of models based on stable functions, namely the coherence spaces of Girard [11, 10]. A general correspondence between intersection types and stable λ-model has been described in [15], based on a previous work studying the correspondence between intersection types and qualitative models [12].

▶ **Definition 4.** Let $\mathtt{C}$ be a countable set of constants. Types and coherence type theory are mutually defined as follows.

- Types ($\mathcal{T}_{\mathtt{C},\nabla}$) are defined as:

$$\mathtt{A}, \mathtt{B} ::= \mathtt{a} \mid \sigma \to \mathtt{A}$$

  where $\mathtt{a} \in \mathtt{C}$, and σ is a finite set of types, such that $\frown_\nabla (\sigma)$. i.e., $\mathtt{A} \frown_\nabla \mathtt{B}$, for all $\mathtt{A}, \mathtt{B} \in \sigma, \mathtt{A} \not\simeq_\triangle \mathtt{B}$.
- A coherence type theory ∇ is a pair $(\frown_\nabla, \simeq_\nabla)$, where
  - $\frown_\nabla$ is a symmetric and antireflexive relation on $\mathcal{T}_{\mathtt{C},\nabla}$, closed under

$$\sigma \to \mathtt{A} \frown_\nabla \tau \to \mathtt{B} \Leftrightarrow \text{ either } \mathtt{A} \frown_\nabla \mathtt{B} \text{ or } \exists \mathtt{A}' \in \sigma, \exists \mathtt{B}' \in \tau \text{ such that } \mathtt{A}' \smile_\nabla \mathtt{B}'$$

    (where $\mathtt{A}' \smile_\nabla \mathtt{B}'$ means $\mathtt{A}' \not\frown_\nabla \mathtt{B}'$ and $\mathtt{A}' \not\simeq_\nabla \mathtt{B}'$).
  - $\simeq_\nabla$ is an equivalence on types, extended to sets to sets in such a way that:

$$\sigma \simeq_\nabla \tau \Leftrightarrow \text{ their elements are pairwise } \simeq_\nabla .$$

The stable parametric type assignment system, parametric with respect to $\mathtt{C}, \nabla$ is defined in Figure 2, where Γ, Δ (contexts) are functions from variables to finite sets of types, such that $\Gamma(\mathtt{x}) = \sigma$ implies $\frown_\nabla (\sigma)$ and such that $\Gamma(\mathtt{x}) \neq \emptyset$ for a finite number of variables. Moreover $\bigstar_\nabla(\Delta_1, ..., \Delta_n)$ means ($\mathtt{x} : \sigma \in \Delta_i$ and $\mathtt{x} : \tau \in \Delta_j$ imply either $\frown_\nabla (\sigma, \tau)$ or $\sigma \simeq_\triangle \tau$).

Every type assignment system of this kind can be seen as a finitary description of the interpretation of terms in a stable linear λ-model. In fact, for every choice of $\mathtt{C}$ and ∇, $\mathtt{S}_{\mathtt{C},\nabla} = (\mathcal{T}_{\mathtt{C}}/\simeq_\nabla, \supset)$, where $\supset$ (*coherence*) is the set theoretic union of $\frown_\nabla$ and $\simeq_\nabla$, is a

$$\frac{}{\mathtt{x} : \{\mathtt{A}\} \vdash_{\mathtt{C},\nabla} \mathtt{x} : \mathtt{A}}\ ax \qquad \frac{\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A} \quad \mathtt{A} \simeq_{\nabla} \mathtt{B}}{\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{B}}\ \simeq_{\nabla}$$

$$\frac{\Gamma, \mathtt{x} : \sigma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A}}{\Gamma \vdash_{\mathtt{C},\nabla} \lambda \mathtt{x}.\mathtt{M} : \sigma \to \mathtt{A}}\ \to I$$

$$\frac{\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \sigma \to \mathtt{A} \quad (\Delta_{\mathtt{B}} \vdash_{\mathtt{C},\nabla} \mathtt{N} : \mathtt{B})_{\mathtt{B}\in\sigma} \quad \bigstar_{\nabla}(\Gamma \cup (\bigcup_{\mathtt{B}\in\sigma} \Delta_{\mathtt{B}}))}{\Gamma \bigcup_{\mathtt{B}\in\sigma} \Delta_B \vdash_{\mathtt{C},\nabla} \mathtt{MN} : \mathtt{A}}\ \to E$$

Figure 2 The stable parametric type assignment system.

coherence space. Moreover, types supply a notation for finite pieces of stable functions, interpreting $\sigma \to \mathtt{A}$ as an element of the trace of every stable function f, such that $\mathtt{A} \in f(\sigma)$ and $\forall \tau \subseteq \sigma$, $\mathtt{A} \in f(\tau)$ if and only if $\tau \simeq_{\nabla} \sigma$. Under this interpretation $\mathtt{S}_{\mathtt{C},\nabla}$ is a linear solution of the domain equation $D \triangleright [D \Rightarrow_s D]$, where $[. \Rightarrow_s .]$ denotes the space of stable functions ordered pointwise. Let define *linear stable models* the models of this class.

So $\mathtt{S}_{\mathtt{C},\nabla}$ gives rise to a λ-model, let $\mathcal{S}_{\mathtt{C},\nabla}$. On the other direction, every linear stable model can be described through a set of types equipped by a suitable type theory. Let $[\![.]\!]^{\mathcal{S}_{\mathtt{C},\nabla}}$ be the interpretation function in $\mathcal{S}_{\mathtt{C},\nabla}$, defined in the usual way; the following theorem holds.

▶ **Theorem 5.** $\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A}$ *if and only if* $\mathtt{A} \in [\![\mathtt{M}]\!]_{\rho}^{\mathcal{S}_{\mathtt{C},\nabla}}$*, for all* ρ *such that* $\Gamma(\mathtt{x}) \subseteq \rho(\mathtt{x})$.

So the interpretation of a term in $\mathcal{S}_{\mathtt{C},\nabla}$ is simply the set of types that can be assigned to it.

▶ **Example 6.**

1. Let $\mathtt{C}$ be a countable infinite set, $\nabla = (\circeq, \simeq)$, where $\circeq$ is the minimum relation induced by the rules: $\mathtt{a} \frown \mathtt{A}, \forall \mathtt{a} \in \mathtt{C}, \mathtt{A} \in \mathcal{T}_{\mathtt{C},\nabla}$ and $\simeq$ is the minimum congruence on types, satisfying the conditions of Def.4. Then $(\mathcal{T}_{\mathtt{C},\nabla}, \circeq)$ supplies a solution of the equation $\mathtt{S} = \mathtt{C} \,\&\, [\mathtt{S} \Rightarrow_s \mathtt{S}]$, choosing the type $\mathtt{a}$ to describe the token $(1, \mathtt{a})$, and $\sigma \to \mathtt{A}$ to describe the token $(2, \sigma \to \mathtt{A})$. The solution is induced by the bijection $(2, \sigma \to \mathtt{A}) \mapsto \sigma \to \mathtt{A}$.
2. Let $\mathtt{C} = \{\mathtt{a}\}$, let $\circeq$ be the minimum relation and let $\simeq$ be the minimum congruence induced by $\{\mathtt{a} \simeq \emptyset \to \mathtt{a}\}$. Then the resulting λ-model is (in some sense) the stable corresponding to $\mathcal{D}_{\infty}$, built in [12].
3. Let $\mathtt{C} = \{\mathtt{a}\}$, let $\circeq$ be as in the previous point, and let $\simeq$ be the minimum congruence induced by $\{\mathtt{a} \simeq \{\mathtt{a}\} \to \mathtt{a}\}$. Then the resulting λ-model is (in some sense) the stable corresponding to the Park model, built in [12].

4 Relational semantics

The relational semantics has been developed by Bucciarelli, Manzonetto and Ehrhard ([5]). A general correspondence between intersection types and relational λ-model has been described in [16].

$$\frac{}{\mathtt{x} : [\mathtt{A}] \vdash_{\mathtt{C},\nabla} \mathtt{x} : \mathtt{A}}\ ax \qquad \frac{\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A} \quad \mathtt{A} \simeq_\nabla \mathtt{B}}{\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{B}}\ \simeq_\nabla$$

$$\frac{\Gamma, \mathtt{x} : \sigma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A}}{\Gamma \vdash_{\mathtt{C},\nabla} \lambda \mathtt{x}.\mathtt{M} : \sigma \to \mathtt{A}} \to I$$

$$\frac{\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \sigma \to \mathtt{A} \quad (\Delta_{\mathtt{B}} \vdash_{\mathtt{C},\nabla} \mathtt{N} : \mathtt{B})_{\mathtt{B}\in\sigma}}{\Gamma \uplus_{\mathtt{B}\in\sigma} \Delta_B \vdash_{\mathtt{C},\nabla} \mathtt{MN} : \mathtt{A}} \to E$$

Figure 3 The relational parametric type assignment system.

▶ **Definition 7.**
- Types ($\mathcal{T}_{\mathtt{C}}$) are defined (starting from a countable set $\mathtt{C}$ of constants) as:

  $$\mathtt{A}, \mathtt{B} ::= \mathtt{a} \mid \sigma \to \mathtt{A}$$

  where σ is a finite multiset of types.
- A relational type theory $\simeq_\nabla$ is a congruence on types, behaving on multisets in the following way:

  $$\sigma \simeq_\nabla \tau \quad \Leftrightarrow \quad \sigma = [\mathtt{A}_1, ..., \mathtt{A}_n], \tau = [\mathtt{B}_1, ..., \mathtt{B}_n] \text{ and } \mathtt{A}_i \simeq_\nabla \mathtt{B}_i$$

  and closed under:

  $$\sigma \to \mathtt{A} \simeq_\nabla \sigma' \to \mathtt{B} \quad \Leftrightarrow \quad \sigma \simeq_\nabla \sigma', \mathtt{A} \simeq \mathtt{B}$$

▶ Remark. The multisets of types $[\mathtt{A}_1, ..., \mathtt{A}_n]$ is an alternative notation for $\mathtt{A}_1 \wedge ... \wedge \mathtt{A}_n$, where the intersection connective enjoys associativity and commutativity, but not idempotence; so the congruence on multisets needs to take into account the multiplicity of elements. Let $[\,]$ denote the empty multiset.

The relational parametric type assignment system, parametric with respect to $\mathtt{C}, \nabla$ is defined in Figure 3, where Γ, Δ (contexts) are functions from variables to finite multisets of types, such that $\Gamma(\mathtt{x}) \neq [\,]$ for a finite number of variables. Moreover $\uplus$ denotes the multiset union.

An arrow type denotes a relation from $\mathcal{M}_{fin}(\mathcal{T}_{\mathtt{C}}/\simeq_\nabla)$ and $\mathcal{T}_{\mathtt{C}}/\simeq_\nabla$, where $\mathcal{M}_{fin}(.)$ is the set of finite multisets. It turns out that $\mathcal{T}_{\mathtt{C}}/\simeq_\nabla$ supplies a solution of the equation $\mathtt{U} \rhd [\mathtt{U} \Rightarrow_r \mathtt{U}]$, where $\mathtt{U}$ is a set and $[. \Rightarrow_r .]$ denotes the space of relations between $\mathcal{M}_{fin}(\mathtt{U})$ and $\mathtt{U}$. But this space does not supply directly a λ-model, as shown in [9], since it has not enough points. It is necessary to consider the space $Fin(\mathtt{U}^{Var}, \mathtt{U})$, which consists of the finitary morphisms from $\mathtt{U}^{Var}$ to $\mathtt{U}$, where Var is a countable set of variables, and gives rise to a λ-model. An element of such a space can be represented by a pre-typing, which is a pair $(\Gamma; \mathtt{A})$ of a context and a type, both considered modulo $\simeq_\nabla$: in fact pre-typings are elements of a space $\mathtt{R}_{\mathtt{C},\nabla} = Fin((\mathcal{T}_{\mathtt{C}}/\simeq_\nabla)^{Var}, \mathcal{T}_{\mathtt{C}}/\simeq_\nabla)$ which supplies a λ-model $\mathcal{R}_{\mathtt{C},\nabla}$; let $[\![.]\!]^{\mathcal{R}_{\mathtt{C},\nabla}}$ be its interpretation function, defined in the usual way. The following theorem holds.

▶ **Theorem 8.** $\Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A}$ *if and only if* $(\Gamma'; \mathtt{A}) \in [\![\mathtt{M}]\!]_\rho^{\mathcal{R}_{\mathtt{C},\nabla}}$, *where* $\Gamma' = \uplus_i \Delta_i$, $(\Delta_i; \mathtt{A}^i) \in \rho(\mathtt{x}_i)$, *for every* $\mathtt{A}^i \in \Gamma(\mathtt{x}_i)$.

▶ **Example 9.**

- Choosing $\mathtt{C} = \{\mathtt{a}\}$ and the type theory induced by $[\,] \to \mathtt{a} \simeq \mathtt{a}$, we obtain the model studied in [5].
- Choosing $\mathtt{C} = \{\mathtt{a}\}$ and the type theory induced by $[\mathtt{a}] \to \mathtt{a} \simeq \mathtt{a}$, we obtain the model studied in [14].

5 Conclusion

Comparing the three parametric type assignment systems in Figure 1, 2 and 3, it turns out that the three systems are quite different, from a proof theoretical point of view. The continuous system uses idempotent intersection, it enjoys weakening, and requires a subsumption rule. The stable system uses idempotent intersection too, but it is relevant (in the sense that weakening is unsound) and requires an equivalence rule. Finally, the relational system uses non-idempotent intersection, is relevant, and requires an equivalence rule. Moreover the interpretation of a term is the set of types derivable for it in the first two systems, while is the set of pre-typings in the third one. Despite these differences, they supply a logical tools for reasoning in a uniform way about the denotational semantics of terms, in particular for comparing terms from a semantics point of view. In fact, the following theorem holds.

▶ **Theorem 10.** *Let $\mathcal{M}_{\mathtt{C},\nabla}$ be a (continous, stable, relational) λ-model, and let $\sqsubseteq_{\mathcal{M}_{\mathtt{C},\nabla}}$ be the order relation between interpretations of terms in it. Then:*

$$(\forall \Gamma, \mathtt{A}.\ \Gamma \vdash_{\mathtt{C},\nabla} \mathtt{M} : \mathtt{A} \textit{ implies } \Gamma \vdash_{\mathtt{C},\nabla} \mathtt{N} : \mathtt{A}) \textit{ if and only if } \mathtt{M} \sqsubseteq_{\mathcal{M}_{\mathtt{C},\nabla}} \mathtt{N}.$$

References

1 Fabio Alessi, Mariangiola Dezani-Ciancaglini, and Furio Honsell. A complete characterization of complete intersection-type preorders. *ACM Transactions on Computational Logic*, 4(1):120–147, jan 2003.

2 Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics.* North-Holland, revised edition, 1984.

3 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, dec 1983.

4 Gérard Berry. Stable models of typed λ-calculi. In Giorgio Ausiello and Corrado Böhm, editors, *Fifth International Colloquium on Automata, Languages and Programming - - ICALP'78, Udine, Italy, July 17-21, 1978*, volume 62 of *Lecture Notes in Computer Science*, pages 72–89. Springer-Verlag, 1978.

5 Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. Not enough points is enough. In *CSL'07*, volume 4646 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2007.

6 Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, oct 1980.

7 Mario Coppo, Mariangiola Dezani-Ciancaglini, Furio Honsell, and Giuseppe Longo. Extended type structure and filter lambda models. In *Logic Colloquim'82*, pages 241–262, 1984.

8 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Maddalena Zacchi. Type theories, normal forms, and D_∞-lambda-models. *Information and Computation*, 72(2):85–116, 1987.

9 Daniel de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *CoRR*, abs/0905.4251, 2009. Available also as INRIA report RR 6638. URL: http://arxiv.org/abs/0905.4251.

10 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

11 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1989.

12 Furio Honsell and Simona Ronchi della Rocca. Reasoning about interpretation in qualitative lambda-models. In *IFIP 2.2*, pages 505–521, 1990.

13 Furio Honsell and Simona Ronchi Della Rocca. An approximation theorem for topological lambda models and the topological incompleteness of lambda calculus. *Journal of Computer and System Sciences*, 45(1):49–75, aug 1992.

14 Giulio Manzonetto and Domenico Ruoppolo. Relational graph models, Taylor expansion and extensionality. *Electronic Notes in Theoretical Computer Science*, 308:245–272, 2014.

15 Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Logical semantics for stability. In *MFPS'09*, volume 249 of *Electronic Notes in Theoretical Computer Science*, pages 429–449, 2009.

16 Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Mathematical Structures in Computer Science*, FirstView:1–25, 9 2015. `doi:10.1017/S0960129515000316`.

17 D. M. R. Park. The Y-combinator in scott's lambda-calculus models. Research Report CS-RR-013, Department of Computer Science, University of Warwick, Coventry, UK, jun 1976. URL: `http://www.dcs.warwick.ac.uk/pub/reports/rr/013.html`.

18 Simona Ronchi Della Rocca and Luca Paolini. *The Parametric λ-Calculus: a Metamodel for Computation*. Texts in Theoretical Computer Science: An EATCS Series. Springer-Verlag, 2004.

19 Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, sep 1976.

A Normalizing Computation Rule for Propositional Extensionality in Higher-Order Minimal Logic

Robin Adams
Chalmers tekniska högskola, Data- och informationsteknik, 412 96 Göteborg, Sweden
robinad@chalmers.se
https://orcid.org/0000-0003-2644-1093

Marc Bezem
Universitetet i Bergen, Institutt for Informatikk, Postboks 7800, N-5020 BERGEN, Norway
bezem@ii.uib.no
https://orcid.org/0000-0002-7320-1976

Thierry Coquand
Chalmers tekniska högskola, Data- och informationsteknik, 412 96 Göteborg, Sweden
coquand@chalmers.se
https://orcid.org/0000-0002-5429-5153

Abstract

The univalence axiom expresses the principle of extensionality for dependent type theory. However, if we simply add the univalence axiom to type theory, then we lose the property of *canonicity* – that every closed term computes to a canonical form. A computation becomes "stuck" when it reaches the point that it needs to evaluate a proof term that is an application of the univalence axiom. So we wish to find a way to compute with the univalence axiom. While this problem has been solved with the formulation of cubical type theory, where the computations are expressed using a nominal extension of lambda-calculus, it may be interesting to explore alternative solutions, which do not require such an extension.

As a first step, we present here a system of propositional higher-order minimal logic (PHOML). There are three kinds of typing judgement in PHOML. There are *terms* which inhabit *types*, which are the simple types over Ω. There are *proofs* which inhabit *propositions*, which are the terms of type Ω. The canonical propositions are those constructed from $\bot$ by implication $\supset$. Thirdly, there are *paths* which inhabit *equations* $M =_A N$, where M and N are terms of type A. There are two ways to prove an equality: reflexivity, and *propositional extensionality* – logically equivalent propositions are equal. This system allows for some definitional equalities that are not present in cubical type theory, namely that transport along the trivial path is identity.

We present a call-by-name reduction relation for this system, and prove that the system satisfies canonicity: every closed typable term head-reduces to a canonical form. This work has been formalised in Agda.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, univalence, canonicity

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.3

Supplement Material The proofs in this paper have been formalized in Agda. The formalization is available at `https://github.com/radams78/TYPES2016`.

1 Introduction

The *univalence axiom* of Homotopy Type theory (HoTT) [11] postulates a constant

$$\mathsf{isotoid} : A \simeq B \to A = B$$

that is an inverse to the obvious function $A = B \to A \simeq B$. However, if we simply add this constant to Martin-Löf type theory, then we lose the important property of *canonicity* – that every closed term of type A computes to a unique canonical object of type A. When a computation reaches a point where we eliminate a path (proof of equality) formed by isotoid, it gets "stuck".

As possible solutions to this problem, we may try to do with a weaker property than canonicity, such as *propositional canonicity*: that every closed term of type $\mathbb{N}$ is *propositionally* equal to a numeral, as conjectured by Voevodsky. Or we may attempt to change the definition of equality to make isotoid definable [9], or add a nominal extension to the syntax of the type theory (e.g. Cubical Type Theory [3]).

We could also try a more conservative approach, and simply attempt to find a reduction relation for a type theory involving isotoid that satisfies all three of the properties above. There seems to be no reason *a priori* to believe this is not possible, but it is difficult to do because the full Homotopy Type Theory is a complex and interdependent system. We can tackle the problem by adding univalence to a much simpler system, finding a well-behaved reduction relation, then doing the same for more and more complex systems, gradually approaching the full strength of HoTT.

In this paper, we present a system we call PHOML, or predicative higher-order minimal logic. It is a type theory with three kinds of typing judgement. There are *proofs* which inhabit *propositions*, which are the terms of type Ω. The canonical propositions are those constructed from $\bot$ by implication $\supset$. There are *terms* which inhabit *types*, which are the simple types over Ω. Thirdly, there are *paths* which inhabit *equations* $M =_A N$, where M and N are terms of type A.

There are two canonical forms for proofs of $M =_\Omega N$. For any term $\varphi : \Omega$, we have $\mathrm{ref}\,(\varphi) : \varphi =_\Omega \varphi$. We also add univalence for this system, in this form: if $\delta : \varphi \supset \psi$ and $\epsilon : \psi \supset \varphi$, then $\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon) : \varphi =_\Omega \psi$.

This entails that in PHOML, two propositions that are logically equivalent are equal. Every function of type $\Omega \to \Omega$ that can be constructed in PHOML must therefore respect logical equivalence. That is, for any F and logically equivalent x, y we must have that Fx and Fy are logically equivalent. Moreover, if for $x : \Omega$ we have that Fx is logically equivalent to Gx, then $F =_{\Omega\to\Omega} G$. Every function of type $(\Omega \to \Omega) \to \Omega$ must respect this equality; and so on. This is the manifestation in PHOML of the principle that only homotopy invariant constructions can be performed in homotopy type theory. (See Section 3.1.)

We present a call-by-name reduction relation for this system, and prove that every typable term reduces to a canonical form. From this, it follows that the system is consistent.

For the future, we wish to include the equations in Ω, allowing for propositions such as $M =_A N \supset N =_A M$. We wish to expand the system with universal quantification, and expand it to a 2-dimensional system (with equations between proofs). We then wish to add more inductive types and more dimensions, getting ever closer to full homotopy type theory.

1.1 Related Work

Another system with many of the same aims is cubical type theory (CTT) [3]. A similar canonicity result has been proved for CTT [6].

The system PHOML is almost a subsystem of cubical type theory. We can attempt to embed PHOML into cubical type theory, mapping Ω to the universe U, and an equation $M =_A N$ to either the type $\mathsf{Path}\, A\, M\, N$ or to $\mathrm{Id}\ A\ M\ N$. However, PHOML has more definitional equalities than the relevant fragment of cubical type theory; that is, there are definitionally equal terms in PHOML that are mapped to terms that are not definitionally equal in cubical type theory. In particular, $\mathrm{ref}\,(x)^+\, p$ and p are definitionally equal, whereas the terms $\mathrm{comp}^i x[]p$ and p are not definitionally equal in cubical type theory (but they are propositionally equal). See Section 3.2.1 for more information.

Other systems with similar aims include Harper and Licata [7], who prove canonicity for a system that includes equality reflection; and Angiuli, Harper and Wilson [1] who prove canonicity for a system with univalence, dependent types and some higher inductive types, but without any universes.

The proofs in this paper have been formalized in Agda. The formalization is available at `https://github.com/radams78/TYPES2016`.

2 Predicative Higher-Order Minimal Logic with Extensional Equality

We call the following type theory PHOML, or *predicative higher-order minimal logic with extensional equality*.

2.1 Syntax

Fix three disjoint, infinite sets of variables, which we shall call *term variables*, *proof variables* and *path variables*. We shall use x and y as term variables, p and q as proof variables, e as a path variable, and z for a variable that may come from any of these three sets.

The syntax of PHOML is given by the grammar:

Type	A, B, C	$::=$	$\Omega \mid A \to B$
Term	$L, M, N, \varphi, \psi, \chi$	$::=$	$x \mid \bot \mid \varphi \supset \psi \mid \lambda x : A.M \mid MN$
Proof	δ, ϵ	$::=$	$p \mid \lambda p : \varphi.\delta \mid \delta\epsilon \mid P^+ \mid P^-$
Path	P, Q	$::=$	$e \mid \mathrm{ref}\,(M) \mid P \supset^* Q \mid \mathrm{univ}_{\varphi,\psi}\,(P, Q) \mid$
			$\lambda\!\!\lambda\!\!\lambda e : x =_A y.P \mid P_{MN}Q$
Context	Γ, Δ, Θ	$::=$	$\langle\rangle \mid \Gamma, x : A \mid \Gamma, p : \varphi \mid \Gamma, e : M =_A N$
Judgement	$\mathbf{J}$	$::=$	$\Gamma \vdash \mathrm{valid} \mid \Gamma \vdash M : A \mid \Gamma \vdash \delta : \varphi \mid$
			$\Gamma \vdash P : M =_A N$

In the path $\lambda\!\!\lambda\!\!\lambda e : x =_A y.P$, the term variables x and y must be distinct. (We also have $x \not\equiv e \not\equiv y$, thanks to our stipulation that term variables and path variables are disjoint.) The term variable x is bound within M in the term $\lambda x : A.M$, and the proof variable p is bound within δ in $\lambda p : \varphi.\delta$. The three variables e, x and y are bound within P in the path $\lambda\!\!\lambda\!\!\lambda e : x =_A y.P$. We identify terms, proofs and paths up to α-conversion. We write $E[z := F]$ for the result of substituting F for z within E, using α-conversion to avoid variable capture.

We shall use the word "expression" to mean either a type, term, proof, path, or equation (an equation having the form $M =_A N$). We shall use E, F, S and T as metavariables that range over expressions.

Note that we use both Roman letters M, N and Greek letters φ, ψ, χ to range over terms. Intuitively, a term is understood as either a proposition or a function, and we shall use Greek letters for terms that are intended to be propositions. Formally, there is no significance to which letter we choose.

Note also that the types of PHOML are just the simple types over Ω; therefore, no variable can occur in a type.

2.1.1 Intuitive Explanation

The intuition behind the new expressions is as follows (see also the rules of deduction in Figure 2). For any object $M : A$, there is the trivial path $\mathrm{ref}\,(M) : M =_A M$. The constructor $\supset^*$ ensures congruence for $\supset$ – if $P : \varphi =_\Omega \varphi'$ and $Q : \psi =_\Omega \psi'$ then $P \supset^* Q : \varphi \supset \psi =_\Omega \varphi' \supset \psi'$. The constructor univ gives "univalence" (propositional extensionality) for our propositions: if $\delta : \varphi \supset \psi$ and $\epsilon : \psi \supset \varphi$, then $\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon)$ is a path $\varphi =_\Omega \psi$. The constructors $^+$ and $^-$ denote the action of transport along a path: if P is a path of type $\varphi =_\Omega \psi$, then P^+ is a proof of $\varphi \supset \psi$, and P^- is a proof of $\psi \supset \varphi$.

The constructor $\lambda\!\!\!\lambda\!\!\!\lambda$ gives functional extensionality. Let F and G be functions of type $A \to B$. If $Fx =_B Gy$ whenever $x =_A y$, then $F =_{A\to B} G$. More formally, if P is a path of type $Fx =_B Gy$ that depends on $x : A$, $y : A$ and $e : x =_A y$, then $\lambda\!\!\!\lambda\!\!\!\lambda e : x =_A y.P$ is a path of type $F =_{A\to B} G$.

Finally, if P is a path $M =_{A\to B} M'$, and Q is a path $N =_A N'$, then $P_{MN}Q$ is a path $MN =_B M'N'$.

Note. The equations $M =_A N$ are quite different from the identity types in Martin-Löf Type Theory. In Martin-Löf Type Theory, the only constructor for the identity type is $\mathrm{ref}\,(\ )$. In our system, the constructors for $M =_A N$ to vary with the type A.

The equations $\phi =_\Omega \psi$ have two constructors:

- $\mathrm{ref}\,(\phi)$ is a canonical path of $\phi =_\Omega \phi$.
- If $\delta : \phi \supset \psi$ and $\epsilon : \psi \supset \varphi$, then $\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon)$ is a canonical path of $\phi =_\Omega \psi$.

The equations $F =_{A\to B} G$ have two constructors:

- $\mathrm{ref}\,(F)$ is a canonical path of $F =_{A\to B} F$
- If P is a path of $Fx =_B Gy$ that depends on $x : A$, $y : A$ and $e : x =_A y$, then $\lambda\!\!\!\lambda\!\!\!\lambda e : x =_A y.P$ is a canonical path of $F =_{A\to B} G$.

We therefore define the canonical paths to be those of the form $\mathrm{ref}\,(M)$, $\mathrm{univ}_{\phi,\psi}\,(\delta, \epsilon)$ or $\lambda\!\!\!\lambda\!\!\!\lambda e : x =_A y.P$ (see Definition 19).

2.1.2 Substitution and Path Substitution

Intuitively, if N and N' are equal then $M[x := N]$ and $M[x := N']$ should be equal. To handle this syntactically, we introduce a notion of *path substitution.* If N, M and M' are terms, x a term variable, and P a path, then we shall define a path $N\{x := P : M = M'\}$. The intention is that, if $\Gamma \vdash P : M =_A M'$ and $\Gamma, x : A \vdash N : B$ then $\Gamma \vdash N\{x := P : M = M'\} : N[x := M] =_B N[x := M']$ (see Lemma 17).

▶ **Definition 1** (Path Substitution)**.** Given terms M_1, …, M_n and N_1, …, N_n; paths P_1, …, P_n; term variables x_1, …, x_n; and a term L, define the path

$$L\{x_1 := P_1 : M_1 = N_1, \ldots, x_n := P_n : M_n = N_n\}$$

as follows.

$$
\begin{aligned}
x_i\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} &\stackrel{\text{def}}{=} P_i \\
y\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} &\stackrel{\text{def}}{=} \mathrm{ref}\,(y) \qquad (y \not\equiv x_1, \dots, x_n) \\
\bot\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} &\stackrel{\text{def}}{=} \mathrm{ref}\,(\bot) \\
(LL')\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} & \\
&\stackrel{\text{def}}{=} L\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\}_{L'[\vec{x}:=\vec{M}]L'[\vec{x}:=\vec{N}]} L'\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} \\
(\lambda y : A.L)\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} & \\
&\stackrel{\text{def}}{=} \lambda\!\!\lambda\!\!\lambda e : a =_A a'.L\{\vec{x} := \vec{P} : \vec{M} = \vec{N}, y := e : a = a'\} \\
(\varphi \supset \psi)\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} &\stackrel{\text{def}}{=} \varphi\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\} \supset^* \psi\{\vec{x} := \vec{P} : \vec{M} = \vec{N}\}
\end{aligned}
$$

We shall often omit the endpoints $\vec{M}$ and $\vec{N}$.

▶ **Note 2.** The case $n = 0$ is permitted, and we shall have that, if $\Gamma \vdash M : A$ then $\Gamma \vdash M\{\} : M =_A M$. There are thus two paths from a term M to itself: $\mathrm{ref}\,(M)$ and $M\{\}$. They are not always equal; for example, $(\lambda x : A.x)\{\} \equiv \lambda\!\!\lambda\!\!\lambda e : x =_A y.e$, which (after we define the reduction relation) will not be convertible with $\mathrm{ref}\,(\lambda x : A.x)$.

The following lemma shows how substitution and path substitution interact.

▶ **Lemma 3.** *Let $\vec{y}$ be a sequences of variables and x a distinct variable. Then*

1. $M[x := N]\{\vec{y} := \vec{P} : \vec{L} = \vec{L'}\}$
   $\equiv M\{x := N\{\vec{y} := \vec{P} : \vec{L} = \vec{L'}\} : N[\vec{y} := \vec{L}] = N[\vec{y} := \vec{L'}], \vec{y} := \vec{P} : \vec{L} = \vec{L'}\}$
2. $M\{\vec{y} := \vec{P} : \vec{L} = \vec{L'}\}[x := N]$
   $\equiv M\{\vec{y} := \vec{P}[x := N] : \vec{L}[x := N] = \vec{L'}[x := N], x := \mathrm{ref}\,(N) : N = N\}$

Proof. An easy induction on M in all cases. ◀

▶ **Note 4.** The familiar substitution lemma also holds as usual: $t[\vec{z_1} := \vec{s_1}][\vec{z_2} := \vec{s_2}] \equiv t[\vec{z_1} := \vec{s_1}[\vec{z_2} := \vec{s_2}], \vec{z_2} := \vec{s_2}]$. We cannot form a lemma about the fourth case, simplifying $M\{\vec{x} := \vec{P}\}\{\vec{y} := \vec{Q}\}$, because $M\{\vec{x} := \vec{P}\}$ is a path, and path substitution can only be applied to a term.

We introduce a notation for simultaneous substitution and path substitution of several variables:

▶ **Definition 5.** A *substitution* is a function that maps term variables to terms, proof variables to proofs, and path variables to paths. We write $E[\sigma]$ for the result of substituting the expression $\sigma(z)$ for z in E, for each variable z in the domain of σ.

A *path substitution* τ is a function whose domain is a finite set of term variables, and which maps each term variable to a path. Given a path substitution τ and substitutions ρ, σ with the same domain $\{x_1, \dots, x_n\}$, we write

$$M\{\tau : \rho = \sigma\} \text{ for } M\{x_1 := \tau(x_1) : \rho(x_1) = \sigma(x_1), \dots, \tau(x_n) : \rho(x_n) = \sigma(x_n)\} \ .$$

2.1.3 Call-By-Name Reduction

▶ **Definition 6** (Call-By-Name Reduction)**.** Define the relation of *call-by-name reduction* $\to$ on the expressions. The inductive definition is given by the rules in Figure 1.

Reduction on Terms

$$\frac{}{(\lambda x : A.M)N \to M[x := N]} \qquad \frac{M \to M'}{MN \to M'N}$$

$$\frac{\varphi \to \varphi'}{\varphi \supset \psi \to \varphi' \supset \psi} \qquad \frac{\psi \to \psi'}{\varphi \supset \psi \to \varphi \supset \psi'}$$

Reduction on Proofs

$$\frac{}{(\lambda p : \varphi.\delta)\epsilon \to \delta[p := \epsilon]} \qquad \frac{}{\mathrm{ref}\,(\varphi)^{+} \to \lambda p : \varphi.p} \qquad \frac{}{\mathrm{ref}\,(\varphi)^{-} \to \lambda p : \varphi.p}$$

$$\frac{\delta \to \delta'}{\delta\epsilon \to \delta'\epsilon} \qquad \frac{}{\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon)^{+} \to \delta} \qquad \frac{}{\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon)^{-} \to \epsilon}$$

$$\frac{P \to Q}{P^{+} \to Q^{+}} \qquad \frac{P \to Q}{P^{-} \to Q^{-}}$$

Reduction on Paths

$$\frac{}{(\lambda\!\!\!\lambda\!\!\!\lambda e : x =_A y.P)_{MN}Q \to P[x := M, y := N, e := Q]}$$

$$\frac{}{\mathrm{ref}\,(\lambda x : A.M)_{NN'}\,P \to M\{x := P : N = N'\}}$$

$$\frac{}{\mathrm{ref}\,(\varphi) \supset^{*} \mathrm{ref}\,(\psi) \to \mathrm{ref}\,(\varphi \supset \psi)}$$

$$\frac{}{\mathrm{ref}\,(\varphi) \supset^{*} \mathrm{univ}_{\psi,\chi}\,(\delta, \epsilon) \to \mathrm{univ}_{\varphi\supset\psi,\varphi\supset\chi}\,(\lambda p : \varphi \supset \psi.\lambda q : \varphi.\delta(pq), \lambda p : \varphi \supset \chi.\lambda q : \varphi.\epsilon(pq))}$$

$$\frac{}{\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon) \supset^{*} \mathrm{ref}\,(\chi) \to \mathrm{univ}_{\varphi\supset\chi,\psi\supset\chi}\,(\lambda p : \varphi \supset \chi.\lambda q : \psi.p(\epsilon q), \lambda p : \psi \supset \chi.\lambda q : \varphi.p(\delta q))}$$

$$\frac{}{\begin{array}{l}\mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon) \supset^{*} \mathrm{univ}_{\varphi',\psi'}\,(\delta', \epsilon') \\ \to \mathrm{univ}_{\varphi\supset\varphi',\psi\supset\psi'}\,(\lambda p : \varphi \supset \varphi'.\lambda q : \psi.\delta'(p(\epsilon q)), \lambda p : \psi \supset \psi'.\lambda q : \varphi.\epsilon'(p(\delta q)))\end{array}}$$

$$\frac{P \to P'}{P_{MN}Q \to P'_{MN}Q} \qquad \frac{M \to M'}{\mathrm{ref}\,(M)_{NN'}\,P \to \mathrm{ref}\,(M')_{NN'}\,P}$$

$$\frac{P \to P'}{P \supset^{*} Q \to P' \supset^{*} Q} \qquad \frac{Q \to Q'}{P \supset^{*} Q \to P \supset^{*} Q'}$$

Figure 1 Reduction in PHOML.

We write $\twoheadrightarrow$ for the reflexive transitive closure of $\to$, and we write $\overset{*}{\leftrightarrow}$ for the reflexive symmetric transitive closure of $\to$. We say an expression E is in *normal form* iff there is no expression F such that $E \to F$.

▶ **Lemma 7** (Confluence). *If $E \twoheadrightarrow F$ and $E \twoheadrightarrow G$, then there exists H such that $F \twoheadrightarrow H$ and $G \twoheadrightarrow H$.*

Proof. The proof is given in Appendix B. ◀

▶ **Lemma 8** (Reduction respects path substitution). *If $M \to N$ then $M\{\tau : \rho = \sigma\} \to N\{\tau : \rho = \sigma\}$.*

Proof. Induction on $M \to N$. The only difficult case is β-contraction. We have

$$\begin{aligned}
&((\lambda x : A.M)N)\{\tau : \rho = \sigma\} \\
\equiv&(\lambda\!\!\!\lambda\!\!\!\lambda e : x =_A x'.M\{\tau : \rho = \sigma, x := e : x = x'\})_{N[\rho]N[\sigma]} N\{\tau : \rho = \sigma\} \\
\to& M\{\tau : \rho = \sigma, x := N\{\tau\} : N[\rho] = N[\sigma]\} \\
\equiv& M[x := N]\{\tau : \rho = \sigma\} && \text{(Lemma 3)}
\end{aligned}$$ ◀

▶ **Note 9.**

1. Reduction on proofs and paths does *not* respect term substitution. For example, let $M \equiv \lambda x : \Omega.x$. Then we have

$$\begin{aligned}
\text{ref}\,(\lambda y : \Omega.y')_{\bot\bot}\,\text{ref}\,(\bot) &\to y'\{y := \text{ref}\,(\bot) : \bot = \bot\} \equiv \text{ref}\,(y') \\
(\text{ref}\,(\lambda y : \Omega.y')_{\bot\bot}\,\text{ref}\,(\bot))[y' := M] &\equiv \text{ref}\,(\lambda y : \Omega.M)_{\bot\bot}\,\text{ref}\,(\bot) && (1) \\
\text{ref}\,(y')\,[y' := M] &\equiv \text{ref}\,(M) \equiv \text{ref}\,(\lambda x : \Omega.x) && (2)
\end{aligned}$$

   Expression (1) does not reduce to (2). Instead, (1) reduces to

$$\begin{aligned}
M\{y := \text{ref}\,(\bot) : \bot = \bot\} &\equiv \lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega x'.x\{y := \text{ref}\,(\bot) : \bot = \bot, x := e : x = x'\} \\
&\equiv \lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega x'.e\ .
\end{aligned}$$

2. Reduction on terms does respect substitution: if $M \to N$ then $M[x := P] \to N[x := P]$, as is easily shown by induction on $M \to N$.

2.2 Rules of Deduction

The rules of deduction of PHOML are given in Figure 2.

2.2.1 Metatheorems

In the lemmas that follow, the letter $\mathcal{J}$ stands for any of the expressions that may occur to the right of the turnstile in a judgement, i.e. valid, $M : A$, $\delta : \varphi$, or $P : M =_A N$.

▶ **Lemma 10** (Context Validity). *Every derivation of* $\Gamma, \Delta \vdash \mathcal{J}$ *has a subderivation of* $\Gamma \vdash$ valid.

Proof. Induction on derivations. ◀

▶ **Lemma 11** (Weakening). *If* $\Gamma \vdash \mathcal{J}$, $\Gamma \subseteq \Delta$ *and* $\Delta \vdash$ valid *then* $\Delta \vdash \mathcal{J}$.

Proof. Induction on derivations. ◀

▶ **Lemma 12** (Type Validity).

1. *If* $\Gamma \vdash \delta : \varphi$ *then* $\Gamma \vdash \varphi : \Omega$.
2. *If* $\Gamma \vdash P : M =_A N$ *then* $\Gamma \vdash M : A$ *and* $\Gamma \vdash N : A$.

Proof. Induction on derivations. The cases where δ or P is a variable use Context Validity. ◀

▶ **Lemma 13** (Generation).

1. *If* $\Gamma \vdash x : A$ *then* $x : A \in \Gamma$.
2. *If* $\Gamma \vdash \bot : A$ *then* $A \equiv \Omega$.
3. *If* $\Gamma \vdash \varphi \supset \psi : A$ *then* $\Gamma \vdash \varphi : \Omega$, $\Gamma \vdash \psi : \Omega$ *and* $A \equiv \Omega$.
4. *If* $\Gamma \vdash \lambda x : A.M : B$ *then there exists* C *such that* $\Gamma, x : A \vdash M : C$ *and* $B \equiv A \to C$.

Contexts

$$(\langle\rangle)\ \frac{}{\langle\rangle \vdash \text{valid}} \qquad (\text{ctx}_T)\ \frac{\Gamma \vdash \text{valid}}{\Gamma, x : A \vdash \text{valid}} \qquad (\text{ctx}_P)\ \frac{\Gamma \vdash \varphi : \Omega}{\Gamma, p : \varphi \vdash \text{valid}}$$

$$(\text{ctx}_E)\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma, e : M =_A N \vdash \text{valid}}$$

$$(\text{var}_T)\ \frac{\Gamma \vdash \text{valid}}{\Gamma \vdash x : A}\ (x : A \in \Gamma) \qquad (\text{var}_P)\ \frac{\Gamma \vdash \text{valid}}{\Gamma \vdash p : \varphi}\ (p : \varphi \in \Gamma)$$

$$(\text{var}_E)\ \frac{\Gamma \vdash \text{valid}}{\Gamma \vdash e : M =_A N}\ (e : M =_A N \in \Gamma)$$

Terms

$$(\bot)\ \frac{\Gamma \vdash \text{valid}}{\Gamma \vdash \bot : \Omega} \qquad (\supset)\ \frac{\Gamma \vdash \varphi : \Omega \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \varphi \supset \psi : \Omega}$$

$$(\text{app}_T)\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad (\lambda_T)\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

Proofs

$$(\text{app}_P)\ \frac{\Gamma \vdash \delta : \varphi \supset \psi \quad \Gamma \vdash \epsilon : \varphi}{\Gamma \vdash \delta\epsilon : \psi} \qquad (\lambda_P)\ \frac{\Gamma, p : \varphi \vdash \delta : \psi}{\Gamma \vdash \lambda p : \varphi.\delta : \varphi \supset \psi}$$

$$(\text{conv}_P)\ \frac{\Gamma \vdash \delta : \varphi \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \delta : \psi}\ (\varphi \overset{*}{\leftrightarrow} \psi)$$

Paths

$$(\text{ref})\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{ref}\,(M) : M =_A M} \qquad (\supset^*)\ \frac{\Gamma \vdash P : \varphi =_\Omega \varphi' \quad \Gamma \vdash Q : \psi =_\Omega \psi'}{\Gamma \vdash P \supset^* Q : \varphi \supset \psi =_\Omega \varphi' \supset \psi'}$$

$$(\text{univ})\ \frac{\Gamma \vdash \delta : \varphi \supset \psi \quad \Gamma \vdash \epsilon : \psi \supset \varphi}{\Gamma \vdash \text{univ}_{\varphi,\psi}\,(\delta, \epsilon) : \varphi =_\Omega \psi}$$

$$(\text{plus})\ \frac{\Gamma \vdash P : \varphi =_\Omega \psi}{\Gamma \vdash P^+ : \varphi \supset \psi} \qquad (\text{minus})\ \frac{\Gamma \vdash P : \psi =_\Omega \psi}{\Gamma \vdash P^- : \psi \supset \varphi}$$

$$(\lambda\!\!\lambda\!\!\lambda)\ \frac{\begin{array}{c}\Gamma, x : A, y : A, e : x =_A y \vdash P : Mx =_B Ny \\ \Gamma \vdash M : A \to B \quad \Gamma \vdash N : A \to B\end{array}}{\Gamma \vdash \lambda\!\!\lambda\!\!\lambda e : x =_A y.P : M =_{A \to B} N}$$

$$(\text{app}_E)\ \frac{\Gamma \vdash P : M =_{A \to B} M' \quad \Gamma \vdash Q : N =_A N' \quad \Gamma \vdash N : A \quad \Gamma \vdash N' : A}{\Gamma \vdash P_{NN'}Q : MN =_B M'N'}$$

$$(\text{conv}_E)\ \frac{\Gamma \vdash P : M =_A N \quad \Gamma \vdash M' : A \quad \Gamma \vdash N' : A}{\Gamma \vdash P : M' =_A N'}\ (M \overset{*}{\leftrightarrow} M', N \overset{*}{\leftrightarrow} N')$$

Figure 2 Rules of Deduction of λoe.

5. *If* $\Gamma \vdash MN : A$ *then there exists* B *such that* $\Gamma \vdash M : B \to A$ *and* $\Gamma \vdash N : B$.
6. *If* $\Gamma \vdash p : \varphi$, *then there exists* ψ *such that* $p : \psi \in \Gamma$ *and* $\varphi \overset{*}{\leftrightarrow} \psi$.
7. *If* $\Gamma \vdash \lambda p : \varphi.\delta : \psi$, *then there exists* χ *such that* $\Gamma, p : \varphi \vdash \delta : \chi$ *and* $\psi \overset{*}{\leftrightarrow} (\varphi \supset \chi)$.
8. *If* $\Gamma \vdash \delta\epsilon : \varphi$ *then there exists* ψ *such that* $\Gamma \vdash \delta : \psi \supset \varphi$ *and* $\Gamma \vdash \epsilon : \psi$.
9. *If* $\Gamma \vdash e : M =_A N$, *then there exist* M', N' *such that* $e : M' =_A N' \in \Gamma$ *and* $M \overset{*}{\leftrightarrow} M'$, $N \overset{*}{\leftrightarrow} N'$.
10. *If* $\Gamma \vdash \mathrm{ref}\,(M) : N =_A P$, *then we have* $\Gamma \vdash M : A$ *and* $M \overset{*}{\leftrightarrow} N \overset{*}{\leftrightarrow} P$.
11. *If* $\Gamma \vdash P \supset^* Q : \varphi =_A \psi$, *then there exist* φ_1, φ_2, ψ_1, ψ_2 *such that* $\Gamma \vdash P : \varphi_1 =_\Omega \psi_1$, $\Gamma \vdash Q : \varphi_2 =_\Omega \psi_2$, $\varphi \overset{*}{\leftrightarrow} (\varphi_1 \supset \psi_1)$, $\psi \overset{*}{\leftrightarrow} (\varphi_2 \supset \psi_2)$, *and* $A \equiv \Omega$.
12. *If* $\Gamma \vdash \mathrm{univ}_{\varphi,\psi}\,(\delta, \epsilon) : \chi =_A \theta$, *then we have* $\Gamma \vdash \delta : \varphi \supset \psi$, $\Gamma \vdash \epsilon : \psi \supset \varphi$, $\chi \overset{*}{\leftrightarrow} \varphi$, $\theta \overset{*}{\leftrightarrow} \psi$ *and* $A \equiv \Omega$.
13. *If* $\Gamma \vdash \lambda\!\!\lambda\!\!\lambda e : x =_A y.P : M =_B N$ *then there exists* C *such that* $\Gamma, x : A, y : A, e : x =_A y \vdash P : Mx =_C Ny$ *and* $B \equiv A \to C$.
14. *If* $\Gamma \vdash P_{MM'}Q : N =_A N'$, *then there exist* B, F *and* G *such that* $\Gamma \vdash P : F =_{B \to A} G$, $\Gamma \vdash Q : M =_B M'$, $N \overset{*}{\leftrightarrow} FM$ *and* $N' \overset{*}{\leftrightarrow} GM'$.
15. *If* $\Gamma \vdash P^+ : \varphi$, *then there exist* ψ, χ *such that* $\Gamma \vdash P : \psi =_\Omega \chi$ *and* $\varphi \overset{*}{\leftrightarrow} (\psi \supset \chi)$.
16. *If* $\Gamma \vdash P^- : \varphi$, *there exist* ψ, χ *such that* $\Gamma \vdash P : \psi =_\Omega \chi$ *and* $\varphi \overset{*}{\leftrightarrow} (\chi \supset \psi)$.

Proof. Induction on derivations. ◀

2.2.2 Substitutions

▶ **Definition 14.** Let Γ and Δ be contexts. A *substitution from* Δ *to* Γ[1], $\sigma : \Delta \Rightarrow \Gamma$, is a substitution whose domain is $\mathrm{dom}\,\Gamma$ such that:

- for every term variable $x : A \in \Gamma$, we have $\Delta \vdash \sigma(x) : A$;
- for every proof variable $p : \varphi \in \Gamma$, we have $\Delta \vdash \sigma(p) : \varphi[\sigma]$;
- for every path variable $e : M =_A N \in \Gamma$, we have $\Delta \vdash \sigma(e) : M[\sigma] =_A N[\sigma]$.

▶ **Lemma 15** (Well-Typed Substitution). *If* $\Gamma \vdash \mathcal{J}$, $\sigma : \Delta \Rightarrow \Gamma$ *and* $\Delta \vdash \mathrm{valid}$, *then* $\Delta \vdash \mathcal{J}[\sigma]$.

Proof. Induction on derivations. ◀

▶ **Definition 16.** If $\rho, \sigma : \Delta \Rightarrow \Gamma$ and τ is a path substitution whose domain is the term variables in $\mathrm{dom}\,\Gamma$, then we write $\tau : \sigma = \rho : \Delta \Rightarrow \Gamma$ iff, for each variable $x : A \in \Gamma$, we have $\Delta \vdash \tau(x) : \sigma(x) =_A \rho(x)$.

▶ **Lemma 17** (Path Substitution). *If* $\tau : \sigma = \rho : \Delta \Rightarrow \Gamma$ *and* $\Gamma \vdash M : A$ *and* $\Delta \vdash \mathrm{valid}$, *then* $\Delta \vdash M\{\tau : \sigma = \rho\} : M[\sigma] =_A M[\rho]$.

Proof. Induction on derivations. ◀

▶ **Proposition 18** (Subject Reduction). *If* $\Gamma \vdash s : T$ *and* $s \twoheadrightarrow t$ *then* $\Gamma \vdash t : T$.

Proof. It is sufficient to prove the case $s \to t$. The proof is by a case analysis on $s \to t$, using the Generation, Well-Typed Substitution and Path Substitution Lemmas. ◀

[1] These have also been called *context morphisms*, for example in Hoffman [5].

2.2.3 Canonicity

▶ **Definition 19** (Canonical Object).

- The *canonical propositions*, are given by the grammar

$$\theta ::= \bot \mid \theta \supset \theta$$

- A *canonical proof* is one of the form $\lambda p : \varphi . \delta$.
- A *canonical path* is one of the form $\mathrm{ref}\,(M)$, $\mathrm{univ}_{\phi,\psi}\,(\delta, \epsilon)$ or $\lambda\!\!\lambda\!\!\lambda e : x =_A y.P$.

▶ **Lemma 20.** *Suppose φ reduces to a canonical proposition θ, and $\varphi \overset{*}{\leftrightarrow} \psi$. Then ψ reduces to θ.*

Proof. This follows from the fact that $\rightarrow$ satisfies the diamond property, and every canonical proposition θ is a normal form. ◀

2.2.4 Neutral Expressions

▶ **Definition 21** (Neutral). The *neutral* terms, paths and proofs are given by the grammar

$$\begin{array}{lccl} \text{Neutral term} & M_n & ::= & x \mid M_n N \\ \text{Neutral proof} & \delta_n & ::= & p \mid P_n^+ \mid P_n^- \mid \delta_n \epsilon \\ \text{Neutral path} & P_n & ::= & e \mid P_n \supset^* Q \mid Q \supset^* P_n \mid (P_n)_{MN} Q \end{array}$$

3 Examples

We present two examples illustrating the way that proofs and paths behave in PHOML. In each case, we compare the example with the same construction performed in cubical type theory.

3.1 Functions Respect Logical Equivalence

As discussed in the introduction, every function of type $\Omega \to \Omega$ that can be constructed in PHOML must respect logical equivalence. This fact can actually be proved in PHOML, in the following sense: there exists a proof δ of

$$f : \Omega \to \Omega, x : \Omega, y : \Omega, p : x \supset y, q : y \supset x \vdash \delta : fx \supset fy$$

and a proof of $fy \supset fx$ in the same context. Together, these can be read as a proof of "if $f : \Omega \to \Omega$ and x and y are logically equivalent, then fx and fy are logically equivalent".

Specifically, take

$$\delta \overset{\text{def}}{=} (\mathrm{ref}\,(f)_{xy}\,\mathrm{univ}_{x,y}\,(p,q))^+ \quad .$$

Note that this is not possible in Martin-Löf Type Theory.

In cubical type theory, we can construct a term δ such that

$$f : \mathsf{Prop} \to \mathsf{Prop}, x : \mathsf{Prop}, y : \mathsf{Prop}, p : x.1 \to y.1, q : y.1 \to x.1 \vdash \delta : (fx).1 \to (fy).1$$

In fact, we can go further and prove that equality of propositions is equal to logical equivalence. That is, we can prove

$$\mathsf{Path}\,U\,(\mathsf{Path}\,\mathsf{Prop}\,x\,y)\,((x.1 \to y.1) \times (y.1 \to x.1)) \quad .$$

3.2 Computation with Paths

Let $\top \overset{\text{def}}{=} \bot \supset \bot$. Using propositional extensionality, we can construct a path of type $\top = \top \supset \top$, and hence a proof of $\top \supset (\top \supset \top)$. Now, there are two canonical proofs of $\top \supset (\top \supset \top)$. We might strongly expect that the proof we have constructed is the one that we used to construct the path $\top = \top \supset \top$, but let us check that this is the one that our computation rules produce.

We define

$$\top := \bot \supset \bot, \quad \iota := \lambda p : \bot . p, \quad I := \lambda x : \Omega . x, \quad F := \lambda x : \Omega . \top \supset x, \quad H := \lambda h . h\top \enspace .$$

Let Γ be the context

$$\Gamma \overset{\text{def}}{=} x : \Omega, y : \Omega, e : x =_\Omega y \enspace .$$

Then we have

$$\begin{array}{ll}
\Gamma \vdash \lambda p : \top \supset x . e^+(p\iota) & : (\top \supset x) \supset y \\
\Gamma \vdash \lambda m : y . \lambda n : \top . e^- m & : y \supset (\top \supset x) \\
\Gamma \vdash \mathsf{univ}\left(\lambda p : \top \supset x . e^- m, \lambda m : y . \lambda n : \top . e^- m\right) & : (\top \supset x) =_\Omega y
\end{array}$$

Let $P \equiv \mathsf{univ}\,(\lambda p : \top \supset x . e^+(p\iota), \lambda m : y . \lambda n : \top . e^- m)$. Then

$$\begin{array}{lll}
\therefore \vdash \lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P & : F =_{\Omega \to \Omega} I & (3) \\
\therefore \vdash (\mathrm{ref}\,(H))_{FI}(\lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P) & : (\top \supset \top) =_\Omega \top & (4) \\
\therefore \vdash ((\mathrm{ref}\,(H))_{FI}(\lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P))^- & : \top \supset (\top \supset \top) & (5)
\end{array}$$

And now we compute:

$$\begin{array}{l}
\quad ((\mathrm{ref}\,(H))_{FI}(\lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P))^- \\
\twoheadrightarrow ((h\top)\{h := \lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P : F = I\})^- \\
\equiv ((\lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P)_{\top\top}(\mathrm{ref}\,(\top)))^- \\
\to (P[x := \top, y := \top, e := \mathrm{ref}\,(\top)])^- \\
\equiv \mathsf{univ}\left(\lambda p : \top \supset \top . \mathrm{ref}\,(\top)^+\,(p\iota), \lambda m : \top . \lambda n : \top . \mathrm{ref}\,(\top)^-\, m\right)^- \\
\to \lambda m : \top . \lambda n : \top . \mathrm{ref}\,(\top)^-\, m
\end{array}$$

Therefore, given proofs $\delta, \epsilon : \top$, we have

$$((\mathrm{ref}\,(H))_{FI}(\lambda\!\!\!\lambda\!\!\!\lambda e : x =_\Omega y . P))^- \delta\epsilon \twoheadrightarrow \delta \enspace .$$

Thus, the construction gives a proof of $\top \supset (\top \supset \top)$ which, given two proofs of $\top$, selects the first. We could have anticipated this: consider the context $\Delta \overset{\text{def}}{=} X : \Omega, Y : \Omega, p : X$. By replacing in our example some occurrences of $\top$ with X and others with Y, and replacing ι with p, we can obtain a path

$$Y =_\Omega X \supset Y$$

and hence a proof of $Y \supset (X \supset Y)$. By parametricity, any proof that we can construct in the context Δ of this proposition must return the left input.

3.2.1 Comparison with Cubical Type Theory

In cubical type theory, we say that a type A is a *proposition* iff any two terms of type A are propositionally equal; that is, there exists a path between any two terms of type A. Let

$$\mathsf{isProp}\,(A) \overset{\mathrm{def}}{=} \Pi x, y : A.\mathsf{Path}\,A\,x\,y$$

and let Prop be the type of all types in U that are propositions:

$$\mathsf{Prop} \overset{\mathrm{def}}{=} \Sigma X : U.\mathsf{isProp}\,(X) \quad .$$

Let $\bot$ be any type in the universe U that is a proposition; that is, there exists a term of type $\mathsf{isProp}\,(\bot)$. ($\bot$ may be the empty type, but we do not require this in what follows.)

Define

$$\top := \bot \to \bot$$

Then there exists a term $\top_{\mathsf{Prop}}$ of type $\mathsf{isProp}\,(\top)$ (we omit the details). Define

$$I := \lambda X : \mathsf{Prop}.X.1, \quad F := \lambda X : \mathsf{Prop}.\top \to X.1, \quad H := \lambda h.h(\top, \top_{\mathsf{Prop}})$$

Then we have

$$\vdash \top : U \quad \vdash I : \mathsf{Prop} \to U \quad \vdash F : \mathsf{Prop} \to U \quad \vdash H : (\mathsf{Prop} \to U) \to U$$

From the fact that univalence is provable in cubical type theory [3], we can construct a term Q such that

$$\vdash Q : \mathsf{Path}\,(\mathsf{Prop} \to U)\,I\,F \quad .$$

Hence we have

$$\vdash \langle i \rangle H(Qi) : \mathsf{Path}\,U\,HI\,HF$$

which is definitionally equal to

$$\vdash \langle i \rangle H(Qi) : \mathsf{Path}\,U\,\top \to \top\,\top$$

From this, we can apply transport to create a term $Q' : \top \to \top \to \top$. Applying this to any terms $\delta, \epsilon : \top$ gives a term that is definitionally equal to

$$Q'\delta\epsilon = \mathsf{mapid}_\top\,\mathsf{mapid}_\top\,\delta$$

where mapid represents transport across the trivial path:

$$\mathsf{mapid}_A\,t \overset{\mathrm{def}}{=} \mathsf{comp}^i\,A\,[]\,t \qquad (i \text{ does not occur in } A) \quad .$$

(For the details of the calculation, see Appendix A.)

The cubical model of type theory given in [2] validates the equations $\mathsf{mapid}_X\,x = x$ and $Q'\delta\epsilon = \delta$. However, these are not definitional equalities in the version of cubical type theory given in [3].

4 Computable Expressions

We now proceed with the proof of canonicity for PHOML. Our proof follows the lines of the Girard-Tait reducibility method [10]: we define what it means to be a *computable* term (proof, path) of a given type (proposition, equation), and prove: (1) every typable expression is computable (2) every computable expression reduces to either a neutral or a canonical expression. In particular, every closed computable expression reduces to a canonical expression.

In this section, we use E, F, S and T as metavariables that range over expressions. In each case, either E and F are terms and S and T are types; or E and F are proofs and S and T are propositions; or E and F are paths and S and T are equations.

▶ **Definition 22** (Computable Expression)**.** We define the relation $\models E : T$, read "E is a computable expression of type T", as follows.

- $\models \delta : \bot$ iff δ reduces to a neutral proof.
- For θ and θ' canonical propositions, $\models \delta : \theta \supset \theta'$ iff, for all ϵ such that $\models \epsilon : \theta$, we have $\models \delta\epsilon : \theta'$.
- If φ reduces to the canonical proposition θ, then $\models \delta : \varphi$ iff $\models \delta : \theta$.
- $\models P : \varphi =_\Omega \psi$ iff $\models P^+ : \varphi \supset \psi$ and $\models P^- : \psi \supset \varphi$.
- $\models P : M =_{A \to B} M'$ iff, for all Q, N, N' such that $\models N : A$ and $\models N' : A$ and $\models Q : N =_A N'$, then we have $\models P_{NN'}Q : MN =_B M'N'$.
- $\models M : A$ iff $\models M\{\} : M =_A M$.

Note that the last three clauses define $\models M : A$ and $\models P : M =_A N$ simultaneously by recursion on A.

▶ **Definition 23** (Computable Substitution)**.** Let σ be a substitution with domain $\mathrm{dom}\,\Gamma$. We write $\models \sigma : \Gamma$ and say that σ is a *computable* substitution on Γ iff, for every entry $z : T$ in Γ, we have $\models \sigma(z) : T[\sigma]$.

We write $\models \tau : \rho = \sigma : \Gamma$, and say τ is a *computable* path substitution between ρ and σ, iff, for every term variable entry $x : A$ in Γ, we have $\models \tau(x) : \rho(x) =_A \sigma(x)$.

▶ **Lemma 24** (Conversion)**.** *If* $\models E : S$ *and* $S \overset{*}{\leftrightarrow} T$ *then* $\models E : T$.

Proof. This follows easily from the definition and Lemma 20. ◀

▶ **Lemma 25** (Expansion)**.** *If* $\models F : T$ *and* $E \to F$ *then* $\models E : T$.

Proof. An easy induction, using the fact that call-by-name reduction respects path substitution (Lemma 8). ◀

▶ **Lemma 26** (Reduction)**.** *If* $\models E : T$ *and* $E \to F$ *then* $\models F : T$.

Proof. An easy induction, using the fact that call-by-name reduction is confluent (Lemma 7). ◀

▶ **Definition 27.** We introduce a closed term c_A for every type A such that $\models c_A : A$.

$$c_\Omega \overset{\text{def}}{=} \bot$$
$$c_{A \to B} \overset{\text{def}}{=} \lambda x : A.c_B$$

▶ **Lemma 28.** $\models c_A : A$

Proof. An easy induction on A. ◀

▶ **Lemma 29** (Weak Normalization).

1. *If $\models \delta : \phi$ then δ reduces to either a neutral proof or canonical proof.*
2. *If $\models P : M =_A N$ then P reduces either to a neutral path or canonical path.*
3. *If $\models M : A$ then M reduces either to a canonical proposition or a λ-term.*

Proof. We prove by induction on the canonical proposition θ that, if $\models \delta : \theta$, then δ reduces to a neutral proof or a canonical proof of θ.

If $\models \delta : \bot$ then δ reduces to a neutral proof. Now, suppose $\models \delta : \theta \supset \theta'$. Then $\models \delta p : \theta'$, so δp reduces to either a neutral proof or canonical proof by the induction hypothesis. This reduction must proceed either by reducing δ to a neutral proof, or reducing δ to a λ-proof then β-reducing.

We then prove by induction on the type A that, if $\models P : M =_A N$, then P reduces to a neutral path or a canonical path. The two cases are straightforward.

Now, suppose $\models M : A$, i.e. $\models M\{\} : M =_A M$. Let $A \equiv A_1 \to \cdots \to A_n \to \Omega$. Then

$$\models M\{\}_{c_{A_1} c_{A_1}} c_{A_1} \{\}_{c_{A_2} c_{A_2}} \cdots c_{A_n} \{\} : M c_{A_1} \cdots c_{A_n} =_\Omega M c_{A_1} \cdots c_{A_n} \quad .$$

Therefore, $M c_{A_1} \cdots c_{A_n}$ reduces to a canonical proposition. The reduction must consist either in reducing M to a canonical proposition (if $n = 0$), or reducing M to a λ-expression then performing a β-reduction. ◀

▶ **Lemma 30.** *If $\models M : A \to B$ then M reduces to a λ-expression.*

Proof. Similar to the last paragraph of the previous proof. ◀

▶ **Lemma 31.** *For any term φ that reduces to a canonical proposition, we have $\models \mathrm{ref}\,(\varphi) : \varphi =_\Omega \varphi$.*

Proof. In fact we prove that, for any terms M and φ such that φ reduces to a canonical proposition, we have $\models \mathrm{ref}\,(M) : \varphi =_\Omega \varphi$.

It is sufficient to prove the case where φ is a canonical proposition. We must show that $\models \mathrm{ref}\,(M)^+ : \varphi \supset \varphi$ and $\models \mathrm{ref}\,(M)^- : \varphi \supset \varphi$. So let $\models \delta : \varphi$. Then $\models \mathrm{ref}\,(M)^+\,\delta : \varphi$ and $\models \mathrm{ref}\,(M)^-\,\delta : \varphi$ by Expansion (Lemma 25), as required. ◀

▶ **Lemma 32.** *$\models \varphi : \Omega$ if and only if φ reduces to a canonical proposition.*

Proof. If $\models \varphi : \Omega$ then $\models \varphi\{\}^+ : \varphi \supset \varphi$. Therefore $\varphi \supset \varphi$ reduces to a canonical proposition, and so φ must reduce to a canonical proposition.

Conversely, suppose φ reduces to a canonical proposition θ. We have $\varphi\{\} \twoheadrightarrow \theta\{\}$, and $\theta\{\} \twoheadrightarrow \mathrm{ref}\,(\theta)$ for every canonical proposition θ. Therefore, $\models \varphi\{\} : \varphi =_\Omega \varphi$ by Expansion (Lemma 25). Hence $\models \varphi : \Omega$. ◀

▶ **Lemma 33.** *If δ is a neutral proof and φ reduces to a canonical proposition, then $\models \delta : \varphi$.*

Proof. It is sufficient to prove the case where φ is a canonical proposition. The proof is by induction on φ.

If $\varphi \equiv \bot$, then $\models \delta : \bot$ immediately from the definition.

If $\varphi \equiv \psi \supset \chi$, then let $\models \epsilon : \psi$. We have that $\delta\epsilon$ is neutral, hence $\models \delta\epsilon : \chi$ by the induction hypothesis. ◀

▶ **Lemma 34.** *Let $\models M : A$ and $\models N : A$. If P is a neutral path, then $\models P : M =_A N$.*

Proof. The proof is by induction on A.

For $A \equiv \Omega$: we have that P^+ and P^- are neutral proofs, and M and N reduce to canonical propositions (by Lemma 32), so $\models P^+ : M \supset N$ and $\models P^- : N \supset M$ by Lemma 33, as required.

For $A \equiv B \to C$: let $\models L : B$, $\models L' : B$ and $\models Q : L =_B L'$. Then we have $\models ML : C$, $\models NL' : C$ and $P_{LL'}Q$ is a neutral path, hence $\models P_{LL'}Q : ML =_C NL'$ by the induction hypothesis, as required. ◀

▶ **Lemma 35.** *If* $\models M : A$ *then* $\models \mathrm{ref}\,(M) : M =_A M$.

Proof. If $A \equiv \Omega$, this is just Lemma 31.

So suppose $A \equiv B \to C$. Using Lemma 30, Reduction (Lemma 26) and Expansion (Lemma 25), we may assume that M is a λ-term. Let $M \equiv \lambda y : D.N$.

Let $\models L : B$ and $\models L' : B$ and $\models P : L =_B L'$. We must show that

$$\models \mathrm{ref}\,(\lambda y : D.N)_{LL'}\,P : (\lambda y : D.N)L =_C (\lambda y : D.N)L' \enspace .$$

By Expansion and Conversion, it is sufficient to prove

$$\models N\{y := P : L = L'\} : N[y := L] =_C N[y := L'] \enspace .$$

We have that $\models (\lambda y : D.N)\{\} : \lambda y : D.N =_{B \to C} \lambda y : D.N$, and so

$$\models (\lambda\!\!\lambda\!\!\lambda e : y =_D y'.N\{y := e : y = y'\})_{LL'}P : (\lambda y : D.N)L =_C (\lambda y : D.N)L' \enspace ,$$

and the result follows by Reduction and Conversion. ◀

▶ **Lemma 36.** *If* $\models P : \varphi =_\Omega \varphi'$ *and* $\models Q : \psi =_\Omega \psi'$ *then* $\models P \supset^* Q : \varphi \supset \psi =_\Omega \varphi' \supset \psi'$.

Proof. By Reduction (Lemma 26) and Expansion (Lemma 25), we may assume that P and Q are either neutral, or have the form $\mathrm{ref}\,(-)$ or $\mathrm{univ}_{-,-}\,(-,-)$ or $\lambda\!\!\lambda\!\!\lambda e : x =_A y.-$.

We cannot have that P reduces to a $\lambda\!\!\lambda\!\!\lambda$-path; for let φ' reduce to the canonical proposition $\theta_1 \supset \cdots \supset \theta_n \supset \bot$. Then we have

$$\models P^+ p q_1 \cdots q_n : \bot$$

and so $P^+ p q_1 \cdots q_n$ must reduce to a neutral path. Similarly, Q cannot reduce to a $\lambda\!\!\lambda\!\!\lambda$-path.

If either P or Q is neutral then $P \supset^* Q$ is neutral, and the result follows from Lemma 34.

Otherwise, let $\models \delta : \varphi \supset \psi$ and $\epsilon \models \varphi'$. We must show that $\models (P \supset^* Q)^+ \delta\epsilon : \psi'$.

If $P \equiv \mathrm{ref}\,(M)$ and $Q \equiv \mathrm{ref}\,(N)$, then we have

$$(P \supset^* Q)^+ \delta\epsilon \to \mathrm{ref}\,(M \supset N)^+\,\delta\epsilon \to \delta\epsilon \enspace .$$

Now, $\models P^- \epsilon : \varphi$, hence $\models \epsilon : \varphi$ by Reduction, and so $\models \delta\epsilon : \psi$. Therefore, $\models Q^+(\delta\epsilon) : \psi'$, and hence by Reduction $\models \delta\epsilon : \psi'$ as required.

If $P \equiv \mathrm{ref}\,(M)$ and $Q \equiv \mathrm{univ}_{N,N'}\,(\chi, \chi')$, then we have

$$\begin{aligned}(P \supset^* Q)^+ \delta\epsilon &\to \mathrm{univ}_{M \supset N, M \supset N'}\,(\lambda pq.\chi(pq), \lambda pq.\chi'(pq))^+\,\delta\epsilon \\ &\to (\lambda pq.\chi(pq))\delta\epsilon \\ &\twoheadrightarrow \chi(\delta\epsilon)\end{aligned}$$

We have $\models P^- \epsilon : \varphi$, hence $\models \epsilon : \varphi$ by Reduction, and so $\models \delta\epsilon : \psi$. Therefore, $\models Q^+(\delta\epsilon) : \psi'$, and hence by Reduction $\models \chi(\delta\epsilon) : \psi'$ as required.

The other two cases are similar. ◀

▶ **Lemma 37.** *If* $\models \delta : \phi \supset \psi$ *and* $\models \epsilon : \psi \supset \phi$ *then* $\models \mathrm{univ}_{\phi,\psi}\,(\delta, \epsilon) : \phi =_\Omega \psi$.

Proof. We must show that $\models \mathrm{univ}_{\phi,\psi}\,(\delta, \epsilon)^+ : \phi \supset \psi$ and $\models \mathrm{univ}_{\phi,\psi}\,(\delta, \epsilon)^- : \psi \supset \phi$. These follow from the hypotheses, using Expansion (Lemma 25). ◀

5 Proof of Canonicity

▶ **Theorem 38.**
1. *If* $\Gamma \vdash \mathcal{J}$ *and* $\models \sigma : \Gamma$, *then* $\models \mathcal{J}[\sigma]$.
2. *If* $\Gamma \vdash M : A$ *and* $\models \tau : \rho = \sigma : \Gamma$, *then* $\models M\{\tau : \rho = \sigma\} : M[\rho] =_A M[\sigma]$.

Proof. The proof is by induction on derivations. Most cases are straightforward, using the lemmas from Section 4. We deal with one case here, the rule (λ_T).

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

1. We must show that

$$\models \lambda x : A.M[\sigma] : A \to B \ .$$

So let $\models Q : N =_A N'$. Define the path substitution τ by

$$\tau(x) \equiv Q, \qquad \tau(y) \equiv \mathsf{ref}\,(\sigma(y)) \ \ (y \in \operatorname{dom}\Gamma)$$

Then we have $\models \tau : (\sigma, x := N) = (\sigma, x := N') : \Gamma, x : A$, and so the induction hypothesis gives

$$\models M\{\tau\} : M[\sigma, x := N] =_B M[\sigma, x := N']$$

We observe that $M\{\tau\} \equiv M[\sigma]\{x := Q : N = N'\}$ (Lemma 3), and so by Expansion (Lemma 25) and Conversion (Lemma 24) we have

$$\models (\lambda x : A.M[\sigma])\{\}_{NN'}Q : (\lambda x : A.M[\sigma])N =_B (\lambda x : A.M[\sigma])N'$$

as required.

2. We must show that

$$\models \lambda\!\!\!\lambda\!\!\!\lambda e : x =_A y.M\{\tau : \rho = \sigma, x := e : x = y\} : \lambda x : A.M[\rho] =_{A \to B} \lambda x : A.M[\sigma] \ .$$

So let $\models P : N =_A N'$. The induction hypothesis gives

$$\models M\{\tau : \rho = \sigma, x := P : N = N'\} : M[\rho, x := N] =_B M[\sigma, x := N'] \ ,$$

and so we have

$$\begin{aligned}&\models (\lambda\!\!\!\lambda\!\!\!\lambda e : x =_A y.M\{\tau : \rho = \sigma, x := e : x = y\})_{NN'}P \\ &\quad : (\lambda x : A.M[\rho])N =_B (\lambda x : A.M[\sigma])N'\end{aligned}$$

by Expansion and Conversion, as required. ◀

▶ **Corollary 39.** *Let* Γ *be a context in which no term variables occur.*
1. *If* $\Gamma \vdash \delta : \phi$ *then* δ *reduces to a neutral proof or canonical proof.*
2. *If* $\Gamma \vdash P : M =_A N$ *then* P *reduces to a neutral path or canonical path.*

Proof. Let id be the substitution $\Gamma \Rightarrow \Gamma$ such that $\mathsf{id}(x) \overset{\text{def}}{=} x$. If $\Gamma \vdash$ valid then $\models \mathsf{id} : \Gamma$ using Lemmas 33 and 34.

Therefore, if $\Gamma \vdash E : T$ then $\models E[\mathsf{id}] : T[\mathsf{id}]$, that is, $\models E : T$. Hence E reduces to a neutral expression or canonical expression. ◀

▶ **Corollary 40** (Canonicity). *Let Γ be a context with no term variables.*

1. *If $\Gamma \vdash \delta : \bot$ then δ reduces to a neutral proof.*
2. *If $\Gamma \vdash \delta : \phi \supset \psi$ then δ reduces either to a neutral proof, or a proof $\lambda p : \phi'.\epsilon$ where $\phi \overset{*}{\leftrightarrow} \phi'$ and $\Gamma, p : \phi \vdash \epsilon : \psi$.*
3. *If $\Gamma \vdash P : \phi =_\Omega \psi$ then P reduces either to a neutral path; or to $\mathrm{ref}\,(\chi)$ where $\phi \overset{*}{\leftrightarrow} \psi \overset{*}{\leftrightarrow} \chi$; or to $\mathrm{univ}_{\phi',\psi'}\,(\delta, \epsilon)$ where $\phi \overset{*}{\leftrightarrow} \phi'$, $\psi \overset{*}{\leftrightarrow} \psi'$, $\Gamma \vdash \delta : \phi \supset \psi$ and $\Gamma \vdash \epsilon : \psi \supset \phi$.*
4. *If $\Gamma \vdash P : M =_{A \to B} M'$ then P reduces either to a neutral path; or to $\mathrm{ref}\,(N)$ where $M \overset{*}{\leftrightarrow} M' \overset{*}{\leftrightarrow} N$; or to $\lambda\!\!\lambda\!\!\lambda e : x =_A y.Q$ where $\Gamma, x : A, y : A, e : x =_A y \vdash Q : Mx =_B M'y$.*

Proof. A closed expression cannot be neutral, so from the previous corollary every typed closed expression must reduce to a canonical expression. We now apply case analysis to the possible forms of canonical expression, and use the Generation Lemma. ◀

▶ **Corollary 41** (Conistency). *There is no δ such that $\vdash \delta : \bot$.*

▶ **Note 42.** We have not proved canonicity for terms. However, we can observe that PHOML restricted to terms and types is just the simply-typed lambda calculus with one atomic type Ω and two constants $\bot$ and $\supset$; and our reduction relation restricted to this fragment is head reduction. Canonicity for this system is already a well-known result (see e.g. [4, Ch. 4]).

6 Conclusion and Future Work

We have presented a system with propositional extensionality, and shown that it satisfies the property of canonicity. This gives hope that it will be possible to find a computation rule for homotopy type theory that satisfies canonicity, and that does not involve extending the type theory, either with a nominal extension of the syntax as in cubical type theory or otherwise.

We now intend to do the same for stronger and stronger systems, getting ever closer to full homotopy type theory. The next steps will be:

- a system with infinitely many propositional universes Ω_0, Ω_1, …, where each equations $M =_A N$ is an object of a universe Ω_n for some n, allowing us to form propositions such as $M =_A N \supset N =_A M$.
- a system with universal quantification over the types A, allowing us to form propositions such as $\forall x : A.x =_A x$ and $\forall x, y : A.x =_A y \supset y =_A x$

Ultimately, we hope to approach full homotopy type theory. The study of how the reduction relation and its properties change as we move up and down this hierarchy of systems should reveal facts about computing with univalence that might be lost when working in a more complex system such as homotopy type theory or cubical type theory.

References

1 Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher-dimensional type theory. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 680–693. ACM, 2017. `doi:10.1145/3093333.3009861`.

2 Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2013.107`.

3 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016. URL: `http://arxiv.org/abs/1611.02108`.

4 Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

5 Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

6 Simon Huber. Canonicity for cubical type theory. *CoRR*, abs/1607.04156, 2016. `arXiv:1607.04156`.

7 Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. In John Field and Michael Hicks, editors, *POPL*, pages 337–348. ACM, 2012. `doi:10.1145/2103656.2103697`.

8 Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

9 Andrew Polonsky. Internalization of extensional equality. *CoRR*, abs/1401.1148, 2014. `arXiv:1401.1148`.

10 W. W. Tait. Intensional iinterpretation of ffunctional of finite type i. *Journal of Symbolic Logic*, 32:198–212, 1967.

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

A Calculation in Cubical Type Theory

We can prove that, if X is a proposition, then the type $\Sigma f : \top \to X.Path\,X\,x\,(fI)$ is contractible (we omit the details). Let $e[X, x, p]$ be the term such that

$$\begin{aligned}&X : \mathsf{Prop}, x : X.1, p : \Sigma f : \top \to X.1.\mathsf{Path}\,X.1\,x\,(fI)\\ \vdash\, &e[X, x, p] : \mathsf{Path}\,(\Sigma f : \top \to X.1.\mathsf{Path}\,X.1\,x\,(fI))\,\langle \lambda t : \top.x, 1_{X.1}\rangle\,p\end{aligned}$$

Let $\mathsf{step}_1[X, x] \overset{\text{def}}{=} \langle\langle \lambda t : \top.x, 1_{X.1}\rangle, \lambda p : \Sigma f : \top \to X.1.\mathsf{Path}\,X.1\,x\,(fI).e[X, x, p]\rangle$. Then

$$X : \mathsf{Prop}, x : X.1 \vdash \mathsf{step}_1[X, x] : \mathsf{isContr}(\Sigma f : \top \to X.1.\mathsf{Path}\,X.1\,x\,(fI)) \enspace .$$

Let $\mathsf{step}_2[X] \equiv \lambda x : X.1.\mathsf{step}_1[X, x]$. Then

$$X : \mathsf{Prop} \vdash \mathsf{step}_2[X] : \mathsf{isEquiv}\,(\top \to X.1)\,X.1\,(\lambda f : \top \to X.1.fI) \enspace .$$

Let $E[X] \equiv \langle \lambda f : \top \to X.1.fI, \mathsf{step}_2[X]\rangle$. Then

$$X : \mathsf{Prop} \vdash E[X] : \mathsf{Equiv}\,(\top \to X.1)\,X.1$$

From this equivalence, we want to get a path from $\top \to X.1$ to $X.1$ in U. We apply the proof of univalence in [3]

Let $P[X] \equiv \langle i\rangle\mathsf{Glue}[(i = 0) \mapsto (\top \to X.1, E[X]), (i = 1) \mapsto (X.1, equiv^k X.1)]X.1$. Then

$$X : \mathsf{Prop} \vdash P[X] : \mathsf{Path}\,U\,(\top \to X.1)\,X.1$$

Let $Q \equiv \langle i\rangle\lambda x : \mathsf{Prop}.P[X]i$. Then

$$\vdash Q : \mathsf{Path}\,(\mathsf{Prop} \to U)\,F\,I$$

This is the term in cubical type theory that corresponds to $\lambda\!\!\lambda\!\!\lambda e : x =_\Omega y.P$ in PHOML (formula 3). We now construct terms corresponding to formulas (4) and (5):

$$\vdash \langle i \rangle H(Qi) : \mathsf{Path}\, U\, (\top \to \top)\, \top$$

$$\vdash \lambda x : \top.\mathsf{comp}^i(H(Q(1-i)))[]x : \top \to \top \to \top$$

Let us write output for this term:

$$\mathsf{output} \stackrel{\text{def}}{=} \lambda x : \top.\mathsf{comp}^i(H(Q(1-i)))[]x \ .$$

And we calculate (using the notation from [3] section 6.2):

$$\begin{aligned}
&\mathsf{output} \\
&= \lambda x : \top.\mathsf{comp}^i(Q(1-i)\top)[]x \\
&= \lambda x : \top.\mathsf{comp}^i(P[\top](1-i))[]x \\
&= \lambda x : \top.\mathsf{comp}^i(\mathsf{Glue}[(i=1) \mapsto (\top \to \top, E[\top]), (i=0) \mapsto (\top, \mathsf{equiv}^k \top)]\top)[]x \\
&= \lambda x : \top.\mathsf{glue}[1_{\mathbb{F}} \mapsto t_1]a_1 \\
&= \lambda x : \top.t_1 \\
&= \lambda x : \top.(\mathsf{equiv}\, E[\top]\, []\, \mathsf{mapid}_\top\, x).1 \\
&= \lambda x : \top.(\mathsf{contr}(\mathsf{step}_1[\top, \mathsf{mapid}_\top\, x])[]).1 \\
&= \lambda x : \top.(\mathsf{comp}^i \\
&\qquad (\Sigma f : \top \to \top.\mathsf{Path}\, \top\, (\mathsf{mapid}_\top\, x)\, (f I)) \\
&\qquad [] \\
&\qquad \langle \lambda t : \top.\mathsf{mapid}_\top\, x, 1_{mapid_\top(x)} \rangle).1 \\
&= \lambda x : \top.\mathsf{mapid}_{\top \to \top}\, (\lambda y : \top.\mathsf{mapid}_\top\, x)
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\mathsf{output}\, m\, n \\
&= \mathsf{mapid}_{\top \to \top}\, (\lambda y : \top.\mathsf{mapid}_\top\, m) n \\
&\equiv (\mathsf{comp}^i(\top \to \top)[](\lambda_{:}\top.\mathsf{mapid}_\top\, m)) n \\
&= \mathsf{mapid}_\top\, \mathsf{mapid}_\top\, m
\end{aligned}$$

B Proof of Confluence

The proof follows the same lines as the proof given in [8].

▶ **Definition 43** (Parallel One-Step Reduction)**.** Define the notion of *parallel one-step reduction* $\triangleright$ by the rules given in Figure 3. Let $\triangleright^*$ be the transitive closure of $\triangleright$.

▶ **Lemma 44.**
1. *If $E \to F$ then $E \triangleright F$.*
2. *If $E \twoheadrightarrow F$ then $E \triangleright^* F$.*
3. *If $E \triangleright^* F$ then $E \twoheadrightarrow F$.*

Proof. These are easily proved by induction. ◀

Our reason for defining $\triangleright$ is that it satisfies the diamond property:

Reflexivity

$$\frac{}{E \rhd E}$$

Reduction on Terms

$$\frac{}{(\lambda x : A.M)N \rhd M[x := N]} \quad \frac{M \rhd M'}{MN \rhd M'N} \quad \frac{\varphi \rhd \varphi' \quad \psi \rhd \psi'}{\varphi \supset \psi \rhd \varphi' \supset \psi'}$$

Reduction on Proofs

$$\frac{}{(\lambda p : \varphi.\delta)\epsilon \rhd \delta[p := \epsilon]} \quad \frac{}{\mathrm{ref}\,(\varphi)^+ \rhd \lambda p : \varphi.p} \quad \frac{}{\mathrm{ref}\,(\varphi)^- \rhd \lambda p : \varphi.p}$$

$$\frac{}{\mathrm{univ}_{\varphi,\psi}\,(\delta,\epsilon)^+ \rhd \delta} \quad \frac{}{\mathrm{univ}_{\varphi,\psi}\,(\delta,\epsilon)^- \rhd \epsilon}$$

$$\frac{\delta \rhd \delta'}{\delta\epsilon \rhd \delta'\epsilon} \quad \frac{P \rhd Q}{P^+ \rhd Q^+} \quad \frac{P \rhd Q}{P^- \rhd Q^-}$$

Reduction on Paths

$$\frac{}{(\lambda\!\!\lambda\!\!\lambda e : x =_A y.P)_{MN} Q \rhd P[x := M, y := N, e := Q]}$$

$$\frac{}{\mathrm{ref}\,(\lambda x : A.M)_{NN'}\, P \rhd M\{x := P : N = N'\}}$$

$$\frac{}{\mathrm{ref}\,(\varphi) \supset^* \mathrm{ref}\,(\psi) \rhd \mathrm{ref}\,(\varphi \supset \psi)}$$

$$\frac{}{\mathrm{ref}\,(\varphi) \supset^* \mathrm{univ}_{\psi,\chi}\,(\delta,\epsilon) \rhd \mathrm{univ}_{\varphi\supset\psi,\varphi\supset\chi}\,(\lambda p : \varphi \supset \psi.\lambda q : \varphi.\delta(pq), \lambda p : \varphi \supset \chi.\lambda q : \varphi.\epsilon(pq))}$$

$$\frac{}{\mathrm{univ}_{\varphi,\psi}\,(\delta,\epsilon) \supset^* \mathrm{ref}\,(\chi) \rhd \mathrm{univ}_{\varphi\supset\chi,\psi\supset\chi}\,(\lambda p : \varphi \supset \chi.\lambda q : \psi.p(\epsilon q), \lambda p : \psi \supset \chi.\lambda q : \varphi.p(\delta q))}$$

$$\frac{}{\begin{array}{l}\mathrm{univ}_{\varphi,\psi}\,(\delta,\epsilon) \supset^* \mathrm{univ}_{\varphi',\psi'}\,(\delta',\epsilon') \\ \rhd \mathrm{univ}_{\varphi\supset\varphi',\psi\supset\psi'}\,(\lambda p : \varphi \supset \varphi'.\lambda q : \psi.\delta'(p(\epsilon q)), \lambda p : \psi \supset \psi'.\lambda q : \varphi.\epsilon'(p(\delta q)))\end{array}}$$

$$\frac{P \rhd P'}{P_{MN}Q \rhd P'_{MN}Q} \quad \frac{M \rhd N}{\mathrm{ref}\,(M)_{NN'}\, P \rhd \mathrm{ref}\,(M')_{NN'}\, P} \quad \frac{P \rhd P' \quad Q \rhd Q'}{P \supset^* Q \rhd P' \supset^* Q}$$

Figure 3 Parallel One-Step Reduction.

▶ **Lemma 45** (Diamond Property). *If $E \rhd F$ and $E \rhd G$ then there exists an expression H such that $F \rhd H$ and $G \rhd H$.*

Proof. The proof is by case analysis on $E \rhd F$ and $E \rhd G$. We give the details for one case here:

$$\mathrm{ref}\,(\phi) \supset^* \mathrm{ref}\,(\psi) \rhd \mathrm{ref}\,(\phi \supset \psi) \text{ and } \mathrm{ref}\,(\phi) \supset^* \mathrm{ref}\,(\psi) \rhd \mathrm{ref}\,(\phi') \supset^* \mathrm{ref}\,(\psi')$$

where $\phi \rhd \phi'$ and $\psi \rhd \psi'$. In this case, we have $\mathrm{ref}\,(\phi \supset \psi) \rhd \mathrm{ref}\,(\phi' \supset \psi')$ and $\mathrm{ref}\,(\phi') \supset^* \mathrm{ref}\,(\psi') \rhd \mathrm{ref}\,(\phi' \supset \psi')$. ◀

▶ **Corollary 46.** *If $E \rhd^* F$ and $E \rhd^* G$ then there exists H such that $F \rhd^* H$ and $G \rhd^* H$.*

▶ **Corollary 47.** *If $E \twoheadrightarrow F$ and $E \twoheadrightarrow G$ then $F \twoheadrightarrow H$ and $G \twoheadrightarrow H$.*

Proof. Immediate from the previous corollary and Lemma 44. ◀

On Natural Deduction for Herbrand Constructive Logics II: Curry-Howard Correspondence for Markov's Principle in First-Order Logic and Arithmetic

Federico Aschieri[1]
Institut für Diskrete Mathematik und Geometrie
Technische Universität Wien
Wiedner Hauptstraße 8-10/104, 1040, Vienna, Austria

Matteo Manighetti[2]
Institut für Diskrete Mathematik und Geometrie
Technische Universität Wien
Wiedner Hauptstraße 8-10/104, 1040, Vienna, Austria

Abstract

Intuitionistic first-order logic extended with a restricted form of Markov's principle is constructive and admits a Curry-Howard correspondence, as shown by Herbelin. We provide a simpler proof of that result and then we study intuitionistic first-order logic extended with unrestricted Markov's principle. Starting from classical natural deduction, we restrict the excluded middle and we obtain a natural deduction system and a parallel Curry-Howard isomorphism for the logic. We show that proof terms for existentially quantified formulas reduce to a list of individual terms representing all possible witnesses. As corollary, we derive that the logic is Herbrand constructive: whenever it proves any existential formula, it proves also an Herbrand disjunction for the formula. Finally, using the techniques just introduced, we also provide a new computational interpretation of Arithmetic with Markov's principle.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases Markov's Principle, first-order logic, natural deduction, Curry-Howard

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.4

1 Introduction

Markov's Principle was introduced by Markov in the context of his theory of Constructive Recursive Mathematics (see [13]). Its original formulation is tied to Arithmetic: it states that 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 a n such that $f(n) = 0$. Markov's original argument for justifying 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 $f(n) = 0$ and will *effectively* recognize it as a witness.

Markov's principle is readily formalized in Heyting Arithmetic as the axiom scheme

$$\neg\neg\exists\alpha^{\mathtt{N}} P \to \exists\alpha^{\mathtt{N}} P$$

[1] Funded by the Austrian Science Fund FWF Lise Meitner grant M 1930–N35.
[2] Funded by the Vienna Science Fund WWTF project VRG12-004.

where P is a primitive recursive predicate [12]. When added to Heyting Arithmetic, Markov's principle gives rise to a *constructive* system, that is, one enjoying the disjunction and the existential witness property [12] (if a disjunction is derivable, one of the disjuncts is derivable too, and if an existential statement is derivable, so it is one instance of it). Furthermore, witnesses for any provable existential formula can be effectively computed using either Markov's unbounded search and Kleene's realizability [9] or much more efficient functional interpretations [7, 3].

1.1 Markov's Principle in First-Order Logic

The very shape of Markov's principle makes it also a purely logical principle, namely an instance of the double negation elimination axiom. But in pure logic, what exactly should Markov's principle correspond to? In particular, what class of formulas should P be restricted to? Since Markov's principle was originally understood as a constructive principle, it is natural to restrict P as little as possible, while maintaining the logical system as constructive as possible. As proven by Herbelin [8], it turns out that asking that P is propositional and with no implication $\rightarrow$ symbols guarantees that intuitionistic logic extended with such a version of Markov's principle is constructive. The proof of this result employs a Curry-Howard isomorphism based on a mechanism for raising and catching exceptions. As opposed to the aforementioned functional interpretations of Markov's principle, Herbelin's calculus is fully isomorphic to an intuitionistic logic: there is a perfect match between reduction steps at the level of programs and detour eliminations at the level of proofs. Moreover, witnesses for provable existential statements are computed by the associated proof terms. Nevertheless, as we shall later show, the mechanism of throwing exceptions plays no role during these computations: intuitionistic reductions are entirely enough for computing witnesses.

A question is now naturally raised: as no special mechanism is required for witness computation using Herbelin's restriction of Markov's principle, can the first be further relaxed so that the second becomes stronger as well as computationally *and* constructively meaningful? Allowing the propositional matrix P to contain implication destroys the constructivity of the logic. It turns out, however, that *Herbrand constructivity* is preserved. An intermediate logic is called Herbrand constructive if it enjoys a strong form of Herbrand's Theorem [5, 4]: for *every* provable formula $\exists\alpha\, A$, the logic proves as well an Herbrand disjunction

$$A[m_1/\alpha] \vee \ldots \vee A[m_k/\alpha]$$

So the Markov principle we shall interpret in this paper is

$$\mathsf{MP} : \neg\neg\exists\alpha\, P \rightarrow \exists\alpha\, P \qquad (P \text{ propositional formula})$$

and show that when added to intuitionistic first-order logic, the resulting system is Herbrand constructive. This is the most general form of Markov's principle that allows a significant constructive interpretation: we shall show how to non-trivially compute lists of witnesses for provable existential formulas thanks to an exception raising construct and a parallel computation operator. MP can also be used in conjunction with negative translations to compute Herbrand disjunctions in classical logic, something which is not possible with Herbelin's form of Markov's principle.

1.2 Restricted Excluded Middle

The Curry-Howard correspondence we present here is by no means an ad hoc construction, only tailored for Markov's principle. It is a simple restriction of the Curry-Howard correspondence for classical first-order logic introduced in [4], where classical reasoning is formalized by the excluded middle inference rule:

$$\frac{\Gamma, a : \forall x\, \mathsf{Q} \vdash u : C \qquad \Gamma, a : \exists x\, \neg \mathsf{Q} \vdash v : C}{\Gamma \vdash u \parallel_a v : C}\ \mathsf{EM}$$

It is enough to restrict the conclusion C of this rule to be an existential statement $\exists x \mathsf{P}$, with P propositional, and the Q in the premises $\forall x\, \mathsf{Q}, \exists x\, \neg \mathsf{Q}$ to be propositional. We shall show that the rule is intuitionistically equivalent to MP. With our approach, strong normalization is just inherited and the transition from classical logic to intuitionistic logic with MP is smooth and natural.

1.3 Markov's Principle in Arithmetic

We shall also provide a computational interpretation of Heyting Arithmetic with MP. The system is constructive and witnesses for provable existential statements can be computed. This time, we shall restrict the excluded middle as formalized in [2] and we shall directly obtain the desired Curry-Howard correspondence. As a matter of fact, the interpretation of MP in Arithmetic ends up to be a simplification of the methods we use in first-order logic, because the decidability of atomic formulas greatly reduces parallelism and eliminates case distinction on the truth of atomic formulas.

1.4 Plan of the Paper

In Section 2, we provide a simple computational interpretation of first-order intuitionistic logic extended with Herbelin's restriction of Markov's principle. We also show that the full Markov principle MP cannot be proved in that system. In Section 3, we provide a Curry-Howard correspondence for intuitionistic logic with MP, by restricting the excluded middle, and show that the system is Herbrand constructive. In Section 4, we extend the Curry-Howard to Arithmetic with MP and show that the system becomes again constructive.

2 Herbelin's Restriction of Markov's Principle

In [8] Herbelin introduced a Curry-Howard isomorphism for an extended intuitionistic logic. By employing exception raising operators and new reduction rules, he proved that the logic is constructive and can derive the axiom scheme

$$\mathsf{HMP} : \neg\neg\exists\alpha\, \mathsf{P} \to \exists\alpha\, \mathsf{P} \qquad (\mathsf{P} \text{ propositional and } \to \text{ not occurring in } \mathsf{P})$$

Actually, Herbelin allowed P also to contain existential quantifiers, but in that case the axiom scheme is intuitionistically equivalent to $\neg\neg\exists\alpha_1 \dots \exists\alpha_n\, \mathsf{P} \to \exists\alpha_1 \dots \exists\alpha_n\, \mathsf{P}$, again with P propositional and $\to$ not occurring in P. All of the methods of our paper apply to this case as well, but for avoiding trivial details, we keep the present HMP.

Our first goal is to show that HMP has a simpler computational interpretation and to provide a straightforward proof that, when added on top of first-order intuitionistic logic, HMP gives rise to a constructive system. In particular, we show that the ordinary Prawitz reduction rules for intuitionistic logic and thus the standard Curry-Howard isomorphism [6] are enough for extracting witnesses for provable existential formulas. The crucial insight, as we shall see, is that HMP can never actually appear in the head of a closed proof term having existential type. It thus plays no computational role in computing witnesses; it plays rather a logical role, in that it may be used to prove the correctness of the witnesses.

We start by fixing the first-order language of logical formulas.

▶ **Definition 1** (Formula Language). The language $\mathcal{L}$ of formulas is defined as follows.

1. The terms of $\mathcal{L}$ are inductively defined as either variables $\alpha, \beta, \ldots$ or constants c or expressions of the form $\mathsf{f}(t_1, \ldots, t_n)$, with f a function constant of arity n and $t_1, \ldots, t_n \in \mathcal{L}$.
2. There is a countable set of *predicate symbols.* The *atomic formulas* of $\mathcal{L}$ are all the expressions of the form $\mathcal{P}(t_1, \ldots, t_n)$ such that $\mathcal{P}$ is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms of $\mathcal{L}$. We assume to have a 0-ary predicate symbol $\bot$ which represents falsity.
3. The formulas of $\mathcal{L}$ are built from atomic formulas of $\mathcal{L}$ by the logical constants $\vee, \wedge, \rightarrow, \forall, \exists$, with quantifiers ranging over variables $\alpha, \beta, \ldots$: if A, B are formulas, then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $\forall \alpha\, A$, $\exists \alpha\, B$ are formulas. The logical negation $\neg A$ can be introduced, as usual, as an abbreviation of the formula $A \rightarrow \bot$.
4. *Propositional formulas* are the formulas whose only logical constants are $\wedge, \vee, \rightarrow$; we say that a propositional formula is *negative* whenever $\vee$ does not occur in it. Propositional formulas will be denoted as $\mathsf{P}, \mathsf{Q} \ldots$ (possibly indexed). Formulas of the form $\forall \alpha_1 \ldots \forall \alpha_n\, \mathsf{P}$, with P propositional and negative, will be called *simply universal.*

To achieve our goals, we now consider the usual natural deduction system for intuitionistic first-order logic [11, 6], in the language $\mathcal{L}$, to which we add HMP. Accordingly, we add to the associated lambda calculus the constants $\mathcal{M}_{\mathsf{P}} : \neg\neg\exists\alpha\, \mathsf{P} \rightarrow \exists \alpha\, \mathsf{P}$. The resulting Curry-Howard system is called $\mathsf{IL} + \mathsf{HMP}$ and is presented in Figure 1.

The reduction rules for $\mathsf{IL} + \mathsf{HMP}$ presented in Figure 2 are just the ordinary ones of lambda calculus. On the other hand, $\mathcal{M}_{\mathsf{P}}$ has no computational content and thus no associated reduction rule. Of course, the strong normalization of $\mathsf{IL} + \mathsf{HMP}$ holds by virtue of the result for standard intuitionistic Curry-Howard.

▶ **Theorem 2.** *The system* $\mathsf{IL} + \mathsf{HMP}$ *is strongly normalizing.*

As we shall see in Theorem 5, the reason why HMP cannot be appear in the head of a closed proof term having existential type is that its premise $\neg\neg\exists\alpha\, \mathsf{P}$ is never classically valid, let alone provable in intuitionistic logic.

▶ **Proposition 3.** *Assume that the symbol* $\rightarrow$ *does not occur in the propositional formula* P*. Then* $\neg\neg\exists\alpha\, \mathsf{P}$ *is not classically provable.*

Proof. We provide a semantical argument. The formula $\neg\neg\exists\alpha\, \mathsf{P}$ is classically provable if and only if it is classically valid and thus if and only if $\exists\alpha\, \mathsf{P}$ is classically valid. For every such a formula, we shall exhibit a model falsifying it. Consider the model $\mathfrak{M}$ where every n-ary predicate is interpreted as the empty n-ary relation. We show by induction on the complexity of the formula P that $\mathsf{P}^{\mathfrak{M}} = \bot$ for every assignment of individuals to the free variables of P, and therefore $(\exists\alpha\, \mathsf{P})^{\mathfrak{M}} = \bot$.

- If P is atomic, then by definition of $\mathfrak{M}$, we have $P^{\mathfrak{M}} = \bot$ for every assignment of the variables.
- If $\mathsf{P} = \mathsf{P}_1 \wedge \mathsf{P}_2$, then since by induction $\mathsf{P}_1^{\mathfrak{M}} = \bot$, $(\mathsf{P}_1 \wedge \mathsf{P}_2)^{\mathfrak{M}} = \bot$
- If $\mathsf{P} = \mathsf{P}_1 \vee \mathsf{P}_2$, then since by induction $\mathsf{P}_1^{\mathfrak{M}} = \bot$ and $\mathsf{P}_2^{\mathfrak{M}} = \bot$, $(\mathsf{P}_1 \vee \mathsf{P}_2)^{\mathfrak{M}} = \bot$ ◀

In order to derive constructivity of $\mathsf{IL} + \mathsf{HMP}$, we shall just have to inspect the normal forms of proof terms. Our main argument, in particular, will use the following well-known syntactic characterization of the shape of proof terms.

Grammar of Untyped Proof Terms

$t, u, v ::= \ x \mid tu \mid tm \mid \lambda x\, u \mid \lambda\alpha\, u \mid \langle t, u\rangle \mid u\pi_0 \mid u\pi_1 \mid \iota_0(u) \mid \iota_1(u) \mid t[x.u, y.v] \mid (m, t) \mid t[(\alpha, x).u] \mid \mathtt{H}^{\perp\to\mathsf{P}} \mid \mathcal{M}_{\mathsf{P}}$

where m ranges over terms of the first-order language of formulas $\mathcal{L}$, x over proof-term variables, α over first-order variables.

Contexts With Γ we denote contexts of the form $x_1 : A_1, \ldots, x_n : A_n$, where each x_i is a proof-term variable, and $x_i \neq x_j$ for $i \neq j$.

Axioms $\Gamma, x : A \vdash x : A$ $\qquad \Gamma \vdash \mathcal{M}_{\mathsf{P}} : \neg\neg\exists\alpha\,\mathsf{P} \to \exists\alpha\,\mathsf{P}$ $\qquad \Gamma \vdash \mathtt{H}^{\perp\to\mathsf{P}} : \perp \to \mathsf{P}$

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

Implication $\dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$ $\quad \dfrac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x\, u : A \to B}$

Disjunction Introduction $\dfrac{\Gamma \vdash u : A}{\Gamma \vdash \iota_0(u) : A \vee B}$ $\quad \dfrac{\Gamma \vdash u : B}{\Gamma \vdash \iota_1(u) : A \vee B}$

Disjunction Elimination $\dfrac{\Gamma \vdash u : A \vee B \quad \Gamma, x : A \vdash w_1 : C \quad \Gamma, y : B \vdash w_2 : C}{\Gamma \vdash u\,[x.w_1, y.w_2] : C}$

Universal Quantification $\dfrac{\Gamma \vdash u : \forall\alpha A}{\Gamma \vdash um : A[m/\alpha]}$ $\quad \dfrac{\Gamma \vdash u : A}{\Gamma \vdash \lambda\alpha\, u : \forall\alpha A}$

where m is any term of the language $\mathcal{L}$ and α does not occur free in any formula B occurring in Γ.

Existential Quantification $\dfrac{\Gamma \vdash u : A[m/\alpha]}{\Gamma \vdash (m, u) : \exists\alpha A}$ $\quad \dfrac{\Gamma \vdash u : \exists\alpha A \quad \Gamma, x : A \vdash t : C}{\Gamma \vdash u\,[(\alpha, x).t] : C}$

where α is not free in C nor in any formula B occurring in Γ.

Figure 1 Term Assignment Rules for IL + HMP.

Reduction Rules for IL

$(\lambda x.u)t \mapsto u[t/x]$

$(\lambda\alpha.u)m \mapsto u[m/\alpha]$

$\langle u_0, u_1\rangle\pi_i \mapsto u_i,$ for i=0,1

$\iota_i(u)[x_1.t_1, x_2.t_2] \mapsto t_i[u/x_i],$ for i=0,1

$(m, u)[(\alpha, x).v] \mapsto v[m/\alpha][u/x],$ for each term m of $\mathcal{L}$

Figure 2 Reduction Rules for IL + HMP.

▶ **Proposition 4** (Head of a Proof Term). *Every proof-term of* IL + HMP *is of the form*

$$\lambda z_1 \ldots \lambda z_n . r u_1 \ldots u_k$$

where

- *r is either a variable or a constant or a term corresponding to an introduction rule: $\lambda x.t$, $\lambda\alpha.t$, $\langle t_1, t_2\rangle$, $\iota_i(t)$, (m, t)*
- *$u_1, \ldots u_k$ are either proof terms, first order terms, or one of the following expressions corresponding to elimination rules: π_i, $[x.w_1, y.w_2]$, $[(\alpha, x).t]$.*

Proof. Standard. ◀

We are now able to prove that IL + HMP is constructive.

▶ **Theorem 5** (Constructivity of $\mathsf{IL} + \mathsf{HMP}$)**.**

1. *If* $\mathsf{IL} + \mathsf{HMP} \vdash t : \exists\alpha\, A$*, and* t *is in normal form, then* $t = (m, u)$ *and* $\mathsf{IL} + \mathsf{HMP} \vdash u : A[m/\alpha]$*.*
2. *If* $\mathsf{IL}+\mathsf{HMP} \vdash t : A \vee B$ *and* t *is in normal form, then either* $t = \iota_0(u)$ *and* $\mathsf{IL}+\mathsf{HMP} \vdash u : A$ *or* $t = \iota_1(u)$ *and* $\mathsf{IL} + \mathsf{HMP} \vdash u : B$*.*

Proof.

1. By Proposition 4, t must be of the form $ru_1 \dots u_k$. Let us consider the possible forms of r.
   - Since t is closed, r cannot be a variable.
   - We show that r cannot be $\mathcal{M}_\mathsf{P}$. If r were $\mathcal{M}_\mathsf{P} : \neg\neg\exists x\, \mathsf{P} \to \exists\alpha\, \mathsf{P}$ for some P, then $\mathsf{IL} + \mathsf{MP} \vdash u_1 : \neg\neg\exists\alpha\, \mathsf{P}$. Since $\mathsf{IL} + \mathsf{HMP}$ is contained in classical logic, we have that $\neg\neg\exists\alpha\, \mathsf{P}$ is classically provable. However we know from Proposition 3 that this cannot be the case, which is a contradiction.
   - We also show that r cannot be $\mathtt{H}^{\perp\to\mathsf{P}}$. Indeed, if r were $\mathtt{H}^{\perp\to\mathsf{P}}$ for some P, then $\mathsf{IL} + \mathsf{MP} \vdash u_1 : \perp$, which is a contradiction.
   - The only possibility is thus that r is one among $\lambda x.t$, $\lambda\alpha.t$, $\langle t_1, t_2\rangle$, $\iota_i(t)$, (m, t). In this case, k must be 0 as otherwise we would have a redex. This means that $t = r$ and thus $t = (m, u)$ with $\mathsf{IL} + \mathsf{HMP} \vdash u : A(m)$.
2. The proof goes along the same lines of case 1. ◀

Finally, we prove that $\mathsf{IL} + \mathsf{HMP}$ is not powerful enough to express full Markov's principle MP. Intuitively, the reason is that $\mathsf{IL} + \mathsf{HMP}$ is a constructive system and thus cannot be strong enough to interpret classical reasoning. This would indeed be the case if $\mathsf{IL} + \mathsf{HMP}$ proved MP, an axiom which complements very well negative translations.

▶ **Proposition 6.** $\mathsf{IL} + \mathsf{HMP} \nvdash \mathsf{MP}$.

Proof. Suppose for the sake of contradiction that $\mathsf{IL} + \mathsf{HMP} \vdash \mathsf{MP}$. Consider any proof in classical first-order logic of a simply existential statement $\exists\alpha\, \mathsf{P}$. By the Gödel-Gentzen negative translation (see [12]), we can then obtain an intuitionistic proof of $\neg\neg\exists\alpha\, \mathsf{P}^N$, where P^N is the negative translation of P, and thus $\mathsf{IL} + \mathsf{HMP} \vdash \exists\alpha\, \mathsf{P}^N$. By Theorem 5, there is a first-order term m such that $\mathsf{IL}+\mathsf{HMP} \vdash \mathsf{P}^N[m/\alpha]$. Since $\mathsf{P}^N[m/\alpha]$ is classically equivalent to $\mathsf{P}[m/\alpha]$, we would have a single witness for every classically valid simply existential statement. But this is not possible: consider for example the first-order language $\mathcal{L} = \{\mathsf{P}, a, b\}$ and the formula $F = (\mathsf{P}(a) \vee \mathsf{P}(b)) \to \mathsf{P}(\alpha)$ where P is an atomic predicate. Then the formula $\exists\alpha\, F$ is classically provable, but there is no term m such that $F[m/\alpha]$ is valid, let alone provable:

- it cannot be $m = a$, as it is shown by picking a model where P is interpreted as the set $\{b\}$
- it cannot be $m = b$, because we can interpret P as the set $\{a\}$. ◀

3 Full Markov Principle and Restricted Excluded Middle in First-Order Logic

In this section we describe the natural deduction system and Curry-Howard correspondence $\mathsf{IL} + \mathsf{EM}_1^-$, which arise by restricting the excluded-middle in classical natural deduction [4]. This computational system is based on delimited exceptions and a parallel operator. We will show that on one hand full Markov principle MP is provable in $\mathsf{IL} + \mathsf{EM}_1^-$ and, on the other hand, that $\mathsf{IL} + \mathsf{MP}$ derives all of the restricted classical reasoning that can be expressed in $\mathsf{IL}+\mathsf{EM}_1^-$, so that the two systems are actually equivalent. Finally, we show that the system $\mathsf{IL}+\mathsf{EM}_1^-$ is Herbrand constructive and that witnesses can effectively be computed.

In order to computationally interpret Markov's principle, we consider the rule EM_1^-, which is obtained by restricting the conclusion of the excluded middle EM_1 [4, 2] to be a simply existential formula:

$$\frac{\Gamma, a : \forall\alpha\,\mathsf{P} \vdash u : \exists\beta\,\mathsf{Q} \qquad \Gamma, a : \exists\alpha\,\neg\mathsf{P} \vdash v : \exists\beta\,\mathsf{Q}}{\Gamma \vdash u \parallel_a v : \exists\beta\,\mathsf{Q}} \ (\mathsf{P} \text{ and } \mathsf{Q} \text{ propositional and negative})$$

This inference rule is complemented by the axioms:

$$\Gamma, a : \forall\alpha\mathsf{P} \vdash \mathtt{H}_a^{\forall\alpha\mathsf{P}} : \forall\alpha\mathsf{P}$$

$$\Gamma, a : \exists\alpha\neg\mathsf{P} \vdash \mathtt{W}_a^{\exists\alpha\neg\mathsf{P}} : \exists\alpha\neg\mathsf{P}$$

These last two rules correspond respectively to a term making an *Hypothesis* and a term waiting for a *Witness* and these terms are put in communication via EM_1^-; the variable a in $u \parallel_a v$ represents their communication channel and all the free occurrences of a in u and v are bound in $u \parallel_a v$. In the terms $\mathtt{H}_a^{\forall\alpha A}$ and $\mathtt{W}_a^{\exists\alpha A}$ the free variables are a and those of A minus α. A term of the form $\mathtt{H}_a^{\forall\alpha\mathsf{P}}m$, with m first-order term, is said to be *active*, if its only free variable is a: it represents a raise operator which has been turned on. The term $u \parallel_a v$ supports an exception mechanism: u is the ordinary computation, v is the exceptional one and a is the communication channel. Raising exceptions is the task of the term $\mathtt{H}_a^{\forall\alpha\mathsf{P}}$, when it encounters a counterexample m to $\forall\alpha\,\mathsf{P}$; catching exceptions is performed by the term $\mathtt{W}_a^{\exists\alpha\neg\mathsf{P}}$. For this reason, the notation $\mathsf{raise}_a^{\forall\alpha A}$, as in [8], would also have been just fine, as well as the far less evocative notation $a^{\forall\alpha A}$. In first-order logic, however, there is an issue: when should an exception be thrown? Since the truth of atomic predicates depends on models, one cannot know. Therefore, each time $\mathtt{H}_a^{\forall\alpha\mathsf{P}}$ is applied to a term m, a new pair of parallel independent computational paths is created, according as to whether $\mathsf{P}[m/\alpha]$ is false or true. In one path the exception is thrown, in the other not, and the two computations will never join again. To render this computational behaviour, we add the rule EM_0 of propositional excluded middle over negative formulas

$$\frac{\Gamma, a : \neg\mathsf{P} \vdash u : A \qquad \Gamma, a : \mathsf{P} \vdash v : A}{\Gamma \vdash u \mid v : A}\ \mathsf{EM}_0$$

even if in principle it is derivable from EM_1^-; we also add the axiom

$$\Gamma, a : \mathsf{P} \vdash \mathtt{H}^{\mathsf{P}} : \mathsf{P}$$

Communication channel variables are not used in terms of the form $u \mid v$ because there is no useful information that can be raised by u and handed to v: the premises of EM_0 are completely void of positive information, because they are negative formulas; a cannot occur in u nor in v. But $u \mid v$ does not prevent the computation to go on, thanks to the permutation rules and because negative propositional assumptions do not stop the computation, that is, do not prevent normal proofs of existential statements to terminate with an $\exists$-introduction rule.

We call the resulting system $\mathsf{IL} + \mathsf{EM}_1^-$ (Figure 3) and present its reduction rules in Figure 4; they just form a restriction of the system $\mathsf{IL}+\mathsf{EM}$ described in [4]. The permutation rules for EM_1^- are left out, because the inference conclusion already behaves like a "data type", so there is no need to further transform it. The other reduction rules are based on the following definition, which formalizes the raise and catch mechanism.

Grammar of Untyped Proof Terms

$$t, u, v ::= \ x \mid tu \mid tm \mid \lambda x\, u \mid \lambda\alpha\, u \mid \langle t, u\rangle \mid u\pi_0 \mid u\pi_1 \mid \iota_0(u) \mid \iota_1(u) \mid t[x.u, y.v] \mid (m, t) \mid t[(\alpha, x).u]$$
$$\mid (u \mid v) \mid (u \parallel_a v) \mid \mathtt{H}_a^{\forall\alpha A} \mid \mathtt{W}_a^{\exists\alpha \mathsf{P}} \mid \mathtt{H}^{\mathsf{P}}$$

where m ranges over terms of $\mathcal{L}$, x over proof-term variables, α over first-order variables, a over hypothesis variables, A is either a negative formula or a simply universal formula, and P is negative. In the term $u \parallel_a v$ there must be some formula P, such that a occurs free in u only in subterms of the form $\mathtt{H}_a^{\forall\alpha\mathsf{P}}$ and a occurs free in v only in subterms of the form $\mathtt{W}_a^{\exists\alpha\mathsf{P}}$, and the occurrences of the variables in P different from α are free in both u and v.

Contexts With Γ we denote contexts of the form $x_1 : A_1, \ldots, x_n : A_n, a_1 : B_1, \ldots, a_m : B_m$,, where $x_1, \ldots, x_n$ are distinct proof-term variables and $a_1, \ldots, a_m$ are distinct EM hypothesis variables.

Axioms $\Gamma, x : A \vdash x : A$ $\qquad \Gamma, a : \forall\alpha\, A \vdash \mathtt{H}_a^{\forall\alpha A} : \forall\alpha\, A$ $\qquad \Gamma, a : \exists\alpha\,\mathsf{P} \vdash \mathtt{W}_a^{\exists\alpha\mathsf{P}} : \exists\alpha\,\mathsf{P}$

$\Gamma, a : \mathsf{P} \vdash \mathtt{H}^{\mathsf{P}} : \mathsf{P}$ $\qquad \Gamma \vdash \mathtt{H}^{\bot\to\mathsf{P}} : \bot \to \mathsf{P}$

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

Implication $$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x\, u : A \to B}$$

Disjunction Introduction $$\frac{\Gamma \vdash u : A}{\Gamma \vdash \iota_0(u) : A \vee B} \qquad \frac{\Gamma \vdash u : B}{\Gamma \vdash \iota_1(u) : A \vee B}$$

Disjunction Elimination $$\frac{\Gamma \vdash u : A \vee B \quad \Gamma, x : A \vdash w_1 : C \quad \Gamma, y : B \vdash w_2 : C}{\Gamma \vdash u\,[x.w_1, y.w_2] : C}$$

Universal Quantification $$\frac{\Gamma \vdash u : \forall\alpha A}{\Gamma \vdash um : A[m/\alpha]} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \lambda\alpha\, u : \forall\alpha A}$$

where m is any term of the language $\mathcal{L}$ and α does not occur free in any formula B occurring in Γ.

Existential Quantification $$\frac{\Gamma \vdash u : A[m/\alpha]}{\Gamma \vdash (m, u) : \exists\alpha A} \qquad \frac{\Gamma \vdash u : \exists\alpha A \quad \Gamma, x : A \vdash t : C}{\Gamma \vdash u\,[(\alpha, x).t] : C}$$

where α is not free in C nor in any formula B occurring in Γ.

EM$_0$ $$\frac{\Gamma, a : \neg\mathsf{P} \vdash u : C \quad \Gamma, a : \mathsf{P} \vdash v : C}{\Gamma \vdash u \mid v : C} \ (\mathsf{P} \text{ propositional and negative})$$

EM$_1^-$ $$\frac{\Gamma, a : \forall\alpha\,\mathsf{P} \vdash u : \exists\beta\ \mathsf{Q} \quad \Gamma, a : \exists\alpha\,\neg\mathsf{P} \vdash v : \exists\beta\ \mathsf{Q}}{\Gamma \vdash u \parallel_a v : \exists\beta\ \mathsf{Q}} \ (\mathsf{P}, \mathsf{Q} \text{ propositional and negative})$$

Figure 3 Term Assignment Rules for $\mathsf{IL} + \mathsf{EM}_1^-$.

▶ **Definition 7** (Exception Substitution)**.** Suppose v is any proof term and m is a term of $\mathcal{L}$. Then:

1. If every free occurrence of a in v is in a subterm of the form $\mathtt{W}_a^{\exists\alpha\mathsf{P}}$, we define

   $$v[a := m]$$

   as the term obtained from v by replacing each subterm $\mathtt{W}_a^{\exists\alpha\mathsf{P}}$ corresponding to a free occurrence of a in v by $(m, \mathtt{H}^{\mathsf{P}[m/\alpha]})$.
2. If every free occurrence of a in v is in a subterm of the form $\mathtt{H}_a^{\forall\alpha\mathsf{P}}$, we define

   $$v[a := m]$$

   as the term obtained from v by replacing each subterm $\mathtt{H}_a^{\forall\alpha\mathsf{P}}m$ corresponding to a free occurrence of a in v by $\mathtt{H}^{\mathsf{P}[m/\alpha]}$.

Reduction Rules for IL

$$(\lambda x.u)t \mapsto u[t/x] \qquad (\lambda\alpha.u)m \mapsto u[m/\alpha]$$

$$\langle u_0, u_1\rangle\pi_i \mapsto u_i,\ \text{ for i=0,1}$$

$$\iota_i(u)[x_1.t_1, x_2.t_2] \mapsto t_i[u/x_i],\ \text{ for i=0,1}$$

$$(m, u)[(\alpha, x).v] \mapsto v[m/\alpha][u/x],\ \text{ for each term } m \text{ of } \mathcal{L}$$

Permutation Rules for EM_0

$$(u \,|\, v)w \mapsto uw \,|\, vw$$

$$(u \,|\, v)\pi_i \mapsto u\pi_i \,|\, v\pi_i$$

$$(u \,|\, v)[x.w_1, y.w_2] \mapsto u[x.w_1, y]w_2 \,|\, v[x.w_1, y]w_2$$

$$(u \,|\, v)[(\alpha, x).w] \mapsto u[(\alpha, x).w] \,|\, v[(\alpha, x).w]$$

Reduction Rules for EM_1^-

$$u \parallel_a v \mapsto u,\ \text{ if } a \text{ does not occur free in } u$$

$$u \parallel_a v \mapsto v[a := m] \,|\, (u[a := m] \parallel_a v),\ \text{ whenever } u \text{ has some } \textit{active} \text{ subterm } \mathtt{H}_a^{\forall\alpha\mathsf{P}} m$$

Figure 4 Reduction Rules for $\mathsf{IL} + \mathsf{EM}_1^-$.

As we anticipated, our system is capable of proving the full Markov Principle MP and thus its particular case HMP.

▶ **Proposition 8** (Derivability of MP)**.** $\mathsf{IL} + \mathsf{EM}_1^- \vdash \mathsf{MP}$

Proof. First note that with the use of EM_0, we obtain that $\mathsf{IL} + \mathsf{EM}_1^- \vdash \mathsf{P} \vee \neg\mathsf{P}$ for any atomic formula P. Therefore $\mathsf{IL} + \mathsf{EM}_1^-$ can prove any propositional tautology, and in particular $\mathsf{IL} + \mathsf{EM}_1^- \vdash \mathsf{P} \vee \mathsf{Q} \leftrightarrow \neg(\neg\mathsf{P} \wedge \neg\mathsf{Q})$ for any propositional formulas P, Q, thus proving that each propositional formula is equivalent to a negative one.

Consider now any instance $\neg\neg\exists\alpha\,\mathsf{Q} \to \exists\alpha\,\mathsf{Q}$ of MP. Thanks to the previous observation, we obtain

$$\mathsf{IL} + \mathsf{EM}_1^- \vdash \big(\neg\neg\exists\alpha\,\mathsf{Q} \to \exists\alpha\,\mathsf{Q}\big) \leftrightarrow \big(\neg\neg\exists\alpha\,\mathsf{P} \to \exists\alpha\,\mathsf{P}\big)$$

for some negative formula P logically equivalent to Q. The following formal proof shows that $\mathsf{IL} + \mathsf{EM}_1^- \vdash \neg\neg\exists\alpha\,\mathsf{P} \to \exists\alpha\,\mathsf{P}$.

$$\dfrac{\dfrac{\dfrac{[\neg\neg\exists\alpha\,\mathsf{P}]_{(2)} \qquad \dfrac{\dfrac{[\exists\alpha\,\mathsf{P}]_{(1)} \qquad \dfrac{\dfrac{[\forall\alpha\,\neg\mathsf{P}]_{\mathsf{EM}_1^-}}{\neg\mathsf{P}} \qquad [P]_{\exists}}{\bot}}{\bot}\,\exists}{\neg\exists\alpha\,\mathsf{P}}\,(1)}{\dfrac{\bot}{\exists\alpha\,\mathsf{P}}} \qquad \dfrac{\dfrac{[\exists\alpha\,\neg\neg\mathsf{P}]_{\mathsf{EM}_1^-} \qquad \dfrac{[\mathsf{P}]_{\mathsf{EM}_0} \qquad \dfrac{\dfrac{[\neg\neg\,\mathsf{P}]_{\exists} \quad [\neg\mathsf{P}]_{\mathsf{EM}_0}}{\bot}}{\mathsf{P}}}{\mathsf{P}}\,\mathsf{EM}_0}{\mathsf{P}}\,\exists}{\exists\alpha\,\mathsf{P}}}{\dfrac{\exists\alpha\,\mathsf{P}}{\neg\neg\exists\alpha\,\mathsf{P} \to \exists\alpha\,\mathsf{P}}\,(2)}\,\mathsf{EM}_1^-$$

Finally, this implies $\mathsf{IL} + \mathsf{EM}_1^- \vdash \neg\neg\exists\alpha\,\mathsf{Q} \to \exists\alpha\,\mathsf{Q}$. ◀

Conversely, everything which is provable within our system can be proven by means of first-order logic with full Markov principle.

▶ **Theorem 9.** *If* $\mathsf{IL} + \mathsf{EM}_1^- \vdash F$*, then* $\mathsf{IL} + \mathsf{MP} \vdash F$.

Proof. We just need to show that $\mathsf{IL} + \mathsf{MP}$ can prove the rules EM_1^- and EM_0. For the case of EM_0, note that $\mathsf{IL} + \mathsf{MP} \vdash \neg\neg\mathsf{P} \to \mathsf{P}$ for all propositional formulas P, thanks to MP. Since for every propositional Q we have $\mathsf{IL} + \mathsf{MP} \vdash \neg\neg(\mathsf{Q} \vee \neg\mathsf{Q})$, we obtain $\mathsf{IL} + \mathsf{MP} \vdash \mathsf{Q} \vee \neg\mathsf{Q}$, and therefore $\mathsf{IL} + \mathsf{MP}$ can prove EM_0 by mean of an ordinary disjunction elimination.

In the case of EM_1^-, if we are given the proofs of $\begin{array}{c}\forall\alpha\,\mathsf{P}\\ \vdots\\ \exists\alpha\,\mathsf{C}\end{array}$ and $\begin{array}{c}\exists\alpha\neg\mathsf{P}\\ \vdots\\ \exists\alpha\mathsf{C}\end{array}$ in $\mathsf{IL} + \mathsf{MP}$, the following derivation shows a proof of $\exists\alpha\,\mathsf{C}$ in $\mathsf{IL} + \mathsf{MP}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[\forall\alpha\mathsf{P}]_{(1)}\\ \vdots\\ \exists\alpha\mathsf{C}\end{array}\quad[\neg\exists\alpha\mathsf{C}]_{(4)}}{\bot}}{\neg\forall\alpha\mathsf{P}}\,(1)\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[\exists\alpha\neg\mathsf{P}]_{(2)}\\ \vdots\\ \exists\alpha\mathsf{C}\end{array}\quad[\neg\exists\alpha\mathsf{C}]_{(4)}}{\bot}}{\neg\exists\alpha\neg\mathsf{P}}\,(2)\quad\dfrac{[\neg\mathsf{P}]_{(3)}}{\exists\alpha\,\neg\mathsf{P}}}{\bot}}{\neg\neg\mathsf{P}}\,(3)\quad \neg\neg\,\mathsf{P}\to\ \mathsf{P}}{\mathsf{P}}}{\forall\alpha\,\mathsf{P}}}{\bot}}{\neg\neg\exists\alpha\mathsf{C}}\,(4)\quad \neg\neg\exists\alpha\,\mathsf{C}\to\exists\alpha\,\mathsf{C}}{\exists\alpha\,\mathsf{C}}$$

◀

As in [4], all of our main results about witness extraction are valid not only for closed terms, but also for quasi-closed ones, which are those containing only pure universal assumptions.

▶ **Definition 10** (Quasi-Closed terms). An untyped proof term t is said to be *quasi-closed*, if it contains as free variables only hypothesis variables $a_1, \ldots, a_n$, such that each occurrence of them is in a term of the form $\mathtt{H}_{a_i}^{\forall\vec{\alpha}\mathsf{P}_i}$, where $\forall\vec{\alpha}\,\mathsf{P}_i$ is simply universal.

$\mathsf{IL} + \mathsf{EM}_1^-$ with the reduction rules in Figure 4 enjoys the Subject Reduction Theorem, as a particular case of the Subject Reduction for $\mathsf{IL} + \mathsf{EM}$ presented in [4].

▶ **Theorem 11** (Subject Reduction). *If* $\Gamma \vdash t : C$ *and* $t \mapsto u$*, then* $\Gamma \vdash u : C$.

No term of $\mathsf{IL} + \mathsf{EM}_1^-$ gives rise to an infinite reduction sequence [4].

▶ **Theorem 12** (Strong Normalization). *Every term typable in* $\mathsf{IL}+\mathsf{EM}_1^-$ *is strongly normalizing.*

We now update the characterization of proof-terms heads given in Proposition 4 to the case of $\mathsf{IL} + \mathsf{EM}_1^-$.

▶ **Theorem 13** (Head of a Proof Term). *Every proof term of* $\mathsf{IL} + \mathsf{EM}_1^-$ *is of the form:*

$$\lambda z_1 \ldots \lambda z_n . r u_1 \ldots u_k$$

where

- *r is either a variable x, a constant $\mathtt{H}^P$ or $\mathtt{H}_a^{\forall\alpha A}$ or $\mathtt{W}_a^{\exists\alpha\mathsf{P}}$ or an excluded middle term $u \parallel_a v$ or $u \,|\, v$, or a term corresponding to an introduction rule $\lambda x.t$, $\lambda\alpha.t$, $\langle t_1, t_2\rangle$, $\iota_i(t)$, (m, t)*

- *$u_1, \dots u_k$ are either lambda terms, first order terms, or one of the following expressions corresponding to elimination rules: π_i, $[x.w_1, y.w_2]$, $[(\alpha, x).t]$*

Proof. Standard. ◀

We now study the shape of the normal terms with the most simple types.

▶ **Proposition 14** (Normal Form Property). *Let $\mathsf{P}, \mathsf{P}_1, \dots \mathsf{P}_n$ be negative propositional formulas, $A_1, \dots, A_m$ simply universal formulas. Suppose that*

$$\Gamma = z_1 : \mathsf{P}_1, \dots z_n : \mathsf{P}_n, a_1 : \forall\alpha_1 A_1, \dots a_m : \forall\alpha_m A_m$$

and $\Gamma \vdash t : \exists\alpha\, \mathsf{P}$ or $\Gamma \vdash t : \mathsf{P}$, with t in normal form and having all its free variables among $z_1, \dots z_n, a_1, \dots a_m$. Then:

1. *Every occurrence in t of every term $\mathtt{H}_{a_i}^{\forall\alpha_i A_i}$ is of the active form $\mathtt{H}_{a_i}^{\forall\alpha_i A_i} m$, where m is a term of $\mathcal{L}$.*
2. *t cannot be of the form $u \parallel_a v$.*

Proof. We prove 1. and 2. simultaneously and by induction on t. There are several cases, according to the shape of t:

- $t = (m, u)$, $\Gamma \vdash t : \exists\alpha\, \mathsf{P}$ and $\Gamma \vdash u : \mathsf{P}[m/\alpha]$. We immediately get 1. by induction hypothesis applied to u, while 2. is obviously verified.
- $t = \lambda x\, u$, $\Gamma \vdash t : \mathsf{P} = \mathsf{Q} \to \mathsf{R}$ and $\Gamma, x : \mathsf{Q} \vdash u : \mathsf{R}$. We immediately get 1. by induction hypothesis applied to u, while 2. is obviously verified.
- $t = \langle u, v\rangle$, $\Gamma \vdash t : \mathsf{P} = \mathsf{Q} \wedge \mathsf{R}$, $\Gamma \vdash u : \mathsf{Q}$ and $\Gamma \vdash v : \mathsf{R}$. We immediately get 1. by induction hypothesis applied to u and v, while 2. is obviously verified.
- $t = u \,|\, v$, $\Gamma, a : \neg\mathsf{Q} \vdash u : \exists\alpha\, \mathsf{P}$ (resp. $u : \mathsf{P}$) and $\Gamma, a : \mathsf{Q} \vdash v : \exists\alpha\, \mathsf{P}$ (resp. $v : \mathsf{P}$). We immediately get 1. by induction hypothesis applied to u and v, while 2. is obviously verified.
- $t = u \parallel_a v$. We show that this is not possible. Note that a must occur free in u, otherwise t is not in normal form. Since $\Gamma, a : \forall\beta\, A \vdash u : \exists\alpha\, \mathsf{P}$, we can apply the induction hypothesis to u, and obtain that all occurrences of hypothetical terms must be active; in particular, this must be the case for the occurrences of $\mathtt{H}_a^{\forall\beta A}$, but this is not possible since t is in normal form.
- $t = \mathtt{H}_{a_i}^{\forall\alpha A_i}$. This case is not possible, for $\Gamma \vdash t : \exists\alpha\, \mathsf{P}$ or $\Gamma \vdash t : \mathsf{P}$.
- $t = \mathtt{W}_a^{\exists\alpha \mathsf{P}}$. This case is not possible, since $a : \exists\alpha\, \mathsf{P}$ is not in Γ.
- $t = \mathtt{H}^{\mathsf{P}}$. In this case, 1. and 2. are trivially true.
- t is obtained by an elimination rule and by Theorem 13 we can write it as $r\, t_1\, t_2 \dots t_n$. Notice that in this case r cannot correspond to an introduction rule neither be a term of the form $u \parallel_a v$, because of the induction hypothesis, nor $u \,|\, v$, because of the permutation rules and t being in normal form; moreover, r cannot be $\mathtt{W}_b^{\exists\alpha P}$, otherwise b would be free in t and $b \neq a_1, \dots, a_n$. We have now two remaining cases:
  1. $r = x_i$ (resp. $r = \mathtt{H}^{\mathsf{P}}$). Then, since $\Gamma \vdash x_i : \mathsf{P}_i$ (resp. $\Gamma \vdash \mathtt{H}^{\mathsf{P}} : \mathsf{P}$), we have that for each i, either t_i is π_j or $\Gamma \vdash t_i : \mathsf{Q}$, where Q is a negative propositional formula. By induction hypothesis, each t_i satisfies 1. and also t, while 2. is obviously verified.
  2. $r = \mathtt{H}_{a_i}^{\forall\alpha_i A_i}$. Then, t_1 is a closed term of $\mathcal{L}$. Let $A_i = \forall\gamma_1 \dots \forall\gamma_l\, \mathsf{Q}$, with Q propositional, we have that for each i, either t_i is a closed term of $\mathcal{L}$ or t_i is π_j or $\Gamma \vdash t_i : \mathsf{R}$, where R is a negative propositional formula. By induction hypothesis, each t_i which is a proof term satisfies 1. and thus also t, while 2. is obviously verified. ◀

If we omit the parentheses, we will show that every normal proof-term having as type an existential formula can be written as $v_0 \,|\, v_1 \,|\, \ldots \,|\, v_n$, where each v_i is not of the form $u \,|\, v$; if for every i, v_i is of the form (m_i, u_i), then we call the whole term an *Herbrand normal form*, because it is essentially a list of the witnesses appearing in an Herbrand disjunction. Formally:

▶ **Definition 15** (Herbrand Normal Forms). We define by induction a set of proof terms, called *Herbrand normal forms*, as follows:

- Every normal proof-term (m, u) is an Herbrand normal form;
- if u and v are Herbrand normal forms, $u \,|\, v$ is an Herbrand normal form.

Our last task is to prove that all quasi-closed proofs of any existential statement $\exists\alpha\, A$ include an exhaustive sequence $m_1, m_2, \ldots, m_k$ of possible witnesses. This theorem is stronger than the usual Herbrand theorem for classical logic [4], since we are stating it for any existential formula and not just for formulas with a single and existential quantifier.

▶ **Theorem 16** (Herbrand Disjunction Extraction). *Let $\exists\alpha\, A$ be a closed formula. Suppose $\Gamma \vdash t : \exists\alpha\, A$ in $\mathsf{IL} + \mathsf{EM}_1^-$ for a quasi closed term t, and $t \mapsto^* t'$ with t' in normal form. Then $\Gamma \vdash t' : \exists\alpha\, A$ and t' is an Herbrand normal form*

$$(m_0, u_0) \,|\, (m_1, u_1) \,|\, \ldots \,|\, (m_k, u_k)$$

Moreover, $\Gamma \vdash A[m_1/\alpha] \vee \cdots \vee A[m_k/\alpha]$.

Proof. By the Subject Reduction Theorem 11, $\Gamma \vdash t' : \exists\alpha\, A$. We proceed by induction on the structure of t'. According to Theorem 13, we can write t' as $r u_1 \ldots u_n$. Note that since t' is quasi closed, r cannot be a variable; moreover, r cannot be a term $\mathtt{H}^{\mathsf{P}}$ or $\mathtt{H}_b^{\forall\alpha B}$, otherwise t' would not have type $\exists\alpha\, A$, nor a term $\mathtt{W}_b^{\exists\alpha \mathsf{P}}$, otherwise t' would not be quasi closed. r also cannot be of the shape $u \parallel_a v$, otherwise $\Gamma \vdash u \parallel_a v : \exists\alpha\, \mathsf{Q}$, for some negative propositional Q, but from Proposition 14 we know that this is not possible. By Theorem 13, we are now left with only two possibilities.

1. r is obtained by an introduction rule. Then $n = 0$, otherwise there is a redex, and thus the only possibility is $t' = r = (n, u)$ which is an Herbrand Normal Form.
2. $r = u \,|\, v$. Again $n = 0$, otherwise we could apply a permutation rule; then $t' = r = u \,|\, v$, and the thesis follows by applying the induction hypothesis on u and v.

We have thus shown that t' is an Herbrand normal form

$$(m_0, u_0) \,|\, (m_1, u_1) \,|\, \ldots \,|\, (m_k, u_k)$$

Finally, we have that for each i, $\Gamma_i \vdash u_i : A[m_i/\alpha]$, for the very same Γ_i that types (m_i, u_i) of type $\exists\alpha\, A$ in t'. Therefore, for each i, $\Gamma_i \vdash u_i^+ : A[m_1/\alpha] \vee \cdots \vee A[m_k/\alpha]$, where u_i^+ is of the form $\iota_{i_1}(\ldots \iota_{i_k}(u_i) \ldots)$. We conclude that

$$\Gamma \vdash u_0^+ \,|\, u_1^+ \,|\, \ldots \,|\, u_k^+ : A[m_1/\alpha] \vee \cdots \vee A[m_k/\alpha]$$ ◀

4 Markov's Principle in Arithmetic

The original statement of Markov's principle refers to Arithmetic and can be formulated in the system of Heyting Arithmetic HA as

$$\neg\neg\exists\alpha\, \mathsf{P} \to \exists\alpha\, \mathsf{P}, \text{ for } \mathsf{P} \text{ atomic}$$

By adapting $\mathsf{IL} + \mathsf{EM}_1^-$ to Arithmetic, following [2], we will now provide a new computational interpretation of Markov's principle. Note first of all that propositional formulas are decidable in intuitionistic Arithmetic HA: therefore we will not need the rule EM_0^- and the pure parallel operator. For the very same reason, we can expect the system $\mathsf{HA} + \mathsf{EM}_1^-$ to be constructive and the proof to be similar to the one of Herbrand constructivity for $\mathsf{IL} + \mathsf{EM}_1^-$. In this section indeed we will give such a syntactic proof. We could also have used the realizability interpretation for $\mathsf{HA} + \mathsf{EM}_1$ introduced in [2] (see [10]).

4.1 The system $\mathsf{HA} + \mathsf{EM}_1^-$

We will now introduce the system $\mathsf{HA} + \mathsf{EM}_1^-$. We start by defining the language:

▶ **Definition 17** (Language of $\mathsf{HA} + \mathsf{EM}_1^-$). The language $\mathcal{L}$ of $\mathsf{HA} + \mathsf{EM}_1$ is defined as follows.

1. The terms of $\mathcal{L}$ are inductively defined as either variables $\alpha, \beta, \dots$ or 0 or $\mathsf{S}(t)$ with $t \in \mathcal{L}$. A numeral is a term of the form $\mathsf{S} \dots \mathsf{S}0$.
2. There is one symbol $\mathcal{P}$ for every primitive recursive relation over $\mathbb{N}$; with $\mathcal{P}^\perp$ we denote the symbol for the complement of the relation denoted by $\mathcal{P}$. The atomic formulas of $\mathcal{L}$ are all the expressions of the form $\mathcal{P}(t_1, \dots, t_n)$ such that $t_1, \dots, t_n$ are terms of $\mathcal{L}$ and n is the arity of $\mathcal{P}$. Atomic formulas will also be denoted as $\mathsf{P}, \mathsf{Q}, \mathsf{P}_\mathsf{i}, \dots$ and $\mathcal{P}(t_1, \dots, t_n)^\perp := \mathcal{P}^\perp(t_1, \dots, t_n)$.
3. The formulas of $\mathcal{L}$ are built from atomic formulas of $\mathcal{L}$ by the connectives $\vee, \wedge, \rightarrow, \forall, \exists$ as usual, with quantifiers ranging over numeric variables $\alpha^{\mathtt{N}}, \beta^{\mathtt{N}}, \dots$.

The system $\mathsf{HA} + \mathsf{EM}_1^-$ in Figure 5 extends the usual Curry-Howard correspondence for HA with our rule EM_1^- and is a restriction of the system introduced in [2]. The purely universal arithmetical axioms are introduced by means of Post rules, as in Prawitz [11].

As we anticipated, there is no need for a parallelism operator. Therefore EM_1^- introduces a pure delimited exception mechanism, explained by the reduction rules in Figure 6: whenever we have a term $u \parallel_a v$ and $\mathtt{H}_a^{\forall \alpha \mathsf{P}} m$ appears inside u, we can recursively *check* whether $\mathsf{P}[m/\alpha]$ holds, and switch to the exceptional path if it doesn't; if it does, we can remove the instance of the assumption. When there are no free assumptions relative to a left in u, we can forget about the exceptional path.

Similarly to the previous sections, we extend the characterization of the proof-term heads to take into account the new constructs.

▶ **Theorem 18** (Head of a Proof Term). *Every proof term of* $\mathsf{HA} + \mathsf{EM}_1^-$ *is of the form:*

$$\lambda z_1 \dots \lambda z_n . r u_1 \dots u_k$$

where

- *r is either a variable x, a constant $\mathtt{H}_a^{\forall \alpha P}$, $\mathtt{W}_a^{\exists \alpha P}$, r or R, an excluded middle term $u \parallel_a v$, or a term corresponding to an introduction rule $\lambda x.t$, $\lambda \alpha.t$, $\langle t_1, t_2 \rangle$, $\iota_i(t)$, (m, t)*
- *$u_1, \dots u_k$ are either lambda terms, first order terms, or one of the following expressions corresponding to elimination rules: π_i, $[x.w_1, y.w_2]$, $[(\alpha, x).t]$*

The new system proves exactly the same formulas that can be proven by making use of Markov's principle in Heyting Arithmetic.

▶ **Theorem 19.** *For any formula F in the language $\mathcal{L}$,* $\mathsf{HA} + \mathsf{MP} \vdash F$ *if and only if* $\mathsf{HA} + \mathsf{EM}_1^- \vdash F$*.*

Proof. The proof is identical as the one in the previous section. ◀

Grammar of Untyped Terms

$$t, u, v ::= \; x \mid tu \mid tm \mid \lambda x\, u \mid \lambda\alpha\, u \mid \langle t, u\rangle \mid u\pi_0 \mid u\pi_1 \mid \iota_0(u) \mid \iota_1(u) \mid t[x.u, y.v] \mid (m, t) \mid t[(\alpha, x).u]$$
$$\mid (u \parallel_a v) \mid \mathtt{H}_a^{\forall\alpha \mathsf{P}} \mid \mathtt{W}_a^{\exists\alpha \mathsf{P}} \mid \mathtt{True} \mid \mathsf{R}uvm \mid \mathsf{r}t_1 \dots t_n$$

where m ranges over terms of $\mathcal{L}$, x over variables of the lambda calculus and a over EM_1 hypothesis variables. Moreover, in terms of the form $u \parallel_a v$ there is a P such that all the free occurrences of a in u are of the form $\mathtt{H}_a^{\forall\alpha\mathsf{P}}$ and those in v are of the form $\mathtt{W}_a^{\exists\alpha\mathsf{P}^\perp}$.

Contexts With Γ we denote contexts of the form $e_1 : A_1, \dots, e_n : A_n$, where e_i is either a proof-term variable $x, y, z \dots$ or a EM_1^- hypothesis variable $a, b, \dots$

Axioms $\Gamma, x : A \vdash x : A \qquad \Gamma, a : \forall\alpha^{\mathbb{N}}\mathsf{P} \vdash \mathtt{H}_a^{\forall\alpha\mathsf{P}} : \forall\alpha^{\mathbb{N}}\mathsf{P} \qquad \Gamma, a : \exists\alpha^{\mathbb{N}}\mathsf{P}^\perp \vdash \mathtt{W}_a^{\exists\alpha P} : \exists\alpha^{\mathbb{N}}\mathsf{P}^\perp$

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

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

Disjunction Introduction $\dfrac{\Gamma \vdash u : A}{\Gamma \vdash \iota_0(u) : A \vee B} \qquad \dfrac{\Gamma \vdash u : B}{\Gamma \vdash \iota_1(u) : A \vee B}$

Disjunction Elimination $\dfrac{\Gamma \vdash u : A \vee B \quad \Gamma, x : A \vdash w_1 : C \quad \Gamma, y : B \vdash w_2 : C}{\Gamma \vdash u\,[x.w_1, y.w_2] : C}$

Universal Quantification $\dfrac{\Gamma \vdash u : \forall\alpha^{\mathbb{N}} A}{\Gamma \vdash um : A[m/\alpha]} \qquad \dfrac{\Gamma \vdash u : A}{\Gamma \vdash \lambda\alpha\, u : \forall\alpha^{\mathbb{N}} A}$

where m is any term of the language $\mathcal{L}$ and α does not occur free in any formula B occurring in Γ.

Existential Quantification $\dfrac{\Gamma \vdash u : A[m/\alpha]}{\Gamma \vdash (m, u) : \exists\alpha^{\mathbb{N}} A} \qquad \dfrac{\Gamma \vdash u : \exists\alpha^{\mathbb{N}} A \quad \Gamma, x : A \vdash t : C}{\Gamma \vdash u\,[(\alpha, x).t] : C}$

where α is not free in C nor in any formula B occurring in Γ.

Induction $\dfrac{\Gamma \vdash u : A(0) \quad \Gamma \vdash v : \forall\alpha^{\mathbb{N}}.A(\alpha) \to A(\mathsf{S}(\alpha))}{\Gamma \vdash \mathsf{R}uvm : A[m/\alpha]}$, where m is a term of $\mathcal{L}$

Post Rules $\dfrac{\Gamma \vdash u_1 : A_1 \; \Gamma \vdash u_2 : A_2 \; \cdots \; \Gamma \vdash u_n : A_n}{\Gamma \vdash u : A}$

where $A_1, A_2, \dots, A_n, A$ are atomic formulas of HA and the rule is a Post rule for equality, for a Peano axiom or for a classical propositional tautology or for booleans and if $n > 0$, $u = \mathsf{r}u_1 \dots u_n$, otherwise $u = \mathtt{True}$.

EM_1^- $\dfrac{\Gamma, a : \forall\alpha\, \mathsf{P} \vdash u : \exists\beta\; \mathsf{Q} \quad \Gamma, a : \exists\alpha\, \neg\mathsf{P} \vdash v : \exists\beta\; \mathsf{Q}}{\Gamma \vdash u \parallel_a v : \exists\beta\; \mathsf{Q}}$ (P atomic, Q negative propositional)

Figure 5 Term Assignment Rules for $\mathsf{HA} + \mathsf{EM}_1$.

$\mathsf{HA} + \mathsf{EM}_1^-$ with the reduction rules in Figure 4 enjoys the Subject Reduction Theorem [2, 10].

▶ **Theorem 20** (Subject Reduction). *If $\Gamma \vdash t : C$ and $t \mapsto u$, then $\Gamma \vdash u : C$.*

No term of $\mathsf{HA} + \mathsf{EM}_1^-$ gives rise to an infinite reduction sequence [1].

▶ **Theorem 21** (Strong Normalization). *Every term typable in $\mathsf{HA} + \mathsf{EM}_1^-$ is strongly normalizing.*

Reduction Rules for HA

$$(\lambda x.u)t \mapsto u[t/x] \qquad (\lambda\alpha.u)m \mapsto u[m/\alpha]$$

$$\langle u_0, u_1\rangle\pi_i \mapsto u_i, \text{ for i=0,1}$$

$$\iota_i(u)[x_1.t_1, x_2.t_2] \mapsto t_i[u/x_i], \text{ for i=0,1}$$

$$(m, u)[(\alpha, x).v] \mapsto v[m/\alpha][u/x], \text{ for each term } m \text{ of } \mathcal{L}$$

$$\mathsf{R}uv0 \mapsto u$$

$$\mathsf{R}uv(\mathsf{S}n) \mapsto vn(\mathsf{R}uvn), \text{ for each numeral } n$$

Reduction Rules for EM_1^-

$$u \parallel_a v \mapsto u, \text{ if } a \text{ does not occur free in } u$$

$$u \parallel_a v \mapsto v[a := n], \text{ if } \mathtt{H}_a^{\forall\alpha\mathsf{P}}n \text{ occurs in } u \text{ and } \mathsf{P}[n/\alpha] \text{ is } \textit{closed} \text{ and } \mathsf{P}[n/\alpha] \text{ is false}$$

$$(\mathtt{H}_a^{\forall\alpha\mathsf{P}})n \mapsto \mathtt{True} \text{ if } \mathsf{P}[n/\alpha] \text{ is } \textit{closed} \text{ and } \mathsf{P}[n/\alpha] \text{ is true}$$

Figure 6 Reduction Rules for $\mathsf{HA} + \mathsf{EM}_1$.

4.2 $\mathsf{HA} + \mathsf{EM}_1^-$ is Constructive

We can now proceed to prove the constructivity of the system, that is the disjunction and existential properties. We will do this again by inspecting the normal forms of the proof terms; the first thing to do is adapting Proposition 14 to $\mathsf{HA} + \mathsf{EM}_1^-$.

▶ **Proposition 22** (Normal Form Property). *Let $\mathsf{P}, \mathsf{P}_1, \dots \mathsf{P}_n$ be negative propositional formulas, $A_1, \dots A_m$ simply universal formulas. Suppose that*

$$\Gamma = z_1 : \mathsf{P}_1, \dots z_n : \mathsf{P}_n, a_1 : \forall\alpha_1 A_1, \dots a_m : \forall\alpha_m A_m$$

and $\Gamma \vdash t : \exists\alpha\,\mathsf{P}$ or $\Gamma \vdash t : \mathsf{P}$, with t in normal form and having all its free variables among $z_1, \dots z_n, a_1, \dots a_m$. Then:

1. *Every occurrence in t of every term $\mathtt{H}_{a_i}^{\forall\alpha_i A_i}$ is of the active form $\mathtt{H}_{a_i}^{\forall\alpha_i A_i}m$, where m is a term of $\mathcal{L}$*
2. *t cannot be of the form $u \parallel_a v$.*

Proof. The proof is identical to the proof of Proposition 14. We just need to consider the following additional cases:

- $t = \mathsf{r}t_1t_2\dots t_n$. Then $\Gamma \vdash t_i : \mathsf{Q}_i$ for some atomic Q_i and for $i = 1 \dots n$; thus 1. holds by applying the inductive hypothesis to the t_i, while 2. is obviously verified.
- $t = \mathsf{R}t_1 \dots t_n$. This case is not possible, otherwise, since t_3 is a numeral, t would not be in normal form. ◀

Thanks to this, we can now state the main theorem. The proof of the existential property is the same as the one for Theorem 16: we just need to observe that since we don't have a parallelism operator in $\mathsf{HA} + \mathsf{EM}_1^-$, every Herbrand disjunction will consist of a single term. The disjunction property will follow similarly.

▶ **Theorem 23** (Constructivity of $\mathsf{HA} + \mathsf{EM}_1^-$).

- *If* $\mathsf{HA} + \mathsf{EM}_1^- \vdash t : \exists\alpha A$*, then there exists a term* $t' = (n, u)$ *such that* $t \mapsto^* t'$ *and* $\mathsf{HA} + \mathsf{EM}_1^- \vdash u : A[n/\alpha]$
- *If* $\mathsf{HA}+\mathsf{EM}_1^- \vdash t : A \vee B$*, then there exists a term* t' *such that* $t \mapsto^* t'$ *and either* $t' = \iota_0(u)$ *and* $\mathsf{HA} + \mathsf{EM}_1^- \vdash u : A$*, or* $t' = \iota_1(u)$ *and* $\mathsf{HA} + \mathsf{EM}_1^- \vdash u : B$

Proof. For both cases, we start by considering a term t' such that $t \mapsto^* t'$ and t' is in normal form. By the Subject Reduction Theorem 11 we have that $\mathsf{HA} + \mathsf{EM}_1^- \vdash t' : \exists\alpha A$ (resp. $\mathsf{HA} + \mathsf{EM}_1^- \vdash t' : A \vee B$). By Theorem 13 we can write t' as $rt_1 \dots t_n$. Since t' is closed, r cannot be a variable x or a term $\mathtt{H}_a^{\forall\alpha\mathsf{P}}$ or $\mathtt{W}_a^{\exists\alpha\mathsf{P}}$; moreover it cannot be r, otherwise the type of t' would have to be atomic, and it cannot be R, otherwise the term would not be in normal form. r also cannot have been obtained by EM_1^-, otherwise $\mathsf{HA} + \mathsf{EM}_1^- \vdash r : \exists\alpha\mathsf{P}$, for P atomic and $r = t_1 \parallel_a t_2$; but this is not possible due to Proposition 22. Therefore, r must be obtained by an introduction rule. We distinguish now the two cases:

- $\mathsf{HA} + \mathsf{EM}_1^- \vdash t' : \exists\alpha B$. Since the term is in normal form, n has to be 0, that is $t' = r$ and $r = (n, u)$; hence also $\mathsf{HA} + \mathsf{EM}_1^- \vdash u : A(n)$.
- $\mathsf{HA} + \mathsf{EM}_1^- \vdash t' : A \vee B$. Then either $t' = \iota_0(u)$, and so $\mathsf{HA} + \mathsf{EM}_1^- \vdash u : A$, or $t' = \iota_1(u)$, and so $\mathsf{HA} + \mathsf{EM}_1^- \vdash u : B$. ◀

References

1 Federico Aschieri. Strong normalization for HA + EM1 by non-deterministic choice. In *Proceedings First Workshop on Control Operators and their Semantics, COS 2013, Eindhoven, The Netherlands, June 24-25, 2013.*, pages 1–14, 2013. `doi:10.4204/EPTCS.127.1`.

2 Federico Aschieri, Stefano Berardi, and Giovanni Birolo. Realizability and strong normalization for a Curry-Howard interpretation of ha+em1. In *Computer science logic 2013*, volume 23 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages 45–60. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2013.

3 Federico Aschieri and Margherita Zorzi. A "game semantical" intuitionistic realizability validating Markov's principle. In *19th International Conference on Types for Proofs and Programs*, volume 26 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages 24–44. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2014.

4 Federico Aschieri and Margherita Zorzi. On natural deduction in classical first-order logic: Curry-Howard correspondence, strong normalization and Herbrand's theorem. *Theoret. Comput. Sci.*, 625:125–146, 2016. `doi:10.1016/j.tcs.2016.02.028`.

5 Samuel R. Buss. On Herbrand's theorem. In *Logic and computational complexity (Indianapolis, IN, 1994)*, volume 960 of *Lecture Notes in Comput. Sci.*, pages 195–209. Springer, Berlin, 1995. `doi:10.1007/3-540-60178-3_85`.

6 Ph. de Groote, editor. *The Curry-Howard isomorphism*, volume 8 of *Cahiers du Centre de Logique [Reports of the Center of Logic]*. Academia-Erasme, Louvain-la-Neuve; Université Catholique de Louvain, Département de Philosophie, Louvain-la-Neuve, 1995.

7 Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. `doi:10.1111/j.1746-8361.1958.tb01464.x`.

8 Hugo Herbelin. An intuitionistic logic that proves Markov's principle. In *25th Annual IEEE Symposium on Logic in Computer Science LICS 2010*, pages 50–56. IEEE Computer Soc., Los Alamitos, CA, 2010.

9 S. C. Kleene. On the interpretation of intuitionistic number theory. *J. Symbolic Logic*, 10:109–124, 1945. `doi:10.2307/2269016`.

10 Matteo Manighetti. Computational interpretations of markov's principle. Master's thesis, Wien, Techn. Univ., Wien, 2016. URL: `https://arxiv.org/abs/1611.03714`.

11 Dag Prawitz. Ideas and results in proof theory. In *Proceedings of the Second Scandinavian Logic Symposium (Univ. Oslo, Oslo, 1970)*, pages 235–307. Studies in Logic and the Foundations of Mathematics, Vol. 63. North-Holland, Amsterdam, 1971.

12 A. S. Troelstra. *Corrections and additions to: Metamathematical investigation of intuitionistic arithmetic and analysis (Lecture Notes in Math., Vol. 344, Springer, Berlin, 1973)*. Mathematisch Intituut, Universiteit van Amsterdam, Amsterdam, 1974. Report 74-16.

13 A. S. Troelstra and D. van Dalen. *Constructivism in mathematics. Vol. I*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1988. An introduction.

Design and Implementation of the Andromeda Proof Assistant

Andrej Bauer[1]
University of Ljubljana, Slovenia

Gaëtan Gilbert
École Normale Supérieure de Lyon, France

Philipp G. Haselwarter[2]
University of Ljubljana, Slovenia

Matija Pretnar
University of Ljubljana, Slovenia

Christopher A. Stone
Harvey Mudd College, Claremont, CA 91711, USA

Abstract

Andromeda is an LCF-style proof assistant where the user builds derivable judgments by writing code in a meta-level programming language AML. The only trusted component of Andromeda is a minimalist nucleus (an implementation of the inference rules of an object-level type theory), which controls construction and decomposition of type-theoretic judgments.

Since the nucleus does not perform complex tasks like equality checking beyond syntactic equality, this responsibility is delegated to the user, who implements one or more equality checking procedures in the meta-language. The AML interpreter requests witnesses of equality from user code using the mechanism of algebraic operations and handlers. Dynamic checks in the nucleus guarantee that no invalid object-level derivations can be constructed.

To demonstrate the flexibility of this system structure, we implemented a nucleus consisting of dependent type theory with equality reflection. Equality reflection provides a very high level of expressiveness, as it allows the user to add new judgmental equalities, but it also destroys desirable meta-theoretic properties of type theory (such as decidability and strong normalization).

The power of effects and handlers in AML is demonstrated by a standard library that provides default algorithms for equality checking, computation of normal forms, and implicit argument filling. Users can extend these new algorithms by providing local "hints" or by completely replacing these algorithms for particular developments. We demonstrate the resulting system by showing how to axiomatize and compute with natural numbers, by axiomatizing the untyped λ-calculus, and by implementing a simple automated system for managing a universe of types.

2012 ACM Subject Classification Theory of computation $\to$ Type theory

Keywords and phrases type theory, proof assistant, equality reflection, computational effects

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.5

[1] This material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.
[2] This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

1 Introduction

A type theory can be interesting and very useful, yet lack metatheoretic properties (e.g., decidability) that permit a straightforward implementation. In fact, the more flexible and expressive the theory, the less likely these properties will hold. Nevertheless, even very expressive type theories deserve automated support in the form of proof assistants. The question is how a useful proof assistant can make minimal demands on the properties of the underlying object language. In this paper, we describe the structure of one such system.

Andromeda is an LCF-style proof assistant [12] in which (derivable) judgments are the fundamental data of the system. These judgments are opaque except within a tiny, trusted nucleus that implements rules of the underlying type theory (to construct new judgments from old) and also implements valid inversion principles (to decompose judgments into sub-judgments). The untrusted remainder of the hard-coded system is a small interpreter for AML, an ML-like meta-language [18] extended with algebraic operations and handlers [21].

The AML interpreter builds and decomposes judgments by making (dynamically checked) requests of the trusted nucleus. When these requests would fail (e.g., because a function is being applied to an argument, and in violation of the appropriate typing rule the domain type of the function is not syntactically identical to the type of the argument), the interpreter triggers a suitable algebraic operation to request additional information (e.g., evidence of equality between the mismatched types) from the user.

User-level AML code directs the construction of judgments, and consists of computations that construct and pattern-match judgment values and user-level handlers that intercept and respond to algebraic operations triggered during judgment construction. The consequence of this design is that most proof-assistant functionality – including equality checking, normalization, unification, and proof tactics – is handled at the user level. Effects and handlers allow default implementations (necessarily incomplete for an undecidable object language) that can be overridden using nested handlers, providing specialized algorithms for specific trouble spots.

The specific expressive object language is largely independent of this system design, but some readers may find our chosen type theory independently interesting. The type theory currently implemented in Andromeda is dependent type theory with *equality reflection*, the principle that propositionally equal terms are judgmentally equal:

$$\frac{\Gamma \vdash u : \mathsf{Eq}_A(s,t)}{\Gamma \vdash s \;\equiv\; t : A}$$

From a mathematical point of view, equality reflection is appealing and natural, as it makes equality in type theory behave like ordinary equality in mathematics. (In Coq, for example, the types “vector of length $0 + n$” and “vector of length n” are equal because $0 + n$ and n are judgmentally equal, but a “vector of length $n + 0$” requires an explicit coercion to be used as a “vector of length n” because $n+0$ is only propositionally equal to n.) From the perspective of homotopy type theory, equality reflection is suitable for “set-level” mathematics, i.e., those mathematical structures that do not exhibit any higher homotopical phenomena. Among these are substantial parts of algebra, analysis, and logic, including many aspects of meta-theory of type theory.

Building equality reflection into a proof assistant has practical advantages. First, equality reflection lets users axiomatize type-theoretic constructions such as natural numbers with *judgmental* equalities, meaning that we can implement a smaller trusted core type theory with a wider variety of possible user extensions. Second, applications of equality reflection are not recorded in the conclusion, and omitting the explicit equality eliminators keeps terms smaller and simpler.

The proof assistant NuPRL [2] validated equality reflection by implementing so-called *computational* type theory, a specific interpretation of type theory akin to realizability models. More recently, however, equality reflection has fallen into disrepute among computer scientists and computationally minded mathematicians. It causes the loss of useful meta-theoretic properties such as strong normalization of terms and decidability of type checking [14], the cornerstones of modern proof assistants like Coq [7], Agda [19] and Lean [10]. Even the property "if an application of a lambda abstraction to an argument is well typed, then its β-reduct is well typed" may not hold if the user assumes nonstandard type equalities.

Nevertheless, the use of effects and handlers allows Andromeda to take advantage of equality reflection and to deal with its negative consequences gracefully.

Contributions. The present paper should be read as a progress report on the development of Andromeda; the system and the underlying type theory may evolve as we gain more experience and consider a wider variety of applications. We focus on the following points of interest:

- the goals of Andromeda and the structure of the system (§2);
- the impact of equality reflection on both the design of the type-theoretic nucleus and the details of its implementation (§3, Appendix A);
- features of the meta-language that allow a variety of proof-development techniques to be implemented at the user level (§4);
- a discussion of the soundness of the system (§5);
- a prototype standard library that provides user-extensible equality checking and implicit-argument filling (§6);
- axiomatization of additional type-theoretic structures (dependent sums, natural numbers, untyped λ-calculus, and universes), *with* the desired judgmental equalities and support for automation (§7).

Andromeda is free software, available at `http://www.andromeda-prover.org/`. Contributions, questions, and requests are most welcome.

2 An overview of Andromeda

Andromeda follows design principles that are similar to those of other proof assistants:

- The system should *work well in the common case*. Equality reflection affords many possibilities for complicating one's life, but we expect most applications to be very reasonable. If the user introduces new computation and extensionality rules that play nicely with the existing ones, the system should work smoothly. Nevertheless, less common scenarios should still work, possibly with more effort on the part of the user.
- The user cannot be expected to write down explicit typing annotations on all terms, or hold in their head various bureaucratic matters, such as the typing contexts. Therefore, the *system should take care of low-level details.*
- There should be a *clearly delineated nucleus* that is the only part of the implementation that the user has to trust[3] in order to believe that Andromeda never produces an invalid

[3] Except for trusting the OCaml compiler, the operating system, the hardware, and the absence of malicious cosmic rays.

judgment. The nucleus should be as small as possible and its functionality should implement type theory in the most straightforward way possible.

- A consequence of this minimalism is that the system should be *user extensible*, so that additional functionality can be introduced without breaking trust.

Andromeda is implemented in the tradition of Robin Milner's Logic for Computable Functions (LCF) [12]. The current implementation, in OCaml [20], consists of around 9500 lines of source code, of which the nucleus comprises 1900 lines. These are very low numbers that clearly classify Andromeda as a prototype. However, we do not expect the nucleus to grow significantly.

The core of Andromeda is the trusted nucleus that directly implements inference rules and inversion rules for dependent type theory with equality reflection. By design, it is the only part of the system that can create and manipulate type-theoretic judgments. The nucleus is small and simple, as it does not perform *any* proof search, unification, equality checking, or normalization. (It cannot, since equality checking is in general undecidable and there is no reasonable notion of normal form [14].) Whenever evidence of equality is needed as a premise to an inference rule, it must be provided to the nucleus explicitly.

The user interacts with the system by writing code in the *Andromeda meta-language (AML)*, a general-purpose programming language in the style of ML. AML exposes the nucleus datatype `judgment` as an abstract datatype of its own. Because judgments may only be constructed by the nucleus, neither the OCaml implementation of AML nor any user code written in AML need be trusted. AML handles only trivial syntactic equality checks. All other evidence of equality is obtained from user-level code, through the mechanism of algebraic operations and handlers [22, 3] (§4.3).

Users are free to organize their AML code in any way they see fit. In most cases they would likely want a good axiomatization of standard type constructors (dependent sums, inductive types, universes, etc.), equality checking algorithms that work well in the common cases, and conveniences such as resolution of implicit arguments. These ought to be provided by a standard library (§6). In principle, there may be several such libraries, or even several equality checking algorithms in a single library. The handlers mechanism allows flexible and local uses of several different equality checking algorithms.

AML is statically typed – and this caught many silly errors while we were coding a standard library – but the AML type system is unrelated to the soundness of the system. Bugs in AML code either prevent code from constructing the desired judgments or construct an unintended judgment, but the abstract type of judgments and run-time checks in the nucleus ensure that only derivable judgments are ever constructed. Any other memory-safe metalanguage (e.g., one modeled on Python or Scheme) would be equally sound, if less robust.

Andromeda in action

Before looking at the three constituent parts of Andromeda in more detail, we provide a small worked example. At this point we cannot explain all the technical details, so we focus on emphasizing the important points and showcasing what Andromeda can do.

We begin by declaring some constants that Andromeda adds to the ambient signature:

```
constant A : Type
constant a : A
constant b : A
constant P : A → Type
constant v : P a
```

Andromeda manipulates *only* judgments. Thus the above declaration binds the AML variable `a` to the nucleus judgment ⊢ `a : A`, not to a bare symbol (and similarly for `b`, `v`, `A` and `P`). Nevertheless, it is often convenient to think of a judgment as “a term with a given type, possibly depending on some hypotheses”.

Let us first show that the type of transport is inhabited:

```
Π (x y : A), x ≡ y → P x → P y
```

In intensional type theory we would use a J eliminator, but here we should be able to use the curried term `λ x y ξ u, u`. Indeed, `u` may be converted from `P x` to `P y` because these are equal types by an application of the congruence rule for applications and equality reflection of `ξ : x ≡ y`. The Andromeda standard library, which is implemented at the user-level, does all this for us if we tell it to use ξ as an *equality hint* while checking that `u` has type `P y`:

```
λ (x y : A) (ξ : x ≡ y) (u : P x), (now hints = add_hint ξ in (u : P y))
```

The above is *not* a proof term but an AML computation that generates a judgment. In particular, AML immediately evaluates the command inside the λ. While doing so it will find it needs a witness for the equality between `P x` (the type of `u`) and `P y` (the type ascribed to `u`); it requests one using the operations and handler mechanism. The standard library handles this, employing an equality checking algorithm that eventually uses the hint ξ to equate `x` and `y`, and passing back the requested evidence to AML. Then, AML asks the nuclues to apply equality reflection to obtain the judgmental equality of `P x` and `P y` (at which point the equality witness provided by the standard library is discarded) and to apply conversion. The interaction betwen AML and the nuclues proceeds in this fashion until the judgment witnessing transport is constructed.

If we need to write `add_hint` to guide the type checker, one might ask why this is better than the intentional approach of applying a `J` eliminator with ξ to coerce `u` from `P x` to `P y`. A single `add_hint` is like the incorporation of a computation rule that can handle an arbitrarily complex development, whereas `J` is like a single application of a computation rule that has to be repeated at every point where the rule is to be applied.

The other benefit, apparent even here, is that without `J` the proof term is smaller. In the end, the judgment built by the nucleus is the expected one:

```
⊢ λ (x : A) (y : A) (_ : x ≡ y) (u : P x), u
  : Π (x : A) (y : A), x ≡ y → P x → P y
```

Although ξ does not appear explicitly in the conclusion, Andromeda is aware it was used. This tracking process becomes apparent if we *temporarily* hypothesize an equality `a ≡ b` and use it as a hint while constructing a judgment that `v` above has type `P b`,

```
assume ζ : a ≡ b in
  now hints = add_hint ζ in (v : P b)
```

This AML expression causes the nucleus to build the *hypothetical* judgment:

```
ζ₀ : a ≡ b ⊢ v : P b
```

The `assume` construct generated a fresh variable ζ_0 of type `a ≡ b` and bound the AML variable ζ to the judgment ζ_0 `: a ≡ b` ⊢ ζ_0 `: a ≡ b`. Because ζ_0 was used to convert `P a` to `P v`, the nucleus produced a judgment that depends on it.

The AML interpreter communicates with user-level code by invoking operations to be handled, but user-level operations are useful as well. Let us define a simple `auto` tactic for automatically inhabiting simple types. We first need an AML function, called `derive`, which attempts to inhabit a given type from the currently available hypotheses by performing a recursive search. This takes about 40 lines of uneventful code, shown in Appendix B. Then we declare a new operation that takes no arguments and yields judgments,

```
operation auto : judgment
```

and install a global handler that handles it:

```
handle
| auto : ?Surr ⇒
    match Surr with
    | Some ?T ⇒ derive T
    | None    ⇒ failure
    end
end
```

When the handler intercepts the operation `auto`, the surroundings of the occurrence of `auto` may or may not have indicated an expected result type `T`. If it does, the handler calls `derive T` to inhabit the type, otherwise it fails by triggering the operation `failure` (also defined in the appendix) because it has no information on what type to inhabit.

Now we can use `auto` to inhabit types. For example,

```
λ (X : Type), (auto : X → X)
```

constructs the judgment

```
⊢ λ (X : Type) (x : X), x
  : Π (X : Type), X → X
```

Given types `A`, `B`, and `C`, the computation

```
auto : (A → B → C) → (A → B) → (A → C)
```

results in the judgment

```
⊢ λ (x : A → B → C) (x0 : A → B) (x1 : A), x x1 (x0 x1)
  : (A → B → C) → (A → B) → A → C
```

The Andromeda standard library (§6) takes full advantage of operations and handlers to produce equality proofs and coercions, with default implementations that users can override with local handlers when the standard heuristics fail.

3 The nucleus

The nucleus is the part of the system that implements the object-level type theory. Its functionality includes the following:

- formation and decomposition of term and type judgments,
- construction of equality judgments,
- substitution and syntactic equality checking,
- pretty-printing of judgments and export to JSON.

Before discussing some of these features we take a closer look at the type theory implemented by Andromeda, and engineering issues that it raises.

3.1 Type theory with equality reflection

The Andromeda nucleus implements an extensional Martin-Löf type theory [17, 14] with dependent products $\prod_{(x:A)} B$ and equality types $\mathsf{Eq}_A(s,t)$. Complete rules are provided in Appendix A. Fundamentally, the system is not too far removed from the more common intensional Martin-Löf type theory, but instead of a J eliminator for equality types, we have equality reflection and uniqueness of equality terms:

$$\text{EQ-REFLECTION}\quad \frac{\Gamma \vdash u : \mathsf{Eq}_A(s,t)}{\Gamma \vdash s \equiv t : A} \qquad \text{EQ-ETA}\quad \frac{\Gamma \vdash t : \mathsf{Eq}_A(s,u) \qquad \Gamma \vdash v : \mathsf{Eq}_A(s,u)}{\Gamma \vdash t \equiv v : \mathsf{Eq}_A(s,u)}$$

The J eliminator can easily be derived from these rules, but direct use of equality reflection is generally simpler. Streicher's K eliminator and uniqueness of identity proofs [25] are also derivable in this setting.

Equality reflection invalidates some common structural rules and inversion principles, so we make further small changes to the type theory to compensate. First, it is usual for products to satisfy an injectivity property, i.e., if $\prod_{(x:A_1)} A_2$ and $\prod_{(x:B_1)} B_2$ are equal then A_1 equals B_1 and A_2 equals B_2. But in our type theory injectivity fails because under the assumption

$$p : \mathsf{Eq}_{\mathsf{Type}}((\mathsf{Nat} \to \mathsf{Nat}), (\mathsf{Nat} \to \mathsf{Bool})) \tag{1}$$

$\mathsf{Nat} \to \mathsf{Nat}$ and $\mathsf{Nat} \to \mathsf{Bool}$ are equal by reflection, *without* equality of Nat and Bool.[4] This may seem a very technical point, but usually one relies on injectivity to prove that β-reductions preserve types. Indeed, under assumption (1) the identity function on Nat also has type $\mathsf{Nat} \to \mathsf{Bool}$, and hence by applying it to 0 and β-reducing, we can show that 0 has type Bool, even though Nat and Bool are not equal.

Andromeda's solution, following [14], is to add explicit typing annotations that can typically be omitted in intentional type theories. A λ-abstraction $\lambda x{:}A.B\,.\,t$ is annotated not only with the domain A of the bound variable but also with the type B of the body t, and an application $s \mathbin{@^{x:A.B}} t$ is similarly annotated with the type of the function being applied. These annotations ensure that terms have unique types up to equality: working again under the assumption (1), we can apply the identity function at type $\mathsf{Nat} \to \mathsf{Nat}$ to get $(\lambda x{:}\mathsf{Nat}.\mathsf{Nat}\,.\,x) \mathbin{@^{x:\mathsf{Nat}.\mathsf{Nat}}} 0$ of type Nat, or at $\mathsf{Nat} \to \mathsf{Bool}$ to get $(\lambda x{:}\mathsf{Nat}.\mathsf{Nat}\,.\,x) \mathbin{@^{x:\mathsf{Nat}.\mathsf{Bool}}} 0$ of type Bool. Crucially, the typing annotations now prevent the latter term from β-reducing to 0, as the β-rule requires that the function and the application match:

$$\text{PROD-BETA}\quad \frac{\Gamma, x : A \vdash s : B \qquad \Gamma \vdash t : A}{\Gamma \vdash (\lambda x{:}A.B\,.\,s) \mathbin{@^{x:A.B}} t \equiv s[t/x] : B[t/x]}$$

Another principle that fails in the presence of equality reflection is *strengthening*, which says that we may safely remove from the context any hypothesis that is not explicitly mentioned in the conclusion of a judgment. Indeed,

$$p : \mathsf{Eq}_{\mathsf{Type}}(\mathsf{Nat} \to \mathsf{Nat}, \mathsf{Nat} \to \mathsf{Bool}) \vdash (\lambda x{:}\mathsf{Nat}.\mathsf{Nat}\,.\,x) \mathbin{@^{x:\mathsf{Nat}.\mathsf{Bool}}} 0 : \mathsf{Bool}$$

[4] The assumption that the Cantor space and the Baire space are equal may seem odd, but it is consistent. For instance, in classical set theory and in the effective topos the two are isomorphic, and with a little work we can arrange them to be equal.

becomes invalid if we remove p, even though there is no explicit use of p in the conclusion. For similar reasons *exchange* is not valid: given types X and Y, the context

$$x : X, p : \mathsf{Eq}_{\mathsf{Type}}(X, Y), q : \mathsf{Eq}_{Y}(x, x)$$

becomes invalid if we exchange the order of p and q, even though their types do not refer to each other. The loss of strengthening and exchange is inconvenient; we discuss an implementation-level solution in §3.2.

Perhaps the biggest difference between Andromeda and standard type theory is that we currently postulate a single universe Type and the rule that makes Type an element of itself:

$$\textsc{ty-type}\quad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}$$

From a logical point of view this is an inconsistent assumption, as Girard's paradox implies that every type is inhabited [11]. From an engineering point of view, however, Type : Type is very useful. For Andromeda implementers, it allows a much simpler implementation strategy with fewer different judgment forms. For Andromeda users, it allows postponing the complexities of dealing with type universes and universe levels, and instead focus on other aspects of derivations. (For the same reason, both Coq and Agda allow the assumption Type : Type as an option.) Nevertheless, although users are unlikely to stumble into inconsistencies by accident, we ultimately want a sound foundation, and plan to remove TY-TYPE, as discussed in §9.

3.2 Implementation of type theory

The type theory implemented in the nucleus differs from the one presented in several ways. The changes are inessential from a theoretical point of view, but have significant practical impact. We describe them in this section.

Signatures

In Andromeda the user extends the type theory by postulating constants, i.e., they work in type theory over a *signature*. In this respect Andromeda is much like other proof assistants that allow the user to state axioms and postulates. The signature is controlled by the nucleus through an abstract datatype whose interface is very simple: there is an empty signature, and a signature may be extended with a new constant of a given *closed* type (which may refer to the previously declared constants). Because signatures are ever increasing, judgments derived over a signature remain valid when the signature changes.

Inversion principles and natural types

The nucleus implements inversion principles for deconstruction of judgments into sub-judgments; these are exposed in AML through pattern matching, cf. §4.4. For example, an application $\Gamma \vdash s \mathbin{@^{x:A.B}} t : C$ can be decomposed into $\Gamma \vdash s : \prod_{(x:A)} B$, $\Gamma \vdash t : A$ and $\Gamma \vdash C\ \mathsf{type}$. The type-theoretic justification for this operation is an *inversion principle*: if $\Gamma \vdash s \mathbin{@^{x:A.B}} t : C$ is derivable then so are $\Gamma \vdash s : \prod_{(x:A)} B$, $\Gamma \vdash t : A$ and $\Gamma \vdash C\ \mathsf{type}$. Proving the principle is not hard but neither is it a complete triviality, because the application could have been formed with the use of type conversions. Similar inversion principles hold for other term and type formers.

If we decompose an application, as above, and put it back together using the application formation rule, we get the judgment $\Gamma \vdash s \mathbin{@^{x:A.B}} t : B[t/x]$. The application has received its "natural type", which may not be the original type C. Nevertheless, the types are equal by *uniqueness of typing*:

If $\Gamma \vdash t : A_1$ *and* $\Gamma \vdash t : A_2$ *then* $\Gamma \vdash A_1 \equiv A_2$.

The nucleus provides evidence of uniqueness of typing by generating, given $\Gamma \vdash t : A$, a witness for equality of A and the *natural type* $\mathcal{N}(t)$ of t, which is read off the typing annotations:

$$\begin{aligned} \mathcal{N}(\mathsf{Type}) &= \mathsf{Type} & \mathcal{N}(\textstyle\prod_{(x:A)} B) &= \mathsf{Type} \\ \mathcal{N}(\mathsf{Eq}_A(s,t)) &= \mathsf{Type} & \mathcal{N}(\lambda x{:}A.B\,.\,t) &= \textstyle\prod_{(x:A)} B \\ \mathcal{N}(s \mathbin{@^{x:A.B}} t) &= B[t/x] & \mathcal{N}(\mathsf{refl}_A\ t) &= \mathsf{Eq}_A(t,t) \end{aligned}$$

The natural types of variables and constants are read off the context and the signature, respectively. Note that the natural type is the one we get if we deconstruct a term judgment and construct it back again. In the standard library the equality of the original type and the natural type is needed in several places during equality checking.

Assumption sets

The nucleus is responsible for decomposing judgments into their component parts, a facility used by pattern matching in AML. For example, we can combine

$$f : \mathsf{Nat} \to \mathsf{Nat} \vdash f : \mathsf{Nat} \to \mathsf{Nat} \qquad \text{and} \qquad x : \mathsf{Nat} \vdash x : \mathsf{Nat}$$

(using weakening and application) to get

$$f : \mathsf{Nat} \to \mathsf{Nat},\, x : \mathsf{Nat} \vdash f \mathbin{@^{\_:\mathsf{Nat}.\mathsf{Nat}}} x : \mathsf{Nat},$$

But if we naively pattern-match on this application to get the function part and the argument part (as judgments), we would get the constituents in weakened form

$$f : \mathsf{Nat} \to \mathsf{Nat},\ x : \mathsf{Nat} \vdash f : \mathsf{Nat} \to \mathsf{Nat} \qquad \text{and} \qquad f : \mathsf{Nat} \to \mathsf{Nat},\ x : \mathsf{Nat} \vdash x : \mathsf{Nat}.$$

In a system with strengthening, we could immediately see that x is unnecessary in the first judgment and f in the second. The loss of strengthening is inconvenient enough that we restore it by explicitly keeping track of dependencies on the assumptions in the context.

In the implementation we use *terms with assumptions*, which are ordinary terms that have every subterm annotated with a set of variables, called the *assumptions*, indicating explicitly which part of the context a subterm depends on. Thus $\Gamma \vdash t^\alpha : A^\beta$ means that we may restrict Γ to variables in α to obtain a smaller context $\Gamma\restriction_\alpha$ in which it is still possible to show that t has type A. Similarly, $\Gamma\restriction_\beta$ suffices to derive the judgment that A is a type. The types and terms appearing in the context are themselves annotated with assumptions, which endows contexts with the structure of directed acyclic graphs. (In the implementation they are stored as such.)

This means that Andromeda can compose and decompose judgments without information loss. The application above will be recorded internally as:

$$f : (\mathsf{Nat}^\emptyset \to \mathsf{Nat}^\emptyset)^\emptyset,\, x : \mathsf{Nat}^\emptyset \vdash (f^{\{f\}} \mathbin{@^{\_:\mathsf{Nat}^\emptyset.\mathsf{Nat}^\emptyset}} x^{\{x\}})^{\{f,x\}} : \mathsf{Nat}^\emptyset.$$

and it is straightforward to recover the two original sub-judgments.

Constants from the signature are not included in assumption sets, since they are omnipresent anyhow.

Context joins

The standard rules of inference require the contexts of the premises to match, for instance the application rule TERM-APP does not allow a change of the context:

$$\frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Gamma \vdash t : A}{\Gamma \vdash s \mathbin{@^{x:A.B}} t : B[t/x]}$$

If we implemented the rule exactly as is, the user would have to plan dependence on hypotheses carefully in advance, which is impractical. Instead, we rely on admissibility of weakening to enlarge contexts as necessary. In every inference rule we accept premises with arbitrary contexts that are then *joined* to form a single extended context, for instance the application rule becomes

$$\frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Delta \vdash t : A}{\Gamma \bowtie \Delta \vdash s \mathbin{@^{x:A.B}} t : B[t/x]}$$

The context join $\Gamma \bowtie \Delta$ is the smallest context that extends both Γ and Δ. In terms of directed acyclic graphs it is just the union of graphs. A context join fails if there is a hypothesis that has different types in Γ and Δ, or if the join would create a cyclic dependency of hypotheses. In practice such failures are infrequent; λ, Π, and `assume` create globally fresh object-level variables, so there is no direct way to create two contexts with the same variable at different types.[5]

Andromeda automatically tracks assumption sets and contexts. Even though the implementation makes an effort to keep them small, they may not be unique or minimal: they merely reflect a history of how judgments were constructed.

4 The Andromeda meta-language

The Andromeda meta-language (AML) is a programming language in the style of ML [18]. We review its structure and capabilities, focusing on the parts that are peculiar to Andromeda. For constructs that are standard in the ML-family of languages, such as type definitions, `let`-bindings, recursive functions, etc., we refer the reader to the Andromeda reference page.[6]

In order to distinguish the expressions of AML from the expressions of the object-level type theory, we refer to the former as *computations* to emphasize that their evaluation may have side effects (such as printing things on the screen), and to the latter as *(type-theoretic) terms*. We refer to the types of AML as *ML-types*.

Keep in mind that the ML-level computations can never enter the object-level terms, as the nucleus knows nothing about AML. What looks like AML code inside an object-level term is *always* just AML code that constructs a judgment. For example, a pattern match inside a λ-abstraction, `λ(x:A), match … end`, is a computation that evaluates the `match` statement immediately to obtain an object-level term, which is then abstracted. In contrast, the ML-level function `fun x ⇒ match … end` does suspend the evaluation of its body.

[5] An indirect method to obtain unjoinable contexts is to take a single judgment with the context X:Type, x:X and explicitly substitute in two different ways, replacing X with two distinct types.
[6] `http://www.andromeda-prover.org/meta-language.html`

4.1 ML-types

AML is equipped with static type inference in the style of Hindley-Milner parametric polymorphism [9]. It supports definitions of parametric ML-types, including inductive types. The only non-standard aspect of the ML-type inference arises from the fact that application is overloaded, as it is used both for invoking ML-level application and for building object-level applications. For instance the type of `f`, defined by

```
let f x y = x y
```

could be either `judgment` $\to$ `judgment` $\to$ `judgment` or $(\alpha \to \beta) \to \alpha \to \beta$; in such cases the inferred type constraints are postponed until we are sure that `x` will be a judgment or that at least one of `x` and `y` will not. This strategy works well in practice, with only the occasional application constraint remaining unresolved at the top level.

4.2 Pattern matching

AML pattern matching in `match` statements and `let`-bindings is more flexible than that of Standard ML and related languages. AML patterns need not be linear (i.e., a pattern variable may appear several times in a pattern) and variables may be interpolated into patterns. Pattern variables are prefixed with `?` so that they can be distinguished from interpolated variables. For example,

- the pattern `(?x, ?y)` matches any pair,
- the pattern `(?x, ?x)` matches a pair whose components are equal,
- the pattern `(?x, y)` matches a pair whose second component equals the value of `y`.

Equality in pattern matching always means syntactic identity (α-equivalence in the case of object-level terms), not arbitrary judgmental equalities. The flexibility of pattern matching is handy when we match on values of type `judgment`; see §4.4, where we also discuss patterns for deconstruction of typing judgments.

4.3 Operations and handlers

During evaluation of a computation of ML-type `judgment` the interpreter may need evidence of equality between two types (in order to present it to the nucleus), which it gets by passing control back to user code, together with information on what needs to be done, and how to resume the evaluation once the evidence is obtained. To accomplish this, AML is equipped with algebraic operations and handlers [22] in the style of Eff [3]. We recommend [23, 3] for background reading, and give just a quick overview here. A more detailed discussion on the use of algebraic operations and handlers for the purposes of computing judgments can be found in §4.4.

One way to think of an operation is as a generalized resumable exception: when an operation is invoked it "propagates" outward to the innermost handler that handles it. The handler may then perform an arbitrary computation, and using `yield` c it may resume the execution at the point at which the operation was invoked, yielding the value of c as the result of the operation. Similarly, we can think of a handler as a generalized exception handler, except that it handles one or more operations, as well as values (computations which do not invoke an operation). An example of handlers in action is given in §7.1.

4.4 The datatype judgment

In Andromeda the user *always works with an entire judgment* $\Gamma \vdash t : A$, and never a bare term t. Similarly a type A never stands by itself, but always in a judgment $\Gamma \vdash A : \mathsf{Type}$. The judgments are represented by values of a special primitive type `judgment`.

Judgment forms

The OCaml interface for the nucleus uses distinct abstract datatypes to represent the different judgment forms. These distinctions are not visible to the user, because AML exposes all forms through the single datatype `judgment` whose values are judgments of the form $\Gamma \vdash t : A$. This is possible because $\mathsf{Type} : \mathsf{Type}$ and equality reflection let us express the three forms $\Gamma \vdash A\ \mathsf{type}$, $\Gamma \vdash s \equiv t : A$, and $\Gamma \vdash A \equiv B$ as $\Gamma \vdash A : \mathsf{Type}$, $\Gamma \vdash p : \mathsf{Eq}_A(s, t)$, and $\Gamma \vdash q : \mathsf{Eq}_{\mathsf{Type}}(A, B)$, respectively.

We hope the user finds it simpler to access all object-level entities in a uniform way. On the other hand, having more precise judgment types in AML would help catch potential errors. We discuss this particular design choice in §9.

In AML no direct datatype constructors for `judgment` are available. (Even at the level of OCaml implementation the datatype constructors are invisible outside the nucleus.) Instead, the user may invoke primitive computations of type `judgment` that *look like* term constructors, but really correspond to inference rules of type theory. For instance, an application $c_1\, c_2$, where c_1 and c_2 are computations of type `judgment`, computes an instance of the TERM-APP rule (actually, the version with context joins). The user does not have to provide the explicit typing annotations on the application, as these are derived using a bidirectional typing strategy, as described next.

Inferring and checking modes of evaluation

There are two modes of AML evaluation, *inferring* and *checking*. In inferring mode the type of the result is unconstrained. In checking mode the type is prescribed in advance: there is given a type A (or more precisely, a judgment $\Gamma \vdash A : \mathsf{Type}$) and the computation must evaluate to a judgment of the form $\Delta \vdash t : A$ where Δ extends Γ.

For instance, an application $c_1\, c_2$ is evaluated in inferring mode as follows. First c_1 is evaluated in inferring mode to $\Gamma \vdash s : \prod_{(x:A)} B$ (we discuss what happens if the type of s is not a product below), then c_2 is evaluated in checking mode at type A to $\Delta \vdash t : A$, and the result is $\Gamma \bowtie \Delta \vdash s \mathbin{@^{x:A.B}} t : B[t/x]$.

Judgment computations

The following primitives for computing judgments are provided:

- Primitives for term and type formation:

  $$\mathtt{Type} \qquad \Pi(x{:}c_1),\, c_2 \qquad c_1\, c_2 \qquad \lambda(x{:}c_1),\, c_2 \qquad c_1 \equiv c_2 \qquad \mathtt{refl}\, c.$$

  Note that the notation $c_1 \equiv c_2$ is used for the equality type, rather than for judgmental equality, which the user never writes down explicitly. We emphasize again that these are *not* datatype constructors for forming terms and types of the object-level type theory, but primitive *computations*, with possible side effects, that build *judgments* from sub-judgments by passing through the nucleus.
- Type ascription $c_1 : c_2$, which first evaluates c_2 to $\Gamma \vdash A : \mathsf{Type}$ and then evaluates c_1 in checking mode at type A.

- Top-level `constant` declarations, which introduce new constants.
- The computation `assume` $x : c_1$ `in` c_2, which evaluates c_1 in inferring mode to $\Gamma \vdash A : \mathsf{Type}$, and then c_2 with x `let`-bound to $\Gamma, x_i : A \vdash x_i : A$, where x_i is a freshly generated name. This should not be confused with constant declarations: a constant is an omnipresent part of the signature, while an assumption is local to a judgment in which it appears, and is tracked in assumption sets. Furthermore, we may replace an assumption with a term, using the substitution primitive, but not a constant.
- Substitution c_1 `where` $x = c_2$, which replaces x with the value of c_2 in the value of c_1, assuming the types match.
- The computation `occurs` $c_1\, c_2$, which evaluates c_1 to a judgment $\Delta \vdash x : A$ and c_2 to a judgment $\Gamma \vdash t : B$, and checks whether x appears in Γ. It returns `None` if not, and `Some`$(\Xi \vdash C : \mathsf{Type})$ if x appears in Γ as a variable of type C.
- Computations that generate witnesses for the β-rule and the congruence rules. There are no primitive computations for extensionality rules EQ-ETA and PROD-ETA because they can be declared with a constant by the user. Indeed, we do so in the standard library.
- The computation `natural` c, which witnesses uniqueness of typing. It evaluates c to a judgment $\Gamma \vdash t : A$ and outputs a witness for equality of A and the natural type $\mathcal{N}(t)$ of t. The witnesses are needed when a tactic deconstructs a term and puts it back together, thus obtaining the original term at its natural type.

Judgment patterns

Apart from computations that form judgments, we also need flexible ways of analyzing and deconstructing them. In AML this is done with the `match` statement and judgment patterns of the form $\vdash p_1 : p_2$, where p_2 may be omitted, and p_1 and p_2 are among the following:

- Anonymous pattern `_`, pattern variables $?x$, and interpolated variables x.
- Patterns for matching terms and types:

  $$\texttt{Type} \qquad \Pi(?x{:}p_1),\, p_2 \qquad p_1\, p_2 \qquad \lambda(?x{:}p_1),\, p_2 \qquad p_1 \equiv p_2 \qquad \texttt{refl}\, p.$$

  Note that the patterns for products and abstractions "open up" the binders so that it is possible to pattern-match under the binders; the judgment matched by p_2 can have the bound variable in its context.
- Patterns for matching free variables `_atom` $?x$ and constants `_constant` $?x$.

More precisely, when $\Gamma \vdash t : A$ is matched with $\vdash p_1 : p_2$, the term t is matched with p_1 and the type A with p_2. Assuming the match succeeds, the pattern variables in p_1 and p_2 are bound to sub-judgments that are obtained through inversion lemmas §3.2. The contexts of the sub-judgments are kept minimal thanks to assumption sets. Examples of pattern matching are shown in Appendix B.

Pattern matching is always executed at the AML level; patterns and the `match` statements exist only as computations, and are not part of the object-level terms. To highlight this point, we show the difference between a `match` inside λ-abstraction and an AML function. Assuming a type `A` with two constants `a, b : A` and an endofunction `f : A → A`, the computation

```
(λ (x : A), match x with
             | ⊢ ?g ?y ⇒ y
             | ⊢ _     ⇒ b
            end            ) (f a)
```

evaluates to the judgment $\vdash$ `(λ (x : A), b) (f a) : A`, while

```
(fun x ⇒ match x with
           | ⊢ ?g ?y ⇒ y
           | ⊢ _     ⇒ b
         end               ) (f a)
```

evaluates to the judgment `⊢ a : A`. In the former case matching occurred inside the abstraction, so `x` evaluated to `⊢ x : A` and the second clause matched; in the latter case matching took place when the function was applied, so `x` was bound to `⊢ f a : A` and the first clause matched.

The AML interpreter matches a judgment against a pattern by first asking the nucleus to invert the judgment. The nucleus returns information about which inversion was used and what constituent parts it produced, from which the interpreter calculates whether the pattern matches and how. If there are sub-patterns, the process continues recursively. Pattern matching uses syntactic equality (up to α-equivalence) and never triggers any operations, although an inexhaustive `match` may fail.

At present there are no judgment patterns for analyzing the context of a judgment. Instead, the primitive computation `context` c evaluates c to a judgment $\Gamma \vdash t : A$ and gives the list of all hypotheses in Γ, sorted so that each hypothesis is preceded by its dependencies.

Equality checks and coercions

AML only verifies syntactic equality automatically. It delegates any other equality $\Gamma \vdash s \equiv t : A$ by triggering the operation `equal` $(\Gamma \vdash s : A)$ $(\Gamma \vdash t : B)$, which passes control back to the user-level AML code. The operation may go unhandled, in which case an error is reported, or it may be intercepted by a handler in the user code. The handler may do whatever it wants, but the intended use is for it to attempt to calculate evidence of the given equality. The handler yields `None` if it fails to compute the evidence (in which case the interpreter reports an error), or `Some`$(\Delta \vdash \xi : \mathsf{Eq}_A(s,t))$ if it finds a witness $\Delta \vdash \xi : \mathsf{Eq}_A(s,t)$. Note that the handler is itself a piece of AML code that may recursively trigger further operations and handling thereof.

Apart from equality checking, there are other situations in which the AML interpreter triggers an operation:

- It may happen that AML needs to know why a given type $\Gamma \vdash A : \mathsf{Type}$ is equal to a product type. Unless A is already syntactically equal to a product type, the interpreter triggers an operation `as_prod` $(\Gamma \vdash A : \mathsf{Type})$. It expects a handler to yield `None` upon failure, or `Some`$(\Delta \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, \prod_{(x:B)} C))$ witnessing that A is equal to a product type.
- Similarly, if AML needs to know why $\Gamma \vdash A : \mathsf{Type}$ is equal to an equality type, it triggers an operation `as_eq` $(\Gamma \vdash A : \mathsf{Type})$. It expects the handler to yield `None` or `Some`$(\Delta \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, \mathsf{Eq}_B(s,t)))$.
- If an inferring term evaluates to $\Gamma \vdash t : A$ in checking mode at type $\Delta \vdash B : \mathsf{Type}$, the interpreter does *not* ask for evidence that A and B are equal, but instead triggers the operation `coerce` $(\Gamma \vdash t : A)$ $(\Delta \vdash B : \mathsf{Type})$ that gives the user code an opportunity to replace t with a value of type B. The handler must yield:
  - `NotCoercible` to indicate failure to coerce t to B,
  - `Convertible`$(\Xi \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, B))$ to indicate that A and B are equal, so that AML may apply conversion to t, or
  - `Coercible`$(\Xi \vdash s : B)$ to have t replaced with s.

  This mechanism allows the user to implement various strategies for coercion of values, and control them completely through handlers.

- If the head c_1 of an application $c_1\, c_2$ evaluates to a term $\Gamma \vdash t : A$ where A is not a product type, the interpreter asks the user code to convert t to a function by triggering the operation `coerce_fun` $(\Gamma \vdash t : A)$. The handler should yield `NotCoercible`, `Convertible`$(\Delta \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, \prod_{(x:B)} C))$, or `Coercible`$(\Delta \vdash s : \prod_{(x:B)} C)$, as the case may be.

4.5 References and dynamic variables

As a convenience, AML provides ML-style mutable references. They are used to store the current state of implicit arguments in the standard library (see §6.3).

AML also supports *dynamic variables*. These are globally defined mutable values with dynamic binding discipline. A dynamic variable x is declared and initialized with the top-level command `dynamic` $x = c$. The computation `now` $x = c_1$ `in` c_2 changes x to the value of c_1 locally in the computation of c_2.

AML maintains a dynamic variable `hypotheses`. It is a list of judgments that plays a role in evaluation of computations under binders. To evaluate $\lambda(x{:}A),\, c$, AML generates a fresh variable x_i of type A, binds x to the judgment $x_i : A \vdash x_i : A$, prepends it to `hypotheses`, evaluates c, and abstracts x_i to get the final result. By accessing `hypotheses` the computation c may discover under what binders it is evaluated. For example, in §2 the handler for the `auto` tactic searched `hypotheses` for ways of inhabiting a type.

The standard library uses dynamic variables `betas`, `etas`, `hints`, and `reducing` to store β-hints, η-hints, general hints, and reduction directives. It is important for these variables to follow a dynamic binding discipline so that *local* equality hints work correctly (see §6.2).

5 Soundness of Andromeda

Soundness in Andromeda has both theoretical and engineering aspects.

Theoretical soundness pertains to the differences between the original type theory, (Appendix A) and the type theory implemented in the nucleus (§3.2), which uses assumption sets, context joins, and natural types. In the following we write s^σ for a term s decorated with assumptions σ, i.e., if we remove assumptions sets from the decorated term s^σ we get the ordinary term s. We follow a similar convention for types and context.

▶ **Claim 5.1.** *Given a context Γ, a term s and a type A:*
1. *If $\Gamma \vdash s : A$ is derivable in the original type theory, then $\Delta^\delta \vdash s^\sigma : A^\alpha$ is derivable for some Δ^δ, s^σ, and A^α such that Δ is a subcontext of Γ.*
2. *If $\Gamma^\gamma \vdash s^\sigma : A^\alpha$ is derivable in the implemented type theory, then $\Gamma \vdash s : A$ is derivable in the original type theory.*

We cannot call the statement a theorem because we have not yet proved it in detail. We leave the task as future work for this progress report, and note that we do not anticipate a particularly enlightening or difficult proof, just the usual grinding of cases by structural induction. The most interesting part of the proof will likely by the formalization of the implemented type theory from §3.2, which we have postponed because it has been regularly modified as we gained experience with the implementation.

The second aspect of soundness is an engineering question: how do we know that the implementation of Andromeda works as intended?

▶ **Claim 5.2.** *If Andromeda evaluates a computation to a judgment, then the judgment is derivable from the implemented type theory with respect to the signature containing all the constants declared by the user.*

Let us reiterate the design choices we have made to give credence to the claim. In the OCaml implementation the datatypes representing the judgment forms are all abstract and kept opaque by an interface to a small trusted OCaml module, the *nucleus*. We rely on the soundness of OCaml's type system to ensure that the untrusted remainder of the system cannot forge new values of these abstract types.[7] The nucleus is kept as simple as possible, and it only supports very straightforward type-theoretic constructions which directly correspond to applications of inference rules and admissible rules. Everything else, the AML type inference, the core AML intepreter, the implementation of operations and handlers, and the user code, is on the other side of the barrier and does not influence soundness. In particular, the nucleus does not know anything about AML at all, does not trigger operations, and no AML code can ever enter the object-level terms (so there is no question about having pattern matching, exotic terms involving AML code, or any other part of AML at the object level).

Formally verifying the 1900 lines of the nucleus code is still a tall order to handle, and at present we have no plans to do it. Once we have formulated the implemented type theory, a careful code review of the nucleus will probably unearth some bugs, and hopefully not very many!

There is a third kind of soundness, namely the consistency of the underlying type theory. Obviously, since we included $\mathsf{Type} : \mathsf{Type}$ the theory is at present inconsistent in the sense that all types are inhabited and all judgmental equalities derivable. As soon as we remove $\mathsf{Type} : \mathsf{Type}$ the theory becomes consistent, since what remains are just bare products and equality types with reflection, and these are consistent in virtue of having a model (such as the hereditarily finite sets). We discuss removal of $\mathsf{Type} : \mathsf{Type}$ in §9.

6 The standard library

To test the viability of our design we implemented a small standard library in AML. By design, anything that is implemented in AML is safe: it may not work as expected, or diverge, but it will never produce an invalid judgment, or derive an invalid equality. (Of course, this does not say much until we have dealt with $\mathsf{Type} : \mathsf{Type}$.)

6.1 Equality checking

The most substantial part of the library is a user-extensible equality checking algorithm with rudimentary support for implicit arguments, based on similar ones by Stone and Harper [24] and Coquand [8]. It computes a witness of equality $\Gamma \vdash s \equiv t : A$ in two phases:

- The *type directed* phase computes the weak head-normal form (whnf) of type A to see whether any extensionality rules apply. For instance, if A normalizes to a product $\prod_{(x:B)} C$, the algorithm applies function extensionality PROD-ETA to reduce the equality to $\Gamma, y : B \vdash s \mathbin{@^{x:B.C}} y \equiv s \mathbin{@^{x:B.C}} y : B[y/x]$ at a smaller type. Similarly, if A is an equality type the equality checks succeeds immediately by uniqueness of equality proofs EQ-ETA. Extensionality rules including PROD-ETA and EQ-ETA are user defined (see §6.2).
- Once the type-directed phase simplifies the type so that no further extensionality rules apply, the *normalization phase* computes the weak head-normal forms of s and t and compares them structurally, which generates new equality problems involving subterms.

[7] If we were to reimplement the system in an unsafe language such as C, or if we lacked faith in OCaml, additional mechanisms such as cryptographic signatures could be used.

The equality checking algorithm relies on the computation of weak head-normal forms of terms, which is also implemented by the standard library. Given a term $\Gamma \vdash t : A$, the library computes a witness $\Gamma \vdash \xi : \mathsf{Eq}_A(t, t')$ where t' is in weak head-normal form. It does so by chaining together a sequence of *computation rules* using transitivity of equality. By default the only computation rule is PROD-BETA for reducing β-redices, but the user may install additional rules as explained in §6.2.

6.2 Equality hints

The equality checking algorithm can be extended by the user with new rules, which we call *equality hints*. There are three kinds:

- an *η-hint*, or an extensionality hint, is a term whose type has the form

  $$\textstyle\prod_{(x_1:A_1)} \cdots \prod_{(x_n:A_m)} \prod_{(y_1:B)} \prod_{(y_2:B)} \mathsf{Eq}_{C_1}(t_1, s_1) \to \cdots \to \mathsf{Eq}_{C_m}(t_m, s_m) \to \mathsf{Eq}_B(y_1, y_2).$$

  It is a universally quantified equation with equational preconditions, where the left-hand and the right-hand side of the equation are distinct variables. The equality checking algorithm matches such a hint against the goal. If the match succeeds, the goal is reduced to deriving the preconditions.
- a *β-hint*, or a computation hint, is a term whose type is a universally quantified equation

  $$\textstyle\prod_{(x_1:A_1)} \cdots \prod_{(x_n:A_m)} \mathsf{Eq}_C(s, t).$$

  The weak head-normal form algorithm matches the left-hand side s of the equation against the term. If the match succeeds, it performs a reduction step from s to t.
- a *general hint* is a term whose type is a universally quantified equation

  $$\textstyle\prod_{(x_1:A_1)} \cdots \prod_{(x_n:A_m)} \mathsf{Eq}_C(s, t).$$

  The equality checking algorithm matches such a hint against the goal during the type directed phase to see whether it can immediately dispose of the goal.

In addition, the user may give a *reduction strategy* for a given constant by specifying which of its arguments should be reduced eagerly. This is necessary for the equality checking algorithm to work correctly when we introduce new eliminators. For instance, when we axiomatize simple products $A \times B$, the extensionality rule

$$\textstyle\prod_{(A:\mathsf{Type})} \prod_{(B:\mathsf{Type})} \prod_{(x,y:A\times B)} \\ \mathsf{Eq}_A(\mathtt{fst}\,A\,B\,x, \mathtt{fst}\,A\,B\,y) \to \mathsf{Eq}_A(\mathtt{snd}\,A\,B\,x, \mathtt{snd}\,A\,B\,y) \to \mathsf{Eq}_{A\times B}(x, y)$$

only works correctly if we also specify that the normal form of a projection $\mathtt{fst}\,A\,B\,t$ should have t normalized, and similarly for `snd`. Another example is the recursor for natural numbers, which should eagerly reduce the number at which it is applied.

Examples of equality hints and uses of reduction strategies will be shown in §7. Let us only remark that the hints and reduction strategies may be installed locally, even under a binder using a temporary equality assumption, and that the user is free to install whatever hints they wish, including ones that break completeness of the algorithm. However, as long as hints are confluent and strongly normalizing, the algorithm behaves sensibly.

6.3 Implicit arguments

The standard library provides basic support for implicit arguments. In other systems these are usually implemented with meta-variables, which are not available in AML. In their place, we use ordinary fresh variables generated using the `assume` construct. We refer to these as *implicit variables*. We collect constraints through the operations and handlers mechanism, and resolve them using a simple first-order unification procedure.

More precisely, in the standard library we declare operations

```
operation ? : judgment
operation resolve : judgment → judgment
```

The user may place `?` anywhere where they want the term to be derived automatically, and call `resolve` c to replace the implicit variables with their derived values in the judgment computed by c.

The handler provided by the library keeps a list of implicit variables it has introduced so far, as well as their types and known solutions. The operation `?` may be triggered either in checking or inferring mode. In checking mode at type A and under binders $x_1 : B_1, \ldots, x_n : B_n$, the handler introduces a fresh implicit variable M of type $\prod_{(x_1:B_1)} \cdots \prod_{(x_n:B_n)} A$ and yields $M\,x_1 \ldots x_n$. In inferring mode the type A is not available. For simplicity, at present the library will report an error, although it might be better to create an implicit variable for the $A : \mathsf{Type}$.

During equality checking we may discover that $M\,x_1 \ldots x_n$ should be equal to a term t in which M does not occur. In this case, using `assume` again, the handler generates a term $\xi : \mathsf{Eq}(M, \lambda x_1 \ldots x_m \,.\, t)$, stores it, and also installs it as a β-hint, so that subsequent equality checks take it into account.

The operation `resolve` c is used to replace the implicit variables with their inferred values in the judgment computed by c. Such replacement does not happen automatically because the library cannot guess when is the best moment for doing so. It may be necessary to evaluate several computations before all the implicit variables become known, so we let the user control when resolution should happen.

While we feel quite encouraged by our implementation of equality checking, the implicit arguments feel a bit heavy-handed, and are quite slow. They are a satisfactory proof of concept and a demonstration of the flexibility of operations and handlers, but we need to improve it quite a bit before it becomes useful.

7 Examples

In this section we show Andromeda at work through several examples.

7.1 Proving equality with handlers

As explained in §4.3, when AML is faced with proving a non-trivial equality, it delegates it to user code by triggering the operation `equal`. To see how this works, let us walk through a computation that constructs a term witnessing symmetry of equality (without the standard library installed):

```
λ (A : Type) (x y : A) (p : x ≡ y),
  (handle
    refl x : y ≡ x
   with
   | equal x y ⇒ yield (Some p)
   end)
```

The λ-abstraction introduces a type A, elements x, y of type A, and a witness p of equality between x and y. Next, the type ascription is evaluated, with the enveloping handler installed. First y ≡ x is evaluated to the equality type $\mathsf{Eq}_{\mathsf{A}}(\mathsf{y},\mathsf{x})$ and then refl x is evaluated in checking mode at this type. This triggers a sub-computation of x and verification that x equals y (and a trivial equality check that x equals to itself). At this point AML triggers the operation equal x y, asking for evidence of equality. The enveloping handler intercepts the operation and yields the evidence p.

The result of the computation is displayed by Andromeda without typing annotations and assumption sets as

```
⊢ λ (A : Type) (x : A) (y : A) (_ : x ≡ y), refl x
  : Π (A : Type) (x : A) (y : A), x ≡ y → y ≡ x
```

7.2 Dependent sums

Our second example shows how to axiomatize dependent sums. This time we use the standard library and rely on its equality checking. We start by postulating the type and term constructors:

```
constant Σ : Π (A : Type) (B : A → Type), Type
constant existT : Π (A : Type) (B : A → Type) (a : A), B a → Σ A B
```

Next, we postulate the projections and tell the standard library that that the third argument of a projection should be evaluated eagerly, so that we get a working extensionality rule later on:

```
constant π₁ : Π (A : Type) (B : A → Type), Σ A B → A
now reducing = add_reducing π₁ [lazy, lazy, eager]

constant π₂ : Π (A : Type) (B : A → Type) (p : Σ A B), B (π₁ A B p)
now reducing = add_reducing π₂ [lazy, lazy, eager]
```

It remains to postulate equalities, and install them as hints. The β-rules are straightforward, except that we must install the β-rule for the first projection before we postulate the second projection, or else Andromeda does not know why the second projection is well typed:

```
constant π₁_β :
  Π (A : Type) (B : A → Type) (a : A) (b : B a),
    (π₁ A B (existT A B a b) ≡ a)

now betas = add_beta π₁_β

constant π₂_β :
  Π (A : Type) (B : A → Type) (a : A) (b : B a),
    (π₂ A B (existT A B a b) ≡ b)

now betas = add_beta π₂_β
```

Similarly, to convince Andromeda that the extensionality rule is well typed, we need to install a *local* hint, as follows (the function symmetry is part of the standard library and it computes the symmetric version of an equality):

```
constant Σ_η :
  Π (A : Type) (B : A → Type) (p q : Σ A B)
    (ξ : π₁ A B p ≡ π₁ A B q),
    now hints = add_hint (symmetry ξ) in
      π₂ A B p ≡ π₂ A B q → p ≡ q

now etas = add_eta Σ_η
```

7.3 Natural numbers

The standard library provides functions for calculating weak head-normal forms that can be used as a computation device at the level of type-theoretic terms. We show how this is done by axiomatizing natural numbers and computing with them.

We postulate the type of natural numbers and its constructors

```
constant nat : Type
constant O : nat
constant S : nat → nat
```

and the induction principle

```
constant nat_rect : Π (P : nat → Type),
  P O → (Π (n : nat), P n → P (S n)) → Π (m : nat), P m
```

The weak head-normal form of the eliminator should have the fourth argument normalized:

```
now reducing = add_reducing nat_rect [lazy, lazy, lazy, eager]
```

To get computation going, we need the computation rules for the eliminator:

```
constant nat_β_O :
  Π (P : nat → Type) (x : P O) (f : Π (n : nat), P n → P (S n)),
    nat_rect P x f O ≡ x

constant nat_β_S :
  Π (P : nat → Type) (x : P O) (f : Π (n : nat), P n → P (S n))
    (m : nat),
    nat_rect P x f (S m) ≡ f m (nat_rect P x f m)
```

which we install as β-hints:

```
now betas = add_betas [nat_β_O, nat_β_S]
```

At this point, we can compute with the recursor, but there is a better way. In Andromeda there is no built-in notion of "definition" at the level of type theory (one can always use ML-level `let`-bindings, but those are always evaluated which has the undesirable effect of complete unfolding of all definitions). Instead, we break down a definition into a declaration of the constant and its defining equality. If we install the defining equality as a β-hint, a definition behaves like it would in other proof assistants, but that is just one possibility.

For example, we may define addition as follows:

```
constant ( + ) : nat → nat → nat
constant plus_def :
  Π (n m : nat), n + m ≡ nat_rect (λ _, nat) n (λ _ x, S x) m
```

Note that `plus_def` could be written as

```
constant plus_def' :
  ( + ) ≡ (λ (n m : nat), nat_rect (λ _, nat) n (λ _ x, S x) m)
```

The difference between the two is visible when we use them as β-hints: `plus_def` will unfold only after it has been applied to two arguments, whereas `plus_def'` will do so immediately.

We can derive Peano axioms by using `plus_def` as a local β-hint:

```
let plus_O =
  now betas = add_beta plus_def in
    (λ n, refl n) : Π (n : nat), n + O ≡ n

let plus_S =
  now betas = add_beta plus_def in
    (λ n m, refl (n + (S m))) : Π (n m : nat), n + (S m) ≡ S (n + m)
```

We are free to use the Peano axioms for computation rather than `plus_def`, so we install them globally as β-hints:

```
now betas = add_betas [plus_O, plus_S]
```

It should be clear from this that Andromeda is quite flexible, which is good for experimentation and tight control of how things are done, but is also bad because the user has to be more specific in what they want. The overall usability of the system depends on having a good standard library with sensible default settings.

The definition of multiplication and its Peano axioms are derived similarly:

```
constant ( * ) : nat → nat → nat
constant mult_def :
  Π (n m : nat), n * m ≡ nat_rect (λ _, nat) O (λ _ x, x + n) m

let mult_O =
  now betas = add_beta mult_def in
    (λ n, refl O) : Π (n : nat), n * O ≡ O

let mult_S =
  now betas = add_beta mult_def in
    (λ n m, refl (n * (S m))) : Π (n m : nat), n * (S m) ≡ n * m + n

now betas = add_betas [mult_O, mult_S]
```

To compute with numbers, we use the standard library function `whnf` that computes evidence that the given term is equal to its weak head-normal form:

```
do now reducing = add_reducing S [eager] in
   now reducing = add_reducing ( * ) [eager, eager] in
   now reducing = add_reducing ( + ) [eager, eager] in
     whnf ((S (S (S O))) * (S (S (S (S O)))))
```

The `do` command is the top-level command for evaluating a computation. Notice that we locally set the arguments of the successor constructor, addition, and multiplication to be computed eagerly. The effect of this is that the weak head-normal form is not weak or head-normal anymore, but rather a strongly normalizing call-by-value strategy. Thus Andromeda outputs

```
⊢ refl (S (S (S (S (S (S (S (S (S (S (S (S O))))))))))))
  : S (S (S O)) * S (S (S (S O))) ≡
    S (S (S (S (S (S (S (S (S (S (S (S O)))))))))))
```

It would be easy to obtain just the result, which is the left-hand side of the equality type. Notice that the proof of equality between 3×4 and 12 is a reflexivity term, even though the normalization procedure generated the proof by stringing together a large number of reduction steps. In order to keep equality proofs small, the standard library aggressively replaces equality proofs with reflection terms, using the fact that whenever $p : \mathsf{Eq}_A(s,t)$ then also $\mathsf{refl}_A\ t : \mathsf{Eq}_A(s,t)$.

7.4 Untyped λ-calculus

An example that cannot be done easily in proof assistants based on intensional type theory is in order. Let us axiomatize the untyped λ-calculus as a type that is judgmentally equal to its own function space, and show that it possesses a fixed-point operator.

We first postulate that there is a type equal to its function space:

```
constant D : Type
constant D_reflexive : D ≡ (D → D)
```

We must *not* install `D_reflexive` as a β-hint because it would lead to non-termination. Instead, we install it and its symmetric version as general hints:

```
now hints = add_hints [D_reflexive, symmetry D_reflexive]
```

With these, whenever AML needs to know that `D` and `D → D` are equal, the standard library will provide `D_reflexive`, or its symmetric version, as evidence.

Now, we can simply define the fixed-point operator:

```
let fix =
  (λ f,
    let y = (λ x : D, f ((x : D → D) x)) in
    y y)
  : (D → D) → D
```

The self-application of `x` is well-typed because Andromeda knows that `x` of type `D` also has type `D → D`, thanks to the hints. We did have to explicitly coerce `x` to the function type. (An alternative would be to use the coercion mechanism, which is demonstrated in §7.5.) Once we overcome the problem of typing the fixed-point operator, the usual mechanisms suffice to show that it does in fact compute fixed points:

```
let fix_eq =
  (λ f, refl (fix f)) : Π (f : D → D), fix f ≡ f (fix f)
```

It is a bit trickier to give a type to a term without weak head-normal form, such as $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$. We must block β-reduction of this particular β-redex, without blocking all of them. To achieve this, we first introduce an alias `D'` for the type `D`:

```
constant D' : Type
constant eq_D_D' : D ≡ D'
```

Next, we define the auxiliary term δ and give it the type `D → D`:

```
let δ = (λ x : D, (x : D → D) x)
```

We now form the self-application $\delta\ \delta$ at type `D'`:

```
let Ω =
  now hints = add_hints [eq_D_D', symmetry eq_D_D'] in
  (δ : D' → D') (δ : D) : D
```

We have the desired term in which β-reduction is blocked because the inner λ-abstractions are typed at `D` and the outer application at `D'`. From here, Andromeda happily computes with Ω without ever attempting to reduce it (installing `eq_D_D'` as a global hint would be a mistake).

The preceding example should be taken as a proof of concept only. We have reached the limits of our small standard library. A more serious development of the untyped λ-calculus would use a custom equality-checking algorithm instead of manually juggling hints and type ascriptions.

7.5 Universes

The final example shows how to use coercions and operations to implement a universe à la Tarski. We postulate a universe `U`, whose elements should be thought of as *names* of types, with an operation `El` that converts the names to the corresponding types:

```
constant U : Type
constant El : U → Type
```

Because `El` is an eliminator, its normal form should have the argument in normal form, so we tell the library to normalize it eagerly:

```
now reducing = add_reducing El [eager]
```

Next, we postulate that the universe contains names for products and equality types, and install the relevant equations as β-hints:

```
constant pi : Π (a : U), (El a → U) → U
constant El_pi :
  Π (a : U) (b : El a → U), El (pi a b) ≡ (Π (x : El a), El (b x))
now betas = add_beta El_pi

constant eq : Π (a : U), El a → El a → U
constant El_eq :
  Π (a : U) (x y : El a), El (eq a x y) ≡ (x ≡ y)
now betas = add_beta El_eq
```

For testing purposes we put the name `b` of a basic type `B` into the universe:

```
constant B : Type
constant b : U
constant El_b : El b ≡ B
now betas = add_beta El_b
```

In principle we can work with `U` and `El`, but explicit uses of `El` gets tedious quickly. Ideally we want Andromeda to translate between names and their types automatically, which is achieved with a handler that intercepts coercion requests. It is easy to coerce names to types with `El`, for instance:

```
handle
  (λ x : b, x) : pi b (λ _, b)
with
 | coerce (⊢ ?t : U) (⊢ Type) ⇒ yield (Coercible (El t))
end
```

In the λ-abstraction AML found the name `b` but expected a type, therefore it triggered a coercion operation. The handler intercepted it and yielded `El b`. The process was repeated when AML found `pi b (λ _, b)` instead of a type. The final result printed by Andromeda is

```
⊢ λ (x : El b), x : El (pi b (λ (_ : El b), b))
```

We have to work harder to perform the reverse coercion, when a type is encountered where its code was expected. One first has to implement an AML function `name_of` that takes a type and returns its name, if it can find one. We do not show its implementation here, and ask the interested readers to consult the examples that come with the source code. Using `name_of` we can handle translation between types and names in both directions with the handler

```
let universe_handler =
handler
  | coerce (⊢ ?a : U) (⊢ Type) ⇒ yield (Coercible (El a))
  | coerce (⊢ ?T : Type) (⊢ U) ⇒
    match name_of T with
     | None ⇒ yield NotCoercible
     | Some ?name ⇒ yield (Coercible name)
    end
  end
```

We added a clause that intercepts coercions from `Type` to `U` and uses `name_of`. The handler automatically translates names to types and vice versa. For instance, the computation

```
with universe_handler handle
  (Π (x : b), x ≡ x) : U
```

evaluates to

```
⊢ pi b (λ (y : El b), eq b y y) : U
```

In one direction the handler coerced the name `b` to the type `B`, and in the other the type `Π (x : B), x ≡ x` to its name, as shown above.

8 Related work

Andromeda draws heavily on the experience and ideas from other proof assistants. It is difficult to do justice to all of them. Its overall design follows the tradition of LCF [12] and its descendants [15, 16, 13]. However, LCF and many of its descendants use Church's *simple* type theory [6], whereas Andromeda is based on the *dependent* type theory of Martin-Löf [17]. Consequently, Andromeda cannot advantageously integrate the ML-level and the object-level types. There is by necessity a sharp distinction between the statically typed ML-level and the dynamically evaluated type-theoretic judgments.

It makes sense to compare Andromeda to other proof assistants based on dependent type theory [19, 7, 10]. For instance, the evaluation strategy for judgments in Andromeda is based on bidirectional type-checking found in dependently-typed assistants. Andromeda is primarily a special-purpose programming language, whereas Coq, Agda, and Lean are tools for interactive proof development. The difference in philosophy of design is visible in the level of control given to the user. Andromeda gives the user full control of the system, and expects them to implement their own proof development tools, whereas Coq and Agda provide more of an end-user environment with a rich selection of ready-made tools. It is interesting to note that recently Coq and Agda have both started giving the user more control. New versions of Coq allow the use of tactics inside type-theoretic terms [26, §2.11.2] and allow fine-tuning of Coq's unification algorithm [27]. Agda even lets the user install new normalization rules [1] that might break the system.

We already mentioned that NuPRL [2] validates equality reflection by interpreting types as partial equivalence relations on terms of a computational model, namely an extension of the untyped λ-calculus. We do not wish to make such a commitment in Andromeda, and instead allow interpretations that are inconsistent with computational type theory.

9 Future work

We feel that Andromeda shows a promising way to design a proof assistant based on type theory with equality reflection, but much remains to be done.

Syntactic sugar and end-user support

AML turned out to be a useful tool for the implementers of the standard library. If we imagine that the end-user is a mathematician who just wants to do mathematics, without learning the intricacies of operations and handlers, then we need further support for creating a more user friendly environment. There ought to be ways of introducing new syntactic constructs, and reasonable error reporting by the standard library. We are not quite sure how to provide such functionality. The approach taken by Bowman [4] seems interesting. Another possibility is to allow user-defined notations in the style of Coq, or to completely separate the end-user interface and AML.

Formal verification of the meta-theoretic properties

While we carefully designed the underlying type theory and made sure it is precisely clear what the type-theoretic rules are, we have not formally verified that the system has the desired meta-theoretic properties, such as uniqueness of typing, validity of inversion principles, and well-behaved context joins. We expect no trouble here, but do insist on formal verification. In the early stages of implementation we managed to delude ourselves more than once about the properties of the underlying type theory.

Recording derivations

Like all LCF-style proof assistants, Andromeda does not record derivations, only their conclusions. (In fact, all practical proof assistants do this, though some implement type theories that allow derivations to be reconstructed.) There might be situations in which we wish to record or communicate the derivation. For instance, we might want to send the derivation to another proof assistant for independent verification. This can be accomplished with a minor modification of AML: if we implement *all* calls of AML to the nucleus as operations (whose default handler is the nucleus), then the user can intercept them and do whatever they like: record them, communicate them, or modify them to obtain a proof translation. Alternatively, the `judgment` type in the nucleus could be made into the type of derivations with no breaking changes to the interface.

Removal of Type : Type

A major forthcoming modification of the current system is elimination of $\mathsf{Type} : \mathsf{Type}$. The syntax of AML takes advantage of $\mathsf{Type} : \mathsf{Type}$ to conflate term and type judgments within a single abstract type `judgment`.

But the nucleus does not rely on $\mathsf{Type} : \mathsf{Type}$ at all and separates the various judgment forms into separate abstract datatypes. There is no technical difficulty in removing $\mathsf{Type} : \mathsf{Type}$, but the question is what to replace it with. One possibility is to add a basic type U and a basic type family El indexed by U, and then use these as a Tarski-style universe, with a standard library employing techniques of §7.5 to make the system usable. (This can be extended to multiple universes if desired.) Paolo Capriotti's recently suggested such a setup [5], based on semantic considerations in categories of presheaves. Finally, AML could be modified to support several abstract datatypes, one for each form of object-level judgment.

Once this is done, several interesting possibilities arise. The user could hypothesize any universe structure they like, including putting back $\mathsf{Type} : \mathsf{Type}$. We might even be able to remove equality reflection from the nucleus, and make it user-definable. We hope to report on these exciting developments in the near future.

References

1 Andreas Abel and Jesper Cockx. Sprinkles of extensionality for your vanilla type theory. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016*, Novi Sad, Serbia, May 2016.

2 S.F. Allen, M. Bickford, R.L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

3 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, January 2015.

4 William J. Bowman. Growing a proof assistant. In *Higher-Order Programming with Effects*, 2016.

5 Paolo Capriotti. Notions of type formers. In *23rd International Conference on Types for Proofs and Programs (TYPES)*. Budapest, Hungary, May 29–June 1 2017.

6 Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. `doi:10.2307/2266170`.

7 Coq Development Team. The Coq proof assistant. Available at `http://coq.inria.fr/`, 2016.

8 Thierry Coquand. An Algorithm for Testing Conversion in Type Theory. In Gérard Huet and G. Plotkin, editors, *Logical frameworks*, pages 255–277. Cambridge University Press, 1991.

9 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

10 Leonardo de Moura, Soonho Kong, Floris van Doorn, Jakob von Raumer, and Jeremy Avigad. The Lean theorem prover (system description). In *25th International Conference on Automated Deduction (CADE-25)*, 2015.

11 Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

12 Michael J. Gordon, Arthur J. Milnter, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Number 78 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

13 John Harrison. The HOL Light theorem prover. Available at `https://www.cl.cam.ac.uk/~jrh13/hol-light/`.

14 Martin Hofmann. *Extensional constructs in intensional type theory.* CPHC/BCS distinguished dissertations. Springer, 1997.

15 HOL Interactive Theorem Prover. Available at `https://hol-theorem-prover.org`.

16 Isabelle proof assistant. Available at `https://isabelle.in.tum.de`.

17 Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory.* Studies in Proof Theory. Bibliopolis, 1984.

18 Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

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

20 OCaml programming language. Available at `https://ocaml.org`.

21 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 80–94, 2009.

22 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. `doi:10.2168/LMCS-9(4:23)2013`.

23 Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. `doi:10.1016/j.entcs.2015.12.003`.

24 Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4):676–722, October 2006.

25 Thomas Streicher. Investigations into intensional type theory. Habilitation Thesis, Ludwig-Maximilians Universität, 1993.

26 Coq Development Team. The Coq proof assistant reference manual, version 8.5. Available at `https://coq.inria.fr/distrib/8.5/refman/`.

27 Beta Ziliani and Matthieu Sozeau. A Unification Algorithm for Coq featuring Universe Polymorphism and Overloading. In *ACM SIGPLAN International Conference on Functional Programming 2015*, 2015.

A The rules of type theory

In this appendix we give the formulation of type theory in a declarative way that minimizes the number of judgments, and so is better suited for a semantic account. We omit formal treatment of bound variables and substitution, which is standard.

A.1 Syntax

Contexts

Γ, Δ	$::=$	$\bullet$	empty context
	$\mid$	$\Gamma,\, x : A$	context Γ extended with $x : A$

Terms and types

s, t, A, B	$::=$	Type	universe
	$\mid$	$\prod_{(x:A)} B$	product
	$\mid$	$\mathsf{Eq}_A(s, t)$	equality type
	$\mid$	x	variable
	$\mid$	$\lambda x{:}A.B\,.\,t$	λ-abstraction
	$\mid$	$s \mathbin{@^{x:A.B}} t$	application
	$\mid$	$\mathsf{refl}_A\ t$	reflexivity

A.2 Judgments

$\Gamma\ \mathsf{ctx}$	Γ is a well formed context
$\Gamma \vdash t : A$	t is a well formed term of type A in context Γ
$\Gamma \vdash s \equiv t : A$	s and t are equal terms of type A in context Γ

We use the following abbreviations:

$\Gamma \vdash A\ \mathsf{type}$	abbreviates $\Gamma \vdash A : \mathsf{Type}$
$\Gamma \vdash A \equiv B$	abbreviates $\Gamma \vdash A \equiv B : \mathsf{Type}$

A.3 Contexts

CTX-EMPTY
$$\frac{}{\bullet\ \mathsf{ctx}}$$

CTX-EXTEND
$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A\ \mathsf{type} \qquad x \notin \mathrm{dom}(\Gamma)}{(\Gamma,\, x : A)\ \mathsf{ctx}}$$

A.4 Terms and types

Conversion

$$\textsc{term-ty-conv}\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}$$

Variable

$$\textsc{term-var}\quad \frac{(\Gamma,\, x \!:\! A)\ \mathsf{ctx}}{\Gamma,\, x \!:\! A \vdash x : A} \qquad \textsc{term-var-skip}\quad \frac{(\Gamma,\, y \!:\! B)\ \mathsf{ctx} \qquad \Gamma \vdash x : A}{\Gamma,\, y \!:\! B \vdash x : A}$$

Universe

$$\textsc{ty-type}\quad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type}\ \mathsf{type}}$$

Product

$$\textsc{ty-prod}\quad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma,\, x \!:\! A \vdash B\ \mathsf{type}}{\Gamma \vdash \prod_{(x:A)} B\ \mathsf{type}} \qquad \textsc{term-abs}\quad \frac{\Gamma,\, x \!:\! A \vdash t : B}{\Gamma \vdash (\lambda x{:}A.B\,.\,t) : \prod_{(x:A)} B}$$

$$\textsc{term-app}\quad \frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Gamma \vdash t : A}{\Gamma \vdash s \mathbin{@^{x:A.B}} t : B[t/x]}$$

Equality type

$$\textsc{ty-eq}\quad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash s : A \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{Eq}_A(s,t)\ \mathsf{type}} \qquad \textsc{term-refl}\quad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}_A\ t : \mathsf{Eq}_A(t,t)}$$

A.5 Equality

General rules

$$\textsc{eq-refl}\quad \frac{\Gamma \vdash t : A}{\Gamma \vdash t \ \equiv\ t : A} \qquad \textsc{eq-sym}\quad \frac{\Gamma \vdash t \ \equiv\ s : A}{\Gamma \vdash s \ \equiv\ t : A} \qquad \textsc{eq-trans}\quad \frac{\Gamma \vdash s \ \equiv\ t : A \qquad \Gamma \vdash t \ \equiv\ u : A}{\Gamma \vdash s \ \equiv\ u : A}$$

Conversion

$$\textsc{eq-ty-conv}\quad \frac{\Gamma \vdash s \ \equiv\ t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash s \ \equiv\ t : B}$$

Equality reflection

$$\textsc{eq-reflection}\quad \frac{\Gamma \vdash u : \mathsf{Eq}_A(s,t)}{\Gamma \vdash s \ \equiv\ t : A}$$

Computation

PROD-BETA

$$\frac{\Gamma, x:A \vdash s : B \qquad \Gamma \vdash t : A}{\Gamma \vdash (\lambda x{:}A.B \,.\, s) \mathbin{@}^{x:A.B} t \;\equiv\; s[t/x] : B[t/x]}$$

Extensionality

EQ-ETA

$$\frac{\Gamma \vdash t : \mathsf{Eq}_A(s,u) \qquad \Gamma \vdash v : \mathsf{Eq}_A(s,u)}{\Gamma \vdash t \;\equiv\; v : \mathsf{Eq}_A(s,u)}$$

PROD-ETA

$$\frac{\begin{array}{c}\Gamma \vdash s : \prod_{(x:A)} B \qquad \Gamma \vdash t : \prod_{(x:A)} B \\ \Gamma, x:A \vdash (s \mathbin{@}^{x:A.B} x) \;\equiv\; (t \mathbin{@}^{x:A.B} x) : B\end{array}}{\Gamma \vdash s \;\equiv\; t : \prod_{(x:A)} B}$$

A.5.1 Congruences

Type formers

CONG-PROD

$$\frac{\Gamma \vdash A \equiv C \qquad \Gamma, x:A \vdash B \equiv D[x/y]}{\Gamma \vdash \prod_{(x:A)} B \equiv \prod_{(y:C)} D}$$

CONG-EQ

$$\frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash s \;\equiv\; u : A \qquad \Gamma \vdash t \;\equiv\; v : A}{\Gamma \vdash \mathsf{Eq}_A(s,t) \equiv \mathsf{Eq}_B(u,v)}$$

Products

CONG-ABS

$$\frac{\Gamma \vdash A \equiv C \qquad \Gamma, x:A \vdash B \equiv D[x/y] \qquad \Gamma, x:A \vdash s \;\equiv\; t[x/y] : B}{\Gamma \vdash (\lambda x{:}A.B \,.\, s) \;\equiv\; (\lambda y{:}C.D \,.\, t) : \prod_{(x:A)} B}$$

CONG-APP

$$\frac{\begin{array}{c}\Gamma \vdash A \equiv C \qquad \Gamma, x:A \vdash B \equiv D[x/y] \\ \Gamma \vdash s \;\equiv\; u : \prod_{(x:A)} B \qquad \Gamma \vdash t \;\equiv\; v : A\end{array}}{\Gamma \vdash (s \mathbin{@}^{x:A.B} t) \;\equiv\; (u \mathbin{@}^{y:C.D} v) : B[t/x]}$$

Equality types

CONG-REFL

$$\frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash s \;\equiv\; t : A}{\Gamma \vdash \mathsf{refl}_A\ s \;\equiv\; \mathsf{refl}_B\ t : \mathsf{Eq}_A(s,s)}$$

B The auto tactic

We include here the complete code for implementing a simple `auto` tactic from § 2.

We first define the `map` function to show how AML syntax works, and the auxiliary `apply` function that folds application of a function over a list of arguments:

```
let rec map f xs =
  match xs with
  | [] ⇒ []
  | ?x :: ?xs ⇒ (f x) :: (map f xs)
  end

let rec apply f xs =
  match xs with
  | [] ⇒ f
  | ?x :: ?xs ⇒ apply (f x) xs
  end
```

Next, we declare the `failure` operation that is triggered when the search fails:

```
operation failure : judgment
```

Next we define the function `subgoals` that takes a goal `A` and a hypothesis `B` and computes a list of subgoals that together with `B` imply `A`. For instance, if `B` is equal to `C → D → A` then the computed subgoals are `[C, D]`:

```
let rec subgoals A B =
  match B with
  | ⊢ A ⇒ []
  | ⊢ ?P → ?Q ⇒ P :: (subgoals A Q)
  | _ ⇒ [failure]
  end
```

The function `derive` takes a goal `A` and attempts to derive it. If the goal is an implication, it introduces the antecedent as a hypothesis and calls itself recursively. Otherwise it tries to prove the goal from the current hypotheses by simple backchaining:

```
let rec derive A =
  match A with
  | ⊢ ?P → ?Q ⇒ λ (x : P), derive Q
  | ⊢ _ ⇒ backchain A hypotheses
  end

and backchain A lst =
  match lst with
  | [] ⇒ failure
  | (⊢ ?f : ?B) :: ?lst ⇒
    handle
      apply f (map derive (subgoals A B))
    with
      failure ⇒ backchain A lst
    end
  end
```

Note how `backchain` uses a handler to intercept `failure`, just like an ordinary exception handler does. Finally, we declare an operation `auto` and define a global handler that handles it. The handler only works when `auto` is used in checking mode:

```
operation auto : judgment

handle
| auto : ?T' ⇒
    match T' with
    | Some ?T ⇒ derive T
    | None ⇒ failure
    end
end
```

Now we can use `auto` to inhabit types. For example,

```
(λ (X : Type), auto : X → X)
```

computes to

```
⊢ λ (X : Type) (x : X), x : Π (X : Type), X → X
```

and given the types

```
constant A : Type
constant B : Type
constant C : Type
```

the computation

```
auto : (A → B → C) → (A → B) → (A → C)
```

results in

```
⊢ λ (x : A → B → C) (x0 : A → B) (x1 : A), x x1 (x0 x1)
  : (A → B → C) → (A → B) → A → C
```

Realizability at Work: Separating Two Constructive Notions of Finiteness

Marc Bezem
Universitetet i Bergen, Institutt for informatikk, Postboks 7800, N-5020 Bergen, Norway
bezem@ii.uib.no

Thierry Coquand
Chalmers tekniska högskola, Data- och informationsteknik, 412 96 Göteborg, Sweden
coquand@chalmers.se

Keiko Nakata
SAP Innovation Center Network, Konrad-Zuse-Ring 10, 14469 Potsdam, Germany

Erik Parmann
Universitetet i Bergen, Institutt for informatikk, Postboks 7800, N-5020 Bergen, Norway
eparmann@gmail.com

Abstract

We elaborate in detail a realizability model for Martin-Löf dependent type theory with the purpose to analyze a subtle distinction between two constructive notions of finiteness of a set A. The two notions are: (1) A is Noetherian: the empty list can be constructed from lists over A containing duplicates by a certain inductive shortening process; (2) A is streamless: every enumeration of A contains a duplicate.

2012 ACM Subject Classification Theory of computation → Type theory, Theory of computation → Constructive mathematics

Keywords and phrases Type theory, realizability, constructive notions of finiteness

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.6

Acknowledgements The research for this paper has been carried out for a large part while the first two authors were member 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. The paper has benefitted much from comments by Michael Beeson on an early draft.

1 Introduction

We will analyze in detail in type theory the following observation. Let P be a unary predicate of natural numbers. Define

$$\overline{P} = \{n \in \mathsf{N} \mid \forall k < n.\, Pk \vee \neg Pk\}$$

Of course, in classical mathematics $\overline{P} = \mathsf{N}$, but in constructive mathematics this is not true for all P. Let D be a unary predicate of lists over $\overline{P}$ with $D\ell$ expressing that ℓ contains a *duplicate*, that is, two occurrences of the same natural number (which by definition is in $\overline{P}$). Then one can prove constructively that D is inductive in the following sense ("inductive shortening"):

if $D(x :: \ell)$ for all $x \in \overline{P}$, then $D\ell$

The proof is as follows. Let ℓ be a list over $\overline{P}$ and assume $D(x :: \ell)$ for all $x \in \overline{P}$ (IH). We clearly have $D\ell \vee \neg D\ell$, so we can reason by contradiction. Assume $\neg D\ell$. We clearly have $\ell = \mathsf{nil} \vee \neg \ell = \mathsf{nil}$, so we can reason by cases. If $\ell = \mathsf{nil}$, we use $0 \in \overline{P}$ and apply IH to get $D(0 :: \mathsf{nil})$, which is absurd. If $\ell \neq \mathsf{nil}$, then ℓ contains a largest natural number, say $m \in \overline{P}$. If $Pm \vee \neg Pm$, then $m + 1 \in \overline{P}$ and we can apply IH to get $D((m + 1) :: \ell)$, which per construction yields $D\ell$, as $m+1$ is larger than all elements of ℓ, and therefore not duplicating some element of ℓ. This conflicts with the assumption $\neg D\ell$, so we conclude $\neg(Pm \vee \neg Pm)$, which is also absurd, since $\neg\neg(Pm \vee \neg Pm)$ is a constructive tautology. This completes the proof of the inductivity of D.

Since D is inductive in the way above, and $D\mathsf{nil}$ is absurd, D cannot be an inductive bar in the tree of lists over $\overline{P}$. (The concept of an inductive bar making a tree of lists well-founded will be formalized by the concept of a Noetherian relation in Section 3.) This should not come as a surprise, since classically $\overline{P} = \mathsf{N}$, so the tree of lists without duplicates is classically not well-founded, since it is always possible to extend a list without introducing a duplicate. The above argument shows that we can also do this constructively with lists over $\overline{P}$, for any unary predicate P.

In view of the above, only a non-classical axiom can cause $\overline{P}$ to have fewer elements. Of course we cannot downright postulate $\exists n \in \mathsf{N}.\, \neg(Pn \vee \neg Pn)$ without running into inconsistency. But we can consistently postulate $\Phi = \neg\forall n \in \mathsf{N}.\, Pn \vee \neg Pn$. From Φ we immediately infer $\neg\forall n \in \mathsf{N}.\, n \in \overline{P}$, so $\overline{P} \neq \mathsf{N}$. Since one easily (and constructively) sees that $\overline{P}$ is downward closed, Φ "somehow" achieves that $\overline{P}$ is finite. Of course the results in the previous paragraphs still stand, and Φ does not make $\overline{P}$ finite in the sense that D is an inductive bar.

In order to better understand in which way Φ achieves that $\overline{P}$ is finite, assume $f : \mathsf{N} \to \overline{P}$ is an injection. Then we would be able to find arbitrarily large elements in $\overline{P}$, and hence prove $\forall n \in \mathsf{N}.\, Pn \vee \neg Pn$, conflicting with Φ. As a consequence, no $f : \mathsf{N} \to \overline{P}$ is injective. In other words, for every $f : \mathsf{N} \to \overline{P}$ we have $\neg\forall m, n \in \mathsf{N}.\, fm{=}fn \to m{=}n$. By an appeal to Markov's Principle we get $\exists m, n \in \mathsf{N}.\, fm = fn \wedge m \neq n$, that is, there exists a prefix of f which contains a duplicate. If we view lists over $\overline{P}$ not containing duplicates as a tree, then we have just proved that this tree is well-founded, which is another way of saying that $\overline{P}$ is finite. (This notion of finiteness will be made precise by the concept of a streamless relation in Section 3.)

The results in the previous two paragraphs capitalize on Markov's Principle (a weak form of classical reasoning) being consistent with Φ (a non-classical axiom). They are known to co-exist in, for example, the recursive model of type theory. In that model, Φ can be validated by the unsolvability of the halting problem. Although this model has been known for quite some time, it is an important side-goal of this paper to give a detailed account. Our main objective is to formalize the argument above in type theory, and prove that finiteness based on equality being Noetherian is strictly stronger than finiteness based on equality being streamless. This confirms a conjecture formulated by Coquand and Spiwack in [3]. We also give a novel proof that every Noetherian relation is streamless. This proof is due to the last author [11, Chapter 4] and formalized in Coq [10].

In type theory, a subset of N is a type $\Sigma x{:}\mathsf{N}.\, Px$ given a type family $P : \mathsf{N} \to \mathsf{U}$. Elements of this Σ-type (to be defined in Section 2) are pairs (n, p) consisting of a natural number n and a proof $p : Pn$. It may happen that also $p' : Pn$, with p' different from p. This phenomenon is called *proof relevance*. We do not want to count (n, p) and (n, p') as two elements of the subset of N defined by P. Therefore we only count the first projections of objects in $\Sigma x{:}\mathsf{N}.\, Px$. Another approach would be to take the type $\Sigma x{:}\mathsf{N}.\, \|Px\|$, where $\|\_\|$ stands for propositional truncation, a way of making all inhabitants of Px indistinguishable, see [15, Section 3.7].

The remainder of the paper is organized as follows. In Section 2, we define the basic type theory. We introduce Noetherian relations and streamless relations in Section 3 and prove that any Noetherian relation is streamless. The realizability model is constructed in Section 4, with realizers for Markov's Principle in Section 5 and for the unsolvability of the halting problem in Section 6. This shows that the type theory can be consistently extended with these two axioms. Then, in Section 7 we show that it cannot be proved in type theory that any streamless set is Noetherian. We conclude with a discussion of related work, in particular the Kleene Tree, in Section 8. For readers already familiar with dependent type theory and inductive bars it might be efficient to read the conclusion first.

2 Dependent Type Theory

We closely follow the approach of Coquand and Spiwack [3]. We define Martin-Löf dependent type theory as a set of typing rules defining a typing relation $\Gamma \vdash M : A$ where M and A are terms in an extension Λ of the untyped lambda calculus, and Γ is a context. A *context* is a sequence $x_1 : A_1, \ldots, x_n : A_n$, where the x_i are pairwise distinct variables and the A_i are terms of Λ (representing types). The approach of [3] makes the construction of a realizability model easier: all typable terms are already terms of Λ realizing their types, and their computational behaviour can be studied in Λ.

An important aspect is that the type theory is open-ended, new constants and inductive definitions can be (and will be) added. If the type theory is extended, then also Λ is extended, and the realizability model is extended accordingly. We start by describing the main characteristics of Λ.

2.1 The Underlying Computational System

The calculus Λ is an extension of the untyped lambda calculus with two sorts of constants, *constructors* and *operators*. Constructors typically represent types (e.g., the type of the natural numbers), type forming constructions (e.g., the sum of two types), and term forming constructions (e.g., $\mathsf{0}$, S, nil). Operators typically represent destructors (e.g., recursors), operations (e.g., the length of a list), and convenient abbreviations.

The abstract syntax for terms of Λ is

$$M, N ::= x \mid \lambda x.\, M \mid M\, N \mid c \mid o,$$

where c is the syntactic category of the constructors and o that of the operators. We write $\mathsf{FV}(M)$ to denote the set of free variables in M; we call M closed if $\mathsf{FV}(M)$ is empty. The computational behavior of the terms is determined by β-reduction plus so-called ι-reduction rules. The latter are left-linear and mutually disjoint (non-overlapping), ensuring confluence of $\beta\iota$-reduction [8]. (Confluence is important to warrant the correct interpretation of elements of an inductive type as $\beta\iota$-equivalence classes of terms. For example, the normal forms $\mathsf{0}$ and $\mathsf{S0}$ are in different classes because of confluence.) All ι-reduction rules are of the form

$$o\, p_1 \ldots p_k = M,$$

where o is an operator and $p_1, \ldots, p_k$ are so-called constructor patterns. For any ι-reduction rule we require $\mathsf{FV}(M) \subseteq \mathsf{FV}(o\, p_1 \ldots p_k)$, that is, no new variables can be introduced. Constructor patterns $p_1, \ldots, p_k$ are defined by the following abstract syntax

$$p ::= x \mid c\, p_1 \ldots p_l,$$

where c is a constructor.

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \quad \Gamma, x : A \vdash B}{\Gamma \vdash \Pi x{:}A.\, B}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash x : A}\; x{:}A \in \Gamma \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B}{\Gamma \vdash M : B}\; A =_{\beta\iota} B$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \Pi x{:}A.\, B \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : \Pi x{:}A.\, B} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.\, B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B(N)}$$

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma, x : A \vdash B : \mathsf{U}}{\Gamma \vdash \Pi x{:}A.\, B : \mathsf{U}}$$

Figure 1 General typing rules of Martin-Löf type theory with universe U.

Constructors, as well as operators with their ι-reduction rules, will be introduced in the sequel, as need arises, always complying with the above syntax.

Notational conventions.

We tacitly assume capture-free substitution and consider terms up to α-conversion. We write $M =_{\beta\iota} N$ or just $M = N$ if M and N are $\beta\iota$-convertible. By $M(x/N)$ we denote the result of substituting N for all the free occurences of the variable x in M. We may write $M(N)$ if the variable x is clear from the context. For example, $(\lambda x.\, M)N = M(x/N)$ and $(\lambda x.\, M)N = M(N)$ both denote a β-step. We abbreviate $(\lambda x.\, xx)(\lambda x.\, xx)$ to Ω.

2.2 General rules of the type theory

There are three forms of judgments in the type theory:

$$\Gamma \vdash \quad \text{and} \quad \Gamma \vdash A \quad \text{and} \quad \Gamma \vdash M : A.$$

The judgment $\Gamma \vdash$ means that Γ is a well-typed context, $\Gamma \vdash A$ means that the type A is well-formed in the context Γ, and $\Gamma \vdash M : A$ means that the term M has the type A in the context Γ. We (mostly) use metavariables A, B for types, and M, N for terms, but recall that they are all terms of Λ.

For the general rules, we have a constructor U for the universe since we want type families to be first-class citizens. We add an operator Pi for dependent products, with ι-reduction $\mathsf{Pi}\, A\, B\, x = B\, x$. For readability, we write $\Pi x{:}A.\, B$ instead of $\mathsf{Pi}\, A\, (\lambda x.\, B)$, and $A \to B$ instead of $\mathsf{Pi}\, A\, (\lambda x.\, B)$ if x does not occur free in B. The typing rules are the standard rules for the Martin-Löf type theory, given in Figure 1.

We can derive, for example, $A : \mathsf{U} \vdash$, and so $\vdash \mathsf{U} \to \mathsf{U}$, and in some more steps $A : \mathsf{U} \vdash A \to \mathsf{U}$. The latter $A \to \mathsf{U}$ is the type of unary predicates on a type A (also called *type families* over A). The former $\mathsf{U} \to \mathsf{U}$ is the type of functions on the universe. Both are *large* types, i.e., types not in U. Types in U are called *small* types.

2.3 Specific rules of the type theory

We extend the type theory by specific inductive types, which are all standard. We add constants and give typing rules, as well as ι-reduction rules for the operators.

Empty type

We define the empty type with no constructors:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{N}_0 : \mathsf{U}}$$

and its elimination rule (also known as the *ex falso* rule):

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{ExF} : \mathsf{N}_0 \to A}$$

We define negation as the abbreviation $\neg := \lambda A.\, A \to \mathsf{N}_0$.

Sum

We have the sum type with its two constructors:

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma \vdash B : \mathsf{U}}{\Gamma \vdash A + B : \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma \vdash A + B : \mathsf{U}}{\Gamma \vdash \mathsf{inl} : A \to A + B} \qquad \frac{\Gamma \vdash B : \mathsf{U} \quad \Gamma \vdash A + B : \mathsf{U}}{\Gamma \vdash \mathsf{inr} : B \to A + B}$$

and may perform case analysis on terms of type $A + B$:

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma \vdash B : \mathsf{U} \quad \Gamma \vdash C : \mathsf{U} \quad \Gamma \vdash (A + B) : \mathsf{U}}{\Gamma \vdash \mathsf{case} : (A \to C) \to (B \to C) \to A + B \to C}$$

where the ι-reductions are given by:

$$\begin{array}{lcl} \mathsf{case}\, M\, N\, (\mathsf{inl}\, a) & = & M\, a \\ \mathsf{case}\, M\, N\, (\mathsf{inr}\, b) & = & N\, b \end{array}$$

With negation and sum type used for constructive disjunction we can define the concept of decidability that will play an important role in the sequel.

▶ **Definition 1.** We call a type $A : \mathsf{U}$ *decidable* if $A : \mathsf{U} \vdash M : A + \neg A$ for some M. The type $A + \neg A$ will often be abbreviated by $\mathsf{dec}\, A$. In such cases we also say that $\mathsf{dec}\, A$ is *inhabited*, without explicit reference to Γ or M. Predicates are called decidable if they are pointwise decidable. For example, $P : A \to \mathsf{U}$ is decidable if $\Pi a{:}A.\, \mathsf{dec}\,(Pa)$ is inhabited; $R : A \to A \to \mathsf{U}$ is decidable if $\Pi a, a'{:}\, A.\, \mathsf{dec}\,(Raa')$ is inhabited. The latter is an example of how we denote two Π-abstractions with the same base type A.

Unit type

We have the unit type with one single constructor:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{N}_1 : \mathsf{U}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathsf{N}_1}$$

Booleans

We have the type for Booleans with two constructors:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{N}_2 : \mathsf{U}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathsf{N}_2} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 1 : \mathsf{N}_2}$$

and a conditional expression:

$$\frac{\Gamma \vdash C : \mathsf{N}_2 \to \mathsf{U}}{\Gamma \vdash \mathsf{brec} : C\,0 \to C\,1 \to \Pi b{:}\mathsf{N}_2.\, C\,b}$$

with the ι-reduction given by

$$\begin{array}{lcl} \mathsf{brec}\, M\, N\, 0 & = & M \\ \mathsf{brec}\, M\, N\, 1 & = & N \end{array}$$

Note that brec does not make it possible to define a function $f : \mathsf{N}_2 \to \mathsf{U}$ with, e.g., $f\,i = \mathsf{N}_i$, since that would require $C = \lambda b.\,\mathsf{U}$, which cannot be typed. Therefore, certain useful operators have to be defined ad-hoc. Here we define a decidable equality for Booleans:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{beq} : \mathsf{N}_2 \to \mathsf{N}_2 \to \mathsf{U}}$$

whose ι-reductions are given by:

$$\begin{array}{llllll} \mathsf{beq} & 0 & 0 & = & \mathsf{N}_1 \\ \mathsf{beq} & 0 & 1 & = & \mathsf{N}_0 \\ \mathsf{beq} & 1 & 0 & = & \mathsf{N}_0 \\ \mathsf{beq} & 1 & 1 & = & \mathsf{N}_1 \end{array}$$

Natural numbers

We have the type N with the two well-known constructors:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{N} : \mathsf{U}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathsf{N}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{S} : \mathsf{N} \to \mathsf{N}}$$

Notice that 0 is ad-hoc polymorphic and is a constructor of $\mathsf{N}_1, \mathsf{N}_2$ and N. We have the recursor (dependent eliminator) rec:

$$\frac{\Gamma \vdash C : \mathsf{N} \to \mathsf{U}}{\Gamma \vdash \mathsf{rec} : C\,0 \to (\Pi n{:}\mathsf{N}.\, C\,n \to C\,(S\,n)) \to \Pi n{:}\mathsf{N}.\, C\,n}$$

with ι-reductions given by

$$\begin{array}{lcl} \mathsf{rec}\, M\, N\, 0 & = & M \\ \mathsf{rec}\, M\, N\, (\mathsf{S}\, n) & = & N\, n\, (\mathsf{rec}\, M\, N\, n). \end{array}$$

We also define a decidable equality on natural numbers:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{eq} : \mathsf{N} \to \mathsf{N} \to \mathsf{U}}$$

whose ι-reductions are given by:

$$\begin{array}{lcclc} \mathsf{eq} & 0 & 0 & = & \mathsf{N}_1 \\ \mathsf{eq} & (\mathsf{S}\, x) & 0 & = & \mathsf{N}_0 \\ \mathsf{eq} & 0 & (\mathsf{S}\, x) & = & \mathsf{N}_0 \\ \mathsf{eq} & (\mathsf{S}\, x) & (\mathsf{S}\, y) & = & \mathsf{eq}\ x\ y. \end{array}$$

By double induction one can easily prove:

▶ **Lemma 2.** *There exist proofs $deq_{\mathsf{N}_2}, deq_{\mathsf{N}}$ such that*

$$\vdash deq_{\mathsf{N}_2} : \Pi x, y{:}\, \mathsf{N}_2.\, \mathsf{dec}\,(\mathsf{beq}\, x\, y) \qquad\qquad \vdash deq_{\mathsf{N}} : \Pi n, m{:}\, \mathsf{N}.\, \mathsf{dec}\,(\mathsf{eq}\, n\, m)$$

Lists

We have the usual type of lists over A, denoted by $[A]$:

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash [A] : \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{nil} : [A]} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{cons} : A \to [A] \to [A]}$$

and the list recursor, writing $a :: l$ for $\mathsf{cons}\, a\, l$ here and below:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash C : [A] \to \mathsf{U}}{\Gamma \vdash \mathsf{lrec} : C\, \mathsf{nil} \to (\Pi a{:}A.\, \Pi l{:}[A].\, C\, l \to C(a :: l)) \to \Pi l{:}[A].\, C\, l}$$

with ι-reductions given by

$$\begin{array}{lcl} \mathsf{lrec}\, M\, N\, \mathsf{nil} & = & M \\ \mathsf{lrec}\, M\, N\, (a :: l) & = & N\, a\, l\, (\mathsf{lrec}\, M\, N\, l). \end{array}$$

Dependent pairs

We have the Σ-type for dependent pairs:

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma, x : A \vdash B : \mathsf{U}}{\Gamma \vdash \Sigma x{:}A.\, B : \mathsf{U}} \qquad \frac{\Gamma \vdash \Sigma x{:}A.\, B : \mathsf{U} \quad \Gamma \vdash W : A \quad \Gamma \vdash P : B(W)}{\Gamma \vdash (W, P) : \Sigma x{:}A.\, B}$$

with dependent eliminator (recursor):

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma, x : A \vdash B : U \quad \Gamma \vdash \Sigma x{:}A.\, B : \mathsf{U} \quad \Gamma \vdash C : (\Sigma x{:}A.\, B) \to \mathsf{U}}{\Gamma \vdash \mathsf{srec} : (\Pi x{:}A.\, \Pi p{:}B.\, C\,(x, p)) \to \Pi y{:}(\Sigma x{:}A.\, B).\, C\, y}$$

with ι-reduction given by:

$$\mathsf{srec}\, Q\, (w, p) \quad = \quad Q\, w\, p$$

We use similar notational conventions for $\Sigma x{:}A.\, B$ as for $\Pi x{:}A.\, B$. This means that the actual syntax is $\mathsf{Sig}\, A\, (\lambda x.\, B)$. When x does not appear free in B, we may abbreviate $\Sigma x{:}A.\, B$ as $A \times B$.

3 Noetherian relations and streamless relations

In this section we define the concepts Noetherian and streamless for relations. When applied to the equality relation on a type A, they yield two classically equivalent definitions of finiteness. We first extend the type theory with new constants and rules to facilitate these definitions.

3.1 Auxiliary constants and rules

Given a list l of elements of a type A and a predicate P on A, we define a predicate $\mathsf{exists}\, P\, l$ to be true if l contains an element that satisfies P. Formally, we add a typing rule for exists

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{exists} : (A \to \mathsf{U}) \to [A] \to \mathsf{U}}$$

and ι-reductions given by:

$$\begin{array}{lcl} \mathsf{exists}\, P\, \mathsf{nil} & = & \mathsf{N}_0 \\ \mathsf{exists}\, P\, (a :: l) & = & (P\, a) + \mathsf{exists}\, P\, l \end{array}$$

Note that exists is not definable by list recursion since it would require $C = \lambda l.\, \mathsf{U}$, which cannot be typed. A similar remark holds for good which we define now.

Given a binary relation R on a type A and a list l over A, we define a predicate $\mathsf{good}\, R\, l$ to be true if l contains elements that are related by R in the same order in which they occur in l. Formally, we define a typing rule for good by

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{good} : (A \to A \to \mathsf{U}) \to [A] \to \mathsf{U}}$$

and ι-reductions by:

$$\begin{array}{lcl} \mathsf{good}\, R\, \mathsf{nil} & = & \mathsf{N}_0 \\ \mathsf{good}\, R\, (a :: l) & = & \mathsf{exists}\, (R\, a)\, l + \mathsf{good}\, R\, l \end{array}$$

The following functions are actually definable by the recursors in Section 2.3. We define the length function on lists:

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{length} : [A] \to \mathsf{N}}$$

with ι-reductions given by:

$$\mathsf{length}\, \mathsf{nil} = 0$$

$$\mathsf{length}\, (h :: t) = \mathsf{S}(\mathsf{length}\, t)$$

Given a function f from natural numbers to a type A, the function $\mathsf{take}\, f$ returns for every natural number n the list consisting of the first n values of f. Formally, we define a typing rule for take by

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{take}\ : (\mathsf{N} \to A) \to \mathsf{N} \to [A]}$$

and, writing $\overline{f}n$ for $(\mathsf{take}\, f\, n)$, ι-reductions:

$$\begin{array}{lcl} \overline{f}0 & = & \mathsf{nil} \\ \overline{f}(S\, n) & = & (f\, n) :: (\overline{f}n). \end{array}$$

Given a type A and a predicate P on $[A]$, we define by induction the predicate $\mathsf{bar}\, A\, P$ of lists over A that are "barred" by P. The classical intuition is that a list is "barred" by P if every extension eventually satisfies P. More precisely, by the inductive definition below, $\mathsf{bar}\, A\, P$ is the smallest predicate which contains P and which is closed under inductive shortening, that is, holds of l whenever it holds of $a :: l$ for all $a : A$. Since the type A will be the same in this section, we will use the abbreviation bAr for $\mathsf{bar}\, A$, where the capital in bAr should remind the reader of the implicit argument A. We add the following typing rules for proving that the list l is barred by P:

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{bAr} : ([A] \to \mathsf{U}) \to [A] \to \mathsf{U}}$$

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma \vdash l : [A] \quad \Gamma \vdash P : [A] \to \mathsf{U} \quad \Gamma \vdash X : P\, l}{\Gamma \vdash \mathsf{base}\, X : \mathsf{bAr}\, P\, l}$$

$$\frac{\Gamma \vdash A : \mathsf{U} \quad \Gamma \vdash l : [A] \quad \Gamma \vdash P : [A] \to \mathsf{U} \quad \Gamma \vdash Y : \Pi a{:}A.\, \mathsf{bAr}\, P\, (a :: l)}{\Gamma \vdash \mathsf{step}\, Y : \mathsf{bAr}\, P\, l}$$

The eliminator for $\mathsf{bAr}\,P$ will be called a bar recursor, but should not be confused with Spector's bar recursor from [13]. The latter is an ingenious combinator of much greater proof theoretic strength than the one here. Our bar recursor has the following type:

$$\frac{\Gamma \vdash A : U \quad \Gamma \vdash P : [A] \to U \quad \Gamma \vdash C : [A] \to U}{\Gamma \vdash \mathsf{barrec} : (\Pi l{:}[A].\, P\,l \to C\,l) \to (\Pi l{:}[A].\, (\Pi a{:}A.\, C\,(a :: l)) \to C\,l) \to \Pi l{:}[A].\, \mathsf{bAr}\,P\,l \to C\,l}$$

and its computational behaviour is described by the following ι-reductions:

$$\begin{aligned} \mathsf{barrec}\,B\,S\,l\,(\mathsf{base}\,X) &= B\,l\,X \\ \mathsf{barrec}\,B\,S\,l\,(\mathsf{step}\,Y) &= S\,l\,(\lambda a.\,\mathsf{barrec}\,B\,S\,(a :: l)\,(Y\,a)). \end{aligned}$$

3.2 Definition of streamless and Noetherian

Streamless and Noetherian can both be defined as properties of a binary relation $R : A \to A \to \mathsf{U}$. Informally speaking, R is streamless if every stream, i.e., infinite sequence in A, contains two elements related by R in the reversed order as they appear in the sequence.[1]

▶ **Definition 3.** A relation $R : A \to A \to \mathsf{U}$ on a type $A : \mathsf{U}$ is *streamless* if every function $f : \mathsf{N} \to A$ from natural numbers to A has a prefix that is R-good, i.e.,

$$\mathsf{streamless}\,R := \Pi f{:}\mathsf{N} \to A.\, \Sigma n{:}\mathsf{N}.\, \mathsf{good}\,R\,(\overline{f}n)$$

The equality relation on A being streamless expresses that every stream contains a duplicate, i.e., is constructively non-injective. This is classically equivalent to saying that A is finite.

Noetherian is not so easily explained, it is an acquired taste. We call P an *inductive bar* in $[A]$ if $\mathsf{bAr}\,P\,\mathsf{nil}$ holds, that is, if nil is barred by P. By the inductive definition of $\mathsf{bAr}\,P$ this means that nil can be obtained from lists satisfying P by inductive shortening. We call a relation R Noetherian if $(\mathsf{good}\,R)$ is an inductive bar.

▶ **Definition 4.** A relation $R : A \to A \to \mathsf{U}$ on a type $A : \mathsf{U}$ is *Noetherian* if $\mathsf{bAr}\,(\mathsf{good}\,R)\,\mathsf{nil}$ holds:

$$\mathsf{Noetherian}\,R := \mathsf{bAr}\,(\mathsf{good}\,R)\,\mathsf{nil}$$

The equality relation on A being Noetherian is also classically equivalent to A being finite. However, unlike streamless, Noetherian allows us to use induction on bAr. It is exactly this which allows us to prove that Noetherian relations are streamless. The novelty of this proof is that it does not use a relation $l \prec f$ of a list being a prefix of a function, and hence not an equality relation on the type A. Of course, adding equality would not be problematic, but is it somehow pleasing that equality is not used for proving a result that does not involve equality (the normal form of any such proof would not involve equality anyway). A nice corollary of the next theorem is that every Noetherian relation is reflexive, a fact that would otherwise require a non-trivial proof.

▶ **Theorem 5.** *There is a proof M such that*

$$A : \mathsf{U}, R : A \to A \to \mathsf{U} \vdash M : \mathsf{Noetherian}\,R \to \mathsf{streamless}\,R$$

[1] The name *streamless* may well be considered a misnomer if R is not an equality relation. Classically, streamless means that there is no f such that $i > j \to \overline{R}ij$ for all i, j ($\overline{R}$ the complement of R).

Proof. We prove in the context $A : \mathsf{U}, R : A \to A \to \mathsf{U}, f : \mathsf{N} \to A$ that the following type is inhabited:

$$\Pi l{:}[A].\,\mathsf{bAr}\,(\mathsf{good}\,R)\,l \to \mathsf{subG}\,R\,f\,l \to \mathsf{subEx}\,R\,f\,l \to \Sigma m{:}\mathsf{N}.\,\mathsf{good}\,R\,(\overline{f}m), \tag{1}$$

where $\mathsf{subG}\,R\,f\,l$ and $\mathsf{subEx}\,R\,f\,l$ are defined as the following abbreviations.

$$\begin{aligned} \mathsf{subG}\,R\,f\,l &:= \mathsf{good}\,R\,l \to \mathsf{good}\,R\,(\overline{f}(\mathsf{length}\,l)) \\ \mathsf{subEx}\,R\,f\,l &:= \Pi a{:}A.\,(\mathsf{exists}\,(R\,a)\,l \to \mathsf{exists}\,(R\,a)\,(\overline{f}(\mathsf{length}\,l))) \end{aligned}$$

These abbreviations express that the good/exists properties of l imply those of $\overline{f}(\mathsf{length}\,l)$, which will suffice to show (1) . Also note that they both trivially hold for nil, so that (1) with $l = \mathsf{nil}$ implies Theorem 5. (Note that both are trivially implied by $l = \overline{f}(\mathsf{length}\,l)$.)

We prove (1) by induction on $\mathsf{bAr}\,(\mathsf{good}\,R)\,l$, that is, using the last rule of the previous section with the predicate $P := (\mathsf{good}\,R)$ and the predicate

$$C := \lambda l{:}[A].\,\mathsf{subG}\,R\,f\,l \to \mathsf{subEx}\,R\,f\,l \to \Sigma m{:}\mathsf{N}.\,\mathsf{good}\,R\,(\overline{f}m).$$

Thus the proof of (1) will be of the form $\mathsf{barrec}\,H_b\,H_s$ with $H_b : \Pi l{:}[A].\,P\,l \to C\,l$ and

$$H_s : \Pi l{:}[A].\,(\Pi a{:}A.\,C(a :: l)) \to Cl,$$

corresponding to the base case and the step case, respectively, elaborated below.

Base case. To construct $H_b : \Pi l{:}[A].\,P\,l \to C\,l$, assume $l : [A]$ such that $\mathsf{good}\,R\,l$, $\mathsf{subG}\,R\,f\,l$, and $\mathsf{subEx}\,R\,f\,l$. From $\mathsf{subG}\,R\,f\,l$ and $\mathsf{good}\,R\,l$ we immediately get the goal $\mathsf{good}\,R\,(\overline{f}(\mathsf{length}\,l))$.

Step case. To construct $H_s : \Pi l{:}[A].\,(\Pi a{:}A.\,C(a :: l)) \to Cl$, assume $l : [A]$ such that $\Pi a{:}A.\,C(a :: l)$. We have to show $C\,l$. The latter expands to $\mathsf{subG}\,R\,f\,l \to \mathsf{subEx}\,R\,f\,l \to \Sigma m{:}\mathsf{N}.\,\mathsf{good}\,R\,(\overline{f}m)$, so we assume $\mathsf{subG}\,R\,f\,l$ and $\mathsf{subEx}\,R\,f\,l$, and show $\Sigma m{:}\mathsf{N}.\,\mathsf{good}\,R\,(\overline{f}m)$. Expanding the induction hypothesis $(\Pi a{:}A.\,C(a :: l))$ yields

$$\Pi a{:}A.\,\mathsf{subG}\,R\,f\,(a :: l) \to \mathsf{subEx}\,R\,f\,(a :: l) \to \Sigma m{:}\mathsf{N}.\,\mathsf{good}\,R\,(\overline{f}m).$$

We apply this to $a = f(\mathsf{length}\,l)$, and proceed to proving the two assumptions in the following two subcases.

Subcase $\mathsf{subG}\,R\,f\,(f(\mathsf{length}\,l) :: l)$. Expanding the abbreviation subG we get

$$\mathsf{good}\,R\,(f(\mathsf{length}\,l) :: l) \to \mathsf{good}\,R\,(\overline{f}(\mathsf{length}\,(f(\mathsf{length}\,l) :: l))).$$

Since $(\mathsf{length}\,(f(\mathsf{length}\,l) :: l))$ reduces to $\mathsf{S}(\mathsf{length}\,l)$, the conclusion of the above formula reduces to $\mathsf{good}\,R\,(f(\mathsf{length}\,l) :: \overline{f}(\mathsf{length}\,l))$. Using the definition of good in both the antecedent and the consequent it becomes clear that we have to prove:

$$\begin{aligned} &(\mathsf{exists}\,(R\,(f\,(\mathsf{length}\,l)))\,l + \mathsf{good}\,R\,l) \to \\ &(\mathsf{exists}\,(R\,(f(\mathsf{length}\,l)))\,(\overline{f}(\mathsf{length}\,l)) + \mathsf{good}\,R\,(\overline{f}(\mathsf{length}\,l))). \end{aligned}$$

The latter is easily proved by cases using the assumption $\mathsf{subEx}\,R\,f\,l$ with $a = f(\mathsf{length}\,l)$ for the left summand and the assumption $\mathsf{subG}\,R\,f\,l$ for the right summand.

Subcase $\mathsf{subEx}\,R\,f\,(f(\mathsf{length}\,l) :: l)$. Expanding the abbreviation subEx and reducing we get

$$\begin{aligned} &\Pi a{:}A.\ \mathsf{exists}\,(R\,a)\,(f(\mathsf{length}\,l) :: l) \to \\ &\qquad\qquad \mathsf{exists}\,(R\,a)\,(f(\mathsf{length}\,l) :: \overline{f}(\mathsf{length}\,l)). \end{aligned}$$

Using the definition of exists we get by reducing

$$\Pi a{:}A.\ ((R\,a\,(f(\mathsf{length}\,l))) + \mathsf{exists}\,(R\,a)\,l) \to \\ ((R\,a\,(f(\mathsf{length}\,l))) + \mathsf{exists}\,(R\,a)\,(\overline{f}(\mathsf{length}\,l))).$$

The latter follows easily from the assumption $\mathsf{subEx}\,R\,f\,l$. This finishes the second subcase of the step case and we are done. ◀

4 The Model Construction

In this section, we construct the realizability model for the type theory, based on the underlying computational system Λ. Terms are interpreted by $\beta\iota$-equivalence classes of the terms of Λ. Types are interpreted by sets of such equivalence classes. Typically, if $\Gamma \vdash M : A$, then the interpretation of M is an element of the interpretation of A (both relative to the interpretation of Γ).

We use the realizability model to show the unprovability of the converse of Theorem 5. For this result, we use (the functional version of) Markov's principle and a non-classical axiom $\neg\Pi n{:}\mathsf{N}.\,(H n) + \neg(H n)$ for a halting predicate H. Both will be shown to be true in the model in later sections.

4.1 Pointed DCPOs and fixpoints

We recall Pataraia's fixpoint theorem [12], which states that every monotone endofunction[2] on a pointed directed complete partial order (DCPO) has a least fixpoint.

▶ **Definition 6.** Let $(P, \leq)$ be a partial order. A subset X of P is *directed* if it is nonempty and, for every $x, y \in X$, there exists $z \in X$ such that $x \leq z$ and $y \leq z$.

▶ **Definition 7.** A partial order $(P, \leq)$ is a *directed complete partial order* (*DCPO*) if every directed subset of P has a least upper bound (supremum) in P. A DCPO $(P, \leq)$ is *pointed* if the empty set has a supremum, which is then the least element $\perp_P$ of P.

▶ **Definition 8.** An endofunction $f : P \to P$ is *monotone* if it is order-preserving, i.e., for every $x, y \in P$, $x \leq y$ implies $f(x) \leq f(y)$.

▶ **Theorem 9.** *Every monotone endofunction function on a pointed DCPO has a least fixpoint, which is also the least pre-fixpoint.*

A short proof can be found in [6]. The standard argument, transfinite iteration of the function starting at the least element, also works. The reason is that the transfinite sequence is directed.

We will use Pataraia's Theorem with the following DCPO. Let D be a set. Elements of the DCPO are pairs (S, F) where $S \in \mathcal{P}(D)$, that is, S is a subset of D, and F is a function $S \to \mathcal{P}(D)$, viewed as a single-valued set of pairs. We can order such pairs by $(S, F) \leq (S', F')$ if $S \subseteq S'$ and $F \subseteq F'$. The latter conjunct can be rephrased by saying that F is the restriction of F' to S. This does *not* yield a complete lattice on pairs (S, F), even though $(\mathcal{P}(D), \subseteq)$ is. For example, if $D = \{d, e\}$ and $S = \{d\}$ and F_d maps d to $\{d\}$ and F_e maps d to $\{e\}$, then there is no F such that $F_d, F_e \subseteq F$. (It is tempting but wrong to think that F mapping d to $\{d, e\}$ extends F_d and F_e.)

[2] An *endofunction* is a function with the same domain and codomain.

However, if X is a directed set of pairs (S, F), then the pair

$$\Big(\bigcup_{(S,F)\in X} S \,,\ \bigcup_{(S,F)\in X} F \Big)$$

is the least upper bound of X. Note that, since X is directed, $\bigcup_{(S,F)\in X} F$ is a function from $\bigcup_{(S,F)\in X} S$ to $\mathcal{P}(D)$ as required. If $X = \emptyset$, we get the least element $(\emptyset, \emptyset)$ by the same formula above. It follows that the set of pairs ordered as above is a pointed DCPO. Consequently, every monotone endofunction has a least fixpoint.

4.2 Realizability model

We shall now define the realizability model. The *domain* D is the set of terms in the extended untyped lambda calculus modulo $\beta\iota$-equality. Hence, elements of D are equivalence classes of terms. For simplicity, however, we will often call them just terms, and write M to denote the equivalence class of M. Next, we define which elements of D represent types and how to find the subset of elements associated with each type.

We shall first prepare the interpretation of the inductive types. Define $\mathsf{Num} \subseteq \mathsf{D}$ as the smallest set containing $\mathsf{0}$ and closed under the successor, i.e., $\mathsf{S}\, n$ is in Num if n is in Num. Formally, we define Num as the least fixpoint of the following monotone endofunction Ψ_{Num} on $\mathcal{P}(\mathsf{D})$:

$$\Psi_{\mathsf{Num}}(X) := \{\mathsf{0}\} \cup \{\mathsf{S}\, n \mid n \in X\}.$$

The poset $(\mathcal{P}(\mathsf{D}), \subseteq)$ is a complete lattice, hence the least fixpoint exists by the Knaster-Tarski theorem, which is a special case of Pataraia's theorem.

Similarly, given a set $A \subseteq \mathsf{D}$, we define $\mathsf{List}(A) \subseteq \mathsf{D}$ as the least fixpoint of the following monotone endofunction $\Psi_{\mathsf{List}(A)}$ on $(\mathcal{P}(\mathsf{D}) \subseteq)$:

$$\Psi_{\mathsf{List}(A)}(X) := \{\mathsf{nil}\} \cup \{\mathsf{cons}\, a\, l \mid a \in A \wedge l \in X\}.$$

Informally, $\mathsf{List}(A)$ consists of classes that are $\beta\iota$-equivalent to lists over A.

Note that bAr is an inductively defined function on $[A] \to \mathsf{U}$. Given a set $A \subseteq \mathsf{D}$, consider the poset $(\mathsf{List}(A) \to \mathcal{P}(\mathsf{D}), \leq)$, where $Q \leq Q'$ if $Q(l) \subseteq Q'(l)$ for all l in $\mathsf{List}(A)$[3]. This poset forms a complete lattice, hence every monotone function on it has a least fixpoint. Given also a function P in $\mathsf{List}(A) \to \mathcal{P}(\mathsf{D})$, define $\mathsf{Bar}(A, P)$ as the least fixpoint of the following monotone endofunction $\Psi_{\mathsf{Bar}(A,P)}$ on $(\mathsf{List}(A) \to \mathcal{P}(\mathsf{D}), \leq)$:

$$\Psi_{\mathsf{Bar}(A,P)}(Q)(l) := \{\mathsf{base}\, X \mid X \in P(l)\} \cup \{\mathsf{step}\, Y \mid \forall a \in A.\, Y\, a \in Q\,(\mathsf{cons}\, a\, l)\}.$$

Finally, we introduce the DCPO $\mathcal{L}$ of pairs (S, F) with $S \subseteq \mathsf{D}$ and $F \in S \to \mathcal{P}(\mathsf{D})$ as described in Section 4.1. Here S is to be viewed as a set of types, and F as a function giving the set of elements $F(T) \subseteq \mathsf{D}$ for each $T \in S$. We are now ready to define which terms of Λ are types, and which elements each type has. We do so in two stages, first for the small types and then for all types, using monotone endofunctions Φ_0 and Φ_1 on $\mathcal{L}$, respectively.

An important observation is that only constructors play a role here, not operators, and that the only difference between Φ_0 and Φ_1 is that the latter includes the constructor U.

[3] Abusing notation, we denote by $A \to B$ the set of functions from the set A to the set B in the ambient naive set theory in which we develop the realizability model.

We define (T_0, El_0) in $\mathcal{L}$ to be the least fixpoint of the following monotone endofunction Φ_0 on $\mathcal{L}$:

$$\Phi_0(S, F) := (S_1 \cup S_2 \cup \cdots \cup S_9, F_1 \cup F_2 \cup \cdots \cup F_9)$$

where

$$\begin{array}{lcl}
S_1 & = & \{\mathsf{N}_0\} \\
F_1 & = & \{(\mathsf{N}_0, \emptyset)\} \\
S_2 & = & \{\mathsf{N}_1\} \\
F_2 & = & \{(\mathsf{N}_1, \{\mathsf{0}\})\} \\
S_3 & = & \{\mathsf{N}_2\} \\
F_3 & = & \{(\mathsf{N}_2, \{\mathsf{0}, \mathsf{1}\})\} \\
S_4 & = & \{\mathsf{N}\} \\
F_4 & = & \{(\mathsf{N}, \mathsf{Num})\} \\
S_5 & = & \{A + B \mid A, B \in S\} \\
F_5 & = & \{(A + B, \{\mathsf{inl}\, a \mid a \in F(A)\} \cup \{\mathsf{inr}\, b \mid b \in F(B)\}) \mid A, B \in S\} \\
S_6 & = & \{[A] \mid A \in S\} \\
F_6 & = & \{([A], \mathsf{List}(F(A))) \mid A \in S\} \\
S_7 & = & \{\Pi x{:}A.\, B \mid A \in S \wedge \forall a \in F(A).\, B(a) \in S\} \\
F_7 & = & \{(\Pi x{:}A.\, B, \{M \mid \forall a \in F(A).\, M\, a \in F(B(a))\}) \mid \Pi x{:}A.\, B \in S_7\} \\
S_8 & = & \{\Sigma x{:}A.\, B \mid A \in S \wedge \forall a \in F(A).\, B(a) \in S\} \\
F_8 & = & \{(\Sigma x{:}A.\, B, \{(a, M) \mid a \in F(A) \wedge M \in F(B(a))\}) \mid \Sigma x{:}A.\, B \in S_8\} \\
S_9 & = & \{\mathsf{bAr}\, P\, l \mid A \in S \wedge l \in \mathsf{List}(F(A)) \wedge \forall l' \in \mathsf{List}(F(A)).\, P\, l' \in S\} \\
F_9 & = & \{(\mathsf{bAr}\, P\, l, \mathsf{Bar}(F(A), \mathsf{List}(F(A)) \ni l' \mapsto F(P\, l'))(l)) \mid \mathsf{bAr}\, P\, l \in S_9\}
\end{array}$$

In the last line, the notation $\mathsf{List}(F(A)) \ni l' \mapsto F(P\, l')$ denotes the function which maps l' in $\mathsf{List}(F(A))$ to $F(P\, l')$.

The endofunction Φ_0 is monotone on the DCPO $\mathcal{L}$, hence the least fixpoint (T_0, El_0) exists by Theorem 9.

▶ **Definition 10.** (T_0, El_0) is the least fixpoint of Φ_0 above.

Given T_0, we define

$$\Phi_1(S, F) := (\{\mathsf{U}\} \cup S_1 \cup S_2 \cup \cdots \cup S_9, \{(\mathsf{U}, T_0)\} \cup F_1 \cup F_2 \cup \cdots \cup F_9)$$

with S_i, F_i for $i = 1, \ldots, 9$ are as defined earlier. The endofunction Φ_1 is monotone.

▶ **Definition 11.** (T_1, El_1) is the least fixpoint of Φ_1 above.

▶ Remark. An important observation about the definitions of Φ_0 and Φ_1 is that F occurs negatively in some clause, namely in S_7 for dependent products. This is the reason that we use DCPOs and not CPOs. Type-theoretically, the construction of the model goes by induction-recursion [5], as opposed to mutual induction. A different device, replacing negative occurrences by positive conditions on the complements, has been used in [1] and can be traced back to Scott and Feferman.

▶ Remark. The set T_1 is intended to contain all representatives of types in the type theory, both small and large, but it actually contains much more. Likewise, for $A \in T_1$, $El_1(A)$ intends to contain all interpretations of terms of type A in the type theory, but it actually contains much more. For instance, the set $El_1(\mathsf{N} \to \mathsf{N})$ contains all (total) recursive functions on Num. In particular, elements in T_1 or $El_1(A)$ (with $A \in T_1$) may not be normalizing. For instance, let

$$f := \lambda n.\, \mathsf{rec}\, 0\, (\lambda m.\, \lambda x.\, 0)\, n$$

and, deliberately using the term Ω which is not typable,

$$g := \lambda n.\, \mathsf{rec}\, 0\, \Omega\, (f\, n).$$

Then f always returns $\mathsf{0}$ on a numeral, i.e., $f\, n$ is $\beta\iota$-equivalent to $\mathsf{0}$ for any n in Num, so that also $gn =_{\beta\iota} \mathsf{0}$ for all numerals n. The term g is in $El_1(\mathsf{N} \to \mathsf{N})$, but is not even weakly normalizing, since g reduces to itself by a contraction of Ω. Likewise, for the term

$$h := \lambda n.\, (\mathsf{rec}\, \mathsf{N}\, \Omega\, (f\, n)$$

we have $hn =_{\beta\iota} \mathsf{N}$ for all numerals n. Hence $\mathsf{Pi}\, \mathsf{N}\, h$ is in T_0, but is not weakly normalizing. We have $f, g \in El_0(\mathsf{Pi}\, \mathsf{N}\, h) = El_0(\mathsf{N} \to \mathsf{N})$. While we have $\vdash f : \mathsf{N} \to \mathsf{N}$, it is neither possible to derive $\vdash g : \mathsf{N} \to \mathsf{N}$, nor $\vdash \mathsf{Pi}\, \mathsf{N}\, h$, let alone $\vdash f : \mathsf{Pi}\, \mathsf{N}\, h$ or $\vdash g : \mathsf{Pi}\, \mathsf{N}\, h$. The realizability model is not a term model. This is not a bug, but a feature that we will exploit: types that are inhabited in the model, can be consistently added as axioms to the type theory.

The following lemma states that El_0 and El_1 agree on T_0.

▶ **Lemma 12.** $(T_0, El_0) \leq (T_1, El_1)$.

Proof. The claim follows from Theorem 9, since (T_1, El_1) is a prefixpoint of Φ_0. ◀

From the fact that (T_1, El_1) (resp. (T_0, El_0)) is a fixpoint of Φ_1 (resp. Φ_0), we obtain the following lemma.

▶ **Lemma 13.** *For $i = 0, 1$, the following conditions hold:*

1) $\mathsf{N}_0 \in T_i$, *and* $El_i(\mathsf{N}_0) = \emptyset$*;*

2) $\mathsf{N}_1 \in T_i$, *and* $El_i(\mathsf{N}_1) = \{\mathsf{0}\}$*;*

3) $\mathsf{N}_2 \in T_i$, *and* $El_i(\mathsf{N}_2) = \{\mathsf{0}, \mathsf{1}\}$*;*

4) $\mathsf{N} \in T_i$, *and* $El_i(\mathsf{N}) = \mathsf{Num}$*;*

5) $A + B \in T_i$ *if* $A, B \in T_i$, *and then*

$$El_i(A + B) = \{\mathsf{inl}\, a \mid a \in El_i(A)\} \cup \{\mathsf{inr}\, b \mid b \in El_i(B)\}\,;$$

6) $[A] \in T_i$ *if* $A \in T_0$, *and then* $El_i([A]) = \mathsf{List}(El_0(A))$*;*

7) $\Pi x{:}A.\, B \in T_i$ *if* $A \in T_i$ *and* $B(a) \in T_i$ *for all* $a \in El_i(A)$, *and then*

$$El_i(\Pi x{:}A.\, B) = \{M \mid \forall a \in El_i(A), M\, a \in El_i(B(a))\}\,;$$

8) $\Sigma x{:}A.\, B \in T_i$ *if* $A \in T_i$ *and* $B(a) \in T_i$ *for all* $a \in El_i(A)$, *and then*

$$El_i(\Sigma x{:}A.\, B) = \{(W, P) \mid W \in El_i(A) \wedge P \in El_i(B(W))\}\,;$$

9) $\mathsf{bAr}\, P\, l \in T_i$ *if* $A \in T_0$, $l \in El_0([A])$ *and* $P\, l' \in T_0$ *for all* $l' \in El_0([A])$, *and then*

$$El_i(\mathsf{bAr}\, P\, l) = \mathsf{Bar}(El_0(A), El_0([A]) \ni l' \mapsto El_0(P\, l'))(l);$$

10) $\mathsf{U} \in T_1$, *and* $El_1(\mathsf{U}) = T_0$ *(but not* $\mathsf{U} \in T_0$*).*

4.3 Soundness

We now give the semantics of expressions and types a priori, that is, without any assumption of them being well-typed.

▶ **Definition 14.** An *environment* is a mapping from the set of variables to the domain D. We let $\rho, \rho', \ldots$ range over environments and let Env denote the set of all environments. By $\rho(x/a)$ we denote the environment ρ' with $\rho'(x) = a$ and $\rho'(y) = \rho(x)$ for variables $y \neq x$.

▶ **Definition 15.** Let M be a term, i.e., either an expression or a type, and ρ an environment. The *semantics* $[\![M]\!]_\rho \in \mathsf{D}$ of M in ρ denotes the ($\beta\iota$-equivalence class of the) result of the simultaneous substitution in M of all free occurrences of variables x by $\rho(x)$.

We write $[\![M]\!]$ to denote $[\![M]\!]_\rho$ with ρ being the empty environment, or when M is closed. We may also write M for $[\![M]\!]_\rho$ in that case.

As usual, we need a substitution lemma.

▶ **Lemma 16.** *For all M, N and ρ, we have $[\![(\lambda x.\, M)N]\!]_\rho = [\![M(N)]\!]_\rho = [\![M]\!]_{\rho(x/[\![N]\!]_\rho)}$.*

Proof. By a routine induction on the structure of M. ◀

We have to take into account certain sanity conditions on environments with respect to typing contexts.

▶ **Definition 17.** An environment ρ is Γ-*correct* if for all $(x : A) \in \Gamma$, $[\![A]\!]_\rho \in T_1$ and $\rho(x) \in El_1([\![A]\!]_\rho)$.

The following lemma states that the type theory is sound with respect to the semantics.

▶ **Lemma 18.** *For all Γ, M, A and for any Γ-correct ρ we have the following:*
1. *if $\Gamma \vdash A$, then $[\![A]\!]_\rho \in T_1$;*
2. *if $\Gamma \vdash M : A$, then $[\![A]\!]_\rho \in T_1$ and $[\![M]\!]_\rho \in El_1([\![A]\!]_\rho)$.*

Proof. Since the rules in Figure 1 mix (1) and (2), we prove the lemma by simultaneous induction on derivations. We start with the general typing rules in Figure 1.

Suppose $\Gamma \vdash \mathsf{U}$ by $\Gamma \vdash$, and ρ is Γ-correct. Then, the claim holds trivially since $[\![\mathsf{U}]\!]_\rho = \mathsf{U} \in T_1$ by Lemma 13.

Suppose $\Gamma \vdash A$ by $\Gamma \vdash A : \mathsf{U}$, and ρ is Γ-correct. We have $[\![A]\!]_\rho \in El_1(\mathsf{U})$ by induction hypothesis and $El_1(\mathsf{U}) = T_0 \subseteq T_1$ by Lemma 12, from which the claim follows.

Suppose $\Gamma \vdash \Pi x{:}A.\, B$ by $\Gamma \vdash A$ and $\Gamma, x : A \vdash B$, and ρ is Γ-correct. We have to prove that $[\![\Pi x{:}A.\, B]\!]_\rho \in T_1$. By induction hypothesis on $\Gamma \vdash A$, we have $[\![A]\!]_\rho \in T_1$. By Lemma 13 and an appeal to the Substitution Lemma, it suffices that $[\![B]\!]_{\rho(x/a)} \in T_1$ for all $a \in El_1([\![A]\!]_\rho)$. For this, it suffices by induction hypothesis on $\Gamma, x : A \vdash B$ that $\rho(x/a)$ is $(\Gamma, x : A)$-correct, which follows from $a \in El_1([\![A]\!]_\rho)$.

Suppose $\Gamma \vdash x : A$ by $(x : A) \in \Gamma$. Then the claim follows by that ρ is Γ-correct and $[\![x]\!]_\rho = \rho(x)$.

Suppose $\Gamma \vdash M : B$ by $A =_{\beta\iota} B$, $\Gamma \vdash M : A$ and $\Gamma \vdash B$. By induction hypothesis on $\Gamma \vdash M : A$, we have $[\![A]\!]_\rho \in T_1$ and $[\![M]\!]_\rho \in El_1([\![A]\!]_\rho)$. By $A =_{\beta\iota} B$, we have $[\![A]\!]_\rho = [\![B]\!]_\rho$, hence we get $[\![B]\!]_\rho \in T_1$ and $[\![M]\!]_\rho \in El_1([\![B]\!]_\rho)$[4]

[4] We do not use the induction hypothesis on $\Gamma \vdash B$. In fact, we do not use the hypothesis $\Gamma \vdash B$. This manifests that the typing rules are more restrictive than the semantics.

Suppose $\Gamma \vdash \lambda x.\, M : \Pi x{:}A.\, B$ by $\Gamma \vdash A$, and $\Gamma \vdash \Pi x{:}A.\, B$ and $\Gamma, x : A \vdash M : B$. By induction hypothesis, we have $[\![A]\!]_\rho \in T_1$ and $[\![\Pi x{:}A.\, B]\!]_\rho \in T_1$. We have to show $[\![\lambda x.\, M]\!]_\rho \in El_1([\![\Pi x{:}A.\, B]\!]_\rho)$. By the Substitution Lemma, it suffices that $[\![\lambda x.\, M]\!]_\rho\, a = [\![M]\!]_{\rho(x/a)} \in El_1([\![B]\!]_{\rho(x/a)})$ for all $a \in El_1([\![A]\!]_\rho)$. This follows from induction hypothesis on $\Gamma, x : A \vdash M : B$, noting that $\rho(x/a)$ is $(\Gamma, x : A)$-correct.

Suppose $\Gamma \vdash M\, N : B(N)$ by $\Gamma \vdash M : \Pi x{:}A.\, B$ and $\Gamma \vdash N : A$. By induction hypothesis, we have $[\![\Pi x{:}A.\, B]\!]_\rho \in T_1$, $[\![M]\!]_\rho \in El_1([\![\Pi x{:}A.\, B]\!]_\rho)$, and $[\![A]\!]_\rho \in T_1$, $[\![N]\!]_\rho \in El_1([\![A]\!]_\rho)$. By Lemma 13, it follows that $[\![M]\!]_\rho\, [\![N]\!]_\rho \in El_1([\![B]\!]_\rho([\![N]\!]_\rho))$. Using the Substitution Lemma, we conclude $[\![M\, N]\!]_\rho = [\![M]\!]_\rho\, [\![N]\!]_\rho \in El_1([\![B]\!]_\rho([\![N]\!]_\rho)) = El_1([\![B]\!]_{\rho(x/[\![N]\!]_\rho)}) = El_1([\![B(N)]\!]_\rho)$.

Suppose $\Gamma \vdash \Pi x{:}A.\, B : \mathsf{U}$ by $\Gamma \vdash A : \mathsf{U}$ and $\Gamma, x : A \vdash B : \mathsf{U}$. By induction hypothesis on $\Gamma \vdash A : \mathsf{U}$ and by Lemma 13, we know that $[\![\mathsf{U}]\!]_\rho = \mathsf{U} \in T_1$, and $[\![A]\!]_\rho \in El_1(\mathsf{U}) = T_0$. We have to show $[\![\Pi x{:}A.\, B]\!]_\rho \in T_0$. By Lemma 13 again and by the Substitution Lemma, it suffices that $[\![B]\!]_{\rho(x/a)} \in T_0$ for all $a \in El_0([\![A]\!]_\rho)$. Recalling that El_0 and El_1 agree on T_0 (Lemma 12), hence in particular on $[\![A]\!]_\rho$, this follows from induction hypothesis on $\Gamma, x : A \vdash B : \mathsf{U}$ since $\rho(x/a)$ is $(\Gamma, x : A)$-correct.

We are done with the general typing rules. We move on to the specific typing rules in Section 2.3. In the following, we will often tacitly use that $El_1(\mathsf{U}) = T_0 \subseteq T_1$ and that El_1 extends El_0 (Lemma 12).

Regarding the empty type, we have $\mathsf{U} \in T_1$ and $\mathsf{N}_0 \in El_1(\mathsf{U})$ from Lemma 13. If $\Gamma \vdash \mathsf{ExF} : \mathsf{N}_0 \to A$ by $\Gamma \vdash A : \mathsf{U}$, then by induction hypothesis, we have $[\![A]\!]_\rho \in T_0$. It follows that $\mathsf{N}_0 \to [\![A]\!]_\rho \in T_0$, and $\mathsf{ExF} \in El_0(\mathsf{N}_0 \to [\![A]\!]_\rho)$ since $El_0(\mathsf{N}_0)$ is empty.

Regarding the typing rules for the unit type, we get the claim from Lemma 13.

Regarding Booleans, we get the claim from Lemma 13 for the typing rules for N_2, $\mathsf{0}$ and $\mathsf{1}$. Suppose $\Gamma \vdash \mathsf{brec} : C\,\mathsf{0} \to C\,\mathsf{1} \to \Pi b{:}\mathsf{N}_2.\, C\, b$ by $\Gamma \vdash C : \mathsf{N}_2 \to \mathsf{U}$. By induction hypothesis, we have $\mathsf{N}_2 \to \mathsf{U} \in T_1$ and $[\![C]\!]_\rho \in El_1(\mathsf{N}_2 \to \mathsf{U})$, so $[\![C\,\mathsf{0}]\!]_\rho \in T_0$, $[\![C\,\mathsf{1}]\!]_\rho \in T_0$, and $[\![C\, b]\!]_\rho \in T_0$ for all $b \in El_1(\mathsf{N}_2)$. It follows that $[\![C\,\mathsf{0} \to C\,\mathsf{1} \to \Pi b{:}\mathsf{N}_2.\, C\, b]\!]_\rho \in T_1$. Let $M \in El_0([\![C\,\mathsf{0}]\!]_\rho)$ and $N \in El_0([\![C\,\mathsf{1}]\!]_\rho)$. For every $b \in El_1(\mathsf{N}_2)$, we have either $b = \mathsf{0}$ or $b = \mathsf{1}$. Hence we get $\mathsf{brec} \in El_1([\![C\,\mathsf{0} \to C\,\mathsf{1} \to \Pi b{:}\mathsf{N}_2.\, C\, b]\!]_\rho)$ from the ι-reduction for brec. We get the claim for the typing rule for beq from Lemma 13 and the ι-reduction for beq.

Regarding natural numbers, the only non-trivial rules are those for rec and eq. Suppose $\Gamma \vdash \mathsf{rec} : C\,\mathsf{0} \to (\Pi n{:}\mathsf{N}.\, C\, n \to C\,(S\, n)) \to \Pi n{:}\mathsf{N}.\, C\, n$ by $\Gamma \vdash C : \mathsf{N} \to \mathsf{U}$. By induction hypothesis, we have $\mathsf{N} \to \mathsf{U} \in T_1$ and $[\![C]\!]_\rho \in El_1(\mathsf{N} \to \mathsf{U})$. It follows that $[\![C\,\mathsf{0}]\!]_\rho \in T_0$, $[\![\Pi n{:}\mathsf{N}.\, C\, n \to C\,(S\, n)]\!]_\rho \in T_0$ and $[\![\Pi n{:}\mathsf{N}.\, C\, n]\!]_\rho \in T_0$, hence $[\![C\,\mathsf{0} \to (\Pi n{:}\mathsf{N}.\, C\, n \to C\,(S\, n)) \to \Pi n{:}\mathsf{N}.\, C\, n]\!]_\rho \in T_0$. We have to prove that $\mathsf{rec}\, M\, N\, n \in El_0([\![C\, n]\!]_\rho)$ for all $M \in El_0([\![C\,\mathsf{0}]\!]_\rho)$, $N \in El_0([\![\Pi n{:}\mathsf{N}.\, Cn \to C(Sn)]\!]_\rho)$ and $n \in El_0(\mathsf{N})$. This is proved by induction on $n \in El_0(\mathsf{N}) = \mathsf{Num}$, using the ι-rule for rec. It follows that $\mathsf{rec} \in El_0([\![C\,\mathsf{0} \to (\Pi n{:}\mathsf{N}.\, C\, n \to C\,(\mathsf{S} n)) \to \Pi n{:}\mathsf{N}.\, C\, n]\!]_\rho)$. Suppose $\Gamma \vdash \mathsf{eq} : \mathsf{N} \to \mathsf{N} \to \mathsf{U}$ by $\Gamma \vdash$. That $\mathsf{N} \to \mathsf{N} \to \mathsf{U} \in T_1$ follows from Lemma 13. In order to show $\mathsf{eq} \in El_1(\mathsf{N} \to \mathsf{N} \to \mathsf{U})$, we have to prove $\mathsf{eq}\, m\, n \in T_0$ for all m and n in Num. This is proved by nested induction on m and n.

Regarding lists, if $\Gamma \vdash [A] : \mathsf{U}$ by $\Gamma \vdash A : \mathsf{U}$, then by induction hypothesis we get $[\![A]\!]_\rho \in T_0$. It follows from Lemma 13 that $[\![[A]]\!]_\rho = [[\![A]\!]_\rho] \in T_0$. If $\Gamma \vdash \mathsf{nil} : [A]$ by $\Gamma \vdash A : \mathsf{U}$, the claim follows easily from the induction hypothesis and Lemma 13. The case for $\Gamma \vdash \mathsf{cons} : A \to [A] \to [A]$ by $\Gamma \vdash A : \mathsf{U}$ is similar. Suppose $\Gamma \vdash \mathsf{lrec} : C\,\mathsf{nil} \to (\Pi a{:}A.\, \Pi l{:}[A].\, C\, l \to C\,(a :: l)) \to \Pi l{:}[A].\, C\, l$ by $\Gamma \vdash A$ and $\Gamma \vdash C : [A] \to \mathsf{U}$. By induction hypothesis on $\Gamma \vdash C : [A] \to \mathsf{U}$, we have $[\![[A] \to \mathsf{U}]\!]_\rho \in T_1$ and $[\![C]\!]_\rho \in El_1([\![[A] \to \mathsf{U}]\!]_\rho)$. From Lemma 13, it follows that $[\![C\mathsf{nil}]\!]_\rho \in T_0$, $[\![\Pi a{:}A.\, \Pi l{:}[A].\, C\, l \to C\,(a :: l)]\!]_\rho \in T_1$ and $[\![\Pi l{:}[A].\, C\, l]\!]_\rho \in T_1$, hence $[\![C\,\mathsf{nil} \to (\Pi a{:}A.\, \Pi l{:}[A].\, C\, l \to C\,(a :: l)) \to \Pi l{:}[A].\, C\, l]\!]_\rho \in T_1$. In order to show that $\mathsf{lrec} \in El_1([\![C\,\mathsf{nil} \to (\Pi a{:}A.\, \Pi l{:}[A].\, C\, l \to C(a :: l)) \to \Pi l{:}[A].\, C\, l]\!]_\rho)$, we have to prove, for all $M \in El_1([\![C\,\mathsf{nil}]\!]_\rho)$, $N \in El_1([\![\Pi a{:}A.\,\Pi l{:}[A].\,C\,l \to C\,(a :: l)]\!]_\rho)$ and $l \in El_1([[\![A]\!]_\rho])$, that $\mathsf{lrec}\,M\,N\,l \in El_1([\![C\,l]\!]_\rho)$. This is proved by induction on $l \in El_1([[\![A]\!]_\rho]) = \mathsf{List}([\![A]\!]_\rho)$, using the ι-rewrite rules for lrec.

Sum types can be dealt with in a similar (but simpler) manner as list types.

Regarding dependent pairs, if $\Gamma \vdash \Sigma x{:}A.\,B : \mathsf{U}$ by $\Gamma \vdash A : \mathsf{U}$ and $\Gamma, x : A \vdash B : \mathsf{U}$, then we argue analogously to the case for $\Gamma \vdash \Pi x{:}A.\,B : \mathsf{U}$. Suppose $\Gamma \vdash (W, P) : \Sigma x{:}A.\,B$ by $\Gamma \vdash \Sigma x{:}A.\,B : \mathsf{U}$, $\Gamma \vdash W : A$ and $\Gamma \vdash P : B(W)$. We obtain $\Sigma x{:}A.\,B \in T_1$ by induction hypothesis. To show that $[\![(W,P)]\!]_\rho = ([\![W]\!]_\rho, [\![P]\!]_\rho) \in El_1([\![\Sigma x{:}A.\,B]\!]_\rho)$, it suffices to prove $[\![W]\!]_\rho \in El_1([\![A]\!]_\rho)$ and $[\![P]\!]_\rho \in El_1([\![B]\!]_\rho([\![W]\!]_\rho)) = El_1([\![B(W)]\!]_\rho)$, both of which follow from induction hypothesis. Suppose $\Gamma \vdash \mathsf{srec} : (\Pi x{:}A.\,\Pi p{:}B.\,C\,(x,p)) \to \Pi y{:}(\Sigma x{:}A.\,B).\,C\,y$ by $\Gamma \vdash A : \mathsf{U}$, $\Gamma, x : A \vdash B : \mathsf{U}$, $\Gamma \vdash \Sigma x{:}A.\,B : \mathsf{U}$, and $\Gamma \vdash C : (\Sigma x{:}A.\,B) \to \mathsf{U}$. Induction hypotheses give us $[\![A]\!]_\rho \in T_0$, $[\![B]\!]_{\rho(x/a)} \in T_0$ for all $a \in El_0([\![A]\!]_\rho)$, $[\![\Sigma x{:}A.\,B]\!]_\rho \in T_0$, and $[\![C]\!]_\rho \in El_1([\![\Sigma x{:}A.\,B]\!]_\rho \to \mathsf{U})$. It follows that $[\![\Pi x{:}A.\,\Pi p{:}B.\,C\,(x,p)]\!]_\rho \in T_0$ and $[\![\Pi y{:}(\Sigma x{:}A.\,B).\,C\,y]\!]_\rho \in T_0$, and hence $[\![\Pi x{:}A.\,\Pi p{:}B.\,C\,(x,p) \to \Pi y{:}(\Sigma x{:}A.\,B).\,C\,y]\!]_\rho \in T_0$. By using the ι-rule for srec and by Lemma 13, we get $\mathsf{srec} \in El_0([\![\Pi x{:}A.\,\Pi p{:}B.\,C\,(x,p) \to \Pi y{:}(\Sigma x{:}A.\,B).\,C\,y]\!]_\rho)$.

Regarding the constant bAr, if $\Gamma \vdash \mathsf{bAr} : ([A] \to \mathsf{U}) \to [A] \to \mathsf{U}$ by $\Gamma \vdash A : \mathsf{U}$, then we have $[\![([A] \to \mathsf{U}) \to [A] \to \mathsf{U}]\!]_\rho \in T_1$ by induction hypothesis and Lemma 13. To show that $\mathsf{bAr} \in El_1([\![([A] \to \mathsf{U}) \to [A] \to \mathsf{U}]\!]_\rho) = El_1(([[\![A]\!]]_\rho \to \mathsf{U}) \to [[\![A]\!]]_\rho \to \mathsf{U})$, it suffices that $\mathsf{bAr}\,P\,l \in T_0$ for all $P \in El_1([[\![A]\!]]_\rho \to \mathsf{U})$ and $l \in El_0([[\![A]\!]]_\rho)$. This follows from Lemma 13, since $P\,l' \in T_0$ for all $l' \in El_0([[\![A]\!]]_\rho)$. Suppose $\Gamma \vdash \mathsf{base}\,X : \mathsf{bAr}\,P\,l$ by $\Gamma \vdash A : \mathsf{U}$, $\Gamma \vdash l : [A]$, $\Gamma \vdash P : [A] \to \mathsf{U}$ and $\Gamma \vdash X : P\,l$. By induction hypothesis, we have $[\![A]\!]_\rho \in T_0$, $[\![l]\!]_\rho \in El_1([[\![A]\!]]_\rho)$, and $[\![P]\!]_\rho \in El_1([\![[A] \to \mathsf{U}]\!]_\rho) = El_1([[\![A]\!]]_\rho \to \mathsf{U})$, hence $[\![P]\!]_\rho\,l' \in T_0$ for all $l' \in El_1([[\![A]\!]]_\rho)$. This proves $\mathsf{bAr}\,P\,l \in T_0$. The induction hypothesis on $\Gamma \vdash X : P\,l$ gives us $[\![X]\!]_\rho \in El_1([\![P]\!]_\rho\,[\![l]\!]_\rho)$, which proves $[\![\mathsf{base}\,X]\!]_\rho \in El_1([\![\mathsf{bAr}\,P\,l]\!]_\rho)$ by Lemma 13. The case for $\Gamma \vdash \mathsf{step}\,Y : \mathsf{bAr}\,P\,l$ follows analogously. We will elaborate the last and most interesting case in some detail. Suppose

$$\begin{aligned}\Gamma \vdash \mathsf{barrec} : (\Pi l{:}[A].\,P\,l \to C\,l) \to \\ (\Pi l{:}[A].\,(\Pi a{:}A.\,C\,(a :: l)) \to C\,l) \to \Pi l{:}[A].\,\mathsf{bAr}P\,l \to C\,l\end{aligned}$$

by $\Gamma \vdash A : U$, $\Gamma \vdash P : [A] \to U$, and $\Gamma \vdash C : [A] \to U$. By induction hypothesis, using Lemma 13 many times, we get that the type of barrec is in T_1. In order to show that barrec is an element of the corresponding set

$$\begin{aligned}El_1(([\![\Pi l{:}[A].\,P\,l \to C\,l) \to \\ (\Pi l{:}[A].\,(\Pi a{:}A.\,C\,(a :: l)) \to C\,l) \to \Pi l{:}[A].\,\mathsf{bAr}P\,l \to C\,l]\!]_\rho),\end{aligned}$$

it suffices that

$$El_1(\mathsf{bAr}\,[\![P]\!]_\rho\,l) \subseteq \{M \in \mathsf{D} \mid \mathsf{barrec}\,B\,S\,l\,M \in El_1([\![C]\!]_\rho\,l)\}$$

for all $B \in El_1([\![\Pi l{:}[A].\,(P\,l \to C\,l)]\!]_\rho)$, $S \in El_1([\![\Pi l{:}[A].\,((\Pi a{:}A.\,C\,(a :: l)) \to C\,l)]\!]_\rho)$, and $l \in El_1([[\![A]\!]]_\rho)$. We prove this by fixpoint induction, recalling that, for fixed A, $El_1(\mathsf{bAr}\,[\![P]\!]_\rho\,l)$ is defined as the fixpoint of $\Psi_{\mathsf{Bar}(El_1([\![A]\!]_\rho),\ El_1([[\![A]\!]]_\rho) \ni l \mapsto El_1([\![P]\!]_\rho l))}$. The latter operator will be abbreviated to Ψ, as A and P do not change in the proof.

We show that the function

$$\phi : El_1([[\![A]\!]]_\rho) \ni l \mapsto \{M \in \mathsf{D} \mid \mathsf{barrec}\,S\,B\,l\,M \in El_1([\![C]\!]_\rho\,l)\}$$

is a prefixpoint of Ψ, i.e., $\Psi(\phi)(l) \subseteq \phi(l)$ for all $l \in El_1([[\![A]\!]]_\rho)$. By definition, there are two forms of elements in $\Psi(\phi)(l)$. The first is $\mathsf{base}\,X$ with $X \in El_1([\![P]\!]_\rho\,l)$. Then we have

$\mathsf{barrec}\,B\,S\,l\,(\mathsf{base}\,X) = B\,l\,X \in El_1([\![C]\!]_\rho\,l$ by the assumptions on B and l. The second is $\mathsf{step}\,Y$ with, for all $a \in El_1([\![A]\!]_\rho)$, $Y a \in \phi\,(a :: l)$, that is, $\mathsf{barrec}\,B\,S\,(a :: l)\,(Y a) \in El_1([\![C]\!]_\rho\,(a :: l))$. Then we have $\mathsf{barrec}\,B\,S\,l\,(\mathsf{step}\,Y) = S\,l\,(\lambda a.\,\mathsf{barrec}\,B\,S\,(a :: l)\,(Y a) \in El_1([\![C]\!]_\rho\,l)$ by the assumptions on S and l.

It remains to prove that the auxiliary rules are sound. These rules define the type and computational behaviour of the constants $\mathsf{exists}, \mathsf{good}, \mathsf{length}, \mathsf{take}$.

Regarding the constant exists, if $\Gamma \vdash \mathsf{exists} : (A \to \mathsf{U}) \to [A] \to \mathsf{U}$ by $\Gamma \vdash A : \mathsf{U}$, then we have $[\![(A \to \mathsf{U}) \to [A] \to \mathsf{U}]\!]_\rho \in T_1$ by induction hypothesis and Lemma 13. To show that $\mathsf{exists} \in El_1([\![(A \to \mathsf{U}) \to [A] \to \mathsf{U}]\!]_\rho) = El_1(([\![A]\!]_\rho \to \mathsf{U}) \to [\![[A]]\!]_\rho \to \mathsf{U})$, it suffices that $\mathsf{exists}\,P\,l \in T_0$ for all $P \in El_1([\![A]\!]_\rho \to \mathsf{U})$ and $l \in \mathsf{List}(El_0([\![A]\!]_\rho)$. This follows by induction on l from the ι-reduction rules for exists. The argumentation for the other constants is very similar, and will hence be left to the reader. ◀

We obtain that if an expression M has a type A, then M realizes A.

▶ **Corollary 19.** *If* $\vdash M : A$*, then* $A \in T_1$ *and* $M \in El_1(A)$*.*

5 A Realizer for Markov's Principle

Markov's Principle is the following type:

$$\mathsf{MP} := \Pi f{:}\mathsf{N}{\to}\mathsf{N}_2.\,(\neg\neg\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1) \to \Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1$$

Clearly $\vdash \mathsf{MP} : \mathsf{U}$, so $\mathsf{MP} \in T_1$ by Corollary 19. As a proposition, MP is classically true but unprovable in Heyting Arithmetic [14]; MP is not inhabited in our type theory. However, as we will show in this section, MP can be consistently added to the type theory: $El_1(\mathsf{MP})$ is non-empty. In other words, MP is true in the realizability model.

The realizer $R_{MP} \in El_1(\mathsf{MP})$ to be defined below essentially performs an unbounded search for an n such that $f\,n = 1$. This is possible since the computational system, based on untyped lambda calculus, is Turing complete. To prove that the search always finds such an n, we use Markov's Principle on the metalevel. This is possible since we are allowed to reason classically in the ambient naive set theory.

Recall that $\mathsf{beq} : \mathsf{N}_2 \to \mathsf{N}_2 \to \mathsf{U}$, $\mathsf{beq}\,1\,1 = \mathsf{N}_1$ and $0 : \mathsf{N}_1$. Let Y be any fixed point operator in the untyped lambda calculus, for example $Y := \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))$. Then $Y\,F = F\,(Y\,F)$ for any F, in particular for

$$F := \lambda s\,f\,n.\,\mathsf{brec}\,(s\,f\,(\mathsf{S}\,n))\,(n, 0)\,(f\,n)$$

Then we have, for $\mathsf{search} := Y\,F$, that

$$\mathsf{search}\,f\,n = (F\,\mathsf{search})\,f\,n = \mathsf{brec}\,(\mathsf{search}\,f\,(\mathsf{S}\,n))\,(n, 0)\,(f\,n)$$

This means that $\mathsf{search}\,f\,0$ performs the required search for the first n such that $f\,n = 1$. If n is found, $(n, 0)$ is returned, that is, the pair consisting of the numeral n and the proof term 0 of type $\mathsf{beq}\,(f\,n)\,1$.

We define $R_{MP} = \lambda f\,p.\,\mathsf{search}\,f\,0$ and it remains to prove $R_{MP} \in El_1(\mathsf{MP})$. Note that p does not occur in $\mathsf{search}\,f\,0$. This is typical for realizability: realizers of negative statements carry no computational content, they only witness that the statement that is negated has no realizers. To show $R_{MP} \in El_1(\mathsf{MP})$, let $f \in El_1(\mathsf{N}{\to}\mathsf{N}_2)$ and $p \in El_1(\neg\neg\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$. We have to prove that $\mathsf{search}\,f\,0 \in El_1(\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$. Towards a contradiction, assume the latter set is empty. Then, any term foo is in $El_1(\neg\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$. It follows that

$p\,foo \in El_1(\mathsf{N}_0)$. This is absurd, so $El_1(\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$ is non-empty. Since it is decidable for any numeral n, whether or not $(n,0) \in El_1(\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$, it follows by Markov's Principle that there exists a pair $(n,0) \in El_1(\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$. Hence there is also such a pair with the smallest n, that is, $\mathsf{search}\,f\,0 \in El_1(\Sigma n{:}\mathsf{N}.\,\mathsf{beq}\,(f\,n)\,1)$. Note the role of p in the above argument: it serves to prove termination of the search but does not influence the actual outcome.

6 A Realizer for the Undecidability of the Halting Problem

The purpose of this section is to argue that, in addition to MP, we can consistently add the undecidablity of the halting problem to the type theory. Define

$$\mathsf{H_t} := \lambda n.\,\Sigma k{:}\mathsf{N}.\,\mathsf{beq}\,(t\,n\,n\,k)\,1$$

for $t : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}_2$ as described below. The intention is that $\mathsf{H_t}$ is a halting predicate, with t the characteristic function of Kleene's T-predicate [7, 2]. Using rec, we can define all primitive recursive functions, and actually many more. Since Kleene's T-predicate is primitive recursive, its characteristic function t is definable in the type theory. Kleene's T-predicate, $T\,e\,x\,w$, is based on a standard encoding of partial recursive functions as natural numbers. The first argument e of T is such a code of a partial recursive function, whereas the second argument x encodes an input to this function. The third argument w encodes a (terminating) computation sparked off by the function with code e on input with code x. Hence, $\mathsf{H_t}\,n$ holds if and only if the function encoded by n terminates on the input coded by n.

Let UH be the type:

$$\mathsf{UH} := \neg\Pi n{:}\mathsf{N}.\,(\mathsf{H_t}\,n + \neg\mathsf{H_t}\,n).$$

Clearly $\vdash \mathsf{UH} : \mathsf{U}$, so $\mathsf{UH} \in T_1$ by Corollary 19. We want to show that $El_1(\mathsf{UH})$ is non-empty. As UH is negative, it suffices to show that $\Pi n{:}\mathsf{N}.\,(\mathsf{H_t}\,n + \neg\mathsf{H_t}\,n)$ cannot be realized. Then any term realizes UH, so $El_1(\mathsf{UH})$ contains all terms of Λ (!). Towards a contradiction, assume $f \in El_1(\Pi n{:}\mathsf{N}.\,(H_t\,n + \neg H_t\,n))$. Diagonalizing over f we define:

$$d = \lambda n.\,\mathsf{case}\,\Omega\,1\,(f\,n)$$

such that in view or the definition of $\mathsf{H_t}$ we have for all $n : \mathsf{N}$:

$$d\,n = \Omega \iff f\,n = \mathsf{inl}(k,0) \iff T(n,n,k),$$

where $(k,0) \in El_1(\Sigma k{:}\mathsf{N}.\,\mathsf{beq}\,(t\,n\,n\,k)\,1)$.

As a lambda term, d represents a partial recursive function with code a numeral n_d[5]. Then we have $d\,n_d = \Omega \iff T(n_d,n_d,k)$ where $\mathsf{inl}(k,0) = f\,n_d$, a plain contradiction with the choice of T and f. Therefore f as above cannot exist, and any term realizes UH. We conclude that UH can be consistently added to the type theory.

7 A set that is provably streamless but not provably Noetherian

In this section we shall prove that the converse of Theorem 5 is unprovable in type theory. The argument sketched in the introduction is that the converse is false when MP and UH

[5] The code n_d can in principle be constructed from d, but this is outside the scope of this paper.

are assumed. The unprovability in type theory then follows from the realizability model in which MP and UH are both valid. We start by some auxiliary definitions.

Given a predicate P on natural numbers, we define a binary relation $\overset{P}{=}$ to be the equality on the set of natural numbers n which satisfy P, irrelevant of the proof of $P\,n$. Formally, we define a typing rule for $\overset{P}{=}$ by

$$\frac{\Gamma \vdash P : \mathsf{N} \to \mathsf{U}}{\Gamma \vdash \overset{P}{=} : (\Sigma n{:}\mathsf{N}.\, P\,n) \to (\Sigma n{:}\mathsf{N}.\, P\,n) \to \mathsf{U}}$$

and ι-reduction, with $\overset{P}{=}$ written infix, given by

$$(n, h_n) \overset{P}{=} (m, h_m) = \mathsf{eq}\, n\, m.$$

Since eq is decidable, $\overset{P}{=}$ is decidable for any P.

Given a predicate P on natural numbers, we define a predicate $\overline{P}n$ to be true if Pk is decidable for all $k < n$. Formally, we define a typing rule for $\overline{P}$ by

$$\frac{\Gamma \vdash P : \mathsf{N} \to \mathsf{U}}{\Gamma \vdash \overline{P} : \mathsf{N} \to \mathsf{U}}$$

and ι-reductions give by

$$\begin{array}{lcl} \overline{P}0 & = & \mathsf{N}_1 \\ \overline{P}(\mathsf{S}n) & = & (P\,n + \neg P\,n) \times \overline{P}n \end{array}$$

The realizability model can easily be extended with sound interpretations of the above.

▶ **Lemma 20.** *There is a proof M such that*

$$P : \mathsf{N} \to \mathsf{U}, n : \mathsf{N} \vdash M : \overline{P}n \to \neg\neg\overline{P}(\mathsf{S}n).$$

Proof. We have to prove absurdity from $\overline{P}n$ and $\neg\overline{P}(\mathsf{S}n)$. Assume $Pn + \neg Pn$, then by $\overline{P}n$ we get $\overline{P}(\mathsf{S}n)$, which contradicts the assumption $\neg\overline{P}(\mathsf{S}n)$. Hence $\neg(Pn + \neg Pn)$, which is absurd, as $\neg\neg(A + \neg A)$ is a constructive tautology. ◀

▶ **Corollary 21.** *There is a proof M such that $P : \mathsf{N} \to \mathsf{U} \vdash M : \Pi n{:}\mathsf{N}.\, \neg\neg\overline{P}n$.*

Proof. By induction on n, using $\overline{P}0$ and basic facts about $\neg\neg$. ◀

Recall the terminology and notaion on decidability from Definition 1. We have the following easy lemmas about decidability.

▶ **Lemma 22.** *There exists proofs M_1, M_2, M_3, M_4 such that*

$$\begin{aligned} A : \mathsf{U}, P : A \to \mathsf{U} &\vdash M_1 : (\Pi x{:}A.\, \mathsf{dec}\,(P\,x)) \to \Pi l{:}[A].\, \mathsf{dec}\,(\mathsf{exists}\, P\, l) : \\ A : \mathsf{U}, R : A \to A \to \mathsf{U} &\vdash M_2 : (\Pi x{:}A.\, \Pi y{:}A.\, \mathsf{dec}\,(R\,x\,y)) \to \Pi l{:}[A].\, \mathsf{dec}\,(\mathsf{good}\, R\, l); \\ P : \mathsf{N} \to \mathsf{U} &\vdash M_3 : \Pi p_1{:}\Sigma n{:}\mathsf{N}.\, P\,n.\, \Pi p_2{:}\Sigma n{:}\mathsf{N}.\, P\,n.\, \mathsf{dec}\,(p_1 \overset{P}{=} p_2); \\ P : \mathsf{N} \to \mathsf{U} &\vdash M_4 : \Pi l{:}[\Sigma n{:}\mathsf{N}.\, P\,n].\, \mathsf{dec}\,(\mathsf{good} \overset{P}{=} l). \end{aligned}$$

Proof. The first two are easily proved by induction on l, where the second uses the first. The third follows from the definition of $\overset{P}{=}$ and Lemma 2. The fourth follows from the second and the third. Note that the fourth typing states that it is decidable whether a list over a *subset* of natural numbers contains proof-irrelevant duplicates. ◀

In order to prove that the converse of Theorem 5 is not provable in the type theory, we construct a set which is provably not Noetherian, but can be proved streamless using MP and UH.

The following lemma is an abstract form of the argument in the introduction. In order to see this, recall that $(\mathsf{good} \overset{Q}{=})$ is a predicate expressing that a list contains a proof-irrelevant duplicate.

▶ **Lemma 23.** *Let A in* bAr *be the type $\Sigma n{:}\mathsf{N}.\, Q\, n$. There is a proof M such that*

$$Q : \mathsf{N} \to \mathsf{U} \vdash M : (\Pi n{:}\mathsf{N}.\, \neg\neg Q\, n) \to \Pi l{:}[\Sigma n{:}\mathsf{N}.\, Q\, n].\, \mathsf{bAr}\,(\mathsf{good} \overset{Q}{=})\, l \to \mathsf{good} \overset{Q}{=} l.$$

Proof. Let $Q : \mathsf{N} \to \mathsf{U}$ and $h_Q : \Pi n{:}\mathsf{N}.\, \neg\neg Q\, n$. We use induction on $\mathsf{bAr}\,(\mathsf{good} \overset{Q}{=})l$. If $\mathsf{bAr}\,(\mathsf{good} \overset{Q}{=})\, l$ by $\mathsf{good} \overset{Q}{=} l$, the claim holds immediately. Assume as induction hypothesis $\Pi x{:}(\Sigma n{:}\mathsf{N}.\, Q\, n).\, \mathsf{good} \overset{Q}{=} (x :: l)$. We have to prove $\mathsf{good} \overset{Q}{=} l$. By Lemma 22 we can reason by contradiction. Assume $\neg(\mathsf{good} \overset{Q}{=} l)$. We prove this is absurd and we are done. We perform case analysis on the shape of l. In case $l = \mathsf{nil}$, if $h_0 : Q\, 0$, then $\mathsf{good} \overset{Q}{=} ((0, h_0) :: \mathsf{nil})$ by the induction hypothesis. This is absurd, so $\neg Q\, 0$, which is absurd by assumption h_Q. In case l is a non-empty list, let (n, h_n) be a maximum element in l, that is, for any (m, h_m) such that $\mathsf{exists}\,(\overset{Q}{=} (m, h_m))\, l$, we have that $n \geq m$. A maximum element exists since l is non-empty. It suffices to prove $\neg Q\,(\mathsf{S}\, n)$, which contradicts h_Q. Assume we have a proof $h_{\mathsf{S}\, n} : Q\,(\mathsf{S}\, n)$. By induction hypothesis, we have $\mathsf{good} \overset{Q}{=} ((\mathsf{S}\, n, h_{\mathsf{S}\, n}) :: l)$. Since we assumed $\neg(\mathsf{good} \overset{Q}{=} l)$, it must be that $\mathsf{exists}\,(\overset{Q}{=} (\mathsf{S}\, n, h_{\mathsf{S}\, n}))\, l$, which contradicts with (n, h_n) being a maximum element in l. ◀

Noticing that it is absurd that the empty list is good, we deduce, from Lemma 23 and Corollary 21, that it is absurd that $\overset{H}{=}$ is Noetherian.

▶ **Lemma 24.** *There is a proof M such that $H : \mathsf{N} \to \mathsf{U} \vdash M : \neg(\mathsf{Noetherian} \overset{H}{=})$.*

On the other hand, in the presence of MP and UH, for $H_t : \mathsf{N} \to \mathsf{U}$ as defined in Section 6, we can prove that $\overset{H_t}{=}$ is streamless.

▶ **Lemma 25.** *There is a proof M such that*

$$\vdash M : \mathsf{MP} \to \mathsf{UH} \to \mathsf{streamless} \overset{\overline{\mathsf{H_t}}}{=}.$$

Proof. Assume MP and UH. Given $f : \mathsf{N} \to \Sigma n{:}\mathsf{N}.\, \overline{\mathsf{H_t}} n$, we want to prove $\Sigma n{:}\mathsf{N}.\, \mathsf{good} \overset{\overline{\mathsf{H_t}}}{=} (\overline{f} n)$. Noting that $(\mathsf{good} \overset{\overline{\mathsf{H_t}}}{=})$ is decidable by Lemma 22, we can construct a function $e : \mathsf{N} \to \mathsf{N_2}$ such that $\mathsf{eq}\,(e\, n)\, 1$ is true if and only if $\mathsf{good} \overset{\overline{\mathsf{H_t}}}{=} (\overline{f} n)$ is true, for all $n : \mathsf{N}$. Thus, we may apply MP and it then suffices to prove $\neg\neg\Sigma n{:}\mathsf{N}.\, \mathsf{good} \overset{\overline{\mathsf{H_t}}}{=} (\overline{f} n)$. Suppose $\neg\Sigma n{:}\mathsf{N}.\, \mathsf{good} \overset{\overline{\mathsf{H_t}}}{=} (\overline{f} n)$, or equivalently, $\Pi n{:}\mathsf{N}.\, \neg\mathsf{good} \overset{\overline{\mathsf{H_t}}}{=} (\overline{f} n)$. Then, for any given $n : \mathsf{N}$, the list $\overline{f}(\mathsf{S}(\mathsf{S} n))$ gives us a proof of $\mathsf{H_t}\, n + \neg \mathsf{H_t}\, n$[6]. This contradicts UH. ◀

[6] Informally, the list $\overline{f}(\mathsf{S}(\mathsf{S} n))$ contains a maximum element (m, h_m) such that $m \geq n$. The proof h_m of $\overline{\mathsf{H_t}} m$ gives us $H\, n + \neg H\, n$.

We have now shown that, in the presence of MP and UH, there is a relation that is streamless but cannot be Noetherian. Recalling that MP and UH are consistent with the type theory we work in, we conclude that it is unprovable in the type theory that every streamless relation is Noetherian.

▶ **Theorem 26.** *There is no proof M such that*

$$\vdash M : \mathsf{streamless} \overset{\overline{\mathsf{H_t}}}{=} \;\rightarrow\; \mathsf{Noetherian} \overset{\overline{\mathsf{H_t}}}{=}.$$

▶ **Corollary 27.** *There is no proof M such that*

$$\vdash M : \Pi A{:}\mathsf{U}.\,\Pi R{:}A \rightarrow A \rightarrow \mathsf{U}.\,\mathsf{streamless}\,R \;\rightarrow\; \mathsf{Noetherian}\,R.$$

8 Related work and concluding remarks

We have constructed a realizability model for Martin-Löf dependent type theory, viewed as a set of typing rules typing terms of an untyped lambda-calculus Λ. Similar realizability models have been given by Martin Löf [9] and Beeson [1]. We have paid extra attention to the details, in particular to those of the type of an inductive bar.

The purpose of the model is to demonstrate a particular unprovability result. It may be illuminating to discuss this result in connection with the well-known Kleene Tree [7, 2], a primitive recursive relation P on binary sequences which defines an infinite tree without an infinite recursive branch. The Kleene Tree is the prime example that Brouwer's Fan Theorem (an intuitionistic version of König's Lemma) fails in recursive analysis. In the context of this paper, Kleene's result yields that, with $A = \mathsf{N_2}$ and for a specific decidable $P : [\mathsf{N_2}] \rightarrow \mathsf{U}$, the following type is not inhabited:

$$(\Pi f{:}\mathsf{N} \rightarrow A.\,\Sigma n{:}\mathsf{N}.\,P\,(\overline{f}n)) \;\rightarrow\; \mathsf{bAr}\,P\,\mathsf{nil}.$$

For comparison, our result, with $A = \Sigma n{:}\mathsf{N}.\,\overline{\mathsf{H_t}}n$ and $Q := \mathsf{good} \overset{\overline{\mathsf{H_t}}}{=}$, states that the following type is not inhabited:

$$(\Pi f{:}\mathsf{N} \rightarrow A.\,\Sigma n{:}\mathsf{N}.\,Q\,(\overline{f}n)) \;\rightarrow\; \mathsf{bAr}\,Q\,\mathsf{nil}.$$

The important difference between the two results is the instantiation of the type A, that is, $A = \mathsf{N_2}$ for Kleene and $A = \Sigma n{:}\mathsf{N}.\,\overline{\mathsf{H_t}}n$ for us. Clearly, $A = \mathsf{N_2}$ is the simpler of the two. On the other hand, with $Ql := \mathsf{good} \overset{\overline{\mathsf{H_t}}}{=} l$ expressing that the list l contains a duplicate, we are allowed far less expressive power than Kleene's P. This explains to some extent why we use a more complicated base type $A = \Sigma n{:}\mathsf{N}.\,\overline{\mathsf{H_t}}n$, which is the simplest type we could find defining a subset of the natural numbers that is not finite in the sense of Noetherian, and at the same time finite in the sense of streamless in a consistent extension of the type theory. This confirms a conjecture formulated by Coquand and Spiwack in [4]. We conclude by formulating an open problem: does there exist a *decidable* P such that $\Sigma n{:}\mathsf{N}.\,Pn$ distinguishes between Noetherian and streamless?

References

1 M. Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.

2 M. J. Beeson. *Foundations of Constructive Mathematics*, volume 6 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer, 1985.

3 T. Coquand and A. Spiwack. A proof of strong normalisation using domain theory. *Logical Methods in Computer Science*, 3(4), 2007.

4 T. Coquand and A. Spiwack. Constructively finite? In L. Lambán, A. Romero, and J. Rubio, editors, *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 217–230. Publicaciones de la Universidad de La Rioja, 2010. ISBN 978-84-96487-50-5.

5 P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000.

6 M. Escardó. Joins in the frame of nuclei. *Applied Categorical Structures*, 11:117–124, 2003.

7 S.C. Kleene. Recursive functions and intuitionistic mathematics. In L.M. Graves, E. Hille, P.A. Smith, and O. Zariski, editors, *Proceedings of the International Congress of Mathematicians*, pages 679–685. AMS, 1952.

8 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science*, 121(1&2):279–308, 1993.

9 P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118, Amsterdam, 1975. North-Holland.

10 E. Parmann. `https://github.com/epa095/noetherian-implies-streamless`.

11 E. Parmann. *Case Studies in Constructive Mathematics*. PhD thesis, University of Bergen, 2016.

12 D. Pataraia. A constructive proof of Tarski's fixed-point theorem for DCPOs. Presented at the 65th Peripatetic Seminar on Sheaves and Logic, November 1997.

13 C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics. In J.C.E. Dekker, editor, *Recursive function theory, Proc. Symp. in pure mathematics V*, pages 1–27. AMS, 1962.

14 A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, 1973.

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

Parametricity, Automorphisms of the Universe, and Excluded Middle

Auke B. Booij
School of Computer Science, University of Birmingham, Birmingham, UK
abb538@cs.bham.ac.uk

Martín H. Escardó
School of Computer Science, University of Birmingham, Birmingham, UK
m.escardo@cs.bham.ac.uk

Peter LeFanu Lumsdaine
Mathematics Department, Stockholm University, Stockholm, Sweden
p.l.lumsdaine@math.su.se

Michael Shulman[1]
Department of Mathematics, University of San Diego, San Diego, USA
shulman@sandiego.edu

Abstract

It is known that one can construct non-parametric functions by assuming classical axioms. Our work is a converse to that: we prove classical axioms in dependent type theory assuming specific instances of non-parametricity. We also address the interaction between classical axioms and the existence of automorphisms of a type universe. We work over intensional Martin-Löf dependent type theory, and for some results assume further principles including function extensionality, propositional extensionality, propositional truncation, and the univalence axiom.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases relational parametricity, dependent type theory, univalent foundations, homotopy type theory, excluded middle, classical mathematics, constructive mathematics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.7

Acknowledgements The first-named author would like to thank Uday Reddy for discussions about parametricity. We would also like to thank Jean-Philippe Bernardy for helpful comments, and Andrej Bauer for discussions and questions. The fact that the implication DNE → LEM in Lemma 14 requires **0**-valued function extensionality was spotted by one of the referees. Since then, we have formalized the results to make sure we didn't miss similar assumptions. All results of the paper, except Section 3.1, are formalized in Agda [3]. All results of Sections 3.1 and 3.2 are formalized in Coq [6] in the files `Spaces/BAut/Rigid.v` and `Spaces/Universe.v`. So some results have been formalized twice, and no numbered result has been left unformalized.

[1] Supported by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government, or Carnegie Mellon University.

1 Introduction: Parametricity in dependent type theory

Broadly speaking, *parametricity* statements assert that type-polymorphic functions definable in some system must be natural in their type arguments, in some suitable sense. Reynolds' original theory of relational parametricity [12] characterizes terms of the polymorphically typed λ-calculus System F. This theory has since been extended to richer and more expressive type theories: to pure type systems by Bernardy, Jansson, and Paterson [2], and more specifically to dependent type theory by Atkey, Ghani, and Johann [1].

Most parametricity results are meta-theorems about a formal system and make claims only about terms in the empty context. For instance, Reynolds' results show that the only term of System F with type $\forall\alpha.\alpha \to \alpha$ definable in the empty context is the polymorphic identity function $\Lambda\alpha.\lambda(x : \alpha).x$. Similarly, Atkey, Ghani, and Johann [1, Thm. 2] prove that any term $f : \prod_{X:\mathcal{U}} X \to X$ definable in the empty context of MLTT must satisfy $e(f_X(a)) = f_Y(e(a))$ for all $e : X \to Y$ and $a : X$ in their model; it follows that f acts as the identity on every type in their model, and hence no such closed term f can be provably not equal to the polymorphic identity function.

Keller and Lasson showed that excluded middle is incompatible with parametricity of the universe of types (in its usual formulation) [7]. In this paper, we show, within type theory, that certain violations of parametricity are possible if and only if certain classical principles hold. For example, we show that there is a function $f : \prod_{X:\mathcal{U}} X \to X$ whose value at the type $\mathbf{2}$ of booleans is different from the identity if and only if excluded middle holds (Theorem 1, where one direction uses function extensionality).

These are theorems *of* dependent type theory, so they apply not only to closed terms but in any context, and the violations of parametricity are expressed using negations of Martin-Löf's identity type rather than judgemental (in-)equality of terms. Similarly, we show that excluded middle also follows from certain kinds of non-trivial automorphisms of the universe.

We work throughout in intensional Martin-Löf type theory, with at least Π-, Σ-, identity, finite, and natural numbers types, and a universe closed under these type-formers. For concreteness, this may be taken to be the theory of [11], or of [14, A.2]. When results require further axioms – e.g. function extensionality, or univalence of the universe – we include these as explicit assumptions, to keep results as sharp as possible.

By the *law of excluded middle*, we mean always the version from univalent foundations [14, 3.4.1], namely that $P + \neg P$ for all *propositions* P. Here a type is called a "proposition" (a "mere proposition" in the terminology of [14]) if it has at most one element, meaning that any two of its elements are equal in the sense of the identity type. Note that $\neg P$ (meaning $P \to \mathbf{0}$) is not itself a proposition unless we assume function extensionality, at least for $\mathbf{0}$-valued functions.

The *propositional truncation* of a type A is the universal proposition $\|A\|$ admitting a map from A. We axiomatize this as in [14, §3.7], and always indicate explicitly when we are assuming it. It is shown in [9] that propositional truncation implies function extensionality.

When propositional truncations exist, the *disjunction* of two propositions $P \vee Q$ is defined to be $\|P + Q\|$. If P and Q are disjoint (i.e. $\neg(P + Q)$ holds), then $P + Q$ is already a proposition and hence equivalent to $P \vee Q$. In particular, when we have propositional truncations, the law of excluded middle could equivalently assert that $P \vee \neg P$ for all propositions P.

By a *logical equivalence* of types X and Y, written $X \leftrightarrow Y$, we mean two functions $X \to Y$ and $Y \to X$ subject to no conditions at all.

By an *equivalence* of types X and Y we mean a function $e : X \to Y$ that has both a left and a right inverse, i.e. functions $s, r : Y \to X$ with $e(s(y)) = y$ for all $y : Y$ and $r(e(x)) = x$ for all $x : X$. This notion of equivalence is logically equivalent to having a *single* two-sided inverse, which is all that we will need in this paper. But the notion of equivalence is better-behaved in univalent foundations (see [14, Chapter 4]); the reason is that the type expressing "being an equivalence" is a proposition, in the presence of function extensionality, whereas the type expressing "having a two-sided inverse" may in general have more than one inhabitant, in particular affecting the consistency of the univalence axiom.

2 Classical axioms from non-parametricity

In this section, we give a number of ways in which classical axioms can be derived from specific violations of parametricity.

2.1 Polymorphic endomaps

Say that a function $f : \prod_{X:\mathcal{U}} X \to X$ is *natural under equivalence* if for any two types X and Y and any equivalence $e : X \to Y$, we have $e(f_X(x)) = f_Y(e(x))$ for any $x : X$, where we have written f_X as a shorthand for $f(X)$ and used the equality sign $=$ to denote identity types.

▶ **Theorem 1.** *If there is a function $f : \prod_{X:\mathcal{U}} X \to X$ such that $f_\mathbf{2}$ is not pointwise equal to the identity (i.e. $\neg \prod_{x:\mathbf{2}} f_\mathbf{2}(x) = x$) and f is natural under equivalence, then the law of excluded middle holds. Assuming function extensionality, the converse also holds.*

Proof. First we derive excluded middle from f. To begin, note that if $\neg \prod_{x:\mathbf{2}} f_\mathbf{2}(x) = x$, then we cannot have both $f_\mathbf{2}(\mathtt{tt}) = \mathtt{tt}$ and $f_\mathbf{2}(\mathtt{ff}) = \mathtt{ff}$, since then we could prove $\prod_{x:\mathbf{2}} f_\mathbf{2}(x) = x$ by case analysis on x. But then by case analysis on $f_\mathbf{2}(\mathtt{tt})$ and $f_\mathbf{2}(\mathtt{ff})$, we must have $(f_\mathbf{2}(\mathtt{tt}) = \mathtt{ff}) + (f_\mathbf{2}(\mathtt{ff}) = \mathtt{tt})$. Without loss of generality, suppose $f_\mathbf{2}(\mathtt{tt}) = \mathtt{ff}$.

Now let P be an arbitrary proposition. We do case analysis on $f_{P+\mathbf{1}}(\mathrm{inr}(\star)) : P + \mathbf{1}$.

1. If it is of the form $\mathrm{inl}(p)$ with $p : P$, we conclude immediately that P holds.
2. If it is of the form $\mathrm{inr}(\star)$, then P cannot hold, for if we had $p : P$, then the map $e : \mathbf{2} \to P + \mathbf{1}$ defined by $e(\mathtt{ff}) = \mathrm{inl}(p)$ and $e(\mathtt{tt}) = \mathrm{inr}(\star)$ would be an equivalence, and hence $e(f_\mathbf{2}(x)) = f_{P+\mathbf{1}}(e(x))$ for all $x : \mathbf{2}$ and so $\mathrm{inl}(p) = e(\mathtt{ff}) = e(f_\mathbf{2}(\mathtt{tt})) = f_{P+\mathbf{1}}(e(\mathtt{tt})) = f_{P+\mathbf{1}}(\mathrm{inr}(\star)) = \mathrm{inr}(\star)$, which is a contradiction.

Therefore P or not P.

For the converse, [14, Exercise 6.9], suppose excluded middle holds, let $X : \mathcal{U}$ and $x : X$, and consider the type $\sum_{x':X}(x' \neq x)$, where $a \neq b$ means $\neg(a = b)$. By excluded middle, this is either contractible or not. (A type Y is *contractible* if $\sum_{y:Y} \prod_{y':Y}(y = y')$. Assuming function extensionality, this is a proposition.) If it is contractible, define $f_X(x)$ to be the center of contraction (the point y in the definition of contractibility); otherwise define $f_X(x) = x$. ◀

▶ Remark.

1. If we assume univalence, any $f : \prod_{X:\mathcal{U}} X \to X$ is automatically natural under equivalence, so that assumption can be dispensed with. And, of course, if function extensionality holds (which follows from univalence) then the hypothesis $\neg \prod_{x:\mathbf{2}} f_\mathbf{2}(x) = x$ is equivalent to $f_\mathbf{2} \neq \lambda(x : \mathbf{2}).x$.
2. We do not know whether the converse direction of Theorem 1 is provable without function extensionality.

The preceding proof can be generalized as follows. We say that a point $x : X$ is *isolated* if the type $x = y$ is decidable for all $y : Y$, i.e. if we have $\prod_{y:X}(x = y) + (x \neq y)$.

▶ **Lemma 2.** *A point $x : X$ is isolated if and only if X is equivalent to $Y + \mathbf{1}$, for some type Y, by a map that sends x to $\mathrm{inr}(\star)$.*

Proof. Since $\mathrm{inr}(\star)$ is isolated, such an equivalence certainly implies that x is isolated. Conversely, from $\prod_{y:X}(x = y) + (x \neq y)$ we can construct a function $d : X \to \mathbf{2}$ such that $d(y) = \mathtt{tt}$ if and only if $x = y$ and $d(y) = \mathtt{ff}$, and if and only if $x \neq y$. Let Y be $\sum_{y:X}(d(y) = \mathtt{ff})$; it is straightforward to show $X \simeq Y + \mathbf{1}$.

If we had function extensionality (for $\mathbf{0}$-valued functions), we could dispense with d and define $Y = \sum_{y:X}(x \neq y)$, since then $x \neq y$ would be a proposition. In general we use $d(y) = \mathtt{ff}$ as it is always a proposition (since $\mathbf{2}$ has decidable equality, hence its identity types are propositions by Hedberg's theorem); this is necessary to show that the composite $Y + \mathbf{1} \to X \to Y + \mathbf{1}$ acts as the identity on Y. ◀

▶ **Theorem 3.** *If there is a function $f : \prod_{X:\mathcal{U}} X \to X$ such that $f_X(x) \neq x$ for some isolated point $x : X$ and f is natural under equivalence, then the law of excluded middle holds. Assuming function extensionality, the converse also holds.*

Proof. To derive excluded middle from f, let Y and $X \simeq Y + \mathbf{1}$ be as in Lemma 2, and let P be an arbitrary proposition. We do case analysis on $f_{P\times Y+\mathbf{1}}(\mathrm{inr}(\star)) : P \times Y + \mathbf{1}$.

1. If it is of the form $\mathrm{inl}((p, y))$ with $p : P$, we conclude immediately that P holds.
2. If it is of the form $\mathrm{inr}(\star)$, then P cannot hold, for if we had $p : P$, then the map $e : X \to P \times Y + \mathbf{1}$ defined by $e(x) = \mathrm{inr}(\star)$ (where x is the isolated point) and $e(y) = \mathrm{inl}((p, y))$ for $y \neq x$ would be an equivalence, and hence $e(f_X(x)) = f_{P\times Y+\mathbf{1}}(e(x))$, and so $\mathrm{inl}((p, f_X(x))) = e(f_X(x)) = f_{P\times Y+\mathbf{1}}(e(x)) = f_{P\times Y+\mathbf{1}}(\mathrm{inr}(\star)) = \mathrm{inr}(\star)$, which is a contradiction.

Therefore either P or not P holds. The converse is proven exactly as in Theorem 1. ◀

Finally, if our type theory includes propositional truncations, we can dispense with isolatedness.

▶ **Theorem 4.** *In a type theory with propositional truncations, there is an equivalence-natural function $f : \prod_{X:\mathcal{U}} X \to X$ and a type $X : \mathcal{U}$ with a point $x : X$ such that $f_X(x) \neq x$ if and only if excluded middle holds.*

Proof. For the "if" direction, note that propositional truncation implies function extensionality [9], so the converse direction of Theorem 1 applies. For the "only if" direction, assume that we are given $f : \prod_{X:\mathcal{U}} X \to X$, a type $X : \mathcal{U}$ and a point $x : X$ with $f_X(x) \neq x$. Let P be any proposition, and define

$$Z = \sum_{y:X} \|x = y\| \vee P, \qquad z = (x, |\mathrm{inl}(|\,\mathrm{refl}_x\,|)|) : Z, \qquad y = \mathrm{pr}_1(f_Z(z)) : X.$$

Recall that $A \vee B$ denotes the truncated disjunction $\|A + B\|$. This binds more tightly than Σ, so $Z = \sum_{y:X}(\|x = y\| \vee P)$. We write $|a| : \|A\|$ for the witness induced by a point $a : A$.

Now the second projection $\mathrm{pr}_2(f_Z(z))$ tells us that $\|x = y\| \vee P$. However, if P holds, then $\mathrm{pr}_1 : Z \to X$ is an equivalence that maps z to x. Thus $f_Z(z) \neq z$ and hence $x \neq y$. In other words, $P \to (x \neq y)$, hence $(x = y) \to \neg P$ and so also $\|x = y\| \to \neg P$. But since $\|x = y\| \vee P$, we have $\neg P \vee P$, which (in the presence of function extensionality) is equivalent to excluded middle. ◀

▶ Remark.

1. If $x : X$ happens to be isolated, then the type Z defined in the proof of Theorem 4 is equivalent to the type $P \times Y + \mathbf{1}$ used in the proof of Theorem 3.
2. Since propositional truncation implies function extensionality [9], it makes excluded middle into a proposition. Thus, the existence hypothesis of Theorem 4 can be truncated or untruncated without change of meaning.
3. The hypothesis can also be formulated as "there is a type X such that f_X is apart from the identity of X", where two functions $g, h : A \to B$ of types A and B are *apart* if there is $a : A$ with $g(a) \neq h(a)$. We don't know whether it is possible to derive excluded middle from the weaker assumption that f_X is simply *unequal* to the identity function of X, or even that f is unequal to the polymorphic identity function.

The above can be applied to obtain classical axioms from other kinds of violations of parametricity. As a simple example, consider $f : \prod_{X:\mathcal{U}}(X \to X) \to (X \to X)$. Parametric elements of this type are Church numerals. Given f, we can define a polymorphic endomap $g : \prod_{X:\mathcal{U}} X \to X$ by $g_X = f_X(\mathrm{id}_X)$, where id_X is the identity function. If f is natural under equivalence, then so is g, and hence the assumption that $f_{\mathbf{2}}(\mathrm{id}_{\mathbf{2}})$ is *not* the identity function gives excluded middle, assuming function extensionality.

2.2 Maps of the universe into the booleans

A function $f : \mathcal{U} \to \mathbf{2}$ is *invariant under equivalence*, or *extensional*, if we have $f(X) = f(Y)$ for any two equivalent types X and Y. We say that it is *strongly non-constant* if we have $X, Y : \mathcal{U}$ with $f(X) \neq f(Y)$. Assuming function extensionality, Escardó and Streicher [5, Thm. 2.2] showed that if $f : \mathcal{U} \to \mathbf{2}$ is extensional and strongly non-constant, then the weak limited principle of omniscience holds (any function $\mathbb{N} \to \mathbf{2}$ is constant or not). Alex Simpson strengthened this as follows (also reported in [5, Thm. 2.8]):

▶ **Theorem 5** (Simpson). *Assuming function extensionality for $\mathbf{0}$-valued functions, there is an extensional, strongly non-constant function $f : \mathcal{U} \to \mathbf{2}$ if and only if weak excluded middle holds (meaning that $\neg A + \neg\neg A$ for all $A : \mathcal{U}$).*

Proof. In one direction, suppose weak excluded middle, and define $f : \mathcal{U} \to \mathbf{2}$ by $f(A) = \mathtt{ff}$ if $\neg A$ and $f(A) = \mathtt{tt}$ if $\neg\neg A$. Then $f(\mathbf{0}) = \mathtt{ff}$ and $f(\mathbf{1}) = \mathtt{tt}$, so f is strongly non-constant. Extensionality follows from the observation that if $A \simeq B$ then $\neg A \leftrightarrow \neg B$ and $\neg\neg A \leftrightarrow \neg\neg B$.

In the other direction, suppose $f : \mathcal{U} \to \mathbf{2}$ is extensional, and strongly non-constant witnessed by types $X, Y : \mathcal{U}$ with $f(X) \neq f(Y)$. Suppose without loss of generality that $f(X) = \mathtt{tt}$ and $f(Y) = \mathtt{ff}$. For any $A : \mathcal{U}$, define $Z = \neg A \times X + \neg\neg A \times Y$. If A, then $\neg A \simeq \mathbf{0}$ and $\neg\neg A \simeq \mathbf{1}$ (using function extensionality), so $Z \simeq Y$ and $f(Z) = \mathtt{ff}$. Similarly, if $\neg A$, then $Z \simeq X$ and so $f(Z) = \mathtt{tt}$. On the other hand, $f(Z)$ must be either $\mathtt{tt}$ or $\mathtt{ff}$ and not both. If it is $\mathtt{tt}$, then it is not $\mathtt{ff}$, and so $\neg A$; while if it is $\mathtt{ff}$, then it is not $\mathtt{tt}$, and so $\neg\neg A$. ◀

In Theorem 6 below we reuse Simpson's argument to establish a similar conclusion for polymorphic functions into the booleans.

2.3 Polymorphic maps into the booleans

A function $f : \prod_{X:\mathcal{U}} X \to \mathbf{2}$ is *invariant under equivalence* if we have $f_Y(e(x)) = f_X(x)$ for any equivalence $e : X \to Y$ and point $x : X$. Such a function "violates parametricity" if it is non-constant. Equivalence invariance means that some such violations are literally impossible:

for instance, there cannot be a type X with points $x, y : X$ such that $f_X(x) \neq f_X(y)$ if there is an automorphism of X that maps x to y.

A violation of constancy *across* types, rather than at a specific type, is equivalent to weak excluded middle.

▶ **Theorem 6.** *Assuming function extensionality for* $\mathbf{0}$*-valued functions, weak excluded middle holds if and only if there is an* $f : \prod_{X:\mathcal{U}} X \to \mathbf{2}$ *that is invariant under equivalence, together with* $X, Y : \mathcal{U}$ *with isolated points* $x : X$ *and* $y : Y$ *such that* $f_X(x) \neq f_Y(y)$*.*

Proof. Assuming weak excluded middle, to show the existence of such an f, let $X : \mathcal{U}$ and $x : X$. Then use weak excluded middle to decide $\neg(\sum_{x':X} x \neq x') + \neg\neg(\sum_{x':X} x \neq x')$. In the left case, expressing that there are no other elements in X than x, define $f_X(x) = \mathrm{ff}$, and in the right case define $f_X(x) = \mathrm{tt}$. So, for example, $f_{\mathbf{1}}(\star) = \mathrm{ff}$ and $f_{\mathbf{2}}(\mathrm{tt}) = \mathrm{tt}$, showing that we constructed a non-constant f as required.

For the other direction, without loss of generality, $f_X(x) = \mathrm{tt}$ and $f_Y(y) = \mathrm{ff}$. By assumption, X is equivalent to $\mathbf{1} + X'$ via an equivalence that sends x to $\mathrm{inl}(\star)$, and similarly Y is equivalent to $\mathbf{1} + Y'$ via an equivalence that sends y to $\mathrm{inl}(\star)$. Let $A : \mathcal{U}$ and define

$$\begin{aligned} Z &= (\mathbf{1} + \neg A \times X') \times (\mathbf{1} + \neg\neg A \times Y'), \\ z &= (\mathrm{inl}(\star), \mathrm{inl}(\star)). \end{aligned}$$

By the invariance under equivalence of f,
1. if $\neg A$ then $Z \simeq X$ via an equivalence that sends z to x, thus $f_Z(z) = \mathrm{tt}$,
2. if A then $Z \simeq Y$ via an equivalence that sends z to y, thus $f_Z(z) = \mathrm{ff}$.

The contrapositives of these two implications are respectively

$$\begin{aligned} f_Z(z) = \mathrm{ff} &\to \neg\neg A, \\ f_Z(z) = \mathrm{tt} &\to \neg A. \end{aligned}$$

Hence we can decide $\neg A$ by case analysis on the value of $f_Z(z)$. ◀

Provided our type theory includes propositional truncations, we can dispense with isolatedness as in Theorem 4, assuming the types $x = x$ and $y = y$ are propositions.

▶ **Theorem 7.** *In a type theory with propositional truncations, weak excluded middle holds if and only if there is an* $f : \prod_{X:\mathcal{U}} X \to \mathbf{2}$ *that is invariant under equivalence, together with* $X, Y : \mathcal{U}$ *with* $x : X$ *and* $y : Y$ *such that* $f_X(x) \neq f_Y(y)$*, where the types* $x = x$ *and* $y = y$ *are propositions.*

Proof. Assuming weak excluded middle, the existence of such an f is shown as in the proof of Theorem 6.

For the other direction, without loss of generality, $f_X(x) = \mathrm{tt}$ and $f_Y(y) = \mathrm{ff}$. Note that since $x = x$ and $y = y$ are propositions, so are $x = x'$ and $y = y'$ for any $x' : X$ and $y' : Y$, since as soon as they have a point they are equivalent to $x = x$ and $y = y$ respectively. Let $A : \mathcal{U}$ and define

$$\begin{aligned} Z &= \left(\sum_{x':X} (x = x') \vee \neg A\right) \times \left(\sum_{y':Y} (y = y') \vee \neg\neg A\right), \\ z &= ((x, |\mathrm{inl}(\mathrm{refl})|), (y, |\mathrm{inl}(\mathrm{refl})|)). \end{aligned}$$

By invariance under equivalence of f, we have the following.

1. If $\neg A$ then $Z \simeq X$ via an equivalence that sends z to x, thus $f_Z(z) = \mathtt{tt}$. This works because the left factor of Z becomes equivalent to X, and the right factor equivalent to $\mathbf{1}$ by the assumptions that $y = y$ is a proposition and $\neg A$.
2. Similarly, if A then $Z \simeq Y$ via an equivalence that sends z to y, thus $f_Z(z) = \mathtt{ff}$, now using the fact that $x = x$ is a proposition.

The contrapositives of these two implications are respectively

$$\begin{aligned} f_Z(z) = \mathtt{ff} \quad &\to \quad \neg\neg A, \\ f_Z(z) = \mathtt{tt} \quad &\to \quad \neg A. \end{aligned}$$

Hence we can decide $\neg A$ by case analysis on the value of $f_Z(z)$. ◀

▶ Remark. In a type theory with pushouts, the assumptions that $x = x$ and $y = y$ are propositions can be removed by using the join $(x = x') * \neg A$ instead of the disjunction $(x = x') \vee \neg A$ in the left factor of Z, and similarly for the right factor of Z. (The *join* $B * C$ of types B and C is the pushout of B and C under $B \times C$.) This works since joining with an empty type is the identity, while joining with a contractible type gives a contractible result; see Theorem 9 below for details. Indeed, the join of two propositions is their disjunction, by [13, Lemma 2.4]; but the version using joins does not quite subsume the one using disjunctions, since if joins are not already assumed to exist, we do not know how to show that the disjunction of two propositions is their join.

2.4 Decompositions of the universe

Theorem 5 can be interpreted as saying that the universe $\mathcal{U}$ cannot be decomposed into two disjoint inhabited parts without weak excluded middle. In fact, disjointness of the parts is not necessary. All that is needed is that both parts be proper, i.e. not the whole of $\mathcal{U}$:

▶ **Theorem 8.** *In a type theory with propositional truncation and function extensionality for $\mathbf{0}$-valued functions, suppose we have equivalence-invariant $P, Q : \mathcal{U} \to \mathcal{U}$ such that for all $Z : \mathcal{U}$ we have $P(Z) \vee Q(Z)$, and that we have types X and Y such that $\neg P(X)$ and $\neg Q(Y)$. Then weak excluded middle holds.*

Proof. For any $A : \mathcal{U}$, let $Z = \neg A \times X + \neg\neg A \times Y$ as in Simpson's proof. If A, then $Z \simeq Y$, and so $\neg Q(Z)$; thus $Q(Z) \to \neg A$. But if $\neg A$, then $Z \simeq X$, and so $\neg P(Z)$; thus $P(Z) \to \neg\neg A$. Hence the assumed $P(Z) \vee Q(Z)$ implies $\neg A \vee \neg\neg A$, which is equivalent to $\neg A + \neg\neg A$ since $\neg A$ and $\neg\neg A$ are (by function extensionality) disjoint propositions. ◀

The proof of Theorem 7 can be similarly adapted.

▶ **Theorem 9.** *In a type theory with propositional truncation and $\mathbf{0}$-valued function extensionality, suppose we have $P, Q : \prod_{X:\mathcal{U}} X \to \mathcal{U}$ that are invariant under equivalence, i.e. if $X \simeq Y$ by an equivalence sending $x : X$ to $y : Y$, then $P_X(x) \simeq P_Y(y)$, and likewise for Q. Suppose also that for all $Z : \mathcal{U}$ and $z : Z$ we have $P_Z(z) \vee Q_Z(z)$, and types X, Y with points $x : X$ and $y : Y$ such that $\neg P_X(x)$ and $\neg Q_Y(y)$. Finally, suppose either that our type theory has pushouts or that the types $x = x$ and $y = y$ are propositions. Then weak excluded middle holds.*

Proof. For variety in contrast to Theorem 7, suppose we have pushouts; we leave the other case to the reader. Let $A : \mathcal{U}$ and define

$$\begin{aligned} Z &= \left(\sum_{x':X} (x = x') * \neg A \right) \times \left(\sum_{y':Y} (y = y') * \neg\neg A \right), \\ z &= ((x, \mathrm{inl}(\mathrm{refl})), (y, \mathrm{inl}(\mathrm{refl}))). \end{aligned}$$

Then if A, $\neg A \simeq \mathbf{0}$, so $(x = x') * \neg A \simeq (x = x')$, and thus the first factor of Z is equivalent to $\sum_{x':X}(x = x')$, which is a "singleton" or "based path space" and hence equivalent to $\mathbf{1}$. On the other hand (still assuming A), $\neg\neg A \simeq \mathbf{1}$, so $(y = y') * \neg\neg A \simeq \mathbf{1}$, and thus the right factor of Z is equivalent to $\sum_{y':Y} \mathbf{1}$ and hence to Y. Thus, A implies $Z \simeq Y$, and it is easy to check that this equivalence sends z to y. Hence $A \to \neg Q_Z(z)$, and so $Q_Z(z) \to \neg A$. A dual argument shows that $\neg A \to \neg P_Z(z)$ and thus $P_Z(z) \to \neg\neg A$, so the assumption $P_Z(z) \vee Q_Z(z)$ gives weak excluded middle. ◀

Since a function $\prod_{X:\mathcal{U}} X \to B$, for any fixed B, is the same as a function $(\sum_{X:\mathcal{U}} X) \to B$, we can interpret Theorem 9 as saying that the universe $\sum_{X:\mathcal{U}} X$ of *pointed types* also cannot be decomposed into two proper parts without weak excluded middle.

The results discussed so far illustrate that different violations of parametricity have different proof-theoretic strength: some violations are impossible, while others imply varying amounts of excluded middle.

3 Classical axioms from automorphisms of the universe

There have been attempts to apply parametricity to show that the only automorphism of a universe of types is the identity. Nicolai Kraus observed in the HoTT mailing list [8] that, assuming univalence, automorphisms of a universe $\mathcal{U}$ living in a universe $\mathcal{V}$ correspond to elements of the loop space[2] $\Omega(\mathcal{V}, \mathcal{U})$, while elements of the higher loop space $\Omega^2(\mathcal{V}, \mathcal{U})$ correspond to "polymorphic automorphisms" $\prod_{X:\mathcal{U}} X \simeq X$, which are at least as strong as polymorphic endomaps. In particular, nontrivial elements of $\Omega^2(\mathcal{V}, \mathcal{U})$ imply violations of parametricity for $\prod_{X:\mathcal{U}} X \to X$. This suggests that parametricity may play a role in automorphisms of the universe.

We are not aware of a proof that parametricity implies that the only automorphism of the universe is the identity. However, in the spirit of the above development, we can show that automorphisms with specific properties imply excluded middle. First, however, we observe that if we do have excluded middle then we can construct various nontrivial automorphisms of the universe.

3.1 Automorphisms from excluded middle

The simplest automorphism of the universe is defined as follows. By *propositional extensionality* we mean that any two logically equivalent propositions are equal. (This follows from propositional univalence, i.e. univalence asserted only for propositions. The converse holds at least assuming function extensionality; we do not know whether this assumption is necessary.)

[2] The *loop space* $\Omega(X, x)$ of a type X at a point $x : X$ is the identity type $x = x$; see [14, §2.1].

▶ **Theorem 10.** *Assuming excluded middle, function extensionality, and propositional extensionality, there is an automorphism* $f : \mathcal{U} \simeq \mathcal{U}$ *such that* $f(\mathbf{1}) \simeq \mathbf{0}$.

Proof. Given a type X, we use excluded middle to decide if it is a proposition (this works because under function extensionality, being a proposition is itself a proposition). If it is, we define $f(X) = \neg X$, and otherwise we define $f(X) = X$. Assuming propositional extensionality and excluded middle, we have $\neg\neg X = X$ for any proposition; thus $f(f(X)) = X$ whether X is a proposition or not, and hence f is a self-inverse equivalence. ◀

We can try to construct other automorphisms of the universe by permuting some other subclass of types. For instance, if we have propositional truncation, then given any two non-equivalent types A and B, excluded middle implies that for any type X we have $\|X = A\| + \|X = B\| + (X \neq A \wedge X \neq B)$, so that the universe $\mathcal{U}$ decomposes as a sum $\mathcal{U}_A + \mathcal{U}_B + \mathcal{U}_{\neq A,B}$, where

$$\mathcal{U}_A = \sum_{X:\mathcal{U}} \|X = A\|\,, \qquad \mathcal{U}_B = \sum_{X:\mathcal{U}} \|X = B\|\,, \qquad \mathcal{U}_{\neq(A,B)} = \sum_{X:\mathcal{U}} (X \neq A \wedge X \neq B).$$

(This requires function extensionality for $X \neq A$ and $X \neq B$ to be propositions, but not univalence.) Thus, if $\mathcal{U}_A \simeq \mathcal{U}_B$ we can switch those two summands to produce an automorphism of $\mathcal{U}$:

▶ **Theorem 11.** *Assuming function extensionality and excluded middle, if* $A \not\simeq B$ *and* $\mathcal{U}_A \simeq \mathcal{U}_B$*, then there is an automorphism* $f : \mathcal{U} \simeq \mathcal{U}$ *such that* $\|f(A) = B\|$*, hence* $f \neq \mathrm{id}$.

Proof. We use the above decomposition and the given equivalence $\mathcal{U}_A \simeq \mathcal{U}_B$ to produce f. And since f maps $\mathcal{U}_A$ to $\mathcal{U}_B$, by definition of $\mathcal{U}_B$ we have $\|f(A) = B\|$. ◀

This leads to the question, when can we have $\mathcal{U}_A \simeq \mathcal{U}_B$ but $A \not\simeq B$? Theorem 10 is the simplest example of this: assuming propositional extensionality, both $\mathcal{U}_{\mathbf{0}}$ and $\mathcal{U}_{\mathbf{1}}$ are contractible, hence equivalent to $\mathbf{1}$. More generally, let us call a type X *rigid* if $\mathcal{U}_X$ is contractible; then we have:

▶ **Theorem 12.** *Assuming function extensionality and excluded middle, if* A *and* B *are rigid types with* $A \not\simeq B$*, then there is an automorphism* $f : \mathcal{U} \simeq \mathcal{U}$ *such that* $f(A) \simeq B$.

Proof. This follows from Theorem 11. In the rigid case we get the stronger conclusion that $f(A) \simeq B$, since $\mathcal{U}_B$ is contractible. ◀

More generally, under excluded middle any permutation of the rigid types yields an automorphism of the universe.

If we assume UIP, then *every* type is rigid, so that with UIP and excluded middle there are plenty of automorphisms of the universe. If we instead assume univalence – as we will do for the rest of this subsection – most types are not rigid. For instance, any type with two distinct isolated points, such as $\mathbb{N}$, is not rigid, since we can swap the isolated points to give a nontrivial automorphism and hence a nontrivial equality in $\mathcal{U}_X$. In particular, if excluded middle holds and X is a *set* (i.e. its identity types are all propositions), then all points of X are isolated. Thus, with excluded middle and univalence, no set with more than one element (i.e. with points $x, y : X$ such that $x \neq y$) is rigid.

However, there exist types that are *connected* (i.e. $\|X\|$ and $\prod_{x,y:X} \|x = y\|$), but that are not trivial; indeed, as remarked above, $\mathcal{U}_A$ is such a type. Moreover, if we also assume higher inductive types, then from any group G that is a set we can construct a connected type BG such that $\Omega(BG) \simeq G$ [10, §3.2].

This leads us to ask, when is BG rigid for a set-group G? Since BG is a 1-type (i.e. its identity types are all sets), $\mathcal{U}_{BG}$ is a 2-type (i.e. its identity types are all 1-types). Hence it is contractible as soon as its loop space is connected and its double loop space is contractible. In general, the connected components of $\Omega(\mathcal{U}_{BG})$ are the *outer automorphisms* of G (equivalence classes of automorphisms of G modulo conjugation), while $\Omega^2(\mathcal{U}_{BG})$ is the *center* of G (the subgroup of elements that commute with everything). A group with trivial outer automorphism group and trivial center is sometimes known as a *complete* group (though there is no apparent relation to any topological notion of completeness), and there are plenty of examples.

For instance, the symmetric group S_n is complete in this sense except when $n = 2$ or 6. Thus, BS_n is rigid for $n \notin \{2, 6\}$. (Note also that BS_n can be constructed without higher inductive types – but with univalence – as $\mathcal{U}_{[n]}$, where $[n]$ is a finite n-element type, although of course this type only lives in a larger universe $\mathcal{V}$.) In particular, assuming univalence and excluded middle, there are countably infinitely many rigid types, and hence *uncountably* many nontrivial automorphisms of $\mathcal{U}$ (one induced by every permutation of the types BS_n for $n \notin \{2, 6\}$).

This does not exhaust the potential automorphisms of $\mathcal{U}$. For instance, we have:

▶ **Theorem 13.** *Let X be an n-type for some $n \geq -1$, and let A and B be n-connected rigid types such that $X \times A \not\simeq X \times B$. Then assuming univalence and excluded middle, there is an automorphism $f : \mathcal{U} \simeq \mathcal{U}$ such that $\|f(X \times A) = (X \times B)\|$.*

Proof. We will show that $\mathcal{U}_{X\times A} \simeq \mathcal{U}_{X\times B}$, by showing that both are equivalent to $\mathcal{U}_X$. It suffices to consider A. We have $(Z \mapsto Z \times A) : \mathcal{U}_X \to \mathcal{U}_{X\times A}$, and since both types are connected it suffices to show that it induces an equivalence of loop spaces $\Omega\mathcal{U}_X \to \Omega\mathcal{U}_{X\times A}$, or equivalently that the induced map $L : (X \simeq X) \to (X \times A \simeq X \times A)$ is an equivalence. Since A is n-connected for $n \geq -1$, we have $\|A\|$; so since being an equivalence is a proposition we may assume given $a_0 : A$.

We claim that for all $a : A$, $x : X$, and $f : X \times A \to X \times A$ we have

$$\mathrm{pr}_1(f(x, a)) = \mathrm{pr}_1(f(x, a_0)). \tag{1}$$

Since this goal is an equality in the n-type X, it is an $(n-1)$-type. And since A is n-connected, the map $a_0 : \mathbf{1} \to A$ is $(n-1)$-connected by [14, Lemma 7.5.11]. Thus, by [14, Lemma 7.5.7], it suffices to assume that $a = a_0$, in which case (1) is clear.

It follows from (1) that if we define $M : (X \times A \to X \times A) \to (X \to X)$ by $M(f)(x) = \mathrm{pr}_1(f(x, a_0))$, then M preserves composition and identities. Thus it preserves equivalences, inducing a map $(X \times A \simeq X \times A) \to (X \simeq X)$. We easily have $M \circ L = \mathrm{id}$, so to prove $L \circ M = \mathrm{id}$ it suffices to show that M is left-cancellable, i.e. that $(Mf = Mg) \to (f = g)$. Since M preserves composition, for this it suffices to show that if $Mf = \mathrm{id}$ then $f = \mathrm{id}$. But if $Mf = \mathrm{id}$, then by (1) we have $\mathrm{pr}_1(f(x, a)) = x$ for all $a : A$. Thus $f(x, a) = (x, g_x(a))$, where $g_x : A \simeq A$ for each $x : X$. But A is rigid, so each $g_x = \mathrm{id}$, hence $f = \mathrm{id}$. ◀

For instance, we could take $n = 0$ and $X = \mathbf{2}$, so that $X \times A \simeq A + A$. Thus if A and B are any connected rigid types, an automorphism of $\mathcal{U}$ can swap $A + A$ with $B + B$.

There might also be rigid types that are not of the form BG, or types A, B not built out of rigid ones but such that $A \not\simeq B$ and $\mathcal{U}_A \simeq \mathcal{U}_B$. But now we will leave such questions and turn to the converse: when does an automorphism of $\mathcal{U}$ imply excluded middle?

3.2 Excluded middle from automorphisms

In fact, without function extensionality, we can only derive a slightly weaker form of excluded middle from a nontrivial automorphism of the universe. As defined in the introduction, the *law of excluded middle* (LEM) is

$$\prod_{P:\mathcal{U}} \mathrm{isProp}(P) \to P + \neg P.$$

We will instead derive the law of *double-negation elimination* (DNE), which is

$$\prod_{P:\mathcal{U}} \mathrm{isProp}(P) \to \neg\neg P \to P.$$

Notice that if $\mathbf{0}$-valued function extensionality holds, then $\neg P$ is a proposition (even if P is not a proposition) and hence, if P is a proposition, $P + \neg P$ is a proposition equivalent to $P \vee \neg P$. In first-order or higher-order logic, the corresponding schemas or axioms of excluded middle and double-negation elimination are equivalent, but, in type theory, one direction seems to require some amount of function extensionality:

▶ **Lemma 14.**
1. LEM *implies* DNE.
2. DNE *implies* LEM *assuming* $\mathbf{0}$*-valued function extensionality.*

Proof. (1): Assume LEM and let $P : \mathcal{U}$ with isProp(P) and assume $\neg\neg P$. By excluded middle, either P or $\neg P$. In the first case we are done, and the second contradicts $\neg\neg P$.
(2): Assume DNE and let $P : \mathcal{U}$ with isProp(P). By $\mathbf{0}$-valued function extensionality, $P + \neg P$ is a proposition, and hence DNE gives $P + \neg P$, because we always have $\neg\neg(P + \neg P)$. ◀

▶ **Lemma 15.** DNE *holds if and only if every proposition is logically equivalent to the negation of some type.*

Proof. ($\Rightarrow$): DNE gives that any proposition P is logically equivalent to the negation of the type $\neg P$. ($\Leftarrow$): For any two types A and B, we have that $A \to B$ implies $\neg B \to \neg A$. Hence $A \to B$ also gives $\neg\neg A \to \neg\neg B$. And, because $X \to \neg\neg X$ for any type X, we have $\neg\neg\neg X \to \neg X$. Therefore, if P is logically equivalent to the negation of X, we have the chain of implications $\neg\neg P \to \neg\neg\neg X \to \neg X \to P$. ◀

Our first automorphism of the universe constructed from excluded middle swapped the empty type with the unit type. We now show that conversely, any such automorphism implies DNE and hence, assuming $\mathbf{0}$-valued function extensionality, also LEM. In fact, not even an *embedding* of $\mathcal{U}$ into itself that maps the unit type to the empty type is possible without classical axioms:

▶ **Theorem 16.** *Assuming propositional extensionality, if there is a left-cancellable map* $f : \mathcal{U} \to \mathcal{U}$ *with* $f(\mathbf{1}) = \mathbf{0}$*, then* DNE *holds.*

Proof. For an arbitrary proposition P, we have:

$$\begin{aligned}
P &\leftrightarrow P = \mathbf{1} && \text{(by propositional extensionality)}\\
&\leftrightarrow f(P) = f(\mathbf{1}) && \text{(because } f \text{ is left-cancellable)}\\
&\leftrightarrow f(P) = \mathbf{0} && \text{(by the assumption that } f(\mathbf{1}) = \mathbf{0})\\
&\leftrightarrow \neg f(P) && \text{(by propositional extensionality).}
\end{aligned}$$

(Note that if $\neg f(P)$, then $f(P) \leftrightarrow \mathbf{0}$, so $f(P)$ is a proposition and we can apply propositional extensionality to get $f(P) = \mathbf{0}$.) Hence P is logically equivalent to the negation of the type $f(P)$, and therefore Lemma 15 gives DNE. ◀

▶ **Corollary 17.** *Assuming propositional extensionality, if there is an automorphism of the universe that maps the unit type to the empty type, then* DNE *holds.*

Now let us further assume univalence and propositional truncations. This implies function extensionality, so the difference between DNE and LEM disappears. Furthermore, we can additionally generalize the result as follows. Say that a type A is *inhabited* if the unique map $A \to \mathbf{1}$ is surjective. This is equivalent to giving an element of the propositional truncation $\|A\|$.

▶ **Lemma 18.** *Assuming univalence and propositional truncations, if A is an inhabited type, then any proposition P is logically equivalent to the identity type $(P \times A) = A$.*

Proof. If P then $P \simeq \mathbf{1}$, so $(P \times A) \simeq A$, and hence by univalence $(P \times A) = A$. Conversely, assume $(P \times A) = A$. Then $\|P \times A\| = \|A\| = \mathbf{1}$ by univalence, as A is inhabited. So $\|P\| \times \|A\| = \mathbf{1}$, and hence $P = \mathbf{1}$. ◀

Using this, we can weaken the hypothesis of Lemma 16 to the requirement that f maps some inhabited type to the empty type, and get the same conclusion, at the expense of requiring univalence rather than just propositional extensionality:

▶ **Lemma 19.** *Assuming univalence and propositional truncations, if there is a left-cancellable map $f : \mathcal{U} \to \mathcal{U}$ with $f(A) = \mathbf{0}$ for some inhabited type A, then excluded middle holds.*

Proof. For an arbitrary proposition P, we have:

$$\begin{aligned}
P &\leftrightarrow (P \times A) = A && \text{(by Lemma 18)}\\
&\leftrightarrow f(P \times A) = f(A) && \text{(because } f \text{ is left-cancellable)}\\
&\leftrightarrow f(P \times A) = \mathbf{0} && \text{(by the assumption that } f(A) = \mathbf{0})\\
&\leftrightarrow \neg f(P \times A) && \text{(by propositional extensionality).}
\end{aligned}$$

Hence P is logically equivalent to the negation of the type $f(P \times A)$, and therefore Lemma 15 gives DNE. But univalence gives function extensionality, and hence Lemma 14 gives LEM. ◀

▶ **Theorem 20.** *Assuming univalence and propositional truncations, if there is an automorphism of the universe that maps some inhabited type to the empty type, then excluded middle holds.*

▶ **Corollary 21.** *Assuming univalence and propositional truncations, if there is an automorphism $g : \mathcal{U} \to \mathcal{U}$ of the universe with $g(\mathbf{0}) \neq \mathbf{0}$, then the double negation*

$$\neg\neg \prod_{P:\mathcal{U}} \mathrm{isProp}(P) \to P + \neg P$$

of the law of excluded middle holds.

(Note that this is not the same as

$$\prod_{P:\mathcal{U}} \mathrm{isProp}(P) \to \neg\neg(P + \neg P),$$

which is of course constructively valid without extra assumptions.)

Proof. Let f be the inverse of g. If $g(\mathbf{0})$ then $\|g(\mathbf{0})\|$, and because f maps $g(\mathbf{0})$ to $\mathbf{0}$, we conclude that excluded middle holds by Theorem 20. But the assumption $g(\mathbf{0}) \neq \mathbf{0}$ is equivalent to $\neg\neg g(\mathbf{0})$ by propositional extensionality, and so it implies the double negation of excluded middle. ◀

It is in general an open question for which X the existence of an automorphism $f : \mathcal{U} \to \mathcal{U}$ with $f(X) \neq X$ implies a non-provable consequence of excluded middle [4]. Not even for $X = \mathbf{1}$ do we know whether this is the case. However, the following two cases for X follow from the case $X = \mathbf{0}$ discussed above:

▶ **Corollary 22.** *Assuming univalence and propositional truncations, for universes $\mathcal{U} : \mathcal{V}$, if there is an automorphism $f : \mathcal{V} \to \mathcal{V}$ with $f(X) \neq X$ for $X = \mathrm{LEM}_{\mathcal{U}}$ or $X = \neg\neg\,\mathrm{LEM}_{\mathcal{U}}$, then $\neg\neg\,\mathrm{LEM}_{\mathcal{U}}$ holds.*

Proof. Suppose that $\neg\,\mathrm{LEM}_{\mathcal{U}}$, and hence $X = \mathbf{0}$. By Corollary 21, we obtain $\neg\neg\,\mathrm{LEM}_{\mathcal{V}}$, which implies $\neg\neg\,\mathrm{LEM}_{\mathcal{U}}$, contradicting the assumption. ◀

References

1 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, San Diego, CA, USA, January 20-21, 2014*, pages 503–516, 2014. `doi:10.1145/2535838.2535852`.

2 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012. `doi:10.1017/S0956796812000056`.

3 Auke Booij. Agda development for "parametricity, automorphisms of the universe, and excluded middle", June 2017. URL: `https://github.com/abooij/parametricityandlem-agda`.

4 Martín Hötzel Escardó. Automorphisms of U. Homotopy Type Theory mailing list, August 2014. URL: `https://groups.google.com/d/msg/homotopytypetheory/8CV0S2DuOI8/b1Co7x-B7aoJ`.

5 Martín Hötzel Escardó and Thomas Streicher. The intrinsic topology of Martin-Löf universes. *Ann. Pure Appl. Logic*, 167(9):794–805, 2016. `doi:10.1016/j.apal.2016.04.010`.

6 HoTT Project. The homotopy type theory Coq library. `http://github.com/HoTT/HoTT/`, 2015.

7 Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, pages 381–395, 2012. `doi:10.4230/LIPIcs.CSL.2012.381`.

8 Nicolai Kraus. Automorphisms of U. Homotopy Type Theory mailing list, August 2014. URL: `https://groups.google.com/d/msg/homotopytypetheory/8CV0S2DuOI8/Phqpk7aMR7cJ`.

9 Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of Anonymous Existence in Martin-Löf Type Theory. *Logical Methods in Computer Science*, 13(1), 2017. `doi:10.23638/LMCS-13(1:15)2017`.

10 Dan Licata and Eric Finster. Eilenberg–MacLane spaces in homotopy type theory. *Logic in Computer Science (LICS)*, 2014. URL: `http://dlicata.web.wesleyan.edu/pubs/lf14em/lf14em.pdf`.

11 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes.* Bibliopolis, Naples, 1984.

12 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

13 Egbert Rijke. The join construction. arXiv:1701.07538, 2017.

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

Coq Support in HAHA

Jacek Chrząszcz
University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
chrzaszc@mimuw.edu.pl

Aleksy Schubert
University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
alx@mimuw.edu.pl

Jakub Zakrzewski
University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
j.zakrzewski@mimuw.edu.pl

Abstract

HAHA is a tool that helps in teaching and learning Hoare logic. It is targeted at an introductory course on software verification. We present a set of new features of the HAHA verification environment that exploit Coq. These features are (1) generation of verification conditions in Coq so that they can be explored and proved interactively and (2) compilation of HAHA programs into CompCert certified compilation tool-chain.

With the interactive Coq proving support we obtain an interesting functionality that makes it possible to carefully examine step-by-step verification conditions and systematically discover flaws in their formulation. As a result Coq back-end serves as a kind of *specification debugger*.

2012 ACM Subject Classification Software and its engineering → Software verification, Theory of computation → Program verification

Keywords and phrases Hoare logic, program verification, Coq theorem prover, teaching

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.8

Funding This work was partially supported by the Polish NCN grant no 2013/11/B/ST6/01381.

1 Introduction

Mainstream imperative programming languages give programmers a platform in which they describe computation as a sequence of operations that transform the state of a computing machine (and interact with the outside world). The resulting programs, as produced by humans, may contain mistakes. A systematic approach to eliminate mistakes leads to an arrangement where one has to provide (at least approximately) the solution again, in a different way, and then confront the two solutions to check if they match well enough. In the mainstream software engineering this is achieved on the one hand by various requirement management frameworks and on the other hand by various testing methodologies.

Software verification techniques bring here an alternative paradigm, which is based on Hoare logic [18]. The different description of the software artefact consists here in giving explicit specification for invariant properties of states that hold between atomic instructions. The obvious advantage of this approach over testing is that a verified condition ensures correctness not limited to a finite base of available test cases, but for all allowed situations.

It is worth stressing that this possibility requires understanding of mechanisms that make it possible to generalise beyond the results of experiments that are directly available to our perception. However, this requires good command of additional theoretical constructions, which are complicated by themselves and require additional educational effort.

The experience of our faculty, most probably shared by other places, shows that typical student sees Hoare logic as tedious, boring, obscure and, above all, impractical. Therefore, it is important to counter this view by showing them the topic in an attractive modern way.

There is a wide variety of mature software verification tools (e.g. ESC/Java [10, 20], Frama-C [5, 12], Verifast [19], Microsoft VCC [13], KeY [6]) that have proven to be usable in the verification process of software projects. However their design is geared towards applications in large scale software development, rather than education, which results in automatising of various tasks at the cost of making the principles behind them difficult to understand. In particular, the base logic behind such systems, with automatic verification condition generation, handling of procedure call stack and heap, is much more complicated than the initial one designed by Hoare. This opens room for tools in which students are exposed to basic techniques only, and HAHA[1] verification environment fits into this scope.

HAHA was successfully applied in the curriculum of University of Warsaw [26]. So far it relied on the proving back-end of Z3 [15]. However, development of a Hoare logic proof with help of Z3 has limitations. In particular, the process of invariant writing is not systematic – in case the formulas are too weak or in some way incorrect, the process of problem discovery is based on guesswork. For some formulas Z3 is able to generate counterexamples, which helps in error correction. Still, the solver does not always find them.

This was our main reason to develop a verification condition generator that produces formulas in the format acceptable for an interactive theorem prover, in our case Coq. The interactive prover enables here the possibility to stepwise examine the verification condition assumptions so that complete understanding is obtained on how they are used to arrive at the prescribed conclusion. Such a stepwise examination is very similar to examination of a program operation with a debugger, but this time the main focus is not on the code of the program, but on another class of statements in an artificial language, namely specifications.

A successful application of this workflow requires two basic elements. First, the formulas should be generated in a comprehensive way that can be easily related to the point of the code they come from and students should be able to quickly start proof construction on their own. Second, most of the generated verification conditions should be discharged automatically. It is acceptable that only few of them are left for manual proof development. We show here how these goals were achieved.

In addition, HAHA can be considered as a small programming language. Therefore, it is crucial to be able to compile programs written in it. As the programs, once verified, are supposed to be highly dependable, we decided to connect compilation to a highly dependable compiler chain of CompCert [22]. As a result we obtained a basis for a miniature platform that enables development of highly dependable small programs.

The paper is structured as follows. Overview of HAHA and its features is in Section 2. Section 3 presents the translation of HAHA programs with assertions to Coq. It is followed in Section 4 by description of the Coq proof tactic to automate handling of most of the proof goals. Section 5 illustrates how Coq can be used to debug specifications. Translation of HAHA programs to CompCert languages is shown in Section 6. We conclude in Section 7.

2 Overview of HAHA

We present the features of HAHA tool and programming language by showing how a particular example procedure can be verified with its help. The code of the procedure is presented

[1] HAHA is available from the page `http://haha.mimuw.edu.pl`

```
function insert(A : ARRAY[Z], n : Z, e : Z) : ARRAY[Z]

  var j : Z
      stop : BOOLEAN
      B : ARRAY[Z]

begin
  B := A
  j := n−1
  stop := false
  while j >= 0 /\ not stop  do
  begin
    if (B[j] <= e) then begin
        stop := true
    end
    else begin
        B[j+1] := B[j]
        j := j−1
    end
  end
  B[j+1] := e
  insert := B
end
```

Figure 1 The insert procedure implemented in the HAHA programming language.

in Figure 1 and it describes the well known insert procedure that inserts an element into a sorted array, so that the array remains sorted after the operation.

The input language of HAHA is that of while programs over integers and arrays. The code of programs is packaged into functions, which serve as the basic structural component that is sufficient for code management in the small. The basic language constructs of HAHA are standard and similar to those of Pascal programming language so we omit its grammar. We only remark here that one bigger departure from the standard Pascal syntax is that we do not use semicolon to terminate statements. As a consequence one line can hold only one statement, which agrees with major coding style guidelines [17, 25]. This design choice brings some difficulties in applying standard LALR parser generators, but it has the effect of keeping the source code of programs more legible for humans. As for the semantics, we designed the language so that its mechanisms and datatypes match those supported by state of the art satisfiability solvers, e.g. Z3 [15] or CVC4 [1]. The main datatypes of the language are arbitrary precision integer numbers and unbounded arrays. This design choice makes it possible to postpone discussion on programming mistakes associated with strict keeping of the available ranges and integer bounds to other stages of instruction.

In our example the function *insert* from Figure 1 works as follows. It takes three input arguments, the array A to insert element to, the length n of its range filled with values of interest, and the value e to insert into the array. We assume that the input array is sorted and the function looks for the location into which the value e can be inserted by traversing the array in a loop from the index $n{-}1$ down to 0. In case the first element that is not greater than e is found, a boolean flag *stop* is set. Then the function inserts e in the found place so that the ordering of the resulting array is preserved.

The procedure cannot directly modify the input array.[2] Therefore, we have to copy its content to an array B and then operate on it. The array with inserted element e is returned as the result, which is realised by the final, typical for Pascal, assignment $insert\ := B$.

[2] We keep input array immutable to avoid more complicated binary verification conditions and be at the same time able to refer to the input array in the postcondition.

```
predicate sorted(A : ARRAY[Z], l : Z, h : Z) =
  forall i : Z, j : Z,  l <= i /\ i <= j /\ j < h -> A[i] <= A[j]

function insert(A : ARRAY[Z], n : Z, e : Z) : ARRAY[Z]
  precondition        n >= 0                   // length is non-negative
  precondition  sort: sorted(A, 0, n)          // array is sorted
  postcondition       sorted(insert, 0, n+1)   // the result is sorted
  var j : Z
      stop : BOOLEAN
      B : ARRAY[Z]
begin
  B := A
  { sorted(B, 0, n) /\ n >= 0 }
  j := n-1
  { sorted(B, 0, n) /\ j = n-1 /\ n >= 0 }
  stop := false
  { sorted(B, 0, n) /\ j = n-1 /\ n >= 0 /\ not stop }
  while j >= 0 /\ not stop  do
    invariant 0 <= j+1 /\ j+1 <= n
    invariant onSorting: sorted(B, 0, n) /\ (j+1 < n -> sorted(B, 0, n+1))
    invariant j+1 < n -> forall k : Z, j < k /\ k <= n -> e <= B[k]
    invariant stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] < e)
    invariant j+1 < n -> B[j+1] > e
  begin
    if (B[j] <= e) then begin
       stop := true
    end
    else begin
       B[j+1] := B[j]
       { 0 < j+1 /\ j+1 <= n }
       { strongerSorting: sorted(B, 0, n+1) }
       { forall k : Z, j < k /\ k <= n -> e <= B[k] }
       { stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] <= e) }
       { B[j] > e /\ B[j+1] = B[j] }
       j := j-1
    end
  end
  { (j+1 = 0 \/ stop) /\ 0 <= j+1 /\ j+1 <= n }
  { sorted(B, 0, n) /\ (j+1 < n -> sorted(B, 0, n+1)) }
  { stop -> forall k : Z, 0 <= k /\ k <= j -> B[k] < e }
  { j+1 < n -> B[j+1] > e }
  B[j+1] := e
  { sorted(B, 0, n+1) }
  insert := B
end
```

Figure 2 The insert procedure with the specifications.

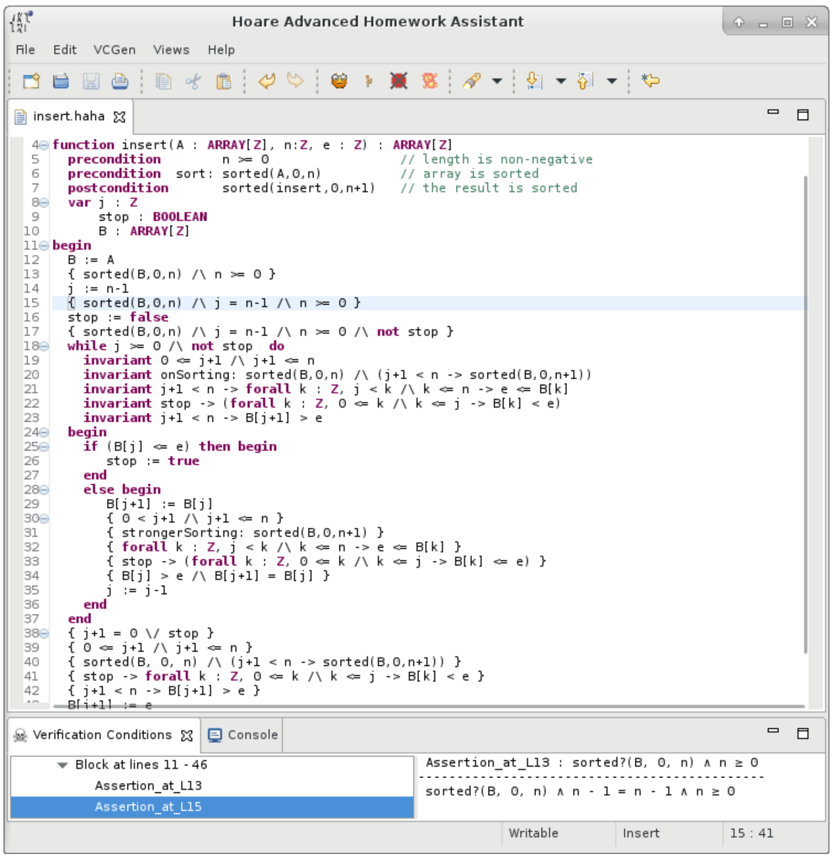
The HAHA environment does not accept directly a program written in the form presented in Figure 1. HAHA programs must contain all necessary specifications to be valid. The version of the program, which is accepted by the HAHA parser, is presented in Figure 2.

The code starts with additional definitions of predicates, which encapsulate under comprehensible identifiers essential properties that we deal with in description of the procedure states. We define there, in particular, the predicate *sorted*

```
predicate sorted(A : ARRAY[Z], l : Z, h : Z) =
   forall i : Z, j : Z,  l <= i /\ i <= j /\ j < h -> A[i] <= A[j]
```

As we can see, interpreting the expressions in natural way, the predicate holds true when the argument array A is ordered in the range $A[l], \ldots, A[h-1]$.

Another addition to the original code is a range of lines that describes the input-output property of the function. This area contains a number of properties of the input (marked with the keyword ***precondition***) and a property of the result (marked with ***postcondition***). The preconditions express that (1) the area of the values that are relevant for the procedure has at least one element, and (2) the array A is sorted in the range $A[1], \ldots, A[n-1]$. Note



```
 4  function insert(A : ARRAY[Z], n:Z, e : Z) : ARRAY[Z]
 5    precondition        n >= 0                 // length is non-negative
 6    precondition  sort: sorted(A,0,n)          // array is sorted
 7    postcondition       sorted(insert,0,n+1)   // the result is sorted
 8    var j : Z
 9        stop : BOOLEAN
10        B : ARRAY[Z]
11  begin
12    B := A
13    { sorted(B,0,n) /\ n >= 0 }
14    j := n-1
15    { sorted(B,0,n) /\ j = n-1 /\ n >= 0 }
16    stop := false
17    { sorted(B,0,n) /\ j = n-1 /\ n >= 0 /\ not stop }
18    while j >= 0 /\ not stop  do
19      invariant 0 <= j+1 /\ j+1 <= n
20      invariant onSorting: sorted(B,0,n) /\ (j+1 < n -> sorted(B,0,n+1))
21      invariant j+1 < n -> forall k : Z, j < k /\ k <= n -> e <= B[k]
22      invariant stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] < e)
23      invariant j+1 < n -> B[j+1] > e
24    begin
25      if (B[j] <= e) then begin
26         stop := true
27      end
28      else begin
29         B[j+1] := B[j]
30         { 0 < j+1 /\ j+1 <= n }
31         { strongerSorting: sorted(B,0,n+1) }
32         { forall k : Z, j < k /\ k <= n -> e <= B[k] }
33         { stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] <= e) }
34         { B[j] > e /\ B[j+1] = B[j] }
35         j := j-1
36      end
37    end
38    { j+1 = 0 \/ stop }
39    { 0 <= j+1 /\ j+1 <= n }
40    { sorted(B, 0, n) /\ (j+1 < n -> sorted(B,0,n+1)) }
41    { stop -> forall k : Z, 0 <= k /\ k <= j -> B[k] < e }
42    { j+1 < n -> B[j+1] > e }
      B[j+1] := e
```

```
▾ Block at lines 11 - 46
    Assertion_at_L13
    Assertion_at_L15
```

```
Assertion_at_L13 : sorted?(B, 0, n) ∧ n ≥ 0
-----------------------------------------
sorted?(B, 0, n) ∧ n - 1 = n - 1 ∧ n ≥ 0
```

Figure 3 The user interface of HAHA.

that we can name a precondition as it is the case with the second one in our code.

For educational purposes the language has an assertion mechanism that forces programmers to insert state descriptions between every two consecutive instructions of their code. We can see that the assertions are enclosed within curly brackets *{ ... }*. This makes it possible to stay very close to the original Hoare logic. In addition, programmers immediately see how big the relevant state they have to keep in mind is and observe its subtle changes along the code of their procedure. This feature forces them to explicate their understanding of programs, and as a result makes them aware of all the program details. To avoid literal repetition of assertions, we forbid assertions to occur at the beginning or at the end of a block. We can see this feature in our example by observing the lack of asserts in the loop body before the ***if*** keyword in line 25 in Figure 2. The asserts may be named as in line 31 and distributed in many statements enclosed in brackets to support structured reading of the properties (see, for example, the block of asserts between lines 30 and 34).

Loop invariants are a necessary element of any system based on Hoare logic. HAHA makes it possible to describe them through its ***invariant*** keyword, which can be used before the loop body to describe the constant property of the state at the entry to the loop, before the loop condition is checked. The textual location of the invariants is a little bit different than their reference point, but it remains in accordance with standard approach [2, 7]. Again the invariants can be named (see line 20) and understood separately, but they are combined into a conjunction when treated as a precondition for the body of the loop.

The way the procedure looks like in the user interface of HAHA is presented in Figure 3.

We can see there a window with the source code of the *insert* procedure. The editor shown there has features that are expected from a modern IDE, such as syntax highlighting, automated completion proposals and error markers. Once a program is entered, we can push one of buttons to start a verification condition generator, which implements the rules of Hoare logic. The resulting formulas are then passed to a proving back-end, which can be either Z3 or Coq, depending on the button pressed. If the solver is unable to ascertain the correctness of the program, error markers are generated to point the user to parts which could not be proven. A very useful feature of Z3 back-end is the ability to find counterexamples for incorrect assertions. These are included in error descriptions displayed by the editor.

We should remark here that to illustrate the way the Coq proof assistant can be used to debug specifications, we introduced in the specifications presented in Figure 2 a small mistake that is exploited hereafter in our explanations.

Axioms. In some cases the proving back-end is too weak to automatically handle some of the proof steps. This can be ameliorated by the use of axioms. A typical situation concerns axiomatisation of multiplication. We can for instance add an axiom

```
axiom multi: forall x : Z, (x + 1)*(x + 1) = x*x + 2*x + 1
```

which conveys some basic property of interest. Fortunately, our example program does not need axioms and actually they are not needed in case we want to use Coq as the proving back-end (although they can help to automatically prove a number of verification conditions).

3 Export to Coq

Motivations. There are several advantages of letting Coq work as the proving back-end in a tool such as HAHA. First of all, this gives the users one more option to choose, possibly one with which they can feel more comfortable. Second, SMT solvers such as Z3 always offer limited strength. This is partly mitigated by the presence of axiom construct in HAHA, but on the other hand axioms may be wrong and as a result they weaken the guarantees for the resulting program. Therefore, Coq back-end offers the possibility to prove arbitrarily difficult conditions in a way that guarantees strong confidence.

However, we would like to focus on another possibility the Coq back-end offers. One of the frequent problems in development of formal specifications in the small, i.e. in devising asserts and loop invariants within programmed functions or procedures, is that the formulas given by developers are not always good enough to carry out the proving process. There are at least three reasons for this situation. One is that the given invariant formulas are too weak to close the loop body verification effort. Another one is that a given formula has some error, which is not visible to the author. At last, the formulas may be strong enough, but they use a feature the proving back-end has difficulty to deal with (e.g. it has to make a proof by induction to exploit the information contained in the formula in a useful way).

The mentioned above weaknesses of specifications result from insufficient understanding of the algorithm details. The reason why the author gives a false specification and cannot immediately see the problem in it is that the person does not fully understand the situation in the particular location of the code. This calls for a process in which better understanding can be gained. Actually, going through an attempt to formally prove a property is a way to systematically find a gap in the understanding of the situation at hand.

Generation of verification conditions – general idea. The general idea of verification condition generation in HAHA is simple. Since asserts are given explicitly between all instructions, one has to directly apply the Hoare logic rules. However, the proof must be done in an appropriate context of defined symbols as well as predicates and the proving back-end must be supplied with this. We sketch below the main ideas behind this. The semantics of the translation is rather standard, but as the result must serve educational purposes we have to make it clear and easily accessible to the users. Therefore, we provide it here in the form close to the actual Coq text, including comments and spaces, to illustrate that the generated code can easily be related to the initial source and has a comprehensible form.

Prelude. The prelude of the Coq file contains declarations necessary to introduce the model of HAHA datatypes. We model base types of HAHA, i.e. integers (Z), and booleans ($BOOLEAN$) as Z and *bool* respectively. The combined type of arrays (***ARRAY***$[\tau]$) is modelled as functions from integers to the type of array elements ($Z \to \tau'$). In particular the type ***ARRAY***$[BOOLEAN]$ is modelled as the Coq type of functions $Z \to$ *bool*.

As a consequence the prelude in the Coq file for our example starts with library imports that introduce integer numbers to model the type Z and the domains of arrays. We also import booleans to model conditions in ***if*** and ***while*** instructions. At last, the file with our set of tactics and notations is imported.

```
Require Export ZArith ZArith.Int ZArith.Zbool.
Local Open Scope Z_scope.
Load "tactics.v".
```

The file `tactics.v` contains not only the definitions of tactics, which are described further here in Section 4, but also a basic support for our handling of booleans and arrays, i.e. a bracket notation and a bunch of lemmas that describe basic properties of the array update operation and booleans. In particular the update operation is defined there as follows.

```
Definition update {Y} (A : array[Y]) (where_ : Z) (val : Y) :=
   fun (i : Z) ⇒ if (Z.eqb i where_) then val else A[i].
```

However, the notation for update is more comfortable: the expression $A[i \leftarrow e]$ means the array A updated at the index i with the value e.

Representation of programs. The subsequent part of the file contains the module with representation of HAHA programs together with their verification conditions. The programs in HAHA are triples the first argument of which is a sequence of predicates, the second one is a sequence of axioms and the last one is a sequence of functions. Translation of a program in this form produces a Coq module called *main*. The module has the following structure

```
(** Verification context: Program correctness*)
Module main.
... Definitions of predicates ...
... Definitions of axioms ...
... Declarations of variables that represent functions ...
... Verification conditions ...
End main.
```

We would like to stress that we add to the mentioned below declarations comments that explain their relation to the original source code and verification conditions visible in the HAHA interface. We show now how the content of a Coq file sections actually looks like.

Predicates. Predicate is actually a triple of the form $(id, args, ex)$ where id is an identifier of the predicate, $args$ is its list of arguments and ex is the actual formula represented by the predicate. These elements are turned to a Coq definition as follows

```
(** Definition of the predicate id?
    – the Coq name is slightly different than in HAHA. *)
Definition id′P (args′) : Prop := ex′.
```

where *id′*, *args′* and *ex′* are Coq translations of the identifier, arguments and formula.

As naming constraints in Coq and HAHA are different, the identifier in HAHA cannot be transferred to Coq with no change. Therefore, we warn the user about it in the comment. We maintain this style of comment for other definitions in the generated file. This repeated routine helps beginners a lot, while experienced users quickly learn to ignore it. In addition, the translation operation adds the suffix P to each identifier to stress that this identifier represents a predicate.[3]

Each HAHA argument of the form *id* : τ is represented in Coq by an expression of the form (*id′* : τ') where *id′* is the Coq representation of the identifier *id* and τ' is the Coq representation of the type τ. The result of translation for a sequence of such arguments is a sequence of above described Coq arguments separated with spaces.

The result of the translation for the HAHA predicate *sorted*, which we mentioned in our overview of the HAHA language in Section 2, page 4, looks as follows.

```
(** Definition of the predicate sorted?
    – the Coq name is slightly different than in HAHA. *)
Definition sortedP (A : array[Z]) (l : Z) (h : Z) : Prop :=
  forall (i : Z) (j : Z), l <= i ∧ i <= j ∧ j < h → A[i] <= A[j].
```

As we can see, the expression that defines the body of the predicate is similar to the expression in the HAHA text, which makes the whole definition easy to digest by novice users.

Axioms. Axioms differ from predicates in that they are not parametrised so an axiom is a pair of the form (id, ex) and is turned to a Coq axiom definition as follows.

```
(** Definition of the axiom id
    – the Coq name is slightly different than in HAHA. *)
Axiom id′A : ex′.
```

with meaning analogous to the one explained for the predicates. We make here one small departure by changing suffixing of the name with P to suffixing with A.

This translation applied to an example axiom written in HAHA in page 2 results in

```
(** Definition of the axiom multi
    – the Coq name is slightly different than in HAHA. *)
Axiom multiA : forall (x : Z), (x + 1) * (x + 1) = x * x + 2 * x + 1.
```

Functions. The internal structure of functions is more complicated. They are actually tuples of the form $(id, args, \tau, com, pres, posts, locs, ht)$ where *id* is the name of the function, *args* is the list of its arguments, τ is the type of the returned result, *com* is a sequence of potential comments concerning the code, *pres* is the sequence of function preconditions, *posts* is the sequence of function postconditions, *locs* is the sequence of local variable declarations, and at last *ht* is the Hoare triple that is the body of the function.

This tuple is turned into a sequence of Coq declarations as follows.

```
(** Declaration of the function name id
    – the Coq name is slightly different than in HAHA. *)
Parameter id′F : args′ → τ′.
```

[3] Similar solution is adopted in Why3 [16], but there the names are prefixed. Our experience shows that it is easier to remember identifiers with suffixes that are devoted to the function than ones with prefixes so we adopted here a solution in accordance with the former choice.

```
(** Verification context: Correctness of id' *)
Module correctness_id'.
  ... translation of the body that depends on id, args, τ, pres, posts, locs, ht ...
End correctness_id'.
```

Again id' is the translation of the HAHA identifier to a Coq one with F as suffix. The arguments are translated as $args'$. This time the argument types are simply the Coq type names or their representations. Similarly, τ' is the result type name.

Before we present the details of translation for items located in the module body, let us see a concrete example for the result of this translation for our insert function *insert*:

```
(** Declaration of the function name insert
   – the Coq name is slightly different than in HAHA. *)
Parameter insertF : array[Z] → Z → Z → array[Z].

(** Verification context: Correctness of insert *)
Module correctness_insert.
...
End correctness_insert.
```

The further translation is divided into two main parts. The first one provides a representation of the local identifiers (i.e. function arguments, the function result identifier and local variables) while the second one, a representation of the verification conditions the proof of which guarantees the correctness of the source program.

The translation of local identifiers translates HAHA declarations of local variables into a series of Coq parameter constructions. For a list of local variables $locs = (id : \tau)locs'$, where id is an identifier, τ is a type and $locs'$ is a list of variable declarations, we obtain:

```
Parameter id' : τ'.
  locs''
```

where id' is a translation of the identifier id, τ' is the translation of the type τ and at last $locs''$ is the translation of the remaining identifiers.

The translation of local identifiers in the program from Figure 1 is as follows.

```
Parameter A : array[Z].
Parameter n : Z.
Parameter e : Z.
(* Variable with the function result. *)
Parameter insert : array[Z].
Parameter j : Z.
(* For boolean types we need both bool and Prop representation. *)
Parameter stop : bool.
Parameter stopP : Prop.
(* Module that ensures equivalence of stop and stopP. *)
Module StopR.
  Definition varb := stop.
  Definition varP := stopP.
  Include BoolRepresents.
End StopR.
Parameter B : array[Z].
```

One additional complexity is associated with booleans. Since these may occur both in expressions that represent values and in predicates, we need polymorphic representation for them. We decided to represent such variables as two Coq variables, in *bool* (*stop*) and one in **Prop** (*stopP*), the equivalence of which is handled by the content of the (*StopR*) module. This module, in particular, contains an axiom $(stop = true) \leftrightarrow stopP$ and some helper lemmas.

As mentioned, the second part of the function translation procedure generates verification conditions. This procedure first combines the list of preconditions into one formula that is the conjunction of the list elements. It then combines the list of postconditions into one formula using the same connective. We can now add the combined precondition formula as the assumption and the combined postcondition formula as the final condition of the mentioned above Hoare triple ht.

Hoare triples. A Hoare triple is a triple $(pres, stmt, post)$ where $pres, post$ are sequences of possibly named assertions and $stmt$ is a HAHA instruction.

The translation to Coq depends on the kind of the instruction in $stmt$. We describe translation for the block, assignment and loop instructions since these show all phenomena of interest. The translation for the remaining instructions follows the same lines.

Triples for blocks. A block is an alternating sequence of instructions and assertions. It starts and ends with an instruction. In the translation of blocks we follow the rule

$$\frac{\{\psi\}\ I\ \{\varphi\}}{\{\psi\}\ \textbf{\textit{begin}}\ I\ \textbf{\textit{end}}\ \{\varphi\}}.$$

In HAHA the situation is a little bit more complex since the body of a block is not an instruction but a sequence of instructions. As a result the Hoare triple in HAHA for which we want to define translation has the form

$$(asserts1, ((st1, asserts2) \cdot sts, st2), asserts3)$$

where $asserts1, asserts2, asserts3$ are assertions, $st1, st2$ are statements, and sts stands for the remaining, possibly empty, part of the block (concatenated with the first statement-assert pair by $\cdot$ operator). As mentioned before, the external assertions $asserts1, asserts3$ are deduced from the external context (i.e. these are external assertions, invariants, pre- or postconditions). Suppose first that sts is not empty. The result of translation starts a new module and generates conditions as follows.

```
(** Verification context: Block at lines $b_{start} - b_{end}$ *)
Module block_L$b_{start}$_$b_{end}$.
  ... translation of $asserts1, st1, asserts2$ ...
  ... translation of $asserts2, (sts, st2), asserts3$ ...
End block_L$b_{start}$_$b_{end}$.
```

where b_{start} is the first line of the block body and b_{end} is the last line of the block, and *asserts2* is the first assert in sts.

In case sts is empty the translation looks slightly differently:

```
(** Verification context: Block at lines $b_{start} - b_{end}$ *)
Module block_L$b_{start}$_$b_{end}$.
  ... translation of $asserts1, st1, asserts2$ ...
  ... translation of $asserts2, st2, asserts3$ ...
End block_L$b_{start}$_$b_{end}$.
```

To illustrate notation in the prescriptions above, we present here the actual code for module creation resulting from the block in lines 31–51 of the Figure 1.

```
(** Verification context: Block at lines 31 - 51*)
Module block_L31_51.
...
End block_L31_51.
```

Triples for assignments. An assignment is a pair (lv, e) where lv is an lvalue one can assign something to and e is an expression. Translation for assignments follows the Hoare's rule

$$\frac{\psi \implies \varphi[e/lv]}{\{\psi\}\ lv := e\ \{\varphi\}}. \tag{1}$$

Assume now that the Hoare triple for the assignment above has the form $(asserts1, (lv, e), (name, assert2))$ where $asserts1$ is the precondition, $name$ is the name of the postcondition assert and $assert2$ is the expression of the assert. The result of translation is as follows.

```
Lemma asserts1'' :
  asserts1'
  →
    assert2' .
```

The value $asserts1''$ is a name of the lemma which retrieves from $asserts1$ the line number n the assertion starts and generates the string of the form Assertion_at_L$n+1$. In case the assertion has a label, the name is simply the text of the label. The expression $asserts1'$ is the translation of the expressions in $asserts1$. The situation with $assert2'$ is more complicated. We have to construct a verification condition as prescribed in the assumption part of the rule (1). As a result, we have to construct first $assert2[e/lv]$ and the translation of this expression to Coq results in $assert2'$ above.

Consider the instruction in line 12 and the following assert in line 13 in Figure 2:

```
  B := A
  { sorted(B, 0, n) /\ n >= 0 }
```

For the assert the following lemma is generated.

```
Lemma Assertion_at_L13:
  n >= 0 ∧
  (sortedP A 0 n)
  →
    (sortedP A 0 n) ∧ n >= 0.
Proof.
  (* Give intros with meaningful names *)
  haha_solve.
Admitted.
(** Change above to Qed.
    when the lemma is proved so that
    "No more subgoals."
    occurs. *)
```

Note that as it is prescribed by the Hoare logic rule, B is replaced in the condition by A. Additionally, the instruction has no assertion that explicitly precedes it, as this is the first instruction of the function body block. In that case the preconditions of the function become the preceding assertion, in this case the conjunction of function preconditions.

One more thing we would like to bring attention to now is that aside from the verification conditions the tool generates a simple proof script. The proof script is strong enough to ascertain the validity of the conditions in most of the cases. We end the script with *Admitted* keyword and suggest to change it to *Qed* when the proof is completed. When the Coq script is finished the lemmas which end with *Admitted* give rise to error markers while those which successfully end with *Qed* lead to clean situation with no alarms in the code.[4]

Loops. The translation of the verification conditions follows here the ***while*** rule

$$\frac{\varphi \implies \theta \quad \{\theta \wedge e\}\ I\ \{\theta\} \quad \theta \wedge \neg e \implies \psi}{\{\varphi\}\ \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ I\ \textbf{\textit{end}}\ \{\psi\}}.$$

A ***while*** loop is a triple (ex, inv, b) where ex is the boolean loop guard expression, inv is the list of loop invariants and b is the Hoare triple that holds the body of the loop. If $asserts1$ is the list of asserts before the loop and $asserts2$ is the list of the asserts after the loop the resulting translation looks as follows

[4] In case *Qed* ends a reasoning which is not successful, the Coq file cannot be compiled and a message is returned to the user that verification cannot be carried out.

```
(** Verification context: Loop statement at line n*)
Module loop_Ln.
(** Verification context: Invariants before the loop*)
Module pre.
... translation of asserts1 ⟹ inv ...
End pre.
(** Verification context: Loop postcondition.*)
Module post.
... translation of inv ∧ ¬ev ⟹ asserts2 ...
End post.
(** Verification context: Invariants after a single iteration.*)
Module invariants.
... translation of the augmented body b′ ...
End invariants.
End loop_Ln.
```

where b' is b with $inv \wedge ex$ as its initial list of conditions and inv as the final one.

Conditionals. The translation of conditionals does not bring any essentially new elements, so we omit it here due to lack of space.

Expressions. The translation for equations is obvious and does not require much description. One thing we can note here is that the array operations take advantage of the function manipulation operations defined in the prelude of the Coq file, and our notations make expressions look close to their counterparts in HAHA.

4 Proof Handling in Coq

The Coq system is difficult to interact with. We mitigate the difficulties in three ways (1) by introduction of extensive comments on the Coq script, (2) by attaching an intuitive how-to instruction to each generated Coq file so that students can consult it to make proving steps of interest and (3) by development of automatic proving tactics.

The Coq system provides a wide variety of automatic verification tools, but they are not tailored to the kind of properties that result from verification condition generators. For this reason, one can significantly improve performance of proving with Coq by developing tactics adapted to the most often used patterns. We devised our own tactic instead of e.g. adopting the work of Chlipala [9] since it is important for such a tactic to be predictable, and it is difficult to control such a big tactic as the one of Chlipala with this regard.

Array updates

The first of our tactics simplifies lookups in updated arrays $A[i \leftarrow v]\,[j]$, depending on the relation of indexes i and j. For a single update there are the following cases:

- the equation $i = j$ or disequation $i <> j$ is already in the context
- one can automatically prove (by *omega*) the equation $i = j$ or disequation $i <> j$
- none of the above.

In the first two cases the tactic simplifies the update expression (either to v or to $A[j]$) and moreover in the second case the proved equation is added to the local context for future use.

In the third case, the proof must be split into two branches, for $i = j$ and for $i <> j$. The branching operation (disjunction elimination) puts the needed premises into the proof context of both branches, so the reduction of update expressions is immediate.

Given a number of update-lookup expressions it is of course more efficient to perform non-splitting reductions first and splitting ones later and this is the case in our tactic.

This procedure is repeated until no more lookups in updated tables are present in the context. The tactic running the above procedure is *solve_updates*.

Rewriting with local hypotheses

Even though there is an automatic rewriting tactic *autorewrite* in Coq, this tactic does not use equations from local context, such as premises of the theorem to be proved. Since equations between variables and other expressions are often present in verification conditions, we proposed a tactic to do exactly this.

Our tactic *simplify* takes every equation (over Z) from the local context, orients them in a way described below and rewrites the whole context with this equation using the Coq tactic **rewrite** $!H$ **in** $*$.

The orientation of the equations is the following:

- equations of the form $v = exp$ are oriented from variable v to the expression, provided that v does not appear in the expression; if it does, the equation is oriented the other way around,
- if neither of the sides of the equation is a variable, the equation is oriented from the larger one towards the smaller one,
- if two sides are of the same nature (two variables, or two expressions of the same size), the equation is used from left to right.

In the first case one practically eliminates a variable from the context; even though the equation itself is not removed to help the user in understanding the proof, it is substituted by the expression everywhere else and does not add up to the complexity of the proof situation. In other cases one can hope for the decrease in the size of the proof context.

After the reduction, the *ring_simplify* tactic is called on all hypotheses and the target in order to tidy up the expressions.

The repeated rewrites (and ring simplifications) are possible thanks to the common technique of temporary hiding "used-up" hypotheses behind an identity constant, so that a given equation does not match the normal *(_ = _)* equation pattern. After the whole series of rewritings the identity is unfolded and the equations become ready to be used again.

Arithmetic forward reasoning

Although the *omega* tactic is very efficient when one wants to prove arithmetical facts, there is no support to generate new facts from existing ones. The simplest example consists in automatically generating *i=n* from the context with *i<=n* and *i>=n*. Given that the inequations may come in a variety of forms (e.g. *i<n+1*, *i−1<n*, *~i>n* etc.) the tactic is more than a simple goal matching.

We have developed an experimental tactic to perform this kind of forward reasoning. The tactic is called *deduce_equality* and its operation consists in the following steps:

1. transform all inequalities into the form *i+a* $<$ *j+b*, where i and j are variables and a, b are arbitrary expressions (hopefully integer constants),
2. given the name of the analysed variable (say i), transform the above inequalities into the form $i < j + b$ and $j + a < i$,
3. search for such a pair of inequalities that the difference $b - a$ is minimal; if it is 2 then one can conclude that $i = j + a + 1$; if it is 3 then we have two cases: $i = j + a + 1$ or $i = j + a + 2$ etc.

Currently, the tactic needs the name of the variable to concentrate on. While local variables can be matched on pretty easily, it is much more difficult to get the names of program variables, modelled in Coq as parameters. If this tactic is supposed to be used completely automatically, one has to either find the way to extract them from the environment or the Coq verification condition generator must pass their names to the tactic.

Cutting the range

It is often the case that some property, typically a loop invariant Inv, is assumed to be true for some range of arguments at hand, e.g. for $i < n$, and one has to prove that it holds for an extended range, e.g. for $i < n + 1$, typically to show that the invariant is preserved after one pass of the loop. In informal proofs one directly concentrates on the "new" part of the range, i.e. $i = n$, taking advantage of the simplifications enabled by the distilled equation.

The formal proof should first split the range into "old" and "new" part, then proof the "old" part using the assumption and leave the user with the "new" part. Unfortunately, given that there are other side-conditions to our invariant (hopefully easy to prove, but difficult to get through by automated tactics), from the Coq perspective it is only possible to know how to split the range by *trying* to do the proof. So given the goal $i < n + 1 \vdash Inv$, we apply our assumption $i < n \to Inv$ and we are left with the impossible goal $i < n + 1 \vdash i < n$ from which we can recover the information that the range should be split into the following two cases: $i < n$ and $n \leq i < n + 1$.

Our tactic *limit_from_hyp* performs this kind of assumption analysis in order to split the range into "old" and "new" part. First, using existential variables and hidden *False* assumption, it sets a trap to recover some information from a following failed proof attempt. Then the tactic tries to **apply** a hypothesis to the goal. If this succeeds it means that the conclusion of the hypothesis has the same form as the goal, as in an invariant preservation proof. In that case, the premises of the applied hypothesis are combined with the current hypotheses using the arithmetic forward reasoning described in the previous paragraph, in order to recover the suitable range splitting point thanks to the prepared existential variables. At this point the tactic "backtracks" using the hidden False assumption. Then it performs the right range splitting, proves the "old" part of the range connected with the used assumption and leaves the user with "new" remaining part(s).

The tactics presented above are incorporated into a general automatic tactic *haha_solve* the invocation of which is generated as the initial body of proofs by the Coq code generator.

Even though the number of automatically proved conditions did not increase dramatically after introducing into *haha_solve* the aforementioned automatic features, these tactics are very useful when one has to resort to manual proving. Basically, using these tactics, one can prove a complex condition in a couple of lines, concentrating on higher-level reasoning instead of technicalities of the proving process.

4.1 Potential for Development

Although *haha_solve* can prove large majority of the generated verification conditions, there are still many cases which are out of reach for our automatic tactic, even though the manual proof is neither long nor particularly involved.

Naturally, there are some clear areas where a progress in automatic proving of verification conditions can and needs to be done. This includes:

1. *Integer division* Integer division, modulo etc. are operations which are not covered by good automatic tactics of Coq, yet they are quite frequent in programming and therefore in verification conditions.

Table 1 The numbers of verification conditions in example algorithms.

File	Number of VCs	Number of unproved VCs	Description
binary_search.haha	8	4	finds a value in an ordered array using bisecting approach
cubic_root.haha	9	0	computes an integer cubic root using squaring
exponent.haha	19	5	computes a power of one number to the other
heapify.haha	38	14	adds an element to a heap so that the heap structure remains intact
insert.haha	39	3	inserts an element into an ordered array (optimised version)
partition.haha	43	3	does the partition subprocedure of quicksort
sortmod2.haha	35	6	sorts elements modulo 2
square_root.haha	12	5	computes an integer square root using addition
sum.haha	7	2	finds the sum of n subsequent numbers starting with 1

2. *First order* The *firstorder* Coq tactic is not very well suited to be used as part of automatic tactics as it often takes a long time to complete (even without success) and cannot be stopped by timeout. In spite of that, elimination and introduction of existential quantification is necessary to prove verification conditions. It would also be desirable to add some support for forward reasoning with generally quantified formulas, i.e. instantiation of quantified formulas with terms at hand to facilitate the proving process.
3. *Transitivity* Sometimes proofs of inequalities require transitivity steps. In automatising such proofs one should allow for transitivity based on existential variables or guided by the facts present in the proof context. This, however, has to be allowed with caution as unlimited transitivity traversal can lead to infinite proof search.
4. *Using equations the other way around* Our automatic rewriting tactic uses some orientation of equalities in rewriting. However, especially in case of expression to expression rewriting, a possibility to rewrite in the opposite direction could sometimes help the proving process. Here also one has to be careful to avoid looping rewriting from l to r and from r to l back again.

4.2 Efficiency of the Tactics

We have run our translation to Coq procedure on a number of publicly available HAHA programs that are downloadable from the HAHA web page and that were previously fully verified with Z3 proving back-end.[5] The examples span a variety of different kinds of algorithms so they can give a reasonable impression on the strength of the tactic. Table 1 shows the number of total goals and the number of goals that were not discharged automatically. As we can see the number of the unproved goals is almost always below 10, which means the

[5] The example programs and generated Coq files can be downloaded from `http://haha.mimuw.edu.pl/coqexamplex.tgz`.

number of cases to be handled interactively is reasonably small. Moreover, out of the 210 totally generated conditions only 20% (42 goals) were not automatically proved. Of course there is a big room for improvement, but at least we have reached the satisfactory level of automatic handling.

5 Specification Debugging Using Coq

To show the support Coq can give in debugging of the specifications, we start working with the *insert* algorithm and its specifications presented in Figure 2. As we mentioned, the specifications in the figure contain mistakes. Let us try to discover it using Coq.

The first step is to generate verification conditions in Coq and see which of them cannot be proved automatically by our tactic *haha_solve*. The tactic leaves six assertions to be solved, namely *Assertion_at_L38*, in *Loop postcondition* section, *Invariant_at_L22_1*, related to the invariant correctness for the invariant in line 22 in case the first branch of the conditional is taken, from *Invariants after a single iteration* section, *Invariant_at_L21_2*, *Invariant_at_L22_2*, *Invariant_at_L23_2*, related to the invariant correctness for the invariant in lines 21, 22 and 23 in case the second branch of the conditional is taken, located in *Invariants after a single iteration* section, and *Assertion_at_L44* after the loop sections.

The correctness of *Assertion_at_L38* can be relatively quickly established by simple logical transformations. Also simple case analysis followed by application of the *haha_solve* tactic resolves goals *Invariant_at_L21_2*, *Invariant_at_L22_2*, *Invariant_at_L23_2*, as well as *Assertion_at_L44*. A different situation is for *Assertion_at_L22_1*.

```
Lemma Invariant_at_L22_1:
   0 <= j + 1 ∧  j + 1 <= n ∧
   (sortedP B 0 n) ∧ (j + 1 < n → (sortedP B 0 (n + 1))) ∧
   (j + 1 < n → forall (k : Z), j < k ∧ k <= n → e <= B[k]) ∧
   (stopP → forall (k : Z), 0 <= k ∧ k <= j → B[k] < e) ∧
   (j + 1 < n → B[j + 1] > e) ∧
   j >= 0 ∧ ~ stopP ∧
   B[j] <= e
  →
     True → forall (k : Z), 0 <= k ∧ k <= j → B[k] < e.
Proof.
   ...
```

This form of the verification condition is difficult to digest so we have to decompose it into smaller pieces. This is done with help of **intros** and **decompose** [**and**] *H* tactics. The resulting proof state is as follows

```
1 subgoal
H0 : True
k : Z
H1 : 0 <= k <= j
H2 : 0 <= j + 1
H4 : j + 1 <= n
H3 : sortedP B 0 n
H5 : j + 1 < n → sortedP B 0 (n + 1)
H6 : j + 1 < n → forall k : Z, j < k <= n → e <= B [k]
H7 : stopP → forall k : Z, 0 <= k <= j → B [k] < e
H8 : j + 1 < n → B [j + 1] > e
H9 : j >= 0
H10 : ~ stopP
H12 : B [j] <= e
______________________________________(1/1)
B [k] < e
```

We can now see that the only way to obtain the inequality $B\ [k]\ < e$ is by using the hypothesis *H7* or by strengthening of *H12*, or by some kind of contradiction. We immediately see that *H7* is not usable, as it is guarded by *stopP*, which does not hold by assumption *H10*. We can now try to see if the assumption $B\ [j]\ <= e$ cannot be made stronger. Indeed, if we forcibly strengthen the condition using Coq **assert** tactic:

```
assert (B[j] < e) by admit.
```

then the proof indeed can be completed in two more steps

```
assert (B[k] <= B[j]) by haha_solve.
haha_solve.
```

as by sortedness of B the condition $B[j]\ < e$ implies the general case. This suggests to take a closer look at the situation for j. We can see that many assumptions are under the condition $j\ +1 < n$. As our conclusion must hold also when $j{+}1 = n$, we can clear all of them to see

```
1 subgoal
H0 : True
k : Z
H1 : 0 <= k <= j
H2 : 0 <= j + 1
H4 : j + 1 <= n
H3 : sortedP B 0 n
H9 : j >= 0
H12 : B [j] <= e
______________________________________(1/1)
B [k] < e
```

As the assumptions do not bring any information on e, except the one in *H12*, we can easily construct a counterexample by letting $B[j]\ = e$. Consequently, the assumptions are not contradictory.

This makes us conclude that the original invariant in line 22 is not provable. Clearly, the obstacle in the proof was the situation that the information in assumption *H12* used $<=$ instead of $<$. Moreover, the information is present in the assumption due to the condition in the **if** instruction. Therefore, a reasonable solution would be to weaken the invariant in line 22 to conclude with $B\ [k]\ <= e$, which indeed leads to a proper invariant.

As we can see from the case above, Coq can be used to systematically examine the situation in a particular point of the code and analyse step-by-step different circumstances that can occur there. This stepwise examination of situations is very similar to stepwise examination of the state in a debugger. Moreover, the possibility to temporarily assume new hypothesis makes it possible to change the internal state of the computation, which is similar to the state update operation available in debuggers.

6 Compilation Using CompCert

In order to provide an execution path for HAHA programs, a compiler front-end has been developed. As firm believers in practising what we preach, we decided that the compiler itself should be formally verified. As such, the front-end is built upon the infrastructure of the CompCert certified compiler [22], and is itself mostly written and verified in Coq, though it uses a trusted parser and typechecker written in OCaml. It translates HAHA into the Cminor intermediate language, which is then compiled to machine code.

6.1 Verified Compilation

By formal verification of a compiler, we understand proving that, given source program S and output code C, some correctness property $Prop(S, C)$ holds. As Leroy [21] points out, many possible properties could be described as "correctness", e.g.

1. "S and C are observationally equivalent";

2. "if S has well-defined semantics (does not go wrong), then S and C are observationally equivalent";
3. "if S is type- and memory-safe, then so is C".
4. "C is type- and memory-safe"

The CompCert compiler uses definition 2, and this is also the definition used in the development of the HAHA front-end.

By the definition above, we consider a compiler *Comp* to be verified correct if the following theorem has been formally proved.

$$\forall S, C, Comp(S) = \texttt{Some}(C) \to Prop(S, C)$$

Or to put it in plain English: if the compiler terminates and produces code in the target language, then this output program is observationally equivalent with the input program. A striking consequence of this definition is that a compiler that never produces any output ($Comp(S) = \texttt{None}$) is trivially considered correct. Indeed, proofs described here only provide for freedom from miscompilation issues. Bugs are possible, though they will be detected at run-time and incorrect code is never produced. Leroy [21] considers this a quality of implementation issue, to be addressed by proper testing.

In more concrete terms, the correctness theorem for the front-end is stated as follows:

$$\begin{aligned}\forall S, C, Comp_{\texttt{HAHA}\to\texttt{Cminor}}(S) = Some(C) \to \\ &(\forall t, r, terminates_{\texttt{HAHA}}(S, t, r) \to terminates_{\texttt{Cminor}}(C, t, r)) \\ &\wedge (\forall T, diverges_{\texttt{HAHA}}(S, T) \to diverges_{\texttt{Cminor}}(C, T))\end{aligned}$$

Where S is the source HAHA program, C is the output Cminor program, t and T are execution traces (terminating and non-terminating, respectively), and r is the program result. In other words, we prove that the front-end, given a HAHA program, may output only those Cminor programs that produce exactly the same execution traces and results as the input program.

6.2 Specification of the HAHA Language

The first step in the process of software verification is providing a formal specification. In the case of a compiler this means formalising the semantics of the input and target languages. To this end, CompCert includes Coq definitions of the semantics for all the languages involved – C99, the target assembly languages and the intermediate languages. The HAHA front-end also provides similar specifications for its own input and intermediate languages, building upon the foundations laid by CompCert.

Dynamic semantics. Unlike recent versions of CompCert, the HAHA front-end relies on big-step (natural) semantics for the specification of languages involved. As noted by Leroy and Grall [24], big-step semantics are more convenient for the purpose of proving the correctness of program transformations, but are at a disadvantage when it comes to describing concurrency and unstructured control flow constructs [23]. The HAHA language does not have such problematic features, however, and we have decided that big-step semantics are sufficient to describe the language and specify correctness theorems in our case.

The specification reuses many elements of CompCert's infrastructure, like execution *traces* and *global environments*. Detailed description of CompCert internals is beyond the scope of this paper, however. For a detailed exposition, see [22].

The semantics is built upon the following definitions:

$$\begin{array}{lcll}
v & ::= & \texttt{hhz}(n \in \mathbb{Z}) \,|\, \texttt{bool}(b \in \{true, false\}) & \\
 & | & \texttt{array}(a \in \mathbb{Z} \to v) & \\
 & | & \texttt{int}(n \in \mathbb{Z}, -2^{31} \leq n < 2^{31}) \,|\, \texttt{undef} & \text{values} \\
t & ::= & \epsilon \,|\, \texttt{cons}(\mathcal{E}, t) & \text{finite trace} \\
T & ::= & \texttt{cons}(\mathcal{E}, T) & \text{infinite trace (coinductive)}
\end{array}$$

Values range over true integers, 32-bit machine integers, Booleans, and arrays. Values `undef` appear only to mark invalid arithmetic operations and should never appear in well-typed programs. Values of variables are never undefined; a variable always has a default initial value, depending on its type (either zero, false or an array of default values).

Notice that arrays are specified as immutable mappings from integers to values and there is no concept of a memory state involved. This seems surprising at first, given that the assignment and LValue syntax suggests mutable data structures. Nevertheless, this choice was made to simplify the specification of function argument passing semantics, which are by-value. Otherwise, every array would have to be copied before being passed as an argument, which would have been costly at runtime and clunky in implementation. Immutable arrays eliminate this problem, though at the same time they generate some, more manageable in our opinion, complexity in semantics of assignment statements.

Traces record details of input/output events, such as system calls or volatile loads and stores.

The following semantic judgements are defined using inductive predicates in Coq:

$$\begin{array}{lcll}
G, E & \vdash & a \Rightarrow v, t & \text{terminating expressions} \\
G & \vdash & s, E \Rightarrow out, E', t & \text{terminating statements} \\
G & \vdash & E, a := v \Rightarrow out, E', t & \text{assignments} \\
G & \vdash & fn(\vec{v}) \Rightarrow v, t & \text{terminating calls} \\
 & \vdash & prog \Rightarrow v, t & \text{terminating programs}
\end{array}$$

Expressions can produce a trace t, i.e. generate side effects in the form of function calls. A separate rule is used to specify the manner assignments affect the local environment, a nontrivial matter due to having to emulate destructive updates on otherwise immutable arrays. The semantics of HAHA statements and expressions do not make explicit use of CompCert memory states. All the state information is contained in environments E and values themselves. The global environment G contains information about functions defined in the translation unit.

At the same time, the usual inductive definitions of natural semantics are not sufficient to describe non-terminating (diverging) programs. In order to cover such cases, we use the approach proposed first by Cousot and Cousot [11], and later implemented in CompCert, which complements the ordinary inductive big-step semantics with co-inductive rules describing non-terminating evaluations.

$$\begin{array}{lcll}
G, E & \vdash & a \overset{\infty}{\Rightarrow} T & \text{diverging expressions} \\
G, E & \vdash & s \overset{\infty}{\Rightarrow} T & \text{diverging statements} \\
G & \vdash & fn(\vec{v}) \overset{\infty}{\Rightarrow} T & \text{diverging calls} \\
 & \vdash & prog \overset{\infty}{\Rightarrow} T & \text{diverging programs}
\end{array}$$

$$\frac{G, E \vdash e_1 \Rightarrow \texttt{array}(a), t_1 \qquad G, E \vdash e_2 \Rightarrow \texttt{hhz}(i), t_2 \qquad a(i) = v}{G, E \vdash e_1[e_2] \Rightarrow v, t_1 \cdot t_2}$$

$$\frac{G, E \vdash \vec{e} \Rightarrow \vec{v}, t_1 \qquad G(id) = fd \qquad G \vdash fd(\vec{v}) \Rightarrow v, t_2}{G, E \vdash id(\vec{e}) \Rightarrow v, t_1 \cdot t_2}$$

$$\frac{G, E \vdash \vec{e} \Rightarrow \vec{v}, t_1 \qquad G \vdash E, id := v \Rightarrow E', t_2}{G \vdash id := e \Rightarrow o, E_2, t_1 \cdot t_2}$$

$$\frac{G \vdash s_1, E_0 \Rightarrow \texttt{out\_normal}, E_1, t_1 \qquad G, E_1 \vdash s_2 \overset{\infty}{\Rightarrow} T}{G, E_0 \vdash (s_1; s_2) \overset{\infty}{\Rightarrow} t_1 \odot T}$$

$$\frac{G, E \vdash e_1 \Rightarrow \texttt{array}(a), t_1 \qquad G, E \vdash e_2 \Rightarrow \texttt{hhz}(i), t_2 \qquad G \vdash E, e_1 := a[i := v] \Rightarrow E', t_3}{G \vdash E, e_1[e_2] := v \Rightarrow E', t_1 \cdot t_2 \cdot t_3}$$

Figure 4 Examples of HAHA semantic rules.

6.3 The Target Language: Cminor

Cminor is the input language of CompCert back-end. It is a low-level imperative language, that has been described as stripped-down variant of C [22]. It is the lowest-level architecture independent language in the CompCert compilation chain and thus is considered to be the entry point to the back-end of the compiler [22].

Cminor has the usual structure of expressions, statements, functions, and programs. Programs are composed of function definitions and global variable declarations. For a more detailed description of the language we refer to [22].

6.4 Implementation

The HAHA front-end for CompCert has a rather conventional design. It consists of a parser, type checker, and several translation passes. These are connected with the CompCert back-end by a simple unverified compiler driver.

The proofs of semantic preservation follow the pattern described by Blazy, Dargaye, and Leroy in [3]. They proceed by induction over big-step HAHA evaluation derivation and case analysis of the last rule used. They show that the output expressions and statements evaluate to the same traces, values, and outcomes as input code, effectively simulating it. Such proofs are conducted for every front-end pass and then composed into a proof of correctness for the whole translation chain. Below, we give short descriptions of the front-end passes.

6.4.1 Parsing and Semantic Analysis

The compiler uses its own unverified GLR parser for the HAHA language. Although initially planned, reusing the Xtext-generated parser of the HAHA environment for a Coq development proved difficult. The additional complexity in defining its semantics and translation into a more usable format would overshadow the benefits of that approach.

The parser produces a raw, untyped AST from the input file. This is then passed into the semantic analyser. Aside from doing type checking, the semantic analysis pass performs several minor, but important, tasks, like string interning and generation of runtime initialisation data for arbitrary precision integer literals, which are stored as global variables in the output program.

6.4.2 Expression Simplification

The first verified pass of the front-end is removal of logic specifications, which do not affect the dynamics of program runs. As it is routine we do not describe it.

The purpose of the next pass is to replace HAHA expressions with no direct equivalents in Cminor. Specifically, function calls, true integer arithmetic, and HAHA array manipulation, all of which in the end are replaced with function calls are pulled out into separate statements. It can be thought of as a form of limited three-address code generation. Following a convention established by CompCert developers, the target language of this pass is called Haha♯minor.

The main idea of the simplification algorithm is very simple. During recursive traversal of the expression tree, every HAHA expression e, which directly results in a side effect, is replaced with a reference to a fresh temporary variable t. A statement reproducing the expression's effect and assigning the resulting value to t is then inserted just before the expression occurs in the code. Additionally, HAHA loops are turned into infinite loop, with a conditional `break` statement inside the iteration.

Temporary variables are semantically similar to HAHA local variables, but separate from them – they reside in their own environment *TE*. Every function can have an unlimited number of temporaries. Although not made explicit in the semantics, identifiers of temporaries should not collide with the identifiers of locals and this is enforced by runtime assertions in the later stages of compilation.

The pass is analogous to *SimplExpr* pass of the CompCert C front-end and uses similar implementation and verification techniques. The semantic preservation is proved with respect to a non-executable, relational specification, expressed using inductive predicates. The specification captures the syntactic conditions under which a Haha♯minor construct could be a valid translation of a given HAHA expression or statement, without prescribing a way in which new variables are generated. It allows a simpler proof semantic preservation, which does not have to deal with the details of implementation. Translation functions are written using monadic programming style, with a dependently typed state monad to generate fresh identifiers. Their outputs are then proved correct with respect to the specification, which is sufficient to establish correctness.

To give an example, the specification for expression is a predicate of the form:

$$a \sim s, a', \vec{id}$$

where a is a HAHA expression, s is a Haha♯minor statement that reproduces the side effects of a, a' is the translated expression and $\vec{id}$ is the set of temporary variables that are referenced in a'. Example rules are given in Figure 5. Notice that uniqueness and disjointedness of temporaries in subexpressions is specified in an abstract way.

6.4.3 Further Simplifications

The next pass translates Haha♯minor into the Hahaminor intermediate language. On a program transformation level, this pass performs two basic tasks, which account for differences in function call semantics between the two languages:

1. Explicit local variable initialisation.
2. Insertion of return statements.

$$\frac{t \in \mathit{tmps}}{\mathtt{hhz}(n) \sim \mathtt{tmp}(t) := \mathtt{alloc}(n), \mathtt{tmp}(t), \mathit{tmps}}$$

$$\frac{\mathtt{typeof}(e) = \mathtt{hhz} \quad e \sim s', e', \mathit{tmps}' \quad t \in \mathit{tmps} \quad t \notin \mathit{tmps}' \quad \mathit{tmps}' \subseteq \mathit{tmps}}{\mathit{op}_1(e) \sim (s'; \mathtt{tmp}(t) := \mathtt{op}_1(e')), \mathtt{tmp}(t), \mathit{tmps}}$$

$$\frac{\begin{array}{c}\mathtt{typeof}(e_1) \neq \mathtt{hhz} \quad \mathtt{typeof}(e_2) \neq \mathtt{hhz} \quad e_1 \sim s_1, e_1', \mathit{tmps}_1 \quad e_2 \sim s_2, e_2', \mathit{tmps}_2 \\ \mathit{tmps}_1 \cap \mathit{tmps}_2 = \emptyset \quad \mathit{tmps}_1 \subseteq \mathit{tmps} \quad \mathit{tmps}_2 \subseteq \mathit{tmps}\end{array}}{\mathit{op}_2(e_1, e_2) \sim (s_1; s_2), \mathtt{op}_2(e_1', e_2'), \mathit{tmps}}$$

Figure 5 Example rules of the expression translation specification.

HAHA and all the intermediate languages up to Haha♯minor use Pascal-like special variable to hold the return value of the function. No special `return` statement is provided. Once the execution of the function body finishes, the value of the special variable is returned to the caller. Additionally, all the local variables are automatically initialised to default values. On the other hand, in Cminor the return value at the end of the function is undefined unless an explicit return statement is provided. Local variables are initially considered undefined and need to have values explicitly assigned.

Initialising variables in the translated program is very simple: it is sufficient to prepend to the function body the appropriate assignments of default values. Providing explicit returns is equally simple. The lack of any unstructured control flow features in the HAHA language means that it is sufficient to simply append a return statement to the function body.

Aside from that, there exists another significant semantic difference between the two languages, Hahaminor deals away with the second local “temporary” environment of Haha♯minor and has only a single environment for local variables. This is an important simplification step before the final pass of Cminor generation, which has plenty of difficulties in proving semantic preservation on its own.

Proof of Correctness

Relating environments. The *Haha♯minor* intermediate language uses two local environments: one for temporary variables introduced by the expression simplification pass, and one for “old” locals. Hahaminor, on the other hand, uses only a single environment. Accordingly, one of the key issues in verification of this pass is to define sensible relation connecting both Haha♯minor environments to the resulting combined Hahaminor environment.

The matching relation $\mathit{MatchEnv}(E, \mathit{TE}, E')$ between a Haha♯minor local and temporary environments E and TE, and a Hahaminor environment E' is defined as follows, assuming that the sets of temporaries and locals are disjoint:

- For all local variables x, $E(x) = E'(x)$.
- For all Haha♯minor temporary variables t, $\mathit{TE}(t) = E'(t)$.

The disjointedness property, despite being easy to provide, is difficult to prove. Instead, a choice was made to insert runtime assertions that would enforce this property in the translation procedures. In the spirit of the definition given in Section 6.1, if the compiler contained a bug that violated this invariant, an error would be produced.

6.4.4 Cminor Code Generation

The next and final pass of the front-end translates Hahaminor into Cminor code, which can then be fed into the CompCert back-end. The translation deals with the encoding of operations of HAHA values into lower-level constructs available in Cminor.

Boolean values are turned into integers and operations on them are rewritten accordingly. Loops are put inside `blocks` and `break` statements are replaced with `exit` statements that jump outside the block of the innermost loop:

$$\texttt{break} \Rightarrow \texttt{exit}(0) \qquad \frac{s \Rightarrow s'}{\texttt{loop}(s) \Rightarrow \texttt{block}(\texttt{loop}(s'))}$$

Array operations and arbitrary precision arithmetic are lowered into runtime library calls. The pointers to initialisation data for integers are provided by the means of an axiom, that is instantiated during program extraction to a hash table lookup.

Proof of Correctness

Despite the simplicity of the code transformations in this pass, the proof of semantic preservation is rather involved. The most problematic aspect is bridging the incompatible worlds of HAHA and Cminor values.

The runtime functions which implement manipulation on HAHA values are specified to accept and return opaque pointers. A sensible preservation proof needs a way to associate those pointers with values they represent.

Following Dargaye [14], we divide memory blocks into following categories:

- Stack blocks (SB). At every function call, Cminor allocates a new stack block to contain the function's activation record. The block is freed upon exit from the call.
- Heap blocks ($\texttt{HB}(v)$). Pointers to these blocks represent HAHA integer and array objects.
- Global blocks (GB). Functions and integer constant initialisation data. A global block is allocated at the very beginning of program execution and it remains live for the entire runtime of the program.
- Invalid blocks (INVALID).

Let $f(M, b)$ be a mapping assigning one of those categories to memory blocks b, for a given memory state M. We may define a value matching relation, parametrised by a memory state M and the mapping f, as follows:

- $\texttt{int}(n) \approx_{(M,f)} \texttt{int}(n)$
- $\texttt{true} \approx_{(M,f)} \texttt{int}(1)$
- $\texttt{false} \approx_{(M,f)} \texttt{int}(0)$
- $\forall v, \texttt{undef} \approx_{(M,f)} v$
- $\dfrac{f(M, blk) = \texttt{HB}(\texttt{hhz}(n))}{\texttt{hhz}(n) \approx_{(M,f)} \texttt{ptr}(blk, 0)}$
- $\dfrac{f(M, blk) = \texttt{HB}(\texttt{array}(a))}{\texttt{array}(a) \approx_{(M,f)} \texttt{ptr}(blk, 0)}$

Values that are represented by pointers are manipulated using runtime library functions, whose behaviour is specified using axioms.

Relating environments. The matching relation $MatchEnv(M, f, E, E')$ that connects a Hahaminor local environment E, block mapping f, and Cminor memory state M and environment E' is defined as follows:

- For all Hahaminor variables x, $E(x) = v$ there exists Cminor value v', such that $E'(x) = v'$ and $v \approx_{M,f} tv$.
- $f \parallel M$,

where $f \parallel M$ denotes a relation that holds if for all b:

- b is not a valid block of M, iff $f(b) = \texttt{INVALID}$,
- b is a valid block of M, iff $f(b) \neq \texttt{INVALID}$.

6.4.5 Runtime Support for HAHA Programs

Like many high-level languages, HAHA has numerous features that do not map directly to features of commodity hardware or the Cminor language. Some of those features, like HAHA Booleans, can be relatively cheaply transformed into inline code. Others require complex algorithms and make use of features like dynamic memory allocation, which in practice require external libraries implementing them.

Since developing the language runtime was considered to be of secondary importance, it was decided to wrap off-the-shelf open-source components to provide most of the functionality. As such, currently the runtime system forms a trusted computing base. Still, we do not consider this a fatal blow, as the libraries we used are widely used and considered reliable.

Current implementation of arbitrary precision integer arithmetic is a thin wrapper around the *GNU Multiple Precision Arithmetic Library* (GMP).

The unusual semantics of HAHA arrays, namely unbounded size and immutability, can be efficiently implemented using functional dictionaries. Indeed, they are currently implemented using balanced binary search trees.

Memory management is currently provided by the Boehm-Demers-Weiser conservative garbage collector [4]. Given that the compiler is not intended to be used in production environments, we consider conservative collection to be adequate.

7 Conclusions and Further Work

Current version of HAHA (0.57) can be viewed as a basic verification platform for programming in the small. It allows one to write imperative procedures and their input-output specifications. Then the specifications can be interactively examined and proved in the Coq proof assistant. Our Coq scripts are systematically filled with information that makes it easy to connect the proof script with the code of the original program. A program, once verified, can be compiled through CompCert compilation chain with its behavioural guarantees. In this way, one can use a types-based tool to teach students the basics of the contemporary software verification technology.

The further steps in the development of HAHA include addition of automatic verification condition computation in the style of weakest precondition generation and introduction of function call stack. However, we are very cautious in introduction of these elements as they will inevitably make presentation of various expressions in the proving back-end complicated and likely to be less readable as it is commonly seen. One possible way to achieve this is to extend ideas used in the CFML project [8] and hide in a careful way the notational overhead introduced there to achieve a more general solution.

References

1 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proc. of CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

2 Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009.

3 Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006. URL: `http://gallium.inria.fr/~xleroy/publi/cfront.pdf`.

4 A garbage collector for C and C++. Retrieved November 16, 2016, from `http://www.hboehm.info/gc/`.

5 Sylvie Boldo and Claude Marché. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4):377–393, December 2011.

6 Richard Bubel and Reiner Hähnle. A Hoare-style calculus with explicit state updates. In Zoltán Instenes, editor, *Proc. of Formal Methods in Computer Science Education (FORMED)*, ENTCS, pages 49–60. Elsevier, 2008.

7 Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proc. of FMCO 2005*, pages 342–363, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

8 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional programming (ICFP)*, pages 418–430. ACM, 2011.

9 Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2013.

10 David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proc. of CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.

11 P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1992. ACM Press, New York, NY.

12 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Proc. of SEFM'12*, volume 7504 of *LNCS*. Springer, 2012.

13 Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430. IEEE, 2009.

14 Zaynah Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot, July 2009.

15 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

16 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proc. of ESOP'13*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

17 Google Java style guide. Retrieved November 16, 2016, from `https://google.github.io/styleguide/javaguide.html`.

18 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

19 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

20 K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000.

21 Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. of POPL'2006*, pages 42–54. ACM Press, 2006. URL: `http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf`.

22 Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

23 Xavier Leroy. Mechanized semantics for compiler verification. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems, 10th Asian Symposium, APLAS 2012*, volume 7705 of *LNCS*, pages 386–388. Springer, 2012. Abstract of invited talk. URL: `http://gallium.inria.fr/~xleroy/publi/mechanized-semantics-aplas-cpp-2012.pdf`.

24 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. URL: `http://gallium.inria.fr/~xleroy/publi/coindsem-journal.pdf`.

25 Linux kernel coding style. Retrieved November 16, 2016, from `https://www.kernel.org/doc/Documentation/CodingStyle`.

26 Tadeusz Sznuk and Aleksy Schubert. Tool support for teaching Hoare logic. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Proc. of SEFM 2014*, volume 8702 of *LNCS*, pages 332–346. Springer, 2014. `doi:10.1007/978-3-319-10431-7_27`.

A Shallow Embedding of Pure Type Systems into First-Order Logic

Łukasz Czajka
DIKU, University of Copenhagen, Copenhagen, Denmark
luta@di.ku.dk

Abstract

We define a shallow embedding of logical proof-irrelevant Pure Type Systems (piPTSs) into minimal first-order logic. In logical piPTSs a distinguished sort $*^p$ of propositions is assumed. Given a context Γ and a Γ-proposition τ, i.e., a term τ such that $\Gamma \vdash \tau : *^p$, the embedding translates τ and Γ into a first-order formula $\mathcal{F}_\Gamma(\tau)$ and a set of first-order axioms Δ_Γ. The embedding is not complete in general, but it is strong enough to correctly translate most of piPTS propositions (by completeness we mean that if $\Gamma \vdash M : \tau$ is derivable in the piPTS then $\mathcal{F}_\Gamma(\tau)$ is provable in minimal first-order logic from the axioms Δ_Γ). We show the embedding to be sound, i.e., if $\mathcal{F}_\Gamma(\tau)$ is provable in minimal first-order logic from the axioms Δ_Γ, then $\Gamma \vdash M : \tau$ is derivable in the original system for some term M. The interest in the proposed embedding stems from the fact that it forms a basis of the translations used in the recently developed CoqHammer automation tool for dependent type theory.

2012 ACM Subject Classification Theory of computation → Logic, Theory of computation → Type theory, Theory of computation → Automated reasoning

Keywords and phrases pure type systems, first-order logic, hammers, proof automation, dependent type theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.9

Funding Supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement number 704111.

1 Introduction

In this paper we define a shallow embedding of any logical proof-irrelevant Pure Type System into untyped minimal first-order logic. Proof-irrelevant PTSs (piPTSs) extend ordinary PTSs with proof-irrelevance by incorporating it into the conversion rule. In logical piPTSs a distinguished sort $*^p$ of propositions is assumed and some restrictions are put on the rules and axioms of the system. The class of logical piPTSs is fairly broad. In particular, a proof-irrelevant version of the Calculus of Constructions with a separate set universe may be presented as a logical piPTS.

Our embedding is shallow, which means that terms of type $*^p$ are translated directly to first-order formulas. The embedding (or an optimised variant of it) is intended to be used to translate dependent type theory goals to formalisms of automated theorem provers (ATPs) for first-order logic. Hence, it is important for efficiency (i.e. the success rate of the ATPs on translated problems) that the embedding be shallow.

The interest in our embedding is justified by the fact that it is used as a basis of the translations employed in the recently developed CoqHammer tool, which is the first hammer for a proof assistant based on dependent type theory [15]. The embedding presented in this paper is only a small "core" version of the translation used in [15]. In particular, here we do not deal at all with inductive types. The translation in [15] handles most of the Coq logic and introduces many optimisations. Consequently, it is quite complex and not easily amenable to a direct theoretical investigation.

The aim of this present paper is to isolate a "core" of the translation from [15] and prove its *soundness*: in a logical piPTS, for any context Γ and a Γ-proposition τ, i.e., a term τ such that $\Gamma \vdash \tau : *^p$, if $\Delta_\Gamma \vdash_{\mathrm{FOL}} \mathcal{F}_\Gamma(\tau)$, i.e., the translation $\mathcal{F}_\Gamma(\tau)$ of τ is derivable in minimal first-order logic from the axioms Δ_Γ which are the translation of Γ, then there exists a term M such that $\Gamma \vdash M : \tau$ in the piPTS. The terminology comes from the hammer and automated reasoning literature [9, 8], where this implication is referred to as soundness and usually formulated in terms of satisfiability. The implication in the other direction, i.e., if $\Gamma \vdash M : \tau$ then $\Delta_\Gamma \vdash_{\mathrm{FOL}} \mathcal{F}_\Gamma(\tau)$, is called *completeness* of the embedding. In type-theoretic literature, e.g. [2, 18], the terminology is flipped. We stick with automated reasoning terminology when referring to soundness or completeness.

Our embedding is not complete, i.e., there exist a context Γ and a Γ-proposition τ such that $\Gamma \vdash M : \tau$ for some M, but $\Delta_\Gamma \not\vdash_{\mathrm{FOL}} \mathcal{F}_\Gamma(\tau)$. However, the presented embedding is "complete enough" to be practically usable, i.e., sufficiently many of the derivable Γ-propositions are provable after the translation for the practical purpose of using an extended and optimised version of the embedding in a hammer tool for dependent type theory. Some empirical evidence for this claim is provided in [15, 14] where over 40% of the translations of Coq standard library theorems are reproved by first-order ATPs, using a (substantially) extended and optimised version of the present embedding. In this paper we do not attempt to rigorously justify or even formulate the "complete enough" claim, but only illustrate the (in)completeness on several examples.

The soundness proof is the main result of this paper. We present the result in a general framework of logical proof-irrelevant Pure Type Systems to avoid unnecessary reliance on any particular variant of dependent type theory. Our soundness proof employs constructive proof-theoretic methods. Assuming the decidability of type checking in the original piPTS, our soundness proof implicitly provides an algorithm to transform a natural deduction proof of the translation of a piPTS proposition into a piPTS term inhabiting the proposition.

1.1 Motivation

In order to give some motivation for our work, we now briefly describe the architecture of a hammer and the relation of the embedding in this paper to the translation used in [15]. For more background on hammers see e.g. [15, 9].

The goal of a hammer is, given a context Γ and a Γ-proposition τ, to find a term M such that $\Gamma \vdash M : \tau$. In practice, the context Γ consists of all declarations accessible at a given point from the proof assistant kernel (typically there are thousands or tens of thousands of them). Hammers work in three phases.

1. Lemma selection which heuristically chooses a subset of the accessible declarations that are likely useful for the conjecture τ. These declarations, together with the declarations they depend on, form a context $\Gamma_0 \subseteq \Gamma$. Typically, the size of Γ_0 is on the order of hundreds of declarations.
2. Translation of the conjecture τ together with the context Γ_0 to the input formats of first-order automated theorem provers (ATPs) like Vampire [22] or Eprover [25], and running the ATPs on the translations.
3. Proof reconstruction which uses the information obtained from a successful ATP run to re-prove the conjecture in the logic of the proof assistant or to directly reconstruct the proof term.

The reason for employing first-order ATPs is that they are currently the strongest and most optimised general-purpose automated theorem provers. They are capable of efficiently handling problems with hundreds of axioms, which is necessary for a hammer tool. The use of state-of-the-art first-order ATPs is the reason why shallowness of the embedding is essential, because the ATPs are heavily optimised for directly handling the primitives of first-order logic. For instance, a declaration $x : \tau$, where $\tau = \Pi y : A.py \to qy$ is a Γ-proposition but A is not (for an appropriate context Γ), should be translated directly to a formula of the form $\forall y.T_A(y) \to p(y) \to q(y)$ where p, q are first-order predicates and the first-order predicate $T_A(y)$ states that y has type A. In contrast, it would be much less efficient to use a deep embedding with Γ-propositions translated to first-order terms and using a binary "inhabitation" predicate T, where the above declaration $x : \tau$ would be translated to an axiom $T(x, \mathcal{C}_\Gamma(\tau))$ and a conjecture τ' to $\exists y.T(y, \mathcal{C}_\Gamma(\tau'))$, with $\mathcal{C}_\Gamma(\alpha)$ the translation of a type α to a first-order term. Such a translation would require the ATPs to synthesise first-order terms corresponding to proof terms which would impact the success rate, even if $T(x, \mathcal{C}_\Gamma(\tau))$ was optimised to e.g. $\forall yz.T_A(y) \to T(z, py) \to T(xyz, qy)$.

The translation in [15] is in fact not sound because of some optimisations. Also the ATPs employed in practice are classical. In the proof reconstruction phase in [15] the conjecture is actually re-proved in the logic of Coq using the lemmas which were needed in an ATP proof. This is feasible because there are typically only a few of these lemmas, so a much weaker method than a state-of-the-art ATP may be used in this final phase. Another issue is that the piPTS formalism does not exactly correspond to common variants of type theory because it assumes proof irrelevance. Since proof irrelevance is crucial to our translation, no soundness proof is possible for ordinary PTSs. However, we believe our soundness proof is still valuable for three main reasons. First, it contributes to the general understanding of the extended translation in [15], and in particular to understanding of which aspects of it are "safe" and which might be not. Second, the proof being constructive implicitly provides an algorithm to transform a natural deduction proof of the translation of a conjecture τ into a piPTS term inhabiting τ. A simplified explicit presentation of the algorithm is given in Algorithm 76. It could form a basis of a partial method for source-level proof reconstruction, i.e., a method for translating a proof found by an ATP back into a proof term in the logic of the proof assistant (possibly using the excluded middle axiom). In mature hammer systems optional source-level proof reconstruction increases success rates. Third, isolating a sound "core" of the translation from [15] might help in devising practical translations for other type theories than just the logic of Coq handled in [15].

From the point of view of proof theory, what we here call completeness of the embedding is perhaps more interesting than soundness. However, all hammer tools essentially give up on completeness. From the automated reasoning perspective it is soundness, or at least understanding the reasons for the lack of it, which is more important.

2 First-order logic

We define a proof notation system for minimal first-order intuitionistic logic. This system of notation is a restriction of the system λP_1 from [28, Chapter 8].

▶ **Definition 1.** An individual term (t, s) is a variable (x, y, z) or a function application $(f(t_1, \ldots, t_n))$. A formula (φ, ψ) is an atom $(R(t_1, \ldots, t_n))$, an implication $(\varphi \to \psi)$ or a universally quantified formula $(\forall x.\varphi)$. A proof term (M, N) is a proof variable (X, Y, Z), an individual abstraction $(\lambda x.M)$, a proof abstraction $(\lambda X : \varphi.M)$, an application of a proof term (MN) or of an individual term (Mt). An environment (Δ) is a finite set of proof

$$\Delta, X : \varphi \vdash X : \varphi$$

$$\frac{\Delta, X : \varphi \vdash M : \psi}{\Delta \vdash (\lambda X : \varphi.M) : \varphi \to \psi} \qquad \frac{\Delta \vdash M : \varphi \to \psi \quad \Delta \vdash N : \varphi}{\Delta \vdash MN : \psi}$$

$$\frac{\Delta \vdash M : \varphi}{\Delta \vdash (\lambda x.M) : \forall x \varphi}\ x \notin \mathrm{FV}(\Delta) \qquad \frac{\Delta \vdash M : \forall x \varphi}{\Delta \vdash Mt : \varphi[t/x]}$$

Figure 1 Rules of minimal first-order logic.

variable declarations of the form $X : \varphi$. We usually write $\Delta, X : \varphi$ instead of $\Delta \cup \{X : \varphi\}$. The system of first-order minimal logic is given by the rules in Figure 1. The relation of β-reduction on proof terms is defined as the contextual closure of the following rules.

$$(\lambda x.M)t \quad \to_\beta \quad M[t/x] \qquad\qquad (\lambda X : \varphi.M)N \quad \to_\beta \quad M[N/X]$$

We write $\Delta \vdash_{\mathrm{FOL}} M : \varphi$ to denote derivability in first-order minimal logic. We drop the subscript when obvious. We also omit the proof terms when irrelevant, writing e.g. $\psi, \theta \vdash \varphi$.

▶ **Lemma 2.** *If $\Delta \vdash M : \varphi$ and $\Delta \vdash M : \varphi'$ then $\varphi = \varphi'$.*

For the proofs of the following two theorems see e.g. [28, Chapter 8].

▶ **Theorem 3** (Confluence and strong normalisation)**.** *If $\Delta \vdash M : \varphi$ then M is confluent and strongly normalising (wrt. β-reduction).*

▶ **Theorem 4** (Subject reduction)**.** *If $\Delta \vdash M : \varphi$ and $M \to_\beta^* M'$ then $\Delta \vdash M' : \varphi$.*

Proof terms in *η-long normal form* or *η-lnf* are defined inductively (wrt. an implicit environment).

- If N is an η-lnf of type α then $\lambda x.N$ is an η-lnf of type $\forall x.\alpha$.
- If N is an η-lnf of type β then $\lambda X : \alpha.N$ is an η-lnf of type $\alpha \to \beta$.
- If $N_1, \ldots, N_n$ are η-lnf or individual terms and $XN_1 \ldots N_n$ is of an atom type, then $XN_1 \ldots N_n$ is an η-lnf.

▶ **Lemma 5.** *If $\Delta \vdash M : \varphi$ then there exists N in η-lnf such that $\Delta \vdash N : \varphi$.*

Proof. Take the β-normal form of M and η-expand it as much as possible, respecting the type and introducing no new β-redexes. The easy details are left to the reader. ◀

The *target* of a formula is defined inductively: $\mathrm{target}(R(t_1, \ldots, t_n)) = R$, $\mathrm{target}(\varphi \to \psi) = \mathrm{target}(\psi)$ and $\mathrm{target}(\forall x.\varphi) = \mathrm{target}(\varphi)$.

▶ **Lemma 6.** *If $\Delta \vdash M : R(t_1, \ldots, t_n)$ and M is in η-lnf then there is $(X : \varphi) \in \Delta$ such that $M = XN_1 \ldots N_k$ and $\mathrm{target}(\varphi) = R$ and each N_i is an individual term or a proof term in η-lnf.*

3 Proof-irrelevant Pure Type Systems

In this section we define proof-irrelevant Pure Type Systems. These extend Pure Type Systems with proof-irrelevance, incorporating it into the conversion rule. Our definition of proof-irrelevant Pure Type Systems is new. It is similar to the definition of a proof-irrelevant version of ECC from [30]. A related treatment of proof-irrelevance for some extensions of the Calculus of Constructions is also present in [5]. The study of the meta-theory of ordinary Pure Type Systems was initiated in [20].

$$
\begin{array}{lcl}
\text{(axiom)} & \langle\rangle \vdash s_1 : s_2 & \text{if } (s_1, s_2) \in \mathcal{A} \\[1em]
\text{(start)} & \dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} & \text{if } x \in V^s \setminus \mathrm{dom}(\Gamma) \\[1em]
\text{(weakening)} & \dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} & \text{if } x \in V^s \setminus \mathrm{dom}(\Gamma) \\[1em]
\text{(product)} & \dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3} & \text{if } (s_1, s_2, s_3) \in \mathcal{R} \\[1em]
\text{(application)} & \dfrac{\Gamma \vdash M : (\Pi x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} & \text{if } N \sim x \\[1em]
\text{(abstraction)} & \dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)} & \\[1em]
\text{(conversion)} & \dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} & \text{if } B =_{\beta\varepsilon} A
\end{array}
$$

Figure 2 Rules of proof-irrelevant PTSs.

▶ **Definition 7.** The set $\mathcal{T}$ of preterms of a proof-irrelevant Pure Type System (piPTS) is defined by the grammar:

$$\mathcal{T} \ ::= \ V^s \mid \mathcal{S} \mid \mathcal{T}\mathcal{T} \mid \lambda V^s : \mathcal{T}.\mathcal{T} \mid \Pi V^s : \mathcal{T}.\mathcal{T} \mid \varepsilon$$

Here $\mathcal{S}$ is a set of sorts, and V^s is a set of variables of sort $s \in \mathcal{S}$. The constant ε represents an arbitrary proof. Its role is technical – it will never occur in well-typed terms. The set $\mathrm{FV}(M)$ of free variables of a preterm M is defined in the usual way. To save on notation we sometimes treat $\mathrm{FV}(M)$ as a list. We use $x, y, z, \ldots$ for variables, $N, M, A, B, \ldots$ for preterms, and $s, s', s_1, s_2, \ldots$ for sorts. We sometimes write x^s to indicate that $x^s \in V^s$. We assume there exists a sort $*^p \in \mathcal{S}$ of propositions.

Note that we tag variables with the sorts of their types, like in [30, 24]. This already appears in [20, 18, 2]. We treat preterms up to α-equivalence, but we do not consider bound variables of different sorts to be α-convertible. For example, if $s_1 \neq s_2$ then $\lambda x^{s_1} : *^p.x^{s_1} \neq_\alpha \lambda x^{s_2} : *^p.x^{s_2}$. Also, whenever we write $\lambda x : A.M$ we assume $x \notin \mathrm{FV}(A)$.

▶ **Definition 8.** The *ε-reduction* is defined as the contextual closure of the rewrite rules:

$$x^{*^p} \ \to_\varepsilon \ \varepsilon \qquad \varepsilon M \ \to_\varepsilon \ \varepsilon \qquad \lambda x : A.\varepsilon \ \to_\varepsilon \ \varepsilon$$

▶ **Definition 9.** A term N is *on the same level as* a variable x, notation $N \sim x$, if one of the following cases holds:

- $x \in V^{*^p}$ and $N \to_\varepsilon^* \varepsilon$, or
- $x \notin V^{*^p}$ and $N \not\to_\varepsilon^* \varepsilon$.

▶ **Definition 10.** We define restricted β-reduction as follows:

$$(\lambda x : A.M)N \ \to_\beta \ M[N/x] \qquad \text{if } N \sim x$$

The restriction $N \sim x$ is necessary to ensure confluence of $\beta\varepsilon$-reduction on preterms. Without the restriction, for e.g. $M = (\lambda x^{*^p} : A.x^{*^p})*^p$ we would have $M \to_\varepsilon^* \varepsilon$ and $M \to_\beta *^p$.

▶ **Definition 11.** The *specification* of a proof-irrelevant PTS is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is a set of sorts, $\mathcal{A}$ is a set of axioms of the form (s_1, s_2) with $s_1, s_2 \in \mathcal{S}$, and $\mathcal{R}$ is a set of rules of the form (s_1, s_2, s_3) with $s_1, s_2, s_3 \in \mathcal{S}$. We often write (s_1, s_2) for $(s_1, s_2, s_2) \in \mathcal{R}$. A *context* is a finite list of declarations of the form $x : A$, or more formally a function from a finite subset of the set of variables to the set of terms. We denote the empty context by $\langle\rangle$. If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ then $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = A_i$. We denote contexts by Γ, Γ', etc. We write $\Gamma' \supseteq \Gamma$ if $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for $x \in \mathrm{dom}(\Gamma)$. A judgement has the form $\Gamma \vdash A : B$. We write $\Gamma \vdash A : B : C$ if $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$. The *proof-irrelevant PTS* (piPTS) determined by the specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is defined by the rules and axioms in Figure 2. We often identify a piPTS with its specification.

▶ **Definition 12.** Let Γ be a context and A a preterm.

1. Γ is *legal* if $\Gamma \vdash M : N$ for some $M, N \in \mathcal{T}$.
2. A is a Γ-*term* if $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ for some $B \in \mathcal{T}$.
3. A is a Γ-*subject* if $\Gamma \vdash A : B$ for some $B \in \mathcal{T}$.
4. A is a Γ-*type* if $\Gamma \vdash A : s$ for some $s \in \mathcal{S}$.
5. A is a Γ-*proposition* if $\Gamma \vdash A : *^p$.
6. A is a Γ-*proof* if $\Gamma \vdash A : B : *^p$,
7. A is *legal* if there exists Γ' such that A is a Γ'-term.

In comparison to ordinary PTSs, as presented in [2, Section 5.2], we only change the application and conversion rules. The side condition in the application rule is necessary because we modify the notion of β-reduction. We need the side condition to prove standard lemmas about piPTSs, in particular the substitution lemma. However, for a class of logical piPTSs, defined below, this side condition may be omitted: $\Gamma \vdash M : A$ iff $\Gamma \vdash^- M : A$ where $\vdash^-$ is the derivation system with the side condition in the application rule omitted (see Lemma 34). The conversion rule is changed to incorporate proof-irrelevance into the system – this is the major difference with ordinary PTSs. In contrast to [30] we do not a priori require $x \in V^{s_1}$ in the product rule.

▶ **Definition 13.** Let $\lambda S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a piPTS.

λS is *functional* if

1. $(s, s_1), (s, s_2) \in \mathcal{A}$ implies $s_1 = s_2$,
2. $(s, s', s_1), (s, s', s_2) \in \mathcal{R}$ implies $s_1 = s_2$.

λS is *logical* if

1. it is functional,
2. $(*^p, *^p, *^p) \in \mathcal{R}$,
3. all rules in $\mathcal{R}$ involving $*^p$ have the form $(s, *^p, *^p)$ or $(*^p, s, s)$,
4. there is no $s \in \mathcal{S}$ with $(s, *^p) \in \mathcal{A}$,
5. there exists $s \in \mathcal{S}$ with $(*^p, s) \in \mathcal{A}$.

Functional PTSs are called *singly-sorted* in [2]. The notion of functional PTSs comes from [20], and also appears in [18]. A notion of logical PTSs similar to ours occurs in [12, 6], but it differs in some technical details. The restrictions in the definition of a logical piPTS ensure that the sort of propositions $*^p$ has the expected properties, which turn out to be needed in the soundness proof.

▶ **Example 14.** A paradigmatic example of a logical piPTS is the calculus of constructions CC^s with a separate impredicative set universe $*^s$.

- $\mathcal{S} = \{*^p, *^s, \Box\}$.

- $\mathcal{A} = \{(*^p, \Box), (*^s, \Box)\}$.
- $\mathcal{R} = \{(*^p, *^p), (*^s, *^p), (*^p, *^s), (*^s, *^s), (\Box, *^p), (*^p, \Box), (\Box, *^s), (*^s, \Box), (\Box, \Box)\}$.

All (piPTS analogons of) systems of the lambda-cube [2, Definition 5.1.10] are also logical piPTSs if we take $*^p = *$. But since they do not have a distinct sort $*^s$ for a set universe, translating them using our embedding does not make much sense – terms intuitively denoting set elements would be erased instead of translated to first-order terms.

▶ **Example 15.** Let $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be any logical piPTS. Let $\Box$ be such that $(*^p, \Box) \in \mathcal{A}$ and let $\alpha \in V^\Box$ and $x \in V^{*^p}$. We have $\alpha : *^p \vdash (\alpha \to \alpha) : *^p$ and $\alpha : *^p \vdash ((\alpha \to \alpha) \to \alpha \to \alpha) : *^p$, because $(*^p, *^p, *^p) \in \mathcal{R}$. Hence by the abstraction rule $\alpha : *^p \vdash (\lambda x : \alpha.x) : \alpha \to \alpha$ and $\alpha : *^p \vdash (\lambda x : \alpha \to \alpha.x) : (\alpha \to \alpha) \to \alpha \to \alpha$. Also $(\lambda x : \alpha.x) \sim x$, because $x \in V^{*^p}$ and $\lambda x : \alpha.x \to_\varepsilon \lambda x : \alpha.\varepsilon \to_\varepsilon \varepsilon$. So $\alpha : *^p \vdash ((\lambda x : \alpha \to \alpha.x)(\lambda x : \alpha.x)) : \alpha \to \alpha$ by the application rule.

The meta-theory of piPTSs is similar to that of ordinary PTSs (see [2, Section 5.2]). The proofs follow the same pattern, except that there is one difficulty caused by the mismatch between $\beta\varepsilon$-reduction in the conversion rule and β-reduction for which the subject reduction theorem holds. Below we only state a few results concerning piPTSs. We delegate the proofs and other details to Appendix A.

The relation $\to_\varepsilon$ (Definition 8) is confluent and strongly normalising. By $\mathrm{nf}_\varepsilon(M)$ we denote the normal form of M w.r.t. $\to_\varepsilon$. Note that $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) \subseteq \mathrm{FV}(M)$.

▶ **Lemma 16.** *If $N \sim x$ then $\mathrm{nf}_\varepsilon(M[N/x]) = \mathrm{nf}_\varepsilon(M)[\mathrm{nf}_\varepsilon(N)/x]$.*

▶ **Lemma 17** (Confluence of $\beta\varepsilon$-reduction)**.** *If $M \to^*_{\beta\varepsilon} M_1$ and $M \to^*_{\beta\varepsilon} M_2$ then there exists M' such that $M_1 \to^*_{\beta\varepsilon} M'$ and $M_2 \to^*_{\beta\varepsilon} M'$.*

▶ **Lemma 18.** *If $M =_{\beta\varepsilon} N$ then $M \to^*_\varepsilon \varepsilon$ is equivalent to $N \to^*_\varepsilon \varepsilon$.*

▶ **Lemma 19.** *If N does not contain ε and $M \to^*_{\beta\varepsilon} N$ then $M \to^*_\beta N$.*

▶ **Lemma 20** (Free variable lemma)**.** *If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma \vdash B : C$ then:*
1. *the $x_1, \ldots, x_n$ are all distinct,*
2. *$\mathrm{FV}(B), \mathrm{FV}(C) \subseteq \{x_1, \ldots, x_n\}$,*
3. *$\mathrm{FV}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $i = 1, \ldots, n$.*

▶ **Lemma 21** (Start lemma)**.** *Let Γ be a legal context.*
1. *If $(s_1, s_2) \in \mathcal{A}$ then $\Gamma \vdash s_1 : s_2$.*
2. *If $(x : A) \in \Gamma$ then $\Gamma \vdash x : A$ and there is $s \in \mathcal{S}$ with $\Gamma_1 \vdash A : s$ and $x \in V^s$, where $\Gamma = \Gamma_1, x : A, \Gamma_2$.*

▶ **Lemma 22** (Substitution lemma)**.** *If $\Gamma, x : A, \Gamma' \vdash B : C$ and $\Gamma \vdash D : A$ and $D \sim x$ then $\Gamma, \Gamma'[D/x] \vdash B[D/x] : C[D/x]$.*

▶ **Lemma 23** (Thinning lemma)**.** *If $\Gamma \vdash A : B$ and $\Gamma' \supseteq \Gamma$ is a legal context then $\Gamma' \vdash A : B$.*

▶ **Lemma 24** (Generation lemma)**.**
1. *If $\Gamma \vdash s : A$ then there is $s' \in \mathcal{S}$ with $A =_{\beta\varepsilon} s'$ and $(s, s') \in \mathcal{A}$.*
2. *If $\Gamma \vdash x : A$ then there are $s \in \mathcal{S}$ and B such that $A =_{\beta\varepsilon} B$ and $\Gamma \vdash B : s$ and $(x : B) \in \Gamma$ and $x \in V^s$.*
3. *If $\Gamma \vdash (\Pi x : A.B) : C$ then there is $(s_1, s_2, s_3) \in \mathcal{R}$ with $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$ and $C =_{\beta\varepsilon} s_3$.*
4. *If $\Gamma \vdash (\lambda x : A.M) : C$ then there are $s \in \mathcal{S}$ and B such that $\Gamma \vdash (\Pi x : A.B) : s$ and $\Gamma, x : A \vdash M : B$ and $C =_{\beta\varepsilon} \Pi x : A.B$.*

5. *If* $\Gamma \vdash MN : C$ *then there are* A, B *such that* $\Gamma \vdash M : (\Pi x : A.B)$ *and* $\Gamma \vdash N : A$ *and* $C =_{\beta\varepsilon} B[N/x]$ *and* $N \sim x$.

▶ **Corollary 25.** *In a logical piPTS, if* $\Gamma \vdash (\Pi x : A.B) : *^p$ *then* $\Gamma, x : A \vdash B : *^p$.

Proof. By the generation lemma there are $s, s' \in \mathcal{S}$ such that $(s, s', *^p) \in \mathcal{R}$ and $\Gamma, x : A \vdash B : s'$. Because the piPTS is logical, we have $s' = *^p$. ◀

▶ **Lemma 26** (Correctness of types lemma). *If* $\Gamma \vdash M : A$ *then there is* $s \in \mathcal{S}$ *such that* $A = s$ *or* $\Gamma \vdash A : s$.

▶ **Lemma 27** (Uniqueness of types lemma).

1. *In a functional piPTS, if* $\Gamma \vdash A : B$ *and* $\Gamma \vdash A : B'$ *then* $B =_{\beta\varepsilon} B'$.
2. *In a logical piPTS, if* $\Gamma \vdash M_1 : A_1$ *and* $\Gamma \vdash M_2 : A_2$ *and* $M_1 =_{\beta\varepsilon} M_2$ *and* $M_1 \not\twoheadrightarrow^*_{\varepsilon} \varepsilon$ *and* $M_2 \not\twoheadrightarrow^*_{\varepsilon} \varepsilon$ *then* $A_1 =_{\beta\varepsilon} A_2$.

▶ **Corollary 28.** *In a functional piPTS, if* $\Pi x : A.B$ *is a* Γ*-term and* $\Gamma \vdash A : s$ *then* $x \in V^s$.

Proof. By the correctness of types and the generation lemmas $\Gamma, x : A$ is a legal context. By the start lemma $\Gamma \vdash A : s'$ and $x \in V^{s'}$ for some $s' \in \mathcal{S}$. But $s' = s$ by the uniqueness of types lemma. ◀

▶ **Theorem 29** (Subject reduction theorem). *If* $\Gamma \vdash A : B$ *and* $A \to^*_{\beta} A'$ *then* $\Gamma \vdash A' : B$.

Subject reduction obviously does not hold for $\beta\varepsilon$-reduction, because ε is not meant to be typable. This generates a small difficulty in proving the following theorem. See Appendix A.

▶ **Theorem 30.** *Assume the piPTS is logical and* M *is a* Γ*-term. Then* M *is a* Γ*-proof if and only if* $M \to^*_{\varepsilon} \varepsilon$.

▶ **Lemma 31.** *In a logical piPTS, if* M *is a* Γ*-term and* $M =_{\beta\varepsilon} N$ *and* $\Gamma \vdash N : s$ *then* $\Gamma \vdash M : s$.

▶ **Lemma 32.** *In a logical piPTS, if* $\Gamma \vdash M : A$ *and* $\Gamma, x : A$ *is a legal context then* $M \sim x$.

▶ **Definition 33.** Given a piPTS specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, we write $\Gamma \vdash^- M : N$ if the judgement $\Gamma \vdash M : N$ is derivable in the piPTS determined by the specification (i.e. using the rules in Figure 2), but with the side condition $N \sim x$ omitted in the application rule.

▶ **Lemma 34.** *In a logical piPTS,* $\Gamma \vdash^- M : N$ *is equivalent to* $\Gamma \vdash M : N$.

▶ Remark. We have not investigated the normalisation or decidability properties of piPTSs. We expect that the (strong) normalisation of an ordinary PTS carries over to its proof-irrelevant version. The same is expected about the decidability of type checking and type inference. The normalisation of our proof-irrelevant version of the Calculus of Constructions (CC^s from Example 14) may probably be shown by adapting a proof-irrelevant model of the ordinary Calculus of Constructions. We do not attempt to answer these questions in the present paper.

4 The embedding

In this and the following section we assume a fixed logical piPTS $\lambda S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

By $\mathbb{T}_{\mathrm{FOL}}$ we denote the set of first-order terms, by $\mathcal{F}_{\mathrm{FOL}}$ the set of first-order formulas, by V_{FOL} the set of first-order variables, and by Σ_{FOL} the first-order signature. We assume that each piPTS variable is also a first-order variable and ε and all piPTS sorts are also first-order constants. Further, we assume five functions $\Lambda_0 : V_{\mathrm{FOL}} \times \mathcal{F}_{\mathrm{FOL}} \times \mathbb{T}_{\mathrm{FOL}} \to \Sigma_{\mathrm{FOL}}$ and $\Lambda_1 : V_{\mathrm{FOL}} \times \mathbb{T}_{\mathrm{FOL}} \times \mathbb{T}_{\mathrm{FOL}} \to \Sigma_{\mathrm{FOL}}$ and $\Phi : \mathcal{F}_{\mathrm{FOL}} \to \Sigma_{\mathrm{FOL}}$ and $\mathcal{G}_0 : V_{\mathrm{FOL}} \times \mathcal{F}_{\mathrm{FOL}} \times \mathbb{T}_{\mathrm{FOL}} \times \mathcal{S} \to \Sigma_{\mathrm{FOL}}$ and $\mathcal{G}_1 : V_{\mathrm{FOL}} \times \mathbb{T}_{\mathrm{FOL}} \times \mathbb{T}_{\mathrm{FOL}} \times \mathcal{S} \to \Sigma_{\mathrm{FOL}}$ returning unique fresh first-order constants. The functions are assumed to yield equal results for terms which differ only in the names of variables, e.g., if σ is a renaming then $\Lambda_1(x, r, t) = \Lambda_1(\sigma(x), \sigma(r), \sigma(t))$. The functions are assumed to give different results for terms which differ not only in the names of variables.

The intention here is that $\Lambda_0, \Lambda_1, \Phi, \mathcal{G}_0, \mathcal{G}_1$ return "fresh" first-order symbol names to be used for translations of "lifted-out" lambda-expressions, propositions or dependent products. The functions should return equal results for translations of lambda-expressions differing only in the names of bound variables (the translations then differ only in the names of first-order variables). Note that such functions always exist – e.g. for $\Lambda_1(x, r, t)$ one may simply choose a new symbol name for each new triple (x, r, t) with variable names standardised, e.g, by renaming to x_i the i-th distinct variable in the triple, counting from the left.

We assume that the first-order signature contains a unary relation symbol P, two binary relation symbols T and E, and a binary function symbol @. An atom $P(t)$ is to be intuitively interpreted as "t is provable", and $T(u, t)$ is to be interpreted as "u has type t". The symbol E represents equality. We prefer to work in minimal first-order logic without equality and add necessary equality axioms in the translation. Using first-order logic with equality would complicate the proof notation system λP_1 and the definition of η-long normal forms. The symbol @ represents application. We usually write tu instead of $@(t, u)$, and we assume application to be left-associative.

We often abbreviate e.g. $MN_1 \ldots N_n$ by $M\vec{N}$, and $M[N_1/x_1] \ldots [N_n/x_n]$ by $M[\vec{N}/\vec{x}]$, and $\Pi x_1 : A_1 \ldots \Pi x_n : A_n.B$ by $\Pi \vec{x} : \vec{A}.B$, and $\lambda x_1 : A_1 \ldots \lambda x_n : A_n.M$ by $\lambda \vec{x} : \vec{A}.M$. We sometimes treat a list of variables $\vec{x}$ as a set. When we write e.g. $\vec{x} = \mathrm{FV}(\varphi, t)$ then $\vec{x}$ is the list of free variables occurring in φ, t in the fixed order from left to right.

The embedding translates a context Γ and a Γ-proposition τ into a set of axioms Δ_Γ and a first-order formula $\mathcal{F}_\Gamma(\tau)$. The embedding uses two functions:

1. $\mathcal{F}_\Gamma$ which translates Γ-propositions to first-order formulas,
2. $\mathcal{C}_\Gamma$ which translates Γ-terms to first-order individual terms.

▶ **Definition 35.** The functions $\mathcal{F}_\Gamma$ and $\mathcal{C}_\Gamma$ are defined by mutual induction on the structure of piPTS terms.

The definition of $\mathcal{F}_\Gamma$ is as follows.

- if $\Gamma \vdash A : *^p$ then $\mathcal{F}_\Gamma(\Pi x : A.B) = \mathcal{F}_\Gamma(A) \to \mathcal{F}_{\Gamma, x:A}(B)$,
- if $\Gamma \nvdash A : *^p$ then $\mathcal{F}_\Gamma(\Pi x : A.B) = \forall x.T(x, \mathcal{C}_\Gamma(A)) \to \mathcal{F}_{\Gamma, x:A}(B)$,
- if M is not a product then $\mathcal{F}_\Gamma(M) = P(\mathcal{C}_\Gamma(M))$.

The definition of $\mathcal{C}_\Gamma$ is as follows. If M is a Γ-proof then $\mathcal{C}_\Gamma(M) = \varepsilon$. Otherwise, we are in one of the following cases.

- $M = s \in \mathcal{S}$. Then $\mathcal{C}_\Gamma(s) = s$.
- $M = x$ is a variable. Then $\mathcal{C}_\Gamma(x) = x$.
- $M = NQ$. Then $\mathcal{C}_\Gamma(NQ) = \mathcal{C}_\Gamma(N)\mathcal{C}_\Gamma(Q)$.

- $M = \lambda x : A.N$ with $\Gamma \vdash A : *^p$. Let $\varphi = \mathcal{F}_\Gamma(A)$ and $t = \mathcal{C}_{\Gamma,x:A}(N)$ and $\vec{y} = \mathrm{FV}(\varphi, t) \setminus \{x\}$ and $f = \Lambda_0(x, \varphi, t)$. Then $\mathcal{C}_\Gamma(\lambda x : A.N) = f\vec{y}$. The idea here is to "lift-out" the translation of a complex lambda-expression M by introducing a name f for it. In Δ_Γ there will be an axiom describing the functional behaviour of f.
- $M = \lambda x : A.N$ with $\Gamma \not\vdash A : *^p$. Let $r = \mathcal{C}_\Gamma(A)$ and $t = \mathcal{C}_{\Gamma,x:A}(N)$ and $\vec{y} = \mathrm{FV}(r, t) \setminus \{x\}$ and $f = \Lambda_1(x, r, t)$. Then $\mathcal{C}_\Gamma(\lambda x : A.N) = f\vec{y}$.
- $M = \Pi x : A.B$ and $\Gamma \vdash M : *^p$. Let $\varphi = \mathcal{F}_\Gamma(\Pi x : A.B)$ and $\vec{y} = \mathrm{FV}(\varphi)$ and $f = \Phi(\varphi)$. Then $\mathcal{C}_\Gamma(\Pi x : A.B) = f\vec{y}$.
- $M = \Pi x : A.B$ and $\Gamma \vdash M : s$ with $s \neq *^p$, and $\Gamma \vdash A : *^p$. Let $\varphi = \mathcal{F}_\Gamma(A)$ and $t = \mathcal{C}_{\Gamma,x:A}(B)$ and $\vec{y} = \mathrm{FV}(\varphi, t) \setminus \{x\}$ and $f = \mathcal{G}_0(x, \varphi, t, s)$. Then $\mathcal{C}_\Gamma(\Pi x : A.B) = f\vec{y}$.
- $M = \Pi x : A.B$ and $\Gamma \vdash M : s$ with $s \neq *^p$, and $\Gamma \not\vdash A : *^p$. Let $t_1 = \mathcal{C}_\Gamma(A)$ and $t_2 = \mathcal{C}_{\Gamma,x:A}(B)$ and $\vec{y} = \mathrm{FV}(t_1, t_2) \setminus \{x\}$ and $f = \mathcal{G}_1(x, t_1, t_2, s)$. Then $\mathcal{C}_\Gamma(\Pi x : A.B) = f\vec{y}$.

Note that it follows from the uniqueness of types lemma that all cases in the definition of $\mathcal{F}_\Gamma$ (resp. $\mathcal{C}_\Gamma$) are exclusive.

▶ **Example 36.** Suppose the piPTS is CC^s from Example 14. Let $\Gamma = \alpha : *^s, p : \alpha \to *^p$ and $\tau = \Pi x : \alpha.px \to px$. Then $\Gamma \vdash \tau : *^p$ and $\mathcal{F}_\Gamma(\tau) = \forall x.T(x, \alpha) \to P(px) \to P(px)$. In practice, the atom $P(px)$ may often be further optimised to $P_p(x)$ with P_p a first-order predicate corresponding to p. This optimisation is performed in [15]. For $Q = \lambda x.\Lambda X : T(x, \alpha).\Lambda Y : P(px).Y$ in η-lnf we have $\vdash_{\mathrm{FOL}} Q : \mathcal{F}_\Gamma(\tau)$. The first-order proof Q may be translated back into a piPTS proof term $M = \lambda x : \alpha.\lambda y : px.y$. In CC^s we have $\Gamma \vdash M : \tau$.

Now let $\Gamma' = \alpha : *^s, p : \alpha \to *^p, a : \alpha, q : pa \to *^p$ and $\tau' = \Pi x : pa.qx \to qx$. Then $\Gamma' \vdash \tau' : *^p$ and $\mathcal{F}_{\Gamma'}(\tau') = P(pa) \to P(q\varepsilon) \to P(q\varepsilon)$. For $Q = \Lambda X : P(pa).\Lambda Y : P(q\varepsilon).Y$ in η-lnf we have $\vdash_{\mathrm{FOL}} Q : \mathcal{F}_{\Gamma'}(\tau')$. The proof Q may be translated back to a piPTS proof term $M = \lambda x : pa.\lambda y : qx.y$. In CC^s we have $\Gamma' \vdash M : \tau'$.

▶ **Definition 37.** The translation $\lceil\Gamma\rceil$ of a context Γ is defined inductively:
- $\lceil\langle\rangle\rceil = \emptyset$,
- $\lceil\Gamma, x : A\rceil = \lceil\Gamma\rceil, \mathcal{F}_\Gamma(A)$ if $\Gamma \vdash A : *^p$,
- $\lceil\Gamma, x : A\rceil = \lceil\Gamma\rceil, T(x, \mathcal{C}_\Gamma(A))$ if $\Gamma \not\vdash A : *^p$.

The set Δ_Γ will consist of $\lceil\Gamma\rceil$ and a set of axioms Δ_{Ax}. To precisely formulate the axioms we need a technical definition of a function $\mathbb{A}_\Gamma$ such that for a FOL formula φ the formula $\mathbb{A}_\Gamma(\varphi)$ is φ prepended with the declarations in Γ translated into guards.

▶ **Definition 38.** The function $\mathbb{A}$ takes a legal context and a FOL formula and returns a FOL formula. It is defined by induction on the length of the context Γ:
- $\mathbb{A}_{\langle\rangle}(\varphi) = \varphi$,
- $\mathbb{A}_{\Gamma,x:A}(\varphi) = \mathbb{A}_\Gamma(\forall x.T(x, \mathcal{C}_\Gamma(A)) \to \varphi)$ if $\Gamma \vdash A : s$ and $s \neq *^p$,
- $\mathbb{A}_{\Gamma,x:A}(\varphi) = \mathbb{A}_\Gamma(\mathcal{F}_\Gamma(A) \to \varphi)$ if $\Gamma \vdash A : *^p$.

The $\mathbb{A}$-*length* of a legal context Γ, denoted $\mathrm{len}_{\mathbb{A}}(\Gamma)$, is defined inductively:
- $\mathrm{len}_{\mathbb{A}}(\langle\rangle) = 0$,
- $\mathrm{len}_{\mathbb{A}}(\Gamma, x : A) = \mathrm{len}_{\mathbb{A}}(\Gamma) + 2$ if $\Gamma \vdash A : s$ and $s \neq *^p$,
- $\mathrm{len}_{\mathbb{A}}(\Gamma, x : A) = \mathrm{len}_{\mathbb{A}}(\Gamma) + 1$ if $\Gamma \vdash A : *^p$.

The $\mathbb{A}$-length of Γ indicates how many arguments need to be applied to a first-order proof of $\mathbb{A}_\Gamma(\varphi)$ in order to obtain a proof of φ. It follows from the uniqueness of types lemma that $\mathbb{A}_\Gamma$ and $\mathrm{len}_{\mathbb{A}}(\Gamma)$ are well-defined for a legal context Γ.

▶ **Example 39.** In CC^s let $\Gamma = \alpha : *^s, a : \alpha, p : \alpha \to *^p, q : pa$ and $\varphi = P(pa)$. Then

$$\mathbb{A}_\Gamma(\varphi) = \forall\alpha.T(\alpha, *^s) \to \forall a.T(a, \alpha) \to \forall p.T(p, f\alpha) \to P(pa) \to P(pa)$$

where $f = \mathcal{G}_1(x, \alpha, *^p, \Box)$. The $\mathbb{A}$-length of Γ is 7, i.e., $\mathrm{len}_{\mathbb{A}}(\Gamma) = 7$.

We need to define a set of axioms Δ_{Ax} for our embedding. These will be axioms concerning the constants introduced in the translation via the functions Λ_0, Λ_1, Φ, $\mathcal{G}_0$ and $\mathcal{G}_1$, and axioms for equality.

▶ **Definition 40.** The set Δ_{Λ_0} contains the following FOL formulas which describe the behaviour of the constants representing "lifted-out" lambda-expressions with propositional arguments.

Given a variable x, a FOL formula φ and a FOL term t, let $f = \Lambda_0(x, \varphi, t)$. Let Γ and A, B be such that:

- $\Gamma \vdash A : *^p$, and
- $\lambda x : A.B$ is a Γ-term but not a Γ-proof, and
- $\varphi = \mathcal{F}_\Gamma(A)$ and,
- $t = \mathcal{C}_{\Gamma,x:A}(B)$.

Let $\vec{y} = \mathrm{FV}(\varphi, t) \setminus \{x\}$. Then Δ_{Λ_0} contains the FOL formula:

- $\mathbb{A}_\Gamma(\varphi \to E(f\vec{y}\varepsilon, t))$.

▶ **Definition 41.** The set Δ_{Λ_1} contains the following FOL formulas which describe the behaviour of the constants representing "lifted-out" lambda-expressions with non-propositional arguments.

Given a variable x and FOL terms r, t, let $f = \Lambda_1(x, r, t)$. Let Γ and A, B be such that:

- $\Gamma \nvdash A : *^p$, and
- $\lambda x : A.B$ is a Γ-term but not a Γ-proof, and
- $r = \mathcal{C}_\Gamma(A)$, and
- $t = \mathcal{C}_{\Gamma,x:A}(B)$.

Let $\vec{y} = \mathrm{FV}(r, t) \setminus \{x\}$. Then Δ_{Λ_1} contains the FOL formula:

- $\mathbb{A}_\Gamma(\forall x.T(x, r) \to E(f\vec{y}x, t))$.

▶ **Example 42.** In CC^s let $\Gamma = \alpha : *^s$ and $M = \lambda x : \alpha.x$. Then $\Gamma \vdash M : \alpha \to \alpha : *^s$. We have $\mathcal{C}_\Gamma(\alpha) = \alpha$ and $\mathcal{C}_{\Gamma,x:\alpha}(x) = x$. Let $f = \Lambda_1(x, \alpha, x)$. Then Δ_{Λ_1} contains the FOL formula

$$\forall \alpha.T(\alpha, *^s) \to \forall x.T(x, \alpha) \to E(f\alpha x, x).$$

Recall that E represents equality.

▶ **Definition 43.** The set Δ_Φ contains the following FOL formulas which are axioms for the constants representing "lifted-out" propositions. Given a FOL formula φ, let $f = \Phi(\varphi)$ and $\vec{y} = \mathrm{FV}(\varphi)$. Then Δ_Φ contains $\forall \vec{y}.\varphi \to P(f\vec{y})$.

▶ **Definition 44.** The set $\Delta_{\mathcal{G}_0}$ contains the following FOL formulas which describe the behaviour of the constants representing "lifted-out" dependent product types with propositional source types.

Given a variable x, a FOL formula φ, a FOL term t and a sort $s \neq *^p$, let $f = \mathcal{G}_0(x, \varphi, t, s)$. Let Γ and A, B be such that:

- $\Gamma \vdash A : *^p$, and
- $\Gamma \vdash \Pi x : A.B : s$, and
- $\varphi = \mathcal{F}_\Gamma(A)$, and
- $t = \mathcal{C}_{\Gamma,x:A}(B)$.

Let $\vec{y} = \mathrm{FV}(\varphi, t) \setminus \{x\}$ and $z \notin \mathrm{FV}(\varphi, t)$. Then $\Delta_{\mathcal{G}_0}$ contains:

- $\mathbb{A}_\Gamma(\forall z.T(z, f\vec{y}) \to \varphi \to T(z\varepsilon, t))$.

▶ **Definition 45.** The set $\Delta_{\mathcal{G}_1}$ contains the following FOL formulas which describe the behaviour of the constants representing "lifted-out" dependent product types with non-propositional source types.

Given a variable x, FOL terms t_1, t_2, and a sort $s \neq *^p$, let $f = \mathcal{G}_1(x, t_1, t_2, s)$. Let Γ and A, B be such that:

- $\Gamma \nvdash A : *^p$, and
- $\Gamma \vdash \Pi x : A.B : s$, and
- $t_1 = \mathcal{C}_\Gamma(A)$, and
- $t_2 = \mathcal{C}_{\Gamma,x:A}(B)$.

Let $\vec{y} = \mathrm{FV}(t_1, t_2) \setminus \{x\}$ and $z \notin \mathrm{FV}(t_1, t_2)$. Then $\Delta_{\mathcal{G}_1}$ contains:

- $\mathbb{A}_\Gamma(\forall z.T(z, f\vec{y}) \to \forall x.T(x, t_1) \to T(zx, t_2))$.

▶ **Example 46.** In CC^s let $\Gamma = \alpha : *^s, p : \alpha \to *^s$. We have $\mathcal{C}_\Gamma(\alpha) = \alpha$ and $\mathcal{C}_{\Gamma,x:\alpha}(px) = px$ and $\Gamma \vdash \Pi x : \alpha.px : *^s$. Let $f = \mathcal{G}_1(x, \alpha, px, *^s)$. Then $\Delta_{\mathcal{G}_1}$ contains

$$\forall \alpha.T(\alpha, *^s) \to \forall p.T(p, g\alpha) \to \forall z.T(z, f\alpha) \to \forall x.T(x, \alpha) \to T(zx, px)$$

where $g = \mathcal{G}_1(x, \alpha, *^s, \Box)$ and $\Delta_{\mathcal{G}_1}$ also contains

$$\forall \alpha.T(\alpha, *^s) \to \forall p.T(p, g\alpha) \to \forall z.T(z, g\alpha) \to \forall x.T(x, \alpha) \to T(zx, *^s).$$

Also $\lceil \Gamma \rceil = T(\alpha, *^p), T(p, g\alpha)$. In [15] there is a distinction between a local context which contains variables bound locally by a λ or a Π, and a global environment which contains the preselected declarations accessible in the proof assistant kernel (Γ_0 from Section 1.1). The guards are not generated for the declarations in the global environment. Assuming Γ here corresponds to the global environment (it is the context translated together with the conjecture), in [15] the last axiom above would be optimised to

$$\forall z.T(z, g\alpha) \to \forall x.T(x, \alpha) \to T(zx, *^s).$$

▶ **Definition 47.** The set $\Delta_{\mathcal{T}_0}$ contains the following FOL formulas which describe the types of the constants representing "lifted-out" lambda-expressions with propositional arguments.

Given a variable x, a FOL formula φ, FOL terms t, u and a sort $s \neq *^p$, let $f = \Lambda_0(x, \varphi, t)$ and $g = \mathcal{G}_0(x, \varphi, u, s)$. Let Γ and A, B, M be such that:

- $\Gamma \vdash A : *^p$, and
- $\Gamma \vdash (\lambda x : A.M) : \Pi x : A.B : s$, and
- $\varphi = \mathcal{F}_\Gamma(A)$, and
- $t = \mathcal{C}_{\Gamma,x:A}(M)$, and
- $u = \mathcal{C}_{\Gamma,x:A}(B)$.

Let $\vec{y} = \mathrm{FV}(\varphi, t) \setminus \{x\}$ and $\vec{z} = \mathrm{FV}(\varphi, u) \setminus \{x\}$. Then $\Delta_{\mathcal{T}_0}$ contains the FOL formula:

- $\mathbb{A}_\Gamma(T(f\vec{y}, g\vec{z}))$.

▶ **Definition 48.** The set $\Delta_{\mathcal{T}_1}$ contains the following FOL formulas which describe the types of the constants representing "lifted-out" lambda-expressions with non-propositional arguments.

Given a variable x, FOL terms r, t, u and a sort $s \neq *^p$, let $f = \Lambda_1(x, r, t)$ and $g = \mathcal{G}_1(x, r, u, s)$. Let Γ and A, B, M be such that:

- $\Gamma \nvdash A : *^p$, and
- $\Gamma \vdash (\lambda x : A.M) : \Pi x : A.B : s$, and
- $r = \mathcal{C}_\Gamma(A)$, and
- $t = \mathcal{C}_{\Gamma,x:A}(M)$, and

- $u = \mathcal{C}_{\Gamma,x:A}(B)$.

Let $\vec{y} = \mathrm{FV}(r,t) \setminus \{x\}$ and $\vec{z} = \mathrm{FV}(r,u) \setminus \{x\}$. Then Δ_{τ_1} contains the FOL formula:

- $\mathbb{A}_\Gamma(T(f\vec{y}, g\vec{z}))$.

▶ **Definition 49.** The set Δ_E, which axiomatises the equality predicate E, contains the following FOL formulas:

- (reflexivity) $\forall x.E(x,x)$,
- (symmetry) $\forall xy.E(x,y) \to E(y,x)$,
- (transitivity) $\forall xyz.E(x,y) \to E(y,z) \to E(x,z)$,
- (congruence) $\forall xyx'y'.E(x,x') \to E(y,y') \to E(xy, x'y')$,
- (substitutivity for P) $\forall xx'.E(x,x') \to P(x) \to P(x')$,
- (substitutivity for T) $\forall xyx'y'.E(x,x') \to E(y,y') \to T(x,y) \to T(x',y')$.

▶ **Definition 50.** We set $\Delta_{\mathrm{Ax}} = \Delta_{\Lambda_0} \cup \Delta_{\Lambda_1} \cup \Delta_\Phi \cup \Delta_{\mathcal{G}_0} \cup \Delta_{\mathcal{G}_1} \cup \Delta_{\tau_0} \cup \Delta_{\tau_1} \cup \Delta_E$ and $\Delta_\Gamma = \Delta_{\mathrm{Ax}} \cup \lceil\Gamma\rceil$.

▶ Remark. Strictly speaking, the set Δ_{Ax} is infinite, because $\Delta_{\Lambda_0}, \Delta_{\Lambda_1}, \Delta_\Phi, \Delta_{\mathcal{G}_0}, \Delta_{\mathcal{G}_1}, \Delta_{\tau_0}, \Delta_{\tau_1}$ are. However, in practice one needs to add the axioms only for the constants f, contexts Γ and terms A, B, M that actually occur during the translation of a given conjecture and its context. There are only finitely many of them. Also, to make proof reconstruction computable we assume that for any constant $f \in \Lambda_1(x,r,t)$ (and analogously for $\Lambda_0, \mathcal{G}_0, \mathcal{G}_1$) it is possible to compute Γ, A, B satisfying the conditions in Definition 41. In practice, Γ, A, B may be associated to f during the translation when an axiom for f is first added.

▶ **Example 51.** In CC^s let $\Gamma = \alpha : *^s, p : (\alpha \to \alpha) \to *^p$ and

$$\tau = p(\lambda x : \alpha.x) \to p((\lambda x : \alpha \to \alpha.x)(\lambda x : \alpha.x)).$$

We have $\lceil\Gamma\rceil = T(\alpha, *^s), T(p, \tau_1\alpha)$ and $\mathcal{F}_\Gamma(\tau) = P(p(f\alpha)) \to P(p(g\alpha(f\alpha)))$ where $f = \Lambda_1(x,\alpha,x)$ and $g = \Lambda_1(x, \tau_2\alpha, x)$ and $\tau_2 = \mathcal{G}_1(x, \alpha, \alpha, *^s)$ and $\tau_1 = \mathcal{G}_1(x, \tau_2\alpha, *^s, \Box)$. The set Δ_{Ax} contains, among others, the following axioms:

- $\forall \alpha.T(\alpha, *^s) \to \forall p.T(p, \tau_1\alpha) \to \forall x.T(x, \tau_2\alpha) \to E(g\alpha x, x)$,
- $\forall \alpha.T(\alpha, *^s) \to T(f\alpha, \tau_2\alpha)$.

In practice, these may be optimised to:

- $\forall x.T(x, \tau_2\alpha) \to E(g\alpha x, x)$,
- $T(f\alpha, \tau_2\alpha)$.

One may derive $\Delta_{\mathrm{Ax}}, \lceil\Gamma\rceil \vdash_{\mathrm{FOL}} E(g\alpha(f\alpha), f\alpha)$. Using the axioms for equality from Δ_E one may thus show $\Delta_{\mathrm{Ax}}, \lceil\Gamma\rceil, P(p(f\alpha)) \vdash_{\mathrm{FOL}} P(p(g\alpha(f\alpha)))$. Hence $\Delta_{\mathrm{Ax}}, \lceil\Gamma\rceil \vdash_{\mathrm{FOL}} \mathcal{F}_\Gamma(\tau)$. Also there is M with $\Gamma \vdash M : \tau$ in CC^s. The use of equality axioms from Δ_E in the derivation of $\Delta_{\mathrm{Ax}}, \lceil\Gamma\rceil \vdash_{\mathrm{FOL}} \mathcal{F}_\Gamma(\tau)$ corresponds to the use of the conversion rule in the derivation of $\Gamma \vdash M : \tau$.

Now consider $\tau' = p(\lambda x : \alpha.x) \to p(\lambda x : \alpha.(\lambda x : \alpha.x)x)$. Then

$$\mathcal{F}_\Gamma(\tau') = P(p(f\alpha)) \to P(p(h\alpha))$$

where $h = \Lambda_1(x, \alpha, f\alpha x)$. In Δ_{Ax} we have the axioms

- $\forall \alpha.T(\alpha, *^s) \to \forall p.T(p, \tau_1\alpha) \to \forall x.T(x, \tau_2\alpha) \to E(h\alpha x, f\alpha x)$,
- $\forall \alpha.T(\alpha, *^s) \to \forall p.T(p, \tau_1\alpha) \to \forall x.T(x, \alpha) \to E(f\alpha x, x)$.

We have $\Delta_{\mathrm{Ax}}, \lceil\Gamma\rceil \nvdash \mathcal{F}_\Gamma(\tau')$, because $\Delta_\Gamma \nvdash_{\mathrm{FOL}} E(h\alpha, f\alpha)$ – only $\Delta_\Gamma, x : \alpha \vdash E(h\alpha x, f\alpha x)$. On the other hand, $\Gamma \vdash (\lambda D : p(\lambda x : \alpha.x).D) : \tau'$ because $\lambda x : \alpha.x =_\beta \lambda x : \alpha.(\lambda x : \alpha.x)x$.

▶ **Example 52.** In CC^s let $\Gamma = p : *^p, q : p \to *^p$ and $\tau = \Pi x : p.\Pi y : p.qx \to qy$. Then $\lceil\Gamma\rceil = T(p, *^p), T(q, \tau_1 p)$ and $\mathcal{F}_\Gamma(\tau) = P(p) \to P(p) \to P(q\varepsilon) \to P(q\varepsilon)$ where $\tau_1 = \mathcal{G}_0(x, p, *^p, \Box)$. The formula $\mathcal{F}_\Gamma(\tau)$ is an intuitionistic tautology. Also $\Gamma \vdash (\lambda x : p.\lambda y : p.\lambda D : qx.D) : \tau$, because $qx =_\varepsilon qy$. This example shows that proof irrelevance is necessary for soundness.

▶ Remark. The incompleteness of the embedding is due to the fact that not enough axioms are present in Δ_{Ax}. After adding axioms expressing the ξ-rule of β-equality, axioms allowing to form new types, axioms corresponding to piPTS axioms, etc., one would probably obtain a complete embedding.

▶ Remark. Assuming the decidability of type checking, the embedding is computable.

Any renaming σ, i.e. a bijection on the set of variables which respects variable sorts, extends in a natural way to a function on first-order terms, formulas (renaming both free and bound variables), piPTS terms, and piPTS contexts.

▶ **Lemma 53.** *Let σ be a renaming.*

1. *If $\Gamma \vdash M : A$ then $\sigma(\Gamma) \vdash \sigma(M) : \sigma(A)$.*
2. *$\mathcal{C}_{\sigma(\Gamma)}(\sigma(M)) = \sigma(\mathcal{C}_\Gamma(M))$.*
3. *$\mathcal{F}_{\sigma(\Gamma)}(\sigma(M)) = \sigma(\mathcal{F}_\Gamma(M))$.*
4. *$\mathbb{A}_{\sigma(\Gamma)}(\sigma(\varphi)) = \sigma(\mathbb{A}_\Gamma(\varphi))$.*

5 Soundness

For the soundness theorem one would want to prove: if $\Delta_{\mathrm{Ax}}, \lceil\Gamma\rceil \vdash \mathcal{F}_\Gamma(A)$ and $\Gamma \vdash A : *^p$ then there is M with $\Gamma \vdash M : A$. However, in the soundness proof we need a weaker notion than the function $\mathcal{F}_\Gamma$. The problem is that $\mathcal{F}_\Gamma$ does not have the necessary substitution properties. For instance if $N = \Pi z : A.B$ with $\Gamma \vdash A : *^p$ and $\Gamma \vdash N : *^p$, then $\mathcal{C}_\Gamma(N) = f$ with $f = \Phi(\mathcal{F}_\Gamma(N))$, and we have

$$\begin{aligned} \mathcal{F}_{\Gamma,x:*^p}(\Pi y : x.x)[\mathcal{C}_\Gamma(N)/x] &= (P(x) \to P(x))[\mathcal{C}_\Gamma(N)/x] \\ &= P(f) \to P(f) \end{aligned}$$

while

$$\begin{aligned} \mathcal{F}_\Gamma((\Pi y : x.x)[N/x]) &= \mathcal{F}_\Gamma(\Pi y : N.N) \\ &= \mathcal{F}_\Gamma(N) \to \mathcal{F}_{\Gamma,y:N}(N) \\ &= (\mathcal{F}_\Gamma(A) \to \mathcal{F}_{\Gamma,z:A}(B)) \to (\mathcal{F}_{\Gamma,y:N}(A) \to \mathcal{F}_{\Gamma,y:N,z:A}(B)). \end{aligned}$$

In the proof we would need these two expressions to be equal. An analogous problem occurs with $\mathcal{C}_\Gamma$. For example if $\Gamma \vdash A : s$ and $\Gamma \nvdash A \to A : *^p$, then

$$\mathcal{C}_{\Gamma,x:A\to A}(\lambda y : A.xy)[\mathcal{C}_\Gamma(\lambda z : A.z)/x] = (fx)[\mathcal{C}_\Gamma(\lambda z : A.z)/x] = fg$$

where $f = \Lambda_1(y, \mathcal{C}_{\Gamma,x:A\to A}(A), xy)$ and $g = \Lambda_1(z, \mathcal{C}_\Gamma(A), z)$, but

$$\mathcal{C}_{\Gamma,x:A\to A}((\lambda y : A.xy)[(\lambda z : A.z)/x]) = \mathcal{C}_{\Gamma,x:A\to A}((\lambda y : A.(\lambda z : A.z)y)) = h$$

where $h = \Lambda_1(y, \mathcal{C}_{\Gamma,x:A\to A}(A), g'y)$ and $g' = \Lambda_1(z, \mathcal{C}_{\Gamma,x:A\to A,y:A}(A), z)$.

The problem is essentially that lambda-abstractions and dependent products may contain free variables. In our setting it does not seem possible to easily solve this problem by e.g. first translating all lambda-abstractions to supercombinators, i.e., terms of the form $\lambda x_1 : A_1 \ldots \lambda x_n : A_n.t$ with $\mathrm{FV}(t) \subseteq \{x_1, \ldots, x_n\}$, and changing the definition of $\mathcal{C}_\Gamma$ to translate multiple consecutive lambda-abstractions at once, thus eliminating the need for the free variables $\vec{y}$ in the axioms in Δ_{Λ_i}. First of all, this is because for a translation to supercombinators to preserve typing additional assumptions on the piPTS would be necessary. Secondly, this would not help with the problem with dependent products exemplified above, which essentially stems from the fact that with our embedding a piPTS proposition may be translated using either $\mathcal{F}$ or $\mathcal{C}$ depending on where it occurs in a term.

We therefore use weaker relations $\succ_\Gamma^{\mathcal{F}}$ and $\succ_\Gamma^{\mathcal{C}}$ instead of the functions $\mathcal{F}_\Gamma$ and $\mathcal{C}_\Gamma$.

▶ **Definition 54.** First, we define a relation $\Gamma' \rightsquigarrow \Gamma$ which expresses the fact that Γ may be obtained from Γ' by repeated substitutions (in the sense of the substitution lemma) and context extensions. More precisely, we define $\rightsquigarrow$ as the transitive-reflexive closure of the relation given by the rule:

$$\Gamma_1, x : A, \Gamma_2 \rightsquigarrow \Gamma \text{ if } \Gamma \supseteq \Gamma_1, \Gamma_2[N/x] \text{ is a legal context and } \Gamma_1 \vdash N : A \text{ and } N \sim x.$$

We write $\Gamma' \rightsquigarrow_{\vec{x},\vec{N}} \Gamma$ to make the terms and the variables substituted for explicit, e.g.,

$$\Gamma_1, x : A, \Gamma_2, y : B, \Gamma_3 \rightsquigarrow_{y,x,N_1,N_2} \Gamma_1, \Gamma_2[N_2/x], \Gamma_3[N_1/y][N_2/x]$$

if $\Gamma_1, x : A, \Gamma_2 \vdash N_1 : B$ and $\Gamma_1 \vdash N_2 : A$ and $N_1 \sim y$ and $N_2 \sim x$. Note that the order of the terms and the variables in the subscript is significant. If additionally $N_i \succ_{\Gamma_i}^{\mathcal{C}} t_i$ (the relation $\succ_\Gamma^{\mathcal{C}}$ is defined below) for appropriate Γ_i, then we write $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. For instance,

$$\Gamma_1, x : A, \Gamma_2, y : B, \Gamma_3 \rightsquigarrow_{x,y,N_1,N_2,t_1,t_2} \Gamma_1, \Gamma_2[N_1/x], \Gamma_3[N_1/x][N_2/y]$$

if $\Gamma_1 \vdash N_1 : A$ and $\Gamma_1, \Gamma_2[N_1/x] \vdash N_2 : B[N_1/x]$ and $N_1 \sim x$ and $N_2 \sim y$ and $N_1 \succ_{\Gamma_1}^{\mathcal{C}} t_1$ and $N_2 \succ_{\Gamma_1,\Gamma_2[N_1/x]}^{\mathcal{C}} t_2$.

▶ **Definition 55.** The relation $\succ_\Gamma^{\mathcal{F}}$ between Γ-propositions and first-order formulas, and the relation $\succ_\Gamma^{\mathcal{C}}$ between Γ-subjects and first-order terms, are defined by mutual induction on the structure of piPTS terms.

The definition of $\succ_\Gamma^{\mathcal{F}}$ is as follows.

- if $\Gamma \vdash A : *^p$ and $A \succ_\Gamma^{\mathcal{F}} \varphi$ and $B \succ_{\Gamma,x:A}^{\mathcal{F}} \psi$ then $\Pi x : A.B \succ_\Gamma^{\mathcal{F}} \varphi \to \psi$,
- if $\Gamma \nvdash A : *^p$ and $A \succ_\Gamma^{\mathcal{C}} t$ and $B \succ_{\Gamma,x:A}^{\mathcal{F}} \varphi$ then $\Pi x : A.B \succ_\Gamma^{\mathcal{F}} \forall x.T(x,t) \to \varphi$,
- if $A \succ_\Gamma^{\mathcal{C}} t$ then $A \succ_\Gamma^{\mathcal{F}} P(t)$.

The last case is not exclusive with the first two.

The definition of $\succ_\Gamma^{\mathcal{C}}$ is as follows. If M is a Γ-proof then $M \succ_\Gamma^{\mathcal{C}} \varepsilon$. Otherwise, we are in one of the following cases.

- $M = s \in \mathcal{S}$. Then $s \succ_\Gamma^{\mathcal{C}} s$.
- $M = x$ is a variable. Then $x \succ_\Gamma^{\mathcal{C}} x$.
- $M = NQ$. If $N \succ_\Gamma^{\mathcal{C}} t_1$ and $Q \succ_\Gamma^{\mathcal{C}} t_2$ then $NQ \succ_\Gamma^{\mathcal{C}} t_1 t_2$.
- $M = (\lambda x : A.Q)[\vec{N}/\vec{x}]$ and there is Γ' such that $\Gamma' \vdash A : *^p$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. Assume $A[\vec{N}/\vec{x}] \succ_\Gamma^{\mathcal{F}} \varphi[\vec{t}/\vec{x}]$ and $Q[\vec{N}/\vec{x}] \succ_{\Gamma,x:A[\vec{N}/\vec{x}]}^{\mathcal{C}} t[\vec{t}/\vec{x}]$. Let $f = \Lambda_0(x,\varphi,t)$ and $\vec{y} = \mathrm{FV}(\varphi,t) \setminus \{x\}$. Then $M \succ_\Gamma^{\mathcal{C}} (f\vec{y})[\vec{t}/\vec{x}]$.
- $M = (\lambda x : A.Q)[\vec{N}/\vec{x}]$ and there is Γ' such that $\Gamma' \nvdash A : *^p$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. Assume $A[\vec{N}/\vec{x}] \succ_\Gamma^{\mathcal{C}} r[\vec{t}/\vec{x}]$ and $Q[\vec{N}/\vec{x}] \succ_{\Gamma,x:A[\vec{N}/\vec{x}]}^{\mathcal{C}} t[\vec{t}/\vec{x}]$. Let $f = \Lambda_1(x,r,t)$ and $\vec{y} = \mathrm{FV}(r,t) \setminus \{x\}$. Then $M \succ_\Gamma^{\mathcal{C}} (f\vec{y})[\vec{t}/\vec{x}]$.
- $M = (\Pi x : A.B)[\vec{N}/\vec{x}]$ and there is Γ' such that $\Gamma' \vdash (\Pi x : A.B) : *^p$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. Assume $M \succ_\Gamma^{\mathcal{F}} \varphi[\vec{t}/\vec{x}]$. Let $f = \Phi(\varphi)$ and $\vec{y} = \mathrm{FV}(\varphi)$. Then $M \succ_\Gamma^{\mathcal{C}} (f\vec{y})[\vec{t}/\vec{x}]$.

- $M = (\Pi x : A.B)[\vec{N}/\vec{x}]$. and there is Γ' such that $\Gamma' \vdash (\Pi x : A.B) : s$ with $s \neq *^p$ and $\Gamma' \vdash A : *^p$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. Assume $A[\vec{N}/\vec{x}] \succ^{\mathcal{F}}_{\Gamma} \varphi[\vec{t}/\vec{x}]$ and $B[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:A[\vec{N}/\vec{x}]} t[\vec{t}/\vec{x}]$. Let $f = \mathcal{G}_0(x, \varphi, t, s)$ and $\vec{y} = \mathrm{FV}(\varphi, t) \setminus \{x\}$. Then $M \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{t}/\vec{x}]$,
- $M = (\Pi x : A.B)[\vec{N}/\vec{x}]$ and there is Γ' such that $\Gamma' \vdash (\Pi x : A.B) : s$ with $s \neq *^p$ and $\Gamma' \nvdash A : *^p$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. Assume $A[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} u_1[\vec{t}/\vec{x}]$ and $B[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:A[\vec{N}/\vec{x}]} u_2[\vec{t}/\vec{x}]$. Let $f = \mathcal{G}_1(x, u_1, u_2, s)$ and $\vec{y} = \mathrm{FV}(u_1, u_2) \setminus \{x\}$. Then $M \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{t}/\vec{x}]$.

Note that not all cases are mutually exclusive.

▶ **Lemma 56.**
1. *If A is a Γ-proposition then $A \succ^{\mathcal{F}}_{\Gamma} \mathcal{F}_{\Gamma}(A)$.*
2. *If A is a Γ-subject then $A \succ^{\mathcal{C}}_{\Gamma} \mathcal{C}_{\Gamma}(A)$.*

Proof. Induction on the definition of $\mathcal{F}_{\Gamma}(A)$ and $\mathcal{C}_{\Gamma}(A)$, using the generation lemma and Corollary 25. ◀

▶ **Definition 57.** The relation $\succ$ between contexts and first-order environments is defined inductively:
- $\langle\rangle \succ \emptyset$,
- if $\Gamma \vdash A : *^p$ and $A \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $\Gamma \succ \Delta$ then $\Gamma, x : A \succ \Delta, \varphi$,
- if $\Gamma \nvdash A : *^p$ and $A \succ^{\mathcal{C}}_{\Gamma} t$ and $\Gamma \succ \Delta$ then $\Gamma, x : A \succ \Delta, T(x, t)$.

The relation $\succ$ is a "relaxed" analogon of the function $\lceil - \rceil$ from Definition 37.

▶ **Definition 58.** We define the relation $\psi \succ^{\mathbb{A}}_{\Gamma;\Gamma'} \varphi$ by induction on Γ':
- $\varphi \succ^{\mathbb{A}}_{\Gamma;\langle\rangle} \varphi$,
- $\psi \succ^{\mathbb{A}}_{\Gamma;\Gamma',x:A} \varphi$ if $\Gamma, \Gamma' \vdash A : s$ and $s \neq *^p$ and $A \succ^{\mathcal{C}}_{\Gamma,\Gamma'} t$ and $\forall x.T(x, t) \to \psi \succ^{\mathbb{A}}_{\Gamma;\Gamma'} \varphi$.
- $\psi \succ^{\mathbb{A}}_{\Gamma;\Gamma',x:A} \varphi$ if $\Gamma, \Gamma' \vdash A : *^p$ and $A \succ^{\mathcal{F}}_{\Gamma,\Gamma'} \psi_A$ and $\psi_A \to \psi \succ^{\mathbb{A}}_{\Gamma;\Gamma'} \varphi$.

Intuitively, $\psi \succ^{\mathbb{A}}_{\Gamma;\Gamma'} \varphi$ means that φ is ψ with prepended relaxed translations of the declarations in Γ' into guards, like in $\mathbb{A}_{\Gamma'}(\psi)$ from Definition 38. The context Γ provides additional declarations for the purpose of typing – they are not translated into guards. By a "relaxed" translation of M we mean a first-order term t (resp. formula θ) satisfying $M \succ^{\mathcal{C}}_{\Gamma''} t$ (resp. $M \succ^{\mathcal{F}}_{\Gamma''} \theta$) for appropriate Γ''.

▶ **Lemma 59.** *If Γ is a legal context then $\Gamma \succ \lceil \Gamma \rceil$ and $\varphi \succ^{\mathbb{A}}_{\langle\rangle;\Gamma} \mathbb{A}_{\Gamma}(\varphi)$.*

Proof. Induction on Γ, using Lemma 56. ◀

▶ **Lemma 60.** *If $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$ and $\Gamma', y : A$ is a legal context and y is fresh, i.e., it does not occur in Γ', Γ or any intermediate context, then $\Gamma', y : A \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma, y : A[\vec{N}/\vec{x}]$.*

Proof. Induction on the definition of $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$. ◀

▶ **Lemma 61.** *If $\Gamma' \vdash A : B$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N}} \Gamma$ then $\Gamma \vdash A[\vec{N}/\vec{x}] : B[\vec{N}/\vec{x}]$.*

Proof. Follows by repeatedly applying the substitution and thinning lemmas. ◀

From now on, whenever we write $M \succ^{\mathcal{F}}_{\Gamma} t$ we implicitly assume that M is a Γ-proposition. Similarly, whenever we write $M \succ^{\mathcal{C}}_{\Gamma} t$ we assume M is a Γ-subject. Note that it follows from the generation lemma and Lemma 61 that if e.g. $\Pi x : A.B \succ^{\mathcal{F}}_{\Gamma} \varphi \to \psi$ and $\Pi x : A.B$ is a Γ-proposition, then A is a Γ-proposition and B is a $(\Gamma, x : A)$-proposition (and analogously for all other cases of Definition 55). So the assumption that the left-hand sides of $\succ^{\mathcal{F}}$ are propositions is preserved for $A \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $B \succ^{\mathcal{F}}_{\Gamma,x:A} \psi$. We will often use this observation implicitly. Because of page limits, proofs of many of the following helper lemmas have been moved to Appendix B.

▶ **Lemma 62.**
1. *If $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $\Gamma' \supseteq \Gamma$ is a legal context then $M \succ^{\mathcal{F}}_{\Gamma'} \varphi$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma} t$ and $\Gamma' \supseteq \Gamma$ is a legal context then $M \succ^{\mathcal{C}}_{\Gamma'} t$.*

▶ **Corollary 63.** *If $\psi \succ^{\mathbb{A}}_{\Gamma;\Gamma_0} \varphi$ and $\Gamma' \supseteq \Gamma$ and Γ', Γ_0 is a legal context then $\psi \succ^{\mathbb{A}}_{\Gamma';\Gamma_0} \varphi$.*

▶ **Lemma 64.** *Assume $N \sim x$. Then $M \to^*_{\varepsilon} \varepsilon$ iff $M[N/x] \to^*_{\varepsilon} \varepsilon$.*

▶ **Lemma 65.** *Assume $\Gamma_1 \vdash N : A$ and $N \succ^{\mathcal{C}}_{\Gamma_1} t$ and $N \sim y$.*
1. *If $M \succ^{\mathcal{F}}_{\Gamma_1, y:A, \Gamma_2} \varphi$ then $M[N/y] \succ^{\mathcal{F}}_{\Gamma_1, \Gamma_2[N/y]} \varphi[t/y]$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma_1, y:A, \Gamma_2} u$ then $M[N/y] \succ^{\mathcal{C}}_{\Gamma_1, \Gamma_2[N/y]} u[t/y]$.*

▶ **Corollary 66.**
1. *If $M \succ^{\mathcal{F}}_{\Gamma'} \varphi$ and $\Gamma' \rightsquigarrow_{\vec{x}, \vec{N}, \vec{t}} \Gamma$ then $M[\vec{N}/\vec{x}] \succ^{\mathcal{F}}_{\Gamma} \varphi[\vec{t}/\vec{x}]$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma'} \varphi$ and $\Gamma' \rightsquigarrow_{\vec{x}, \vec{N}, \vec{t}} \Gamma$ then $M[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} \varphi[\vec{t}/\vec{x}]$.*

▶ **Lemma 67.** *Assume $y \in V^{*^p}$.*
1. *If $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ then $y \notin \mathrm{FV}(\varphi)$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma} t$ then $y \notin \mathrm{FV}(t)$.*

▶ **Lemma 68.**
1. *If $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ then $\mathrm{FV}(\varphi) = \mathrm{FV}(\mathrm{nf}_{\varepsilon}(M))$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma} t$ then $\mathrm{FV}(t) = \mathrm{FV}(\mathrm{nf}_{\varepsilon}(M))$.*

▶ **Lemma 69.** *Assume $\Gamma =_{\varepsilon} \Gamma'$.*
1. *If $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $M' \succ^{\mathcal{F}}_{\Gamma'} \varphi$ then $M =_{\varepsilon} M'$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma} t$ and $M' \succ^{\mathcal{C}}_{\Gamma'} t$ then $M =_{\varepsilon} M'$.*

▶ **Lemma 70.** *If $\psi \in \Delta$ and $\Gamma \succ \Delta$ and ψ has target P, then there are $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ and C such that $\Gamma = \Gamma_1, x : C, \Gamma_2$ and $\Delta = \Delta_1, \psi, \Delta_2$ and $\Gamma_1 \succ \Delta_1$ and $\Gamma_1 \vdash C : *^p$ and $C \succ^{\mathcal{F}}_{\Gamma_1} \psi$.*

▶ **Lemma 71.** *If $\psi \in \Delta$ and $\Gamma \succ \Delta$ and ψ has target T, then $\psi = T(x, t)$ and there are Γ_1, Γ_2 and C such that $\Gamma = \Gamma_1, x : C, \Gamma_2$ and $C \succ^{\mathcal{C}}_{\Gamma_1} t$.*

▶ **Lemma 72.** *If $C \succ^{\mathcal{F}}_{\Gamma} \varphi$ then $C = \Pi x_1 : A_1 \ldots \Pi x_n : A_n.B$ with $B \succ^{\mathcal{C}}_{\Gamma,\Gamma_0} t$ and $P(t) \succ^{\mathbb{A}}_{\Gamma;\Gamma_0} \varphi$ and $\Gamma_0 = x_1 : A_1, \ldots, x_n : A_n$.*

▶ **Definition 73.** Assume $\Delta_{\mathrm{Ax}}, \Delta \vdash Q : \varphi$ and $\Gamma \succ \Delta$. A $\Gamma, \Delta, A, \varphi$*-reconstruction* of Q, or just a *reconstruction* of Q, is defined as follows, depending on the form of φ.
1. If $A \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $\Gamma \vdash A : *^p$ then any M such that $\Gamma \vdash M : A$ is a $\Gamma, \Delta, A, \varphi$*-reconstruction* of Q.
2. If $\varphi = T(t, t')$ and $A \succ^{\mathcal{C}}_{\Gamma} t'$ and $\Gamma \vdash A : s$ with $s \neq *^p$ then any M such that $M \succ^{\mathcal{C}}_{\Gamma} t$ and $\Gamma \vdash M : A$ is a $\Gamma, \Delta, A, \varphi$*-reconstruction* of Q.
3. If $\varphi = E(t', t)$ and $A \succ^{\mathcal{C}}_{\Gamma} t'$ (resp. $A \succ^{\mathcal{C}}_{\Gamma} t$) and A is a Γ-subject then any Γ-subject M such that $M \succ^{\mathcal{C}}_{\Gamma} t$ (resp. $M \succ^{\mathcal{C}}_{\Gamma} t'$) and $M =_{\beta\varepsilon} A$ is a $\Gamma, \Delta, A, \varphi$*-reconstruction* of Q.

Note that if $A \succ^{\mathcal{F}}_{\Gamma} \varphi$ then φ does not have the form $T(t, t')$ or $E(t, t')$, so the three above cases are actually exclusive. We stress that the notion of a reconstruction depends on the $\Gamma, \Delta, A, \varphi$, but we often omit them when clear.

A first-order proof term Q is *reconstructible* if for any $\Gamma, \Delta, A, \varphi$, satisfying the appropriate conditions as above, a $\Gamma, \Delta, A, \varphi$-reconstruction of Q exists.

▶ **Lemma 74.** *Suppose $\Gamma \succ \Delta$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash X Q_1 \ldots Q_m : \psi$, where each Q_i is either an individual term or a reconstructible proof term. Let $\Gamma_0 = x_1 : A_1, \ldots, x_n : A_n$ be such that $m = \mathrm{len}_{\mathbb{A}}(\Gamma_0)$ and Γ, Γ_0 is a legal context. If $(X : \gamma) \in \Delta_{\mathrm{Ax}}, \Delta$ with $\varphi \succ^{\mathbb{A}}_{\Gamma;\Gamma_0} \gamma$, then there exist $N_1, \ldots, N_n$ and $u_1, \ldots, u_n$ such that $\psi = \varphi[\vec{u}/\vec{x}]$ and $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma$.*

▶ **Theorem 75** (Soundness of the embedding). *Every first-order proof term Q in η-long normal form is reconstructible.*

The proof of the soundness of the embedding is a bit long and tedious because of the many cases that need to be considered. As mentioned before, the soundness proof implicitly defines an algorithm to transform a first-order proof term Q in η-lnf into its reconstruction. More precisely, given a proof term Q in η-lnf and $\Gamma, \Delta, A, \varphi$ satisfying the conditions in Definition 73, the algorithm constructs a $\Gamma, \Delta, A, \varphi$-reconstruction M of Q. We first informally sketch this algorithm. The proof of Theorem 75 is essentially a proof of its correctness. Because any proof term may be β-reduced and η-expanded to a proof term in η-lnf (Lemma 5), this provides a general proof reconstruction method.

▶ **Algorithm 76.** Assume $\Delta_{\mathrm{Ax}}, \Delta \vdash Q : \varphi$ and $\Gamma \succ \Delta$. We assume that $\Gamma \succ \Delta$ is given constructively, i.e., given $(X : \varphi) \in \Delta$ it is possible to retrieve $(x : C) \in \Gamma$ such that $C \succ_\Gamma^{\mathcal{F}} \varphi$, or $\varphi = T(x, t)$ and $C \succ_\Gamma^{\mathcal{C}} t$ (c.f. Definition 57 and Lemma 62). We have the following cases. For the sake of readability we do not treat all cases in full generality.

1. $A \succ_\Gamma^{\mathcal{F}} \varphi$ and $\Gamma \vdash A : *^p$. We seek M with $\Gamma \vdash M : A$. Consider possible forms of φ.
   - $\varphi = \varphi_1 \to \varphi_2$. Then $Q = \lambda X : \varphi_1.Q'$ (because Q is in η-lnf) and $A = \Pi x : B.C$ (by Definition 55) with $B \succ_\Gamma^{\mathcal{F}} \varphi_1$ and $C \succ_{\Gamma,x:B}^{\mathcal{F}} \varphi_2$. Recursively construct a $\Gamma', \Delta', C, \varphi_2$-reconstruction M' of Q', where $\Gamma' = \Gamma, x : B$ and $\Delta' = \Delta, \varphi_1$. Take $M = \lambda x : B.M'$.
   - $\varphi = \forall x.T(x, t) \to \psi$. Then $Q = \lambda x \lambda X : T(x, t).Q'$ and $A = \Pi x : B.C$. Recursively construct a $\Gamma', \Delta', C, \psi$-reconstruction M' of Q', where $\Gamma' = \Gamma, x : B$ and $\Delta' = \Delta, T(x, t)$. Take $M = \lambda x : B.M'$.
   - $\varphi = P(t_A)$ with $A \succ_\Gamma^{\mathcal{C}} t_A$. Then $Q = XD_1 \dots D_k$ where $(X : \psi) \in \Delta_{\mathrm{Ax}}, \Delta$ and $\mathrm{target}(\psi) = P$, and each D_i is a first-order proof term in η-long normal form or an individual term. We consider possible forms of ψ.
     - $(X : \psi) \in \Delta$. For example, $Q = XtD_1D_2$ where D_1, D_2 are proof terms in η-lnf and $\psi = \forall x.T(x, t_B) \to \psi' \to P(fx)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : T(t, t_B)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : \psi'[t/x]$ and f is a *variable.* There is $(z : C) \in \Gamma$ such that $C \succ_\Gamma^{\mathcal{F}} \psi$ and $C = \Pi x : B.\Pi y : B'.fx$. Recursively construct a $\Gamma, \Delta, B, T(x, t_B)$-reconstruction M_1 of D_1 and a $\Gamma, \Delta, B', \psi'[t/x]$-reconstruction M_2 of D_2. Take $M = zM_1M_2$.
     - $(X : \psi) \in \Delta_E$ and $\psi = \forall xx'.E(x, x') \to P(x) \to P(x')$. Then $Q = Xtt_AD_1D_2$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : E(t, t_A)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : P(t)$. Recursively construct a $\Gamma, \Delta, A, E(t, t_A)$-reconstruction B of D_1 and then a $\Gamma, \Delta, B, P(t)$-reconstruction M' of D_2. Take $M = M'$.
     - $(X : \psi) \in \Delta_\Phi$ and e.g. $\psi = \forall y.\psi' \to P(fy)$ and $\mathrm{FV}(\psi') = \{y\}$ and $f = \Phi(\psi')$. Then $t_A = ft$ and $Q = XtD$ with $\Delta_{\mathrm{Ax}}, \Delta \vdash D : \psi'[t/y]$. Since $A \succ_\Gamma^{\mathcal{C}} ft$, by Definition 55 there are Γ', N with $\Gamma' \rightsquigarrow_{y,N,t} \Gamma$. So $N \succ_\Gamma^{\mathcal{C}} t$ by Definition 54 and the thinning lemma. Also $A \succ_\Gamma^{\mathcal{F}} \psi'[t/y]$ (Definition 55). Recursively construct a $\Gamma, \Delta, A, \psi'[t/y]$-reconstruction M of D. This is also a $\Gamma, \Delta, A, P(ft)$-reconstruction of Q.
2. $\varphi = T(t, t')$ and $A \succ_\Gamma^{\mathcal{C}} t'$ and $\Gamma \vdash A : s$ with $s \neq *^p$. We seek M such that $M \succ_\Gamma^{\mathcal{C}} t$ and $\Gamma \vdash M : A$. We have $Q = X\vec{D}$ where $(X : \psi) \in \Delta_{\mathrm{Ax}}, \Delta$ and $\mathrm{target}(\psi) = T$. Consider possible forms of ψ.
   - $(X : \psi) \in \Delta$. Then $\psi = T(x, t')$ and $t = x$ and there is $(x : C) \in \Gamma$ such that $C \succ_\Gamma^{\mathcal{C}} t'$. Take $M = x$.
   - $(X : \psi) \in \Delta_{\mathcal{G}_1}$ and e.g. $\psi = \forall z.T(z, f) \to \forall x.T(x, r_1) \to T(zx, r_2)$ where $f = \mathcal{G}_1(x, r_1, r_2, s)$ and $\mathrm{FV}(r_1, r_2) \subseteq \{x\}$. Then $Q = XuD_1wD_2$ and $t = uw$ and $t' = r_2[w/x]$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : T(u, f)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : T(w, r_1)$. We may

compute Γ_0, C with $C = \Pi x : C_1.C_2$ and $C_1 \succ^{\mathcal{C}}_{\Gamma_0} r_1$ and $C_2 \succ^{\mathcal{C}}_{\Gamma_0, x:C_1} r_2$ and $C \succ^{\mathcal{C}}_{\Gamma_0} f$ (see Remark 4). One shows that $\Gamma = \Gamma_0$ may be assumed in the case $\mathrm{FV}(r_1, r_2) \subseteq \{x\}$. Recursively construct a $\Gamma, \Delta, C, T(u, f)$-reconstruction M_1 of D_1 and a Γ, Δ, C_1, r_1-reconstruction M_2 of D_2. Take $M = M_1 M_2$.

- $(X : \psi) \in \Delta_{\tau_1}$ and e.g. $\psi = T(f, g)$ where $f = \Lambda_1(x, r, t)$ and $g = \mathcal{G}_1(x, r, u, s)$ with $\mathrm{FV}(r, t, u) \subseteq \{x\}$. We may compute C, N such that $C \succ^{\mathcal{C}}_{\Gamma} r$ and $N \succ^{\mathcal{C}}_{\Gamma, x:C} t$, so $\lambda x : C.N \succ^{\mathcal{C}}_{\Gamma} f$. Take $M = \lambda x : C.N$.
- $(X : \psi) \in \Delta_E$. Then $\psi = \forall x y x' y'. E(x, x') \to E(y, y') \to T(x, y) \to T(x', y')$ and $Q = X u u' t t' D_1 D_2 D_3$ where $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : E(u, t)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : E(u', t')$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_3 : T(u, u')$. Recursively construct a $\Gamma, \Delta, A, E(u', t')$-reconstruction A' of D_2, then a $\Gamma, \Delta, A, T(u, u')$-reconstruction M' of D_3, then a $\Gamma, \Delta, M', E(u, t)$-reconstruction N of D_1. Take $M = N$.

3. $\varphi = E(t_0, t_1)$ and e.g. $A \succ^{\mathcal{C}}_{\Gamma} t_0$. We need to find M with $M \succ^{\mathcal{C}}_{\Gamma} t_1$ and $M =_{\beta\varepsilon} A$. Since $E(t_0, t_1)$ is an atom and Q is in η-lnf, $Q = X\vec{D}$ where $(X : \psi) \in \Delta_{\mathrm{Ax}}, \Delta$ and $\mathrm{target}(\psi) = E$. Consider possible forms of ψ.

- $(X : \psi) \in \Delta_{\Lambda_1}$ and e.g. $\psi = \forall x. T(x, r_1) \to E(fx, r_2)$ where $f = \Lambda_1(x, r_1, r_2)$ and $\mathrm{FV}(r_1, r_2) \subseteq \{x\}$. Then $Q = XuD$ and $t_0 = fu$ and $t_1 = r_2[u/x]$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D : T(u, r_1)$. We may compute C, N such that $C_1 \succ^{\mathcal{C}}_{\Gamma} r_1$ and $C_2 \succ^{\mathcal{C}}_{\Gamma, x:C_1} r_2$ and $\lambda x : C.N \succ^{\mathcal{C}}_{\Gamma} f$. Recursively construct a Γ, Δ, C, r_1-reconstruction M_1 of D. Then $(\lambda x : C.N) M_1 \succ^{\mathcal{C}}_{\Gamma} fu$ and $A \succ^{\mathcal{C}}_{\Gamma} fu$, so $A =_{\beta\varepsilon} N[M_1/x]$, using Lemma 69. Also $N[M_1/x] \succ^{\mathcal{C}}_{\Gamma} r_2[u/x]$. Take $M = N[M_1/x]$.
- $(X : \psi) \in \Delta_E$ and $\psi = \forall x y x' y'. E(x, x') \to E(y, y') \to E(xy, x'y')$. Then $Q = X u w u' w' D_1 D_2$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : E(u, u')$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : E(w, w')$ and $t_0 = uw$ and $t_1 = u'w'$. Since $A \succ^{\mathcal{C}}_{\Gamma} uw$, we have $A = A_1 A_2$ with $A_1 \succ^{\mathcal{C}}_{\Gamma} u$ and $A_2 \succ^{\mathcal{C}}_{\Gamma} w$. Recursively construct a $\Gamma, \Delta, A_1, E(u, u')$-reconstruction B_1 of D_1 and a $\Gamma, \Delta, A_2, E(w, w')$-reconstruction B_2 of D_2. Take $M = B_1 B_2$.

Cases omitted in the above sketch are trivial or similar to other cases considered.

Together with Lemma 56, Lemma 59 and Lemma 5, Theorem 75 gives us the following.

▶ **Corollary 77.** *If $\Delta_{\mathrm{Ax}}, \lceil \Gamma \rceil \vdash \mathcal{F}_{\Gamma}(A)$ and $\Gamma \vdash A : *^p$ then there exists M such that $\Gamma \vdash M : A$.*

We now give a rigorous proof of the soundness of the embedding.

Proof of Theorem 75. We show that every first-order proof term Q in η-lnf is reconstructible. We proceed by induction on the size of Q. First of all, note that because of Lemma 68 for any M, Γ and $x \in V^{*^p}$ and φ, t, Δ with $M \succ^{\mathcal{F}}_{\Gamma} \varphi$, $M \succ^{\mathcal{C}}_{\Gamma} t$, $\Gamma \succ \Delta$, we have $x \notin \mathrm{FV}(\varphi, t, \Delta)$. Hence we may assume that if $x \in V^{*^p}$ then x does not occur free in any individual term used in Q.

We need to consider the three cases in Definition 73.

1. Assume $\Delta_{\mathrm{Ax}}, \Delta \vdash Q : \varphi$ and $\Gamma \succ \Delta$ and $\Gamma \vdash A : *^p$ and $A \succ^{\mathcal{F}}_{\Gamma} \varphi$. We need to find M with $\Gamma \vdash M : A$. We consider possible forms of φ.

- $\varphi = \varphi_1 \to \varphi_2$. Then $A = \Pi x : B.C$ and $\Gamma \vdash B : *^p$ and $B \succ^{\mathcal{F}}_{\Gamma} \varphi_1$ and $C \succ^{\mathcal{F}}_{\Gamma, x:B} \varphi_2$ and $Q = \lambda X : \varphi_1.Q'$. Hence $\Delta_{\mathrm{Ax}}, \Delta, X : \varphi_1 \vdash Q' : \varphi_2$. Note that $\Gamma, x : B \succ \Delta, X : \varphi_1$. Also $\Gamma, x : B \vdash C : *^p$ by Corollary 25. Thus by the inductive hypothesis there is M' with $\Gamma, x : B \vdash M' : C$. Because $\Gamma \vdash (\Pi x : B.C) : *^p$, by the abstraction rule we obtain $\Gamma \vdash (\lambda x : B.M') : (\Pi x : B.C)$. Hence take $M = \lambda x : B.M'$.
- $\varphi = \forall x. T(x, t) \to \psi$. Then $A = \Pi x : B.C$ and $\Gamma \nvdash B : *^p$ and $B \succ^{\mathcal{C}}_{\Gamma} t$ and $C \succ^{\mathcal{F}}_{\Gamma, x:B} \psi$. Hence $Q = \lambda x \lambda X : T(x, t).Q'$, so $\Delta_{\mathrm{Ax}}, \Delta, X : T(x, t) \vdash Q' : \psi$. Note that $\Gamma, x : B \succ \Delta, X : T(x, t)$. By Corollary 25 we have $\Gamma, x : B \vdash C : *^p$. Thus by the

inductive hypothesis there is M' with $\Gamma, x : B \vdash M' : C$. Since $\Gamma \vdash A : *^p$, by the abstraction rule we obtain $\Gamma \vdash (\lambda x : B.M') : A$. Hence take $M = \lambda x : B.M'$.

- $\varphi = P(t_A)$ with $A \succ^{\mathcal{C}}_{\Gamma} t_A$. Then $Q = XD_1 \dots D_k$ where $(X : \psi) \in \Delta_{\mathrm{Ax}}, \Delta$ and $\mathrm{target}(\psi) = P$, and each D_i is a first-order proof term in η-long normal form or an individual term. By the inductive hypothesis all proof terms among $D_1, \dots, D_k$ are reconstructible. We consider possible forms of ψ.
  - $(X : \psi) \in \Delta$. By Lemma 70 there are $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ and C such that $\Gamma_1 \succ \Delta_1$ and $\Gamma = \Gamma_1, x : C, \Gamma_2$ and $\Delta = \Delta_1, \psi, \Delta_2$ and $\Gamma_1 \vdash C : *^p$ and $C \succ^{\mathcal{F}}_{\Gamma_1} \psi$. By Lemma 72 we have $C = \Pi x_1 : A_1 \dots \Pi x_n : A_n.B$ and $P(t) \succ^{\mathbb{A}}_{\Gamma_1,\Gamma_0} \psi$ and $B \succ^{\mathcal{C}}_{\Gamma_1,\Gamma_0} t$ where $\Gamma_0 = x_1 : A_1, \dots, x_n : A_n$. We may assume that $x_1, \dots, x_n \notin \mathrm{dom}(\Gamma)$. Hence Γ, Γ_0 is a legal context and $\Gamma \supseteq \Gamma_1$, so $P(t) \succ^{\mathbb{A}}_{\Gamma;\Gamma_0} \psi$ by Corollary 63. By Lemma 74 there are $N_1, \dots, N_n$ and $u_1, \dots, u_n$ such that $\varphi = P(t_A) = P(t)[\vec{u}/\vec{x}]$, i.e. $t_A = t[\vec{u}/\vec{x}]$, and $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x},\vec{N},\vec{u}} \Gamma$. By Lemma 62 we have $B \succ^{\mathcal{C}}_{\Gamma,\Gamma_0} t$. Hence $B[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} t[\vec{u}/\vec{x}] = t_A$ by Corollary 66. Since also $A \succ^{\mathcal{C}}_{\Gamma} t_A$, by Lemma 69 we obtain $A =_{\varepsilon} B[\vec{N}/\vec{x}]$. Because $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x},\vec{N},\vec{u}} \Gamma$ we must have $\Gamma \vdash N_i : A_i[N_1/x_1] \dots [N_{i-1}/x_{i-1}]$ and $N_i \sim x_i$ for $i = 1, \dots, n$. Recall that $\Gamma \vdash x : \Pi x_1 : A_1 \dots \Pi x_n : A_n.B$. Hence, using the application rule n times we conclude that $\Gamma \vdash xN_1 \dots N_n : B[\vec{N}/\vec{x}]$. Thus $\Gamma \vdash xN_1 \dots N_n : A$ by the conversion rule.
  - $(X : \psi) \in \Delta_E$ and $\psi = \forall xx'.E(x, x') \to P(x) \to P(x')$. Then $k = 4$, $D_1 = t_1$, $D_2 = t_2$ are individual terms, and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_3 : E(t_1, t_2)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_4 : P(t_1)$ and $P(t_2) = P(t_A)$. Hence $t_2 = t_A$. Because D_3 is reconstructible (by induction), there is a Γ-term B with $B \succ^{\mathcal{C}}_{\Gamma} t_1$ and $B =_{\beta\varepsilon} A$. By Lemma 31 we have $\Gamma \vdash B : *^p$. Since $\Delta_{\mathrm{Ax}}, \Delta \vdash D_4 : P(t_1)$ and $B \succ^{\mathcal{C}}_{\Gamma} t_1$ and $\Gamma \vdash B : *^p$ and $\Gamma \succ \Delta$, because D_4 is reconstructible there is M with $\Gamma \vdash M : B$. By the conversion rule also $\Gamma \vdash M : A$.
  - $(X : \psi) \in \Delta_{\Phi}$ and $\psi = \forall \vec{y}.\psi' \to P(f\vec{y})$ and $\vec{y} = \mathrm{FV}(\psi')$ and $f = \Phi(\psi')$. Then $t_A = f\vec{t}$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash X\vec{t}D_k : P(f\vec{t})$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_k : \psi'[\vec{t}/\vec{y}]$ for some individual terms $t_1, \dots, t_n$ (without loss of generality we may assume that $\mathrm{FV}(t_i) \cap \mathrm{FV}(\psi') = \emptyset$). Since $A \succ^{\mathcal{C}}_{\Gamma} f\vec{t}$, by the definition of $\succ^{\mathcal{C}}_{\Gamma}$ there exist B, C and $N_1, \dots, N_m$ and $u_1, \dots, u_m$ and Γ' such that $\Gamma' \vdash (\Pi x : B.C) : *^p$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{u}} \Gamma$ and $A = (\Pi x : B.C)[\vec{N}/\vec{x}] \succ^{\mathcal{F}}_{\Gamma} \psi'[\vec{u}/\vec{x}]$ and $(\Pi x : B.C)[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{u}/\vec{x}] = f\vec{t}$. Let $u'_i = u_i[u_{i+1}/x_{i+1}] \dots [u_m/x_m]$. Because $f\vec{t} = (f\vec{y})[u'_1/x_1, \dots, u'_m/x_m]$ and $\vec{y} = \mathrm{FV}(\psi') = \{y_1, \dots, y_n\}$, without loss of generality we may assume $u'_i = t_i$ and $x_i = y_i$ for $i \leq n$, and $x_i \notin \mathrm{FV}(\psi')$ for $i > n$. Then $\psi'[\vec{u}/\vec{x}] = \psi'[u'_1/x_1, \dots, u'_m/x_m] = \psi'[u'_1/y_1, \dots, u'_n/y_n] = \psi'[\vec{t}/\vec{y}]$. By Lemma 61 we have $\Gamma \vdash (\Pi x : B.C)[\vec{N}/\vec{x}] : *^p$. Since also $A = (\Pi x : B.C)[\vec{N}/\vec{x}] \succ^{\mathcal{F}}_{\Gamma} \psi'[\vec{t}/\vec{y}]$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_k : \psi'[\vec{t}/\vec{y}]$, because D_k is reconstructible there exists M such that $\Gamma \vdash M : A$.

2. Assume $\Delta_{\mathrm{Ax}}, \Delta \vdash Q : T(t, t')$ and $\Gamma \succ \Delta$ and $\Gamma \vdash A : s$ and $s \neq *^p$ and $A \succ^{\mathcal{C}}_{\Gamma} t'$. We need to find M with $M \succ^{\mathcal{C}}_{\Gamma} t$ and $\Gamma \vdash M : A$. Since $T(t, t')$ is an atom and Q is in η-lnf, we have $Q = X\vec{D}$ where $(X : \psi) \in \Delta_{\mathrm{Ax}}, \Delta$ and $\mathrm{target}(\psi) = T$ and $\vec{D}$ is a sequence of first-order individual terms and reconstructible (by induction) proof terms in η-lnf. We consider possible forms of ψ.

- $(X : \psi) \in \Delta$. By Lemma 71 we have $\psi = T(x, r)$ and there are Γ_1, Γ_2 and C such that $\Gamma = \Gamma_1, x : C, \Gamma_2$ and $C \succ^{\mathcal{C}}_{\Gamma_1} r$. Then $t = x$ and $t' = r$. By Lemma 62 we have $C \succ^{\mathcal{C}}_{\Gamma} t'$. Since also $A \succ^{\mathcal{C}}_{\Gamma} t'$, by Lemma 69 we obtain $A =_{\varepsilon} C$. Because $\Gamma \vdash x : C$ and $\Gamma \vdash A : s$, by the conversion rule $\Gamma \vdash x : A$.
- $(X : \psi) \in \Delta_{\mathcal{G}_1}$ and $\psi = \mathbb{A}_{\Gamma'}(\forall z.T(z, f\vec{y}) \to \forall x.T(x, r_1) \to T(zx, r_2))$ where $f = \mathcal{G}_1(x, r_1, r_2, s)$ and $\vec{y} = \mathrm{FV}(r_1, r_2) \setminus \{x\}$ and $z \notin \mathrm{FV}(r_1, r_2)$ and $\Gamma' = x_1 : A_1, \dots, x_n : A_n$. Then $Q = X\vec{R}uP_1wP_2$ and there are C_1, C_2 such that $r_1 = \mathcal{C}_{\Gamma'}(C_1)$ and $r_2 =$

$\mathcal{C}_{\Gamma',x:C_1}(C_2)$ and $\Gamma' \vdash (\Pi x : C_1.C_2) : s$ and $s \neq *^p$ and $\Gamma' \nvdash C_1 : *^p$. Note that ψ is closed, because $\vec{y} = \mathrm{FV}(r_1, r_2) \setminus \{x\} \subseteq \mathrm{dom}(\Gamma')$ by Lemma 68. Hence, by Lemma 53 we may assume $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$, possibly renaming the variables in Γ', ψ, r_1, r_2 and $\Pi x : C_1.C_2$. Hence Γ, Γ' is a legal context. Let $\psi' = \forall z.T(z, f\vec{y}) \to \forall x.T(x, r_1) \to T(zx, r_2)$. By Lemma 59 and Corollary 63 we have $\psi' \succ^{\mathbb{A}}_{\Gamma;\Gamma'} \psi$. By Lemma 74 there are $N_1, \ldots, N_n$ and $u_1, \ldots, u_n$ such that $\Gamma, \Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{u}} \Gamma$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash X\vec{R} : \psi'[\vec{u}/\vec{x}]$. Note that $z \notin \vec{y}$. Hence $\Delta_{\mathrm{Ax}}, \Delta \vdash P_1 : T(u, (f\vec{y})[\vec{u}/\vec{x}])$. Let $C_i' = C_i[\vec{N}/\vec{x}]$. By Lemma 56 and Lemma 60 and Corollary 66 we have $C_1' \succ^{\mathcal{C}}_{\Gamma} r_1[\vec{u}/\vec{x}]$ and $C_2' \succ^{\mathcal{C}}_{\Gamma,x:C_1[\vec{N}/\vec{x}]} r_2[\vec{u}/\vec{x}]$. Also $\Gamma, \Gamma' \vdash (\Pi x : C_1.C_2) : s$ by the thinning lemma, and $\Gamma, \Gamma' \nvdash C_1 : *^p$ by the generation, thinning and uniqueness of types lemmas. Hence $\Pi x : C_1'.C_2' \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{u}/\vec{x}]$. We also have $\Gamma \vdash (\Pi x : C_1'.C_2') : s$ (and $s \neq *^p$) by Lemma 61. Because P_1 is reconstructible, there is M_1 with $\Gamma \vdash M_1 : (\Pi x : C_1'.C_2')$ and $M_1 \succ^{\mathcal{C}}_{\Gamma} u$. Note that $x \notin \mathrm{FV}(r_1)$ by Lemma 68, because $C_1 \succ^{\mathcal{C}}_{\Gamma'} r_1$ and $x \notin \mathrm{dom}(\Gamma')$. Since also $z \notin \mathrm{FV}(r_1)$, we have $\Delta_{\mathrm{Ax}}, \Delta \vdash P_2 : T(w, r_1[\vec{u}/\vec{x}])$. Since $\Gamma, \Gamma' \nvdash C_1 : *^p$ and $\Gamma, \Gamma' \vdash (\Pi x : C_1.C_2) : s$, by the generation lemma and Lemma 61 we have $\Gamma \vdash C_1' : s'$ for some $s' \in \mathcal{S}$, $s' \neq *^p$. Since also $C_1' \succ^{\mathcal{C}}_{\Gamma} r_1[\vec{u}/\vec{x}]$, because P_2 is reconstructible there is M_2 with $M_2 \succ^{\mathcal{C}}_{\Gamma} w$ and $\Gamma \vdash M_2 : C_1'$. By Lemma 32 we have $M_2 \sim x$. Hence $\Gamma \vdash M_1M_2 : C_2'[M_2/x]$ by the application rule. Because M_1 is not a Γ-proof (recall that $\Gamma \vdash M_1 : (\Pi x : C_1'.C_2') : s$ with $s \neq *^p$), neither is M_1M_2 by Theorem 30. Hence $M_1M_2 \succ^{\mathcal{C}}_{\Gamma} uw = t$. Since $C_2' \succ^{\mathcal{C}}_{\Gamma,x:C_1'} r_2[\vec{u}/\vec{x}]$, by Lemma 65 we have $C_2'[M_2/x] \succ^{\mathcal{C}}_{\Gamma} r_2[\vec{u}/\vec{x}][u/x] = t'$. Since also $A \succ^{\mathcal{C}}_{\Gamma} t'$, we have $C_2'[M_2/x] =_{\varepsilon} A$ by Lemma 69. Thus $\Gamma \vdash M_1M_2 : A$ by the conversion rule. Therefore, we may take $M = M_1M_2$.

- $(X : \psi) \in \Delta_{\mathcal{G}_0}$. This case is analogous to the previous one.
- $(X : \psi) \in \Delta_{\mathcal{T}_1}$ and $\psi = \mathbb{A}_{\Gamma'}(T(f\vec{y}, g\vec{z}))$ where $f = \Lambda_1(x, r, u)$ and $g = \mathcal{G}_1(x, r, w, s)$ and $\vec{y} = \mathrm{FV}(r, u) \setminus \{x\}$ and $\vec{z} = \mathrm{FV}(r, w) \setminus \{x\}$ and $\Gamma' = x_1 : A_1, \ldots, x_n : A_n$. Then $Q = X\vec{R}$ and there are C_1, C_2, N such that $r = \mathcal{C}_{\Gamma'}(C_1)$ and $u = \mathcal{C}_{\Gamma',x:C_1}(N)$ and $w = \mathcal{C}_{\Gamma',x:C_1}(C_2)$ and $\Gamma' \vdash (\lambda x : C_1.N) : \Pi x : C_1.C_2 : s$ and $\Gamma' \nvdash C_1 : *^p$. By Lemma 53 we may assume that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$, possibly renaming the variables in Γ', ψ, r, u, w and $\Pi x : C_1.C_2$ and $\lambda x : C_1.N$. Hence Γ, Γ' is a legal context. Let $\psi' = T(f\vec{y}, g\vec{z})$. By Lemma 59 and Corollary 63 we have $\psi' \succ^{\mathbb{A}}_{\Gamma;\Gamma'} \psi$. By Lemma 74 there are $N_1, \ldots, N_n$ and $u_1, \ldots, u_n$ such that $\Gamma, \Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{u}} \Gamma$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash X\vec{R} : \psi'[\vec{u}/\vec{x}]$. We thus have $t = (f\vec{y})[\vec{u}/\vec{x}]$ and $t' = (g\vec{z})[\vec{u}/\vec{x}]$. By the generation lemma, the thinning lemma and the uniqueness of types lemma $\Gamma, \Gamma' \nvdash C_1 : *^p$. By the generation lemma C_1 is a Γ'-subject and N is a $\Gamma', x : C_1$-subject. Hence $C_1[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} r[\vec{u}/\vec{x}]$ and $N[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:C_1[\vec{N}/\vec{x}]} u[\vec{u}/\vec{x}]$ by Lemma 56 and Corollary 66. Hence by Definition 55 we have $(\lambda x : C_1.N)[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{u}/\vec{x}]$. Also $\Gamma \vdash (\lambda x : C_1.N)[\vec{N}/\vec{x}] : C[\vec{N}/\vec{x}]$ by Lemma 60, where $C = \Pi x : C_1.C_2$. We have $C_2[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:C_1[\vec{N}/\vec{x}]} u[\vec{u}/\vec{x}]$ by Lemma 56 and Corollary 66. Also $\Gamma, \Gamma' \vdash C : s$ by the thinning lemma. Thus $C[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (g\vec{z})[\vec{u}/\vec{x}] = t'$ by Definition 55. Since also $A \succ^{\mathcal{C}}_{\Gamma} t'$, by Lemma 69 we obtain $A =_{\varepsilon} C[\vec{N}/\vec{x}]$. Thus $\Gamma \vdash (\lambda x : C_1.N)[\vec{N}/\vec{x}] : A$ by the conversion rule. So we may take $M = (\lambda x : C_1.N)[\vec{N}/\vec{x}]$.
- $(X : \psi) \in \Delta_{\mathcal{T}_0}$. This case is analogous to the previous one.
- $(X : \psi) \in \Delta_E$. Then $\psi = \forall xyx'y'.E(x, x') \to E(y, y') \to T(x, y) \to T(x', y')$ and $Q = Xuu'tt'D_1D_2D_3$ where $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : E(u, t)$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : E(u', t')$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_3 : T(u, u')$. Since $A \succ^{\mathcal{C}}_{\Gamma} t'$, because D_2 is reconstructible there exists a Γ-term A' such that $A' \succ^{\mathcal{C}}_{\Gamma} u'$ and $A' =_{\beta\varepsilon} A$. Since $\Gamma \vdash A : s$ ($s \neq *^p$), by Lemma 31 we have $\Gamma \vdash A' : s$. Hence, because D_3 is reconstructible there exists M' such that

$M' \succ^{\mathcal{C}}_{\Gamma} u$ and $\Gamma \vdash M' : A'$. Because D_1 is reconstructible there is a Γ-subject M with $M \succ^{\mathcal{C}}_{\Gamma} t$ and $M =_{\beta\varepsilon} M'$. By the uniqueness of types lemma and Theorem 30 we have $M' \not\to^*_{\varepsilon} \varepsilon$, hence also $M \not\to^*_{\varepsilon} \varepsilon$ by Lemma 18. Since M is a Γ-subject, there is B with $\Gamma \vdash M : B$. Then $B =_{\beta\varepsilon} A'$ by the second point in the uniqueness of types lemma. Since $\Gamma \vdash A : s$ and $B =_{\beta\varepsilon} A' =_{\beta\varepsilon} A$, we have $\Gamma \vdash M : A$ by the conversion rule. Therefore, we have found M with $M \succ^{\mathcal{C}}_{\Gamma} t$ and $\Gamma \vdash M : A$, as desired.

3. Assume $\Delta_{\mathrm{Ax}}, \Delta \vdash Q : E(t_0, t_1)$ and $\Gamma \succ \Delta$ and $M \succ^{\mathcal{C}}_{\Gamma} t_q$ with $q \in \{0, 1\}$. We need to find N with $N \succ^{\mathcal{C}}_{\Gamma} t_{1-q}$ and $N =_{\beta\varepsilon} M$. Since $E(t_0, t_1)$ is an atom and Q is in η-lnf, $Q = X\vec{D}$ where $(X : \psi) \in \Delta_{\mathrm{Ax}}, \Delta$ and $\mathrm{target}(\psi) = E$ and $\vec{D}$ is a sequence of first-order individual terms and reconstructible (by induction) proof terms in η-lnf. We consider possible forms of ψ.

- $(X : \psi) \in \Delta_{\Lambda_0}$ and $\psi = \mathbb{A}_{\Gamma'}(\varphi \to E(f\vec{y}\varepsilon, r))$ where $\Gamma' = x_1 : A_1, \ldots, x_n : A_n$ and $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$ (we may assume this by Lemma 53) and $f = \Lambda_0(x, \varphi, r)$ and $\vec{y} = \mathrm{FV}(\varphi, r) \setminus \{x\}$ and there are B, C_1, C_2 with $\varphi = \mathcal{F}_{\Gamma'}(C_1)$ and $r = \mathcal{C}_{\Gamma', x:C_1}(C_2)$ and $\Gamma' \vdash (\lambda x : C_1.C_2) : B$ and $\Gamma' \vdash C_1 : *^p$ and $\Gamma' \vdash B : s$ and $s \neq *^p$. We have $\psi = \mathbb{A}_{\Gamma', x:C_1}(E(f\vec{y}\varepsilon, r))$. We may assume $x \notin \mathrm{dom}(\Gamma)$, so $\Gamma, \Gamma', x : C_1$ is a legal context. Thus by Lemma 59 and Corollary 63 we obtain $E(f\vec{y}\varepsilon, r) \succ^{\mathbb{A}}_{\Gamma;\Gamma', x:C_1} \psi$. Hence by Lemma 74 there are $N_1, \ldots, N_n, U$ and $u_1, \ldots, u_n, u$ such that $E(t_0, t_1) = E(f\vec{y}\varepsilon, r)[\vec{u}/\vec{x}][u/x]$ and $\Gamma, \Gamma', x : C_1 \rightsquigarrow_{\vec{x}, x, \vec{N}, U, \vec{u}, u} \Gamma$. Note that then also $\Gamma, \Gamma' \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma$ and $\Gamma, \Gamma', x : C_1 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma, x : C_1[\vec{N}/\vec{x}]$. We have $C_1 \succ^{\mathcal{F}}_{\Gamma, \Gamma'} \varphi$ and $C_2 \succ^{\mathcal{C}}_{\Gamma, \Gamma', x:C_1} r$ by Lemma 56 and Lemma 62. Hence $C_1[\vec{N}/\vec{x}] \succ^{\mathcal{F}}_{\Gamma} \varphi[\vec{u}/\vec{x}]$ and $C_2[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma, x:C_1[\vec{N}/\vec{x}]} r[\vec{u}/\vec{x}]$ by Corollary 66. Also $\Gamma, \Gamma' \vdash (\lambda x : C_1.C_2) : B : s$ and $\Gamma, \Gamma' \vdash C_1 : *^p$ by the thinning lemma. Hence $\Gamma \vdash (\lambda x : C_1.C_2)[\vec{N}/\vec{x}] : B[\vec{N}/\vec{x}] : s$ by Lemma 61, i.e., $(\lambda x : C_1.C_2)[\vec{N}/\vec{x}]$ is not a Γ-proof (by the uniqueness of types lemma, recalling that $s \neq *^p$). Thus $(\lambda x : C_1.C_2)[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{u}/\vec{x}]$. Since $N_i \succ^{\mathcal{C}}_{\Gamma} u_i$ and N_i is a Γ-term and $x \notin \mathrm{dom}(\Gamma)$, by Lemma 68 and the free variable lemma we obtain $x \notin \mathrm{FV}(u_1, \ldots, u_n)$. Also $x \notin \vec{y}$. Hence $(f\vec{y})[\vec{u}/\vec{x}] = (f\vec{y})[\vec{u}/\vec{x}][u/x]$. Because $\Gamma, \Gamma', x : C_1 \rightsquigarrow_{\vec{x}, x, \vec{N}, U, \vec{u}, u} \Gamma$, we have $U \sim x$ and $\Gamma \vdash U : C_1[\vec{N}/\vec{x}] : *^p$. Hence U is a Γ-proof, and thus $U \succ^{\mathcal{C}}_{\Gamma} \varepsilon$. Because $(\lambda x : C_1.C_2)[\vec{N}/\vec{x}]$ is not a Γ-proof, neither is $((\lambda x : C_1.C_2)[\vec{N}/\vec{x}])U$, by Theorem 30. Therefore $((\lambda x : C_1.C_2)[\vec{N}/\vec{x}])U \succ^{\mathcal{C}}_{\Gamma} ((f\vec{y})[\vec{u}/\vec{x}][u/x])\varepsilon = t_0$. We also have $C_2[\vec{N}/\vec{x}][U/x] \succ^{\mathcal{C}}_{\Gamma} r[\vec{u}/\vec{x}][u/x] = t_1$ by Corollary 66. Note that $((\lambda x : C_1.C_2)[\vec{N}/\vec{x}])U =_{\beta} C_2[\vec{N}/\vec{x}][U/x]$. First assume $q = 0$, i.e., $M \succ^{\mathcal{C}}_{\Gamma} t_0$. Using Lemma 69 we obtain $M =_{\beta\varepsilon} C_2[\vec{N}/\vec{x}][U/x] \succ^{\mathcal{C}}_{\Gamma} t_1$. Now assume $q = 1$, i.e., $M \succ^{\mathcal{C}}_{\Gamma} t_1$. Using Lemma 69 we obtain $M =_{\beta\varepsilon} ((\lambda x : C_1.C_2)[\vec{N}/\vec{x}])U \succ^{\mathcal{C}}_{\Gamma} t_0$. Also $((\lambda x : C_1.C_2)[\vec{N}/\vec{x}])U$ and $C_2[\vec{N}/\vec{x}][U/x]$ are Γ-subjects, by the generation lemma, the application rule (recall that $U \sim x$) and the subject reduction theorem.
- $(X : \psi) \in \Delta_{\Lambda_1}$. This case is analogous to the case $(X : \psi) \in \Delta_{\Lambda_0}$.
- $(X : \psi) \in \Delta_E$ and $\psi = \forall x.E(x, x)$. This case follows from reflexivity of $=_{\beta\varepsilon}$.
- $(X : \psi) \in \Delta_E$ and $\psi = \forall xy.E(x, y) \to E(y, x)$. This case follows directly from the inductive hypothesis.
- $(X : \psi) \in \Delta_E$ and $\psi = \forall xyz.E(x, y) \to E(y, z) \to E(x, z)$. This case follows from the inductive hypothesis and the transitivity of $=_{\beta\varepsilon}$.
- $(X : \psi) \in \Delta_E$ and $\psi = \forall xyx'y'.E(x, x') \to E(y, y') \to E(xy, x'y')$. Then $Q = Xuwu'w'D_1D_2$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_1 : E(u, u')$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash D_2 : E(w, w')$ and $t_0 = uw$ and $t_1 = u'w'$. Assume $q = 0$, i.e., $M \succ^{\mathcal{C}}_{\Gamma} uw$ (the case $q = 1$ is analogous). Then $M = M_1M_2$ with $M_1 \succ^{\mathcal{C}}_{\Gamma} u$ and $M_2 \succ^{\mathcal{C}}_{\Gamma} w$. Since M is a Γ-subject, by the generation lemma $\Gamma \vdash M_1 : \Pi z : A.B$ and $\Gamma \vdash M_2 : A$ and $M_2 \sim z$ for some A, B. Because D_1, D_2 are reconstructible there are Γ-subjects N_1, N_2 such that $N_i =_{\beta\varepsilon} M_i$ and $N_1 \succ^{\mathcal{C}}_{\Gamma} u'$

and $N_2 \succ_\Gamma^\mathcal{C} w'$. Because $\mathrm{nf}_\varepsilon(M) \neq \varepsilon$, also $\mathrm{nf}_\varepsilon(M_1) \neq \varepsilon$. Hence by the second point in the uniqueness of types lemma, the generation lemma and the conversion rule $\Gamma \vdash N_1 : \Pi z : A.B$. Without loss of generality we may assume $\mathrm{nf}_\varepsilon(M_2) \neq \varepsilon$, because otherwise $u' = w' = \varepsilon$ and we may take $N_2 = M_2$. If $\mathrm{nf}_\varepsilon(M_2) \neq \varepsilon$ then analogously as with M_1 we conclude $\Gamma \vdash N_2 : A$. Note that also $N_2 \sim z$, because $M_2 \sim z$ and $M_2 =_{\beta\varepsilon} N_2$. Hence $N_1 N_2$ is a Γ-subject by the application rule. Using Theorem 30 we may also conclude that $N_1 N_2$ is not a Γ-proof. Thus $M =_{\beta\varepsilon} N_1 N_2 \succ_\Gamma^\mathcal{C} u'w' = t_1$. ◀

6 Conclusions and related work

Below we make a few remarks on the embedding, the soundness proof and related work.

▶ Remark. In the literature there are various translations of languages with dependent types to less expressive logics, but as far as we know none of them are both shallow, include the Calculus of Constructions as the source formalism, and target first-order logic. The paper [16] defines a deep embedding of the Calculus of Constructions into a higher-order logic and shows it complete. In [17] a similar deep embedding of LF into a fragment of higher-order logic is shown sound and complete. The paper [21] shows how to simulate dependent types in higher-order logic.

In [27] a translation from first-order logic with dependent types into ordinary first-order logic is shown sound by model-theoretic methods. The aim of [27] is also to use the translation with first-order ATPs. The logic is much simpler than dependent type theory – it allows dependent types, but not function types, i.e., no λ-abstraction or partial application is possible.

The paper [29] defines a sound and complete deep embedding Tri of Martin-Löf's type theory into first-order logic. The embedding is deep in the sense that e.g. $b \in B$ is translated to $\mathrm{In}(b, B)$, so b is not erased. For a fragment F_2, which essentially disallows dependent function types as arguments, the translation may be optimized to a shallow one, i.e., $\mathrm{In}(b, B)$ is optimised to $\mathrm{Inh}(B)$. This restriction corresponds to disallowing quantifiers on the left side of implication, which makes it possible to prove soundness and completeness of the embedding. In contrast to our approach, since there is no separate sort of propositions, all terms inhabiting types are erased, not only those intuitively representing proofs of propositions.

The general ideas behind the translations in [29, 27] are broadly similar to ours, but our work is not a direct extension of any of them.

The paper [19] defines a translation Tr from λP to FOL in order to show a conservativity result. The general idea of Tr, to translate a dependent type $\Pi x : A.B$ into a quantification and an implication, is similar to how we translate piPTS propositions. Essentially, the translation Tr is defined only for terms that "originate from" an embedding of FOL into λP, not on arbitrary λP-terms.

In [26] an essentially deep embedding from LF to the higher-order hereditary Harrop language is shown sound and complete. It is deep because even in its optimised variant the proof terms are retained as additional arguments. On the other hand, it allows to omit more type guards than our translation.

The report [1] defines and proves sound a translation from a fragment of the dependently typed F* language to intuitionistic first-order logic. The soundness proof uses a broadly similar method to the one in this paper, using induction on first-order proof terms in η-long normal form. However, the considered language fragment is essentially simpler and the soundness proof does not have to deal with the problems mentioned at the beginning of

Section 5, or with proof irrelevance. On the other hand, the target fragment of first-order logic is richer and includes conjunction and falsity.

▶ Remark. Our soundness proof relies on proof-irrelevance incorporated into the piPTS conversion rule. Proof-irrelevance is necessary for the soundness of a shallow embedding. It is an open question if the embedding is sound for the Calculus of Constructions with proof-irrelevance expressed by axioms.

▶ Remark. Note the use of the function $\mathbb{A}_\Gamma$ in the axioms in Δ_{Λ_0}, Δ_{Λ_1}, $\Delta_{\mathcal{G}_0}$ and $\Delta_{\mathcal{G}_1}$. In contrast, for the axioms in Δ_Φ the use of $\mathbb{A}_\Gamma$ is not necessary – we may simply quantify over the free variables without requiring them a priori to have the right types. This is because they all occur in the target atom $P(f\vec{y})$ which in the soundness proof is assumed to encode a well-typed term. This is not necessarily true for the axioms which use $\mathbb{A}_\Gamma$, and our soundness proof cannot be easily adapted to avoid the use of $\mathbb{A}_\Gamma$.

Nonetheless, we expect that the use of $\mathbb{A}_\Gamma$ could be avoided without compromising soundness. For instance, the axioms in Δ_{Λ_1} could be $\forall \vec{y}x.T(x,r) \to E(f\vec{y}x,t)$ or even $\forall \vec{y}x.E(f\vec{y}x,t)$. We expect the embedding to remain sound after this modification, because we would essentially omit the type information only for free variables of subterms that are "lifted out" of already well-typed terms. The problems that arise in the study of such a modified embedding are broadly similar to problems that arise in the study of systems of illative combinatory logic [3, 13] or the "liberal" Pure Type Systems from [10]. Domain-free Pure Type Systems [7], domain-free variants of the Calculus of Inductive Constructions [4], the Implicit Calculus of Constructions [23] and generally the work on ignoring computationally irrelevant information also seem related.

In fact, in the practical translation from [15] we omit type information for free variables of the terms "lifted-out" by the translation. This may increase the success rate in some circumstances, as then the formulas are simpler and the ATPs do not need to prove too many well-typedness conditions. See [15, Section 5.6].

▶ Remark. In [8, 11] it is shown that in a translation from (polymorphic) many-sorted classical first-order logic to untyped classical first-order logic much of the type information may be omitted using monotonicity inference. The methods of the cited papers are model-theoretic, so they are probably not useful in our setting. Nonetheless, it is an interesting problem to investigate the possibility of adapting monotonicity inference to embeddings of constructive dependent type theory into first-order logic.

References

1 A. Aguirre. Towards a provably correct encoding from F^* to SMT. Technical report, INRIA, 2016.

2 H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992.

3 H. Barendregt, M. Bunder, and W. Dekkers. Systems of illative combinatory logic complete for first-order propositional and predicate calculus. *J. Symb. Logic*, 58(3):769–788, 1993.

4 B. Barras and B. Grégoire. On the role of type decorations in the calculus of inductive constructions. In *CSL 2005*, pages 151–166, 2005.

5 G. Barthe. The relevance of proof-irrelevance. In *ICALP'98*, pages 755–768, 1998.

6 G. Barthe, J. Hatcliff, and M.H. Sørensen. A notion of classical pure type system. *Electr. Notes Theor. Comput. Sci.*, 6:4–59, 1997.

7 G. Barthe and M.H. Sørensen. Domain-free pure type systems. *J. Funct. Program.*, 10(5):417–452, 2000.

8 J. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science*, 12(4), 2016.

9 J. Blanchette, C. Kaliszyk, L. Paulson, and J. Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

10 M. Bunder and W. Dekkers. Pure type systems with more liberal rules. *J. Symb. Logic*, 66(4):1561–1580, 2001.

11 K. Claessen, A. Lillieström, and N. Smallbone. Sort it out with monotonicity. In *CADE 2011*, pages 207–221. Springer, 2011.

12 T. Coquand and H. Herbelin. A-translation and looping combinators in pure type systems. *J. Funct. Program.*, 4(1):77–88, 1994.

13 Ł. Czajka. Higher-order illative combinatory logic. *J. Symb. Logic*, 73(3):837–872, 2013.

14 Ł. Czajka and C. Kaliszyk. Goal translation for a hammer for Coq (extended abstract). In *HaTT 2016*, volume 210 of *EPTCS*, pages 13–20, 2016.

15 Ł. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.

16 A. Felty. Encoding the calculus of constructions in a higher-order logic. In *LICS '93*, pages 233–244, 1993.

17 A. Felty and D. Miller. Encoding a dependent-type lambda-calculus in a logic programming language. In *CADE '90*, pages 221–235, 1990.

18 H. Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993.

19 H. Geuvers and E. Barendsen. Some logical and syntactical observations concerning the first-order dependent type system lambda-P. *Mathematical Structures in Computer Science*, 9(4):335–359, 1999.

20 H. Geuvers and M.-J. Nederhof. Modular proof of strong normalization for the Calculus of Constructions. *J. Funct. Program.*, 1(2):155–189, 1991.

21 B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In *TLCA '93*, pages 209–229, 1993.

22 L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, pages 1–35, 2013.

23 A. Miquel. The implicit calculus of constructions. In *TLCA 2001*, pages 344–359, 2001.

24 A. Miquel and B. Werner. The not so simple proof-irrelevant model of CC. In *TYPES 2002*, volume 2646 of *LNCS*. Springer, 2003.

25 S. Schulz. System description: E 1.8. In *LPAR 2013*, pages 735–743, 2013.

26 Z. Snow, D. Baelde, and G. Nadathur. A meta-programming approach to realizing dependently typed logic programming. In *PPDP '10*, pages 187–198, 2010.

27 K. Sojakova and F. Rabe. Translating a dependently-typed logic to first-order logic. In *WADT 2008*, pages 326–341, 2008.

28 M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

29 T. Tammet and J.M. Smith. Optimized encodings of fragments of type theory in first order logic. In *TYPES '95*, pages 265–287, 1995.

30 B. Werner. On the strength of proof-irrelevant type theories. *Logical Methods in Computer Science*, 4(3:13):1–20, 2008.

A Properties of proof-irrelevant Pure Type Systems

In this appendix we develop the meta-theory of proof-irrelevant Pure Type Systems. The development follows [2, Section 5.2] and is mostly standard, except for one difficulty caused by the mismatch between $\beta\varepsilon$-reduction used in the conversion rule and β-reduction for which the subject reduction theorem holds.

First, we need some lemmas concerning ε-reduction, β-reduction and $\beta\varepsilon$-reduction. Note that $\to_\varepsilon$ is not closed under substitutions. As a result, neither is $\to_\beta$, because it depends on $\to_\varepsilon$ through the side condition $N \sim x$. For example, $M = (\lambda x^{*^p} : A.x^{*^p})y^{*^p} \to_\beta y^{*^p}$. But $M[*^p/y^{*^p}] \not\to_\beta *^p$ because $*^p \not\sim x^{*^p}$. However, both relations are closed under substitutions if the condition $N \sim x$ is required when substituting N for x.

▶ **Lemma 78.** *If $M \to_\varepsilon M'$ and $N \sim x$ then $M[N/x] \to_\varepsilon^* M'[N/x]$.*

Proof. Induction on M. The assumption $N \sim x$ is needed when $x \in V^{*^p}$ and $M = x \to_\varepsilon \varepsilon$. ◀

▶ **Lemma 79** (Confluence and strong normalisation of ε-reduction)**.** *ε-reduction is confluent and strongly normalising.*

Proof. It is obvious that ε-reduction is strongly normalising. One also easily checks that the reflexive closure of $\to_\varepsilon$ has the diamond property. ◀

▶ **Corollary 80.** *If $M \to_\varepsilon^* M'$ and $M \sim x$ then $M' \sim x$.*

▶ **Lemma 16.** *If $N \sim x$ then $\mathrm{nf}_\varepsilon(M[N/x]) = \mathrm{nf}_\varepsilon(M)[\mathrm{nf}_\varepsilon(N)/x]$.*

Proof. Note that $M[N/x] \to_\varepsilon^* \mathrm{nf}_\varepsilon(M)[\mathrm{nf}_\varepsilon(N)/x]$ by Lemma 78. It suffices to show that the latter term is in ε-normal form. Otherwise, $\mathrm{nf}_\varepsilon(M)$ must have a subterm of the form xt or $\lambda y.x$, and $\mathrm{nf}_\varepsilon(N) = \varepsilon$. But then $x \in V^{*^p}$, which contradicts the fact that $\mathrm{nf}_\varepsilon(M)$ is in ε-normal form. ◀

▶ **Lemma 81.** *If $M \sim x$ and $N \sim y$ then $M[N/y] \sim x$.*

Proof. Follows directly from Lemma 16. ◀

▶ **Lemma 82.** *If $M \to_\beta M'$ and $N \sim x$ then $M[N/x] \to_\beta M'[N/x]$.*

Proof. Induction on M, using Lemma 81. ◀

▶ **Lemma 83.** *If $M \to_{\beta\varepsilon}^* M'$ and $N \to_{\beta\varepsilon}^* N'$ and $N \sim x$ then $M[N/x] \to_{\beta\varepsilon}^* M'[N'/x]$.*

Proof. Using Lemma 78 and Lemma 82 repeatedly we obtain $M[N/x] \to_{\beta\varepsilon}^* M'[N/x]$. Since $N \to_{\beta\varepsilon}^* N'$, we have $M'[N/x] \to_{\beta\varepsilon}^* M'[N'/x]$. ◀

▶ **Lemma 84.** *If $M \to_\beta M_1$ and $M \to_\varepsilon M_2$ then there is M' with $M_1 \to_\varepsilon^* M'$ and $M_2 \to_{\beta\varepsilon} M'$.*

Proof. Induction on M. The interesting case is when $M = (\lambda x : A.B)C \to_\beta B[C/x] = M_1$. Then $C \sim x$. First assume $M_2 = (\lambda x : A.B)C'$ with $C \to_\varepsilon C'$. Then $C' \sim x$ by Corollary 80. Hence $M_2 \to_\beta B[C'/x]$. We also have $B[C/x] \to_\varepsilon^* B[C'/x]$, so we may take $M' = B[C'/x]$. Now assume $M_2 = (\lambda x : A.B')C$ with $B \to_\varepsilon B'$. Then $B[C/x] \to_\varepsilon^* B'[C/x]$ by Lemma 78. Also $M_2 \to_\beta B'[C/x]$, so we may take $M' = B'[C/x]$. Finally, assume $M_2 = \varepsilon C$ where $B = \varepsilon$ and $\lambda x : A.B \to_\varepsilon \varepsilon$. Then $M_1 = B[C/x] = \varepsilon$. Since $M_2 = \varepsilon C \to_\varepsilon \varepsilon$, we may take $M' = \varepsilon$.

The remaining cases are easy. ◀

▶ **Corollary 85.** *If $M \to_\beta^* M_1$ and $M \to_\varepsilon^* M_2$ then there is M' with $M_1 \to_\varepsilon^* M'$ and $M_2 \to_{\beta\varepsilon}^* M'$.*

▶ **Lemma 86.** *If $M \to_\beta N \to_\varepsilon^* \varepsilon$ then $M \to_\varepsilon^* \varepsilon$.*

Proof. Induction on the length of the reduction $N \to_\varepsilon^* \varepsilon$.

If $M = (\lambda x.M')Q$ and $N = M'[Q/x]$ and $Q \sim x$ then $\mathrm{nf}_\varepsilon(N) = \mathrm{nf}_\varepsilon(M')[\mathrm{nf}_\varepsilon(Q)/x]$ by Lemma 16. Hence $\mathrm{nf}_\varepsilon(M')[\mathrm{nf}_\varepsilon(Q)/x] = \varepsilon$. This is possible if either $\mathrm{nf}_\varepsilon(M') = \varepsilon$, or $\mathrm{nf}_\varepsilon(M') = x$ and $\mathrm{nf}_\varepsilon(Q) = \varepsilon$. If $\mathrm{nf}_\varepsilon(M') = \varepsilon$ then $M \to_\varepsilon^* \varepsilon$. In the other case $x \in V^{*^p}$ because $Q \sim x$ and $Q \to_\varepsilon^* \varepsilon$. Hence also $M \to_\varepsilon^* \varepsilon$.

If $M = M'Q$ and $N = N'Q$ then $M' \to_\beta N' \to_\varepsilon^* \varepsilon$. Then $M' \to_\varepsilon^* \varepsilon$ by the inductive hypothesis, and thus $M \to_\varepsilon^* \varepsilon$.

Otherwise $M = \lambda x.M'$ and $N = \lambda x.N'$ and $M' \to_\beta N'$. Then $M' \to_\beta N' \to_\varepsilon^* \varepsilon$, so by the inductive hypothesis $M' \to_\varepsilon^* \varepsilon$. Hence $M \to_\varepsilon^* \varepsilon$. ◀

▶ **Corollary 87.** *If $M \to_{\beta\varepsilon}^* M'$ then $M \sim x$ iff $M' \sim x$.*

▶ **Lemma 88** (Postponement of ε-reduction)**.** *If $M \to_{\beta\varepsilon}^* M'$ then there exists N such that $M \to_\beta^* N \to_\varepsilon^* M'$.*

Proof. One shows: if $M \to_\varepsilon N \to_\beta M'$ then there is N' with $M \to_\beta N' \to_\varepsilon^* M'$. This follows easily, using Lemma 78, because ε-reduction cannot create or duplicate β-redexes. ◀

▶ **Corollary 89** (β-reduction requests ε-reduction)**.** *If $M \to_\beta^* M_1$ and $M \to_\varepsilon^* M_2$ then there are M_2', M' with $M_1 \to_\varepsilon^* M'$ and $M_2 \to_\beta^* M_2' \to_\varepsilon^* M'$.*

▶ **Lemma 90** (Confluence of β-reduction)**.** *If $M \to_\beta^* M_1$ and $M \to_\beta^* M_2$ then there exists M' such that $M_1 \to_\beta^* M'$ and $M_2 \to_\beta^* M'$.*

Proof. By a straightforward adaptation of the Tait–Martin-Löf method. The parallel reduction relation $\to_1$ is defined as follows:

- $x \to_1 x$, $s \to_1 s$, $\varepsilon \to_1 \varepsilon$,
- if $M \to_1 M'$ and $N \to_1 N'$ and $N \sim x$ then $(\lambda x : A.M)N \to_1 M'[N'/x]$,
- if $M \to_1 M'$ and $N \to_1 N'$ then $MN \to_1 M'N'$,
- if $A \to_1 A'$ and $M \to_1 M'$ then $\lambda x : A.M \to_1 \lambda x : A'.M'$,
- if $A \to_1 A'$ and $M \to_1 M'$ then $\Pi x : A.M \to_1 \Pi x : A'.M'$.

One then shows:

1. if $M \to_1 M'$ and $N \to_1 N'$ and $N \sim x$ then $M[N/x] \to_1 M'[N'/x]$,
2. if $M \to_1 M_1$ and $M \to_1 M_2$ then there exists M' with $M_1 \to_1 M'$ and $M_2 \to_1 M'$.

The first point is shown by induction on M, using Lemma 81 when $M = (\lambda y : A.M_1)M_2 \to_1 M_1'[M_2'/y] = M'$. The second point is shown by a standard argument, using the first point and Corollary 87. Confluence of β-reduction then follows from the second point, because $\to_\beta \subseteq \to_1 \subseteq \to_\beta^*$. ◀

▶ **Lemma 17** (Confluence of $\beta\varepsilon$-reduction)**.** *If $M \to_{\beta\varepsilon}^* M_1$ and $M \to_{\beta\varepsilon}^* M_2$ then there exists M' such that $M_1 \to_{\beta\varepsilon}^* M'$ and $M_2 \to_{\beta\varepsilon}^* M'$.*

Proof. This follows from the confluence of β- and ε-reduction and the fact that β-reduction requests ε-reduction. More precisely, one shows that $\to_\beta^* \cdot \to_\varepsilon^*$ has the diamond property. See Figure 3. ◀

▶ **Corollary 91.** *If $M =_{\beta\varepsilon} M'$ and $N \sim x$ and $N =_{\beta\varepsilon} N'$ then $M[N/x] =_{\beta\varepsilon} M'[N'/x]$.*

$$\begin{tikzcd}
\cdot \arrow[rr, "*", "\beta"'] \arrow[dd, "\beta"', "*"] & & \cdot \arrow[r, "*", "\varepsilon"'] \arrow[dd, dashed, "\beta"', "*"] & \cdot \arrow[d, dashed, "\beta"', "*"] \\
& & & \cdot \arrow[d, dashed, "\varepsilon"', "*"] \\
\cdot \arrow[rr, dashed, "*", "\beta"'] \arrow[d, "\varepsilon"', "*"] & & \cdot \arrow[r, dashed, "*", "\varepsilon"'] \arrow[d, dashed, "\varepsilon"', "*"] & \cdot \arrow[d, dashed, "\varepsilon"', "*"] \\
\cdot \arrow[r, dashed, "*", "\beta"'] & \cdot \arrow[r, dashed, "*", "\varepsilon"'] & \cdot \arrow[r, dashed, "*", "\varepsilon"'] & \cdot
\end{tikzcd}$$

Figure 3 Confluence of $\beta\varepsilon$-reduction.

Note that confluence of $\beta\varepsilon$-reduction on arbitrary preterms would fail if we did not restrict β-reduction as in Definition 10. For example, for $M = (\lambda x^{*^p} : A.x^{*^p})*^p$ we would have $M \to_\varepsilon^* \varepsilon$ and $M \to_\beta *^p$.

▶ **Lemma 18.** *If $M =_{\beta\varepsilon} N$ then $M \to_\varepsilon^* \varepsilon$ is equivalent to $N \to_\varepsilon^* \varepsilon$.*

Proof. Suppose $M \to_\varepsilon^* \varepsilon$. By confluence of $\beta\varepsilon$-reduction $N \to_{\beta\varepsilon}^* \varepsilon$. So $N \to_\beta^* N' \to_\varepsilon^* \varepsilon$ by Lemma 88. Now by repeatedly applying Lemma 86 we obtain $N \to_\varepsilon^* \varepsilon$. ◀

▶ **Lemma 19.** *If N does not contain ε and $M \to_{\beta\varepsilon}^* N$ then $M \to_\beta^* N$.*

Proof. By postponement of ε-reduction there is M' with $M \to_\beta^* M' \to_\varepsilon^* N$. Because N does not contain ε, we must in fact have $M' = N$. ◀

Proofs of most of the following lemmas for ordinary PTSs may be found e.g. in [2, Section 5.2]. The proofs for piPTSs are essentially the same or very similar. We only briefly indicate how to carry out the proofs and note the differences with the standard proofs.

▶ **Lemma 20** (Free variable lemma). *If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma \vdash B : C$ then:*
1. *the $x_1, \ldots, x_n$ are all distinct,*
2. $\mathrm{FV}(B), \mathrm{FV}(C) \subseteq \{x_1, \ldots, x_n\}$,
3. $\mathrm{FV}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ *for $i = 1, \ldots, n$.*

Proof. Induction on the derivation $\Gamma \vdash B : C$. ◀

▶ **Lemma 21** (Start lemma). *Let Γ be a legal context.*
1. *If $(s_1, s_2) \in \mathcal{A}$ then $\Gamma \vdash s_1 : s_2$.*
2. *If $(x : A) \in \Gamma$ then $\Gamma \vdash x : A$ and there is $s \in \mathcal{S}$ with $\Gamma_1 \vdash A : s$ and $x \in V^s$, where $\Gamma = \Gamma_1, x : A, \Gamma_2$.*

Proof. Since Γ is legal, $\Gamma \vdash B : C$ for some B, C. The lemma follows by induction on the length of the derivation of $\Gamma \vdash B : C$. ◀

▶ **Lemma 22** (Substitution lemma). *If $\Gamma, x : A, \Gamma' \vdash B : C$ and $\Gamma \vdash D : A$ and $D \sim x$ then $\Gamma, \Gamma'[D/x] \vdash B[D/x] : C[D/x]$.*

Proof. Induction on the derivation of $\Gamma, x : A, \Gamma' \vdash B : C$. We need the assumption $D \sim x$ and Corollary 91 to treat the conversion rule. For the application rule we need Lemma 81. ◀

▶ **Lemma 23** (Thinning lemma)**.** *If $\Gamma \vdash A : B$ and $\Gamma' \supseteq \Gamma$ is a legal context then $\Gamma' \vdash A : B$.*

Proof. Induction on the derivation of $\Gamma \vdash A : B$. ◀

▶ **Lemma 24** (Generation lemma)**.**

1. *If $\Gamma \vdash s : A$ then there is $s' \in \mathcal{S}$ with $A =_{\beta\varepsilon} s'$ and $(s, s') \in \mathcal{A}$.*
2. *If $\Gamma \vdash x : A$ then there are $s \in \mathcal{S}$ and B such that $A =_{\beta\varepsilon} B$ and $\Gamma \vdash B : s$ and $(x : B) \in \Gamma$ and $x \in V^s$.*
3. *If $\Gamma \vdash (\Pi x : A.B) : C$ then there is $(s_1, s_2, s_3) \in \mathcal{R}$ with $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$ and $C =_{\beta\varepsilon} s_3$.*
4. *If $\Gamma \vdash (\lambda x : A.M) : C$ then there are $s \in \mathcal{S}$ and B such that $\Gamma \vdash (\Pi x : A.B) : s$ and $\Gamma, x : A \vdash M : B$ and $C =_{\beta\varepsilon} \Pi x : A.B$.*
5. *If $\Gamma \vdash MN : C$ then there are A, B such that $\Gamma \vdash M : (\Pi x : A.B)$ and $\Gamma \vdash N : A$ and $C =_{\beta\varepsilon} B[N/x]$ and $N \sim x$.*

Proof. Completely analogous to the standard proof for ordinary PTSs, using the thinning lemma. ◀

▶ **Lemma 26** (Correctness of types lemma)**.** *If $\Gamma \vdash M : A$ then there is $s \in \mathcal{S}$ such that $A = s$ or $\Gamma \vdash A : s$.*

Proof. Induction on the derivation $\Gamma \vdash M : A$. The non-obvious case is when the application rule is used. Then $M = M_1M_2$ and $A = C[M_2/x]$ and $\Gamma \vdash M_1 : (\Pi x : B.C)$ and $\Gamma \vdash M_2 : B$ and $M_2 \sim x$. By the inductive hypothesis there is $s' \in \mathcal{S}$ such that $\Gamma \vdash (\Pi x : B.C) : s'$. By the generation lemma there is $s \in \mathcal{S}$ such that $\Gamma, x : B \vdash C : s$. Since $\Gamma \vdash M_2 : B$ and $M_2 \sim x$, by the substitution lemma we obtain $\Gamma \vdash C[M_2/x] : s$, so $\Gamma \vdash A : s$. Note that the side condition $M_2 \sim x$ in the application rule was necessary to carry out the proof. ◀

▶ **Theorem 29** (Subject reduction theorem)**.** *If $\Gamma \vdash A : B$ and $A \to_\beta^* A'$ then $\Gamma \vdash A' : B$.*

Proof. Completely analogous to the standard proof, using the generation, correctness of types and substitution lemmas, and Corollary 91. To be able to apply the substitution lemma and Corollary 91 the side condition in the application rule is necessary. ◀

Subject reduction obviously does not hold for $\beta\varepsilon$-reduction, because ε is not meant to be typable. The following lemma is a direct consequence of subject reduction.

▶ **Lemma 92.** *If $\Gamma \vdash M : A$ and $A =_{\beta\varepsilon} s$ then $\Gamma \vdash M : s$.*

Proof. By confluence of $\beta\varepsilon$-reduction and Lemma 19 we have $A \to_\beta^* s$. By the correctness of types lemma $\Gamma \vdash A : s'$ or $A = s'$. If $A = s'$ then $s = s'$ and we are done. So assume $\Gamma \vdash A : s'$. Then $\Gamma \vdash s : s'$ by the subject reduction theorem. Hence $\Gamma \vdash M : s$ by the conversion rule. ◀

The mismatch between the β-reduction in the subject reduction theorem and the $\beta\varepsilon$-conversion in the conversion rule generates some difficulties in the meta-theory of piPTSs. In ordinary functional PTSs, it is a direct consequence of the subject reduction theorem and the uniqueness of types lemma (to be stated below) that if $\Gamma \vdash B : s$ and $B =_\beta B'$ and $\Gamma \vdash A' : B'$ then $\Gamma \vdash B' : s$. This can also be easily established for functional piPTSs, by a similar proof. But we would want a stronger analogous property with $\beta\varepsilon$-conversion instead of β-conversion. Then the standard argument breaks down, because subject reduction does not hold for $\beta\varepsilon$-reduction.

In particular, we are interested in showing that in a logical piPTS if M is a Γ-proof then $M \to_{\varepsilon}^{*} \varepsilon$. This presents a difficulty already when $M = x$. Then we have $\Gamma \vdash x : A : *^{p}$ for some A. But from this we cannot immediately conclude $x \in V^{*^{p}}$ because the derivation of $\Gamma \vdash x : A$ may end with the conversion rule. From the generation lemma we may only conclude that there are $s \in \mathcal{S}$ and $B =_{\beta\varepsilon} A$ such that $\Gamma \vdash B : s$ and $(x : B) \in \Gamma$ and $x \in V^{s}$. It would suffice if from $\Gamma \vdash B : s$ and $\Gamma \vdash A : *^{p}$ and $B =_{\beta\varepsilon} A$ we could conclude $s = *^{p}$. But this does not seem completely straightforward to establish without subject reduction for $\beta\varepsilon$-reduction. We will ultimately show this property using a slight sharpening of the uniqueness of types lemma for logical piPTSs.

Our next aim is to show that in a logical piPTS a Γ-type does not ε-reduce to ε. For this we need the following technical definition.

▶ **Definition 93.** We define the relation $B \leadsto_{\Gamma} C$ inductively:

- if $B =_{\beta\varepsilon} C$ then $B \leadsto_{\Gamma} C$,
- if there exist N and C' such that $\Gamma \vdash N : A$ and $N \sim x$ and $B[N/x] =_{\beta\varepsilon} C' \leadsto_{\Gamma} C$ then $\Pi x : A.B \leadsto_{\Gamma} C$.

▶ **Lemma 94.** *If $\Gamma \vdash B : s$ and $B =_{\beta\varepsilon} B' \leadsto_{\Gamma} C$ then $B \leadsto_{\Gamma} C$.*

Proof. If $B' =_{\beta\varepsilon} C$ then this is obvious. Otherwise $B' = \Pi x : A_0.B_0$ and $\Gamma \vdash N : A_0$ and $N \sim x$ and $B_0[N/x] =_{\beta\varepsilon} C' \leadsto_{\Gamma} C$. By the confluence of $\beta\varepsilon$-reduction we have $B = \Pi x : A_1.B_1$ with $A_1 =_{\beta\varepsilon} A_0$ and $B_1 =_{\beta\varepsilon} B_0$. Since $N \sim x$, by Corollary 91 we obtain $B_1[N/x] =_{\beta\varepsilon} B_0[N/x] =_{\beta\varepsilon} C'$. Because $\Gamma \vdash B : s$, by the generation lemma $\Gamma \vdash A_1 : s'$ for some $s' \in \mathcal{S}$. Hence $\Gamma \vdash N : A_1$ by the conversion rule. Thus $B \leadsto_{\Gamma} C$. ◀

▶ **Lemma 95.** *In a logical piPTS, if $\Gamma \vdash B : *^{p}$ and $B =_{\beta\varepsilon} B' \leadsto_{\Gamma} C$, then there exists C' such that $\Gamma \vdash C' : *^{p}$ and $C' =_{\beta\varepsilon} C$.*

Proof. Induction on the definition of $B' \leadsto_{\Gamma} C$. If $B' =_{\beta\varepsilon} C$ then this is obvious. Otherwise $B' = \Pi x : A_1.B_1$ and $\Gamma \vdash N : A_1$ and $N \sim x$ and $B_1[N/x] =_{\beta\varepsilon} C_1 \leadsto_{\Gamma} C$. By the confluence of $\beta\varepsilon$-reduction $B = \Pi x : A_0.B_0$ with $A_0 =_{\beta\varepsilon} A_1$ and $B_0 =_{\beta\varepsilon} B_1$. Because $\Gamma \vdash (\Pi x : A_0.B_0) : *^{p}$ and the piPTS is logical, by the generation lemma $\Gamma \vdash A_0 : s$ for some $s \in \mathcal{S}$ and $\Gamma, x : A_0 \vdash B_0 : *^{p}$. Since $\Gamma \vdash N : A_1$, by the conversion rule $\Gamma \vdash N : A_0$. Hence $\Gamma \vdash B_0[N/x] : *^{p}$ by the substitution lemma. By Corollary 91 we also have $B_0[N/x] =_{\beta\varepsilon} B_1[N/x]$. Thus $\Gamma \vdash B_0[N/x] : *^{p}$ and $B_0[N/x] =_{\beta\varepsilon} C_1$ and $C_1 \leadsto_{\Gamma} C$. We may therefore apply the inductive hypothesis to obtain C' with $\Gamma \vdash C' : *^{p}$ and $C' =_{\beta\varepsilon} C$. ◀

▶ **Lemma 96.** *In a logical piPTS, if $\Gamma \vdash M : s$ then $M \not\to_{\varepsilon}^{*} \varepsilon$.*

Proof. By induction on M we show that if $(\star)$ below holds for M then $M \not\to_{\varepsilon}^{*} \varepsilon$. Then taking $n = 0$ in $(\star)$ gives us the lemma.

$(\star)$ There exist $A_1, \ldots, A_n$ and $N_1, \ldots, N_n$ such that

$$\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash M : B$$

and $N_i \sim x_i$ and $\Gamma \vdash N_i : A_i[N_1/x_1] \ldots [N_{i-1}/x_{i-1}]$ for $i = 1, \ldots, n$ and

$$B[N_1/x_1] \ldots [N_n/x_n] \leadsto_{\Gamma} s.$$

Assume $(\star)$ and $M \to_{\varepsilon}^{*} \varepsilon$. Let $\Gamma' = \Gamma, x_1 : A_1, \ldots, x_n : A_n$. There are three possibilities.

1. $M = x \in V^{*^p}$. Then by the generation lemma there is B' with $B' =_{\beta\varepsilon} B$ and $\Gamma' \vdash B' : *^p$. Using the substitution lemma repeatedly we obtain $\Gamma \vdash B'[N_1/x_1] \ldots [N_n/x_n] : *^p$. Using Corollary 91 repeatedly we obtain

$$B'[N_1/x_1] \ldots [N_n/x_n] =_{\beta\varepsilon} B[N_1/x_1] \ldots [N_n/x_n] \rightsquigarrow_\Gamma s.$$

Hence by Lemma 95 there is C with $\Gamma \vdash C : *^p$ and $C =_{\beta\varepsilon} s$. By Lemma 19 and the confluence of $\beta\varepsilon$-reduction we have $C \to_\beta^* s$. Hence by the subject reduction theorem we obtain $\Gamma \vdash s : *^p$. This contradicts the fact that the piPTS is logical.
2. $M = M_1 M_2$ with $M_1 \to_\varepsilon^* \varepsilon$. By the generation lemma there exist A_0 and B_0 such that $\Gamma' \vdash M_1 : (\Pi x : A_0.B_0)$ and $\Gamma' \vdash M_2 : A_0$ and $B =_{\beta\varepsilon} B_0[M_2/x]$ and $M_2 \sim x$. Let $M_2' = M_2[N_1/x_1] \ldots [N_n/x_n]$. Using the substitution lemma repeatedly we obtain

$$\Gamma \vdash M_2' : A_0[N_1/x_1] \ldots [N_n/x_n].$$

Also $M_2' \sim x$ by repeated use of Lemma 81. By the correctness of types and generation lemmas there is s' with $\Gamma', x : A_0 \vdash B_0 : s'$. Using the substitution lemma repeatedly with the N_i's and M_2' we obtain

$$\Gamma \vdash B_0[N_1/x_1] \ldots [N_n/x_n][M_2'/x] : s'.$$

Using Corollary 91 repeatedly we also obtain

$$B_0[M_2/x][N_1/x_1] \ldots [N_n/x_n] =_{\beta\varepsilon} B[N_1/x_1] \ldots [N_n/x_n] \rightsquigarrow_\Gamma s.$$

By α-conversion we may assume $x \notin \mathrm{FV}(N_1, \ldots, N_n)$. Thus

$$B_0[M_2/x][N_1/x_1] \ldots [N_n/x_n] = B_0[N_1/x_1] \ldots [N_n/x_n][M_2'/x].$$

Hence

$$(\Pi x : A_0.B_0)[N_1/x_1] \ldots [N_n/x_n] \rightsquigarrow_\Gamma s.$$

Now applying the inductive hypothesis yields a contradiction.
3. $M = \lambda x : A_0.M'$ with $M' \to_\varepsilon^* \varepsilon$. By the generation lemma there are $s' \in \mathcal{S}$ and B_0 such that $\Gamma' \vdash (\Pi x : A_0.B_0) : s$ and $\Gamma', x : A_0 \vdash M' : B_0$ and $B =_{\beta\varepsilon} \Pi x : A_0.B_0$. By the confluence of $\beta\varepsilon$-reduction we have $B = \Pi x : C_0.D_0$ with $A_0 =_{\beta\varepsilon} C_0$ and $B_0 =_{\beta\varepsilon} D_0$. Let $A_0^{*^p} = A_0[N_1/x_1] \ldots [N_n/x_n]$ and analogously for $B_0^{*^p}$, $C_0^{*^p}$ and $D_0^{*^p}$. Since $B[N_1/x_1] \ldots [N_n/x_n] = \Pi x : C_0^{*^p}.D_0^{*^p} \rightsquigarrow_\Gamma s$, there is N with $\Gamma \vdash N : C_0^{*^p}$ and $N \sim x$ and $D_0^{*^p}[N/x] =_{\beta\varepsilon} C \rightsquigarrow_\Gamma s$ (the case $\Pi x : C_0^{*^p}.D_0^{*^p} =_{\beta\varepsilon} s$ is impossible by the confluence of $\beta\varepsilon$-reduction). By repeated use of Corollary 91 we have $A_0^{*^p} =_{\beta\varepsilon} C_0^{*^p}$ and $B_0^{*^p}[N/x] =_{\beta\varepsilon} D_0^{*^p}[N/x]$. Since $\Gamma' \vdash (\Pi x : A_0.B_0) : s'$, by the generation lemma there are $s_1, s_2 \in \mathcal{S}$ with $\Gamma' \vdash A_0 : s_1$ and $\Gamma', x : A_0 \vdash B_0 : s_2$. By repeated use of the substitution lemma $\Gamma \vdash A_0^{*^p} : s_1$. Since also $C_0^{*^p} =_{\beta\varepsilon} A_0^{*^p}$ and $\Gamma \vdash N : C_0^{*^p}$, by the conversion rule we have $\Gamma \vdash N : A_0^{*^p}$. Now by repeated use of the substitution lemma we obtain $\Gamma \vdash B_0^{*^p}[N/x] : s_2$. Since also $B_0^{*^p}[N/x] =_{\beta\varepsilon} C \rightsquigarrow s$, by Lemma 94 we obtain $B_0^{*^p}[N/x] \rightsquigarrow_\Gamma s$. Therefore, because $\Gamma', x : A_0 \vdash M' : B_0$ and $N \sim x$ and $\Gamma \vdash N : A_0[N_1/x_1] \ldots [N_n/x_n]$ and $B_0[N_1/x_1] \ldots [N_n/x_n][N/x] \rightsquigarrow_\Gamma s$, we may apply the inductive hypothesis to conclude $M' \not\to_\varepsilon^* \varepsilon$. This gives a contradiction. ◀

A simpler proof of Lemma 96 would be possible if we changed the definitions in one of the following two ways.

(1) In the definition of a logical piPTS, require $(s, *^p, *^p) \in \mathcal{R}$ for any $s \in \mathcal{S}$.
(2) In the definition of a piPTS, add the side condition $x \in V^{s_1}$ in the product rule, and restrict ε-reduction of lambda-abstractions to:

$$\lambda x^s : A.\varepsilon \quad \to_\varepsilon \quad \varepsilon \qquad \text{if } (s, *^p, *^p) \in \mathcal{R}$$

Then we could prove ($\star$) below by a relatively straightforward induction, without relying on Lemma 99. Lemma 96 would then easily follow from ($\star$).
($\star$) In a logical piPTS, if $\Gamma \vdash M : C$ and $M \to_\varepsilon^* \varepsilon$ then there is C' with $C' =_{\beta\varepsilon} C$ and $\Gamma \vdash M : C' : *^p$.
However, none of the changes (1) or (2) seem to allow avoiding the use of Lemma 99 in the proof of Lemma 100.

▶ **Definition 97.** An *n-ary term context* $C[\Box_1, \ldots, \Box_n]$ is a term with n holes into which some terms $N_1, \ldots, N_n$ may be substituted possibly capturing their free variables, yielding $C[N_1, \ldots, N_n]$. For example, $C[\Box_1, \Box_2] = \lambda xy.\Box_1\Box_2$ is a term context, and $C[x, xy] = \lambda xy.x(xy)$.

We write $\Gamma_1 =_\varepsilon \Gamma_2$ if $\Gamma_1 = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma_2 = x_1 : A_1', \ldots, x_n : A_n'$ and $A_i =_\varepsilon A_i'$. The following simple lemma will be used implicitly.

▶ **Lemma 98.** *If $M =_\varepsilon M'$ then there are $x_1, \ldots, x_n$ and an n-ary term context $C[\Box_1, \ldots, \Box_n]$ such that $M = C[N_1, \ldots, N_n]$ and $M' = C[N_1', \ldots, N_n']$ and $N_i \to_\varepsilon^* \varepsilon$ and $N_i' \to_\varepsilon^* \varepsilon$.*

Proof. Follows from confluence of ε-reduction. ◀

For logical piPTSs we need a somewhat sharpened version of the uniqueness of types lemma.

▶ **Lemma 27** (Uniqueness of types lemma).
1. *In a functional piPTS, if $\Gamma \vdash A : B$ and $\Gamma \vdash A : B'$ then $B =_{\beta\varepsilon} B'$.*
2. *In a logical piPTS, if $\Gamma \vdash M_1 : A_1$ and $\Gamma \vdash M_2 : A_2$ and $M_1 =_{\beta\varepsilon} M_2$ and $M_1 \not\to_\varepsilon^* \varepsilon$ and $M_2 \not\to_\varepsilon^* \varepsilon$ then $A_1 =_{\beta\varepsilon} A_2$.*

Proof. We show the second point. The proof of the first point is similar but simpler, and it is also completely analogous to the standard uniqueness of types proof for ordinary functional PTSs.

So assume the piPTS is logical. First, we show the following condition ($\star$).
($\star$) If $\Gamma_1 \vdash M_1 : A_1$ and $\Gamma_2 \vdash M_2 : A_2$ and $M_1 =_\varepsilon M_2$ and $\Gamma_1 =_\varepsilon \Gamma_2$ and $M_1 \not\to_\varepsilon^* \varepsilon$ and $M_2 \not\to_\varepsilon^* \varepsilon$ then $A_1 =_{\beta\varepsilon} A_2$.
We proceed by induction on M_1. We have the following possibilities.

- $M_1 = s = M_2$. By the generation lemma there are $s_1, s_2 \in \mathcal{S}$ such that $A_1 =_{\beta\varepsilon} s_1$ and $A_2 =_{\beta\varepsilon} s_2$ and $(s, s_1), (s, s_2) \in \mathcal{A}$. Hence $s_1 = s_2$ because the piPTS is functional. Thus $A_1 =_{\beta\varepsilon} A_2$.
- $M_1 = x = M_2$. By the generation lemma there exist C_1, C_2 such that $A_1 =_{\beta\varepsilon} C_1$ and $A_2 =_{\beta\varepsilon} C_2$ and $(x : C_1) \in \Gamma_1$ and $(x : C_2) \in \Gamma_2$. Since $\Gamma_1 =_\varepsilon \Gamma_2$, we have $C_1 =_\varepsilon C_2$. Thus $A_1 =_{\beta\varepsilon} A_2$.
- $M_1 = \Pi x : B_1.C_1$ and $M_2 = \Pi x : B_2.C_2$ with $B_1 =_\varepsilon B_2$ and $C_1 =_\varepsilon C_2$. By the generation lemma there exist $(s_1, s_2, s_3), (s_1', s_2', s_3') \in \mathcal{R}$ such that $\Gamma_1 \vdash B_1 : s_1$ and $\Gamma_1, x : B_1 \vdash C_1 : s_2$ and $\Gamma_2 \vdash B_2 : s_1'$ and $\Gamma_2, x : B_2 \vdash C_2 : s_2'$ and $A_1 =_{\beta\varepsilon} s_3$ and $A_2 =_{\beta\varepsilon} s_3'$. Note that $B_1 \not\to_\varepsilon^* \varepsilon$ and $B_2 \not\to_\varepsilon^* \varepsilon$ and $C_1 \not\to_\varepsilon^* \varepsilon$ and $C_2 \not\to_\varepsilon^* \varepsilon$, by Lemma 96. Hence, by the inductive hypothesis and the confluence of $\beta\varepsilon$-reduction $s_1 = s_1'$ and $s_2 = s_2'$. Thus $s_3 = s_3'$ because the piPTS is functional. Hence $A_1 =_{\beta\varepsilon} A_2$.

- $M_1 = \lambda x : B_1.N_1$ and $M_2 = \lambda x : B_2.N_2$ and $B_1 =_\varepsilon B_2$ and $N_1 =_\varepsilon N_2$. By the generation lemma there exist $s_1, s_2 \in \mathcal{S}$ and C_1, C_2 such that $\Gamma_i \vdash B_i : s_i$ and $\Gamma_i, x : B_i \vdash N_i : C_i$ and $A_i =_{\beta\varepsilon} \Pi x : B_i.C_i$. Note that $N_i \not\to_\varepsilon^* \varepsilon$ because $M_i \not\to_\varepsilon^* \varepsilon$. Hence, by the inductive hypothesis $C_1 =_{\beta\varepsilon} C_2$. Hence $A_1 =_{\beta\varepsilon} A_2$, because also $B_1 =_\varepsilon B_2$.
- $M_1 = N_1 N_1'$ and $M_2 = N_2 N_2'$ and $N_1 =_\varepsilon N_2$ and $N_1' =_\varepsilon N_2'$. By the generation lemma there exist $x_1, x_2, B_1, B_2, C_1, C_2$ such that $\Gamma_i \vdash N_i : (\Pi x_i : B_i.C_i)$ and $\Gamma_i \vdash N_i' : B_i$ and $N_i' \sim x_i$ and $A_i =_{\beta\varepsilon} C_i[N_i'/x_i]$. Note that $N_i \not\to_\varepsilon^* \varepsilon$ because $M_i \not\to_\varepsilon^* \varepsilon$. Hence, by the inductive hypothesis $\Pi x_1 : B_1.C_1 =_{\beta\varepsilon} \Pi x_2 : B_2.C_2$. Thus $x_1 = x_2$ and $C_1 =_{\beta\varepsilon} C_2$ by confluence of $\beta\varepsilon$-reduction. Hence $C_1[N_1'/x_1] =_{\beta\varepsilon} C_2[N_2'/x_2]$ by Corollary 91. Therefore $A_1 =_{\beta\varepsilon} A_2$.

We have thus shown ($\star$). Now assume $\Gamma \vdash M_i : A_i$ and $M_1 =_{\beta\varepsilon} M_2$ and $M_i \not\to_\varepsilon^* \varepsilon$. By confluence of $\beta\varepsilon$-reduction and by Lemma 88 there are N_1, N_2 with $M_i \to_\beta^* N_i$ and $N_1 =_\varepsilon N_2$. By the subject reduction theorem $\Gamma \vdash N_i : A_i$. Because $M_i \to_\beta^* N_i$ and $M_i \not\to_\varepsilon^* \varepsilon$, Lemma 86 implies that $N_i \not\to_\varepsilon^* \varepsilon$. Hence by ($\star$) we obtain $A_1 =_{\beta\varepsilon} A_2$. ◀

▶ **Lemma 99.** *In a logical piPTS, if* $\Gamma \vdash B : s$ *and* $B =_{\beta\varepsilon} B'$ *and* $\Gamma \vdash A' : B'$ *then* $\Gamma \vdash B' : s$.

Proof. By the correctness of types lemma there are two cases.

- $B' = s'$. Then $B \to_\beta^* s'$ by confluence of $\beta\varepsilon$-reduction and Lemma 19. Hence $\Gamma \vdash s' : s$ by the subject reduction theorem, i.e., $\Gamma \vdash B' : s$.
- $\Gamma \vdash B' : s'$. Note that $B \not\to_\varepsilon^* \varepsilon$ and $B' \not\to_\varepsilon^* \varepsilon$ by Lemma 96. Hence, by the second point of the uniqueness of types lemma and by confluence of $\beta\varepsilon$-reduction $s = s'$. Therefore $\Gamma \vdash B' : s$. ◀

▶ **Lemma 100.** *In a logical piPTS, if* $\Gamma \vdash M : C : *^p$ *then* $M \to_\varepsilon^* \varepsilon$.

Proof. Induction on M. We have the following cases.

- $M = s$. By the generation lemma there is $s' \in \mathcal{S}$ such that $C =_{\beta\varepsilon} s'$. By confluence of $\beta\varepsilon$-reduction and Lemma 19 we have $C \to_\beta^* s'$. By the subject reduction theorem $\Gamma \vdash s' : *^p$. This is a contradiction, because the piPTS is logical.
- $M = x$. By the generation lemma there are $s \in \mathcal{S}$ and B such that $B =_{\beta\varepsilon} C$ and $\Gamma \vdash B : s$ and $(x : B) \in \Gamma$ and $x \in V^s$. By Lemma 99 we obtain $\Gamma \vdash C : s$, and thus $s = *^p$ by the uniqueness of types lemma. So $x \in V^{*^p}$. Hence $M = x \to_\varepsilon \varepsilon$.
- $M = \Pi x : A.B$. By the generation lemma there is $s' \in \mathcal{S}$ with $C =_{\beta\varepsilon} s'$. Like in the case $M = s$, using confluence of $\beta\varepsilon$-reduction, Lemma 19 and the subject reduction theorem, we derive a contradiction.
- $M = \lambda x : A.N$. By the generation lemma there are $s \in \mathcal{S}$ and B such that $\Gamma \vdash (\Pi x : A.B) : s$ and $\Gamma, x : A \vdash N : B$ and $C =_{\beta\varepsilon} \Pi x : A.B$. By Lemma 99 we have $\Gamma \vdash C : s$, and thus $s = *^p$ by the uniqueness of types lemma. Since $\Gamma \vdash (\Pi x : A.B) : *^p$, by the generation lemma there is $(s_1, s_2, *^p) \in \mathcal{R}$ such that $\Gamma, x : A \vdash B : s_2$. Because the piPTS is logical $s_2 = *^p$. Hence $\Gamma, x : A \vdash N : B : *^p$. By the inductive hypothesis $N \to_\varepsilon^* \varepsilon$. Hence $M = \lambda x : A.N \to_\varepsilon^* \lambda x : A.\varepsilon \to_\varepsilon \varepsilon$.
- $M = M_1 M_2$. By the generation lemma there are A, B such that $\Gamma \vdash M_1 : (\Pi x : A.B)$ and $\Gamma \vdash M_2 : A$ and $C =_{\beta\varepsilon} B[M_2/x]$ and $M_2 \sim x$. By the correctness of types lemma and the generation lemma there is $(s_1, s_2, s_3) \in \mathcal{R}$ such that $\Gamma \vdash (\Pi x : A.B) : s_3$ and $\Gamma, x : A \vdash B : s_2$. Since $\Gamma \vdash M_2 : A$ and $M_2 \sim x$, by the substitution lemma $\Gamma \vdash B[M_2/x] : s_2$. By Lemma 99 we have $\Gamma \vdash C : s_2$, and thus $s_2 = *^p$ by the uniqueness of types lemma. Hence $s_3 = *^p$ because the piPTS is logical. Thus $\Gamma \vdash M_1 : (\Pi x : A.B) : *^p$, and by the inductive hypothesis we conclude $M_1 \to_\varepsilon^* \varepsilon$. Therefore $M = M_1 M_2 \to^* \varepsilon M_2 \to_\varepsilon \varepsilon$. ◀

▶ **Lemma 101.** *In a logical piPTS, if* $\Gamma \vdash M : C$ *and* $M \to_\varepsilon^* \varepsilon$ *then* $\Gamma \vdash C : *^p$.

Proof. Induction on M. There are three possibilities.

- $M = x \in V^{*^p}$. By the generation lemma there exists B such that $B =_{\beta\varepsilon} C$ and $\Gamma \vdash B : *^p$. By Lemma 99 we have $\Gamma \vdash C : *^p$.
- $M = \lambda x : A.M'$ with $M' \to_\varepsilon^* \varepsilon$. By the generation lemma there exist $s \in \mathcal{S}$ and B such that $\Gamma \vdash (\Pi x : A.B) : s$ and $\Gamma, x : A \vdash M' : B$ and $C =_{\beta\varepsilon} \Pi x : A.B$. By the generation lemma there is $(s_1, s_2, s) \in \mathcal{R}$ with $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$. Since $\Gamma, x : A \vdash M' : B$ and $M' \to_\varepsilon^* \varepsilon$, by the inductive hypothesis and the uniqueness of types lemma we obtain $s_2 = *^p$. Because the piPTS is logical also $s = *^p$. Then $\Gamma \vdash C : *^p$ by Lemma 99.
- $M = M_1 M_2$ with $M_1 \to_\varepsilon^* \varepsilon$. By the generation lemma there are A, B such that $\Gamma \vdash M_1 : (\Pi x : A.B)$ and $\Gamma \vdash M_2 : A$ and $C =_{\beta\varepsilon} B[M_2/x]$ and $M_2 \sim x$. By the inductive hypothesis $\Gamma \vdash M_1 : (\Pi x : A.B) : *^p$. By the generation lemma there is $(s_1, s_2, *^p) \in \mathcal{R}$ such that $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$. Because the piPTS is logical, $s_2 = *^p$. By the substitution lemma we thus obtain $\Gamma \vdash B[M_2/x] : *^p$. By Lemma 99 we have $\Gamma \vdash C : *^p$. ◀

▶ **Theorem 30.** *Assume the piPTS is logical and M is a Γ-term. Then M is a Γ-proof if and only if $M \to_\varepsilon^* \varepsilon$.*

Proof. Follows from the correctness of types lemma, Lemma 96, Lemma 100 and Lemma 101. ◀

▶ **Lemma 31.** *In a logical piPTS, if M is a Γ-term and $M =_{\beta\varepsilon} N$ and $\Gamma \vdash N : s$ then $\Gamma \vdash M : s$.*

Proof. By the correctness of types lemma either $\Gamma \vdash M : s'$ or $M = s'$ for some $s' \in \mathcal{S}$. If $\Gamma \vdash M : s'$ then $M \not\to_\varepsilon^* \varepsilon$ and $N \not\to_\varepsilon^* \varepsilon$ by Lemma 96, so $s' = s$ by the uniqueness of types lemma and confluence of $\beta\varepsilon$-reduction. If $M = s'$ then $N \to_\beta^* M = s'$ by confluence of $\beta\varepsilon$-reduction and Lemma 19. Hence $\Gamma \vdash M : s$ by the subject reduction theorem. ◀

▶ **Lemma 102.** *In a logical piPTS, if M is a Γ-proof and $\Gamma \vdash M : A$ then $\Gamma \vdash A : *^p$.*

Proof. Since M is a Γ-proof, $M \to_\varepsilon^* \varepsilon$ by Theorem 30. Hence $\Gamma \vdash A : *^p$ by Lemma 101. ◀

▶ **Lemma 32.** *In a logical piPTS, if $\Gamma \vdash M : A$ and $\Gamma, x : A$ is a legal context then $M \sim x$.*

Proof. Since $\Gamma, x : A$ is a legal context, by the start lemma there is $s \in \mathcal{S}$ with $x \in V^s$ and $\Gamma \vdash A : s$. First, assume $s = *^p$. Since then $\Gamma \vdash M : A : *^p$, the term M is a Γ-proof, and thus $M \to_\varepsilon^* \varepsilon$ by Theorem 30. So $M \sim x \in V^{*^p}$. If $s \neq *^p$ then M is not a Γ-proof, by Lemma 102 and the uniqueness of types lemma. Hence, then also $M \sim x \in V^s$. ◀

▶ **Lemma 34.** *In a logical piPTS, $\Gamma \vdash^- M : N$ is equivalent to $\Gamma \vdash M : N$.*

Proof. The implication from right to left follows by induction on the length of the derivation of $\Gamma \vdash M : N$. For the other direction we proceed by induction on the length of the derivation of $\Gamma \vdash^- M : N$. Lemma 32 is needed to handle the application rule. ◀

B Proofs for Section 5

▶ **Lemma 62.**

1. *If $M \succ_\Gamma^\mathcal{F} \varphi$ and $\Gamma' \supseteq \Gamma$ is a legal context then $M \succ_{\Gamma'}^\mathcal{F} \varphi$.*
2. *If $M \succ_\Gamma^\mathcal{C} t$ and $\Gamma' \supseteq \Gamma$ is a legal context then $M \succ_{\Gamma'}^\mathcal{C} t$.*

Proof. Induction on the definition of $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $M \succ^{\mathcal{C}}_{\Gamma} t$. We show a few cases. The other cases are similar, trivial, or follow directly from the inductive hypothesis.

- $M = \Pi x : A.B \succ^{\mathcal{F}}_{\Gamma} \varphi_1 \to \varphi_2 = \varphi$. Then $\Gamma \vdash A : *^p$ and $A \succ^{\mathcal{F}}_{\Gamma} \varphi_1$ and $B \succ^{\mathcal{F}}_{\Gamma,x:A} \varphi_2$. By Corollary 28 we have $x \in V^{*^p}$. By the variable convention we may assume $x \notin \mathrm{dom}(\Gamma')$. By the thinning lemma $\Gamma' \vdash A : *^p$. Hence $\Gamma', x : A \supseteq \Gamma, x : A$ is a legal context. So $B \succ^{\mathcal{F}}_{\Gamma',x:A} \varphi_2$ by the inductive hypothesis. Also $A \succ^{\mathcal{F}}_{\Gamma'} \varphi_1$ by the inductive hypothesis. Thus $\Pi x : A.B \succ^{\mathcal{F}}_{\Gamma'} \varphi_1 \to \varphi_2$.
- $M = \Pi x : A.B \succ^{\mathcal{F}}_{\Gamma} \forall x.T(x,t) \to \psi = \varphi$. Then $\Gamma \nvdash A : *^p$ and $A \succ^{\mathcal{C}}_{\Gamma} t$ and $B \succ^{\mathcal{F}}_{\Gamma,x:A} \psi$. By Corollary 28 we have $x \in V^s$. By the variable convention we may assume $x \notin \mathrm{dom}(\Gamma')$. Since $\Pi x : A.B$ is a Γ-subject, by the correctness of types and the generation lemmas $\Gamma \vdash A : s$ for some $s \in \mathcal{S}$. So $\Gamma' \vdash A : s$ by the thinning lemma. Hence $\Gamma', x : A \supseteq \Gamma, x : A$ is a legal context. So $B \succ^{\mathcal{F}}_{\Gamma',x:A} \psi$ by the inductive hypothesis. Also $A \succ^{\mathcal{C}}_{\Gamma'} t$ by the inductive hypothesis. We also have $\Gamma' \nvdash A : *^p$, because otherwise $s = *^p$ by the uniqueness of types lemma. Thus $\Pi x : A.B \succ^{\mathcal{F}}_{\Gamma'} \forall x.T(x,t) \to \psi$.
- $M \succ^{\mathcal{C}}_{\Gamma} \varepsilon$ and M is a Γ-proof. Then M is a Γ'-proof by the thinning lemma, so $M \succ^{\mathcal{C}}_{\Gamma'} \varepsilon$.
- $M = M_1 M_2 \succ^{\mathcal{C}}_{\Gamma} t_1 t_2 = t$ and $M_1 \succ^{\mathcal{C}}_{\Gamma} t_1$ and $M_2 \succ^{\mathcal{C}}_{\Gamma} t_2$. We have $M_i \succ^{\mathcal{C}}_{\Gamma'} t_i$ by the inductive hypothesis. Note that M is not a Γ'-proof, because otherwise $M \to^*_{\varepsilon} \varepsilon$ by Theorem 30 and thus M would also be a Γ-proof. Hence $M_1 M_2 \succ^{\mathcal{C}}_{\Gamma'} t_1 t_2$.
- $M = (\lambda x : A.M')[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{t}/\vec{x}] = t$ and $\Gamma_0 \vdash (\lambda x : A.M') : B$ and $\Gamma_0 \nvdash A : *^p$ and $f = \Lambda_1(x,r,t)$ and $\vec{y} = \mathrm{FV}(r,t) \setminus \{x\}$ and $\Gamma_0 \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$ and $A[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} r[\vec{t}/\vec{x}]$ and $M'[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:A[\vec{N}/\vec{x}]} t[\vec{t}/\vec{x}]$. Then also $\Gamma_0 \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma'$ by the definition of $\rightsquigarrow$. By the variable convention we may assume $x \notin \mathrm{dom}(\Gamma')$, so $\Gamma', x : A[\vec{N}/\vec{x}] \supseteq \Gamma, x : A[\vec{N}/\vec{x}]$ is a legal context. Thus $A[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma'} r[\vec{t}/\vec{x}]$ and $M'[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma',x:A[\vec{N}/\vec{x}]} t[\vec{t}/\vec{x}]$ by the inductive hypothesis. Additionally, like in the case $M = M_1 M_2$, using Theorem 30 we conclude that M is not a Γ'-proof. Hence $M = (\lambda x : A.M')[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma'} (f\vec{y})[\vec{t}/\vec{x}] = t$. ◀

▶ **Lemma 64.** *Assume $N \sim x$. Then $M \to^*_{\varepsilon} \varepsilon$ iff $M[N/x] \to^*_{\varepsilon} \varepsilon$.*

Proof. By Lemma 16 we have $\mathrm{nf}_{\varepsilon}(M[N/x]) = \mathrm{nf}_{\varepsilon}(M)[\mathrm{nf}_{\varepsilon}(N)/x]$. Thus if $\mathrm{nf}_{\varepsilon}(M) = \varepsilon$ then also $\mathrm{nf}_{\varepsilon}(M[N/x]) = \varepsilon$. Conversely, if $\mathrm{nf}_{\varepsilon}(M[N/x]) = \varepsilon$ and $\mathrm{nf}_{\varepsilon}(M) \neq \varepsilon$ then $\mathrm{nf}_{\varepsilon}(M) = x$ and $\mathrm{nf}_{\varepsilon}(N) = \varepsilon$. Then also $x \in V^{*^p}$, because $N \sim x$. This is impossible because then $x \to_{\varepsilon} \varepsilon$. ◀

▶ **Lemma 65.** *Assume $\Gamma_1 \vdash N : A$ and $N \succ^{\mathcal{C}}_{\Gamma_1} t$ and $N \sim y$.*

1. *If $M \succ^{\mathcal{F}}_{\Gamma_1,y:A,\Gamma_2} \varphi$ then $M[N/y] \succ^{\mathcal{F}}_{\Gamma_1,\Gamma_2[N/y]} \varphi[t/y]$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma_1,y:A,\Gamma_2} u$ then $M[N/y] \succ^{\mathcal{C}}_{\Gamma_1,\Gamma_2[N/y]} u[t/y]$.*

Proof. By induction on the definition of $M \succ^{\mathcal{F}}_{\Gamma_1,y:A,\Gamma_2} \varphi$ and $M \succ^{\mathcal{C}}_{\Gamma_1,y:A,\Gamma_2} u$. Again, we show a few cases.

Let $\Gamma = \Gamma_1, y : A, \Gamma_2$ and $\Gamma' = \Gamma_1, \Gamma_2[N/y]$. Note that because we implicitly assume M is a Γ-subject (Γ-proposition), by the substitution lemma $M[N/y]$ is also a Γ'-subject (Γ'-proposition). Also note that M is a Γ-proof iff $M[N/y]$ is a Γ'-proof. Indeed, this follows from Lemma 64 and Theorem 30.

- $M = \Pi x : C.B \succ^{\mathcal{F}}_{\Gamma} \varphi_1 \to \varphi_2 = \varphi$. Then $\Gamma \vdash C : *^p$ and $C \succ^{\mathcal{F}}_{\Gamma} \varphi_1$ and $B \succ^{\mathcal{F}}_{\Gamma,x:C} \varphi_2$. By the substitution lemma $\Gamma' \vdash C[N/y] : *^p$. By the inductive hypothesis $C[N/y] \succ^{\mathcal{F}}_{\Gamma'} \varphi_1[t/y]$ and $B[N/y] \succ^{\mathcal{F}}_{\Gamma',x:C[N/y]} \varphi_2[t/y]$. Hence $(\Pi x : C.B)[N/y] = \Pi x : C[N/y].B[N/y] \succ^{\mathcal{F}}_{\Gamma'} \varphi_1[t/y] \to \varphi_2[t/y] = \varphi[t/y]$.
- M is a Γ-proof and $M \succ^{\mathcal{C}}_{\Gamma} \varepsilon$. Then by the substitution lemma $M[N/y]$ is also a Γ'-proof. Hence $M[N/y] \succ^{\mathcal{C}}_{\Gamma'} \varepsilon$.

- $M = y \succ^{\mathcal{C}}_{\Gamma} y = u$. By the substitution lemma y is a Γ'-subject, so Γ' is a legal context. We also have $N \succ^{\mathcal{C}}_{\Gamma_1} t$ and $\Gamma_1 \subseteq \Gamma'$. Hence $M[N/y] = N \succ^{\mathcal{C}}_{\Gamma'} t = u[t/y]$ by Lemma 62.
- $M = M_1 M_2 \succ^{\mathcal{C}}_{\Gamma} u_1 u_2$ and $M_1 \succ^{\mathcal{C}}_{\Gamma} u_1$ and $M_2 \succ^{\mathcal{C}}_{\Gamma} u_2$. By the inductive hypothesis $M_i[N/y] \succ^{\mathcal{C}}_{\Gamma'} u_i[t/y]$. Recall that $M[N/y]$ is not a Γ'-proof, by the discussion in the second paragraph of the proof of this lemma. Therefore $M[N/y] = M_1[N/y]M_2[N/y] \succ^{\mathcal{C}}_{\Gamma'} u_1[t/y]u_2[t/y] = u[t/y]$.
- $M = (\lambda x : A.M')[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{t}/\vec{x}] = u$ and $\Gamma_0 \vdash (\lambda x : A.M') : B$ and $\Gamma_0 \nvdash A : *^p$ and $f = \Lambda_1(x, r_1, r_2)$ and $\vec{y} = \mathrm{FV}(r_1, r_2) \setminus \{x\}$ and $\Gamma_0 \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$ and $A[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} r_1[\vec{t}/\vec{x}]$ and $M'[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:A[\vec{N}/\vec{x}]} r_2[\vec{t}/\vec{x}]$. Then by definition also $\Gamma_0 \rightsquigarrow_{\vec{x},y,\vec{N},N,\vec{t},t} \Gamma'$. Moreover, we obtain $A[\vec{N}/\vec{x}][N/y] \succ^{\mathcal{C}}_{\Gamma'} r_1[\vec{t}/\vec{x}][t/y]$ and $M'[\vec{N}/\vec{x}][N/y] \succ^{\mathcal{C}}_{\Gamma',x:A[\vec{N}/\vec{x}][N/y]} r_2[\vec{t}/\vec{x}][t/y]$ by the inductive hypothesis. Hence, recalling that $M[N/y]$ is not a Γ'-proof,

$$M[N/y] = (\lambda x : A.M')[\vec{N}/\vec{x}][N/y] \succ^{\mathcal{C}}_{\Gamma'} (f\vec{y})[\vec{t}/\vec{x}][t/y] = u[t/y]. \qquad ◀$$

▶ **Lemma 67.** *Assume $y \in V^{*^p}$.*

1. *If $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ then $y \notin \mathrm{FV}(\varphi)$.*
2. *If $M \succ^{\mathcal{C}}_{\Gamma} t$ then $y \notin \mathrm{FV}(t)$.*

Proof. Induction on the definition of $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $M \succ^{\mathcal{C}}_{\Gamma} t$. Note that if y is a Γ-subject then y is a Γ-proof, by the generation and start lemmas. Hence, the case $M = y \succ^{\mathcal{C}}_{\Gamma} y = t$ is impossible. ◀

▶ **Lemma 68.**

1. *If $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ then* $\mathrm{FV}(\varphi) = \mathrm{FV}(\mathrm{nf}_\varepsilon(M))$.
2. *If $M \succ^{\mathcal{C}}_{\Gamma} t$ then* $\mathrm{FV}(t) = \mathrm{FV}(\mathrm{nf}_\varepsilon(M))$.

Proof. Induction on the definition of $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $M \succ^{\mathcal{C}}_{\Gamma} t$, using Lemma 67. We show a few cases. The other cases are similar, trivial, or follow directly from the inductive hypothesis.

- $M = \Pi x : A.B \succ^{\mathcal{F}}_{\Gamma} \varphi_1 \to \varphi_2 = \varphi$. Then $\Gamma \vdash A : *^p$ and $A \succ^{\mathcal{F}}_{\Gamma} \varphi_1$ and $B \succ^{\mathcal{F}}_{\Gamma,x:A} \varphi_2$. By the inductive hypothesis $\mathrm{FV}(\mathrm{nf}_\varepsilon(A)) = \mathrm{FV}(\varphi_1)$ and $\mathrm{FV}(\mathrm{nf}_\varepsilon(B)) = \mathrm{FV}(\varphi_2)$. Note that $\mathrm{nf}_\varepsilon(M) = \Pi x : \mathrm{nf}_\varepsilon(A).\mathrm{nf}_\varepsilon(B)$, so $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = \mathrm{FV}(\mathrm{nf}_\varepsilon(A)) \cup (\mathrm{FV}(\mathrm{nf}_\varepsilon(B)) \setminus \{x\})$. Since $\Gamma \vdash A : *^p$, we have $x \in V^{*^p}$ by Corollary 28. Thus $x \notin \mathrm{FV}(\varphi_2)$ by Lemma 67, so $\mathrm{FV}(\varphi_2) = \mathrm{FV}(\mathrm{nf}_\varepsilon(B)) \setminus \{x\}$. Hence $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = \mathrm{FV}(\mathrm{nf}_\varepsilon(A)) \cup (\mathrm{FV}(\mathrm{nf}_\varepsilon(B)) \setminus \{x\}) = \mathrm{FV}(\varphi_1) \cup \mathrm{FV}(\varphi_2) = \mathrm{FV}(\varphi)$.
- $M = \Pi x : A.B \succ^{\mathcal{F}}_{\Gamma} \forall x.T(x,t) \to \psi = \varphi$. Then $\Gamma \vdash A : *^p$ and $A \succ^{\mathcal{C}}_{\Gamma} t$ and $B \succ^{\mathcal{F}}_{\Gamma,x:A} \psi$. By the inductive hypothesis $\mathrm{FV}(\mathrm{nf}_\varepsilon(A)) = \mathrm{FV}(t)$ and $\mathrm{FV}(\mathrm{nf}_\varepsilon(B)) = \mathrm{FV}(\psi)$. Note that $\mathrm{nf}_\varepsilon(M) = \Pi x : \mathrm{nf}_\varepsilon(A).\mathrm{nf}_\varepsilon(B)$, so $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = \mathrm{FV}(\mathrm{nf}_\varepsilon(A)) \cup (\mathrm{FV}(\mathrm{nf}_\varepsilon(B)) \setminus \{x\}) = (\mathrm{FV}(\mathrm{nf}_\varepsilon(A)) \cup \mathrm{FV}(\mathrm{nf}_\varepsilon(B))) \setminus \{x\}$ (by the variable convention we may assume $x \notin \mathrm{FV}(A)$). Thus $\mathrm{FV}(\varphi) = (\mathrm{FV}(t) \cup \mathrm{FV}(\psi)) \setminus \{x\} = \mathrm{FV}(\mathrm{nf}_\varepsilon(M))$.
- If M is a Γ-proof and $M \succ^{\mathcal{C}}_{\Gamma} \varepsilon = t$, then $\mathrm{nf}_\varepsilon(M) = \varepsilon$ by Theorem 30, so $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = \mathrm{FV}(t)$.
- $M = x \succ^{\mathcal{C}}_{\Gamma} x = t$. Then M is not a Γ-proof, so $\mathrm{nf}_\varepsilon(x) = x$ by Theorem 30. Hence $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = \mathrm{FV}(t)$.
- $M = M_1 M_2 \succ^{\mathcal{C}}_{\Gamma} t_1 t_2 = t$ and $M_1 \succ^{\mathcal{C}}_{\Gamma} t_1$ and $M_2 \succ^{\mathcal{C}}_{\Gamma} t_2$. In this case M is not a Γ-proof, so $M \not\to^*_\varepsilon \varepsilon$. Hence $\mathrm{nf}_\varepsilon(M) = \mathrm{nf}_\varepsilon(M_1)\mathrm{nf}_\varepsilon(M_2)$. Thus $\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = \mathrm{FV}(\mathrm{nf}_\varepsilon(M_1), \mathrm{nf}_\varepsilon(M_2)) = \mathrm{FV}(t_1, t_2) = \mathrm{FV}(t)$, using the inductive hypothesis.
- $M = (\lambda x : A.M')[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} (f\vec{y})[\vec{t}/\vec{x}]$ and $\Gamma' \vdash (\lambda x : A.M') : B$ and $\Gamma' \nvdash A : *^p$ and $f = \Lambda_1(x, r, t)$ and $\vec{y} = \mathrm{FV}(r, t) \setminus \{x\}$ and $\Gamma' \rightsquigarrow_{\vec{x},\vec{N},\vec{t}} \Gamma$ and $A[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} r[\vec{t}/\vec{x}]$ and $M'[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:A[\vec{N}/\vec{x}]} t[\vec{t}/\vec{x}]$. By the inductive hypothesis $\mathrm{FV}(\mathrm{nf}_\varepsilon(A[\vec{N}/\vec{x}])) = \mathrm{FV}(r[\vec{t}/\vec{x}])$

and $\mathrm{FV}(\mathrm{nf}_\varepsilon(M'[\vec{N}/\vec{x}])) = \mathrm{FV}(t[\vec{t}/\vec{x}])$. By the variable convention we may assume $x \notin \mathrm{FV}(A, N_1, \ldots, N_n)$, so

$$\begin{array}{rcl}\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) & = & \mathrm{FV}(\mathrm{nf}_\varepsilon(\lambda x : A[\vec{N}/\vec{x}].M'[\vec{N}/\vec{x}])) \\ & = & \mathrm{FV}(r[\vec{t}/\vec{x}], t[\vec{t}/\vec{x}]) \setminus \{x\}.\end{array}$$

Let $\{x_{i_1}, \ldots, x_{i_k}\} = \mathrm{FV}(r, t) \cap \{x_1, \ldots, x_n\}$ and let $t'_i = t_i[t_{i+1}/x_{i+1}] \ldots [t_n/x_n]$ for $i = 1, \ldots, n$. We then have $r[\vec{t}/\vec{x}] = r[t'_{i_1}/x_{i_1}, \ldots, t'_{i_k}/x_{i_k}]$ and $t[\vec{t}/\vec{x}] = t[t'_{i_1}/x_{i_1}, \ldots, t'_{i_k}/x_{i_k}]$. Thus

$$\mathrm{FV}(r[\vec{t}/\vec{x}], t[\vec{t}/\vec{x}]) = (\mathrm{FV}(r, t) \setminus \{x_{i_1}, \ldots, x_{i_k}\}) \cup \mathrm{FV}(t'_{i_1}, \ldots, t'_{i_k}).$$

Hence

$$\mathrm{FV}(\mathrm{nf}_\varepsilon(M)) = ((\mathrm{FV}(r, t) \setminus \{x_{i_1}, \ldots, x_{i_k}\}) \cup \mathrm{FV}(t'_{i_1}, \ldots, t'_{i_k})) \setminus \{x\}.$$

On the other hand, also $t = (f\vec{y})[\vec{t}/\vec{x}] = (f\vec{y})[t'_{i_1}/x_{i_1}, \ldots, t'_{i_k}/x_{i_k}]$ and $\vec{y} = \mathrm{FV}(r, t) \setminus \{x\}$. By the inductive hypothesis $\mathrm{FV}(t_i) = \mathrm{FV}(\mathrm{nf}_\varepsilon(N_i))$, so $x \notin \mathrm{FV}(t_i)$. Hence also $x \notin \mathrm{FV}(t'_i)$. Therefore

$$\begin{array}{rcl}\mathrm{FV}(t) & = & (\mathrm{FV}(r, t) \setminus \{x, x_{i_1}, \ldots, x_{i_k}\}) \cup \mathrm{FV}(t'_{i_1}, \ldots, t'_{i_k}) \\ & = & ((\mathrm{FV}(r, t) \setminus \{x_{i_1}, \ldots, x_{i_k}\}) \cup \mathrm{FV}(t'_{i_1}, \ldots, t'_{i_k})) \setminus \{x\} \\ & = & \mathrm{FV}(\mathrm{nf}_\varepsilon(M)).\end{array}$$ ◀

▶ **Lemma 69.** *Assume* $\Gamma =_\varepsilon \Gamma'$.

1. *If* $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ *and* $M' \succ^{\mathcal{F}}_{\Gamma'} \varphi$ *then* $M =_\varepsilon M'$.
2. *If* $M \succ^{\mathcal{C}}_{\Gamma} t$ *and* $M' \succ^{\mathcal{C}}_{\Gamma'} t$ *then* $M =_\varepsilon M'$.

Proof. Induction on the definition of $M \succ^{\mathcal{F}}_{\Gamma} \varphi$ and $M \succ^{\mathcal{C}}_{\Gamma} t$. We show a few cases. The other cases are similar, trivial, or follow directly from the inductive hypothesis.

- $M = \Pi x : A.B$ and $M' = \Pi x : A'.B'$ and $\varphi = \varphi_1 \to \varphi_2$. Then $A \succ^{\mathcal{F}}_{\Gamma} \varphi_1$ and $A' \succ^{\mathcal{F}}_{\Gamma} \varphi_1$ and $B \succ^{\mathcal{F}}_{\Gamma,x:A} \varphi_2$ and $B' \succ^{\mathcal{F}}_{\Gamma',x:A'} \varphi_2$. By the inductive hypothesis $A =_\varepsilon A'$. Hence $\Gamma, x : A =_\varepsilon \Gamma', x : A'$, so $B =_\varepsilon B'$ by the inductive hypothesis. Therefore $M =_\varepsilon M'$.
- $t = \varepsilon$ and M is a Γ-proof and M' is a Γ'-proof. By Theorem 30 we have $M \to^*_\varepsilon \varepsilon$ and $M' \to^*_\varepsilon \varepsilon$, so $M =_\varepsilon M'$.
- $t = t_1 t_2$ and $M = M_1 M_2$ and $M' = M'_1 M'_2$ and $M_i \succ^{\mathcal{C}}_{\Gamma} t_i$ and $M'_i \succ^{\mathcal{C}}_{\Gamma'} t_i$. By the inductive hypothesis $M_i =_\varepsilon M'_i$. Hence $M =_\varepsilon M'$.
- $t = (f\vec{y})[\vec{t}/\vec{x}] = (f\vec{y'})[\vec{t'}/\vec{x'}]$ and $f = \Lambda_1(x, r, u) = \Lambda_1(x', r', u')$ and $\vec{y} = \mathrm{FV}(r, u) \setminus \{x\}$ and $\vec{y'} = \mathrm{FV}(r', u') \setminus \{x'\}$ and $M = (\lambda x : A.B)[\vec{N}/\vec{x}]$ and $M' = (\lambda x : A'.B')[\vec{N'}/\vec{x'}]$ and there is a bijection $\sigma : V \to V$ such that $\sigma(y'_i) = y_i$ and $\sigma(x') = x$ and $\sigma(r') = r$ and $\sigma(u') = u$. We also have $A[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} r[\vec{t}/\vec{x}]$ and $B[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma,x:A[\vec{N}/\vec{x}]} u[\vec{t}/\vec{x}]$ and $A'[\vec{N'}/\vec{x'}] \succ^{\mathcal{C}}_{\Gamma'} r'[\vec{t'}/\vec{x'}]$ and $B'[\vec{N'}/\vec{x'}] \succ^{\mathcal{C}}_{\Gamma',x:A'[\vec{N'}/\vec{x'}]} u'[\vec{t'}/\vec{x'}]$. Let $p_i = t_i[t_{i+1}/x_{i+1}] \ldots [t_n/x_n]$ and $p'_i = t'_i[t'_{i+1}/x'_{i+1}] \ldots [t'_m/x'_m]$. We have

$$(f\vec{y})[p_1/x_1, \ldots, p_n/x_n] = (f\vec{y'})[p'_1/x'_1, \ldots, p'_m/x'_m].$$

  Without loss of generality we may assume that there is $k \le n$ such that $x_i = y_i$ and $x'_i = y'_i$ and $p_i = p'_i$ for $i \le k$ (this may be always achieved by taking the "missing" p_is (p'_is) to be equal to y_is (y'_is)), and $x_i \notin \vec{y} = \mathrm{FV}(r, u) \setminus \{x\}$ for $i > k$. We may also assume there is $k \le k' \le m$ such that $x'_i = y'_i$ for $k < i \le k'$ and $x'_i \notin \vec{y'} = \mathrm{FV}(r', u') \setminus \{x'\}$ for $i > k'$. Then $p'_i = y_i$ for $k < i \le k'$. Hence, in fact we may assume $k = k'$, by taking the missing p_is equal to y_is.

By the variable convention we may also assume $x, x' \notin \{x_1, \ldots, x_n, x'_1, \ldots, x'_m\}$. Hence for $i > k$ we have $x_i \notin \mathrm{FV}(r, u)$ and $x'_i \notin \mathrm{FV}(r', u')$. Recall that $\sigma^{-1}(y_i) = y'_i$. Since also $p_i = p'_i$ and $x_i = \sigma(x'_i)$ for $i \leq k$:

$$\begin{array}{rcl} r[\vec{t}/\vec{x}] & = & r[p_1/x_1, \ldots, p_k/x_k] \\ & = & r[p'_1/\sigma(x'_1), \ldots, p'_k/\sigma(x'_k)] \\ & = & \sigma^{-1}(r)[p'_1/x'_1, \ldots, p'_k/x'_k] \\ & = & r'[p'_1/x'_1, \ldots, p'_k/x'_k] \\ & = & r'[\vec{t'}/\vec{x'}] \end{array}$$

and

$$\begin{array}{rcl} u[\vec{t}/\vec{x}] & = & u[p_1/x_1, \ldots, p_k/x_k] \\ & = & u[p'_1/\sigma(x'_1), \ldots, p'_k/\sigma(x'_k)] \\ & = & \sigma^{-1}(u)[p'_1/x'_1, \ldots, p'_k/x'_k] \\ & = & u'[p'_1/x'_1, \ldots, p'_k/x'_k] \\ & = & u'[\vec{t'}/\vec{x'}]. \end{array}$$

So by the inductive hypothesis $A[\vec{N}/\vec{x}] =_\varepsilon A'[\vec{N'}/\vec{x'}]$ and thus also $\Gamma, x : A[\vec{N}/\vec{x}] =_\varepsilon \Gamma', x : A'[\vec{N'}/\vec{x'}]$, so $B[\vec{N}/\vec{x}] =_\varepsilon B'[\vec{N'}/\vec{x'}]$ by applying the inductive hypothesis again. This implies that $M =_\varepsilon M'$. ◀

▶ **Lemma 74.** *Suppose $\Gamma \succ \Delta$ and $\Delta_{\mathrm{Ax}}, \Delta \vdash XQ_1 \ldots Q_m : \psi$, where each Q_i is either an individual term or a reconstructible proof term. Let $\Gamma_0 = x_1 : A_1, \ldots, x_n : A_n$ be such that $m = \mathrm{len}_{\mathbb{A}}(\Gamma_0)$ and Γ, Γ_0 is a legal context. If $(X : \gamma) \in \Delta_{\mathrm{Ax}}, \Delta$ with $\varphi \succ^{\mathbb{A}}_{\Gamma;\Gamma_0} \gamma$, then there exist $N_1, \ldots, N_n$ and $u_1, \ldots, u_n$ such that $\psi = \varphi[\vec{u}/\vec{x}]$ and $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma$.*

Proof. Induction on n. If $n = 0$ then $m = 0$ and $\psi = \varphi$, so we are done. Thus suppose $\Gamma_0 = \Gamma'_0, x_{n+1} : A_{n+1}$.

First assume $\Gamma, \Gamma'_0 \vdash A_{n+1} : s$ and $s \neq *^p$. Then $A_{n+1} \succ^{\mathcal{C}}_{\Gamma,\Gamma'_0} t$ and $\forall x_{n+1}.T(x_{n+1}, t) \to \varphi \succ^{\mathbb{A}}_{\Gamma;\Gamma'_0} \gamma$ and $\mathrm{len}_{\mathbb{A}}(\Gamma'_0) = \mathrm{len}_{\mathbb{A}}(\Gamma_0) - 2$. Also

$$\Delta_{\mathrm{Ax}}, \Delta \vdash XQ_1 \ldots Q_{m-2} : \forall x_{n+1}.T(x_{n+1}, r) \to \psi'$$

and $Q_{m-1} = u_{n+1}$ is an individual term and $Q_m = D$ is a reconstructible proof term such that $\Delta_{\mathrm{Ax}}, \Delta \vdash D : T(u_{n+1}, r)$ and $\psi = \psi'[u_{n+1}/x_{n+1}]$. By the inductive hypothesis there exist $N_1, \ldots, N_n$ and $u_1, \ldots, u_n$ such that $r = t[\vec{u}/\vec{x}]$ and $\psi' = \varphi[\vec{u}/\vec{x}]$ and $\Gamma, \Gamma'_0 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma$. Then $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma, x_{n+1} : A_{n+1}[\vec{N}/\vec{x}]$ by Lemma 60. Also $A_{n+1}[\vec{N}/\vec{x}] \succ^{\mathcal{C}}_{\Gamma} r$ by Corollary 66. Since $\Gamma, \Gamma'_0 \vdash A_{n+1} : s$, we have $\Gamma \vdash A_{n+1}[\vec{N}/\vec{x}] : s$ by Lemma 61. Because we also have $\Delta_{\mathrm{Ax}}, \Delta \vdash D : T(u_{n+1}, r)$ and D is reconstructible, by 2 in Definition 73 there is N_{n+1} with $N_{n+1} \succ^{\mathcal{C}}_{\Gamma} u_{n+1}$ and $\Gamma \vdash N_{n+1} : A_{n+1}[\vec{N}/\vec{x}]$. Also $N_{n+1} \sim x_{n+1}$ by Lemma 32. Thus $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x}, x_{n+1}, \vec{N}, N_{n+1}, \vec{u}, u_{n+1}} \Gamma$ by definition of $\rightsquigarrow$. Moreover, $\psi = \psi'[u_{n+1}/x_{n+1}] = \varphi[u_1/x_1] \ldots [u_{n+1}/x_{n+1}]$.

Now assume $\Gamma, \Gamma'_0 \vdash A_{n+1} : *^p$. Then $A_{n+1} \succ^{\mathcal{F}}_{\Gamma,\Gamma'_0} \varphi'$ and $\varphi' \to \varphi \succ^{\mathbb{A}}_{\Gamma;\Gamma'_0} \gamma$ and $\mathrm{len}_{\mathbb{A}}(\Gamma'_0) = \mathrm{len}_{\mathbb{A}}(\Gamma_0) - 1$. Also

$$\Delta_{\mathrm{Ax}}, \Delta \vdash XQ_1 \ldots Q_{m-1} : \alpha \to \psi$$

and $Q_m = D$ is a reconstructible proof term such that $\Delta_{\mathrm{Ax}}, \Delta \vdash D : \alpha$. By the inductive hypothesis there are $N_1, \ldots, N_n$ and $u_1, \ldots, u_n$ such that $\alpha = \varphi'[\vec{u}/\vec{x}]$, $\psi = \varphi[\vec{u}/\vec{x}]$ and $\Gamma, \Gamma'_0 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma$. Then $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x}, \vec{N}, \vec{u}} \Gamma, x_{n+1} : A_{n+1}[\vec{N}/\vec{x}]$ by Lemma 60. Also $A_{n+1}[\vec{N}/\vec{x}] \succ^{\mathcal{F}}_{\Gamma}$

$\varphi'[\vec{u}/\vec{x}] = \alpha$ by Corollary 66. Since $\Gamma, \Gamma_0' \vdash A_{n+1} : *^p$, we have $\Gamma \vdash A_{n+1}[\vec{N}/\vec{x}] : *^p$ by Lemma 61. Because we also have $\Delta_{\mathrm{Ax}}, \Delta \vdash D : \alpha$ and D is reconstructible, by 1 in Definition 73 there is N_{n+1} with $\Gamma \vdash N_{n+1} : A_{n+1}[\vec{N}/\vec{x}]$. Because N_{n+1} is a Γ-proof, we have $N_{n+1} \succ_{\Gamma}^{\mathcal{C}} \varepsilon$. Also $N_{n+1} \sim x_{n+1}$ by Lemma 32. Thus $\Gamma, \Gamma_0 \rightsquigarrow_{\vec{x}, x_{n+1}, \vec{N}, N_{n+1}, \vec{u}, \varepsilon} \Gamma$ by definition of $\rightsquigarrow$. Moreover, because $x_{n+1} \in V^{*^p}$ we have $x_{n+1} \notin \mathrm{FV}(\psi)$, and thus $\psi = \varphi[u_1/x_1] \dots [u_n/x_n][\varepsilon/x_{n+1}]$. ◀

Permutability in Proof Terms for Intuitionistic Sequent Calculus with Cuts

José Espírito Santo
Centro de Matemática, Universidade do Minho, Portugal

Maria João Frade
HASLab/INESC TEC & Universidade do Minho, Portugal

Luís Pinto
Centro de Matemática, Universidade do Minho, Portugal

Abstract

This paper gives a comprehensive and coherent view on permutability in the intuitionistic sequent calculus with cuts. Specifically we show that, once permutability is packaged into appropriate global reduction procedures, it organizes the internal structure of the system and determines fragments with computational interest, both for the computation-as-proof-normalization and the computation-as-proof-search paradigms. The vehicle of the study is a λ-calculus of multiary proof terms with generalized application, previously developed by the authors (the paper argues this system represents the simplest fragment of ordinary sequent calculus that does not fall into mere natural deduction). We start by adapting to our setting the concept of *normal* proof, developed by Mints, Dyckhoff, and Pinto, and by defining *natural* proofs, so that a proof is normal iff it is natural and cut-free. Natural proofs form a subsystem with a transparent Curry-Howard interpretation (a kind of formal vector notation for λ-terms with vectors consisting of lists of lists of arguments), while searching for normal proofs corresponds to a slight relaxation of focusing (in the sense of LJT). Next, we define a process of permutative conversion to natural form, and show that its combination with cut elimination gives a concept of *normalization* for the sequent calculus. We derive a systematic picture of the full system comprehending a rich set of reduction procedures (cut elimination, flattening, permutative conversion, normalization, focalization), organizing the relevant subsystems and the important subclasses of cut-free, normal, and focused proofs.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases sequent calculus, permutative conversion, Curry-Howard isomorphism, vector of arguments, generalized application, normal proof, natural proof, cut elimination, flatenning, normalization, focalization

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.10

Acknowledgements The first and the last authors were partially financed by Fundação para a Ciência e a Tecnologia (FCT) through project UID/MAT/00013/2013. The first author got financial support by the COST action CA15123 EUTYPES.

1 Introduction

Traditionally, the sequent calculus is associated with the computation-as-proof-search paradigm [16], but progress in the understanding of the Curry-Howard correspondence showed that sequent calculus has a lot to offer to the computation-as-proof-normalization paradigm as well, from alternative λ-term representations which are useful for machine handling [12, 2] to logical foundations for evaluation strategies [3, 24]. Nevertheless, the mentioned

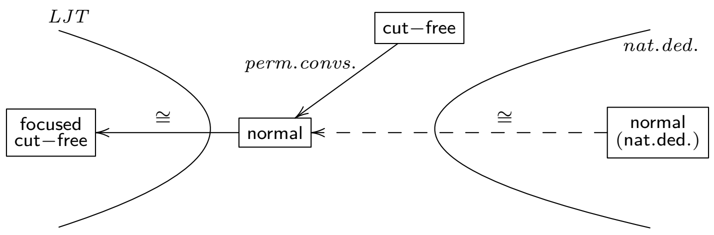


Figure 1 The cut-free setting.

progress has been slow: even if we are not anymore in the situation where textbooks had almost nothing to report about Curry-Howard for sequent calculus [11, 22, 18], it seems basic discoveries are still being made after decades of investigation [1, 5].

One source of difficulties in completing the Curry-Howard interpretation of sequent calculus and cut-elimination is the phenomenon of permutability of inferences [14], which sometimes is dubbed "bureaucracy". Permutability can be faced with several attitudes: either by decreeing Curry-Howard for sequent calculus an outright impossibility [11]; or by regarding the sequent calculus as meta-notation for alternative, supposedly permutation-free formalisms, like natural deduction [19] or proof nets [10]; or by restricting one's attention to permutability-free fragments of sequent calculus - cf. the flourishing area of focusing [15, 21].

In this paper we face permutability squarely, in the context of intuitionistic propositional logic, for a simple and standard sequent calculus including cut, the latter system presented as a typed λ-calculus, and we show that the (perhaps dull) complexity engendered by permutability can be tamed and organized appropriately and meaningfully, in a way that enlightens the internal structure and the computational interpretation of the entire sequent calculus.

Our starting point is the familiar situation in the cut-free setting, depicted in Fig. 1: there is a set of permutation-free proofs, named *normal* by Mints [17], which are in 1-1 correspondence with normal natural deductions; in addition [4]: (i) normal derivations are normal (i.e. irreducible) w.r.t. a rewriting system of permutative conversions; (ii) normal derivations are in 1-1 correspondence with cut-free LJT-proofs (that is, cut-free $\overline{\lambda}$-terms [12]). So permutation-freeness has a privileged relationship with natural deduction (as we already knew since Zucker [25]); and, in this setting, permutation-freeness is indistinguishable from focusedness (in the sense of LJT).

What is the high-level lesson of this situation? Permutability can be organized into a reduction procedure determining a class of normal forms which are meaningful both for functional computation and for proof-search. Shorter: if permutability of inferences is packaged into a global reduction procedure, it becomes an organizing tool at the macro level that brings out meaning.

In this paper, guided by this heuristic, we move to the cut-full setting. Needless to say, the situation becomes rather more complex, as cut-elimination is present and potentially interacts with permutability, we have to deal with (sub)systems of the full rewriting system rather than classes of normal forms, and desirably the familiar cut-free situation falls out as a corollary of the cut-full picture.

In a nutshell, these are our results: we adapt to our setting the notion of *normal* proof [17, 4] and pin down the bottom-up proof-search procedure it determines, which is a slight relaxation of focusing; we introduce a permutation-free notion of *natural* proof so that a proof is normal iff it is natural and cut-free; we prove natural proofs are closed for cut-elimination, constituting a subsystem with a transparent Curry-Howard interpretation; we prove natural proofs are the normal forms w.r.t. a certain permutative conversion γ; we give a systematic description of the internal structure of the sequent calculus we consider in terms of the two "bureaucratic" conversions: the conversion γ, and another conversion named μ, which, among other things, is the bridge between natural proofs and focused proofs; we investigate the commutation between the (macro level) reduction procedures and this allows us to identify a *normalization* procedure on the set of all sequent calculus proofs, for which the normal proofs are the irreducible forms, and which is a combination of cut-elimination and permutative conversion.

Technical overview. In order to isolate the syntactic difficulties caused by permutability, we reduce the logical apparatus to a minimum: intuitionistic implication is the single connective studied, and the sequent calculus analyzed is designed to be the simplest one that goes beyond natural deduction with general elimination rules [23] (i.e. beyond the λ-calculus with generalized application ΛJ [13]). Quite conveniently, the resulting system is precisely the $\lambda\mathbf{Jm}$-calculus introduced by two of the authors [8, 9] and further studied in [6] - a system which may be seen as the "multiary" [20] version of ΛJ. Multiarity just means that the generalized application constructor handles a non-empty list of arguments, thus ΛJ may be recast as the *unary fragment* $\lambda\mathbf{J}$, where the list of arguments is singular [9]; but multiarity engenders the mentioned conversion μ, firstly introduced in [20] as a technical tool in a termination argument, which turns out to play a crucial role in the description of the internal structure of $\lambda\mathbf{Jm}$ and its subtle connection with natural deduction [8, 6].

In extending the situation in Fig. 1 to the cut-full setting, we have to avoid an immediate pitfall: to consider an excessively narrow class of derivations possibly containing cuts. Ordinary cut-free derivations may be seen as fully-normal natural deductions with general application [23]. So, if we merely "close under substitution" such derivations, we end up with natural deduction, or the ΛJ-calculus [13]. Similarly, if we merely add appropriate cut-rules to LJT, we end up with some variant of the $\overline{\lambda}$-calculus [12]. We do something different: we recast in $\lambda\mathbf{Jm}$ (a system designed to not fall in mere natural deduction) the situation in Fig. 1, and the result is illustrated in Fig. 2. In $\lambda\mathbf{Jm}$, natural deduction and LJT are captured internally[1], and the normal derivations of [17, 4] are just the unary case of a more general concept of normal derivation, which is studied here for the first time, as it escaped the catalogue of normal forms in [6].

In fact, we will rather develop Fig. 3, concerning the cut-full setting, and extract Fig. 2 as a corollary, given that cut-elimination links each system in Fig. 3 to a corresponding class in Fig. 2. Specifically: Section 3 defines and studies natural derivations and how they define a subsystem $\lambda\mathbf{nm}$ with clear computational interpretation. This includes studying the cut-free natural (=normal) derivations, in particular in their relation to focused proofs. Section 4 goes beyond the permutation-free fragment $\lambda\mathbf{nm}$, and studies permutative conversion γ, for which the natural proofs are the irreducible forms. This includes studying the interaction

[1] This is in contrast with [4], where natural deduction, LJT and sequent calculus are three different systems - this is why in Fig. 1 we see the curved borders, while in Fig. 2 these borders disappear, their location being memorized with dotted lines. Beware that there are several inclusion that hold in Fig. 2, since all classes live in the same system: the class of unary cut-free (resp. unary normal) derivations is included in the class of cut-free (resp. normal) derivations; and normal natural deductions are a subclass of unary cut-free derivations ($\cong$ fully normal natural deductions with general eliminations). Such inclusions are not depicted to avoid clutter and because they are not witnessed by reduction rules of $\lambda\mathbf{Jm}$. The map denoted with a dashed line is not a mere inclusion, but is not studied in this paper. Similar remarks apply as well to Fig. 3.

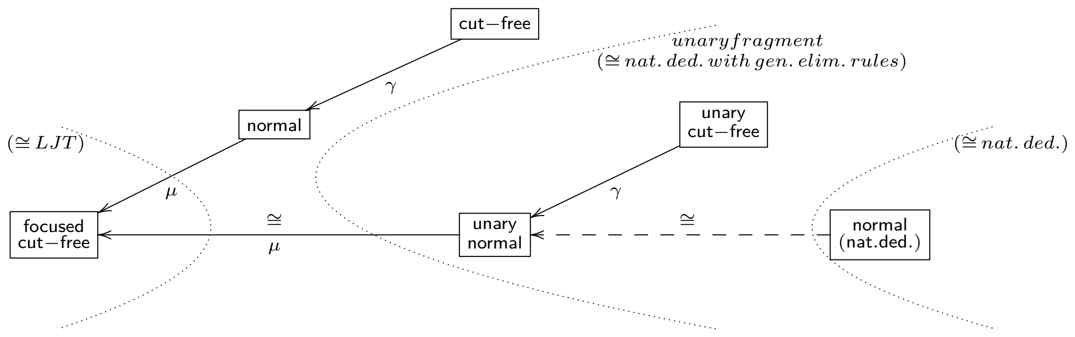


Figure 2 The cut-free setting in the multiary calculus $\lambda\mathbf{Jm}$ (classes and maps).

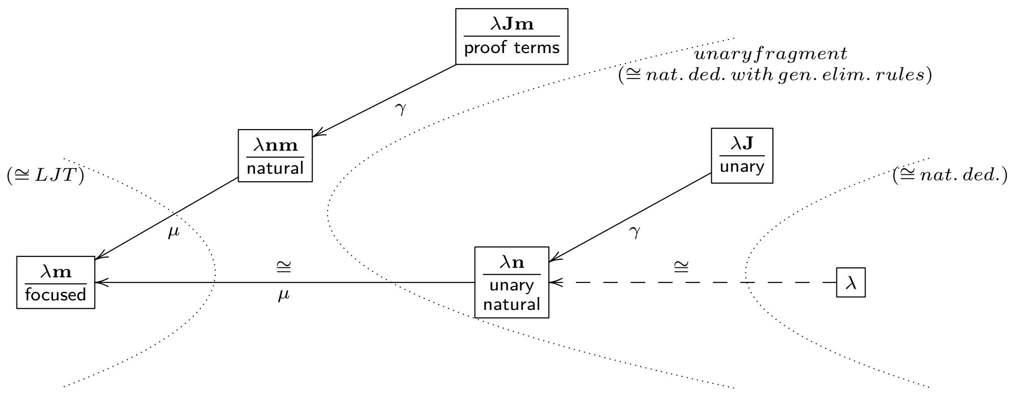


Figure 3 The cut-full setting in the multiary calculus $\lambda\mathbf{Jm}$ (calculi and morphisms).

of γ with cut-elimination, which leads to the definition of normalization in $\lambda\mathbf{Jm}$. Section 2 recapitulates $\lambda\mathbf{Jm}$, while Section 5 concludes.

2 The sequent calculus $\lambda\mathbf{Jm}$

In the first subsection we recall $\lambda\mathbf{Jm}$, while in the second we argue why $\lambda\mathbf{Jm}$ is a simple and standard presentation of the intuitionistic sequent calculus.

2.1 A recapitulation of $\lambda\mathbf{Jm}$

Proof expressions and typing. Expressions E are generated by the following grammar:

$$\begin{array}{rrcl}
\text{(proof terms)} & t,u,v & ::= & x \mid \lambda x.t \mid ta \\
\text{(gm-arguments)} & a & ::= & (u,l,c) \\
\text{(lists)} & l & ::= & u::l \mid [] \\
\text{(continuations)} & c & ::= & (x)v
\end{array}$$

We will just say “term” instead of “proof term”. A *value* V is a term of the form x or $\lambda x.t$. The word “continuation” is chosen for its intuitive appeal, with no connection with technical meanings of the word intended.[2]

[2] In the previous publications on $\lambda\mathbf{Jm}$, the system was presented with two syntactic classes only: terms and lists. In fact, since continuations are generated by a single constructor and used only once in the

$$\frac{}{x:A,\Gamma\vdash x:A}\ \textit{Axiom} \qquad \frac{x:A,\Gamma\vdash t:B}{\Gamma\vdash\lambda x.t:A\supset B}\ \textit{Right}$$

$$\frac{\Gamma\vdash t:A\supset B \quad \Gamma;A\supset B\vdash a:C}{\Gamma\vdash ta:C}\ \textit{Cut} \qquad \frac{\Gamma\vdash u:A \quad \Gamma;B\vdash l:C \quad \Gamma|C\vdash c:D}{\Gamma;A\supset B\vdash(u,l,c):D}\ \textit{Leftm}$$

$$\frac{}{\Gamma;C\vdash[]:C}\ \textit{Ax} \qquad \frac{\Gamma\vdash u:A \quad \Gamma;B\vdash l:C}{\Gamma;A\supset B\vdash u::l:C}\ \textit{Lft} \qquad \frac{x:C,\Gamma\vdash v:D}{\Gamma|C\vdash(x)v:D}\ \textit{Select}$$

Figure 4 Typing rules for $\lambda\mathbf{Jm}$.

We identify simple types with formulas of intuitionistic, propositional, implicational logic. They are ranged over by A, B, C, D. If $B = B_1 \supset \cdots \supset B_n$ ($n \geq 1$) then we say C is a *suffix* of B if $C = B_j \supset \cdots \supset B_n$, for some $1 \leq j \leq n$. Contexts Γ are sets of variable declarations $x : A$, with at most one declaration per variable. The typing rules are in Fig. 4. They handle four kinds of sequents, one per syntactic class:

$$(i)\quad \Gamma\vdash t:A \qquad (ii)\quad \Gamma;A\supset B\vdash a:D \qquad (iii)\quad \Gamma;B\vdash l:C \qquad (iv)\quad \Gamma|C\vdash c:D\ . \qquad (1)$$

In the sequents of kinds (ii) and (iii), the distinguished formula on the left hand side (the formula separated by ;) is main in the last inference, whereas in the sequents of kind (iv) the distinguished formula C is merely selected from the context. In addition, in a derivable sequent of kind (iii), C is a suffix of B.

Inference rule Lft is a special left-introduction rule, because its right premiss is a sequent of kind (iii): this implies that $B = B_1 \supset \cdots \supset B_m \supset C$, for some $m \geq 0$, and the referred premiss is the conclusion of a chain of m other Lft inferences. There is another primitive left introduction rule, $Leftm$, the single rule for typing gm-arguments. Since its middle premiss is a sequent of kind (iii), the main formula of $Leftm$ has the form $A \supset B_1 \supset \cdots \supset B_m \supset C$, for some $m \geq 0$, and is obtained after a sequence of $m+1$ left introductions. We call $Leftm$ a *multiary* left-introduciton rule, while its particular case where the middle premiss is the conclusion of Ax, $m = 0$, $l = []$, and $B = C$ may be called a *unary* left introduction.

In $\Gamma;A \supset B \vdash a : D$, with $a = (u,l,c)$, and $\Gamma;A \supset B \vdash l' : C$, the formula $A \supset B$ is introduced linearly, *i.e* without contraction, in the last inference; the difference between the two sequents is that C is a suffix of B, whereas the same is not true of D, unless $c = (x)x$. The trivial cut xa gives name x to the formula $A \supset B$: we have the admissible rules

$$\frac{\Gamma;A\supset B\vdash a:D}{\Gamma\vdash xa:D}\ \textit{Unselect} \qquad \frac{\Gamma\vdash u:A \quad \Gamma;B\vdash l:C \quad \Gamma|C\vdash c:D}{\Gamma\vdash x(u,l,c):D}$$

where $(x : A \supset B) \in \Gamma$. So xa represents simultaneously an inference that "unselects" an antecedent formula, and a form of left introduction without linearity constraint.

If $x \notin a$ and $t = xa$ we say x is *main and linear in the application* t (abbreviation $mla(x,t)$). In that case, $c = (x)xa$ represents an argument a coerced to a continuation. The

grammar (in the formation of gm-arguments), they could easily be dispensed with; and the very same holds of gm-arguments. However, the separation into finer classes gives more flexibility. This flexibility is a convenience, as quite often we can avoid writing the entire expression $t(u,l,(x)v)$ - see e.g. the simpler definition of reduction rules π and μ; but such flexibility is also a necessity - see the particular form of continuations (called pseudo-lists) extensively studied in the next section.

admissible typing rule is

$$\frac{\Gamma;A\supset B\vdash a:D}{\Gamma|A\supset B\vdash (x)xa:D} \qquad \text{due to} \qquad \cfrac{\cfrac{\cfrac{\Gamma;A\supset B\vdash a:D}{x:A\supset B,\Gamma;A\supset B\vdash a:D}\,Weak}{x:A\supset B,\Gamma\vdash xa:C}\,Unselect}{\Gamma|A\supset B\vdash (x)xa:D}\,Select \tag{2}$$

where $(x : A \supset B) \notin \Gamma$. In the first figure we see that the distinguished position in the l.h.s. is changed, losing the information about linearity. In general, there is no coercion of a continuation to an argument or list. As hinted above, a non-empty list $u :: l$ can be coerced to an argument $(u, l, (x)x)$ (and then to a continuation). A direct "coercion" of a list to a continuation is given by $[]^\natural = (x)x$ and $(u :: l)^\natural = (x)x(u, [], l^\natural)$, with $x \notin u, l$. The admissible typing rule is

$$\frac{\Gamma;B\vdash l:C}{\Gamma|B\vdash l^\natural:C} \tag{3}$$

Derived syntax. In order to formulate the reduction rules, we have to introduce some derived syntactic operations. A familiar one is ordinary substitution of variables by terms, denoted $\mathbf{s}(t, x, E)$. It is becoming increasingly clear [12, 5] (and this paper just confirms this) that mechanisms of vectorization of arguments for functional applications are at the heart of the computational interpretation of sequent calculus. Here is a careful definition of the append operations in $\lambda\mathbf{Jm}$:

▶ **Definition 1** (Append operations).
1. The term $t@a$ is defined by $V@a = Va$ if V is a value; and by $(ta')@a = t(a'@a)$.
2. The argument $a'@a$ is defined by $(u, l, c)@a = (u, l, c@a)$.
3. The continuation $c@a$ is defined by $((x)v)@a = (x)(v@_x a)$.
4. The term $v@_x a$ is defined by $(xa')@_x a = x(a'@a)$ if $x \notin a'$; and by $v@_x a = va$, otherwise.
5. The continuation $c@c'$ is defined by: $((x)x)@c' = c'$; $((x)x(u, l, c))@c' = (x)x(u, l, c@c')$, if $x \notin u, l, c$; and $((x)v)@c' = (x)(v@c')$, otherwise.
6. The term $t@c$ is defined by $t@(x)v = \mathbf{s}(t, x, v)$.
7. The list $l@l'$ is defined by $[]@l' = l'$ and $(u :: l)@l' = u :: (l@l')$.

Some immediate comments about these append operators: $t@a$ will be used in the definition of a special substitution operator (Def. 39 in Section 4); $v@_x a$ is used in the definition of $c@a$, and the idea goes back to [8]; $a@a'$ allows a very short definition of the reduction rule π; $c@a$ is used in the definition of $a@a'$; $c@c'$ will allow the definition of $L@L'$ in Section 3; $l@l'$ is necessary for the definition of reduction rule μ.

Recall that an argument a can be "coerced" to a continuation $(z)za$, if $z \notin a$. The next lemma shows $c@a$ could have been defined via $c@c'$.

▶ **Lemma 2** (Coherence of append).
1. *$c@a = c@(z)za$, if $z \notin a$.*
2. *$(x)(v@_x a) = ((x)v)@(z)za$, if $x, z \notin a$.*

Proof. By simultaneous induction on c and v. It is interesting to see how the various definitions in Def. 1 cooperate to produce the result. ◀

▶ **Lemma 3** (Admissible typing rules). *The typing rules in Fig. 5 are admissible.*

$$\frac{\Gamma \vdash t : A \supset B \quad \Gamma; A \supset B \vdash a : C}{\Gamma \vdash t@a : C}\ (i) \qquad \frac{\Gamma; A \supset B \vdash a' : C_1 \supset C_2 \quad \Gamma; C_1 \supset C_2 \vdash a : D}{\Gamma; A \supset B \vdash a'@a : D}\ (ii)$$

$$\frac{\Gamma | A \vdash c : B_1 \supset B_2 \quad \Gamma; B_1 \supset B_2 \vdash a : C}{\Gamma | A \vdash c@a : C}\ (iii) \qquad \frac{x : D, \Gamma \vdash v : A \supset B \quad \Gamma; A \supset B \vdash a : C}{x : D, \Gamma \vdash v@_x a : C}\ (iv)$$

$$\frac{\Gamma | C \vdash c : D \quad \Gamma | D \vdash c' : E}{\Gamma | C \vdash c@c' : E}\ (v) \qquad \frac{\Gamma \vdash t : A \quad \Gamma | A \vdash c : B}{\Gamma \vdash t@c : B}\ (vi) \qquad \frac{\Gamma \vdash t : A \quad \Gamma, x : A \vdash v : B}{\Gamma \vdash \mathbf{s}(t, x, v) : B}\ (vii)$$

$$\frac{\Gamma; A \vdash l : B \quad \Gamma; B \vdash l' : C}{\Gamma; A \vdash l@l' : C}\ (viii)$$

Figure 5 Typing rules for derived syntactic operators.

$$\begin{array}{rrcl}
(\beta_1) & (\lambda x.t)(u, [], (y)v) & \to & \mathbf{s}(\mathbf{s}(u, x, t), y, v) \\
(\beta_2) & (\lambda x.t)(u, u' :: l, c) & \to & (\mathbf{s}(u, x, t))(u', l, c) \\
(\pi) & (ta)a' & \to & t(a@a') \\
(\mu) & (u, l, (x)x(u', l', c')) & \to & (u, l@(u' :: l'), c'),\ \text{ if } x \notin u', l', c'
\end{array}$$

Figure 6 Reduction rules of $\lambda\mathbf{Jm}$.

Proof. Rule (i) follows immediately from rule (ii). Rules (ii), (iii) and (iv) are proved by simultaneous induction on a', c and v. Rule (vi) follows immediately from rule (vii). Rule (vii) is proved together with similar statements for a, l and c by simultaneous induction. Rule (v) follows by induction on c with the help of rule (vi). Rule (viii) is proved by induction on l'. ◀

So, every derived syntactic operator is typed with a corresponding variant of the cut rule, and each such operator is the term representation of the operation on derivations produced by the elimination of the corresponding cut. Such operations on derivations may be extracted from the proof of the previous lemma. All of them, except for the cuts (v), (vi) and (vii), consist in permuting the cut to the left, as long as this is made possible by the repetition of the cut formula; for cuts (vi) and (vii) the corresponding operation performs a similar permutation to the right; for cut (v) the operation is an hybrid of permutation to the left and to the right.

Reduction rules. The reduction rules of $\lambda\mathbf{Jm}$ are given in Fig. 6. All rules but μ are relations on terms, while μ is a relation on arguments. We let $\beta := \beta_1 \cup \beta_2$. Rule μ is the "abbreviation" conversion due to [20]. Rule π of this paper is not the "lazy" variant of [8, 9], where argument a' is appended to argument a in a stepwise fashion, but rather corresponds to the rule π' of the cited papers. This is due to the definition of $v@_x a$, which is not merely va, but instead triggers a new appending process in some cases.[3] See some remarks about the computational interpretation of these rules after Lemma 4.

The compatible closure $\to_R$ of a reduction rule R is obtained by closing R under the rules in Fig.7[4]. We use the notations $\to_R^=$, $\to_R^+$, and $\to_R^*$ to denote the reflexive, the transitive,

[3] In ΛJ [13] rule π is also of the "lazy" kind.
[4] This detailed naming of the closure rules will be intensively used in Section 3, where we will consider alternative notions of compatible closure.

$$\frac{t \to t'}{\lambda x.t \to \lambda x.t'}\ (I) \qquad \frac{t \to t'}{ta \to t'a}\ (II) \qquad \frac{a \to a'}{ta \to ta'}\ (III)$$

$$\frac{u \to u'}{(u,l,c) \to (u',l,c)}\ (IV) \qquad \frac{l \to l'}{(u,l,c) \to (u,l',c)}\ (V) \qquad \frac{c \to c'}{(u,l,c) \to (u,l,c')}\ (VI)$$

$$\frac{u \to u'}{u :: l \to u' :: l}\ (VII) \qquad \frac{l \to l'}{u :: l \to u :: l'}\ (VIII) \qquad \frac{v \to v'}{(x)v \to (x)v'}\ (IX)$$

Figure 7 Compatible closure.

and the reflexive-transitive closure of $\to_R$, respectively. If $R = R_1 \cup R_2$, $\to_R$ can be denoted $\to_{R_1R_2}$ (e.g. $\to_{\beta\pi}$). A *R-normal form* (or R-nf, for short) is an expression E such that $E \to_R E'$ for no E'. When existing, we write $\downarrow_R (E)$ to denote the unique R-nf of an expression E.

The following result will be important later, and closes the discussion of derived syntax.

▶ **Lemma 4** (Associativity of append).
1. $(t@a)@a' \to_\pi^= t@(a@a')$ *and* $(t@_x a)@_x a' \to_\pi^= t@_x(a@a')$.
2. $(a@a')@a'' \to_\pi^= a@(a'@a'')$.
3. $(c@a)@a' \to_\pi^= c@(a@a')$.

Proof. By simultaneous induction in t, a and c. Everything follows from definitions and IHs, except in the single case where π-steps are generated, which is this: suppose t is neither x nor xa with $x \notin a$. Then $(t@_x a)@_x a' = (ta)a' \to_\pi t(a@a') = t@_x(a@a')$. ◀

Cut-elimination and computational interpretation. Rules β and π define a cut-elimination procedure in $\lambda\mathbf{Jm}$, whose purpose is not to eliminate all cuts ta, but rather to reduce them to the form xa: as seen above, xa represents a left introduction, not a cut to be eliminated. Still, we refer to $\beta\pi$-nfs as *cut-free*. A cut ta is necessarily principal on the right premiss, so its elimination starts by analyzing the left premiss t. If t is not a variable, then either it is another cut (in which case the original cut ta is permutable to the left, and a π-redex), or it is a λ-abstraction (in which case the cut is principal in both premisses, and a β-redex). Rule π performs left permutation, while rule β performs the key step of cut-elimination, breaking the cut into two cuts with simpler cut-formulas. If any of these two cuts is permutable to the right, it is not formed, but rather eliminated immediately, and represented by a substitution.

In a μ-redex we find a continuation $c = (x)xa'$ with $x \notin a'$, which represents a derivation of the form found in the right figure of (2), where a formula is selected immediately after being "unselected". The redex itself is a sequence of two $Leftm$ inferences, with the first, represented by a', being coerced to a continuation c, before being used in the second $Leftm$ inference (u,l,c). In addition, xa' represents a left introduction with the principal formula being introduced linearly, due to the proviso $x \notin a'$. The construction $u' :: l'$ found in the *contractum* of rule μ represents a linear left introduction by alternative and more primitive means, dispensing with the temporary name x, and eliminating the described sequence of inferences.

We also refer to ta as a *generalised, multiary application* (or gm-application for short), and think of $\lambda\mathbf{Jm}$ as a λ-calculus with the *multiarity* and *generality* features. In ta, t is the function expression, a is its gm-argument. A gm-argument consists of a first ordinary argument u, a list l of further ordinary arguments l (the multiarity feature), and a "continuation" c, indicating where to substitute the result of passing the last argument (the generality feature).

This interpretation follows from the reduction rules β_1 and β_2. A π-redex is an iterated gm-application. Contrary to ordinary arguments in, say, the λ-calculus, gm-arguments can be appended and the function expression simplified - this is the effect of the π-reduction. In a μ-redex, the generality feature is being used just to "link" two lists of arguments. The effect of the μ-reduction is to append these two lists. In the sequel, these interpretation of π and μ will be specialized to a fragment of $\lambda\mathbf{Jm}$; there, it will become appropriate to call μ-nfs *flat* expressions, and to call μ-normalization *flattening*. We adopt such terminology for the entire $\lambda\mathbf{Jm}$. For instance, μ-nfs constitute a subsystem of $\lambda\mathbf{Jm}$ [8]; we call it the *flat subsystem*.

Properties. The meta-theory of $\lambda\mathbf{Jm}$ is well developed, we just recall the results we need below. Some proofs have to be adapted to cover the variant of π we employ here.

▶ **Theorem 5** (Confluence and SN). *In $\lambda\mathbf{Jm}$, $\beta\pi$- and $\beta\pi\mu$-reductions are confluent, and $\beta\pi\mu$-reduction is SN on typable expressions.*

Proof. The existing proofs are easily adapted. ◀

In isolation, μ-reduction is easily seen to be confluent and terminating [8, 9]. The μ-nf of an expression E, $\mu(E)$, is defined by recursion on E, with all clauses given homomorphicaly, except in the following case: if $\mu v = x(u', l', (y)v')$ and $x \notin u', l', v'$, then $\mu(t(u, l, (x)v)) = \mu t(\mu u, \mu l @ (u' :: l'), (y)v')$.

▶ **Lemma 6** (Preservation of cut-freeness by μ-reduction). *In $\lambda\mathbf{Jm}$, if t is a $\beta\pi$-nf and $t \to_\mu t'$, then t' is a $\beta\pi$-nf.*

Proof. Easy induction on $t \to_\mu t'$. ◀

This lemma says μ-reduction preserves cut-freeness. Conversely, neither β-reduction nor π-reduction preserve μ-normality. Given a reduction rule R, by *R'-reduction* we will mean R-reduction followed by reduction to μ-nf.

▶ **Theorem 7** (Preservation of reduction by μ). *In $\lambda\mathbf{Jm}$:*
1. *If $t \to_\beta t'$ then there exists t'' s.t. $\mu(t) \to_\beta t'' \to_\mu^* \mu(t')$.*
2. *If $t \to_\pi t'$ then there exists t'' s.t. $\mu(t) \to_\pi t'' \to_\mu^* \mu(t')$.*

Proof. Statement 1 of the previous theorem is already used in [8] (Lemma 5), while statement 2 is also present in [8] (Lemma 7), but only for the terms in the $\lambda\mathbf{J}$-subsystem. ◀

Subsystems. A $\lambda\mathbf{m}$-expression is a $\lambda\mathbf{Jm}$-expression where all gm-applications have the form $t(u, l, (x)x)$, a form which we abbreviate as $t(u, l)$ and call *multiary application*. Based on such expressions one defines a subsystem $\lambda\mathbf{m}$ of $\lambda\mathbf{Jm}$: the expressions are μ-nfs; they are closed for β; they are not closed for π, but we return to the subsystem by post-composition with μ-normalization. The reduction rules of $\lambda\mathbf{m}$ are given in Fig. 8. The $\lambda\mathbf{m}$-calculus is a variant of the λ-calculus, called the *multiary λ-calculus*, or $\lambda\mathbf{m}$-calculus, where functions are applied to non-empty lists of arguments. The rules β_i pass to the function the first argument, adjusting the remainder of the list, while rule π' appends lists of arguments. The $\lambda\mathbf{m}$ is also a variant of the $\overline{\lambda}$-calculus [12]. The normal forms of $\lambda\mathbf{m}$ are either x, $\lambda x.t$ or $x(u, l)$, which are a variant of the cut-free $\overline{\lambda}$-terms, and represent the cut-free LJT derivations. For this reason $\lambda\mathbf{m}$ is also the *focused* subsystem of $\lambda\mathbf{Jm}$.

A $\lambda\mathbf{J}$-expression is a $\lambda\mathbf{Jm}$-expression where all gm-applications have the form $t(u, [], (x)v)$ (hence, just one argument u), a form which we abbreviate as $t(u, (x)v)$ and call *generalized*

$$\begin{array}{lrcl}
(\beta_1) & (\lambda x.t)(u,[]) & \to & \mathbf{s}(u,x,t) \\
(\beta_2) & (\lambda x.t)(u,u'::l) & \to & \mathbf{s}(u,x,t)(u',l) \\
(\pi') & t(u,l)(u',l') & \to & t(u,l@(u'::l'))
\end{array}$$

Figure 8 The reduction rules of the multiary λ-calculus $\lambda\mathbf{m}$.

application. Such expressions define the *unary* subsystem $\lambda\mathbf{J}$ of $\lambda\mathbf{Jm}$, as they are closed for β_1 and π. This is a copy, inside $\lambda\mathbf{Jm}$ [9], of natural deduction with general elimination rules [23], or rather its presentation as the typed λ-calculus ΛJ [13]. Conversely, $\lambda\mathbf{Jm}$ is a generalization of $\lambda\mathbf{J}$ obtained by allowing the left-introduction rule Lft, or constructor $u::l$. This is a small difference with numerous consequences: reduction rules β and π of ΛJ have to be taken in a multiary form, and two new reduction rules, β_2 and μ, appear; lists are not restricted to [], so the syntactic class of lists, as well as the third form of sequents $\Gamma;B\vdash l:C$, are not degenerate. So $\lambda\mathbf{Jm}$ is a system that goes slightly but decisively beyond natural deduction.

2.2 Why $\lambda\mathbf{Jm}$?

We choose to base on $\lambda\mathbf{Jm}$ our study of permutability in the sequent calculus. Before we proceed, we would like to justify our choice. The justification has two parts. First, we give a fresh explanation of the place of $\lambda\mathbf{Jm}$ among possible formulations of the sequent calculus, trying to dissipate some misunderstandings. Second, we explain our methodology in the study of permutability, and why $\lambda\mathbf{Jm}$ is an adequate tool for that methodology.

Understanding $\lambda\mathbf{Jm}$. Given that $\lambda\mathbf{Jm}$ captures several known systems as subsystems, one might have the impression that $\lambda\mathbf{Jm}$ is some *ad hoc* gluing. Of course we think otherwise, and would like to argue that $\lambda\mathbf{Jm}$ is rather a standard and important fragment of sequent calculus. Actually, this has been argued technically elsewhere [7], but the sceptical reader may object against the formulations of the sequent calculus with which $\lambda\mathbf{Jm}$ is compared in *op. cit.* So, here we formulate *ordinary* sequent calculus as a λ-calculus named $\lambda\mathbf{LJ}$, and show what fragment of this calculus $\lambda\mathbf{Jm}$ corresponds to.

The proof expressions of $\lambda\mathbf{LJ}$ are given by the following grammar:

$$\begin{array}{rrcl}
(LJ\text{-proof terms}) & t,u,v & ::= & x \mid \lambda x.t \mid x\hat{}(u;c) \mid tc \\
(LJ\text{-continuations}) & c & ::= & (x)v
\end{array}$$

The various term forms represent, respectively, the inference rules axiom, right introduction, left introduction, and cut; the continuation $(x)v$ represents a selection. Separating the class of continuations is convenient, as they are used twice in the grammar of terms. The formulation of the system as a typing system is quite obvious, here are most of the rules:

$$\frac{\Gamma\vdash u:A \quad \Gamma|B\vdash c:C}{\Gamma\vdash x\hat{}(u;c):C}\,((x:A\supset B)\in\Gamma) \qquad \frac{\Gamma\vdash t:A \quad \Gamma|A\vdash c:B}{\Gamma\vdash tc:B} \qquad \frac{x:B,\Gamma\vdash v:C}{\Gamma|B\vdash (x)v:C}$$

We will specify a subset of the set of LJ-terms, whose elements are called *Jm-terms*, by imposing two restrictions. The first restriction is that no Jm-term has the third form, corresponding to a left introduction. One reason for this is that we want a term to be either a value (variable or abstraction) or a single other form: having to sacrifice either left introduction or cut, there is no doubt the first form is the chosen to be sacrificed, since cuts represent computation, and can mimic left introductions.

This first restriction on terms determines three subsets of the set of LJ-continuations: (i) *Jm-continuations* $(x)v$, where v is a Jm-term; (ii) LJ-continuations of the form $(x)x\hat{}(u;c')$, to be called *Jm-arguments*, where $x \notin u, c'$, and u is a Jm-term, and c' is to be specified soon; (iii) the union of these two subsets, to be ranged over by k, whose elements are to be called *Jm-contexts*. Notice a Jm-argument $(x)x\hat{}(u;c')$ is not a Jm-continuation, because $x\hat{}(u;c')$ is not a Jm-term.

We have to specify which class c' in Jm-arguments belongs to; and the same is true of Jm-cuts, terms of the form tc'' with t a Jm-term and c'' to be specified now. We impose (and this is the second restriction on terms) c'' to be a Jm-argument: this implies that a Jm-cut is right-principal and a generalized form of function application, and that the cut-formula is an implication; this also justifies the terminology "Jm-argument". As to c': (i) imposing it to be a Jm-argument is not an option, otherwise the inductive definition of Jm-arguments would not have a base case; (ii) imposing it to be a Jm-continuation is not an option either, as otherwise Jm-terms would be isomorphic to ΛJ-terms, and the fragment would be equivalent to natural deduction; (iii) so we have to choose c' to be a Jm-context. Therefore, a Jm-argument is a LJ-continuation of the form $(x)x\hat{}(u;k)$, where $x \notin u, k$, and u is a Jm-term and k is a Jm-context. A Jm-argument $(x)x\hat{}(u;k)$ is abbreviated (u, k).

Summing up: the sets of Jm-terms, arguments, contexts, and continuations are given by

$$t, u, v \ ::= \ x \mid \lambda x.t \mid ta \qquad a \ ::= \ (u, k) \qquad k \ ::= \ a \,|\, c \qquad c \ ::= \ (x)v \tag{4}$$

We now see this syntax is a formulation of $\lambda\mathbf{Jm}$, let us call it the first formulation.

In fact, the syntax of $\lambda\mathbf{Jm}$ has many equivalent formulations. From (4) we can dispense with the class of arguments: cuts become $t(u, k)$ and contexts are given by $k \ ::= \ (u, k) \,|\, c$. This second formulation was used in [7]. Alternatively, from (4) we can dispense with the class of contexts: in this third formulation, which has never been used, arguments are given by $a \ ::= \ (u, a) \,|\, (u, c)$. In this paper we are using a fourth formulation: in (4), it is equivalent to take contexts as given by $k \ ::= \ (u, k) \,|\, c$; then arguments have the general form $(u_1, (u_2, (\cdots (u_m, c) \cdots)))$ for some $m \geq 1$; finally, we bring c to the surface of arguments, rearranging them as: $(u_1, (u_2 :: \cdots :: (u_m :: []) \cdots), c)$. To have c at the surface of arguments will be important precisely for the formulation of the process of permutative conversion[5].

So, $\lambda\mathbf{Jm}$ has several formulations, we are using one that suits better the purpose of this paper; but, independently of the several formulations, $\lambda\mathbf{Jm}$ has a special status, as it is a syntax that follows necessarily from $\lambda\mathbf{LJ}$ by imposing proof terms to be either values or cuts, and cuts to be restricted to a form of function application.

Methodology. Our methodology in the study of permutability in the sequent calculus is modular: we want to isolate and highlight the syntactic intricacies of permutability, avoiding to mix them with other issues that a wrong choice of system could bring. So, we need in the background a system as simple and as close to the ordinary λ-calculus as possible - but without falling into mere natural deduction or ΛJ (which would be undesirable in a study about the sequent calculus).

The system $\lambda\mathbf{Jm}$ has a number of characteristics appropriate to this aim (some of which were stressed by the reconstruction of $\lambda\mathbf{Jm}$ inside $\lambda\mathbf{LJ}$ given above). First, the logic we consider is the simplest one (intuitionistic implication as sole connective). Second, the cut=redex paradigm [12, 3] is not followed, so that variables in proof terms can be treated

[5] See equation (7) below.

as ordinary term variables, and substitution can be treated as ordinary term substitution [5]. Third, the primitive cut of the system is right-principal, hence a cut-formula is always an implication, hence the cut can be interpreted as some sort of function application; concomitantly, substitution is treated as a meta-operation (no explicit substitution), with the corresponding cut-rule treated as an admissible typing rule. Fourth, the immediate call of substitution(s) in the β-rules induces the call-by-name character of cut-elimination [3], which is the approach closest to the ordinary λ-calculus.

3 Permutation-freeness in the sequent calculus $\lambda\mathbf{Jm}$

In this section we study natural proofs, which are a generalization of normal proofs to the cut-full setting. They are introduced in the second subsection, after a technical subsection which develops the concept of pseudo-list. After natural proofs are proved to be closed for typing and reduction, they are given a computational interpretation in the third subsection, through the calculus $\lambda\mathbf{nm}$, which we prove to be isomorphic to the natural subsystem. In the final subsection we investigate the relationship between natural and focused proofs, paying particular attention to the search for normal proofs.

3.1 Pseudo-lists

The notion of x-normality goes back to [4] and was used in the context of $\lambda\mathbf{Jm}$ in [6]. Here we rename the notion as x-naturality, since we are not restricted to the cut-free setting. The concept of *pseudo-list* arises from the particular syntactic organization of $\lambda\mathbf{Jm}$ we employ in this paper, which includes the syntactic class of continuations c. In the remainder of the paper, pseudo-lists will be crucial in the study of naturality. In this subsection we see some of their basic properties, and their use in the analysis of continuations and gm-applications.

▶ **Definition 8** (Pseudo-lists). *x-natural* terms and arguments and *pseudo-lists* are defined simultaneously as follows:

- v is x-natural if $v = x$ or $v = xa$ and a is x-natural.
- a is x-natural if $a = (u, l, c)$ and $x \notin u, l, c$ and c is a pseudo-list.
- c is a pseudo-list if $c = (x)v$ with v x-natural.

Pseudo-lists are ranged over by L. We introduce the following abbreviations for pseudo-lists:

$$L \ ::= \ \mathbf{nil} \,|\, (u{+}l\,{+}L) \tag{5}$$

- $\mathbf{nil}$ abbreviates $(x)x$
- $(u{+}l\,{+}L)$ abbreviates $(x)x(u, l, c)$ if L abbreviates c and $x \notin u, l, c$.

▶ **Lemma 9** (Typing of pseudo-lists).

1. *In $\lambda\mathbf{Jm}$ a typing derivation of $\Gamma|C \vdash L : D$ ends with an application of the Select inference rule which has one of two forms:*
   - *either the inference selects the left-principal formula of an Axiom inference (with the whole derivation consisting of the two mentioned inferences);*
   - *or the inference ends a derivation of the form of the right figure in (2) - which entails that the Select inference selects a formula which had just been unselected, and the latter, being the distinguished formula in the l.h.s. of a sequent of kind (ii), is the principal formula of a $Leftm$ inference.*
2. *The typing rules for pseudo-lists in Fig. 9 are admissible typing rules of $\lambda\mathbf{Jm}$.*

$$\frac{}{\Gamma|A\vdash \mathbf{nil}:A}\,Axm \qquad \frac{\Gamma\vdash u:A \quad \Gamma;B\vdash l:C \quad \Gamma|C\vdash L:D}{\Gamma|A\supset B\vdash (u{+}l+L):D}\,multi-Lft$$

Figure 9 Typing rules for pseudo-lists.

$$\frac{u\to u'}{(u{+}l+L)\to(u'{+}l+L)}\,(a) \qquad \frac{l\to l'}{(u{+}l+L)\to(u{+}l'+L)}\,(b)$$
$$\frac{L\to L'}{(u{+}l+L)\to(u{+}l+L')}\,(c)$$

Figure 10 Closure rules for pseudo-lists.

Proof. 1. is by case analysis on L. The case Axm of 2. uses 1. and the case $multi-Lft$ of 2. uses admissibility of weakening for pseudo-lists. ◀

▶ **Lemma 10** (Derived substitution rules). *$\mathbf{s}(u,x,L)$ is a pseudo-list and satisfies $\mathbf{s}(u,x,\mathbf{nil})=\mathbf{nil}$ and $\mathbf{s}(u,x,(v{+}l+L))=(\mathbf{s}(u,x,v){+}\mathbf{s}(u,x,l)+\mathbf{s}(u,x,L))$.*

Proof. First one proves $\mathbf{s}(u,x,v)$ z-natural, for v z-natural, $z\neq x$ and $z\notin u$. Then, the statement of the lemma is proved by case analysis of L. ◀

▶ **Lemma 11** (Derived append rules).

1. *$L@a$ is a continuation and satisfies: $\mathbf{nil}@a=(z)za$, if $z\notin a$; and $(u{+}l+L)@a=(z)z(u,l,L@a)$, if $z\notin u,l,L,a$.*
2. *$L@c$ is a continuation and satisfies: $\mathbf{nil}@c=c$; and $(u{+}l+L)@c=(z)z(u,l,L@c)$, if $z\notin u,l,L,c$.*
3. *$L@L'$ is a pseudo-list and satisfies: $\mathbf{nil}@L'=L'$ and $(u{+}l+L)@L'=(u{+}l+(L@L'))$.*

Proof. 1. (resp. 2.) Immediate by definition of $c@a$ (resp. $c@c$) 3. Particular case of 2. ◀

Notice that $L@c$ is the continuation obtained by replacing $\mathbf{nil}$ by c in L.

▶ **Lemma 12** (Derived closure rules). *The closure rules for pseudo-lists in Fig. 10 are derived closure rules of $\to_R$, for any R.*

Proof. The derivations are easy. ◀

Pseudo-lists allow a useful representation of continuations:

▶ **Lemma 13** (Unique decomposition). *Every continuation c can be written in a unique way as $L@(x)v$ with $\neg mla(x,v)$.*

Proof. Existence of decomposition: we prove that, for all $t\in\lambda\mathbf{Jm}$, there are L and v such that $\neg mla(x,v)$ and $(x)t=L@(x)v$. The proof is by induction on t. Uniqueness of decomposition: we prove that, for all $t\in\lambda\mathbf{Jm}$, if $(z)t=L@(x)v=L'@(y)v'$, with $\neg mla(x,v)$ and $\neg mla(y,v')$, then $L=L'$ and $(x)v=(y)v'$. The proof is by induction on t. ◀

▶ **Definition 14.** When we write $\langle u,l,L,(x)v\rangle$ we mean $(u,l,L@(x)v)$ with $\neg mla(x,v)$.

In the argument $\langle u,l,L,(x)v\rangle$ the continuation is analyzed into its unique decomposition as given by Lemma 13. Of course we can write a gm-application as $t\langle u,l,L,(x)v\rangle$.

▶ **Corollary 15** (Pseudo-lists). *A continuation c is a pseudo-list iff $c=L@(x)x$.*

Proof. $L@(x)x = L$ is a pseudo-list. Conversely, suppose c is a pseudo-list and $c = L@(x)v$ with $\neg mla(x, v)$. The only case of v where the replacement of **nil** by $(x)v$ in L yields a pseudo-list is $v = x$. ◀

▶ **Lemma 16** (Associativity of append).
1. $(L@c)@c' = L@(c@c')$.
2. $(L@c)@a = L@(c@a)$.
3. $(L@a)@a' = L@(a@a')$. *(Compare with the third statement in Lemma 4.)*

Proof. Each by easy induction on L. Alternatively, the second (resp. third) statement follows from Lemma 2 and the first (resp. second) statement. ◀

Pseudo-lists can be used to give an handy alternative presentation of reduction rule π: $t\langle u, l, L, (x)v\rangle a \to t(u, l, L@(x)va)$.

Pseudo-lists also allow an alternative characterisation of the mapping μ for generalised multiary applications. For that, we need a flattening operation on pseudo-lists, denoted by $L^\flat$, and defined by: (i) $\mathbf{nil}^\flat := []$; (ii) $(u+l+L)^\flat := (u :: l)@L^\flat$. We also need μ extended to pseudo-lists homomorphically, that is: (i) $\mu(\mathbf{nil}) := \mathbf{nil}$; (ii) $\mu((u+l+L)) := (\mu(u)+\mu(l)+\mu(L))$.

▶ **Lemma 17.** $\mu(t\langle u, l, L, (x)v\rangle) = \mu t(\mu u, \mu l@(\mu L)^\flat, (x)\mu v)$.

Proof. By induction on L. The base case requires the fact that if $\neg mla(x, v)$, then also $\neg mla(x, \mu(v))$. The inductive case follows from the IH and uses associativity of the append operation on lists. ◀

3.2 Naturality

In this subsection we will introduce the concept of natural expression, and observe that this class of expressions is closed both for the reduction and the typing relations of $\lambda\mathbf{Jm}$, thus constituting the *natural subsystem* of $\lambda\mathbf{Jm}$.

▶ **Definition 18** (Natural and normal expressions). An expression of $\lambda\mathbf{Jm}$ is *natural* if all continuations occurring in it are pseudo-lists. An expression of $\lambda\mathbf{Jm}$ is *normal* if it is both natural and cut-free.[6]

A normal expression corresponds to a typing derivation where the inference rule *Select* is constrained to be of the two forms described in item 1 of Lemma 9.

Natural expressions are generated by the following grammar:

$$\begin{array}{rrcl} \text{(natural proof terms)} & t, u, v & ::= & x \mid \lambda x.t \mid ta \\ \text{(natural gm-arguments)} & a & ::= & (u, l, L) \\ \text{(natural lists)} & l & ::= & u :: l \mid [] \\ \text{(natural continuations)} & L & ::= & (x)v, \text{ with } v\ x\text{-natural} \end{array} \tag{6}$$

Notice that a natural continuation is a pseudo-list, but not conversely: in a natural continuation $(x)v$, v is not only x-natural, but also natural. A natural continuation is a natural pseudo-list.

When one coerces a natural argument $a = (u, l, L)$ to the natural continuation $(z)za$, with $z \notin a$, one obtains the natural pseudo-list $(u+l+L)$.

[6] Natural proofs were called "normal proofs" in [6].

$$\frac{L \to L'}{L@c \to L'@c}\ (d) \qquad \frac{v \to v' \quad \neg mla(x,v)}{L@(x)v \to L@(x)v'}\ (e)$$

Figure 11 Some rules for the restricted closure.

In view of Corollary 15, a natural application ta has the form $t\langle u, l, L, (x)x\rangle$; the last component is **nil** and so this representation does not give more information than $t(u, l, L)$.

The natural expressions of $\lambda\mathbf{Jm}$ are closed for typing in the following sense: in a typing derivation of a natural expression, every expression occurring in the derivation is natural itself. This is easily seen: the axioms of the typing system of $\lambda\mathbf{Jm}$ type natural expressions; in every other typing rule, the expressions in the premisses are subexpressions of the expression in the conclusion; and every subexpression of a natural expression is natural.

We now see the natural expressions of $\lambda\mathbf{Jm}$ are also closed for reduction. This is harder. The following lemma establishes that natural expressions are closed for the operations of substitution and append of gm-arguments.

▶ **Lemma 19.**
1. *If u, E are natural expressions, then $\mathbf{s}(u, x, E)$ is a natural expression.*
2. *If a, a' are natural gm-arguments, then $a@a'$ is also a natural gm-argument.*
3. *If l, l' are natural lists, then $l@l'$ is also a natural list.*
4. *If L, L' are natural continuations, then $L@L'$ is also a natural continuation.*

Proof. Part 1 is proved by simultaneous induction on $E = v, a, l, c$. Part 2 follows from the fact that, given a natural continuation L and a natural gm-argument a', $L@a'$ is a natural continuation - and this is easily proved by induction on L. Parts 3 and 4 are proved by straightforward induction on l and L respectively. ◀

▶ **Definition 20.** A relation ρ on expressions of $\lambda\mathbf{Jm}$ *preserves naturality* if $E\rho E'$ and E natural implies E' natural.

We will see that $\to_R$ preserves naturality. For the reduction rules R this is done directly.

▶ **Lemma 21.** *For each $R \in \{\beta_1, \beta_2, \pi, \mu\}$, R preserves naturality.*

Proof. The cases $R = \beta_1$ and $R = \beta_2$ (resp. $R = \pi$, $R = \mu$) follow from Part 1 (resp. Part 2, Part 3) of Lemma 19. ◀

For the compatible closure $\to_R$, preservation of naturality is proved in an easier way with the help of a restricted notion of closure.

▶ **Definition 22** (Restricted closure)**.** The *restricted closure* of a relation on expressions of $\lambda\mathbf{Jm}$ is defined by replacing closure rule (IX) in Fig. 7 by the rules (a), (b) and (c) in Fig. 10, and the rules (d) and (e) in Fig. 11. If R is a reduction rule, the closure of R under the restricted closure is denoted $\rightsquigarrow_R$.

▶ **Lemma 23.** *If R preserves naturality, so does $\rightsquigarrow_R$.*

Proof. Suppose R preserves naturality. We prove by simultaneous induction four statements. The first three are: if $E \rightsquigarrow_R E'$ and E natural then E' natural, for terms, arguments and lists. The last is: if $L \rightsquigarrow_R L'$ and $L@c$ natural then $L'@c$ natural. ◀

We now must relate $\to_R$ and $\rightsquigarrow_R$. We will see that the two closures coincide for $R \in \{\beta_1, \beta_2, \pi\}$, but there are small differences for $R = \mu$, which, nonetheless, allow to conclude preservation of naturality by $\to_\mu$ from preservation of naturality by $\rightsquigarrow_\mu$.

▶ **Lemma 24** (Admissible closure rules of $\to_R$). *Let $R \in \{\beta_1, \beta_2, \pi, \mu\}$. Closure rules (d) and (e) in Fig. 11 are admissible closure rules of $\to_R$.*

Proof. Case closure rule (d). One proves:
(i) if $t \to_R t'$, with t and t' x-natural, then $((x)t)@c \to_R ((x)t')@c$.
(ii) if $a \to_R a'$, with a and a' x-natural, then $((x)xa)@c \to_R ((x)xa')@c$.
(iii) if $c_1 \to_R c_1'$, with c_1 and c_1' pseudo-lists, then $c_1@c_2 \to_R c_1'@c_2$.
Case closure rule (e). In fact, one proves that the following are admissible closure rules of $\to_R$:

$$\frac{c \to c'}{t@c \to t@c'}\ (i) \qquad \frac{c_2 \to c_2'}{c_1@c_2 \to c_1@c_2'}\ (ii) \qquad \frac{v \to v'}{L@(x)v \to L@(x)v'}\ (iii)$$ ◀

Putting together the previous lemma and Lemma 10, we conclude $\rightsquigarrow_R \subseteq \to_R$ for the reduction rules of $\lambda\mathbf{Jm}$. For the converse inclusion, to address the case $R = \mu$ we will need the followig new μ-rule on pseudo-lists:

$$(\mu_2) \quad (u{+}l + (u'{+}l' + L)) \to (u{+}(l@(u' :: l')) + L) \ .$$

▶ **Lemma 25** (Admissible closure rules of $\rightsquigarrow_R$).
1. *For any reduction rule R, the following are admissible closure rules of $\rightsquigarrow_R$:*

$$\frac{L_1 \rightsquigarrow L_1'}{L_1@L_2 \rightsquigarrow L_1'@L_2}\ (i) \qquad \frac{L_2 \rightsquigarrow L_2'}{L_1@L_2 \rightsquigarrow L_1@L_2'}\ (ii) \qquad \frac{c \rightsquigarrow c'}{L@c \rightsquigarrow L@c'}\ (iii)$$

2. *Let $R \in \{\beta_1, \beta_2, \pi\}$. Closure rule (IX) of Fig. 7 is an admissible closure rule of $\rightsquigarrow_R$.*
3. *Let $R = \mu \cup \mu_2$. Closure rule (IX) of Fig. 7 is an admissible closure rule of $\rightsquigarrow_R$.*

Proof. The closure rule (iii) of part 1 is used in the proof of part 2. The new rule (μ_2) is needed to fix the base case of the inductive proof of part 3. ◀

▶ **Corollary 26.** *For each $R \in \{\beta_1, \beta_2, \pi, \mu\}$, $\to_R$ and $\rightsquigarrow_{R'}$ are the same relation, where $R' = R$ if $R \neq \mu$, and $R' = \mu \cup \mu_2$ otherwise.*

Proof. We had seen that $\rightsquigarrow_R \subseteq \to_R$. For $R \neq \mu$, part 2 of Lemma 25 completes the proof that $\to_R$ and $\rightsquigarrow_R$ are the same relation. In the case of μ, part 3 of Lemma 25 gives $\to_\mu \subseteq \rightsquigarrow_{R'}$, with $R' = \mu \cup \mu_2$. One still has to argue for $\rightsquigarrow_{R'} \subseteq \to_\mu$. Observe that μ_2 is a subset of the closure of $\rightsquigarrow_\mu$ under (IX). Hence $\rightsquigarrow_{R'}$ is a subset of the same closure. But such closure is a subset of $\to_\mu$, since $\rightsquigarrow_\mu \subseteq \to_\mu$ and $\to_\mu$ is closed under (IX). ◀

With this characterization of $\to_R$ in terms of the restricted closure, we can now show that the natural expressions of $\lambda\mathbf{Jm}$ are closed for reduction.

▶ **Theorem 27** (Preservation of naturality).
$\to_R$ preserves naturality, for each $R \in \{\beta_1, \beta_2, \pi, \mu\}$.

Proof. By the previous corollary $\to_R = \rightsquigarrow_{R'}$, where $R' = R$ if $R \neq \mu$, and $R' = \mu \cup \mu_2$ otherwise. By Lemma 21, each R preserves naturality. It is clear that also μ_2 preserves naturality. So, in each case, the reduction rule R' preserves naturality; by Lemma 23, so does $\rightsquigarrow_{R'}$. ◀

Given that the natural expressions are closed for typing and reduction, we define:

▶ **Definition 28** (Natural subsystem)**.** The *natural subsystem* of $\lambda\mathbf{Jm}$ is obtained by restriction to the natural expressions of the typing and reduction relations of $\lambda\mathbf{Jm}$. That is:

- given a natural term t, $\Gamma \vdash t : A$ in the natural subsystem if $\Gamma \vdash t : A$ in $\lambda\mathbf{Jm}$; and similarly for gm-arguments, lists, and continuations.
- given natural terms t, t', $t \to_R t'$ in the natural subsystem if $t \to_R t'$ in $\lambda\mathbf{Jm}$; and similarly for gm-arguments, lists, and continuations.

▶ **Corollary 29** (Confluence, SN, and uniqueness of normal form)**.** *In the natural subsystem, $\beta\pi$- and $\beta\pi\mu$-reductions are confluent, and $\beta\pi\mu$-reduction is SN on typable expressions. In particular, every typable natural expression has a unique $\beta\pi$-nf, which is a normal expression.*

Proof. By the same properties of $\lambda\mathbf{Jm}$ (Theorem 5). ◀

3.3 Computational interpretation

The natural subsystem was defined by restricting the typing and reduction relations of $\lambda\mathbf{Jm}$. We now give a direct, self-contained, equivalent definition of the natural subsystem. The advantage is that the alternative definition has a transparent computational interpretation.

The key idea is to handle the abbreviations for pseudo-lists as if they were first-class expressions. In the resulting system, named $\lambda\mathbf{nm}$, pseudo-lists L behave properly as lists of non-empty lists of ordinary arguments; and arguments (u, l, L) may be seen as (and coerced to) non-empty pseudo-lists $(u+l+L)$. If we call lists of lists *multi-lists*, $\lambda\mathbf{nm}$ is then a *multi-multiary* λ-calculus, in the sense of a λ-calculus where functions are applied to multi-lists of arguments. The reduction rules of $\lambda\mathbf{nm}$ will confirm this interpretation.

Definition of $\lambda\mathbf{nm}$. The expressions of $\lambda\mathbf{nm}$ are the natural expressions, given by grammar (6). It is easy to prove that the same expressions are generated if, in the grammar, the class L is generated by $L ::= \mathbf{nil} \,|\, (u+l+L)$. These are the abbreviations (in the meta-language) we adopted to denote pseudo-lists - recall (5). Now we define typing and reduction rules for the natural expressions, alternative to those of Def. 28. The idea is to treat these abbreviations as if they were object syntax, and handle them with the derived rules contained in Lemmas 9, 10, 11, and 12, together with reduction rules that can be proved to be derived rules as well. Since the new system $\lambda\mathbf{nm}$ is built with derived rules of the natural subsystem given by Def. 28, the former will be immediately "contained" in the latter. We will check that the two systems are actually isomorphic.

▶ **Definition 30** (Typing system of $\lambda\mathbf{nm}$)**.** The typing rules of $\lambda\mathbf{nm}$ are all the typing rules in Fig. 4 except *Select*, plus the typing rules in Fig. 9 (of course, in both cases with meta-variables t, a, u, l, c ranging over expressions of $\lambda\mathbf{nm}$).

Recall the four kinds of sequent of $\lambda\mathbf{Jm}$, displayed in (1). Observing the typing rules in Fig. 9 we conclude that, in $\lambda\mathbf{nm}$, sequents $\Gamma|C \vdash c : D$ of kind (iv) are such that D is a suffix of C; and sequents $\Gamma; A \supset B \vdash a : D$ of kind (ii) are such that D is a suffix of B.

The reduction rules of $\lambda\mathbf{nm}$ are given in Fig. 12. We let $\beta_1 := \beta_{11} \cup \beta_{12}$ and $\mu := \mu_1 \cup \mu_2$. Observe that reduction rule β_{12} can be derived as μ_1 followed by β_2. However, if we would omit β_{12}, the wanted 1-1 correspondence of reduction steps with the natural subsystem would be lost. The meta-operations used in the reduction rules of $\lambda\mathbf{nm}$ are as follows:

- $\mathbf{s}(u, x, E)$ denotes ordinary substitution on $\lambda\mathbf{nm}$ expression E, with $E = t, a, l, L$. In the case $E = L$, the operation is defined by the equations in Lemma 10.

$$\begin{array}{rrcl}
(\beta_{11}) & (\lambda x.t)(u,[],\mathbf{nil}) & \to & \mathbf{s}(u,x,t) \\
(\beta_{12}) & (\lambda x.t)(u,[],(u'{+}l+L)) & \to & \mathbf{s}(u,x,t)(u',l,L) \\
(\beta_{2}) & (\lambda x.t)(u,u'::l,L) & \to & \mathbf{s}(u,x,t)(u',l,L) \\
(\pi) & t(u,l,L)(u',l',L') & \to & t(u,l,L@(u'{+}l'+L')) \\
(\mu_1) & (u,l,(u'{+}l'+L)) & \to & (u,l@(u'::l'),L) \\
(\mu_2) & (u{+}l+(u'{+}l'+L)) & \to & (u{+}l@(u'::l')+L)
\end{array}$$

Figure 12 The reduction rules of the multi-multiary λ-calculus $\lambda\mathbf{nm}$.

- $L@L'$ denotes the append of two pseudo-lists of $\lambda\mathbf{nm}$ and is defined by the same equations as those in Lemma 11.
- $l@l'$ denotes the append of two lists of $\lambda\mathbf{nm}$ and is defined by the same equations as those in Definition 1.

▶ **Definition 31** (Compatible closure for $\lambda\mathbf{nm}$-expressions)**.** A *compatible* relation on $\lambda\mathbf{nm}$-expressions is one closed for the closure rules in Fig. 7 except (IX), plus the closure rules in Fig. 10 (with meta-variables ranging over expressions of $\lambda\mathbf{nm}$). The *compatible closure* of a rule R of $\lambda\mathbf{nm}$, denoted $\to_R$, is the smallest compatible relation containing R.

Having completed the definition of the system $\lambda\mathbf{nm}$, we pause to observe its **computational interpretation**: *$\lambda\mathbf{nm}$ is a lambda-calculus where functions are applied to non-empty multi-lists, where a multi-list is a list of non-empty lists of arguments. The reduction rules have a transparent meaning in terms of these multi-lists: β-rules pass to the applied function the first element of the first list of arguments in the multi-list, while π and μ append and flatten multi-lists of arguments, respectively.*

▶ **Proposition 32** (Natural subsystem $\cong$ $\lambda\mathbf{nm}$)**.**

1. *$\Gamma\vdash t:A$ in the natural subsystem iff $\Gamma\vdash t:A$ in $\lambda\mathbf{nm}$. Similarly for gm-arguments, lists and continuations.*
2. *Let $R\in\{\beta_1,\beta_2,\pi,\mu\}$. $t\to_R t'$ in the natural subsystem iff $t\to_R t'$ in $\lambda\mathbf{nm}$. Similarly for gm-arguments, lists and continuations.*

Proof. 1. There are four "if" statements (one for each $E=t,a,l,L$) proved by simultaneous induction. The only interesting point is that the typing rules in Fig. 9 are derived typing rules of the natural subsystem. Similarly, there are four "only if" statements, proved by simultaneous induction.

2. The "if" statement for $E=t$ is proved together with similar statements for $E=a,l,L$, by simultaneous induction on $E\to_R E'$ in $\lambda\mathbf{nm}$. The "only if" statement for $E=t$ is proved together with similar statements for $E=a,l,L$, by simultaneous induction on $E\to_R E'$ in the natural subsystem. ◀

The natural subsystem of $\lambda\mathbf{Jm}$ benefits largely from this isomorphism. The presentation of its typing and reduction rules as in Def. 30 and Fig. 12 is much more perspicuous than through Def. 28: think of the sequent invariants noted after Def. 30, or the computational interpretation of $\lambda\mathbf{nm}$, that the natural subsystem inherits. The isomorphism lets us see that the natural subsystem corresponds to a multi-multiary λ-calculus, where the generality feature is reduced to a mechanism to form lists of lists of arguments for functional application.

3.4 Naturality and focusedness

Natural proofs are a generalization of focused proofs (in the sense of LJT). We will show this both for the computation-as-cut-elimination and computation-as-proof-search paradigms. In the former case, we show the relationship between the calculi $\lambda\mathbf{nm}$ and $\lambda\mathbf{m}$; in the latter, we explain how normal(=natural and cut-free) proofs can be searched by a procedure that is a relaxed form of focusing.

Recall that the map μ calculates the unique μ-nf of a $\lambda\mathbf{Jm}$ expression. Its restriction to $\lambda\mathbf{nm}$ has a recursive description in which the single interesting clause is given by $\mu(t(u, l, L)) = \mu t(\mu u, \mu l@(\mu L)^\flat)$, thanks to Lemma 17. So we see μ maps natural proofs to focused proofs; the case $t = x$ also gives that μ maps normal proofs to cut-free, focused proofs[7]. The latter is also a consequence of the fact that μ-reduction in $\lambda\mathbf{nm}$ preserves cut-freeness, a particular case of Lemma 6.

▶ **Theorem 33** (Preservation of reduction on natural proofs by μ).
1. *If $t \to_\beta t'$ in $\lambda\mathbf{nm}$ then $\mu(t) \to_\beta \mu(t')$ in $\lambda\mathbf{m}$.*
2. *If $t \to_\pi t'$ in $\lambda\mathbf{nm}$ then $\mu(t) \to_{\pi'} \mu(t')$ in $\lambda\mathbf{m}$..*

Proof. By Theorem 7 and the following two facts: (i) $\lambda\mathbf{m}$ is closed for β-reduction; (ii) $\to_{\pi'}$ in $\lambda\mathbf{m}$ is the same as $\to_\pi$ followed by μ-reduction to μ-nf in $\lambda\mathbf{nm}$. ◀

This theorem says μ is a morphism between the natural and the focused subsystems of $\lambda\mathbf{Jm}$.

In Fig. 13 we recapitulate the typing system for normal expressions[8]. The rule $Leftm$ has been renamed to $outer - multi - Lft$ to reflect its resemblance with $multi - Lft$, which in turn has been renamed to $inner - multi - Lft$. Cut inferences are restricted to the $Unselect$ form, which behaves as a focusing inference.

We will now see in detail how the good properties enjoyed by focused proof systems (invertibility, completeness w.r.t. provability, disciplined proof search) apply to the proof system for normal proofs.

One observation used several times below is that *weakening* is an admissible rule for the various forms of sequents in the proof system for normal proofs. Let us look first into invertibility of rules $multi - Lft$, which is not immediate because of the foreign formula C.

▶ **Proposition 34** (Invertibility of *multi-Lft* rules). *If $\Gamma;A \supset B \vdash a : D$ or $\Gamma|A \supset B \vdash L : D$ and D is an atomic formula, then there exists u_0 s.t. $\Gamma \vdash u_0 : A$, and for all C suffix of B, there exist l_0, L_0 s.t. $\Gamma;B \vdash l_0 : C$, and $\Gamma|C \vdash L_0 : D$.*

Proof. Case $\Gamma;A \supset B \vdash a : D$ with $a = (u_0, l_0, L_0)$, we must have $\Gamma \vdash u_0 : A$, and, for some C_0, $\Gamma;B \vdash l_0 : C_0$, and $\Gamma|C_0 \vdash L_0 : D$. The result follows then with the help of the following *suffix lemma*: for D the atomic suffix of B, if, for some C_0, l_0, L_0, $\Gamma;B \vdash l_0 : C_0$ and $\Gamma|C_0 \vdash L_0 : D$, then, for all C suffix of B there exist l, L s.t. $\Gamma;B \vdash l : C$ and $\Gamma|C \vdash L : D$. (This lemma follows by induction on B.) Case $\Gamma|A \supset B \vdash L : D$, as D is atomic, the derivation cannot be solely an axiom Axm. So, we must have $L = (u_0 + l_0 + L_0)$, and proceed as in the previous case. ◀

Invertibility of the *multi-Lft* rules is guaranteed only if the r.h.s. formula of the conclusion is atomic, but this is in line with LJT, where typically proof search imposes atomic r.h.s. in

[7] Note that mapping μ restricted to the class of *unary* normal expressions is a 1-1 correspondence with cut-free, focused proofs (which are the cut-free LJT proofs, or the cut-free $\overline{\lambda}$-terms, as already shown in [4] - but there the name used for the mapping is $\overline{\varphi}$).

[8] The division into groups of rules will be useful later.

$$\boxed{\text{I}} \qquad \frac{}{x\!:\!D,\Gamma\vdash x\!:\!D}\ \mathit{Axiom} \qquad \frac{x\!:\!A,\Gamma\vdash t\!:\!B}{\Gamma\vdash\lambda x.t\!:\!A\supset B}\ \mathit{Right}$$

$$\boxed{\text{II}} \qquad \frac{\Gamma;A\supset B\vdash a\!:\!D}{\Gamma\vdash xa\!:\!D}\ \mathit{Unselect}\ ((x\!:\!A\supset B)\in\Gamma)$$

$$\boxed{\text{III}} \qquad \frac{\Gamma\vdash u\!:\!A \quad \Gamma;B\vdash l\!:\!C \quad \Gamma|C\vdash L\!:\!D}{\Gamma;A\supset B\vdash(u,l,L)\!:\!D}\ \mathit{Outer\text{-}multi\text{-}Lft}$$

$$\frac{}{\Gamma|D\vdash\mathbf{nil}\!:\!D}\ \mathit{Axm} \qquad \frac{\Gamma\vdash u\!:\!A \quad \Gamma;B\vdash l\!:\!C \quad \Gamma|C\vdash L\!:\!D}{\Gamma|A\supset B\vdash(u{+}l+L)\!:\!D}\ \mathit{Inner\text{-}multi\text{-}Lft}$$

$$\boxed{\text{IV}} \qquad \frac{}{\Gamma;C\vdash[\,]\!:\!C}\ \mathit{Ax} \qquad \frac{\Gamma\vdash u\!:\!A \quad \Gamma;B\vdash l\!:\!C}{\Gamma;A\supset B\vdash u::l\!:\!C}\ \mathit{Lft}$$

Figure 13 Proof system for normal proofs (in the atomized system, rules $Axiom$ and $Unselect$ are restricted to atomic D).

the conclusion of Lft inferences (see e.g. [2] for a system corresponding to LJT with this atomic restriction). Next, we consider a restriction of the proof system for normal proofs, for which invertibility of the *multi-Lft* rules holds and a focused proof search discipline can be followed.

▶ **Definition 35** (Atomized normal system)**.** The *atomized system for normal proofs* is the system obtained from the proof system for normal proofs in Fig. 13 by imposing that at the rules $Axiom$ and $Unselect$ the r.h.s. formula is atomic. We denote these restricted versions of the rules by $Axiom_{atom}$ and $Unselect_{atom}$. We write $\vdash_{atom}$, instead of $\vdash$, to mean that a sequent has a derivation in the atomized system.

Before we describe proof search in the atomized system, we will show that nothing is lost in the atomized system regarding provability of sequents $\Gamma\vdash t\!:\!A$.

▶ **Definition 36** (η-expansion)**.** The η-expansion rules for normal expressions are

$$y\to\lambda x.y(x,[\,],\mathbf{nil}) \qquad y(u,l,L)\to\lambda x.y(u,l,\eta exp_x L)$$

where $x\neq y$ and $x\notin u,l,L$, and $\eta exp_x L$ is defined by: $\eta exp_x\mathbf{nil}=(x+[\,]+\mathbf{nil})$ and $\eta exp_x(u{+}l+L)=(u{+}l+\eta exp_x L)$. The compatible closure of these rules is denoted $\to_{\eta exp}$.

▶ **Lemma 37** (Admissibility of $Axiom$ and $Unselect$)**.** *For any A:*

1. *There exists t s.t. $x\to^*_{\eta exp} t$ and $x:A,\Gamma\vdash_{atom} t\!:\!A$.*
2. *If $\Gamma;B\supset C\vdash_{atom} a\!:\!A$ and $x:B\supset C\in\Gamma$, there exists t s.t. $xa\to^*_{\eta exp} t$ and $\Gamma\vdash_{atom} t\!:\!A$.*
3. *If $\Gamma|C\vdash_{atom} L\!:\!A\supset B$, there exists L' s.t. $\eta exp_y L\to^*_{\eta exp} L'$ and $y:A,\Gamma|C\vdash_{atom} L'\!:\!B$.*

Proof. Proved simultaneously by induction on A. ◀

▶ **Theorem 38** (Completeness of the atomized system)**.** *If $\Gamma\vdash t\!:\!A$, then there exists t' s.t. $t\to^*_{\eta exp} t'$ and $\Gamma\vdash_{atom} t'\!:\!A$. Similarly for gm-arguments, lists, and continuations.*

Proof. The proof of the four statements is done by simultaneous induction. All cases follow routinely, except for the cases $t = x$ and $t = xa$. The case $t = x$ follows by 1. of the lemma before, whereas the case $t = xa$ is by IH and 2. of the lemma before. ◀

Proof search in the atomized system. Proof search in the atomized system will find a derivation of $\Gamma \vdash t : A$, if one exists, following a disciplined alternation between *asynchronous* and *synchronous* phases which we now explain. In this explanation, *bottom-up* application of inference rules is meant; we also refer to the groups of rules in Fig. 13.

The asynchronous phase searches for proofs of sequents $\Gamma \vdash t : A$ by applying rules of group I. Rule *Right* decomposes implications until an atomic formula is reached. If this atom is in the l.h.s. of the sequent, rule $Axiom_{atom}$ ends the search with success. Otherwise, the only rule in group II picks a formula from the context, and a synchronous phase starts.

The synchronous phase searches for proofs of sequents $\Gamma; A \supset B \vdash a : D$ or $\Gamma|C \vdash L : D$, by applying rules of group III. This phase consists of a chain of $multi - Lft$ inferences, starting with an $Outer - multi - Lft$ inference, continuing with $n \geq 0$ $Inner - multi - Lft$ inferences, and ending with an application of Axm when successful.

Each application of a $multi - Lft$ inference (either an outer or an inner one) transforms the distinguished formula $A \supset B$ in the l.h.s. of the sequent to be proved into a formula C, which is not necessarily the immediate positive subformula B, but rather some suffix of B which has to be chosen (provability is not affected by this choice - recall Proposition 34), triggering a subprocess of proof search for $\Gamma \vdash u : A$, and a subsidiary search for $\Gamma; B \vdash l : C$. The search for $\Gamma; B \vdash l : C$ is done by *focusing* on B, through application of rules in group IV.

So focusing is a subsidiary process of the synchronous phase. In fact, we may say the chain of $n + 1$ $multi - Lft$ inferences that constitutes the synchronous phase that started with sequent $\Gamma; A \supset B \vdash a : D$ breaks into a succession of $n + 1$ focusing proofs (that can be conducted independently and in parallel) what in a focused system like LJT or $\lambda\mathbf{m}$ would rather be a single focusing proof leading from $A \supset B$ to D.[9]

4 Permutability in the sequent calculus $\lambda\mathrm{Jm}$

In this section we study permutative conversions in $\lambda\mathbf{Jm}$ such that the proofs irreducible by such conversions are the natural proofs studied in the previous section. This justifies our description of natural proofs as "permutation-free". Our approach to permutative conversions is the simplest one: we introduce a map γ that translates any $\lambda\mathbf{Jm}$ proof into a natural one (and leaves natural proofs invariant); in addition, it maps cut-free proofs to normal ones, as required [4]. Map γ, studied in the second subsection, is defined in terms of a special substitution operator over natural proofs, which is introduced in the first subsection. Such an operator is an essential ingredient of the computational process involved in γ. In the third subsection, we prove that permutative conversion to natural form commutes with cut-elimination. Hence, the two immediate senses for the concept of *normalization*, either permutative conversion of cut-free proofs to normal form, or cut-elimination in the natural subsystem, are coherent and have a common generalization to $\lambda\mathbf{Jm}$. In the final fourth subsection we systematize the internal structure of $\lambda\mathbf{Jm}$ with the help of γ.

[9] This has nothing to do with multifocusing, where the focus contains simultaneously several formulas.

4.1 Special substitution

The special substitution operation on $\lambda\mathbf{nm}$ that we will introduce now is the key element in the permutative conversion of $\lambda\mathbf{Jm}$ expressions to natural form.

▶ **Definition 39** (Special substitution of $\lambda\mathbf{nm}$). Given $t \in \lambda\mathbf{nm}$, we define $\mathbb{S}(t,x,u)$, $\mathbb{S}(t,x,a)$, $\mathbb{S}(t,x,l)$ and $\mathbb{S}(t,x,L)$ (for $u,a,l,L \in \lambda\mathbf{nm}$) by simultaneous recursion:

$$\begin{array}{ll}
\mathbb{S}(t,x,x) = t & \mathbb{S}(t,x,(u,l,L)) = (\mathbb{S}(t,x,u),\mathbb{S}(t,x,l),\mathbb{S}(t,x,L)) \\
\mathbb{S}(t,x,y) = y \ \text{ if } x \neq y & \mathbb{S}(t,x,[]) = [] \\
\mathbb{S}(t,x,\lambda y.v) = \lambda y.\mathbb{S}(t,x,v) & \mathbb{S}(t,x,(u::l)) = \mathbb{S}(t,x,u)::\mathbb{S}(t,x,l) \\
\mathbb{S}(t,x,xa) = t@\mathbb{S}(t,x,a) & \mathbb{S}(t,x,\mathbf{nil}) = \mathbf{nil} \\
\mathbb{S}(t,x,t'a) = \mathbb{S}(t,x,t')\mathbb{S}(t,x,a) \ \text{ if } t' \neq x & \mathbb{S}(t,x,(u{+}l{+}L)) = (\mathbb{S}(t,x,u){+}\mathbb{S}(t,x,l)\,{+}\mathbb{S}(t,x,L))
\end{array}$$

The difference to ordinary substitution is seen in the fourth clause, with $t@\mathbb{S}(t,x,a)$ instead of $t\mathbb{S}(t,x,a)$. The precise relation between ordinary and special substitution is:

▶ **Lemma 40** (Subst. vs special subst.). $\mathbf{s}(u,x,E) \to_\pi^* \mathbb{S}(u,x,E)$*, for all* $E \in \lambda\mathbf{nm}$.

Proof. By simultaneous induction on $E = v,a,l,L$. ◀

▶ **Lemma 41** (Typing of special substitution). *The following rules are admissible in* $\lambda\mathbf{nm}$.

$$\frac{\Gamma\vdash t:A \quad x:A,\Gamma\vdash u:B}{\Gamma\vdash\mathbb{S}(t,x,u):B} \qquad \frac{\Gamma\vdash t:A \quad x:A,\Gamma;B\supset C\vdash a:D}{\Gamma;B\supset C\vdash\mathbb{S}(t,x,a):D}$$

$$\frac{\Gamma\vdash t:A \quad x:A,\Gamma;B\supset C\vdash l:D}{\Gamma;B\supset C\vdash\mathbb{S}(t,x,l):D} \qquad \frac{\Gamma\vdash t:A \quad x:A,\Gamma|B\supset C\vdash L:D}{\Gamma|B\supset C\vdash\mathbb{S}(t,x,L):D}$$

Proof. By simultaneous induction on u,a,l,L. The case $u = xa$ uses first the IH to type $\mathbb{S}(t,x,a)$, and then uses admissibility of the first rule of Fig. 5 to type $t@\mathbb{S}(t,x,a)$. ◀

From this proof we extract the *operation on typing/logical derivation of* $\lambda\mathbf{nm}$ *whose term representation is* $\mathbb{S}(t,x,u)$, performing the elimination of the cut which types this substitution. In general, such operation performs the permutation to the right as long as the repetition of the cut formula permits, supplemented in the exceptional case $u = xa$ by the operation associated with the operation $t@a'$ (recall discussion after Lemma 3).

▶ **Lemma 42** (Substitution Lemma). *Let* $t,u \in \lambda\mathbf{nm}$*,* $x \neq y$*, and* $y \notin u$*. For all* $E \in \lambda\mathbf{nm}$*:*
1. $\mathbf{s}(u,x,\mathbb{S}(t,y,E)) \to_\pi^* \mathbb{S}(\mathbf{s}(u,x,t),y,\mathbf{s}(u,x,E))$*;*
2. $\mathbb{S}(u,x,\mathbb{S}(t,y,E)) = \mathbb{S}(\mathbb{S}(u,x,t),y,\mathbb{S}(u,x,E))$*;*
3. $\mathbf{s}(\mathbb{S}(u,x,t),y,\mathbb{S}(u,x,E)) \to_\pi^* \mathbb{S}(u,x,\mathbf{s}(t,y,E))$.

Proof. By simultaneous induction on $E = v,a,l,L$. ◀

4.2 Permutative conversion to natural form

Now we introduce the map that realises conversion to natural form, and, in particular, show that it preserves typing, leaves invariant natural expressions, and preserves reduction.

▶ **Definition 43** (Conversion to natural form map). For $t,a,c,l \in \lambda\mathbf{Jm}$ and $t' \in \lambda\mathbf{nm}$, we define $\gamma(t)$, $\gamma(t',a)$, $\gamma(t',c)$, and $\gamma(l)$, by simultaneous recursion on t, a, c, and l:

$$\begin{array}{rclrcl}
\gamma(x) & = & x & \gamma(t',(u,l,c)) & = & \gamma(t'(\gamma u,\gamma l,\mathbf{nil}),c) \\
\gamma(\lambda x.t) & = & \lambda x.\gamma(t) & \gamma(t',(x)v) & = & \mathbb{S}(t',x,\gamma v) \\
\gamma(ta) & = & \gamma(\gamma t,a) & \gamma([]) & = & [] \\
 & & & \gamma(u::l) & = & \gamma(u)::\gamma(l)
\end{array}$$

This is summarized in the following equation:

$$\gamma(t(u,l,(x)v)) = \mathbb{S}(\gamma t(\gamma u, \gamma l, \mathbf{nil}), x, \gamma v) \tag{7}$$

▶ **Proposition 44** (Preservation of typing by γ). *The following typing rules are admissible (where $\vdash$ and $\vdash'$ denote derivability in $\lambda\mathbf{Jm}$ and $\lambda\mathbf{nm}$ resp., and so $t, a, l, c \in \lambda\mathbf{Jm}$ and $t' \in \lambda\mathbf{nm}$).*

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash' \gamma t : A} \qquad \frac{\Gamma \vdash' t' : A \supset B \quad \Gamma; A \supset B \vdash a : C}{\Gamma \vdash' \gamma(t', a) : C} \qquad \frac{\Gamma; A \vdash l : B}{\Gamma; A \vdash' \gamma l : B} \qquad \frac{\Gamma \vdash' t' : A \qquad \Gamma | A \vdash c : B}{\Gamma \vdash' \gamma(t', c) : B}$$

Proof. By simultaneous induction on t, a, l, c. The case $a = (u', l', c')$ needs to first show $\gamma(t')(\gamma(u'), \gamma(l'), \mathbf{nil})$ is typable, using the IH relative to u' and l', and then use the IH relative to c'. The case $c = (x)v$ needs the typing rule of the special substitution on terms (Lemma 41). The other cases are routine. ◀

From this proof we extract the *operation on typing/logical derivation of* $\lambda\mathbf{Jm}$ *associated with* γ: it is an innermost-outermost application of the operation associated with transformation (7), and the latter, in turn, is the operation on derivations of $\lambda\mathbf{nm}$ associated with special substitution (see discussion after Lemma 41), applied after the transformation of the given sub-derivations (represented by t, u, l, v).

γ is extended to pseudo-lists:

$$\gamma(\mathbf{nil}) := \mathbf{nil} \qquad\qquad \gamma((u{+}l{+}L)) := (\gamma(u){+}\gamma(l){+}\gamma(L)) \ . \tag{8}$$

▶ **Proposition 45** (Invariance of natural expressions under γ). *For $E = t, l, L \in \lambda\mathbf{nm}$, $\gamma E = E$.*

Proof. By simultaneous induction on t, l, L. The interesting case is where $t = t'(u', l', L')$, which follows by IH and the fact $\gamma(t_0(u, l_0, L_0@(x)v)) = \mathbb{S}(\gamma t_0(\gamma u, \gamma l_0, \gamma L_0), x, \gamma v)$, for any $t_0, u, l_0, L_0, v \in \lambda\mathbf{Jm}$, which, in turn, uses the following auxiliary result: given $t', u', l', L' \in \lambda\mathbf{nm}$, $\gamma(t'(u', l', L'), L@(x)v) = \mathbb{S}(t'(u', l', L'@\gamma L), x, \gamma v)$ (proved by induction on L). ◀

This means that, if we want to see γ as defining the naive, long-step reduction rule $E \to \gamma(E)$, we have to require the redex E not to be normal, and so the normal expressions are the irreducible expressions for this rule.

The following result says γ sends cut-free proofs to normal proofs.

▶ **Lemma 46** (γ preserves cut-freeness). *If t is a $\beta\pi$-nf of $\lambda\mathbf{Jm}$, $\gamma(t)$ is a $\beta\pi$-nf of $\lambda\mathbf{nm}$.*

Proof. Proved together with analogue statements for gm-arguments, lists and pseudo-lists. The case $t = xa$ requires an auxiliary result about preservation of $\beta\pi$-nfs by substitutions of the form $\mathbb{S}(x(u, l, L), y, t)$. ◀

▶ **Theorem 47** (Preservation of reduction by conversion to natural form).

1. *If $t \to_\beta t'$ in $\lambda\mathbf{Jm}$ then $\gamma(t) =_{\beta\pi} \gamma(t')$ in $\lambda\mathbf{nm}$.*
2. *If $t \to_R t'$ in $\lambda\mathbf{Jm}$ then $\gamma(t) \to_R^* \gamma(t')$ in $\lambda\mathbf{nm}$, for $R \in \{\pi, \mu\}$.*

Proof. We use the inductive characterisation of reduction in $\lambda\mathbf{Jm}$ given by Corollary 26. Notice $=_{\beta\pi}$ in statement 1. ◀

4.3 Normalisation

We have so far two processes of obtaining a normal(=natural and cut-free) proof: either by cut-elimination on a natural proof (as natural proofs are closed for cut-elimination, recall Theorem 27), or by permutative conversion of a cut-free proof (as γ preserves cut-freeness, recall Lemma 46). We may call such processes *normalization* processes. The question is whether there is a normalization procedure defined on arbitrary $\lambda\mathbf{Jm}$ proofs which generalizes both these two processes. The answer is positive, due to the following result.

▶ **Theorem 48** (Commutation between cut-elim. and conversion to natural form). *For all typable $t \in \lambda\mathbf{Jm}$, $\gamma(\downarrow_{\beta\pi}(t)) = \downarrow_{\beta\pi}(\gamma(t))$.*

Proof. Firstly observe that all the required nfs exist since the starting terms are typable and the map γ preserves typing. By Theorem 47, $\gamma(t) =_{\beta\pi} \gamma(\downarrow_{\beta\pi}(t))$. By Lemma 46, $\gamma(\downarrow_{\beta\pi}(t))$ is a $\beta\pi$-nf. Hence, by confluence of $\to_{\beta\pi}$ in $\lambda\mathbf{nm}$, $\gamma(t) \to^*_{\beta\pi} \gamma(\downarrow_{\beta\pi}(t))$. ◀

▶ **Definition 49** (Normalisation map). $\rho(E) := \gamma(\downarrow_{\beta\pi}(E))$, for all typed $\lambda\mathbf{Jm}$ expression E.

If E is cut-free, then $\rho(E) = \gamma(E)$, which is the permutative conversion of E; if E is natural, then $\rho(E) = \downarrow_{\beta\pi}(\gamma(E)) = \downarrow_{\beta\pi}(E)$, which is the result of cut elimination from E in $\lambda\mathbf{nm}$.

4.4 The taming of "bureaucracy"

The permutative conversion γ and the reduction process μ are the "bureaucratic" processes of $\lambda\mathbf{Jm}$, as opposed to $\beta\pi$-reduction, which represents cut-elimination. We are now in position to converge to a systematic picture of the internal organization of $\lambda\mathbf{Jm}$, fulfilling the promise made in the introduction of linking Figs. 3 and 2. The final result we want to achieve is in Fig. 14, where:

- Cut-free classes are below the line (a).
- Unary fragments (isomorphic to fragments of natural deduction with generalized elimination rule) are to the right of line (b).
- The class of unary proof (resp. unary natural; unary cut-free; unary normal) terms is contained in the class of proof (resp. natural; cut-free; normal) terms.
- $\beta\pi$ corresponds to cut-elimination; ρ corresponds to normalization; and γ, μ are the "bureaucracy" conversions.
- The faces in the right cube are named: N, S, E, W, F(=Front), B(=Back).
- The faces in the left cube are named: NL(=North face of the Left cube), SL, FE (=Front East), FW(=Front West), BE(=Back East), BW(=Back West).

  Recall μ preserves "γ-normality", as the range of γ is $\lambda\mathbf{nm}$, which is closed for μ.

▶ **Theorem 50** (Commutation between μ and γ). $\gamma(t) \to^*_{\mu} \gamma(\mu t) \to^*_{\mu} \mu(\gamma t)$.

Proof. First observe that part 3 of Theorem 47 gives: if $t \to^*_{\mu} t'$ in $\lambda\mathbf{Jm}$, then $\gamma(t) \to^*_{\mu} \gamma(t')$ in $\lambda\mathbf{nm}$. From this, together with $t \to^*_{\mu} \mu t$, we get $\gamma(t) \to^*_{\mu} \gamma(\mu t)$. From this, together with $\gamma t \to^*_{\mu} \mu(\gamma t)$, we conclude that $\mu(\gamma t)$ is the μ-nf of $\gamma(\mu t)$. ◀

In general, $\gamma(\mu t) = \mu(\gamma t)$ does not hold, as γ does not preserve μ-normality. By the theorem, we only have $\mu(\gamma(\mu t)) = \mu(\gamma t)$. Therefore, we define the combination of γ and μ to be $\mu \circ \gamma$, denoted γ'. This defines a map from $\lambda\mathbf{Jm}$ to $\lambda\mathbf{m}$, sending an arbitrary proof to a focused one. In this sense, this map may be called a *focalization* process.

▶ **Theorem 51** (Commutation). *Every face of the two cubes in Fig. 14 commutes.*

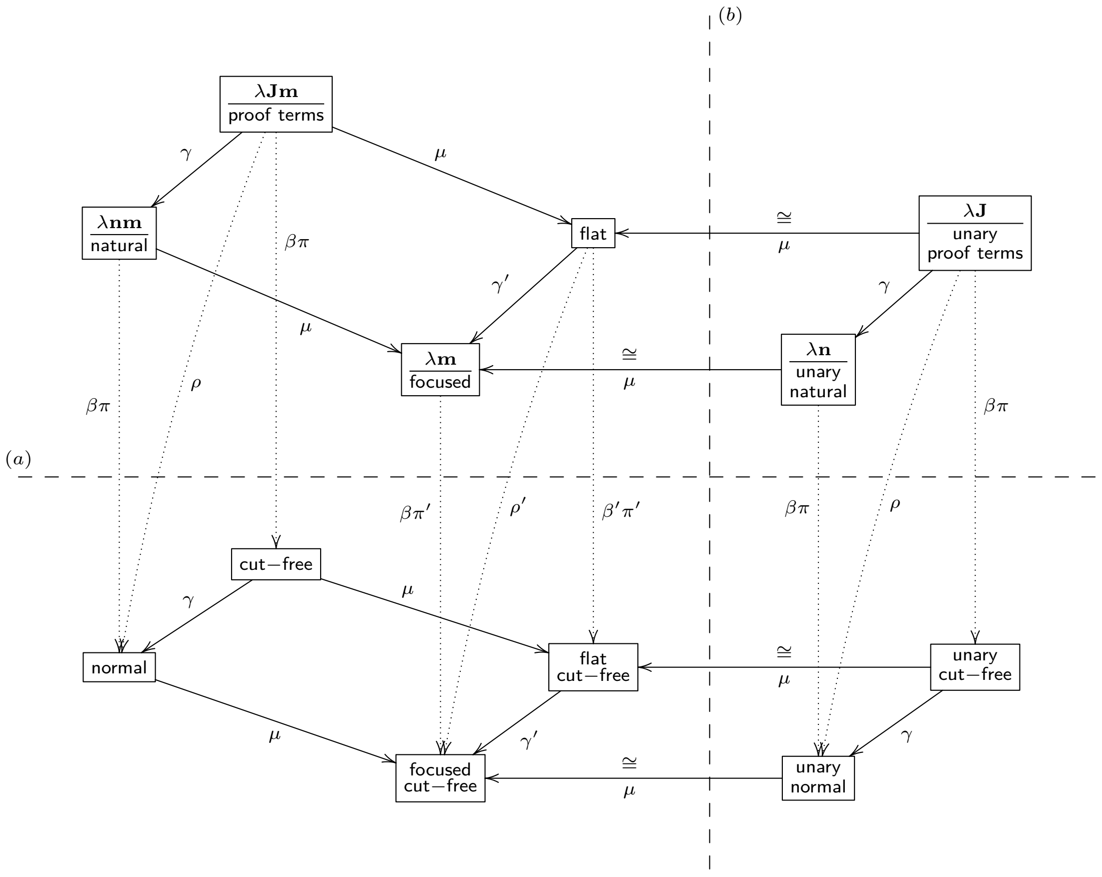


Figure 14 The internal structure of the sequent calculus $\lambda\mathbf{Jm}$.

Proof. The equality $\mu(\gamma(\mu t)) = \mu(\gamma t)$ is the commutativity of face NL in Fig. 14. We now argue the commutativity of every other face, with faces named according to the explanation given below the figure. Face SL: particular case of face NL, as γ and μ preserve cut-freeness. Face BW: Theorem 48. Face BE: $\mu(\downarrow_{\beta\pi}(t))$ and $\downarrow_{\beta'\pi'}(\mu t)$ are $\beta\pi\mu$-nfs of t, hence are the same term by confluence of $\beta\pi\mu$-reduction. Face B: by the isomorphism between $\lambda\mathbf{J}$ and the flat subsystem [8], which links β, π with β', π', respectively. Face F: by the isomorphism between $\lambda\mathbf{n}$ and $\lambda\mathbf{m}$ [6]. Face N: the unary particular case of face NL. This may be seen as extending to γ, γ' the isomorphism between $\lambda\mathbf{J}$ and the flat subsystem. Face S: the unary particular case of face SL, or particular case of face N, as γ' and μ preserve cut-freeness. Face E: the unary particular case of face BW. Face FE: by the isomorphism between $\lambda\mathbf{J}$ and the flat subsystem, which means that face E is isomorphic to face FE. That the "diagonal" map of face FE is ρ' (*i.e.* $\mu \circ \rho$) follows from the commutativity of faces SL and BE. ◀

To conclude, Fig. 14 says that $\lambda\mathbf{Jm}$ consists of two levels linked by cut-elimination, each level organized by the "bureaucratic" conversions γ and μ - and we see that the organization is quite tidy. Above the line (a) the maps are "morphisms" of λ-calculi: in addition to the isomorphisms that cross the line (b), recall the properties of μ and γ, namely Theorems 7, 33, and 47. The permutation-free fragment $\lambda\mathbf{nm}$ and its sub-fragment $\lambda\mathbf{m}$ have clear computational meaning: (multi-)multiary λ-calculi whose normal forms can be found by a (relaxed) focusing proof-search strategy.

5 Final remarks

This paper is a study of the computational interpretation of the sequent calculus that deals with the permutability phenomenon, hence distinguished either from the approaches that avoid permutability altogether by staying in some permutation-free fragment, or from approaches that simplify the problem by staying in the cut-free fragment or in some fragment that is indistinguishable from natural deduction. Our contribution is two-staged: first we studied the permutation-free fragment, then we mediated the full and the permutation-free systems by means of permutative conversions. In the permutation-free level, the novelty is in the computational interpretation: the "multi-multiary" λ-calculus is the transparent Curry-Howard interpretation of natural proofs, and normal proofs can be searched by a new, relaxed form of focusing. Beyond the permutation-free level, the novelty is in the permutative conversion γ and how, with its help, a complete picture of the internal structure of the sequent calculus $\lambda\mathbf{Jm}$ is achieved, as seen in Fig. 14: two halves mediated by cut-elimination and organized by the "bureaucracy" conversions γ and μ. To measure the progress achieved, this picture should be compared with the wisdom established long ago [4] for the cut-free setting and depicted in Fig. 1.

On the way to such a complete picture, numerous side contributions were made, including: the technicalities involving append operators and pseudo-lists that permitted a smooth handling of natural proofs; the always surprising richness of "abbreviation" conversion μ, this time promoted to a morphism between the natural and the focused fragments; the concept of special substitution on natural proofs, which is the computational process behind permutative conversion γ; and the indirect contributions to ΛJ *qua* unary fragment $\lambda\mathbf{J}$.

Among the previous papers on $\lambda\mathbf{Jm}$ [8, 6, 9], the present is closer to [6] in its attempt to refine the naive view that $\lambda\mathbf{Jm}$ is obtained from the λ-calculus by the addition of the multiarity and generality dimensions. But the purpose of [6] was to catalogue classes of normal forms (and rewriting systems giving rise to them). Curiously, the class of normal proofs studied here escaped that catalogue; and even if we find there the statement that natural proofs form a subsystem, no computational interpretation was developed. In addition, a conversion γ was proposed in [6], but it employed ordinary substitution, which does not preserve cut-freeness, hence does not preserve normality. In the present paper, we backtrack, employ special substitution in the definition of γ, and start afresh.

It is important to notice that the purpose of this paper is just to identify computational meaning: to assess whether that meaning is useful in practice is out of scope. For instance, we are happy to pin down the relaxation of focusing that constitutes the proof-search procedure for normal proofs. Such variation on focusing seems to be new, and seems useful in practice, allowing some parallelism in the synchronous phase - but we do not say more. Also the Curry-Howard interpretation of the natural subsystem (a λ-calculus where functions are applied to a vector of vectors of arguments) is perhaps not exciting, but is transparent and illuminating: it means that, in the natural fragment, the generality feature is reduced to a second-level vectorization mechanism.

Only space limitation prevented us from developing the study of other reduction procedures inside $\lambda\mathbf{Jm}$ like focalization (captured by the combination of γ and μ) and its combination with cut-elimination or normalization. On the other hand, further work is needed if one is interested in rewriting systems of permutative conversions, like those in [4, 20]. The present concept of special substitution gives a hint of what global operation the local rewrite steps should be calculating; but a generalization of that operation from natural proofs to arbitrary proofs is required, and this is on-going work.

References

1 T. Brock-Nannestad, N. Guenot, and D. Gustafsson. Computation in focused intuitionistic logic. In *Proc. of PPDP'15*, pages 43–54. ACM, 2015.

2 I. Cervesato and F. Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

3 P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of ICFP'00*, pages 233–243. IEEE, 2000.

4 R. Dyckhoff and L. Pinto. Permutability of proofs in intuitionistic sequent calculi. *Theoretical Computer Science*, 212:141–155, 1999.

5 J. Espírito Santo. Curry-Howard for sequent calculus at last! In *Proc. of TLCA'15*, volume 38 of *LIPIcs*, pages 165–179. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

6 J. Espírito Santo, M. J. Frade, and L. Pinto. Structural proof theory as rewriting. In *Proc. of RTA'06*, volume 4098 of *Lecture Notes in Computer Science*, pages 197–211. Springer, 2006.

7 J. Espírito Santo, R. Matthes, and L. Pinto. Continuation-passing style and strong normalisation for intuitionistic sequent calculi. *Logical Methods in Computer Science*, 5(2), 2009.

8 J. Espírito Santo and L. Pinto. Confluence and strong normalisation of the generalised multiary λ-calculus. In *Revised selected papers from TYPES'03*, volume 3085 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 2004.

9 J. Espírito Santo and L. Pinto. A calculus of multiary sequent terms. *ACM Trans. Comput. Log.*, 12(3):22, 2011.

10 J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

11 J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Univ. Press, 1989.

12 H. Herbelin. A λ-calculus structure isomorphic to a Gentzen-style sequent calculus structure. In *Proc. of CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995.

13 F. Joachimski and R. Matthes. Standardization and confluence for a lambda calculus with generalized applications. In *Proc. of RTA'00*, volume 1833 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2000.

14 S. Kleene. Permutability of inferences in gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10:1–26, 1952.

15 C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logic. *Theoretical Computer Science*, 410:4747–4768, 2009.

16 D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, 1991.

17 G. Mints. Normal forms for sequent derivations. In P. Odifreddi, editor, *Kreiseliana*, pages 469–492. A. K. Peters, Wellesley, Massachusetts, 1996.

18 S. Negri and J. von Plato. *Structural proof theory.* Cambridge University Press, 2001.

19 D. Prawitz. *Natural Deduction. A Proof-Theoretical Study.* Almquist and Wiksell, 1965.

20 H. Schwichtenberg. Termination of permutative conversions in intuitionistic gentzen calculi. *Theoretical Computer Science*, 212(1-2):247–260, 1999.

21 R. J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3):21:1–21:33, 2014.

22 A. Troelstra and H. Schwitchtenberg. *Basic Proof Theory.* Cambridge Univ. Press, 2000.

23 J. von Plato. Natural deduction with general elimination rules. *Annals of Mathematical Logic*, 40(7):541–567, 2001.

24 N. Zeilberger. On the unity of duality. *Annals of Pure and Appllied Logic*, 153(1-3):66–96, 2008.

25 J. Zucker. The correspondence between cut-elimination and normalization. *Annals of Mathematical Logic*, 7:1–112, 1974.

Covering Spaces in Homotopy Type Theory

Kuen-Bang Hou (Favonia)
Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, MN, USA
favonia@umn.edu
https://orcid.org/0000-0002-2310-3673

Robert Harper
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
rwh@cs.cmu.edu
https://orcid.org/0000-0002-9400-2941

Abstract

Broadly speaking, algebraic topology consists of associating algebraic structures to topological spaces that give information about their structure. An elementary, but fundamental, example is provided by the theory of covering spaces, which associate groups to covering spaces in such a way that the universal cover corresponds to the fundamental group of the space. One natural question to ask is whether these connections can be stated in homotopy type theory, a new area linking type theory to homotopy theory. In this paper, we give an affirmative answer with a surprisingly concise definition of covering spaces in type theory; we are able to prove various expected properties about the newly defined covering spaces, including the connections with fundamental groups. An additional merit is that our work has been fully mechanized in the proof assistant Agda.

2012 ACM Subject Classification Mathematics of computing → Lambda calculus, Mathematics of computing → Algebraic topology, Theory of computation → Type structures, Theory of computation → Constructive mathematics

Keywords and phrases homotopy type theory, covering space, fundamental group, mechanized reasoning

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.11

Supplement Material Mechanized proofs available at `https://github.com/HoTT/HoTT-Agda`.

Funding This research was sponsored by the National Science Foundation under grant numbers CCF-1116703 and Air Force Office of Scientific Research under grant numbers FA-95501210370 and FA-95501510053. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Acknowledgements We are grateful to Carlo Angiuli, Steve Awodey, Spencer Breiner, Guillaume Brunerie, Ulrik Buchholtz, Daniel Grayson, Chris Kapulkin, Peter LeFanu Lumsdaine, Ed Morehouse and Jonathan Sterling for teaching the first author the classical theory and providing useful feedback on earlier presentations or drafts. We also thank anonymous reviewers for providing detailed and useful comments. The category-theoretic argument for the representation theorem was contributed by Steve Awodey.

1 Introduction

Homotopy type theory [33] is a new area arising from surprising connections between type theory, homotopy theory and category theory. Using a variant of Martin-Löf type theory [29, 33, 36] extended with Vladimir Voevodsky's univalence axiom [19] and higher inductive types [27, 33], homotopy-theoretic concepts can be expressed in type theory in a direct and intuitive way, as we will see in the case of covering spaces.

The connection between this variant of Martin-Löf type theory and homotopy theory is through the *identification-as-path*[1] interpretation [3, 12, 19, 28, 33–35, 38]. According to this interpretation, types may be treated as spaces,[2] elements of a type as points in a space, functions as continuous mappings, families of types as fibrations, and of course identifications as paths;[3] the higher-dimensional structures induced by iterated identification make every type an ∞-groupoid.[4] With this connection, we can use type-theoretic approaches to state, prove and even mechanize theorems from classical homotopy theory, making the type theory a framework of *synthetic homotopy theory*; proofs are dependently typed functional programs, except that they do not run due to incomplete support of computation of the univalence axiom and higher inductive types in current mature proof assistants and the type theory we use in this paper.

A wide range of homotopy-theoretic results have been developed and mechanized in proof assistants such as Agda [6, 30], Coq [1, 4] and Lean [9, 11], for example homotopy groups of spheres [5, 23, 26, 33], the Seifert–van Kampen theorem [17], the Blakers–Massey connectivity theorem [16], the Eilenberg–Mac Lane spaces [25], the Mayer–Vietoris sequences [10], the Cayley–Dickson construction [8], the double groupoids [37] and many more [24, 32, 33]. Proofs done in homotopy type theory have the advantage that they admit many models other than the homotopy theory of topological spaces; some even stimulated new research in mathematics [2, 31]. As a side note, many theorems were actually first mechanized in proof assistants and then "unmechanized" to engage wider audience, which is only possible through a powerful, high-level framework such as homotopy type theory.

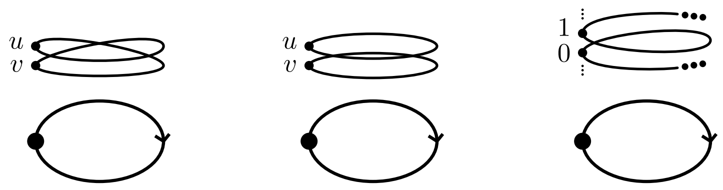
Covering spaces are one of the important constructs in homotopy theory, and given the connection between type theory and homotopy theory, a natural question to ask is whether such a notion can be stated in type theory as well. It turns out that we can express covering spaces concisely as follows.

▶ **Definition 1.** A *covering space* of a type (space) A is a family of sets indexed by A.

That is, the type of covering spaces of A is simply $A \to \texttt{Set}$ where `Set` is the type of all sets, the universe of all types that have at most one identification between any two points. Several examples are shown in Figure 1.

How do we know this definition really defines covering spaces? A characteristic feature of covering spaces of a connected space A in the classical theory is that they are represented by sets with a group action of the fundamental group of A (the set of loops at some point

1 Identification types are also called *identity types* in the literature.
2 More precisely, *simplicial sets*, and the interpretation was given in [19]. *Spaces* in this paper really mean *simplicial sets* unless we are explicitly discussing point-set topology. For clarity, spaces in point-set topology will be denoted as *topological* spaces.
3 All topological terms in this paper should be understood "up-to-homotopy" in some appropriate sense, because every construct in the type theory will respect homotopy equivalence under the intended interpretation to simplicial sets.
4 To avoid confusion, this result was a meta-theoretical result given in [34], and we believe it has not been internalized in type theory yet.



$$\begin{cases} \text{base} \mapsto \mathbb{2} \\ \text{loop} \mapsto \texttt{ua}(\texttt{not}) \end{cases} \qquad \begin{cases} \text{base} \mapsto \mathbb{2} \\ \text{loop} \mapsto \texttt{ua}(\texttt{id}) \end{cases} \qquad \begin{cases} \text{base} \mapsto \mathbb{Z} \\ \text{loop} \mapsto \texttt{ua}(+1) \end{cases}$$

set: $\{u, v\}$	set: $\{u, v\}$	set: $\mathbb{Z}$
action: $u \mapsto v$	action: $u \mapsto u$	action: $i \mapsto (i + 1)$
$v \mapsto u$	$v \mapsto v$	

Figure 1 Correspondence between covering spaces and sets equipped with a group action. The top row is the visualization of the covering spaces. The middle row shows their type-theoretic formulations (as a function from the circle to the universe) specified by a fiber for the base generator and an identification from the fiber to itself for the loop generator of the circle. The last row is the corresponding sets and their actions, represented by their acts on the (only) loop generator.

in A). Therefore, we may justify our definitions by proving this theorem, as we will in Section 4. See Figure 1 which also lists such sets corresponding to the covering spaces in the figure. Moreover, considering the category[5] of pointed covering spaces where morphisms are fiberwise functions, we also know there should be an initial covering space (named the *universal* covering space) and it should be represented by the fundamental group itself through the representation theorem stated above.[6] We also managed to show these results as demonstrated in Section 5. Before transitioning to these main theorems, in Section 3 we will also discuss briefly about the discrepancies between our formulation and the classical definition. More discussions and future research directions can be found in Section 6.

All the results mentioned in this paper have been mechanized in the proof assistant Agda [6]. The representation theorem was briefly mentioned in the book without proofs [33] and an extended abstract without peer review was posted before [15], but a full paper was never published.

2 Type-Theoretic Notation and Background

We assume readers are already familiar with basic concepts in homotopy type theory, including higher inductive types; interested readers are recommended to read the book [33] for introduction, especially its Chapter 2 describing how type-theoretic concepts may be understood homotopy-theoretically. This section is mainly a brief overview of the notation used in this paper with remarks on some subtle differences from the book [33] or the proof assistant Agda. Overall we are loosely following the style of the book [33] while keeping an Agda translation obvious.

[5] The *category* here is the same as the *category* introduced in [33, §9.1]. The type of morphisms (fiberwise functions) between two covering spaces is a set because each fiber of a covering space is a set, and the notion of isomorphism in this category collides with identification.

[6] See Section 5 for a more precise statement.

Throughout this paper, $\equiv$ is judgmental equality and $:\equiv$ indicates a definition. The equal sign $=$ is reserved for identification as mentioned below.

2.1 Sums and Products

Let B be a family of types indexed by a type A. *Dependent sum types* are written $\sum_{x:A} B(x)$ with pairs $\langle a; b \rangle$ as elements and *dependent function types* $\prod_{x:A} B(x)$ with λ-functions. The type $\sum_{x:A} B(x)$ is also called the *total space* of B. If $B(x) \equiv B'$ actually does not depend on the index $x : A$, we have the binary product type $A \times B'$ meaning the *non-dependent* sum type $\sum_{\_:A} B'$ and the arrow type $A \to B'$ the *non-dependent* function type $\prod_{\_:A} B'$. Function compositions are written $f \circ g$.

Multi-argument application is written $f(x_1, x_2, \cdots, x_n)$ and nested sum types will be presented as records types with labels (like "`label`"). As a notational abuse, a label is also the projection function which projects out the corresponding component from a record.

2.2 Identification

Let a and b be two points in some type A. The *identification type* or the *path type* between a and b is written $a =_A b$, and A may be omitted if clear from the context. The reflexivity identification at a is written $\texttt{refl}_a$, the concatenation (in the diagram order) written $p \bullet q$, and the inverse identification written p^{-1}.

The induction principle of identification types intuitively states that, given a statement about identifications, one can just consider the `refl` case. The argument is that one may continuously grow a `refl` to arbitrary identifications, and because every function in the type theory is continuous, the conclusion remains valid. However, to make this "continuous-growing" argument work, the precise formulation of this principle is quite delicate and is discussed in more details in [33, §1.12]. For example, the statement about identifications must make sense for identifications between two possibly different points in order to allow the `refl` case to "grow".

As mentioned in the introduction, identification types may be iterated as $p =_{a=_A b} q$, $P =_{p=_{a=_A b} q} Q$ and so on. Throughout the paper the word *dimension* refers to the level of identification iteration; that is, the n-dimensional structures in type A refer to the nth iteration of identification starting from the type A.

2.3 Universes, Equivalence and Univalence

Both the type theory and the proof assistant Agda have a ramified hierarchy of cumulative universes to avoid Girard's paradox [13, 18], but in this paper we will suppress the universe level, pretending there is only one universe written $\mathcal{U}$. Universe levels are explicit and universe lifting is manual in the current Agda system, but they did not constitute an obstacle to mechanizing covering spaces.

The equivalence type $A \simeq B$ intuitively collects all the equivalences between types A and B. It is actually tricky to obtain a good definition for equivalences in homotopy type theory; interested readers are recommended to consult [33, Chap. 4] for a precise definition. For this paper it suffices to know that the following data are sufficient to build a good equivalence: a function from A to B, a function from B to A, and two proofs showing the two compositions are homotopic to the identity functions.

With a good definition of equivalences, we have the univalence axiom stating that the equivalence type between types A and B is itself equivalent to the identification type $A =_{\mathcal{U}} B$.

The univalence axiom not only recognizes new identifications between types, but also has profound impact on the type theory; in particular, functional extensionality becomes provable, and is used throughout the paper for our covering spaces are defined as functions from the base type to the universe.

Two families of types indexed by the same type are equivalent if they are fiberwise equivalent, and a *fiberwise function* between two families is a family of functions between corresponding fibers. By the univalence axiom (and functional extensionality), fiberwise equivalence also implies the identification of two families of types.

For a family of types B indexed by a type A, an identification p in A from point a to point b will force an equivalence between corresponding fibers $B(a)$ and $B(b)$. The intuition is that a family of types indexed by B is a function from B to the universe $\mathcal{U}$; it will preserve identifications, and by the univalence axiom identifications in the universe are equivalences. The equivalence (as a function) is called *transport* and is written $\mathtt{transport}^{x.B(x)}(p; a')$, meaning the transport of $a' : B(a)$ along $p : a =_A b$ across the family B to the fiber $B(b)$. It is also functorial in p in the sense that it sends reflexivity to identity equivalence and path concatenation to equivalence composition.

2.4 Truncation and Connectivity

Truncation levels denote the dimension (iteration level of identification) *above which* a type is trivial: a type is at level -2 if it is *contractible*, which means it is equivalent to the unit type and is trivial at all dimensions, and a type is at level $(n + 1)$ if its identification types lie at level n. It may seem odd that the level starts with -2, not 0, but it matches well with other theories such as groupoid theory; for example, there is a tight connection between types at level 1 and 1-groupoids.

A type at level -1 is called a *mere proposition*, where any two points are identified, and a type at level 0 is called a *set*, where any two parallel identifications are identified. Equivalences between sets are called *isomorphisms*.[7] It can be shown that the truncation levels form a cumulative hierarchy, in addition to the existing one based on their universe levels (which are suppressed in this paper).

An *n-type* [33, §7.1] is a type at truncation level n. The type $\mathtt{Set}$, as mentioned above, is the type of all 0-types. The *n-truncation* of a type A is, intuitively, the *best* n-type approximation of the type A, written $\|A\|_n$, where the projection of $a : A$ into the truncation is written $|a|_n$. More precisely, $\|A\|_n$ is the n-type with the universal property that there is a unique extension of any function of type $A \to B$ to $\|A\|_n$ for any n-type B, as shown below. The n-truncation of an n-type is equivalent to the n-type itself.

$$\begin{tikzcd} A \arrow[r] \arrow[d, "|-|_n"'] & B \\ \|A\|_n \arrow[ur, dashed] & \end{tikzcd}$$ for any n-type B.

Connectivity [33, §7.5] is the "dual" of truncation level in the sense that an n-connected type is trivial *below or at* the dimension n. See Figure 2 for a visualization of 0-connected and 1-connected spaces. In this work we critically rely on the fact that, for any two points in an n-connected type, there is an element in the $(n - 1)$-truncation of the identification type between those two points. Technically, an n-connected type is defined to be a type whose n-truncation is contractible, meaning that it can only have non-trivial structures above dimension n.

[7] This follows the convention in [33, §2.4].

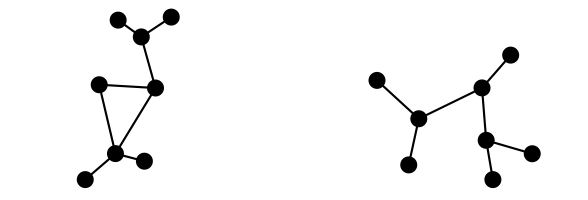


Figure 2 Examples of connected spaces without structures above dimension 1.

Vertices represent the elements and edges represent the identification generators. The space on the left is not 1-connected because paths between points are not unique. Conversely, a 1-connected space is always 0-connected.

Because we will be working closely with many elements in the n-truncation of identification types, we may call such elements *n-truncated identifications* for short, or even *truncated identifications* if the truncation level is clear from the context.

Throughout this paper, some mere propositions are called *properties*, hinting they are mathematical properties whose witnesses are irrelevant (except ther existence), in contrast with mathematical *structures* which might carry non-trivial information.

2.5 Set Quotients

Let A be a type and $R : A \to A \to \mathcal{U}$ a family of types doubly indexed by A. We write A/R as the set quotient of A by R, $[a]$ as the equivalence class of $a : A$, and $\texttt{quot}(r)$ for $r : R(a, b)$ as a witness of $[a] =_{A/R} [b]$. The family R need not be an equivalence relation itself, but the set quotient in type theory effectively takes the reflexive, symmetric and transitive closure of R. Note that we did not require R to be a family of mere propositions as in the book [33] because in theory it made little difference and in practice it is convenient not to be concerned about truncation levels. Similarly, A is not required to be a set, even though the set quotient A/R always is.

2.6 Fundamental Groups and Truncated Identification

As mentioned earlier, iterated identification forms the structure of ∞-groupoids. The 0-truncation of identification thus behaves like ordinary groupoids, which reduce to groups if we only focus on some particular point [33]. More precisely, given a type A with a distinguished point a, the *fundamental group* of the type A at a, written $\pi_1(A, a)$, is the set $\|a = a\|_0$ along with concatenation as composition and (truncated) reflexivity as the unit. When the distinguished point a is clear from the context, we may omit the point and write $\pi_1(A)$ for short.

We will reuse the path concatenation and path inverse ($p \cdot q$ and p^{-1}) on 0-truncated identifications for a cleaner presentation; however, the distinction is still important and so we will mark every other use of truncation. In particular, the transport function on 0-truncated identifications is written with an additional subscript "0" as

$$\texttt{transport}_0^{x.B(x)}(p; a') : B(b)$$

where B is a family of 0-type indexed by A, p is a 0-truncated identification in $\|a = b\|_0$, and a' is a point in the fiber $B(a)$. It is important that the truncation level of p (which is 0 here) matches the truncation level of B so that we may apply the universal property of truncation.

2.7 Implicit Coercion

To further reduce notational clutter, we adopt implicit coercion when no confusion would occur. For example, a group may be implicitly coerced into its underlying set; an n-type, which in reality carries a proof of its truncation level, may drop the proof silently; and a pointed type may be coerced into its carrier. AGDA has limited support of coercion through instance arguments, but we did not use them in our mechanization except for numeric literals.

3 Comparison with Classical Definition

Our type-theoretic formulation appears quite different from the classical definition of covering spaces, and thus readers with the background in classical algebraic topology might wonder how this definition links to the classical one.

The situation is somewhat complicated because our construction lies in the type theory while the most common classical definition is expressed in point-set topology. We have an interpretation of the type theory into simplicial sets, and then geometric realization of simplicial sets into topological spaces, but not a direct interpretation into topological spaces yet to the best of our knowledge. A rigorous mathematical proof will involve interpreting our construction (Definition 1) into simplicial sets and then topological spaces, and is unfortunately beyond the scope of this paper. Instead, we will only give some intuition about the linkage in this section, and provide more internal evidence throughout the paper.

Here is a definition of covering spaces in terms of point-set topology [14, p. 29]:

▶ **Definition 2** (classical definition of covering space). A *covering space* of a topological space A is a topological space C with a continuous surjective map $\pi : C \to A$ such that for each point $a \in A$ there is an open neighborhood U of a in A such that $p^{-1}(U)$ is a union of disjoint open sets, each mapped homeomorphically onto U by p.

The connection between the two kinds of covering spaces lies in several critical observations:

- Definition 1 defines a covering space as an A-indexed family of type F while Definition 2 focuses on a map p from C to A. To fit a covering space of the first kind, F, into the latter definition, one may choose the total space $\sum_{a:A} F(a)$ as C and the first projection as the map from C to A; the notion preimage $p^{-1}(a)$ is then replaced by the fiber $F(a)$. In general, type families and fibrations (for example p here) are equivalent and this connection is discussed in details in [33, §2.3]. We chose families over fibrations because it is easier to work with families of types inside the type theory.
- Next, the use of neighborhoods can be largely avoided because every space constructed by the standard geometric realization of a simplicial set is a CW complex and thus satisfies all local connectedness properties (for example local path-connectedness or semi-local simple connectedness). Moreover, every construct in type theory is continuous under this interpretation. Therefore, there is no need to mention local connectivity or continuity, because we cannot define any "bad" space in the type theory.
- Homeomorphism is weakened to homotopic equivalence because, again, it is impossible to distinguish homeomorphic but not homotopic objects inside the type theory.[8]

[8] This does not take into account of the possibility of, for example, internalizing the entire set theory in the type theory and redoing the point-set topology. We assume a more direct interpretation into simplicial sets and then topological spaces.

The real discrepancy is that the classical definition requires that the total space C (or $\sum_{a:A} F(a)$ from the first definition) to be non-empty and that p (or the first projection from $\sum_{a:A} F(a)$ to A) is surjective. This condition is needed for the universal covering to be *universal*, as we will discuss in Section 5; otherwise the empty space would be the universal covering space for any base type. However, without the non-emptiness or surjectivity requirement, the representation theorem (Theorem 4) does not have to rule out empty sets with actions; moreover, in a constructive setting there are many possible formulations of these conditions that are all classically equivalent but with different constructive content. Indeed, in Section 5 where we discuss universal covering spaces, we derive a pointedness condition that is constructively much stronger than (but classically equivalent to) mere non-emptiness. It is important to isolate the usage of non-emptiness or surjectivity to study their impact in constructive mathematics.

As a further justification, one can immediately prove the following lemma in the type theory when the base type A is the circle S^1:

▶ **Lemma 3.** *There is an equivalence between $S^1 \to \mathtt{Set}$ and sets with an automorphism.*

Proof. (Omitted, but fully mechanized in the proof assistant AGDA as a separate lemma.) ◀

This lemma is a special case of the main theorem we will present in the next section.

4 Representation Theorem

The first main result of this paper is that covering spaces of a 0-connected, pointed space A are represented by *sets equipped with a group action of the fundamental group of* A, which is to say there is an equivalence between covering spaces and such sets. The intuition is that everything in homotopy type theory must respect identification, and the fact that the base type A is 0-connected indicates that there is a (-1)-truncated identification between any two points and thus a (-1)-truncated isomorphism between any two fibers. Therefore, it is represented by one copy of these isomorphic sets and a description of how they are isomorphic, encoded as an action of the fundamental group. See Figure 1 for examples of how a covering space is represented by a set with an action.

Formally, a set with a group action of G is called a *G-set*, a functor from the group G (treated as a category with one object and elements in G as morphisms) to the category of sets up to isomorphism; a *group set* is a G-set without the group G being specified. In type theory, a *G-set* is a record with the following components:

- $\mathtt{El}$: a set.
- α: a (right) group action of type $\mathtt{El} \to G \to \mathtt{El}$.
- α-$\mathtt{unit}$: a proof of the property that α preserves the group identity:

$$\prod_{x:\mathtt{El}} \alpha(x, \mathtt{unit}(G)) =_{\mathtt{El}} x.$$

- α-$\mathtt{comp}$: a proof of the property that α preserves the group composition:

$$\prod_{x:\mathtt{El}} \prod_{g_1, g_2 : G} \alpha(x, \mathtt{comp}(G)(g_1, g_2)) =_{\mathtt{El}} \alpha(\alpha(x, g_1), g_2).$$

The representation theorem is then about covering spaces being represented by $\pi_1(A, a)$-sets, which can be formally stated as follows:

▶ **Theorem 4** (representation by group sets)**.** *For any 0-connected type A with a point a, we have $(A \to \mathtt{Set}) \simeq \pi_1(A, a)$-$\mathtt{Set}$.*

Proof. The standard methodology to show equivalence in homotopy type theory is to establish two functions inverse to each other. That is, we want to establish two functions from covering spaces $A \to \texttt{Set}$ to group sets $\pi_1(A)\text{-}\texttt{Set}$ and vice versa, and show that the round-trips are the identity function.

The direction from covering spaces $A \to \texttt{Set}$ to group sets $\pi_1(A)\text{-}\texttt{Set}$ is relatively straightforward: the group set should capture a representative fiber with isomorphisms between fibers. Because the base type A is 0-connected, every fiber is equally qualified, and so we choose the one over the distinguished point a. Moreover, recall that the isomorphism forced by an identification in the base type, as discussed in Section 2, is the transport function. Putting these together, we can define a $\pi_1(A)$-set from a covering space $F : A \to \texttt{Set}$ by taking

$$\begin{aligned} \texttt{El} &:\equiv F(a) \\ \alpha &:\equiv \lambda x.\lambda g.\texttt{transport}_0^{x.F(x)}(g; x) \end{aligned}$$

with properties α-`unit` and α-`comp` derived from functoriality of $\texttt{transport}_0$. The reason that we only have to record the automorphisms of $F(a)$ forced by loops at a (instead of all isomorphisms between all fibers) is because A is 0-connected; that is, every point in A is merely connected to a by a (-1)-truncated identification, and thus the automorphisms at a determine the isomorphisms between all fibers.

The other direction, from group sets $X : \pi_1(A)\text{-}\texttt{Set}$ to covering spaces, is more technically involved. A good guide is to focus on a group set generated from some covering space $F' : A \to \texttt{Set}$ through the above process; if the theorem is true, we should be able to recreate a covering space $F : A \to \texttt{Set}$ equivalent to F'. A key observation is that every point in any fiber of F' is a result of transporting some point in the fiber $F'(a)$ to that fiber, noting that X was defined to be $F'(a)$. Thus, one idea is to populate the new family F with *formal transports from X quotiented by the supposed functoriality of transports and the agreement with α*, in the hope to mimic the real $\texttt{transport}_0$ in F'. The formal definition is shown as follows; in the definition, the quotient relation $\sim_b$ can be seen as a succinct summary of the functoriality of transports and the agreement with α.

▶ **Definition 5** (reconstructed covering space)**.** Let A be a type with a point a and X be a $\pi_1(A, a)$-set with an action α. The *reconstructed covering space*, $F : A \to \texttt{Set}$, is defined as

$$F :\equiv \lambda b.(X \times \|a =_A b\|_0)/{\sim_b}$$

where the relation $\sim_b$ is defined as the least relation containing

$$\langle \alpha(x, \ell); p\rangle \sim_b \langle x; \ell \bullet p\rangle \text{ for any } x : X,\ \ell : \|a = a\|_0 \text{ and } p : \|a = b\|_0.$$

This completes the construction of the new covering space F.

The next step is to show that these two functions are indeed inverse to each other. However, in this paper we will only highlight the interesting part in proving the reconstructed covering space F is indeed equivalent to the original F'. Following the standard recipe of equivalence, two functions back and forth are needed for the equivalence between two covering spaces. The direction from F to F' is simply realizing the formal transports; that is, for any point $b : A$ and any representive $\langle x; p\rangle$ in the fiber $F(b)$ (defined as a set quotient), we have

$$\texttt{transport}_0^{x.F'(x)}(p; x) : F'(b)$$

because $x : X$, $p : \|a = b\|_0$ and $X :\equiv F'(a)$. One can then show this expression respects the quotient relation $\sim_b$ imposed on $F(b)$ in Definition 5. The other direction is somewhat unclear – given a point y in the fiber $F'(b)$, how shall we locate a point x in $F(a)$ and compute a truncated identification p such that y will be the result of transporting x along p?

Recall that the connectivity of A implies that there is a (-1)-truncated identification between any two points. That is, for any point $b : A$ we have a truncated identification $p : \|a =_A b\|_{-1}$. One attempt is then to transport y along the *inverse* of p to some point x in $F(a)$, for transporting x back along p should cancel the opposite transportation and recover y; the pair $\langle x; p\rangle$ in $F(b)$ then corresponds to y. The problem is that all the transportation and pairing demand 0-truncated identifications but p is a (-1)-truncated identification. In other words, there is a gap between the truncation level of the identifications from connectivity (-1) and that of the fibers of covering spaces (0), which prevents the application of the universal property of truncation.

Fortunately, such a truncation level gap can be filled by a constancy condition. We can show that different choices of identifications between a and b result in pairs related by the quotient relation imposed on $F(b)$, and then, by the following lemma, we can extend the above construction to (-1)-truncated identifications. The intuition is that if a function does not depend on the value of the input but only its existence, a (-1)-truncated input should suffice.

▶ **Lemma 6** (extension by weak constancy[9]). *Let A be a type and B a set. For any function $f : A \to B$ such that $\prod_{x,y:A} f(x) =_B f(y)$ there exists a function $g : \|A\|_{-1} \to B$ such that $f \equiv g \circ |{-}|_{-1}$.*

We will now carefully construct the function from F' to F sketched above, using this lemma. For any point $b : A$, we have a function $f_b : F'(b) \to (a =_A b) \to F(b)$ as

$$f_b :\equiv \lambda y.\lambda p.\Big[\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1}; y\Big); |p|_0\Big\rangle\Big],$$

which transports y to some point in $F(a)$. We want to show Lemma 6 applies to $f_b(y, -)$ for any $y : F'(b)$ so that a (-1)-truncated identification suffices. To satisfy the constancy condition in Lemma 6, it is sufficient to demonstrate that for any two identifications p, q of type $a =_A b$

$$\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1}; y\Big); |p|_0\Big\rangle \sim_b \Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|q|_0{}^{-1}; y\Big); |q|_0\Big\rangle$$

where $\sim_b$ is the quotient relation of $F(b)$ and thus $f_b(y,p) =_{F(b)} f_b(y,q)$. This can be proved by the groupoid laws of identification and the definition of $\sim_b$; we have

$$\begin{aligned}
&\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1}; y\Big); |p|_0\Big\rangle \\
=\ &\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1}; y\Big); |p|_0 \cdot |q|_0{}^{-1} \cdot |q|_0\Big\rangle \\
\sim_b\ &\Big\langle \alpha\Big(\mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1}; y\Big), |p|_0 \cdot |q|_0{}^{-1}\Big); |q|_0\Big\rangle \\
\equiv\ &\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0 \cdot |q|_0{}^{-1}; \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1}; y\Big)\Big); |q|_0\Big\rangle \quad \text{(by definition)} \\
=\ &\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|p|_0{}^{-1} \cdot |p|_0 \cdot |q|_0{}^{-1}; y\Big); |q|_0\Big\rangle \\
=\ &\Big\langle \mathtt{transport}_0^{x.F'(x)}\Big(|q|_0{}^{-1}; y\Big); |q|_0\Big\rangle.
\end{aligned}$$

[9] The word *weak* here indicates that we do not know the value in the codomain, which is weaker than other possible definitions of *constancy*. In particular, all functions from or to the empty type are weakly constant.

This means $f_b(y, -)$ is (pairwise) constant, and thus by Lemma 6 there exists an extension $g_{b,y} : \|a =_A b\|_{-1} \to F'(b)$ to the constant function $f_b(y, -)$. Putting these together, we have the following function of type $F'(b) \to F(b)$:

$$\lambda y. g_{b,y}(p(a, b))$$

where $p(x, y)$ is the (-1)-truncated identification between x and y derived from the connectivity of A. This concludes the two functions between $F'(b)$ and $F(b)$; the remaining parts of the equivalence proof are a routine calculation. ◀

The proof of Theorem 4 critically relies on Lemma 6, which provides a sufficient condition for establishing a function from types at lower truncation level to ones at higher level, which is usually impossible because of missing coherence conditions in codomains. The lemma asserts that constancy can fill in the gap so that there is a way to extend a function to truncated types. Nicolai Kraus *et al.* have significantly generalized the result and considered the cases from mere propositions to types at arbitrary levels; see [20–22]. The following is a proof of the special case (Lemma 6):

Proof of Lemma 6. Given a function f from A to B satisfying the constancy condition, construct the set quotient $A/{\sim}$ where

$$a \sim b :\equiv f(a) =_B f(b).$$

One can then show that the function f factors through $A/{\sim}$. Because $A/{\sim}$ is provably a mere proposition, the function f can be extended to the (-1)-truncation of A. The judgmental equality is derived from the computation rules of truncations and set quotients on points. ◀

There is also an alternative argument (provided by Steve Awodey) for Theorem 4 that proceeds as follows: In the context of $A \to \texttt{Set}$, because the codomain $\texttt{Set}$ is itself a 1-type (as the type of all n-types is an $(n+1)$-type [33, Theorem 7.1.11]), structures at dimension higher than 1 in the domain A are irrelevant, which means that $(A \to \texttt{Set}) \simeq (\|A\|_1 \to \texttt{Set})$. (This can also be argued from the universal property of the 1-truncation of A.) Moreover, the 1-truncation of a pointed, 0-connected type A can be represented by its fundamental group $\pi_1(A, a)$ where a is the point,[10] and so the type $\|A\|_1 \to \texttt{Set}$ is really the collection of functors from $\pi_1(A, a)$ (as a category) to $\texttt{Set}$, or simply $\pi_1(A, a)$-sets. However, this argument relies on several components that are still not available in the Agda development; in comparison our proof is more elementary.

5 Universal Covering Spaces

In addition to the representation theorem, we also mechanize several well-known properties about a special covering space, the *universal covering space*, which is intuitively the most general or the most "unfolded" covering space over a space. It has two equivalent definitions, one based on connectivity and one based on initiality (and hence the name *universal*). In addition to the two definitions, when the base type is 0-connected it is also represented by

[10] The equivalence between $\|A\|_1$ and the Eilenberg-Mac Lane space $K(\pi_1(A, a), 1)$ was mechanized by Floris van Doorn in the library of LEAN [9, 11]. However, the authors are not aware of a published paper discussing this result in details.

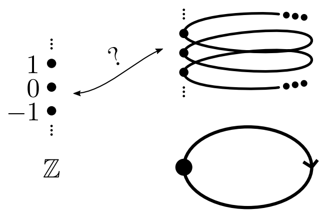


Figure 3 The lack of a canonical equivalence.

the fundamental group – which is itself a $\pi_1(A,a)$-set – through the representation theorem in Section 4; this argument was implicitly used in the calculation about the fundamental group of the circle in [26] and here we show a general result.

In this section the base type is fixed to be a type A with a distinguished point a.

▶ **Definition 7** (pointed covering space). A *pointed covering space* is a covering space whose fiber over a is pointed.

▶ **Definition 8** (universal covering space). A *universal covering space* is a pointed covering space whose total space is 1-connected.

The reason we stipulated a point in the specific fiber over the specific point is to make available a canonical choice among fiberwise equivalents. Considering the helix in Figure 3, the universal covering space over the circle whose fundamental group is integers, there are multiple different equivalences between integers and any fiber of the helix, and there is no canonical choice – until we pin down a particular point in the helix and demand it be mapped to zero. To fit the definition of fiberwise equivalences, distinguished points of different covering spaces should be in the matching fibers, and thus we further demand the distinguished point lie in the fiber over the point a.

As hinted above, the following definition should be equivalent.

▶ **Definition 9** (alternative definition of universal covering space). A *universal covering space* is a covering space which is initial in the category of pointed covering spaces with point-preserving fiberwise functions as morphisms.

The main observation to unify all these properties and simplify the proof is that the covering space consisting of 0-truncated identifications from the distinguished point

$$P :\equiv \lambda b.\|a =_A b\|_0$$

with its own distinguished point $|\mathtt{refl}_a|_0$ in $P(a)$ *is* the universal covering space. This means that it suffices to show the covering space P is the one and only pointed covering space satisfying the two definitions of universal covering spaces, and that it is represented by the fundamental group. In fact, its correspondence to the fundamental group is trivial because its fiber over the distinguished point a is exactly (the underlying set of) the fundamental group, and it is not difficult to prove the group action is the concatenation. The rest of the section is dedicated to showing the equivalence of two definitions.

First, we will show P is the one and only 1-connected covering space.

▶ **Lemma 10.** *The total space of P is 1-connected.*

Proof. To show that the total space is 1-connected, by definition it suffices to show that the 1-truncation of $\sum_{b:A} P(b)$ is contractible, which means the 1-truncation is pointed and there is an identification to any point in that truncation. The truncated pair $|\langle a; |\mathtt{refl}_a|_0\rangle|_1$ is clearly a point, and the identification between $|\langle a; |\mathtt{refl}_a|_0\rangle|_1$ and some other truncated pair $|\langle b; p\rangle|_1$ can be established by applying truncation induction and identification induction on p, which states that it suffices to consider the case $p \equiv |\mathtt{refl}_a|_0$ (and that $b \equiv a$). ◀

▶ **Lemma 11.** *Any pointed covering space whose total space is* 1*-connected is equivalent to* P*.*

Proof. Let F be a pointed covering space whose total space is 1-connected. Once again we will follow the recipe of equivalence by establishing two functions inverse to each other. The direction from P to F can be done fiberwise by transports; that is, for any $b : A$, we can define a function from $P(b)$ to $F(b)$ as evaluating the transport of the distinguished point in $F(a)$ along the input in $P(b)$ (which is a truncated identification from a to b) to the fiber $F(b)$. Formally, it is

$$\lambda p.\mathtt{transport}_0^{x.F(x)}(p; a_F^*)$$

where a_F^* is the distinguished point of F over a. The other direction is to exploit the 1-connectivity: for any point y in the total space of F, there is a 0-truncated identification from the distinguished point $\langle a; a_F^*\rangle$ to y in the total space, which can then be "projected down" to the base type as a 0-truncated identification from the point a to the point over which y is. It can then be shown that these two functions are inverse to each other. ◀

Lemmas 10 and 11 tell us P is the only 1-connected universal covering space. Thus the remaining step is to prove that P is the initial object in the category up to homotopy. Note that we did not explicitly define the category but directly talked about its morphisms.

▶ **Lemma 12.** *For any pointed covering space* F*, there exists one and only one point-preserving fiberwise function from* P *to* F*.*

Proof. The existence is again by transporting the distinguished point of F along the points in P, which are themselves 0-truncated identifications. The uniqueness is by applying truncation induction and identification induction on points in the total space P, which suggests we only have to consider the case $|\mathtt{refl}_a|_0$, the distinguished point of P. However, a point-preserving function must send $|\mathtt{refl}_a|_0$ to the distinguished point of F, and thus all such functions must agree. ◀

Now we are ready to conclude this section with the following theorem:

▶ **Theorem 13.** *For any type* A *with a point* a*, the covering space* $P :\equiv \lambda b.\|a =_A b\|_0$ *of type* A *with* $|\mathtt{refl}_a|_0$ *as its distinguished point is* the universal covering*. It is also represented by* $\pi_1(A, a)$ *if* A *is* 0*-connected.*

Proof. The first statement directly follows Lemmas 10, 11 and 13. The second statement comes from the definition of P whose fiber over a is exactly the underlying set of $\pi_1(A, a)$. ◀

6 Discussion

In this paper we show that covering spaces, an important concept in homotopy theory, can be elegantly expressed in the new framework homotopy type theory, whose synthetic nature also makes possible AGDA mechanization of length comparable to proofs on paper. The code is available at [6], and a snapshot that matches this paper is available at [7]. The development is broken into four files:

- `theorems/homotopy/CircleCover.agda`: Lemma 3.
- `theorems/homotopy/GroupSetsRepresentCovers.agda`: Theorem 4.
- `theorems/homotopy/AnyUniversalCoverIsPathSet.agda`: Lemmas 10 and 11.
- `theorems/homotopy/PathSetIsInitalCover.agda`: Lemma 12.

This paper is only the starting point of the study of covering spaces in homotopy type theory. There are still many properties unproven: for example, the representation theorem in classical theory is actually a correspondence between two categories, not just the objects. Also, the connectivity condition may be dropped if we replace fundamental groups by fundamental groupoids. Moreover, there are other possible generalizations such as n-covering spaces over a space as families of n-types indexed by that space (as a type), which to our knowledge do not immediately correspond to well-known structures in classical homotopy theory.

References

1 The Coq proof assistant. URL: `https://coq.inria.fr/`.

2 Mathieu Anel, Georg Biedermann, Eric Finster, and André Joyal. A generalized Blakers-Massey theorem. *arXiv*, 2017. `arXiv:1703.09050v2`.

3 Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1):45–55, 1 2009. `doi:10.1017/S0305004108001783`.

4 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Bas Spitters, et al. The HoTT library. URL: `https://github.com/HoTT/HoTT`.

5 Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory.* PhD thesis, Université Nice Sophia Antipolis, 2016. `arXiv:1606.05916v1`.

6 Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy type theory in Agda. URL: `https://github.com/HoTT/HoTT-Agda`.

7 Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy type theory in Agda. `doi:10.6084/m9.figshare.5161546`.

8 Ulrik Buchholtz and Egbert Rijke. The Cayley-Dickson construction in homotopy type theory. `arXiv:1610.01134v1`.

9 Ulrik Buchholtz, Floris van Doorn, and Jakob von Raumer. Homotopy type theory in Lean. To appear in the Proceedings of the 8th International Conference on Interactive Theorem Proving. `arXiv:1704.06781v1`.

10 Evan Cavallo. Synthetic cohomology in homotopy type theory. Master's thesis, Carnegie Mellon University, 2015. URL: `http://www.cs.cmu.edu/~ecavallo/works/thesis.pdf`.

11 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388, Switzerland, 2015. Springer International Publishing. `doi:10.1007/978-3-319-21401-6_26`.

12 Nicola Gambino and Richard Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409(1):94–109, 2008. `doi:10.1016/j.tcs.2008.08.030`.

13 Jean-Yves Girard. *Interprétation Fonctionnelle et élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Université Paris 7, 1972.

14 Allen Hatcher. *Algebraic Topology.* Cambridge University Press, Cambridge, UK, 2002. URL: `http://www.math.cornell.edu/~hatcher/AT/ATpage.html`.

15 Kuen-Bang Hou (Favonia). Covering spaces in homotopy type theory. In Maria del Mar González, Paul C. Yang, Nicola Gambino, and Joachim Kock, editors, *Extended Abstracts*

Fall 2013: Geometrical Analysis; Type Theory, Homotopy Theory and Univalent Foundations, pages 77–82, Cham, 2015. Birkhäuser. doi:10.1007/978-3-319-21284-5_15.

16 Kuen-Bang Hou (Favonia), Eric Finster, Daniel R. Licata, and Peter LeFanu Lumsdaine. A mechanization of the Blakers–Massey connectivity theorem in homotopy type theory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 565–574, New York, NY, USA, 2016. ACM. doi:10.1145/2933575.2934545.

17 Kuen-Bang Hou (Favonia) and Michael Shulman. The Seifert-van Kampen theorem in homotopy type theory. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2016.22.

18 Antonius J. C. Hurkens. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer, Berlin, Heidelberg, 1995. doi:10.1007/BFb0014058.

19 Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after Voevodsky), 2012. arXiv:1211.2851v4.

20 Nicolai Kraus. The general universal property of the propositional truncation. arXiv:1411.2682v3.

21 Nicolai Kraus, Martín Hötzel Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf type theory. arXiv:1610.03346v1.

22 Nicolai Kraus, Martín Hötzel Escardó, Thierry Coquand, and Thorsten Altenkirch. Generalizations of Hedberg's theorem. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, pages 173–188, 2013. doi:10.1007/978-3-642-38946-7_14.

23 Daniel R. Licata and Guillaume Brunerie. $\pi_n(s^n)$ in homotopy type theory. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs: Third International Conference, CPP 2013*, pages 1–16, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-03545-1_1.

24 Daniel R. Licata and Guillaume Brunerie. A cubical approach to synthetic homotopy theory. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS '15, pages 92–103, Washington, DC, USA, 2015. IEEE Computer Society. doi:10.1109/LICS.2015.19.

25 Daniel R. Licata and Eric Finster. Eilenberg-MacLane spaces in homotopy type theory. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 66:1–66:9, New York, NY, USA, 2014. ACM. doi:10.1145/2603088.2603153.

26 Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 223–232, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/LICS.2013.28.

27 Peter LeFanu Lumsdaine. Higher inductive types: a tour of the menagerie. URL: https://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/.

28 Peter LeFanu Lumsdaine. Weak ω-categories from intensional type theory. In *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2009. doi:10.1007/978-3-642-02273-9_14.

29 Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975. doi:10.1016/S0049-237X(08)71945-1.

30 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007. URL: http://www.cse.chalmers.se/~ulfn/papers/thesis.html.

31 Charles Rezk. Proof of the Blakers-Massey theorem. Prepublished, 2015. URL: http://www.math.uiuc.edu/~rezk/freudenthal-and-blakers-massey.pdf.

32 Egbert Rijke. Homotopy type theory. Master's thesis, Utrecht University, 2012. URL: http://hottheory.files.wordpress.com/2012/08/hott2.pdf.

33 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, git commit hash g662cdd8 edition, 2013.

34 Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011. doi:10.1112/plms/pdq026.

35 Benno van den Berg and Richard Garner. Topological and simplicial models of identity types. *ACM Transactions on Computational Logic*, 13(1):3:1–3:44, 2012. doi:10.1145/2071368.2071371.

36 Vladimir Voevodsky. A very short note on homotopy λ-calculus, 09 2006. URL: http://www.math.ias.edu/vladimir/files/2006_09_Hlambda.pdf.

37 Jakob von Raumer. Formalization of non-abelian topology for homotopy type theory. Master's thesis, Karlsruhe Institute of Technology, 2015. URL: http://von-raumer.de/msc-thesis.pdf.

38 Michael A. Warren. *Homotopy theoretic aspects of constructive type theory*. PhD thesis, Carnegie Mellon University, 2008. URL: http://mawarren.net/papers/phd.pdf.

Defining Trace Semantics for CSP-Agda

Bashar Igried[1]
The Hashemite University, Faculty of Prince Al-Hussein Bin Abdallah II for Information Technology, Zarqa, Jordan
bashar.igried@yahoo.com
https://orcid.org/0000-0001-6255-236X

Anton Setzer[2]
Swansea University, Dept. of Computer Science, Swansea, Wales, UK
a.g.setzer@swansea.ac.uk
https://orcid.org/0000-0001-5322-6060

Abstract

This article is based on the library CSP-Agda, which represents the process algebra CSP coinductively in the interactive theorem prover Agda. The intended application area of CSP-Agda is the proof of properties of safety critical systems (especially the railway domain). In CSP-Agda, CSP processes have been extended to monadic form, allowing the design of processes in a more modular way. In this article we extend the trace semantics of CSP to the monadic setting. We implement this semantics, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. Because of the monadic settings, some adjustments need to be made to these laws. The examples covered in this article are the laws of refinement, commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. All proofs and definitions have been type checked in Agda. Further proofs of algebraic laws will be available in the repository of CSP-Agda.

2012 ACM Subject Classification Theory of computation → Process calculi, Theory of computation → Logic and verification, Theory of computation → Type structures, Theory of computation → Operational semantics, Software and its engineering → Software verification, Software and its engineering → Semantics

Keywords and phrases Agda, CSP, Coalgebras, Coinductive Data Types, Dependent Type Theory, IO-Monad, Induction-Recursion, Interactive Program, Monad, Monadic Programming, Process Algebras, Sized Types, Universes, Trace Semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.12

Funding This research was supported by the CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962; and by CA COST Action CA15123 European research network on types for programming and verification (EUTYPES). The PhD project by B. Igried is supported by Hashemite University (FFNF150).

Acknowledgements We would like to thank the referees for their invaluable suggestions and detailed comments.

[1] Supported by Hashemite University
[2] Supported by CORCON (Correctness by Construction, FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638); COMPUTAL (Computable Analysis, FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962); CID (Computing with Infinite Data, Marie Curie RISE project, H2020-MSCA-RISE-2016-731143); CA COST Action CA15123 European research network on types for programming and verification (EUTYPES)

1 Introduction

Communicating Sequential Processes (CSP) [20, 28] is a formal specification language which was developed in order to model concurrent systems through their communications. It was developed by Hoare in 1978 [20]. It is a member of the family of process algebras. Process algebras are one of the most important concepts for describing concurrent behaviours of programs.

The starting point of this work was the modelling of processes of the European Railway Train Management System (ERTMS) in CSP by the first author. Having expertise in modelling railway interlocking systems in Agda (PhD project by Kanso [24, 25]), we thought that an interesting step forward would be to model CSP in Agda. A first step towards this project was the development of the library CSP-Agda [22, 21]. CSP-Agda represents CSP processes coinductively and in monadic form. The purpose of this article is to introduce CSP trace semantics in Agda, and carry out examples of proofs in CSP-Agda.

In CSP-Agda we developed a monadic extension of CSP, which is based on Moggi's IO monad [27]. This IO monad ($\mathsf{IO}\ A$) is currently the main construct for representing interactive programs in pure functional programming. An element of ($\mathsf{IO}\ A$) is an interactive program, which may or may not terminate, and, if it terminates, returns an element of type A. The monad provides the bind construct for combining elements of ($\mathsf{IO}\ A$): It composes a $p : \mathsf{IO}\ A$ with a function $f : A \rightarrow \mathsf{IO}\ B$ to form an element of ($\mathsf{IO}\ B$). The program is executed by first running p. If p terminates with result a, one continues running ($f\ a$). This allows to write sequences of operations in a way which looks similar to sequences of assignments in imperative style programming languages.

Hancock and the second author [18, 17, 19] have developed a version of the IO monad in dependent type theory, which we call the HS-monad. The HS-monad reduces the IO monad to coinductively defined types. An element of ($\mathsf{IO}\ A$) is either a terminated program, or it is node of a non-well-founded tree having as label a command to be executed, and as branching degree the set of responses the real world gives in response to this command. The HS-Monad has been extensively used for writing interactive programs in the paper [4] on object-based programming in Agda.

In [22], we modelled processes in a similar way as a monad and developed the library CSP-Agda. In the IO monad a program can terminate or it can issue a command and depending on the response continue. Similarly, a CSP-Agda process can either terminate, returning a result. Or it can be a tree branching over external and internal choices, where for each such choice a continuing process is given. So instead of forming processes by using high level operators, as it is usually done in process algebras, our processes are given by these atomic one step operations. The high level operators are defined operations on these processes. CSP-Agda introduces a new concept to process algebra, namely that of a monadic processes. A monadic process may run or terminate. If it terminates, it returns a value. This facilitates the combination of processes in a modular way. Processes are defined coinductively, and therefore we can introduce processes directly corecursively without having to use the recursion combinator.

One can regard process with return type A as well as possibly non-well-founded trees, with each node branching over external and internal choice, and having leaves labelled by elements of the return type A (which are terminated processes).[3]

Abel, Pientka, Thibodeau and the second author have [5, 31] developed the notion of coinductive types as being defined by their elimination rules or observations. This notion

[3] This way of viewing processes was suggested by one of the anonymous referees.

has now been implemented in Agda. It turns out that classes and objects in object oriented programming are of similar nature: Classes are defined by their methods, and therefore given by their observations. The second author [30] has used this approach in order to develop the notion of objects in dependent type theory. This has recently been substantially extended together with Abel and Adelsberger [4] to a library [3] for objects in Agda including correctness proofs, state dependent objects, server side programs, and a methodology for developing graphical user interfaces in Agda.

In CSP-Agda [22, 21] we made extensively use of the aforementioned representation of coinductive types by their elimination rules. Using a record type, we accessed directly for non-terminating processes the choice sets and corresponding subprocesses, without having to extract them first using auxiliary definitions. This gives rise to compact definitions, see for instance the definition of _+≫=_ in Subsection 4.2.1 below.

The goal of this paper is to extend CSP-Agda by adding (finite) trace semantics of CSP. Because of the monadic settings, possible return values need to be added to the traces. It turns out that in the algebraic laws of CSP, the return values of the left and right hand side of the laws are usually different. Therefore one needs to add an extra function fmap in these laws to adjust these return values. We show how to prove selected (adjusted) algebraic laws of CSP in Agda using this semantics: the laws of refinement, commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. Further proofs of algebraic laws will be available in the repository of CSP-Agda [21].

Use of literal Agda. All displayed proofs in this article have been written using literal Agda [7] (which allows to combine LaTeX-code and Agda) and have been type checked in Agda. However, as usual when presenting formal code, only the most important parts of the definitions and proofs are presented. Full versions can be found in the repository of CSP-Agda [21].

The **structure of this paper** is as follows: In Section 2, we review the process algebra CSP. In Section 3, we give a brief introduction into the type theoretic language of Agda. In Section 4, we review CSP-Agda, and introduce the CSP operators used in the examples of this paper (monadic bind, interleaving, and parallel). In Section 5 we extend CSP-Agda by adding (finite) trace semantics of CSP. In Section 6 we prove selected algebraic laws of CSP processes. In Section 7, we will look at related work, give a short conclusion, and indicate directions for future research.

2 CSP

Process algebras were initiated in 1982 by Bergstra and Klop [9] in order to provide a formal semantics to concurrent systems. A "process" is a representation of the behaviour of a concurrent system. "Algebra" means that the system is dealt with in an algebraic and axiomatic way [8]. In this article we represent a process algebra in the interactive theorem prover Agda in order to prove properties of processes. The process algebra chosen is Communicating Sequential Processes (CSP). CSP [20, 28, 29] was developed by Hoare in 1978 [20].

Processes in CSP form a labelled transition system, where the one step transition is written as

$$P \xrightarrow{\mu} Q \quad \text{where } P, Q \text{ are processes and } \mu \text{ is an action,}$$

which means that process P can evolve to process Q by event μ. The event μ can be a label, the silent transition τ, or the termination event $\checkmark$. In case of the label $\checkmark$, Q will always be the specific process STOP. Using standard CSP syntax, the process $(a \to P)$ is the process which has an only transition $(a \to P) \xrightarrow{a} P$.[4] As an example, we give here the execution of the process $(a \to b \to \mathsf{STOP})$:

$$(\ a \to b \to \mathsf{STOP}) \xrightarrow{a} (b \to \mathsf{STOP}) \xrightarrow{b} \mathsf{STOP}$$

The operational semantics of CSP defines processes as states, and defines the transition rules between the states using firing rule. In CSP-Agda [22, 21] we introduced the firing rules for CSP operators (taken from [29]), and modelled them in Agda. We followed the version of CSP used in [29, 28]. All rules (as well those in this paper) are taken from [29]. In the rules we follow the convention of [29] that a ranges over Label $\cup \{\checkmark\}$ and μ over Label $\cup \{\checkmark, \tau\}$. $A^{\checkmark}$ denotes $A \cup \{\checkmark\}$.

In the following table, we list the constructs for forming CSP processes. Here Q represent CSP processes (Page numbers refer to [29] where the constructs are introduced):

Q ::= STOP	STOP	p.6
| SKIP	SKIP	p.11
| prefix	$a \to Q$	p.6
| external choice	$Q \,\Box\, Q$	p.18
| internal choice	$Q \sqcap Q$	p.22
| hiding	$Q \setminus a$	p.53
| renaming	$Q[R]$	p.60
| parallel	$Q\ {}_X\|_Y\ Q$	p.29
| interleaving	$Q \mid\mid\mid Q$	p.43
| interrupt	$Q \bigtriangleup Q$	p.70
| composition	$Q \mathbin{;} Q$	p.67

There are as well indexed versions of $\Box$, $\sqcap$, $\|$, $|||$. They are indexed over finite sets, and therefore can be reduced to the binary case.

3 Agda

In this chapter we introduce the main concepts of Agda [6, 10], a more extensive introduction can be found in [22].

Agda is based on dependent type theory. There are several levels of types in Agda, the lowest is for historic reasons called Set. Types in Agda are given as dependent function types, and inductive types. In addition, there exist record types (which are in the newer approach used as well for defining coinductive types) and a generalisation of inductive-recursive and inductive-inductive definitions. Inductive data type are dependent versions of algebraic data types as they occur in functional programming. Inductive data types are given as sets A together with constructors which are strictly positive in A. For instance, the set of vectors (i.e. lists of fixed length) of elements of A and of length n is given as

```
data Vec (A : Set) : ℕ → Set where
  []    : {n : ℕ} → Vec A zero
  _::_ : {n : ℕ} (a : A) (l : Vec A n) → Vec A (suc n)
```

[4] In Agda we use an arrow which looks similar to the one used for the function type, but is a different Unicode character. The reason for this choice is to be as much as possible in accordance with standard CSP syntax.

Here $\{n : \mathbb{N}\}$ is an implicit argument. Implicit arguments are omitted, provided they can be uniquely determined by the type checker. We can make a hidden argument explicit by writing for instance ([] $\{n\}$) for the application of [] to the hidden argument n. The symbol _::_ is Agda's notation for mixfix symbols. The arguments of a mixfix operator are denoted by underscore (_). The expression $a :: l$ stands for (_::_ a l).

The above definition introduces a new type Vec : $(A :$ Set$) \to \mathbb{N} \to$ Set, where (Vec A n) is a type of vectors of type A of length n. A is a parameter, so the constructors always refer to the same parameter A. The variable n is an index, and constructors refer to different indices. The vectors have constructors [] and _::_. The elements of (Vec A n) are those constructed from applying these constructors. Therefore we can define functions by case distinction on these constructors using pattern matching. The following defines the sum of elements of a vector of type $\mathbb{N}$:

```
sum : ∀ {n} → Vec ℕ n → ℕ
sum []       = 0
sum (n :: l) = n + sum l
```

Here we used the notation $\forall \{n\} \to \cdots$, which stands for $\{n : A\} \to \cdots$, where A (here $\mathbb{N}$) can be inferred by Agda. Nested patterns are allowed. The coverage checker checks completeness and the termination checker checks that the recursive calls follow a schema of extended primitive recursion.

In this paper we use the approach of defining coinductive types in Agda by their elimination rules as introduced in [5, 31]. The standard example is the set of streams:

```
record Stream (i : Size) : Set where
  coinductive
  field
    head : ℕ
    tail  : {j : Size< i} → Stream j
```

If we first ignore the arguments Size, Size<, which will be discussed below, we see that the type Stream is given as a record type in Agda. It is defined coinductively by its observations head, tail. So we have if a : Stream i then head a : $\mathbb{N}$ and tail a : $\{j$: Size< $i\} \to$ Stream j. Elements of Stream are defined by copattern matching, i.e. by determining the result of applying head, tail to them. A simple (non-recursive) operation is the function cons for adding a new element in front of a stream (the symbol ↑ will be explained when discussing Size below):

```
cons : {i : Size} → ℕ → Stream i → Stream (↑ i)
head (cons n s) = n
tail  (cons n s) = s
```

Functions introduced by the principle of guarded recursion [11] or primitive corecursion can only make corecursive calls to the same functions applied to arbitrary arguments. Especially, no functions can be applied to the corecursive calls. However, there are no restrictions on the arguments, the corecursive function calls can be applied to. As an example we give the pointwise addition of two streams:

```
_+s_ : ∀ {i} → Stream i → Stream i → Stream i
head (s +s s') = head s +  head s'
tail  (s +s s') = tail   s +s tail   s'
```

_+s_ makes a corecursive call to (tail s +s tail s'). Note that s, s' are arguments of _+s_, so we can apply tail to them freely.

Without the guarded recursion restriction, one could define non productive definitions, e.g. define tail $(f\,x)$ = tail $(f\,x)$. However, the guardedness restriction makes it difficult to define streams in a modular way, since we cannot in a corecursive call refer to other functions for forming streams at all, although many operations will not cause problems. Therefore Abel has introduced sized types [1, 2] in the context of coinductive types, which allow to apply size preserving and size increasing functions to corecursive calls.

Sizes are essentially ordinals (without infinite branching one can think of them as natural numbers), however there is an additional infinite size ∞. We have as operations for forming sizes the infinite size ∞, the successor operation on sizes $\uparrow$, and have the type of sizes less than i denoted by (Size< i).

For ordinal sizes $i \neq \infty$, a stream s : Stream i allows up to i applications of tail. The true streams is the set Stream ∞ and s : Stream ∞ allows arbitrary many applications of tail. When defining an element f : (i : Size) $\rightarrow A\ i \rightarrow$ Stream i by corecursion, (tail $(f\ i\ a)$ $\{j\}$) must be an element of size $\geq j$ which can refer to a corecursive call $(f\ j\ a')$, and we can apply functions to it as long as the resulting size is $\geq j$. Elimination on the corecursive call is prevented, since we do not have access to any size $< j$. However, we can apply size preserving and size increasing functions to the corecursive call. This guarantees that streams are productive. We have ∞ : Size< ∞, so a corecursive definition of elements of (Stream ∞) can refer to itself.

Agda offers let and where expressions in order to declare a local definition. In comparison, where expressions allow a pattern matching or recursive function, whereas pattern matching and recursive functions are not allowed in let expressions. In Agda the let expressions can be represented as follows:

```
let
 a₁  :  A₁
 a₁  =  s₁
 a₂  :  A₂
 a₂  =  s₂
    ...
 an  :  An
 an  =  sₙ
in t
```

In the above definition, we use let expressions in order to introduce new local constants:

```
a₁  :  A₁ s.t. a₁  =  s₁,
a₂  :  A₂ s.t. a₂  =  s₂,
    ...
an  :  An s.t. an  =  sₙ
```

The syntax for where is similar, except that the auxiliary definitions introduced by where occur after the main definition they are used in, whereas for let they occur before it.

4 The Library CSP-Agda

In this section we repeat the main definition of processes in CSP-Agda from [22]. The reader might consult that paper for a more detailed motivation of the definitions in CSP-Agda.

4.1 Representing CSP Processes in Agda

As outlined before, we represent processes in Agda in a monadic way. Therefore, a process P : Process A is either a terminating process (terminate a), which has return value a : A, or it is process (node Q) which progresses. Here Q : Process+ A, where (Process+ A) is the type of progressing processes. A progressing process can proceed at any time with labelled transitions (external choices), silent transitions (internal choices), or ✓-events (termination). After a ✓-event, the process becomes deadlocked, so there is no need to determine the process after that event. We will however add a return value a : A to ✓-events. Note that there is a subtle difference between terminated processes and processes with termination events (see [23] for full details.[5])

Elements of (Process+ A) are therefore determined by

(1) an index set E of external choices, and for each external choice e the Label (Lab e) and the next process (PE e);

(2) an index set of internal choices I, and for each internal choice i the next process (PI i); and

(3) an index set of termination choices T corresponding to ✓-events, and for each termination choice t the return value PT t : A.

In addition we add in CSP-Agda a type (Process∞ A). This makes it easy to define processes by guarded recursion, when the right hand side is defined directly and without having to define all 8 components[6] of (Process+ A). Furthermore, in order to display processes, we add eliminators Str+ and Str∞ to (Process+ A) and (Process∞ A), respectively. They return a string representing the process. In case of (Process∞ A), this cannot be reduced to the string component of (Process+ A): in order to do this one would need a smaller size, which we do not have in general for arbitrary sizes.

We model the sets of external, internal, and termination choices as elements of an inductive-recursively defined universe Choice. Elements c of Choice are codes for finite sets, and (ChoiceSet c) is the set it denotes. In addition we define a string (choice2Str c) representing c, and a function choice2Enum which computes from c a list of all choices. This can be used to print a list of choices, for instance for testing or simulating CSP processes.

We require as well that the set of return values are elements of Choice. This allows us to print the result returned when a process terminates. However, for the return types it is not needed that they are finite sets. So one could use a different universe for the return values of processes, which would allow for instance the set of natural numbers as a return type.

The resulting code for processes in Agda is as follows[7]:

[5] For instance, let P_0 have a τ-transition to (terminate a), and an l'-transition to STOP. Let P_1 having a ✓-event with return value a and the same l'-transition to STOP. Process (P_0 ||| R) can have a τ transition to ((terminate a) ||| R), a state in which it can refuse l'. Process (P_1 ||| R) cannot execute the ✓-transition, since it needs to synchronise with a ✓-transition for R. It is stable, and cannot refuse l'.

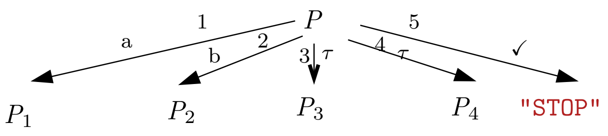
[6] The 8th component Str+ is introduced in the next sentence.

[7] Both occurrences of coinductive are needed by the current version of Agda; one could argue that in case of Process+ Agda should allow to omit it, allowing η-equality for Process+.

```
mutual
  record Process∞ (i : Size) (c : Choice) : Set where
    coinductive
    field
      forcep : {j : Size< i} → Process j c
      Str∞ : String

  data Process (i : Size) (c : Choice) : Set where
    terminate : ChoiceSet c   → Process i c
    node      : Process+  i c → Process i c




  record Process+ (i : Size) (c : Choice) : Set where
    constructor process+
    coinductive
    field
      E    : Choice
      Lab  : ChoiceSet E → Label
      PE   : ChoiceSet E → Process∞ i c
      I    : Choice
      PI   : ChoiceSet I  → Process∞ i c
      T    : Choice
      PT   : ChoiceSet T → ChoiceSet c
      Str+ : String
```

So an element of Process+ is defined by copattern matching, e.g. by determining its components E, Lab, PE, etc. Note that the Agda notation E : Choice means that if we apply E to an element of (Process+ i c) we obtain an element of Choice, so the full type is Process+ i c → Choice. Therefore, an element of Q : Process+ is determined by determining E Q : Choice, Lab Q l : Label , etc. An example of a process is as follows:

```
P   =     node Q   : Process String  where
E   Q    =  code for {1, 2}       I    Q    =   code for {3, 4}
T   Q    =  code for {5}
Lab Q 1  =  a                     Lab Q  2  =  b                 PE Q 1  =  P₁
PE  Q 2  =  P₂                    PI  Q  3  =  P₃                PI Q 4  =  P₄
PT  Q 5  =  "STOP"
```



The universe of choices is given by a set Choice of codes for choice sets, and a function ChoiceSet, which maps a code to the choice set it denotes. Universes were introduced by Martin-Löf (e.g. [26]) in order to formulate the notion of a type consisting of types. Universes are defined in Agda by an inductive-recursive definition [13, 12, 14, 15]: we define inductively the set of codes in the universe while recursively defining the decoding function.

We give here the code expressing that Choice is closed under fin, ⊎', ×', subset', Σ', and namedElements, which correspond to the set operations Fin, ⊎, ×, subset, Σ, and NamedElements. Here (fin n) denotes the set (Fin n), which is the finite set having n elements. The element (Σ' a b) denotes the set (Σ[x ∈ ChoiceSet a] (ChoiceSet (b x))), where (Σ'[x ∈ A] B) is the set of pairs (x , y) where x : A and y : B, and B might depend on x.[8] The element (namedElements l) denotes the type (NamedElements l), which is essentially (Fin (length l)).[9] The function choice2Str will for elements of this set print the nth element of l, giving them more meaningful names.[10] We do not equate (NamedElements l) with (Fin (length l)). This facilitates type inference.[11]

The set (subset A f) is the set of $a : A$ such that (f a) is true. The definition of ChoiceSet is as follows:

```
data Choice : Set where
  fin      : ℕ → Choice
  _⊎'_  : Choice → Choice → Choice
  _×'_  : Choice → Choice → Choice
  namedElements : List String → Choice
  subset' : (E : Choice) → (ChoiceSet E → Bool)
                              → Choice
  Σ'       : (E : Choice) → (ChoiceSet E → Choice)
                              → Choice

ChoiceSet : Choice → Set
ChoiceSet (fin n)      = Fin n
ChoiceSet (s ⊎' t)    = ChoiceSet s ⊎ ChoiceSet t
ChoiceSet (E ×' F) = ChoiceSet E × ChoiceSet F
ChoiceSet (namedElements s) = NamedElements s
ChoiceSet (subset' E f) = subset (ChoiceSet E) f
ChoiceSet (Σ' A B) = Σ[ x ∈ ChoiceSet A ] ChoiceSet (B x)

choice2Str : {c : Choice} → ChoiceSet c → String
choice2Str {fin n} m = showℕ (toℕ m)
  ...

choice2Enum : (c : Choice) → List (ChoiceSet c)
choice2Enum (fin n) = fin2Option0 n
  ...
```

[8] The type (A ×' B) has essentially the same elements as (Σ[$x \in A$] B) for some fresh x. However, if we know a type C is of the form (A ×' B), we can pattern match and obtain A and B from it, whereas from the form (Σ[$x \in A$] B) we can only infer A because B is a function type. This requires to make frequently hidden arguments A and B explicit.

[9] It was suggested to us to let NamedElements depend on an n and an element of (Vec String n). But those two elements are just a long form for writing an element of (List String). The only advantage of Vec is that the standard library has a lookup function for it, which should be added as well for List.

[10] As pointed out by one of the anonymous referees, (fin n) is redundant and could be replace by (namedElements l) for some suitable l. We keep it because when developing proofs, (fin n) behaves better because one does not have length expressions of the form (length l) for some long expression l.

[11] Assume c is a hidden argument of type Choice, and l : ChoiceSet c. If we equated (NamedElements l) with (Fin (length l)), then from the type of l we could not infer c, since in case l : Fin n we could have c = fin n and c = namedElements l for some l. Therefore, one would need to make the hidden argument explicit.

4.2 Definition of the Monadic Bind, Interleaving, and the Parallel Operators

We introduce the three operators, for which we will prove algebraic properties in this paper: monadic bind, interleaving and the parallel operator. Monadic bind and interleaving have already been defined in [22], and are repeated here to make it easier to follow the proofs of the algebraic laws.

As in [22], when defining operators on processes, we introduce in most cases simultaneously operators on the three categories of processes Process∞, Process, and Process+. We use qualifiers ∞, p, + attached to the operators for refer to the 3 categories of processes, respectively. For infix operators they will occur before the infix symbol if they refer to the first argument, otherwise after the infix symbol. Note that we deviate from [22], where all qualifiers were put after the symbol. We often omit p. We have as well a string forming operation indicated by Str. For some binary operators we need versions where the arguments are from different categories of processes, in which case we add two qualifiers to the operators, one before and one after the operator, and sometimes we need even 3 or more qualifiers. We will only present the main cases of the operators. Especially, we will usually omit the functions involving Process∞, which follow usually the same pattern (an example can be found in Subsection 4.2.1 below when defining _∞≫=_). The full code can be found at [21].

4.2.1 The Monadic Bind Operator

In our article [22] we introduced the monadic bind operation. In Section 6.2 we will prove the monadic laws and therefore will briefly repeat the definition of the monadic bind. A more extensive motivation can be found in [22]. The monadic bind ($P \gg= Q$) allows to compose two processes P and Q while allowing the second process depend on the return type c_0 of P. So Q has an an extra argument of the return type (ChoiceSet c_0).

Let us consider first the version _+≫=_ where the first process is an element of set of progressing processes Process+. The transitions of (P +≫= Q) are as follows: First they follow external and internal choices of P. If P is the terminated process with return type a, the process continues as process ($Q\ a$). A special case is a termination event in P with return value a. Following the operational semantics of CSP, (P +≫= Q) has in this case an internal choice (i.e. a τ-transition) to process ($Q\ a$). In total, (P +≫= Q) has two possible internal choice events, namely internal choices of P and termination events of P. It has no termination events.

In case of the monadic bind _≫= on Process, we have a special case, when P = terminate x. In this case $P \gg= Q$ is equal to ($Q\ x$) (one needs to apply forcep in order to obtain an element of Process). This is different from termination events for P, where a silent transition is required before obtaining ($Q\ x$). In case of progressing processes $P \gg= Q$ makes a direct call to _+≫=_. The function _∞≫=_ makes as well a direct call to _≫=_.

The full definition of monadic bind is as follows (the symbol "()" in the definition of PT below denotes the empty case distinction on the empty type (ChoiceSet ∅'))[12]:

[12] Note that _++s_ is concatenation of string.

```
_≫=Str_ : {c₀ : Choice} → String
                → (ChoiceSet c₀ → String) → String
s ≫=Str f = s ++s ";" ++s choice2Str2Str f

mutual
  _∞≫=_ : {i : Size} → {c₀ c₁ : Choice}
                → Process∞ i c₀
                → (ChoiceSet c₀ → Process∞ i c₁)
                → Process∞ i c₁
  forcep (P ∞≫= Q)        = forcep P ≫= Q
  Str∞        (P ∞≫= Q) = Str∞ P ≫=Str (Str∞ ∘ Q)

  _≫=_ : {i : Size} → {c₀ c₁ : Choice}
                → Process i c₀
                → (ChoiceSet c₀ → Process∞ (↑ i) c₁)
                → Process i c₁
  node             P ≫= Q  = node   (P +≫= Q)
  terminate x        ≫= Q  = forcep (Q x)

  _+≫=_ : {i : Size} → {c₀ c₁ : Choice}
                → Process+ i c₀
                → (ChoiceSet c₀ → Process∞ i c₁)
                → Process+ i c₁
  E   (P +≫= Q)          = E   P
  Lab (P +≫= Q)          = Lab P
  PE  (P +≫= Q) c        = PE P c ∞≫= Q
  I   (P +≫= Q)          = I P ⊎' T P
  PI  (P +≫= Q) (inj₁ c) = PI  P c ∞≫= Q
  PI  (P +≫= Q) (inj₂ c) = Q (PT P c)
  T   (P +≫= Q)          = ∅'
  PT  (P +≫= Q) ()
  Str+ (P +≫= Q) = Str+ P ≫=Str (Str∞ ∘ Q)
```

4.2.2 The Interleaving Operator

The interleaving operator executes the external and internal choices of its arguments P and Q completely independently of each other. The CSP rules are as follows (having two conclusions of a rule is an abbreviation for two rules having the same premises: one deriving the first and one deriving the second conclusion):

$$\frac{P \xrightarrow{\checkmark} \bar{P} \qquad Q \xrightarrow{\checkmark} \bar{Q}}{P \mid\mid\mid Q \xrightarrow{\checkmark} \bar{P} \mid\mid\mid \bar{Q}} \qquad\qquad \frac{P \xrightarrow{\mu} \bar{P}}{\begin{array}{c} P \mid\mid\mid Q \xrightarrow{\mu} \bar{P} \mid\mid\mid Q \\ Q \mid\mid\mid P \xrightarrow{\mu} Q \mid\mid\mid \bar{P} \end{array}}\ \mu \neq \checkmark$$

The definition of the two main cases in CSP-Agda is as follows:

```
_|||_ : {i : Size} → {c₀ c₁ : Choice} → Process i c₀
          → Process i c₁ → Process i (c₀ ×' c₁)
node P ||| node Q   = node (P +|||+ Q)
terminate a ||| Q   = fmap (λ b → (a ,, b)) Q
P |||   terminate b = fmap (λ a → (a ,, b)) P

_+|||+_ : {i : Size} → {c₀ c₁ : Choice}
  → Process+ i c₀ → Process+ i c₁
  → Process+ i (c₀ ×' c₁)
E   (P +|||+ Q)            = E P ⊎' E Q
Lab (P +|||+ Q) (inj₁ c)   = Lab P c
Lab (P +|||+ Q) (inj₂ c)   = Lab Q c
PE  (P +|||+ Q) (inj₁ c)   = PE P c ∞|||+ Q
PE  (P +|||+ Q) (inj₂ c)   = P +|||∞ PE Q c
I   (P +|||+ Q)            = I P ⊎' I Q
PI  (P +|||+ Q) (inj₁ c)   = PI P c ∞|||+ Q
PI  (P +|||+ Q) (inj₂ c)   = P +|||∞ PI Q c
T   (P +|||+ Q)            = T P ×' T Q
PT  (P +|||+ Q) (c ,, c₁)  = (PT P c ,, PT Q c₁)
Str+ (P +|||+ Q)           = Str+ P |||Str Str+ Q
```

When processes P and Q have not terminated, then $(P\,|||\,Q)$ will not terminate. The external choices are the external choices of P and Q. The labels are the labels from the processes P and Q, and we continue recursively with the interleaving combination. The internal choices are defined similarly. A termination event can happen only if both processes have a termination event.

If one process terminates but the other not, the rules of CSP express that one continues as the other process, until it has terminated. We can therefore equate, if P has terminated, $(P\,|||\,Q)$ with Q. However, we record the result obtained by P, and therefore apply fmap to Q in order to add the result of P to the result of Q when it terminates. Here (fmap f P) is the process obtained from P by applying f to any termination results.

If both processes terminate with results a and b, then, the interleaving combination terminates with result $(a\ ,,\ b)$, since (fmap (λ b → (a ,, b)) (terminate b)) evaluates to this expression.

4.2.3 The Parallel Operator

The parallel operator gives the possibility to enforce two processes to work together and interact through synchronous events. For each of the two processes sets of labels A, B are given. For labels which are not in the intersection, both processes can execute independently, as long as their processes are in A or B, respectively. For labels in the intersection, both processes need to synchronise on that event. The transition rules for the parallel operator are as follows:

$$\frac{P \xrightarrow{a} \bar{P} \qquad Q \xrightarrow{a} \bar{Q}}{P\ {}_A\|_B\ Q \xrightarrow{a} \bar{P}\ {}_A\|_B\ \bar{Q}}\ [\, a \in A^{\checkmark} \cap B^{\checkmark}\,] \qquad \frac{P \xrightarrow{\mu} \bar{P}}{\begin{array}{c} P\ {}_A\|_B\ Q \xrightarrow{\mu} \bar{P}\ {}_A\|_B\ Q \\ Q\ {}_B\|_A\ P \xrightarrow{\mu} Q\ {}_B\|_A\ \bar{P} \end{array}}\ [\, \mu \in ((A \cup \tau)\backslash B)]$$

In CSP-Agda we define the parallel operator as follows: Assume $A\ B$: Label → Bool, which determine the label sets A and B as above. The external choices of (P [A]+||+[B] Q) are:

- The external choices of c : E P, for which the label in P is in $(A \setminus B)$, i.e. such that $((A \setminus B)$ (Lab $P\ c$)) = true. Here $(A \setminus B)$: Label → Bool is defined by $(A \setminus B)\ b$ = true if and only if $A\ b$ = true and $B\ b$ = false. For such c the label for this external choice is the label of P for choice c, and the process obtained following this transition is the parallel construct applied to (PE $P\ c$) and Q.
- The external choices of c : E Q, for which the label in Q is in $(B \setminus A)$, with similar definitions of the label and next process obtained.
- The combined external choices for P and Q, i.e. pairs (e_1 , e_2) s.t. e_1 : E P and e_2 : E Q, and s.t. their labels are equal, and the labels are in A and in B, i.e. such that

  ((Lab $P\ e_1$ ==l Lab $Q\ e_2$) ∧ A (Lab $P\ e_1$) ∧ B (Lab $Q\ e_2$)) = true

  Here _==l_ is Boolean valued equality on Labels, and _∧_ is Boolean valued conjunction. The label for this external choice is the label of P (which is w.r.t. _==l_ equal to the corresponding label of Q). The process obtained when following this external choice is the parallel construct applied to the result of following the external choices in both P and Q.

Furthermore

- The internal choices are the internal choices of P and Q, and the process obtained when following those transitions is obtained by following the corresponding transition in process P or Q, respectively.
- A termination event can happen only if both processes have a termination event. If they terminate with results a and b, then the parallel combination terminates with result (a ,, b). Therefore the result type of the parallel construct is the product of the result type of the first and second process.

In order to define the above we use the subset' constructor of Choice which has equality rule

```
ChoiceSet (subset' E f) = subset (ChoiceSet E ) f
```

Here, (subset $a\ f$) is the set of pairs (sub $a\ b$) such that a : A and b : T ($f\ a$), i.e. it is essentially the set $\{a : A \mid f\ a = \text{true}\}$. We have T : Bool → Set, such that (T true) is provable and (T false) is empty, i.e. not provable.

The definition of the parallel operator in CSP-Agda for Process+ is as follows:

```
_[_]+||+[_]_ : {i : Size} → {c₀ c₁ : Choice}
               → Process+ i c₀
               → (A B : Label → Bool)
               → Process+ i c₁
               → Process+ i (c₀ ×' c₁)
E   (P [ A ]+||+[ B ] Q) = subset' (E P) ((A \ B) ∘ (Lab P)) ⊎'
                           subset' (E Q) ((B \ A) ∘ (Lab Q)) ⊎'
                           subset' (E P ×' E Q)
                               (λ {(e₁ ,, e₂)
                                 → Lab P e₁ ==l Lab Q e₂ ∧ A (Lab P e₁) ∧ B (Lab Q e₂)})
```

```
Lab  (P [ A ]+||+[ B ] Q) (inj₁ (inj₁ (sub c p)))   = Lab P c
Lab  (P [ A ]+||+[ B ] Q) (inj₁ (inj₂ (sub c p)))   = Lab Q c
Lab  (P [ A ]+||+[ B ] Q) (inj₂ (sub (c₀ ,, c₁) p)) = Lab P c₀
PE   (P [ A ]+||+[ B ] Q) (inj₁ (inj₁ (sub c p)))   = PE P c [ A ]∞||+[ B ] Q
PE   (P [ A ]+||+[ B ] Q) (inj₁ (inj₂ (sub c p)))   = P      [ A ]+||∞[ B ] PE Q c
PE   (P [ A ]+||+[ B ] Q) (inj₂ (sub (c₀ ,, c₁) p)) = PE P c₀ [ A ]∞||∞[ B ] PE Q c₁
I    (P [ A ]+||+[ B ] Q)                           = I P ⊎' I Q
PI   (P [ A ]+||+[ B ] Q) (inj₁ c)                  = PI P c  [ A ]∞||+[ B ] Q
PI   (P [ A ]+||+[ B ] Q) (inj₂ c)                  = P       [ A ]+||∞[ B ] PI Q c
T    (P [ A ]+||+[ B ] Q)                           = T P ×' T Q
PT   (P [ A ]+||+[ B ] Q) (c₀ ,, c₁)                = (PT P c₀ ,, PT Q c₁)
Str+ (P [ A ]+||+[ B ] Q)                           = Str+ P [ A ]||Str[ B ] Str+ Q
```

When defining the parallel construct for elements of Process, we need to deal with the case one of the processes is the terminated process. As for _|||_, one continues in this case as the other process, until it has terminated. However, in case of P having terminated, only labels in the set $(B \setminus A)$ are allowed for Q. We can therefore equate, if P has terminated, $(P$ [A]+||+[B] $Q)$ with $(Q \upharpoonright (B \setminus A))$. Here for a process P' and a set of labels A' the process $P \upharpoonright A'$ is the process obtained by restricting the external transitions to those with label in A'. Note that this is different from hiding, external transitions with labels not in A' are not turned into τ-transitions. As for _|||_, we need to record the result obtained by P, and therefore apply fmap to Q in order to add the result of P to the result of the restriction of Q, when it terminates.

The definition of the parallel operator for Process is therefore as follows:

```
_[_]|||[_]_ : {i : Size} → {c₀ c₁ : Choice}
              → Process i c₀
              → (A B : Label → Bool)
              → Process i c₁
              → Process i (c₀ ×' c₁)
node P [ A ]|||[ B ] node Q       = node (P [ A ]+||+[ B ] Q)
terminate a [ A ]|||[ B ] Q       = fmap (λ b → (a ,, b)) (Q ↾ (B \ A))
P        [ A ]|||[ B ] terminate b = fmap (λ a → (a ,, b)) (P ↾ (A \ B))
```

5 Defining Trace Semantics for CSP-Agda

In CSP, traces of a process are the sequences of actions or labels of external choices a process can perform. Since the processes in CSP, are non-deterministic, a process can follow different traces during its execution. The trace semantics of a process is the set of its traces.

Since in CSP-Agda processes are monadic, we need to record, in case after following a trace we obtain a terminated process, the result returned by the process following this trace. So we add a possible element of the result set to the trace. We can use for the set of possible elements the set (Maybe (ChoiceSet c)). Here the type (Maybe A) has elements (just a) for $a : A$, denoting defined elements, and an undefined element nothing. So (just a) denotes that the process has terminated with result a, whereas nothing means that it has not terminated (or more precisely not been determined to have terminated[13]).

[13] A process having trace l with result (just a) has as well trace l with result nothing, see below.

Taking this together, we obtain that traces are given by a list of labels and an element of (Maybe (ChoiceSet c)). We define the set of traces (Tr l m P) as a predicate which determines for a process the lists of labels l and elements m : Maybe (ChoiceSet c), which form a trace. We define as well traces (Tr+ l m P) and (Tr∞ l m P) for processes in Process+ and Process∞, respectively.

In the trace semantics of CSP, a process having a termination event has two traces, the empty list, and the list consisting of a ✓-event. In order to be consistent with CSP, we will add therefore in case of a termination event or terminated process two traces: the empty list together with possible return value nothing, and with possible return value (just a) for the return value a.

For an element of (Process+ ∞ c) we obtain the following traces:

- The empty trace without termination is a trace of any process, and we denote the proof by empty.
- If a process P has external choice x, then from every trace for the result of following this choice, consisting of a list of labels l and a possible result res, we obtain a trace of P consisting of the result of adding in front of l the label of that external choice, and of the same possible result res. The resulting proof will be denoted by (extc l res x tr).
- Internal choices are ignored in traces. Therefore if a process P has an internal choice x, every trace of the result of following this choice is as well a trace of P. The proof is denoted by (intc l res x tr)
- If a process has a termination event x with return value t, then the empty trace with termination choice (just t) is a trace of process, having proof (terc x).

The corresponding definition for Process+ is as follows:

```
data Tr+ {c : Choice} : (l : List Label) → Maybe (ChoiceSet c) → (P : Process+ ∞ c)
              → Set where
  empty : {P : Process+ ∞ c} → Tr+ [] nothing P
  extc : {P : Process+ ∞ c} → (l : List Label) → (res : Maybe (ChoiceSet c))
              → (x : ChoiceSet (E P)) → Tr∞ l res (PE P x) → Tr+ (Lab P x :: l) res P
  intc : {P : Process+ ∞ c} → (l : List Label) → (res : Maybe (ChoiceSet c))
           → (x : ChoiceSet (I P)) → Tr∞ l res (PI P x) → Tr+ l res P
  terc : {P : Process+ ∞ c} → (x : ChoiceSet (T P)) → Tr+ [] (just (PT P x)) P
```

In case of Process we need to consider the termination events:

- The terminated process has two traces, namely the empty list of labels [] with termination event nothing, and the same list but with termination event (just x), where x is the return value.
- The traces of a non-terminated process are the traces of the corresponding element of Process+.

We obtain the following definition of the traces of Process:

```
data Tr {c : Choice} : (l : List Label) → Maybe (ChoiceSet c) → (P : Process ∞ c)
           → Set where
  ter : (x : ChoiceSet c) → Tr [] (just x) (terminate x)
  empty : (x : ChoiceSet c) → Tr [] nothing (terminate x)
  tnode : {l : List Label} → {x : Maybe (ChoiceSet c)} → {P : Process+ ∞ c}
             → Tr+ {c} l x P → Tr l x (node P)
```

Finally the traces for Process∞ are just the traces of the underlying Process:

```
record Tr∞ {c : Choice} (l : List Label) (res : Maybe (ChoiceSet c))
           (P : Process∞ ∞ c) : Set where
  coinductive
  field
    forcet : Tr l res (forcep P)
```

In CSP, a process P refines a process Q, written $(P \sqsubseteq Q)$ if and only if any observable behaviour of Q is an observable behaviour of P, i.e. if $traces(Q) \subseteq traces(P)$:

```
_⊑_ : {c : Choice} (P : Process ∞ c) (Q : Process ∞ c) → Set
_⊑_ {c} P Q = (l : List Label) → (m : Maybe (ChoiceSet c)) → Tr l m Q → Tr l m P
```

Two processes P, Q are equal w.r.t. trace semantics, written $P \equiv Q$, if they refine each other, i.e. if $traces(P) = traces(Q)$:

```
_≡_ : {c₀ : Choice} → (P Q : Process ∞ c₀) → Set
P ≡ Q = P ⊑ Q × Q ⊑ P
```

6 Proof of the Algebraic Laws

Trace equivalence gives rise to algebraic laws for individual operators, and also concerning the relationships between different operators. Laws for individual operators are concerned with general algebraic properties such as commutativity and associativity of operators, the identification of zeros and units for specific operators, and idempotence of operators; these properties allow a process to be composed in any order, and allow process descriptions to be simplified. An example of the relationship between different operators is the expansion of the interleaving of processes, each of which is introduced by an event prefix, into a prefix choice process. We will present examples of how to prove algebraic laws of CSP in Agda using this semantics. The examples covered in this article are commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. Further examples will be available in the repository of CSP-Agda.

6.1 Proof of the Laws of Refinement

The refinement relation is reflexive, anti-symmetric and transitive, i.e. fulfils the following laws:

$$P \sqsubseteq P$$
$$P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_0 \Rightarrow P_0 = P_1$$
$$P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_2 \Rightarrow P_0 \sqsubseteq P_2$$

These laws are a direct consequence of the fact that $P \sqsubseteq Q$ means essentially $traces(Q) \subseteq traces(P)$ and $P \equiv Q$ means $traces(P) = traces(Q)$:

```
refl⊑ : {c : Choice} (P : Process ∞ c) → P ⊑ P
refl⊑ {c} P l m x = x

antiSym⊑ : {c₀ : Choice} → (P Q : Process ∞ c₀) → P ⊑ Q → Q ⊑ P → P ≡ Q
antiSym⊑ P Q PQ QP = PQ , QP

trans⊑ : {c : Choice}(P : Process ∞ c)(Q : Process ∞ c)(R : Process ∞ c)
         → P ⊑ Q → Q ⊑ R → P ⊑ R
trans⊑ {c} P Q R PQ QR l m tr = PQ l m (QR l m tr)
```

6.2 Proof of the Monadic Laws

We defined processes in a monadic way, and will in this section prove the monad laws for processes.

In functional programming, a monad is given by a functor M together with morphisms $\gg\!= : \mathsf{M}\ A \to (A \to \mathsf{M}\ B) \to \mathsf{M}\ B$ and $\mathsf{return} : A \to \mathsf{M}\ A$ such that the following laws hold:

$$\begin{aligned}
\mathsf{return}\ a \gg\!= f &= f\ a\\
p \gg\!= \mathsf{return} &= p\\
(p \gg\!= f) \gg\!= g &= p \gg\!= (\lambda\, x \to f\ x \gg\!= g)
\end{aligned}$$

For each monadic law we have to prove 2 directions, ("⊑" and "⊒"). Furthermore the laws need to be shown for Process+, Process and Process∞. We will present only one direction and one version of the processes for each law. Since proofs of _≡_ just follow from the left to right and right to left refinement, we will present this proof only for the first monadic law.

The proof of the first monadic law is trivial since (terminate $a \gg\!= P$) is definitionally equal to P:

```
monadLaw₁ : {c₀ c₁ : Choice} (a : ChoiceSet c₀)(P : ChoiceSet c₀ → Process ∞ c₁)
            → (terminate a >>= P) ⊑ P a
monadLaw₁ a P l m q = q

≡monadLaw₁ : {c₀ c₁ : Choice} (a : ChoiceSet c₀)(P : ChoiceSet c₀ → Process ∞ c₁)
  → (P a) ≡ (terminate a >>= P)
≡monadLaw₁ {c₀} {c₁} a P = (monadLaw₁ a P) , (monadLaw₁r a P)
```

In case of the second monadic law the proof is by induction over the proofs of traces for (P ≫=+ terminate), which immediately turn into traces of P:

```
monadLaw₂₊ : {c₀ : Choice} (P : Process+ ∞ c₀)→ (P >>=+ terminate) ⊑+ P
monadLaw₂₊ P .[] .nothing empty = empty
monadLaw₂₊ P .(Lab P x :: l) m (extc l .m x x₁) = extc l m x (monadLaw₂∞ (PE P x) l m x₁)
monadLaw₂₊ P l m (intc .l .m x x₁) = intc l m (inj₁ x) (monadLaw₂∞ (PI P x) l m x₁)
monadLaw₂₊ P .[] .(just (PT P x))(terc x) = intc [] (just (PT P x)) (inj₂ x) (lemTrTerBind P x)
```

In third monadic law the proof is by induction over the proofs of traces for (P ≫=+ (Q ≫=+ R)). In most cases the proof of traces carry over after applying the induction hypothesis. One special case if the first process P has a termination event, which results in an internal choice to ($Q\ x \gg= R$) on both sides. In this case the traces are essentially the same, but only after applying forcet. We use here an operation

```
monadPT+ P Q R y l m tr
```

which is modulo an application of forcet equal to tr. There are no immediate termination events, and therefore no proofs of traces of the form (terc x). We use efq (ex falsum quodlibet), which constructs from an element of the empty set an element of any set, for dealing with this case. The resulting proof is as follows:

```
monadLaw₃₊ : {c₀ c₁ c₂ : Choice} (P : Process+ ∞ c₀)
                  (Q : ChoiceSet c₀ → Process ∞ c₁)
                  (R : ChoiceSet c₁ → Process ∞ c₂)
               → ((P ≫=+ Q) ≫=+ R) ⊑+ (P ≫=+ (λ x → Q x ≫= R))
monadLaw₃₊ P Q R .[] .nothing empty = empty
monadLaw₃₊ P Q R .(Lab P x :: l) m (extc l .m x x₁) =
                                  extc l m x (monadLaw∞ P Q R l x m x₁)
monadLaw₃₊ P Q R l m (intc .l .m (inj₁ x) x₁) =
                                  intc l m (inj₁ (inj₁ x))(monadLaw₃∞ (PI P x) Q R l m x₁)
monadLaw₃₊ P Q R l m (intc .l .m (inj₂ y) x₁) =
                                  intc l m (inj₁ (inj₂ y))(monadPT+ P Q R y l m x₁)
monadLaw₃₊ P Q R .[] .(just (PT (P ≫=+ (λ x → Q x ≫= R)) x)) (terc x) = efq x
```

6.3 Proof of Commutativity of the Interleaving Operator

The interleaving combination ($P\ |||\ Q$) executes each component completely independent of the other, until termination. Traces of the interleaving combination ($P\ |||\ Q$) will, therefore, appear as interleaving of traces of the two component, and therefore it is easy to see that ($P\ |||\ Q$) and ($Q\ |||\ P$) are trace equivalent.

However, because of the monadic setting, for most algebraic laws the return types of the left and right hand side of an equation are different. Assume the return types of P and Q are c_0 and c_1, respectively. Then for instance the return type of ($P\ |||\ Q$) is ($c_0 \times' c_1$) whereas the return type of ($Q\ |||\ P$) is ($c_1 \times' c_0$). Therefore the algebraic laws hold only modulo applying an adjustment of the return types using the operation fmap, which applies a function to the return types.

Once we have taken this into account, a proof of commutativity of _|||_ is obtained by exchanging the external/internal/termination choices, which means swapping inj₁ and inj₂. Here inj₁ refers to choices in the first and inj₂ to choices in the second process. We give here the main case referring to Process+ (swap× swaps the two sides of a product):

```
S+|||+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀) (Q : Process+ ∞ c₁)
       → (P +|||+ Q) ⊑+ (fmap+ swap× (Q +|||+ P))
S+|||+ P Q .[] .nothing empty = empty
S+|||+ P Q .(Lab Q x :: l) m (extc l .m (inj₁ x) q) = extc l m (inj₂ x) (S+|||∞ P (PE Q x) l m q)
S+|||+ P Q .(Lab P x :: l) m (extc l .m (inj₂ x) q) = extc l m (inj₁ x) (S∞|||+ (PE P x) Q l m q)
S+|||+ P Q l m (intc .l .m (inj₁ x) q)              = intc l m (inj₂ x) (S+|||∞ P (PI Q x) l m q)
S+|||+ P Q l m (intc .l .m (inj₂ x) q)              = intc l m (inj₁ x) (S∞|||+ (PI P x) Q l m q)
S+|||+ P Q .[] .(just (PT P x ,, PT Q y)) (terc (y ,, x)) = terc (x ,, y)
```

```
≡S+|||+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀) (Q : Process+ ∞ c₁)
  → (P +|||+ Q) ≡+ (fmap+ swap× (Q +|||+ P))
≡S+|||+ P Q = (S+|||+ P Q) , (S+|||+R P Q)
```

6.4 Proof of Commutativity of the Parallel Operator

Most cases in the proof of the commutativity of _[_]||+[_]_ are similar to the proof of commutativity _|||_ – one swaps inj₁ and inj₂ and uses induction. The only more difficult case is when we have two processes synchronising, resulting in both processes following choices having the same labels. This case uses a proof that the two choices for the two processes result have the same label and that both labels are in the synchronised sets. We obtain in this case from a proof that we have a trace a proof of the Boolean conjunction:

```
Lab Q x ==l Lab P x₁ ∧ B (Lab X x) ∧ A (Lab P x₁)
```

which we need to transform into a proof of the Boolean conjunction

```
Lab P x₁ ==l Lab Q x ∧ A (Lab X x₁) ∧ B (Lab P x)
```

We will make use of functions which introduce and eliminate proofs of Boolean conjunctions, i.e.

```
∧BoolIntro  :  (a b : Bool) → T a → T b → T (a ∧ b)
∧BoolEliml  :  (a b : Bool) → T (a ∧ b) → T a
∧BoolElimr  :  (a b : Bool) → T (a ∧ b) → T b
```

Furthermore, we make use of a proof sym of symmetry of the Boolean equality _==l_ on labels, and the transfer lemma

```
transf : (Q : Label → Set) → (l  l′ : Label ) → T (l ==l l′ ) → Q l → Q l′
```

We now take the proof of the conjunction apart into its three components, apply the proof of symmetry to the equality proof and recombine them. Finally we need to carry out a transfer to replace the first label (Lab $P\,x_1$) in the trace by (Lab $Q\,x$), which are known to be equal. The resulting proof is as follows:

```
S+||+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀) (A B : Label → Bool) (Q : Process+ ∞ c₁)
         → (P [ A ]+||+[ B ] Q) ⊑+ fmap+ swap× (Q [ B ]+||+[ A ] P)

S+||+    P A B Q .[] .nothing empty = empty
S+||+    P A B Q .(Lab Q a :: l) m (extc l .m (inj₁ (inj₁ (sub a x))) x₁) =
                              extc l m (inj₁ (inj₂ (sub a x))) (S+||∞ P A B (PE Q a) l m x₁)
S+||+    P A B Q .(Lab P a :: l) m (extc l .m (inj₁ (inj₂ (sub a x))) x₁) =
                              extc l m (inj₁ (inj₁ (sub a x))) (S∞||+ (PE P a) A B Q l m x₁)
S+||+    P A B Q .(Lab Q x :: l) m (extc l .m (inj₂ (sub (x ,, x₁) x₂)) x₃) =
  let
    lxlx₁ : T (Lab Q x ==l Lab P x₁)
    lxlx₁  = ∧BoolEliml (Lab Q x ==l Lab P x₁)
                       (B (Lab Q x) ∧ A (Lab P x₁)) x₂
```

```
      BQx : T (B (Lab Q x))
      BQx  = ∧BoolEliml (B (Lab Q x)) (A (Lab P x₁))
                        (∧BoolElimr (Lab Q x ==l Lab P x₁)
                                    (B (Lab Q x) ∧ A (Lab P x₁)) x₂)

      APx₁ : T (A (Lab P x₁))
      APx₁  = ∧BoolElimr (B (Lab Q x)) (A (Lab P x₁))
                         (∧BoolElimr (Lab Q x ==l Lab P x₁)
                                     (B (Lab Q x) ∧ A (Lab P x₁)) x₂)

      lx₁lx : T (Lab P x₁ ==l Lab Q x)
      lx₁lx  = sym (Lab Q x) (Lab P x₁) lxlx₁

      x₂'    : T ((Lab P x₁ ==l Lab Q x) ∧ A (Lab P x₁) ∧ B (Lab Q x))
      x₂'    = ∧BoolIntro (Lab P x₁ ==l Lab Q x)
                          (A (Lab P x₁) ∧ B (Lab Q x))
                          lx₁lx
                          (∧BoolIntro (A (Lab P x₁)) (B (Lab Q x)) APx₁ BQx)

      auxpr : Tr+ (Lab P x₁ :: l) m (P [ A ]+||+[ B ] Q)
      auxpr = extc l m (inj₂ (sub (x₁ ,, x) x₂'))
                       (S∞||∞ (PE P x₁) A B (PE Q x) l m x₃)

   in transf (λ l' → Tr+ (l' :: l) m (P [ A ]+||+[ B ] Q))
             (Lab P x₁) (Lab Q x) lx₁lx auxpr

S+||+ P A B Q l m (intc .l .m (inj₁ x) x₁) = intc l m (inj₂ x) (S+||∞ P A B (PI Q x) l m x₁)
S+||+ P A B Q l m (intc .l .m (inj₂ y) x₁) = intc l m (inj₁ y) (S∞||+ (PI P y) A B Q l m x₁)
S+||+ P A B Q .[] .(just (PT P x₁ ,, PT Q x)) (terc (x ,, x₁)) = terc (x₁ ,, x)


≡+||+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀)(A B : Label → Bool)(Q : Process+ ∞ c₁)
        → (P [ A ]+||+[ B ] Q) ≡+ (fmap+ swap× ((Q [ B ]+||+[ A ] P)))
≡+||+    P A B Q = (S+||+ P A B Q) , (S+||+r P A B Q)
```

7 Related Work and Conclusion

Related Work. A detailed report on related work, which we do not want to repeat here, can be found in our previous paper [22].

Conclusion. The aims of this research is to give the type theoretic interactive theorem prover Agda the ability to model and verify concurrent programs by representing the process algebra CSP in monadic form. We implement trace semantics of CSP in Agda, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. In our approach we define processes coinductively and the trace semantic inductively.

Future Work. We are currently working on defining the failures/divergences model and stable failures model of CSP in Agda. Since those semantics are rather complicated, proofs of algebraic properties are much more involved. The first author has developed elements of the European Rail Traffic Management System ERTMS [16] in CSP, and one goal is to implement those processes in CSP-Agda and prove safety properties. For larger case studies automated theorem proving techniques will be used. Here we can build on Kanso's PhD thesis [24] (see as well [25]), in which he verified real world railway interlocking systems in Agda. Verifying larger examples might require to upgrade the integration of SAT solvers into Agda2, which has been developed by Kanso [24], to the current version of Agda.

One goal is to integrate the CSP model checker FDR2 into Agda. One ambitious goal is to write prototypes of programs, e.g. of some elements of the ERTMS, in Agda and make them directly executable in Agda. This uses the unique feature of Agda of being both a theorem prover and a dependently typed programming language. So in Agda there is no distinction between proofs and programs, between data types and propositions, and therefore the prototype can be implemented and verified in the same language, without the need to translate between two different languages.

References

1 Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006. URL: `http://www2.tcs.ifi.lmu.de/~abel/publications.html`.

2 Andreas Abel. Compositional coinduction with sized types. In Ichiro Hasuo, editor, *Coalgebraic Methods in Computer Science*, pages 5–10. Springer, 2016. `doi:10.1007/978-3-319-40370-0_2`.

3 Andreas Abel, Stephan Adelsberger, and Anton Setzer. ooAgda, 2016. URL: `https://github.com/agda/ooAgda`.

4 Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming*, 27, Jan 2017. `doi:10.1017/S0956796816000319`.

5 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *Proceedings of POPL'13*, pages 27–38. ACM, 2013. `doi:10.1145/2429069.2429075`.

6 Agda Community. The Agda Wiki, 2017. URL: `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

7 Agda Community. Literal Agda, 2017. URL: `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.LiterateAgda`.

8 JCM Baeten, Dirk Albert van Beek, and JE Rooda. Process algebra. *Handbook of Dynamic System Modeling*, pages 19–1, 2007. URL: `http://mate.tue.nl/mate/pdfs/8509.pdf`.

9 J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. CWI technical report, Stichting Mathematisch Centrum. Informatica-IW 206/82, 1982. URL: `http://oai.cwi.nl/oai/asset/6750/6750A.pdf`.

10 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — a functional language with dependent types. In *Proceedings of TPHOLs '09*, pages 73–78. Springer, 2009. `doi:10.1007/978-3-642-03359-9_6`.

11 Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78. Springer, 1994. doi:10.1007/3-540-58085-9_72.

12 Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In Gérard Huet and Gordon Plotkin, editors, *Logical frameworks*, pages 280–306. Cambridge University Press, 1991. URL: http://www.cse.chalmers.se/~peterd/papers/Setsem_Inductive.pdf.

13 Peter Dybjer. Universes and a general notion of simultaneous inductive-recursive definition in type theory. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Proceedings of the 1992 workshop on types for proofs and programs, Båstad*, June 1992. URL: http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz.

14 Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 2000. doi:10.2307/2586554.

15 Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1–47, 2003. doi:10.1016/S0168-0072(02)00096-9.

16 ERTMS. The European Rail Traffic Mangement System, 2013. URL: http://www.ertms.net/.

17 P. Hancock and A. Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. URL: http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html.

18 Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, LNCS, Vol. 1862, pages 317–331, 2000. doi:10.1007/3-540-44622-2_21.

19 Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal,7 July 2000*, 2000. URL: http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

20 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.

21 Bashar Igried and Anton Setzer. CSP-Agda. Agda library, 2016. URL: http://www.cs.swan.ac.uk/~csetzer/software/agda2/cspagda/.

22 Bashar Igried and Anton Setzer. Programming with monadic CSP-style processes in dependent type theory. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 28–38, New York, NY, USA, 2016. ACM. doi:10.1145/2976022.2976032.

23 Bashar Igried and Anton Setzer. Trace and stable failures semantics for csp-agda. In Ekaterina Komendantskaya and John Power, editors, *Proceedings of the First Workshop on Coalgebra, Horn Clause Logic Programming and Types, Edinburgh, UK, 28-29 November 2016*, volume 258 of *Electronic Proceedings in Theoretical Computer Science*, pages 36–51. Open Publishing Association, 2017. doi:10.4204/EPTCS.258.3.

24 Karim Kanso. *Agda as a Platform for the Development of Verified Railway Interlocking Systems*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea, UK, August 2012. URL: http://cs.swan.ac.uk/~cskarim/files/.

25 Karim Kanso and Anton Setzer. A light-weight integration of automated and interactive theorem proving. *Mathematical Structures in Computer Science*, FirstView:1–25, 12 November 2014. doi:10.1017/S0960129514000140.

26 Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984. ISBN: 88-7088-105-9.

27 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

28 A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0136744095.

29 Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley, 1st edition, 1999. ISBN: 978-0-471-62373-1.

30 Anton Setzer. Object-oriented programming in dependent type theory. In *Conference Proceedings of TFP 2006*, 2006. URL: `http://www.cs.swan.ac.uk/~csetzer/index.html`.

31 Anton Setzer, Andreas Abel, Brigitte Pientka, and David Thibodeau. Unnesting of copatterns. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi*, volume 8560 of *LNCS*, pages 31–45. Springer, 2014. `doi:10.1007/978-3-319-08918-8_3`.

On Subtyping in Type Theories with Canonical Objects

Georgiana Elena Lungu
Department of Computer Science, Royal Holloway, University of London, U.K.
georgiana.lungu.2013@live.rhul.ac.uk

Zhaohui Luo[1]
Department of Computer Science, Royal Holloway, University of London, U.K.
Zhaohui.Luo@rhul.ac.uk

Abstract

How should one introduce subtyping into type theories with canonical objects such as Martin-Löf's type theory? It is known that the usual subsumptive subtyping is inadequate and it is understood, at least theoretically, that coercive subtyping should instead be employed. However, it has not been studied what the proper coercive subtyping mechanism is and how it should be used to capture intuitive notions of subtyping. In this paper, we introduce a type system with signatures where coercive subtyping relations can be specified, and argue that this provides a suitable subtyping mechanism for type theories with canonical objects. In particular, we show that the subtyping extension is well-behaved by relating it to the previous formulation of coercive subtyping. The paper then proceeds to study the connection with intuitive notions of subtyping. It first shows how a subsumptive subtyping system can be embedded faithfully. Then, it studies how Russell-style universe inclusions can be understood as coercions in our system. And finally, we study constructor subtyping as an example to illustrate that, sometimes, injectivity of coercions need be assumed in order to capture properly some notions of subtyping.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases subtyping, type theory, conservative extension, canonical objects

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.13

Acknowledgements Thanks go to Sergei Soloviev for extremely helpful remarks on this work during his visit to Royal Holloway and the anonymous referees for their helpful comments.

1 Introduction

Type theories with canonical objects such as Martin-Löf's type theory [26] have been used as the basis for both theoretical projects such as Homotopy Type Theory [32] and practical applications in proof assistants such as Coq [10] and Agda [1]. In this paper, we investigate how to extend such type theories with subtyping relations, an issue that is important both theoretically and practically, but has not been settled.

Subsumptive Subtyping. The usual way to introduce subtyping is via the following subsumption rule:

$$\frac{a : A \quad A \leq B}{a : B}$$

[1] This work is partially supported by the EU COST Action CA15123 and the CAS/SAFEA International Partnership Program.

This is directly related to the notion of subset in mathematics and naturally linked to type assignment systems in programming languages like ML or Haskell. However, subsumptive subtyping is not adequate for type theories with canonical objects since it would destroy key properties of such type theories including canonicity (every object of an inductive type is equal to a canonical object) and subject reduction (computation preserves typing) [21, 16].

For instance, the Russell-style type universes $U_i : U_{i+1}$ $(i \in \omega)$ [23] constitute a special case of subsumptive subtyping with $U_i \leq U_{i+1}$ [18]. If we adopt the standard notation of terms with full type information, the resulting type theory with Russell-style universes would fail to have canonicity or subject reduction.[2] An alternative is to use proof terms with less typing information like using (a, b) instead of $pair(A, B, a, b)$ to represent pairs, as in HoTT (see Appendix 2 of [32]). The problem with this approach is that not only the property of type uniqueness fails, but a proof term may have incompatible types. For example, for $a : A$ and $A : U$, where U is a type universe, the pair (A, a) has both types $U \times A$ and $\Sigma X{:}U.X$, which are incompatible in the sense that none of them is a subtype of the other. This would lead to undecidability of type checking,[3] which is unacceptable for type theories with logics based on the propositions-as-types principle.

In §3 we will show how we can embed a subtyping system with the above subsumption rule into the coercive subtyping system we introduce in this paper.

Coercive Subtyping. An alternative way to introduce subtyping is coercive subtyping, where a subtyping relationship between two types is modelled by means of a unique coercion between them. The early developments of coercion semantics of subtyping for programming languages include [25, 29, 28, 6], among others. At the theoretical level, previous work on coercive subtyping for dependent type theories such as [15, 21] show that coercive subtyping can be adequately employed for dependent type theories with canonical objects to preserve the meta-theoretic properties such as canonicity and normalisation of the original type theories. Based on this, coercive subtyping has been successfully used in various applications based on the implementations of coercions in Coq and several other proof assistants [30, 3, 7].

However, the theoretical research on coercive subtyping such as [21] considers a rather abstract way of extension with coercive subtyping. For any type theory T, it extends it with a (coherent, but possibly infinite) set C of subtyping judgements to form a new type theory $T[C]$. Although this is well-suited in a theoretical study, it does not tell one how the extension should be formulated concretely in practice. In fact, a proposal of adding coercive subtyping assumptions in contexts [22] has met with potential difficulties in meta-theoretic studies that cast doubts on the seemingly attractive proposal. The complication was caused by the fact that coercion relations specified in a context can be moved to the right of the turnstile sign $\vdash$ to introduce terms with the so-called local coercions that are only effective in a localised scope. It is still unknown whether such mechanisms can be employed successfully. This has partly led to the current research that studies a more restrictive calculus that only allows coercive subtyping relations to be specified in signatures whose entries cannot be localised in terms.

[2] See §4.1 of the current paper for an example of the former and §4.3 of [16] for an example of the latter.

[3] To see the problem of type checking, it may be worth pointing out that, for a dependent type theory, type checking depends on type inference; put in another way, in a type-checking algorithm one has to infer the type of a term in many situations.

Main Contributions. In this paper, we study a type theory with signatures where coercive subtyping relations can be specified and argue that this provides a suitable subtyping mechanism for type theories with canonical objects.[4] This claim is backed up by first showing that the subtyping extension is conservative over the original type theory and that all its valid derivations correspond to valid derivations in the original calculus, and then studying its connection with subsumptive subtyping and its use in modelling some of the intuitive notions of subtyping including that induced by Russell-style universes in type theory.

The notion of signature in type theory was first studied in the Edinburgh Logical Framework [12] with judgements of the form $\Gamma \vdash_\Sigma J$, where the signatures Σ are used to describe constants of a logical system, in contrast with the contexts Γ that introduce variables which can be abstracted to the right of the turnstile sign by means of quantification or λ-abstraction. We will introduce the notion of signature by extending (the typed version of) Martin-Löf's logical framework LF (Chapter 9 of [14]) to obtain the system LF_S, which can be used similarly as LF in specifying type theories such as Martin-Löf's type theory [26]. Formulating the coercive subtyping relation in a type theory based on a logical framework makes it possible to extend the formulation to other type constructors too. We then introduce Π_S, a system with Π-types specified in LF_S, and $\Pi_{S,\leq}$ that extends Π_S to allow specification in signatures of subtyping entries $A \leq_c B$ that specifies that A is a subtype of B via coercion c, a function from A to B. We will justify that the coercive subtyping mechanism is abbreviational by showing that $\Pi_{S,\leq}$ is equivalent to a similar system as previously studied [21] and hence has desirable properties [31, 13, 33].

Although it is incompatible with the notion of canonical objects, subsumptive subtyping is widely used and, intuitively, it is the concept in mind in the first place when considering subtyping. It is therefore worth studying its relationship with the coercive subtyping calculus. Aspinall and Compagnoni [2] approached the topic of subsumptive subtyping in dependent type theory by developing a type system, with contextual subsumptive subtyping entries of the form $\alpha \leq A$ to declare that the type variable α is a subtype of A, and its checking algorithm in the Edinburgh Logical Framework. In this paper we shall define a subsumptive subtyping system in LF_S, one similar to that in [2], and prove that it can be faithfully embedded in $\Pi_{S,\leq}$.

It is worth noting that subtyping becomes particularly complicated in the case of dependent types. In a type system with contextual subtyping entries such as $\alpha \leq A$ as in Aspinall and Compagnoni's system, one has to decide whether to allow abstraction (for example, by λ or Π) over the subtyping entries. If one did, it would lead to types with bounded quantification of the form $\Pi\alpha \leq A.B$, which would result in complications and, most likely, undecidability of type checking (cf., Pierce's work that shows undecidability of type checking in $F_\leq$, an extension of the second-order λ-calculus with subtyping and bounded quantification [27]). In order to avoid bounded quantification, Aspinall and Compagnoni [2] present the subtyping entries in contexts, but do not enable their moving to the right of $\vdash$. In consequence, abstraction by λ or Π of those entries that occur to the left of a subtyping entry is obstructed. We chose to represent subtyping entries in the signatures in order to allow abstraction to happen freely for contextual entries.

We shall then consider two case studies, showing how coercive subtyping may be used to capture an intuitive notion of subtyping. Type universes [23] are our first example here. The Russell-style universes constitute a typical example of subsumptive subtyping. The

[4] A type theory with signatures was also proposed by the second author in [19] in the context of applying type theories to natural language semantics.

second author [18] observed that, although subsumptive subtyping causes problems with the notion of canonicity, one can obtain the essence of Russell-style universes by means of Tarski-style universes together with coercive subtyping by taking the explicit lifting operators between Tarski-style universes as coercions. Our embedding theorem (Theorem 34) that relates subsumptive and coercive subtyping can be extended for type systems with universes, therefore justifying this claim.

Subsumptive subtyping, esp. in its extreme forms, intuitively embodies a notion of injectivity that is in general not the case for coercive subtyping. One of such extreme forms of subtyping is constructor subtyping [4]. As the second case study, we shall relate it to our coercive subtyping system and show that, once equipped with injectivity of coercions, coercive subtyping can faithfully model the notion of injectivity intuitively assumed in subsumptive subtyping.

Related Work. Subtyping has been studied extensively both for type systems of programming languages and type theories implemented in proof assistants. Early studies of subtyping for programming languages have considered both subsumptive and coercive subtyping, mainly for simpler and non-dependent type systems (see, for example, [25, 29, 28, 6]). For example, Reynolds [28] considered extrinsic and intrinsic models of coercions and their applications to programming.

Subtyping in dependent type theories has been studied by Aspinall and Compagnoni [2] for Edinburgh LF, Betarte and Tasistro [5] about subkinding between kinds (called types) for Martin-Löf's logical framework, and Barthe and Frade [4] on constructor subtyping, among others. A theoretical framework of coercive subtyping for type theories with canonical objects has been developed and studied by the second author and colleagues in a series of papers and PhD theses [15, 21, 31, 13, 33]. In this setting, any dependent type theory T can be extended with coercive subtyping by giving a (possibly infinite) set C of basic subtyping judgements, resulting in the extended calculus $T[C]$. The meta-theory of such a calculus $T[C]$ was first studied in [31] where, among other things, the basic approach to proving that coercive subtyping is an abbreviational extension was developed, which was further studied and improved in, for example, [13, 33]. Coercions have been implemented in several proof assistants such as Coq [10, 30], Lego [20, 3], Matita [24] and Plastic [7] and used effectively for large proof development and, more recently, in formal development of natural language semantics based on type theory [17, 8, 9].

The above framework of coercive subtyping [21] has served as a theoretical tool to show in principle that coercive subtyping is adequate for type theories with canonical objects. However, as pointed out above, such a theoretical framework does not serve as a concrete system in practice. In this paper, we shall use subtyping entries in signatures to specify basic subtyping relations and study the resulting calculus, both in meta-theory and in practical modelling.

In §2, we present $\Pi_{S,\leq}$ and study its meta-theoretic properties. §3 presents a subsumptive subtyping system based on [2] and shows that it can be embedded faithfully in $\Pi_{S,\leq}$. The two case studies on type universes and injectivity are studied in §4, with the relationship between Russell-style and Tarski-style universes studied in §4.1 and constructor subtyping and injectivity in §4.2. The Conclusion discusses possible further research directions.

2 Coercive Subtyping in Signatures

We aim to introduce a calculus that can model intuitive notions of subtyping such as subsumption and, at the same time, preserves the desirable properties of the original type theory. In this section, we present $\Pi_{S,\leq}$, a type system with signatures where we can specify coercive subtyping relations, and then study its properties by relating it to the earlier formulation of coercive subtyping.

In what follows we use $\equiv$ for syntactic identity and assume that the signatures are coherent.

2.1 $\Pi_{S,\leq}$, a Type Theory with Subtyping in Signatures

2.1.1 Logical Framework with Signatures

Type theories can be specified in a logical framework such as Martin-Löf's logical framework [26] or its typed version LF [14]. We shall extend LF with signatures to obtain LF_S.

Informally, a signature is a sequence of entries of several forms, one of which is the form of membership entries $c : K$, which is the traditional form of entries as occurred in contexts (we shall add another form of entries in the next section). If a signature has only membership entries, it is of the form $c_1 : K_1, ..., c_n : K_n$.

▶ Remark (Constants and Variables). Intuitively, we shall call c declared in a signature entry $c : K$ as a constant, while x in a contextual entry $x : K$ as a variable. The formal difference is that, as declared in a signature entry, c cannot be substituted or abstracted (to the right of $\vdash$), while x declared in a contextual entry can either be substituted or abstracted by λ or Π (see later for the formal details.)

LF_S is a dependent type theory whose types are called *kinds* to distinguish them from types in the object type theory. It has the kind $Type$ of all types of the object type theory and dependent Π-kinds of the form $(x{:}K)K'$, which can be written as $(K)K'$ if $x \notin FV(K')$, whose objects are $\lambda-$abstractions of the form $[x{:}K]b$. For each type $A : Type$, we have a kind $El(A)$ which is often written just as A. In LF_S, there are six forms of judgements:

- Σ *valid*, asserting that Σ is a valid signature.
- $\vdash_\Sigma \Gamma$, asserting that Γ is a valid context under Σ.
- $\Gamma \vdash_\Sigma K$ *kind*, asserting that K is a kind in Γ under Σ.
- $\Gamma \vdash_\Sigma k : K$, asserting that k is an object of kind K in Γ under Σ.
- $\Gamma \vdash_\Sigma K_1 = K_2$, asserting that K_1 and K_2 are equal kinds in Γ under Σ.
- $\Gamma \vdash_\Sigma k_1 = k_2 : K$, asserting that k_1 and k_2 are equal objects of kind K in Γ under Σ.

The inference rules of the logical framework LF_S are given in Figure 1; they are the same as those of LF [14], except that we have judgements for signature validity, all other forms of judgements are adjusted accordingly with signatures attached, and we include some structural rules such as those for weakening and signature and context replacement (or signature and contextual equality), as done in the previous formulations in, for example, [21, 31, 33].

2.1.2 Type Theory with Π-types

Let Π_S be the type system with Π-types specified in LF_S. These Π-types are specified in the logical framework by introducing the constants, together with the definition rule, in Figure 2. Note that, with the constants in Figure 2, the rules in Figure 3 become derivable.

Validity of Signature/Contexts, Assumptions

$$\frac{}{\langle\rangle\ \mathit{valid}} \qquad \frac{\vdash_\Sigma K\,\mathit{kind} \quad c \notin \mathit{dom}(\Sigma)}{\Sigma, c{:}K\ \ \mathit{valid}} \qquad \frac{\vdash_{\Sigma,c:K,\Sigma'} \Gamma}{\Gamma \vdash_{\Sigma,c:K,\Sigma'} c{:}K}$$

$$\frac{\Sigma\ \ \mathit{valid}}{\vdash_\Sigma \langle\rangle} \qquad \frac{\Gamma \vdash_\Sigma K\,\mathit{kind} \quad x \notin \mathit{dom}(\Sigma) \cup \mathit{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}K} \qquad \frac{\vdash_\Sigma \Gamma, x{:}K, \Gamma'}{\Gamma, x{:}K, \Gamma' \vdash_\Sigma x{:}K}$$

Weakening

$$\frac{\Gamma \vdash_{\Sigma,\ \Sigma'} J \quad \vdash_\Sigma K\ \mathit{kind} \quad c \notin \mathit{dom}(\Sigma, \Sigma')}{\Gamma \vdash_{\Sigma,\ c:K,\ \Sigma'} J} \qquad \frac{\Gamma, \Gamma' \vdash_\Sigma J \quad \Gamma \vdash_\Sigma K\ \mathit{kind} \quad x \notin \mathit{dom}(\Gamma, \Gamma')}{\Gamma, x{:}K, \Gamma' \vdash_\Sigma J}$$

Equality Rules

$$\frac{\Gamma \vdash_\Sigma K\ \ \mathit{kind}}{\Gamma \vdash_\Sigma K = K} \qquad \frac{\Gamma \vdash_\Sigma K = K'}{\Gamma \vdash_\Sigma K' = K} \qquad \frac{\Gamma \vdash_\Sigma K = K' \quad \Gamma \vdash_\Sigma K' = K''}{\Gamma \vdash_\Sigma K = K''}$$

$$\frac{\Gamma \vdash_\Sigma k{:}K}{\Gamma \vdash_\Sigma k = k{:}K} \qquad \frac{\Gamma \vdash_\Sigma k = k'{:}K}{\Gamma \vdash_\Sigma k' = k{:}K} \qquad \frac{\Gamma \vdash_\Sigma k = k'{:}K \quad \Gamma \vdash_\Sigma k' = k''{:}K}{\Gamma \vdash_\Sigma k = k''{:}K}$$

$$\frac{\Gamma \vdash_\Sigma k{:}K \quad \Gamma \vdash_\Sigma K = K'}{\Gamma \vdash_\Sigma k{:}K'} \qquad \frac{\Gamma \vdash_\Sigma k = k'{:}K \quad \Gamma \vdash_\Sigma K = K'}{\Gamma \vdash_\Sigma k = k'{:}K'}$$

Signature Replacement

$$\frac{\Gamma \vdash_{\Sigma_0, c:L, \Sigma_1} J \quad \vdash_{\Sigma_0} L = L'}{\Gamma \vdash_{\Sigma_0, c:L', \Sigma_1} J}$$

Context Replacement

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma J \quad \Gamma_0 \vdash_\Sigma K = K'}{\Gamma_0, x{:}K', \Gamma_1 \vdash_\Sigma J}$$

Substitution Rules

$$\frac{\vdash_\Sigma \Gamma_0, x{:}K, \Gamma_1 \quad \Gamma_0 \vdash_\Sigma k{:}K}{\vdash_\Sigma \Gamma_0, [k/x]\Gamma_1}$$

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma K'\ \ \mathit{kind} \quad \Gamma_0 \vdash_\Sigma k{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]K'\ \ \mathit{kind}} \qquad \frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma L = L' \quad \Gamma_0 \vdash_\Sigma k{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]L = [k/x]L'}$$

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma k'{:}K' \quad \Gamma_0 \vdash_\Sigma k{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]k'{:}[k/x]K'} \qquad \frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma l = l'{:}K' \quad \Gamma_0 \vdash_\Sigma k{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]l = [k/x]l'{:}[k/x]K'}$$

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma K'\ \ \mathit{kind} \quad \Gamma_0 \vdash_\Sigma k = k'{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]K' = [k'/x]K'} \qquad \frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma l{:}K' \quad \Gamma_0 \vdash_\Sigma k = k'{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]l = [k'/x]l{:}[k/x]K'}$$

Dependent Product Kinds

$$\frac{\Gamma \vdash_\Sigma K\,\mathit{kind} \quad \Gamma, x{:}K \vdash_\Sigma K'\,\mathit{kind}}{\Gamma \vdash_\Sigma (x{:}K)K'\,\mathit{kind}} \qquad \frac{\Gamma \vdash_\Sigma K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash_\Sigma K_1' = K_2'}{\Gamma \vdash_\Sigma (x{:}K_1)K_1' = (x{:}K_2)K_2'}$$

$$\frac{\Gamma, x{:}K \vdash_\Sigma y{:}K'}{\Gamma \vdash_\Sigma [x{:}K]y{:}(x{:}K)K'} \qquad \frac{\Gamma \vdash_\Sigma K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash_\Sigma k_1 = k_2{:}K}{\Gamma \vdash_\Sigma [x{:}K_1]k_1 = [x{:}K_2]k_2{:}(x{:}K_1)K}$$

$$\frac{\Gamma \vdash_\Sigma f{:}(x{:}K)K' \quad \Gamma \vdash_\Sigma k{:}K}{\Gamma \vdash_\Sigma f(k){:}[k/x]K'} \qquad \frac{\Gamma \vdash_\Sigma f = f'{:}(x{:}K)K' \quad \Gamma \vdash_\Sigma k_1 = k_2{:}K}{\Gamma \vdash_\Sigma f(k_1) = f'(k_2){:}[k_1/x]K'}$$

$$\frac{\Gamma, x{:}K \vdash_\Sigma k'{:}K' \quad \Gamma \vdash_\Sigma k{:}K}{\Gamma \vdash_\Sigma ([x{:}K]k')(k) = [k/x]k'{:}[k/x]K'} \qquad \frac{\Gamma \vdash_\Sigma f{:}(x{:}K)K' \quad x \notin FV(f)}{\Gamma \vdash_\Sigma [x{:}K]f(x) = f{:}(x{:}K)K'}$$

The kind Type

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathit{Type}\ \ \mathit{kind}} \qquad \frac{\Gamma \vdash_\Sigma A{:}\mathit{Type}}{\Gamma \vdash_\Sigma \mathit{El}(A)\ \ \mathit{kind}} \qquad \frac{\Gamma \vdash_\Sigma A = B{:}\mathit{Type}}{\Gamma \vdash_\Sigma \mathit{El}(A) = \mathit{El}(B)}$$

Figure 1 Inference Rules for LF_S.

Constant declarations:

$$\begin{array}{rcl} \Pi & : & (A{:}Type)(B{:}(A)Type)Type \\ \lambda & : & (A{:}Type)(B{:}(A)Type)((x{:}A)B(x))\Pi(A,B) \\ app & : & (A{:}Type)(B{:}(A)Type)(\Pi(A,B))(x{:}A)B(x) \end{array}$$

Definitional equality rule

$$app(A,B,\lambda(A,B,f),a) = f(a) : B(a).$$

Figure 2 Constants for Π-types in logical framework.

$$\frac{\Gamma \vdash_\Sigma A : Type \quad \Gamma, x{:}A \vdash_\Sigma B(x) : Type}{\Gamma \vdash_\Sigma \Pi(A,B) : Type}$$

$$\frac{\Gamma \vdash_\Sigma A : Type \quad \Gamma \vdash_\Sigma B : (A)Type \quad \Gamma \vdash_\Sigma f : (x{:}A)B(x)}{\Gamma \vdash_\Sigma \lambda(A,B,f) : \Pi(A,B)}$$

$$\frac{\Gamma \vdash_\Sigma g : \Pi(A,B) \quad \Gamma \vdash_\Sigma a : A}{\Gamma \vdash_\Sigma app(A,B,g,a) : B(a)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma A : Type \quad \Gamma \vdash_\Sigma B : (A)Type \\ \Gamma \vdash_\Sigma f : (x{:}A)B(x) \quad \Gamma \vdash_\Sigma a : A\end{array}}{\Gamma \vdash_\Sigma app(A,B,\lambda(A,B,f),a) = f(a) : B(a)}$$

Figure 3 Inference Rules for Π_S.

2.1.3 Subtyping Entries in Signatures

We present the whole system $\Pi_{S,\leq}$. First, subtyping is represented by means of two forms of judgements:

- subtyping judgements $\Gamma \vdash_\Sigma A \leq_c B : Type$, and
- subkinding judgements $\Gamma \vdash_\Sigma K \leq_c K'$.

Subtyping relations between types (not kinds) can be specified in a signature by means of entries $A \leq_c B : Type$ (or simply written as $A \leq_c B$), where A and B are types and $c : (A)B$.[5]

The specifications of subtyping relations are also required to be *coherent*. Coherence is crucial as it ensures a coercive application abbreviates a unique functional application. To define this notion of coherence, we need to introduce a subsystem of $\Pi_{S,\leq}$, called $\Pi^{0K}_{S,\leq}$, defined by the rules of Π_S together with those in Figures 4 and 5, where in the rule for dependent products in Figures 4, the notation $c_2[x]$ was explained in, for example, [16]: it means that x may occur free in c_2, although only inessentially[6]. The composition of functions is defined as follows: For $f{:}(K1)K2$ and $g{:}(K2)K3, g \circ f = [x{:}K1]g(f(x)){:}(K1)K3$.

Here is the definition of coherence of a signature, which intuitively says that, under a coherent signature, there cannot be two different coercions between the same types.

[5] Using some types not contained in $\Pi_{S,\leq}$, more interesting subtyping relations can be specified. For example, for $A \leq_c B$, we could have $A \equiv Vect(N,n)$, $B \equiv List(N)$ and c maps vector $< m_1, ..., m_n >$ to list $[m_1, ..., m_n]$. We shall not formally deal with such extended type systems in the current paper, but the ideas and results are expected to extend to the type systems with such data types (eg, all those in Martin-Löf's type theory).

[6] For instance, one might have (by using the congruence rule) $x{:}A \vdash_\Sigma B \leq_{([y:A]e)(x)} B'$, where $B \leq_e B'$ and $x \notin FV(e)$.

Signature Rules for Subtyping

$$\frac{\vdash_\Sigma A : \mathit{Type} \quad \vdash_\Sigma B : \mathit{Type} \quad \vdash_\Sigma c : (A)B}{\Sigma, A \le_c B \ \ \mathit{valid}} \qquad \frac{\vdash_{\Sigma_0, A \le_c B:\mathit{Type}, \Sigma_1} \Gamma}{\Gamma \vdash_{\Sigma_0, A \le_c B:\mathit{Type}, \Sigma_1} A \le_c B : \mathit{Type}}$$

Congruence

$$\frac{\Gamma \vdash_\Sigma A \le_c B : \mathit{Type} \quad \Gamma \vdash_\Sigma A = A' : \mathit{Type} \quad \Gamma \vdash_\Sigma B = B' : \mathit{Type} \quad \Gamma \vdash_\Sigma c = c' : (A)B}{\Gamma \vdash_\Sigma A' \le_{c'} B' : \mathit{Type}}$$

Transitivity

$$\frac{\Gamma \vdash_\Sigma A \le_c A' : \mathit{Type} \quad \Gamma \vdash_\Sigma A' \le_{c'} A'' : \mathit{Type}}{\Gamma \vdash_\Sigma A \le_{c' \circ c} A'' : \mathit{Type}}$$

Weakening

$$\frac{\Gamma \vdash_{\Sigma,\ \Sigma'} A \le_d B : \mathit{Type} \quad \vdash_\Sigma K \ \mathit{kind}}{\Gamma \vdash_{\Sigma,\ c:K,\ \Sigma'} A \le_d B : \mathit{Type}} \quad (c \notin \mathit{dom}(\Sigma, \Sigma'))$$

$$\frac{\Gamma, \Gamma' \vdash_\Sigma A \le_d B : \mathit{Type} \quad \Gamma \vdash_\Sigma K \ \mathit{kind}}{\Gamma, x{:}K, \Gamma' \vdash_\Sigma A \le_d B : \mathit{Type}} \quad (x \notin \mathit{dom}(\Gamma, \Gamma'))$$

Signature Replacement

$$\frac{\Gamma \vdash_{\Sigma_0, c:L, \Sigma_1} A \le_d B : \mathit{Type} \quad \vdash_{\Sigma_0} L = L'}{\Gamma \vdash_{\Sigma_0, c:L', \Sigma_1} A \le_d B : \mathit{Type}}$$

Context Replacement

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma A \le_d B : \mathit{Type} \quad \Gamma_0 \vdash_\Sigma K = K'}{\Gamma_0, x{:}K', \Gamma_1 \vdash_\Sigma A \le_d B : \mathit{Type}}$$

Substitution

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma A \le_c B{:}\mathit{Type} \quad \Gamma_0 \vdash_\Sigma k{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]A \le_{[k/x]c} [k/x]B}$$

Identity Coercion

$$\frac{\Gamma \vdash_\Sigma A : \mathit{Type}}{\Gamma \vdash_\Sigma A \le_{[x:A]x} A : \mathit{Type}}$$

Dependent Product

$$\frac{\Gamma \vdash_\Sigma A' \le_{c_1} A : \mathit{Type} \quad \Gamma \vdash_\Sigma B, B' : (A)\mathit{Type} \quad \Gamma, x{:}A \vdash_\Sigma B(x) \le_{c_2[x]} B'(x) : \mathit{Type}}{\Gamma \vdash_\Sigma \Pi(A, B) \le_d \Pi(A', B' \circ c_1) : \mathit{Type}}$$

where $d \equiv [F : \Pi(A,B)]\lambda(A', B' \circ c_1, [x{:}A']c_2[x](\mathit{app}(A, B, F, c_1(x))))$.

Figure 4 Inference Rules for $\Pi^{0K}_{S,\le}$ (1).

▶ **Definition 1.** A signature Σ is **coherent** if, in $\Pi^{0K}_{S,\le}$, $\Gamma \vdash_\Sigma A \le_c B$ and $\Gamma \vdash_\Sigma A \le_{c'} B$ imply $\Gamma \vdash_\Sigma c = c' : (A)B$.

Note that, in comparison with earlier formulations such as [21], we have switched from strict subtyping relation $<$ to $\le$ and the coherence condition is changed accordingly as well; in particular, under a coherent signature, any coercion from a type to itself must be equal to the identity function. (This is a special case of the above condition when $B \equiv A$: because

Basic Subkinding Rule and Identity Coercion

$$\frac{\Gamma \vdash_\Sigma A \leq_c B{:}Type}{\Gamma \vdash_\Sigma El(A) \leq_c El(B)} \qquad \frac{\Gamma \vdash_\Sigma K\ \ kind}{\Gamma \vdash_\Sigma K \leq_{[x:K]x} K}$$

Structural Subkinding Rules

$$\frac{\Gamma \vdash_\Sigma K_1 \leq_c K_2 \quad \Gamma \vdash_\Sigma K_1 = K_1' \quad \Gamma \vdash_\Sigma K_2 = K_2' \quad \Gamma \vdash_\Sigma c = c'{:}(K_1)K_2}{\Gamma \vdash_\Sigma K_1' \leq_{c'} K_2'}$$

$$\frac{\Gamma \vdash_\Sigma K \leq_c K' \quad \Gamma \vdash_\Sigma K' \leq_{c'} K''}{\Gamma \vdash_\Sigma K \leq_{c' \circ c} K''}$$

$$\frac{\Gamma \vdash_{\Sigma,\ \Sigma'} K \leq_d K' \quad \vdash_\Sigma K_0\ kind}{\Gamma \vdash_{\Sigma,\ c:K_0,\ \Sigma'} K \leq_d K'} \quad (c \notin dom(\Sigma, \Sigma'))$$

$$\frac{\Gamma, \Gamma' \vdash_\Sigma K \leq_d K' \quad \Gamma \vdash_\Sigma K_0\ kind}{\Gamma, x{:}K_0, \Gamma' \vdash_\Sigma K \leq_d K'} \quad (x \notin dom(\Gamma, \Gamma'))$$

$$\frac{\Gamma \vdash_{\Sigma_0, c:L, \Sigma_1} K \leq_d K' \quad \vdash_{\Sigma_0} L = L'}{\Gamma \vdash_{\Sigma_0, c:L', \Sigma_1} K \leq_d K'} \quad \frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma L \leq_d L' \quad \Gamma_0 \vdash_\Sigma K = K'}{\Gamma_0, x{:}K', \Gamma_1 \vdash_\Sigma L \leq_d L'}$$

$$\frac{\Gamma_0, x{:}K, \Gamma_1 \vdash_\Sigma K_1 \leq_c K_2 \quad \Gamma_0 \vdash_\Sigma k{:}K}{\Gamma_0, [k/x]\Gamma_1 \vdash_\Sigma [k/x]K_1 \leq_{[k/x]c} [k/x]K_2}$$

Subkinding for Dependent Product Kind

$$\frac{\Gamma \vdash_\Sigma K_1' \leq_{c_1} K_1 \quad \Gamma, x{:}K_1 \vdash_\Sigma K_2\ kind \quad \Gamma, x'{:}K_1' \vdash_\Sigma K_2'\ kind \quad \Gamma, x{:}K_1 \vdash_\Sigma [c_1(x')/x]K_2 \leq_{c_2} K_2'}{\Gamma \vdash_\Sigma (x{:}K_1)K_2 \leq_{[f:(x:K_1)K_2][x':K_1']c_2(f(c_1(x')))} (x{:}K_1')K_2'}$$

Figure 5 Inference Rules for $\Pi^{0K}_{S,\leq}$ (2).

we always have $A \leq_{[x:A]x} A$, if $A \leq_c A$, then $c = [x{:}A]x : (A)A$.) Note also that, it is easy to prove by induction that, if $\Gamma \vdash_\Sigma A \leq_c B : Type$, then $\Gamma \vdash_\Sigma A, B : Type$ and $\Gamma \vdash_\Sigma c : (A)B$.

It is also important to note the difference between a judgement with signature in the current calculus and that in the calculus employed in [21] where there are no signatures. For example, the signatures Σ_1 that contains $A \leq_c B$ and Σ_2 that contains $A \leq_d B$ can both be coherent signatures even when $c \neq d$, while such a situation can only be considered in the earlier setting by having two different type systems $T[C_1]$ and $T[C_2]$, which is rather cumbersome to say the least.[7]

We can, at this point, complete the specification of the system $\Pi_{S,\leq}$ as the extension of $\Pi^{0K}_{S,\leq}$ by adding the rules in Figure 6.

▶ Remark. We can now explain why we have to present the system $\Pi^{0K}_{S,\leq}$ first. The reason is that the coercive definition rule (CD) will force any two coercions to be equal. Therefore, we cannot define the notion of coherence for the system including the (CD) rule as, if we did so, every signature would be coherent by definition.

[7] This has some unexpected consequences concerning parameterised coercions as well. But it is a topic beyond the current paper and will be discussed somewhere else.

Coercive Application

(CA_1) $$\frac{\Gamma \vdash_\Sigma f{:}(x{:}K)K' \quad \Gamma \vdash_\Sigma k_0{:}K_0 \quad \Gamma \vdash_\Sigma K_0 \leq_c K}{\Gamma \vdash_\Sigma f(k_0){:}[c(k_0)/x]K'}$$

(CA_2) $$\frac{\Gamma \vdash_\Sigma f = f'{:}(x{:}K)K' \quad \Gamma \vdash_\Sigma k_0 = k_0'{:}K_0 \quad \Gamma \vdash_\Sigma K_0 \leq_c K}{\Gamma \vdash_\Sigma f(k_0) = f'(k_0'){:}[c(k_0)/x]K'}$$

Coercive Definition

(CD) $$\frac{\Gamma \vdash_\Sigma f{:}(x{:}K)K' \quad \Gamma \vdash_\Sigma k_0{:}K_0 \quad \Gamma \vdash_\Sigma K_0 \leq_c K}{\Gamma \vdash_\Sigma f(k_0) = f(c(k_0)){:}[c(k_0)/x]K'}$$

Figure 6 The coercive application and definition rules in $\Pi_{S,\leq}$.

2.2 Coherence for Kinds and Conservativity

In this subsection, we prove two basic properties of $\Pi_{S,\leq}$: (1) coherence, as defined for types, extends to kinds; (2) it is a conservative extension of the system Π_S.

2.2.1 Coherence for Kinds

Note that the coherence definition refers to types. In what follows we prove that coherence for types implies coherence for kinds. We categorise kinds and show that they can be related via definitional equality or subtyping only if they are of the same category. For this we also define the degree of a kind which intuitively denotes how many dependent product occurrences are in a kind.

▶ **Lemma 2.** *If $\Gamma \vdash_\Sigma A \leq_c B$ is derivable in $\Pi^{0K}_{S,\leq}$ then $\Gamma \vdash_\Sigma c{:}(A)B$ is derivable in $\Pi^{0K}_{S,\leq}$.*

Proof. By induction on the structure of derivations ◀

▶ **Lemma 3.** *If $\Gamma \vdash K \leq_c L$ is derivable in $\Pi^{0K}_{S,\leq}$ then $\Gamma \vdash c{:}(K)L$ is derivable in $\Pi^{0K}_{S,\leq}$.*

Proof. By induction on the structure of derivations. We consider $K \equiv (x{:}K_1)K_2$ and $L \equiv (x{:}L_1)L_2$. If a derivation tree for $\Gamma \vdash K \leq_c L$ ends with the rule for dependent product kind with premises $\Gamma \vdash_\Sigma L_1 \leq_{c_1} K_1$, $\Gamma, x{:}K_1 \vdash_\Sigma K_2$ *kind*, $\Gamma, y{:}L_1 \vdash_\Sigma L_2$ *kind* and $\Gamma, y{:}L_1 \vdash_\Sigma [c_1(y)/x]K_2 \leq_{c_2} L_2$. By IH we have $\Gamma \vdash_\Sigma c_1{:}(L_1)K_1$ and $\Gamma, y{:}L_1 \vdash_\Sigma c_2{:}([c_1(y)/x]K_2)L_2$. By weakening $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma c_2{:}([c_1(y)/x]K_2)L_2$ and $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma c_1{:}(L_1)K_1$. We have $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma y{:}L_1$ so by application $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma c_1(y){:}K_1$. We have $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma f{:}(x{:}K_1)K_2$ so by application we have $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma f(c_1(y)){:}[c_1(y)/x]K_2$. By application again we have $\Gamma, f{:}(x{:}K_1)K_2, y{:}L_1 \vdash_\Sigma c_2(f(c_1(y))){:}L_2$ and by abstraction $\Gamma \vdash_\Sigma [f{:}(x{:}K_1)K_2][y{:}L_1]c_2(f(c_1(y))){:}((x{:}K_1)K_2)(y{:}L_1)L_2$ ◀

▶ **Lemma 4.** *Let $\Gamma \vdash_\Sigma K \leq_c L$ be derivable in $\Pi^{0K}_{S,\leq}$. Then K and L are of the same form, i.e., both are El-terms or both are dependent product kinds. Furthermore,*

- *if $K \equiv El(A)$ and $L \equiv El(B)$, then $\Gamma \vdash_\Sigma A \leq_c B : Type$ is derivable in $\Pi^{0K}_{S,\leq}$ and*
- *if $K \equiv (x{:}K_1)K_2$ and $L \equiv (x{:}L_1)L_2$, then $\Gamma \vdash_\Sigma K_1$ kind, $\Gamma, x{:}K_1 \vdash_\Sigma K_2$ kind, $\Gamma \vdash_\Sigma L_1$ kind, and $\Gamma, x{:}L_1 \vdash_\Sigma L_2$ kind are derivable in $\Pi^{0K}_{S,\leq}$.*

The following lemma states that, if there is a subtyping relation between two dependent kinds, then the coercion can be obtained by the subtyping for dependent product kind rule from Figure 5. Note that for this to hold it is essential that we only have subtyping entries in signatures and not subkinding.

▶ **Lemma 5.** *If* $\Gamma \vdash_\Sigma (x{:}K_1)K_2 \leq_d (y{:}L_1)L_2$ *is derivable in* $\Pi^{0K}_{S,\leq}$ *then there exist derivable judgements in* $\Pi^{0K}_{S,\leq}$, $\Gamma \vdash_\Sigma c_1{:}(L_1)K_1$ *and* $\Gamma, y{:}L_1 \vdash_\Sigma c_2{:}([c_1(y)/x]K_2)L_2$ *s.t.*

- $\Gamma \vdash_\Sigma L_1 \leq_{c_1} K_1$
- $\Gamma, y{:}K_1' \vdash_\Sigma [c_1(y)/x]K_2 \leq_{c_2} L_2$ *and*
- $\Gamma \vdash_\Sigma d = [f{:}(x{:}K_1)K_2][y{:}L_1]c_2(f(c_1(y))){:}((x{:}K_1)K_2)(y{:}L_1)L_2$

are derivable in $\Pi^{0K}_{S,\leq}$.

Proof. By induction on the structure of derivation of $\Gamma \vdash_\Sigma (x{:}K_1)K_2 \leq_d (y{:}L_1)L_2$. The only non trivial case is when it comes from transitivity.

$$\frac{\Gamma \vdash_\Sigma (x{:}K_1)K_2 \leq_{d_1} C \quad \Gamma \vdash_\Sigma C \leq_{d_2} (y{:}L_1)L_2}{\Gamma \vdash_\Sigma (x{:}K_1)K_2 \leq_{d_2 \circ d_1} (y{:}L_1)L_2}$$

By the previous lemma $\Gamma \vdash_\Sigma C \equiv (z{:}M_1)M_2$. By IH we have that

- $\Gamma \vdash_\Sigma M_1 \leq_{c_1'} K_1$
- $\Gamma, z{:}M_1 \vdash_\Sigma [c_1'(z)/x]K_2 \leq_{c_2'} M_2$
- $\Gamma \vdash_\Sigma d_1 = [f{:}(x{:}K_1)K_2][z{:}M_1]c_2'(f(c_1'(z))){:}((x{:}K_1)K_2)(z{:}M_1)M_2$

and

- $\Gamma \vdash_\Sigma L_1 \leq_{c_1''} M_1$
- $\Gamma, y{:}L_1 \vdash_\Sigma [c_1''(y)/z]M_2 \leq_{c_2''} L_2$
- $\Gamma \vdash_\Sigma d_2 = [f{:}(z{:}M_1)M_2][y{:}L_1]c_2''(f(c_1''(y))){:}((z{:}M_1)M_2)(y{:}L_1)L_2$

are derivable. We apply transitivity to obtain $\Gamma \vdash_\Sigma L_1 \leq_{c_1' \circ c_1''} K_1$ and by weakening and substitution in addition, $\Gamma, y{:}L_1 \vdash_\Sigma [c_1'(c_1''(y))/x]K_2 \leq_{c_2'' \circ [c_1''(y)/z]c_2'} L_2$ and what is left to prove is that $\Gamma \vdash_\Sigma d_2 \circ d_1 = [f{:}(x{:}K_1)K_2][y{:}L_1](c_2'' \circ [c_1''(y)/z]c_2')(f((c_1' \circ c_1'')(y))){:}((x{:}K_1)K_2)(y{:}L_1)L_2$. Let $\Gamma \vdash_\Sigma F{:}(x{:}K_1)K_2$

$$\begin{aligned}
d_2 \circ d_1(F) &= d_2(d_1(F)) \\
&= d_2([f{:}(x{:}K_1)K_2][z{:}M_1]c_2'(f(c_1'(z)))(F)) \\
&= d_2([F/f][z{:}M_1]c_2'(f(c_1'(z)))) \\
&= d_2([z{:}M_1]c_2'(F(c_1'(z)))) \\
&= ([f{:}(z{:}M_1)M_2][y{:}L_1]c_2''(f(c_1''(y))))([z{:}M_1]c_2'(F(c_1'(z)))) \\
&= [z{:}M_1]c_2'(F(c_1'(z)))/f]([y{:}L_1]c_2''(f(c_1''(y)))) \\
&= [y{:}L_1]c_2''([z{:}M_1]c_2'(F(c_1'(z)))(c_1''(y))) \\
&= [y{:}L_1]c_2''([c_1''(y)/z]c_2'(F(c_1'(c_1''(y))))) \\
&= [y{:}L_1]c_2''([c_1''(y)/z]c_2'(F(c_1'(c_1''(y))))) \\
&= [y{:}L_1](c_2'' \circ [c_1''(y)/z]c_2')(F((c_1' \circ c_1'')(y))) \\
&= ([f{:}(x{:}K_1)K_2][y{:}L_1](c_2'' \circ [c_1''(y)/z]c_2')(f((c_1' \circ c_1'')(y))))(F)
\end{aligned}$$ ◀

The following de

nition gives us a measure for the structure of kinds. We will use this measure when proving coherence for kinds. It is particularly important and we will use the fact that this measure is not increased by substitution.

▶ **Definition 6.** For $\Gamma \vdash_\Sigma K$ we define the **degree of K** where $\Gamma \vdash_\Sigma K$ *kind* as $\boldsymbol{deg(K)} \in \mathbb{N}$ as follows:

1. $deg(Type) = 1$
2. $deg(El(A)) = 1$
3. $deg((x{:}K)L) = deg(K) + deg(L)$

▶ **Lemma 7.** *The following hold:*
- *if* $\Gamma \vdash_\Sigma K = L$ *is derivable in* $\Pi^{0K}_{S,\le}$ *then* $deg(K) = deg(L)$
- *if* $\Gamma \vdash_\Sigma K \le_c L$ *is derivable in* $\Pi^{0K}_{S,\le}$ *then* $deg(K) = deg(L)$

Proof. We do induction on the structure of derivations of $\Gamma \vdash_\Sigma K = L$ respectively $\Gamma \vdash_\Sigma K \le L$. For example if it comes from the rule

$$\frac{\Gamma \vdash_\Sigma K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash_\Sigma K_1' = K_2'}{\Gamma \vdash_\Sigma (x{:}K_1)K_1' = (x{:}K_2)K_2'}$$

by IH, $deg(K_1) = deg(K_2)$ and $deg(K_1') = deg(K_2')$, hence $deg((x{:}K_1)K_1') = deg((x{:}K_2)K_2')$ ◀

▶ **Lemma 8** (Coherence for Kinds). *If* $\Gamma \vdash_\Sigma K \le_c L$ *and* $\Gamma \vdash_\Sigma K \le_{c'} L$ *are derivable in* $\Pi^{0K}_{S,\le}$, *then* $\Gamma \vdash_\Sigma c = c' : (K)L$ *is derivable in* $\Pi^{0K}_{S,\le}$.

Proof. By induction on $n = deg(K)$.
1. For $n = 1$:
   - If $\Gamma \vdash_\Sigma K = El(A)$ and $\Gamma \vdash_\Sigma L = El(B)$ then by Lemma 4 we have $\Gamma \vdash_\Sigma A \le_c B$ and $\Gamma \vdash_\Sigma A \le_{c'} B$ and from coherence for types $\Gamma \vdash_\Sigma c = c'{:}(A)B$, hence $\Gamma \vdash_\Sigma c = c'{:}(K)L$
   - If $\Gamma \vdash_\Sigma K = Type$ and $\Gamma \vdash_\Sigma L = Type$ then we can only have $\Gamma \vdash_\Sigma c = Id{:}(K)L$.
2. For $n > 1$, $\Gamma \vdash_\Sigma K \equiv (x{:}K_1)K_2$ and $\Gamma \vdash_\Sigma L \equiv (x{:}L_1)L_2$, by Lemma 5
   - $\Gamma \vdash_\Sigma L_1 \le_{c_1} K_1$,
   - $\Gamma, x{:}K_1 \vdash_\Sigma [c_1(y)/x]K_2 \le_{c_2} L_2$ and
   - $\Gamma \vdash_\Sigma c = [f{:}(x{:}K_1)K_2][y{:}L_1]c_2(f(c_1(y))){:}((x{:}K_1)K_2)(y{:}L_1)L_2$

   are derivable for some $\Gamma \vdash_\Sigma c_1{:}(L_1)K_1$ and $\Gamma, x{:}K_1 \vdash_\Sigma c_2{:}([c_1(x)/x]K_2)L_2$ and $deg(L_1)$, $deg(K_1)$, $deg([c_1(y)/x]K_2)$, $deg(L_2)$ are all smaller than n. If
   - $\Gamma \vdash_\Sigma L_1 \le_{c_1'} K_1$,
   - $\Gamma, x{:}K_1 \vdash_\Sigma [c_1'(y)/x]K_2 \le_{c_2'} L_2$ and
   - $\Gamma \vdash_\Sigma c' = [f{:}(x{:}K_1)K_2][y{:}L_1]c_2'(f(c_1'(y))){:}((x{:}K_1)K_2)(y{:}L_1)L_2$

   are derivable for some other coercions $\Gamma \vdash_\Sigma c_1'{:}(L_1)K_1$ and $\Gamma, x{:}K_1 \vdash_\Sigma c_2'{:}([c_1'(y)/x]K_2)L_2$ then by IH we have $\Gamma \vdash_\Sigma c_1 = c_1'{:}(L_1)K_1$ and $\Gamma, x{:}K_1 \vdash_\Sigma c_2 = c_2'{:}([c_1'(y)/x]K_2)L_2$ and we are done. ◀

2.2.2 Conservativity

Here we prove that, if the signatures are coherent, our calculus $\Pi_{S,\le}$ is conservative over Π_S in the traditional sense. It follows directly from the fact that $\Pi_{S,\le}$ keeps track of subtyping entries in the signatures and it carries them along in derivations. More precisely we prove that if a judgement is derivable in $\Pi_{S,\le}$ and not in Π_S then it cannot be written in Π_S.

The following two lemmas state that any subtyping or subkinding judgement can only be derived with a signature containing subtyping entries, and hence the signature cannot be written in Π_S.

▶ **Lemma 9.** *If* $\Gamma \vdash_\Sigma A \le_c B{:}Type$ *is derivable in* $\Pi_{S,\le}$, *then* Σ *contains at least a subtyping entry or* $\Gamma \vdash_\Sigma A = B{:}Type$ *and* $\Gamma \vdash_\Sigma c = Id{:}(A)A$ *are derivable in* $\Pi_{S,\le}$.

Proof. By induction on the structure of derivation. For example if it comes from transitivity from premises $\Gamma \vdash_\Sigma A \le_c A' : Type$ and $\Gamma \vdash_\Sigma A' \le_{c'} B : Type$ then the statement simply is true by IH. ◀

▶ **Lemma 10.** *If* $\Gamma \vdash_\Sigma K \leq_c L$ *is derivable in* $\Pi_{S,\leq}$*, then* Σ *contains at least a subtyping entry or* $\Gamma \vdash_\Sigma K = L$ *and* $\Gamma \vdash_\Sigma c = Id{:}(K)L$ *are derivable in* $\Pi_{S,\leq}$*.*

Proof. By induction on the structure of derivation. For example if it comes from transitivity from premises $\Gamma \vdash_\Sigma K \leq_c M$ and $\Gamma \vdash_\Sigma M \leq_{c'} L$ then the statement simply is true by IH.

If it comes from the rule

$$\frac{\Gamma \vdash_\Sigma A \leq_c B{:}Type}{\Gamma \vdash_\Sigma El(A) \leq_c El(B)}$$

then it follows from $\Gamma \vdash_\Sigma A \leq_c B{:}Type$ by the previous lemma ◀

The following lemma extends the statement to express the fact that it is enough for a judgement to contain a non trivial subtyping or subkinding entry (not the identity coercion) in its derivation tree to have a signature that cannot be written in Π_S.

▶ **Lemma 11.** *If* D *is a valid derivation tree for* $\Gamma \vdash_\Sigma J$ *in* $\Pi_{S,\leq}$ *and* $\Gamma_1 \vdash_{\Sigma_1} K_1 \leq_{c_0} K_2$ *is present in* D *then, either* Σ *contains at least a subtyping entry or* $\Gamma_1 \vdash_{\Sigma_1} K_1 = K_2$ *and* $\Gamma_1 \vdash_{\Sigma_1} c_0 = Id_{K_1}{:}(K_1)K_1$ *are derivable in* $\Pi_{S,\leq}$*.*

Proof. If $\Gamma \vdash_\Sigma J$ is a subtyping or subkinding judgement it follows directly from the previous lemmas 9, 5. Likewise, if the judgement comes from a coercive application or coercive definition rule with one of the premises $\Gamma \vdash_\Sigma K \leq L$, then, by the previous lemma the statement holds. Otherwise we do induction on the structure of derivations of $\Gamma \vdash_\Sigma J$. For example if the derivation tree containing the subkinding judgement ends with the rule

$$\frac{\Gamma \vdash_\Sigma K\, kind \quad \Gamma, x{:}K \vdash_\Sigma K'\, kind}{\Gamma \vdash_\Sigma (x{:}K)K' \;\; kind}$$

then the subkinding judgements must be in at least one of the subderivations concluding $\Gamma \vdash_\Sigma K \;\; kind$ and $\Gamma, x{:}K \vdash_\Sigma K' \;\; kind$. The statement then holds by induction hypothesis. ◀

The following lemma states that, if a judgements is derived in $\Pi_{S,\leq}$ using only trivial coercions, then it can be derived in Π_S.

▶ **Lemma 12.** *If in a derivation tree of a judgement derivable in* $\Pi_{S,\leq}$ *which is not subtyping or subkinding judgement all of the subtyping and subkinding judgements are of the form* $\Gamma_1 \vdash_{\Sigma_1} A \leq_{Id_A} A{:}Type$ *respectively* $\Gamma_1 \vdash_{\Sigma_1} K \leq_{[x:K]x} K$ *then the judgement is derivable in* Π_S*.*

Proof. By induction on the structure of derivations. If the derivation tree D that only contains trivial coercions ends with one of the rules of Π_S,

$$\frac{\frac{D_1}{J_1} \cdots \frac{D_n}{J_n}}{J}(R)$$

then J_1,..., J_n also have derivation trees D_1,...,D_n which only contain at most trivial coercions, hence, by IH, they are derivable in Π_S. We can apply to them, with D_1,...,D_n replaced by their derivation in Π_S the same rule R to obtain the judgement J and the derivation is in Π_S.

Otherwise, if for example the derivation containing only trivial coercions ends with coercive application

$$\frac{\Gamma \vdash_\Sigma f{:}(x{:}K)K' \quad \Gamma \vdash_\Sigma k_0{:}K \quad \Gamma \vdash_\Sigma K \leq_{[x:K]x} K}{\Gamma \vdash_\Sigma f(k_0){:}[[x{:}K]x(k_0)/x]K'}$$

$\Gamma \vdash_\Sigma [[x{:}K]x(k_0)/x]K' = [k_0/x]K'$ and $\Gamma \vdash_\Sigma f{:}(x{:}K)K'$ and $\Gamma \vdash_\Sigma k_0{:}K$ are derivable in Π_S by IH, and from them it follows directly by functional application, in Π_S, $\Gamma \vdash_\Sigma f(k_0){:}[k_0/x]K'$ ◀

▶ **Theorem 13** (Conservativity). *If a judgement is derivable in $\Pi_{S,\leq}$ but not in Π_S, its signature will contain subtyping entries, and hence it cannot be written in Π_S.*

Proof. From the previous lemma, a judgement can only be derivable in $\Pi_{S,\leq}$ but not in Π_S when it contains in all of its derivation trees non trivial subtyping or subkinding judgements. If the judgements is itself a subtyping or subkinding judgement then it vacuously cannot be written in Π_S. Otherwise, by lemma 11 it follows that either all of the subtyping and subkinding judgements are of the form $\Gamma_1 \vdash_{\Sigma_1} A \leq_{Id_A} A{:}Type$ respectively $\Gamma_1 \vdash_{\Sigma_1} K \leq_{[x:K]x} K$ in which case the judgement is derivable in Π_S or its signature contains subtyping entries, in which case it cannot be written in Π_S. ◀

2.3 Justification of $\Pi_{S,\leq}$ as a Well Behaved Extension

We shall show in this subsection that extending the type theory Π_S by coercive subtyping in signatures results in a well-behaved system. In order to do so, we relate the extension with the previous formulation: more precisely, for every signature Σ, we consider a corresponding system $\Pi[C_\Sigma]^{;}$, which is similar to the system $T[C_\Sigma]$ in [21, 33], and we prove the equivalence between judgements in $\Pi_{S,\leq}$ and judgements in such corresponding systems from the point of view of derivability. (see Theorems 22 and 29 below for a more precise description).

This way we argue that there exists a stronger relation between the extension with coercive subtyping entries and the base system based on the fact that was shown in [21, 33] that every derivation tree in $T[\mathcal{C}]$ the extension can be translated to a derivation tree in T such that their conclusion are equal.

2.3.1 The relation between $\Pi^{0K}_{S,\leq}$ and Π_S

Here we show that, if a judgement J is derivable in $\Pi^{0K}_{S,\leq}$, we obtain a set of judgements, one of which is of same as J up to erasing the subtyping entries from a signature. The idea here is that, for any the valid signature in $\Pi^{0K}_{S,\leq}$ and all the judgements using it, we can remove the subtyping entries from it to obtain a valid signature in Π_S and corresponding judgements using this signature.

▶ **Definition 14.** We define $erase(\cdot)$, a map which simply removes subtyping entries from signature as follows:

- $erase(<>) =<>$
- $erase(\Sigma, c{:}K) = erase(\Sigma), c{:}K$
- $erase(\Sigma, A \leq_c B) = erase(\Sigma)$

The following lemma is a completion of weakening and signature replacement for the cases when a signature is weakened with subtyping entries or a subtyping entry is replaced in the signature.

▶ **Lemma 15.**

- *If $\Gamma \vdash_{\Sigma,\Sigma'} J$ and $\vdash_{\Sigma,A\leq_c B:Type,\Sigma'} \Gamma$ are derivable in $\Pi^{0K}_{S,\leq}$ then $\Gamma \vdash_{\Sigma,A\leq_c B:Type,\Sigma'} J$ is derivable in $\Pi^{0K}_{S,\leq}$.*
- *If $\Gamma \vdash_{\Sigma,A\leq_c B\Sigma'} J$, $\vdash_\Sigma A = A'{:}Type$, $\vdash_\Sigma B = B'{:}Type$, $\vdash_\Sigma c = c'{:}(A)B$ $\vdash_{\Sigma,A'\leq_{c'} B':Type,\Sigma'} \Gamma$ are derivable in $\Pi^{0K}_{S,\leq}$ then $\Gamma \vdash_{\Sigma,A'\leq_{c'} B':Type,\Sigma'} J$ is derivable in$\Pi^{0K}_{S,\leq}$.*

Proof. By induction on the structure of derivation. ◀

▶ **Lemma 16.** *For $\Sigma \equiv \Sigma_0, A_0 \leq_{c_0} B_0, \Sigma_1, ..., A_{n-1} \leq_{c_{n-1}} B_{n-1}, \Sigma_n$ a valid signature as above we will consider the following judgements judgements $(\star) \vdash_{erase(\Sigma_0,...,\Sigma_i)} c_i{:}(A_i)B_i$, where $i \in 0, ..., n$. Then the following statements hold:*

1. *$\vdash_\Sigma \Gamma$ is derivable in $\Pi^{0K}_{S,\leq}$ if and only if $\vdash_{erase(\Sigma)} \Gamma$ and $(\star)$ are derivable in Π_S.*
2. *$\Gamma \vdash_\Sigma J$ is not a subtyping judgement and is derivable in $\Pi^{0K}_{S,\leq}$ if and only if $\Gamma \vdash_{erase(\Sigma)} J$ and $(\star)$ are derivable in Π_S.*
3. *If $\Gamma \vdash_\Sigma A \leq_c B$ is derivable in $\Pi^{0K}_{S,\leq}$ then $\Gamma \vdash_{erase(\Sigma)} c{:}(A)B$ and $(\star)$ are derivable in Π_S.*
4. *If $\Gamma \vdash_\Sigma K \leq_c L$ is derivable in $\Pi^{0K}_{S,\leq}$ then $\Gamma \vdash_{erase(\Sigma)} c{:}(K)L$ and $(\star)$ are derivable in Π_S.*

Proof. The only if implication for the first three cases is straightforward by induction on the structure of derivations as subtyping judgements do not contribute to deriving any other type of judgement in $\Pi^{0K}_{S,\leq}$. For the if implication, Lemma 15 is used. The last two points also follow by induction. ◀

2.3.2 $\Pi[\mathcal{C}]^;$

Here we consider a system $\Pi[\mathcal{C}]^;$ similar to the system $T[\mathcal{C}]$ as presented in [21, 33] with T being the type theory with Π-types.

Here we consider a system similar to the system $T[\mathcal{C}]$ from [21, 33] with dependent product. The difference is that here we fix some prefixes of the context, not allowing substitution and abstraction for these prefixes. In more details, the judgements of $T[\mathcal{C}]^;$ will be of the form $\Sigma; \Gamma \vdash J$ instead of $\Gamma \vdash J$, where Σ and Γ are just contexts and substitution and abstraction can be applied to entries in Γ but not Σ. We call this system $\Pi[\mathcal{C}]^;$. To delimitate these prefixes we use the symbol ";" and the judgements forms will be as follows:

- $\vdash \Sigma; \Gamma$ signifies a judgement of valid context
- $\Sigma; \Gamma \vdash K\ \ kind$
- $\Sigma; \Gamma \vdash k{:}K$
- $\Sigma; \Gamma \vdash K = K'$
- $\Sigma; \Gamma \vdash k = k'{:}K$

The rules of the system $\Pi[\mathcal{C}]^;$ are the ones in Figures 8,9,10, 11 and 12 in the appendix. The difference between these rules and those described in [21, 33] is that, in addition to regular contexts, they also refer to the prefixes apart from substitution and abstraction which is only available for regular contexts. More detailed, we duplicate contexts, assumptions, weakening, context replacement. For all other rules we adjust them to the new forms of judgements by replacing $\Gamma \vdash J$ with $\Sigma; \Gamma \vdash J$. Notice that we do not duplicate substitution as only the context at the righthand side of the ; supports substitution. We will consider the system $\Pi[\mathcal{C}]^;_{0K}$ to be the one without coercive application and definition rules, namely the ones in figures 8,9,10 and 11. $\mathcal{C}$ is formed of subtyping judgements and we have the following rule in $\Pi[\mathcal{C}]^;_{0K}$

$$\frac{\Gamma \vdash A \leq_c B \in \mathcal{C}}{\Gamma \vdash A \leq_c B}$$

For the system $T[\mathcal{C}]$ coercive application is added as an abbreviation to ordinary functional application and this is ensured by coercive definition together *coherence* of $\mathcal{C}$. Indeed, it was proved in [21, 33] that, when $\mathcal{C}$ is coherent, $\Pi[\mathcal{C}]$ is a well behaved extension of $\Pi[\mathcal{C}]_{0K}$ in that every valid derivation tree D in $\Pi[\mathcal{C}]$ can be translated into a valid derivation tree D' in $\Pi[\mathcal{C}]_{0K}$ and the conclusion of D is definitionally equal to the conclusion of D' in $\Pi[\mathcal{C}]$. We want to avoid doing the complex proof in [21, 33] again and assume that the properties of $\Pi[\mathcal{C}]$ carry over to $\Pi[\mathcal{C}]^;$. So next we give the definition of coherence for the set $\mathcal{C}$.

▶ **Definition 17.** The set $\mathcal{C}$ of subtyping judgements is coherent if the following two conditions hold in $\Pi[\mathcal{C}]^{;}_{0K}$:

- If $\Sigma;\Gamma \vdash A \leq_c B$ is derivable, then $\Sigma;\Gamma \vdash c{:}(A)B$ is derivable.
- If $\Sigma;\Gamma \vdash A \leq_c B$ and $\Sigma;\Gamma \vdash A \leq_{c'} B$ are derivable, then $\Sigma;\Gamma \vdash c = c'{:}(A)B$ is derivable.

Notice that in the original formulation $\Sigma;\Gamma \vdash A \leq_{[x:A]x} A$ was not allowed. However the condition that $\Sigma;\Gamma \nvdash A \leq_c A$ was used to prove that a judgement cannot come from both coercive application and functional application. However with the current condition one can prove that, if this is the case, the coercion has to be equal to the identity.

2.3.3 The relation between $\Pi[\mathcal{C}]^{;}$ and $\Pi_{S,\leq}$

Although there is a difference between the new $\Pi_{S,\leq}$ and $\Pi[\mathcal{C}]^{;}$ which lies mainly in the fact that, by introducing coercive subtyping via signature, we introduce them locally to the specific signature, this allowing us to have more coercions between two types under the same kinding assumptions(of the form c:K, x:K) and still have coherence satisfied, whereas by enriching a system with a set of coercive subtyping, our coercions are introduced globally and only one coercion(up to definitional equality) can exist between two types under the same kinding assumptions. However, because signatures are technically just prefix of contexts for which abstraction and substitution are not available [12], we naturally expect that there is a relation between $\Pi_{S,\leq}$ and $\Pi[\mathcal{C}]^{;}$. And indeed here we shall show that for any valid signature Σ in $\Pi_{S,\leq}$, we can represent a class of judgements of $\Pi_{S,\leq}$ depending on Σ as judgements in a $\Pi[\mathcal{C}_\Sigma]^{;}$.

First we consider just $\Pi^{0K}_{S,\leq}$ and $\Pi[\mathcal{C}]^{;}_{0K}$ which are the systems without coercive application and coercive definition and we define a way to transfer coercive subtyping entries of a signature Σ in $\Pi^{0K}_{S,\leq}$ to a set of coercive subtyping judgements of $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$.

▶ **Definition 18.** Let Σ be a signature (not necessarily valid) in $\Pi^{0K}_{S,\leq}$ we define Γ_Σ as follows:

- $\Gamma_{<>} = <>$
- $\Gamma_{\Sigma_0,k:K} = \Gamma_{\Sigma_0}, k{:}K$
- $\Gamma_{\Sigma_0,A\leq_c B:Type} = \Gamma_{\Sigma_0}$

If Σ is valid in $\Pi^{0K}_{S,\leq}$ we define $\mathcal{C}_\Sigma$ as follows:

- $\mathcal{C}_{<>} = \emptyset$
- $\mathcal{C}_{\Sigma_0,k:K} = \mathcal{C}_{\Sigma_0}$
- $\mathcal{C}_{\Sigma_0,A\leq_c B:Type} = \mathcal{C}_{\Sigma_0} \cup \{\Gamma_{\Sigma_0}; <>\vdash A \leq_c B{:}Type\}$

▶ **Lemma 19.** *If $\Sigma \equiv \Sigma_0, A \leq_c B{:}Type, \Sigma_1$ valid is derivable in $\Pi^{0K}_{S,\leq}$, then $\Gamma_\Sigma \equiv \Gamma_{\Sigma_0,\Sigma_1}$ and $\mathcal{C}_\Sigma = \mathcal{C}_{\Sigma_0,\Sigma_1} \cup \{\Gamma_{\Sigma_0}; <>\vdash A \leq_c B{:}Type\}$*

Proof. By induction on the length of Σ. ◀

▶ **Lemma 20.** *Let Σ_1, Σ_3 and $\Sigma_1, \Sigma_2, \Sigma_3$ be valid signatures in $\Pi^{0K}_{S,\leq}$. If J is derivable in $\Pi[\mathcal{C}_{\Sigma_1,\Sigma_3}]^{;}_{0K}$ then J is derivable in $\Pi[\mathcal{C}_{\Sigma_1,\Sigma_2,\Sigma_3}]^{;}_{0K}$*

Proof. By induction on the structure of derivation of J. ◀

First we mention the following notation which we will use throughout the section and which is really just a generalization of definitional equality:

- $<>=<>$, $\Sigma, c{:}K = \Sigma', c{:}K'$ iff $\Sigma = \Sigma'$ and $\vdash_\Sigma K = K'$
- $\vdash_\Sigma \Gamma, x{:}K =\vdash_{\Sigma'} \Gamma, x{:}K'$ iff $\vdash_\Sigma \Gamma =\vdash_{\Sigma'} \Gamma$ and $\Gamma \vdash_\Sigma K = K'$
- $\Gamma \vdash_\Sigma K = \Gamma' \vdash_{\Sigma'} K'$ iff $\vdash_\Sigma \Gamma =\vdash_{\Sigma'} \Gamma'$ and $\Gamma \vdash_\Sigma K = K'$
- $\Gamma \vdash_\Sigma k{:}K = \Gamma' \vdash_{\Sigma'} k'{:}K'$ iff $\Gamma \vdash_\Sigma K \;\; kind = \Gamma' \vdash_{\Sigma'} K' \;\; kind$ and $\Gamma \vdash_\Sigma k = k'{:}K$

- $\Gamma \vdash_\Sigma k = l{:}K = \Gamma' \vdash_{\Sigma'} k' = l'{:}K'$ iff $\Gamma \vdash_\Sigma K\ \ kind = \Gamma' \vdash_{\Sigma'} K'\ \ kind$ and $\Gamma \vdash_\Sigma k = k'{:}K$ and $\Gamma \vdash_\Sigma l = l'{:}K$
- $\Gamma \vdash_\Sigma A \leq_c B = \Gamma' \vdash_{\Sigma'} A' \leq_{c'} B'$ iff $\Gamma \vdash_\Sigma A{:}Type = \Gamma' \vdash_{\Sigma'} A'{:}Type$ and $\Gamma \vdash_\Sigma B{:}Type = \Gamma' \vdash_{\Sigma'} B'{:}Type$ and $\Gamma \vdash_\Sigma c{:}(A)B = \Gamma' \vdash_{\Sigma'} c'{:}(A')B'$

We consider the analogous notation for judgements of the form $\vdash \Gamma_0; <>$, $\vdash \Gamma_0; \Gamma$ and $\Gamma_0; \Gamma \vdash J$. We will say that the judgements are definitionally equal in a certain system if all the corresponding definitional equality judgements are derivable in that system.

According to [21, 33], if we add coercive subtyping and coercive definition rules from Figure 12 in the appendix to a system enriched with a coherent set of subtyping judgements $\mathcal{C}_\Sigma$, any derivation tree in $\Pi[\mathcal{C}_\Sigma]^{;}$ can be translated to a derivation tree in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ (that is a derivation tree that does not use coercive application and definition rules - CA_1, CA_2 and CD) and their conclusions are definitionally equal. We aim to use that result to prove that for any judgement using a coherent signature in $\Pi_{S,\leq}$, there exists a judgement definitionally equal to it in $\Pi^{0K}_{S,\leq}$. For this we shall first prove that $\mathcal{C}_\Sigma$ is coherent in the sense of the definition 17 if Σ is coherent in the sense of the definition 1. To prove this we need to describe the possible contexts at the lefthand side of ; in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ used to infer coercive subtyping judgements.

We first prove a theorem used throughout the section which allows us to argue about judgements in $\Pi^{0K}_{S,\leq}$ and judgements in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ interchangeably. We start by presenting a lemma representing the base case and then the theorem appears as an extension easily proven by induction. The lemma is not required to prove the theorem but it gives a better intuition. The theorem essentially states that for contexts at the lefthand side of ; obtained by interleaving membership entries in a the image through $\Gamma.$ of a valid signature Σ or its prefixes give judgements in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ corresponding to judgements in $\Pi^{0K}_{S,\leq}$. We will see later that all the contexts at the lefthand side of ; in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ are in fact obtained by interleaving membership entries in prefixes of Σ.

▶ **Lemma 21.** *Let $\Sigma \equiv \Sigma_1, \Sigma_2, \Sigma_3$ be a valid signature in $\Pi^{0K}_{S,\leq}$ then, for any c, K and Σ'_1, Σ'_2 s.t. $\Sigma_1 = \Sigma'_1$ and $\Sigma_1, \Sigma_2 = \Sigma'_1, \Sigma'_2$ the following hold:*

- *$\vdash \Gamma_{\Sigma'_1}, c{:}K, \Gamma_{\Sigma'_2}; <>$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ iff $\Sigma'_1, c{:}K, \Sigma'_2$ valid is derivable in $\Pi^{0K}_{S,\leq}$*
- *$\vdash \Gamma_{\Sigma'_1}, c{:}K, \Gamma_{\Sigma'_2}; \Gamma$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ iff $\vdash_{\Sigma'_1, c:K, \Sigma'_2} \Gamma$ is derivable in $\Pi^{0K}_{S,\leq}$*
- *$\Gamma_{\Sigma'_1}, c{:}K, \Gamma_{\Sigma'_2}; \Gamma \vdash J$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ iff $\Gamma \vdash_{\Sigma'_1, c:K, \Sigma'_2} J$ is derivable in $\Pi^{0K}_{S,\leq}$.*

Proof. By induction on the structure of derivation. ◀

Mainly by repeatedly applying the previous lemma (except for the case when we weaken with the empty sequence, which is straight forward by induction on the structure of derivations) we can prove:

▶ **Theorem 22** (Equivalence for $\Pi^{0K}_{S,\leq}$). *Let $\Sigma \equiv \Sigma_1, ..., \Sigma_n$ bea valid signature in $\Pi^{0K}_{S,\leq}$ then, for any $1 \leq k \leq n$, for any $\{\Gamma_i\}_{i \in \{0..k\}}$ sequences free of subtyping entries and and $\Sigma'_1, ..., \Sigma'_k$ s.t. $\Sigma_1, ..., \Sigma_k = \Sigma'_1, ..., \Sigma'_k$ for any $i \in \{1..k\}$ the following hold:*

- *$\vdash \Gamma_0, \Gamma_{\Sigma'_1}, \Gamma_1, \Gamma_{\Sigma'_2}, \Gamma_2, ..., \Gamma_{k-1}\Gamma_{\Sigma'_k}, \Gamma_k; <>$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ if and only if $\Gamma_0, \Sigma'_1, \Gamma_1, \Sigma'_2, \Gamma_2, ..., \Gamma_{k-1}, \Sigma'_k, \Gamma_k$ valid is derivable in $\Pi^{0K}_{S,\leq}$*
- *$\vdash \Gamma_0, \Gamma_{\Sigma'_1}, \Gamma_1, \Gamma_{\Sigma'_2}, \Gamma_2, ..., \Gamma_{k-1}\Gamma_{\Sigma'_k}, \Gamma_k; \Gamma$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ if and only if $\vdash_{\Gamma_0, \Sigma'_1, \Gamma_1, \Sigma'_2, \Gamma_2, ..., \Gamma_{k-1}\Sigma'_k, \Gamma_k} \Gamma$ is derivable in $\Pi^{0K}_{S,\leq}$*
- *$\Gamma_0, \Gamma_{\Sigma'_1}, \Gamma_1, \Gamma_{\Sigma'_2}, \Gamma_2, ..., \Gamma_{k-1}\Gamma_{\Sigma'_k}, \Gamma_k; \Gamma \vdash J$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ if and only if $\Gamma \vdash_{\Gamma_0, \Sigma'_1, \Gamma_1, \Sigma'_2, \Gamma_2, ..., \Gamma_{k-1}\Sigma'_k, \Gamma_k} J$ is derivable in $\Pi^{0K}_{S,\leq}$.*

Now we aim to prove that we do not introduce any new subtyping entries in $\Pi^{0K}_{S,\leq}$ by weakening (up to definitional equality). Note that, for this, it is essential that the weakening rules do not add subtyping entries. More precisely, in the following Lemma we prove a form of strengthening, which roughly says that by strengthening the assumptions of a subtyping judgement, we can still derive it(up to definitional equality).

▶ **Lemma 23.** *Let* $\Sigma \equiv \Sigma_1, \Sigma_2$ *a valid signature in* $\Pi^{0K}_{S,\le}$*, for any* c, K*,* $\Sigma'_1 = \Sigma_1$ *and* $\Sigma'_1, \Sigma'_2 = \Sigma_1, \Sigma_2$*, if* $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A \le_c B$ *is derivable in* $\Pi^{0K}_{S,\le}$ *then there exists* A', c', B' *such that* $\vdash_\Sigma A' \le_{c'} B'$*,* $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A = A'{:}Type$*,* $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} B = B'{:}Type$ *and* $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} c = c'{:}(A)B$ *derivable in* $\Pi^{0K}_{S,\le}$*.*

Proof. By induction on the structure of derivation of $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A \le_c B$. If it comes from transitivity with the premises $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A \le_{c_1} C$ and $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} C \le_{c_2} B$ then by IH, there exist $A', C', c'_1, C'', B', c'_2$ s.t. $\vdash_\Sigma A' \le_{c'_1} C'$ and $\vdash_\Sigma C'' \le_{c'_2} B'$ and $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A = A'{:}Type$, $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} B = B'{:}Type$, $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} C = C'{:}Type$, $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} C = C''{:}Type$, $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} c_1 = c'_1{:}(A)C$ and $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} c_2 = c'_2{:}(C)B$. By transitivity of equality we have $\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} C' = C''{:}Type$. By lemma 16 we have that $\Gamma \vdash_{erase(\Sigma'_1,k:K,\Sigma'_2)} C' = C''{:}Type$ is derivable in Π_S. Similarly, because $\vdash_\Sigma C'{:}Type$ and $\vdash_\Sigma C''{:}Type$ we have that $\vdash_{erase(\Sigma'_1,k:K,\Sigma'_2)} C'{:}Type$ and $\vdash_{erase(\Sigma'_1,k:K,\Sigma'_2)} C''{:}Type$ are derivable in Π_S. From Strengthening Lemma([11]) which holds for Π_S we have that $\vdash_{erase(\Sigma)} C' = C''{:}Type$. Again, by 16 we obtain $\vdash_\Sigma C' = C''{:}Type$. At last, we can apply congruence and transitivity $\vdash_\Sigma A' \le_{c'_2 \circ c'_1} B'$.

Let us now consider the dependent product rule

$$\frac{\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A'' \le_{c_1} A' \quad \Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} B', B'' : (A')Type \quad \Gamma, x{:}A' \vdash_{\Sigma'_1,k:K,\Sigma'_2} B'(x) \le_{c_2[x]} B''(x)}{\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} \Pi(A', B') \le_c \Pi(A'', B'' \circ c_1)}$$

with $A \equiv \Pi(A', B')$, $B \equiv \Pi(A'', B'' \circ c_1)$ and

$$c \equiv [F : \Pi(A', B')]\lambda(A'', B'' \circ c_1, [x{:}A'']c_2[x](app(A', B', F, c_1(x)))).$$

By IH, there exist $A''_0, A'_0, c'_1, B'_0, B''_0, c'_2$ s.t. $\vdash_\Sigma A''_0 \le_{c'_1} A'_0$, $\vdash_\Sigma B' \le_{c'_2} B''$ and

$$\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A'' = A''_0{:}Type,\ \Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A' = A'_0{:}Type,$$
$$\Gamma, x{:}A' \vdash_{\Sigma'_1,k:K,\Sigma'_2} B''(x) = B''_0{:}Type,\ \Gamma, x{:}A' \vdash_{\Sigma'_1,k:K,\Sigma'_2} B'(x) = B'_0{:}Type,$$
$$\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} c_1 = c'_1{:}(A'')A' \text{ and } \Gamma, x{:}A' \vdash_{\Sigma'_1,k:K,\Sigma'_2} c_2(x) = c'_2{:}(B'(x))B''(x){:}Type.$$

We apply dependent product rule for the case when types are constants and obtain

$$\vdash A'_0 \longrightarrow B'_0 \le'_c A''_0 \longrightarrow B''_0 \text{ with } c' \equiv [F : A'_0 \longrightarrow B'_0][x{:}A''_0](c''_2(F(c'_1(x)))).$$

By normal equality rules for dependent product and its terms we have that

$$\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A'_0 \longrightarrow B'_0 = \Pi(A', B'),\ \Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} A''_0 \longrightarrow B''_0 = \Pi(A'', B'') \text{ and}$$
$$\Gamma \vdash_{\Sigma'_1,k:K,\Sigma'_2} c = c'{:}(\Pi(A', B'))\Pi(A'', B'')$$ ◀

By repeatedly applying the previous lemma we obtain

▶ **Corollary 24.** *For* Σ *valid derivable in* $\Pi^{0K}_{S,\le}$*,* $\Sigma \equiv \Sigma_1, ..., \Sigma_n$*, for any* $\{\Gamma_i\}_{i\in\{0..n\}}$ *sequences free of subtyping entries and* $\{\Sigma'_i\}_{i\in\{1..n\}}$ *s.t.* $\Sigma_1, ..., \Sigma_i = \Sigma'_1, ..., \Sigma'_i$ *for any* $i \in \{1..n\}$*, if* $\Gamma \vdash_{\Gamma_0,\Sigma_1,\Gamma_1,\Sigma_2,\Gamma_2,...,\Gamma_{n-1}\Sigma_n,\Gamma_n} A \le_c B$ *is derivable in* $\Pi^{0K}_{S,\le}$ *then there exists* A', c', B' *s.t.* $\vdash_\Sigma A' \le_{c'} B'$*,* $\Gamma \vdash_{\Gamma_0,\Sigma_1,\Gamma_1,\Sigma_2,\Gamma_2,...,\Gamma_{n-1}\Sigma_n,\Gamma_n} A = A'{:}Type$*,* $\Gamma \vdash_{\Gamma_0,\Sigma_1,\Gamma_1,\Sigma_2,\Gamma_2,...,\Gamma_{n-1}\Sigma_n,\Gamma_n} B = B'{:}Type$ *and* $\Gamma \vdash_{\Gamma_0,\Sigma_1,\Gamma_1,\Sigma_2,\Gamma_2,...,\Gamma_{n-1}\Sigma_n,\Gamma_n} c = c'{:}(A)B$ *derivable in* $\Pi^{0K}_{S,\le}$*.*

Next we prove that weakening does not break coherence:

▶ **Lemma 25.** *For Σ valid in $\Pi^{0K}_{S,\leq}$, if $\Sigma \equiv \Sigma_1, \Sigma_2, \Sigma_3$ is coherent, for any Σ'_1, Σ'_2 s.t. $\Sigma_1 = \Sigma'_1$ and $\Sigma_1, \Sigma_2 = \Sigma'_1, \Sigma'_2$, for any c, K s.t. $\Sigma'_1, c{:}K,, \Sigma'_2$ is valid, $\Sigma'_1, c{:}K,, \Sigma'_2$ coherent.*

Proof. Let us consider the derivable judgements $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2} A \leq_c B$ and $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2} A \leq_d B$. Then we know from Lemma 24 that there exist A', B', A'', B'', c', d' s.t. $\vdash_{\Sigma_1, \Sigma_2} A' \leq_{c'} B'$, $\vdash_{\Sigma_1, \Sigma_2} A'' \leq_{d'} B''$, $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2} A' = A{:}Type$, $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2} B'' = B{:}Type$, $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2 s} B'' = B{:}Type$ $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2} c = c'{:}(A)B$ and $\Gamma \vdash_{\Sigma'_1, c:K,, \Sigma'_2} d = d'{:}(A)B$. As in the proof of the previous lemma, using Lemma 16 and Strengthening Lemma from [11] we have that $\vdash_{\Sigma_1, \Sigma_2} A' = A''{:}Type$, $\vdash_{\Sigma_1, \Sigma_2} B' = B''{:}Type$. By congruence we have that $\vdash_{\Sigma_1, \Sigma_2} A' \leq_{d'} B'$ is derivable in $\Pi^{0K}_{S,\leq}$. If Σ is coherent then any prefix of it $\Sigma_1, ..., \Sigma_k$ is coherent so $\vdash_{\Sigma_1, \Sigma_2} c' = d'{:}(A')B'$. Further, by weakening and Lemma 15, we have the desired result. ◀

By repeatedly applying the previous lemma we obtain:

▶ **Lemma 26.** *For Σ valid in $\Pi^{0K}_{S,\leq}$, if $\Sigma \equiv \Sigma_1, ..., \Sigma_n$ is coherent, for any $1 \leq k \leq n$, for any $\{\Gamma_i\}_{i \in \{0..k\}}$ sequences free of subtyping entries, for any $\{\Sigma'_i\}_{i \in \{0..k\}}$ s.t. $\Sigma_1, ..., \Sigma_i = \Sigma'_1, ..., \Sigma'_i$ for any $i \in \{1..k\}$ s.t. $\Gamma_0, \Sigma_1, \Gamma_1, \Sigma_2, \Gamma_2, ..., \Gamma_{k-1}, \Sigma_k, \Gamma_k$ is valid, $\Gamma_0, \Sigma_1, \Gamma_1, \Sigma_2, \Gamma_2, ..., \Gamma_{k-1}, \Sigma_k, \Gamma_k$ is coherent.*

Finally, the following lemma describes the relation between parts of the context at the lefthand side of the ; of judgements in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ and Σ. This is a very important result for proving the coherence of $\mathcal{C}_\Sigma$ based on the coherence of Σ. It states that any such context is in fact obtained from weakening of a prefix of Σ. In addition from this Lemma, because all the derivable judgements in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ that are not in $\Pi^{;}$ are subtyping judgements, we have as a consequence that all the judgements of $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ are equivalent to judgements in $\Pi^{0K}_{S,\leq}$.

▶ **Lemma 27.** *For Σ a valid signature in $\Pi^{0K}_{S,\leq}$, for any derivable judgement $\Gamma'; \Gamma \vdash J$ in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ there exists a partition of $\Sigma \equiv \Sigma_1, ..., \Sigma_n$, $1 \leq k \leq n$, $\Gamma_0, ..., \Gamma_k$ free of subtyping entries and $\Sigma'_1, ..., \Sigma'_k$ with $\Sigma'_1, ..., \Sigma'_i = \Sigma'_1, ..., \Sigma'_i$ for any $1 \leq i \leq k$ s.t. $\Gamma' \equiv \Gamma_{\Gamma_0, \Sigma_1, \Gamma_1, ..., \Sigma_k, \Gamma_k}$*

Proof. By induction on the structure of derivation of the judgement in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$. We only prove a case for third point when the judgement is $\Gamma'; \Gamma \vdash A \leq_c B$. The only nontrivial case is when the judgements follows from weakening. Let us assume it comes from a derivation tree ending with

$$\frac{\Gamma'_1, \Gamma'_2; \Gamma \vdash A \leq_c B \quad \Gamma'_1; <>\vdash K \ \ kind}{\Gamma'_1, c{:}K, \Gamma'_2; \Gamma \vdash A \leq_c B}$$

with $\Gamma' \equiv \Gamma_1, c{:}K, \Gamma_2$. By IH we know that there exists a partition of $\Sigma \equiv \Sigma_1, ..., \Sigma_n$ and $1 \leq k \leq n$ and $\Gamma_0, ..., \Gamma_k$ and $\Sigma'_1, ..., \Sigma'_k$ with $\Sigma'_1, ..., \Sigma'_i = \Sigma'_1, ..., \Sigma'_i$ for any $1 \leq i \leq k$ s.t. $\Gamma'_1, \Gamma'_2 \equiv \Gamma_{\Gamma_0, \Sigma'_1, \Gamma_1, ..., \Sigma'_k, \Gamma_k}$ with $\Gamma \vdash_{\Gamma_0, \Sigma'_1, \Gamma_1 ..., \Sigma'_k, \Gamma_k} A \leq_c B$. Let us consider the case when $\Gamma'_1 \equiv \Gamma_{\Gamma_0, \Sigma'_1, \Gamma_1, ..., \Gamma_{i-1}, \Sigma^{1\prime}_i}$ and $\Gamma'_2 \equiv \Gamma_{\Sigma^{2\prime}_i, \Gamma_i, ..., \Sigma'_k, \Gamma_k}$. With $\Sigma'_i \equiv \Sigma^{1\prime}_i, \Sigma^{2\prime}_i$ for some $1 \leq i \leq k$. We consider the partition of $\Sigma \equiv \Sigma_1, ..., \Sigma^1_i, \Sigma^2_i, ..., \Sigma_n$ s.t. $\Sigma'_1, ..., \Sigma^{1\prime}_i, \Sigma^{2\prime}_i, ..., \Sigma'_n$ $\Sigma_1, ..., \Sigma_l = \Sigma'_1, ..., \Sigma'_l$ for any $l \in 1..i-1$, $\Sigma_1, ..., \Sigma^1_i = \Sigma'_1, ..., \Sigma^{1\prime}_i$, $\Sigma_1, ..., \Sigma^1_i, \Sigma^2_i = \Sigma'_1, ..., \Sigma^{1\prime}_i, \Sigma^{2\prime}_i$ and $\Sigma_1, ..., \Sigma_l = \Sigma'_1, ..., \Sigma'_l$ for any $l \in i+1..n$ and $\Gamma_0, ..., \Gamma_{i-1}, c{:}K, \Gamma_i, ..., \Gamma_k$ s.t. $\Gamma' = \Gamma_{\Gamma_0, \Sigma_1, ..., \Gamma_{i-1}, \Sigma^1_i, c:K, \Sigma^2_i, \Gamma_i, ..., \Sigma_k, \Gamma_k}$. ◀

The next lemma refers to the ability to argue about coherence of a set of coercive subtyping judgements corresponding to a signature.

▶ **Theorem 28** (Equivalence of Coherence). *Let Σ be a valid signature in $\Pi^{0K}_{S,\leq}$. Then Σ is coherent in the sense of the Definition 1 iff $\mathcal{C}_\Sigma$ is coherent for $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ in the sense of the Definition 17.*

Proof. Only if: Let $\Gamma';\Gamma \vdash A \leq_c B$ and $\Gamma';\Gamma \vdash A \leq_d B$ be derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$. From Lemma 27, it follows that there exists a partition of $\Sigma \equiv \Sigma_1, ..., \Sigma_n$ and $1 \leq k \leq n$ and $\Gamma_0, ..., \Gamma_k$ s.t. $\Gamma' = \Gamma_{\Gamma_0,\Sigma_1,...,\Sigma_k,\Gamma_k}$. If Σ is coherent, then $\Gamma_0, \Sigma_1, ..., \Sigma_k, \Gamma_k$ is coherent (from Lemma 26). From Theorem 22, $\Gamma';\Gamma \vdash A \leq_c B$ and $\Gamma';\Gamma \vdash A \leq_d B$ are derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ iff $\Gamma \vdash_{\Gamma_0,\Sigma_1,...,\Sigma_k,\Gamma_k} A \leq_c B$ and $\Gamma \vdash \Gamma_0, \Sigma_1, ..., \Sigma_k, \Gamma_k A \leq_d B$ are derivable in $\Pi^{0K}_{S,\leq}$. From coherence here we have $\Gamma \vdash_{\Gamma_0,\Sigma_1,...,\Sigma_k,\Gamma_k} c = d{:}(A)B$ which is derivable in $\Pi^{0K}_{S,\leq}$ iff $\Gamma';\Gamma \vdash c = d{:}(A)B$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ (again by Theorem Theorem 22).

If: By Theorem 22, $\Gamma \vdash_\Sigma A \leq_c B{:}Type$ and $\Gamma \vdash_\Sigma A \leq_d B{:}Type$ are derivable in $\Pi^{0K}_{S,\leq}$ iff $\Gamma_\Sigma;\Gamma \vdash A \leq_c B$ and $\Gamma_\Sigma;\Gamma \vdash A \leq_d B$ are derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$. Because $\mathcal{C}_\Sigma$ is coherent, $\Gamma_\Sigma;\Gamma \vdash c = d{:}(A)B$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ which happens iff $\Gamma \vdash_\Sigma c = d{:}(A)B$ is derivable in $\Pi^{0K}_{S,\leq}$ ◀

To prove that the system $\Pi_{S,\leq}$ is well behaved we first prove that it is well behaved when all the signatures considered are valid in the restricted system $\Pi^{0K}_{S,\leq}$. First we prove another equivalence lemma for this situation.

▶ **Theorem 29** (Equivalence for $\Pi_{S,\leq}$). *For Σ valid in $\Pi^{0K}_{S,\leq}$, the following hold:*

- $\vdash \Gamma_\Sigma;\Gamma$ *is derivable in* $\Pi[\mathcal{C}_\Sigma]^{;}$ *iff* $\vdash_\Sigma \Gamma$ *is derivable in* $\Pi_{S,\leq}$
- $\Gamma_\Sigma;\Gamma \vdash J$ *is derivable in* $\Pi[\mathcal{C}_\Sigma]^{;}$ *iff* $\Gamma \vdash_\Sigma J$ *is derivable in* $\Pi_{S,\leq}$.

Proof. By induction on the structure of derivation. ◀

The following theorem shows that the system we defined here is well behaved and that every coercive subtyping application is really just an abbreviation.

▶ **Lemma 30.** *If a valid signature Σ in $\Pi^{0K}_{S,\leq}$ is coherent the following hold:*

1. *If $\vdash_\Sigma \Gamma$ is derivable in $\Pi_{S,\leq}$ then there exists Γ' s.t. $\vdash_\Sigma \Gamma'$ is derivable in $\Pi^{0K}_{S,\leq}$ and $\vdash_\Sigma \Gamma = \Gamma'$ is derivable in $\Pi_{S,\leq}$.*
2. *If $\Gamma \vdash_\Sigma J$ is derivable in $\Pi_{S,\leq}$ then there exists Γ', J' s.t. $\Gamma' \vdash_\Sigma J'$ is derivable in $\Pi^{0K}_{S,\leq}$ and $\vdash_\Sigma \Gamma = \Gamma'$ and $\Gamma \vdash_\Sigma J = J'$ are derivable in $\Pi_{S,\leq}$.*

Proof. By Theorem 28, since Σ is coherent in, $\mathcal{C}_\Sigma$ is coherent. If we look at the last case, by Theorem 29, $\Gamma \vdash_\Sigma J$ is derivable in $\Pi_{S,\leq}$ iff $\Gamma_\Sigma;\Gamma \vdash J$ is derivable in $\Pi[\mathcal{C}_\Sigma]^{;}$. From [21, 33] we know that, when $\mathcal{C}_\Sigma$ is coherent, any derivation tree of $\Gamma_\Sigma;\Gamma \vdash J$ can be translated into a derivation tree in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ which concludes with the judgement definitionally equal to $\Gamma_\Sigma;\Gamma \vdash J$. So let us consider one such derivation tree, its translation and the definitionally equal conclusion $\Gamma_\Sigma;\Delta \vdash J'$ ($\vdash \Gamma_\Sigma;<>$ is already derivable in $\Pi[\mathcal{C}_\Sigma]^{;}_{0K}$ so by inspecting the definition of the translation in [21, 33] we observe that Γ_Σ will not be changed by the translation). We have $\vdash \Gamma_\Sigma;\Gamma = \Gamma_\Sigma;\Delta$ and $\Gamma_\Sigma;\Gamma \vdash J = J'$ are derivable in $\Pi[\mathcal{C}_\Sigma]^{;}$. From Lemma 29 we know that in this case $\vdash_\Sigma \Gamma = \Delta$ and $\Gamma \vdash_\Sigma J = J'$ are derivable in $\Pi_{S,\leq}$ so the desired derivable judgement is simply $\Delta \vdash_\Sigma J'$. ◀

Note that the previous theorem covers the well-behavedness of judgements derived under a signature that is valid in $\Pi^{0K}_{S,\leq}$. We now prove further that any signature valid in $\Pi_{S,\leq}$ is definitionally equal to a signature valid in $\Pi^{0K}_{S,\leq}$, then because of signature replacement we have that any judgement derivable in in $\Pi_{S,\leq}$ is definitionally equal to a judgement derivable in $\Pi^{0K}_{S,\leq}$.

▶ **Lemma 31.** *For any signature Σ valid in $\Pi_{S,\leq}$ there exists Σ' valid in $\Pi^{0K}_{S,\leq}$ s.t. $\Sigma = \Sigma'$ in $\Pi_{S,\leq}$*

Proof. By induction on the length of Σ. We assume $\Sigma = \Sigma_0, c{:}K$. By IH we have that there exists Σ'_0 valid in $\Pi^{0K}_{S,\leq}$ s.t. $\Sigma_0 = \Sigma'_0$. By repeatedly applying signature replacement to $\vdash_{\Sigma_0} K\ \ kind$ we have $\vdash_{\Sigma'_0} K\ \ kind$ is derivable in $\Pi_{S,\leq}$. By Theorem 30, we have that there exists K' s.t. $\vdash_{\Sigma'_0} K'\ \ kind$ is derivable in $\Pi^{0K}_{S,\leq}$ with $\vdash_{\Sigma'_0} K = K'$. That means we can derive, in $\Pi^{0K}_{S,\leq}$, $\Sigma'_0, c{:}K'\ \ valid$. Going back with context replacement we also have $\vdash_{\Sigma_0} K = K'$ derivable, so $\Sigma'_0, c{:}K'$ is the signature we are looking for. ◀

We finish this section with the following theorem:

▶ **Theorem 32.** *If a valid signature Σ in $\Pi_{S,\leq}$ is coherent the following hold:*

1. *If $\vdash_\Sigma \Gamma$ is derivable in $\Pi_{S,\leq}$ then there exists Σ', Γ' s.t. $\vdash_{\Sigma'} \Gamma'$ is derivable in $\Pi^{0K}_{S,\leq}$ and $\Sigma = \Sigma'$ and $\vdash_\Sigma \Gamma = \Gamma'$ are derivable in $\Pi_{S,\leq}$.*
2. *If $\Gamma \vdash_\Sigma J$ is derivable in $\Pi_{S,\leq}$ then there exists Σ', Γ', J' s.t. $\Gamma' \vdash_{\Sigma'} J'$ is derivable in $\Pi^{0K}_{S,\leq}$ and $\Sigma = \Sigma'$, $\vdash_\Sigma \Gamma = \Gamma'$ and $\Gamma \vdash_\Sigma J = J'$ are derivable in $\Pi_{S,\leq}$.*

Proof. According to the Lemma 31 there exist Σ' valid in $\Pi^{0K}_{S,\leq}$ s.t. $\Sigma = \Sigma'$. If we consider the last point, by signature replacement $\Gamma \vdash_{\Sigma'} J$ is derivable $\Pi_{S,\leq}$. Because Σ' valid in $\Pi^{0K}_{S,\leq}$, we can apply the Lemma 30 to obtain $\Gamma' \vdash_{\Sigma'} J'$ s.t. $\vdash_{\Sigma'} \Gamma = \Gamma'$ and $\Gamma \vdash_{\Sigma'} J = J'$ are derivable in $\Pi_{S,\leq}$. Again by signature replacement $\vdash_\Sigma \Gamma = \Gamma'$ and $\Gamma \vdash_\Sigma J = J'$. ◀

Further, according to the lemma 16, the derivability of any nonsubtyping judgement in $\Pi^{0K}_{S,\leq}$ is equivalent to the derivability of a judgement in Π_S and any subtyping judgement in $\Pi^{0K}_{S,\leq}$ implies a judgement in Π_S.

3 Embedding Subsumptive Subtyping

In this section, we consider how to embed subsumptive subtyping into coercive subtyping. To this end, we consider a subtyping system which is a reformulation of the one studied by [2] and show how it can be faithfully embedded into our system of coercive subtyping.

We consider a system analogous to Π_S with the difference that we leave out the signatures. The types of judgements in this system are Γ *valid*, $\Gamma \Vdash K\ kind$, $\Gamma \Vdash k{:}K$, $\Gamma \Vdash K = K'$ and $\Gamma \Vdash k = k'{:}K$ syntactically analogous to $\vdash_{<>} \Gamma$, $\Gamma \vdash_{<>} K\ kind$, $\Gamma \vdash_{<>} k{:}K$, $\Gamma \vdash_{<>} K = K'$ respectively $\Gamma \vdash_{<>} k = k'{:}K$, baring rules analogous to the ones in the appendix and Figure 2. Note that there will be no Signature Validity and Assumption rules as there are no signatures. On top of these judgements we add $\Gamma \Vdash A \leq B\ type$ and $\Gamma \Vdash K \leq K'$ obtained with the rules from Figure 7. Besides the ordinary variables in Π, we allow Γ to have subtyping variables like $\alpha \leq A$. We name this extension $\Pi_\leq$.

$\Pi_\leq$ is the subsumptive subtyping system specified in LF that corresponds to the system $\lambda P_\leq$ in [2]. There are some subtle differences between Edinburgh LF (λP) [12] and the logical framework LF we use (eg, the η-rule holds for the latter but not the former), but they are irrelevant to the point we are trying to show: the subsumptive subtyping system can be faithfully embedded in the coercive subtyping system.

Once we introduced this system we will proceed by giving an interpretation of it in the coercive subtyping system that we introduced in section 2, namely we will show that this calculus can be faithfully embedded in the coercive subtyping one.

We mentioned that, in this system, an important thing to note is how placing subtyping entries in contexts interferes with abstraction and hence dependent types, specifically, the abstraction is not allowed at the lefthand side of subtyping entries. We will give a mapping

General Subtyping Rules

$$\frac{\Gamma \Vdash K = K'}{\Gamma \Vdash K \leq K'} \quad \frac{\Gamma \Vdash K \leq K' \quad \Gamma \Vdash K' \leq K''}{\Gamma \Vdash K \leq K''} \quad \frac{\Gamma \Vdash A = B{:}Type}{\Gamma \Vdash A \leq B{:}Type}$$

$$\frac{\Gamma \Vdash A \leq B{:}Type \quad \Gamma \Vdash B \leq C{:}Type}{\Gamma \Vdash A \leq C{:}Type}$$

Subtyping in Contexts

$$\frac{\Gamma \Vdash A{:}Type \quad \alpha \notin FV(\Gamma)}{\Gamma, \alpha \leq A \;\; valid} \quad \frac{\Gamma, \alpha \leq A, \Gamma' \;\; valid}{\Gamma, \alpha \leq A, \Gamma' \Vdash \alpha{:}Type} \quad \frac{\Gamma, \alpha \leq A, \Gamma' \;\; valid}{\Gamma, \alpha \leq A, \Gamma' \Vdash \alpha \leq A{:}Type}$$

Type Lifting and Subtyping

$$\frac{\Gamma \Vdash A \leq B{:}Type}{\Gamma \Vdash El(A) \leq El(B)} \quad \frac{\Gamma \Vdash k{:}K \quad \Gamma \Vdash K \leq K'}{\Gamma \Vdash k{:}K'} \quad \frac{\Gamma \Vdash k = k'{:}K \quad \Gamma \Vdash K \leq K'}{\Gamma \Vdash k = k'{:}K'}$$

Dependent Product

$$\frac{\begin{array}{c}\Gamma \Vdash \Pi(A,B){:}Type \quad \Gamma \Vdash \Pi(A',B'){:}Type \\ \Gamma \Vdash A' \leq A{:}Type \quad \Gamma, x{:}A' \Vdash B \leq B'{:}Type\end{array}}{\Gamma \Vdash \Pi(A,B) \leq \Pi(A',B'){:}Type}$$

Figure 7 Inference Rules for $\Pi_{\leq}$.

that sends the contexts with subtyping entries in the subsumptive system to signatures in the coercive system, prove that these signatures are coherent, and, finally, that we can embed the subsumptive subtyping system into the coercive subtyping system via this mapping. We are motivated, on the one hand by giving a coercive subtyping system in which we can represent this subsumptive system and at the same time allowing abstraction happen freely and on the other hand by the fact that we could not employ coercive subtyping in context as we could make coherent contexts incoherent with substitution. For example if $\alpha_1 \leq_{c_1} A, \alpha_2 \leq_{c_2} A, \Gamma$ is a coherent context (i.e. under this context any two coercions between the same types are equal), by substitution we can obtain the incoherent context $\alpha \leq_{c_1} A, \alpha \leq_{c_2} A, [\alpha_1/\alpha][\alpha_2/\alpha]\Gamma$.

We will assume that Δ is an arbitrary context in $\Pi_{\leq}$. We can also assume without loss of generality that $\Delta \equiv \Delta_1, \alpha_1 \leq A_1, ..., \Delta_n, \alpha_n \leq A_n, \Delta_{n+1}$, where $\{\alpha_i \leq A_i\}_{i=\overline{1,n}}$ are all of the subtyping entries of Δ. If Δ_{n+1} is free of subtyping entries we can abstract over its entries freely but the abstraction is obstructed by $\alpha_n \leq A_n$ for the entire prefix. We move this prefix, together with the obstructing entry to the signature using constant coercions $\Sigma_\Delta = \Delta_1, \alpha_1{:}Type, c_1{:}(\alpha_1)A_1, \alpha_1 \leq_{c_1} A_1{:}Type, ..., \Delta_n, \alpha_n{:}Type, c_n{:}(\alpha_n)A_n, \alpha_n \leq_{c_n} A_n{:}Type$. We map the left Δ_{n+1} to a context. This way we translate $\Delta \equiv \Delta_1, \alpha_1 \leq A_1, ..., \Delta_n, \alpha_n \leq A_n, \Delta_{n+1} \vdash J$ in $\Pi_{\leq}$ to $\Delta_{n+1} \vdash_{\Sigma_\Delta} J$ in $\Pi_{S,\leq}$, with Σ_Δ as above. In the rest of the section we shall prove that mapping subsumptive subtyping entries in context to constant coercions in signature is indeed adequate. For this, we first prove that such a signature is coherent.

▶ **Lemma 33.** *For any valid context Δ in $\Pi_{\leq}$, Σ_Δ is coherent w.r.t. $\Pi_{S,\leq}$.*

Proof. We need to show that, in $\Pi_{S,\leq}$, if we have $\Gamma \vdash_{\Sigma_\Delta} T_1 \leq_c T_2$ and $\Gamma \vdash_{\Sigma_\Delta} T_1 \leq_{c'} T_2$, then $c = c'{:}(T_1)T_2$. There are two cases:

1. $T_1 \equiv \alpha$ is a constant. By the validity of Δ, we have that, if $\alpha_i \leq A_i$ and $\alpha_j \leq A_i$ are two different subtyping entries in Δ, then $\alpha_i \neq \alpha_j$, therefore, if $\alpha_i \leq_{c_i} A_i$ and $\alpha_j \leq_{c_j} A_i$ are two different coercions in Σ_Δ, then necessarily, $\alpha_i \neq \alpha_j$.

2. $T_1 \equiv \Pi(A,B)$ and $T_2 \equiv \Pi(A'',B'')$. In this case the non trivial situation is:

$$\frac{\Gamma \vdash_\Sigma \Pi(A,B) \leq_{c_1} C \quad \Gamma \vdash_\Sigma C \leq_{c_2} \Pi(A'',B'')}{\Gamma \vdash_\Sigma \Pi(A,B) \leq_{c_2 \circ c_1} \Pi(A'',B'')}$$

and C is equal to dependent product too. What we need to show is that applying dependent product rule followed by transitivity leads to the same coercion as applying transitivity first and then the dependent product rule. Namely that, for some A', B' s.t.

$$\frac{\Gamma \vdash_{\Sigma_\Delta} A'' \leq_{c_2} A' \leq_{c_1} A \quad \Gamma \vdash_{\Sigma_\Delta} B \leq_{d_1} B' \leq_{d_2} B''}{\Gamma \vdash_{\Sigma_\Delta} \Pi(A,B) \leq_{e_1} \Pi(A',B') \leq_{e_2} \Pi(A'',B'')}$$

where, for $F{:}A \longrightarrow B$ and $G{:}\Pi(A',B')$, $e_1(F) = \lambda[x'{:}A']d_1(app(F, c_1(x')))$ and $e_2(G) = \lambda[x''{:}A'']d_2(app(G, c_2(x'')))$ applying transitivity rule, first to A, A', A'' and to B, B', B'' and then to $\Pi(A,B)$, $\Pi(A',B')$, $\Pi(A'',B'')$ results in the same coercion, that is:

$$\begin{aligned} e_2 \circ e_1 &= e_2(e_1(F)) \\ &= \lambda[x''{:}A'']d_2(app(e_1(F), c_2(x''))) \\ &=_\beta \lambda[x''{:}A'']d_2(d_1(app(F, c_1(c_2(x''))))) \\ &= d_2 \circ d_1(app(F, c_1(c_2(x'')))) \end{aligned}$$ ◀

Notation. If $\Gamma \vdash_\Sigma k{:}K$ and $\Gamma \vdash_\Sigma K \leq_c K'$ are derivable in $\Pi_{S,\leq}$, we write $\Gamma \vdash_\Sigma k :: K'$.

In what follows we essentially prove that we can represent the previously introduced subsumptive subtyping system in our system with coercive subtyping in signatures, meaning that we can argue about the former system with the sematic richness of the latter.

▶ **Theorem 34** (Embedding Subsumptive Subtyping). *Let Δ and Γ be valid contexts in $\Pi_\leq$, such that Γ does not contain any subtyping entries. Then we have:*

1. *If Δ, Γ is valid in $\Pi_\leq$ then $\vdash_{\Sigma_\Delta} \Gamma$ valid in $\Pi_{S,\leq}$.*
2. *If $\Delta, \Gamma \Vdash K$ kind, then $\Gamma \vdash_{\Sigma_\Delta} K$ kind in $\Pi_{S,\leq}$.*
3. *If $\Delta, \Gamma \Vdash K = K'$, then $\Gamma \vdash_{\Sigma_\Delta} K = K'$ in $\Pi_{S,\leq}$.*
4. *If $\Delta, \Gamma \Vdash k{:}K$, then $\Gamma \vdash_{\Sigma_\Delta} k{::}K$ in $\Pi_{S,\leq}$.*
5. *If $\Delta, \Gamma \Vdash k = k'{:}K$, then $\Gamma \vdash_{\Sigma_\Delta} k = k'{::}K$ in $\Pi_{S,\leq}$.*
6. *If $\Delta, \Gamma \Vdash A \leq B{:}Type$ then $\Gamma \vdash_{\Sigma_\Delta} A \leq_c B{:}Type$ for some coercion $c{:}(A)B$ in $\Pi_{S,\leq}$.*
7. *If $\Delta, \Gamma \Vdash K \leq K'$, then $\Gamma \vdash_{\Sigma_\Delta} K \leq_c K'$ for some $c{:}(K)K'$ in $\Pi_{S,\leq}$.*

Proof. The proof proceeds by induction on derivations for all the points of the theorem and we only exhibit it for the sixth point here and in particular when the last rule in the derivation tree is the one for the dependent product. We have by IH that, for $\Gamma \vdash_{\Sigma_\Delta} \Pi(A,B){::}Type$ and $\Gamma \vdash_{\Sigma_\Delta} \Pi(A',B'){::}Type$ we have $\Gamma \vdash_{\Sigma_\Delta} A' \leq_c A{:}Type$ and $\Gamma, x{:}A' \vdash_{\Sigma_\Delta} B \leq_{c'} B'{:}Type$. Note that, if $K \leq_c Type$, then $K \equiv Type$, so $\Gamma \vdash_{\Sigma_\Delta} \Pi(A,B){::}Type$ is equivalent to $\Gamma \vdash_{\Sigma_\Delta} \Pi(A,B){:}Type$, and $\Gamma \vdash_{\Sigma_\Delta} \Pi(A',B'){::}Type$ with $\Gamma \vdash_{\Sigma_\Delta} \Pi(A',B'){:}Type$, hence we can directly apply the rule for dependent product in $\Pi_{S,\leq}$ to obtain $\Gamma \vdash_{\Sigma_\Delta} \Pi(A,B) \leq_d \Pi(A',B'){:}Type$ where, for $F{:}\Pi(A,B)$, $d(F) = \lambda[x{:}A']c'(app(F, c(x)))$. ◀

4 Intuitive Notions of Subtyping as Coercion

In this section, we consider two case studies of how intuitive notions of subtyping may be considered in the framework of coercive subtyping. The first is about type universes in type theory and the second is about how injectivity of coercions may play a crucial role in modelling intuitive notions of subtyping.

4.1 Subtyping between Type Universes

A universe is a type of types. One may consider a sequence of universes indexed by natural numbers $U_0 : U_1 : U_2 : ...$ and $U_0 \leq U_1 \leq U_2 \leq ...$

Martin Löf [23] introduced two styles of universes in type theory: the Tarski-style and the Russell-style. The Tarski-style universes are semantically more fundamental but the Russell-style universes are easier to use in practice. In fact, the Russell-style universes are a special case of subsumptive subtyping, which is incompatible with the idea of canonical objects. As observed by the second author in [18], the two styles of universes are not equivalent and the Russell-style universes can be emulated by Tarski-style universes with coercive subtyping and this allows one to reason about Russell universes with the semantic richness of Tarski universes, but without the overhead of their syntax.

Problem with Russell-style Universes. We extend the subsumptive subtyping system $\Pi_{\leq}$ with Russell-style universes by adding the following rules ($i \in \omega$):

$$\frac{\Gamma\ valid}{\Gamma \Vdash U_i : Type} \qquad \frac{\Gamma \Vdash A : U_i}{\Gamma \Vdash A : Type} \qquad \frac{\Gamma\ valid}{\Gamma \Vdash U_i : U_{i+1}} \qquad \frac{\Gamma\ valid}{\Gamma \Vdash U_i \leq U_{i+1}}$$

and the rules for the Π-types:

$$\frac{\Gamma \Vdash A : U_i \quad \Gamma \Vdash B : (A)U_i}{\Gamma \Vdash \Pi(A, B) : U_i}$$

Unfortunately, as mentioned in the introduction, this straightforward formulation of universes does not satisfy the properties of canonicity or subject reduction if one adopts the standard notation of terms with full type information. For instance, the term $\lambda X{:}U_1.Nat$, where $Nat : U_0$, would be represented as $\lambda(U_1, [\_{:}U_1]U_0, [\_{:}U_1]Nat)$, but this term, which is of type $U_0 \to U_0$ (by subsumption, since $U_1 \to U_0 \leq U_0 \to U_0$ by contravariance), is not definitionally equal to any canonical term which is of the form $\lambda(U_0, ...)$. As explained in the introduction, if one used terms with less type information (eg, pairs (a, b), as in HoTT [32], rather than $pair(A, B, a, b)$, there would be incompatible types of the same term and that would cause problems in type-checking.

Tarski-style Universes with Coercive Subtyping. The Tarski-style universes are introduced into $\Pi_{S,\leq}$ by adding the following rules ($i \in \omega$):

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma U_i : Type} \qquad \frac{\Gamma \vdash_\Sigma a : U_i}{\Gamma \vdash_\Sigma T_i(a) : Type} \qquad \frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma t_{i+1} : (U_i)U_{i+1}}$$

where t_{i+1} are the lifting operators,

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma u_i : U_{i+1}} \qquad \frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma T_{i+1}(u_i) = U_i : Type}$$

where u_i is the name of U_i in U_{i+1}, together with the following rule for the names of Π-types:

$$\frac{\Gamma \vdash_\Sigma a : U_i \quad \Gamma,\ x : T_i(a) \vdash_\Sigma b(x) : U_i}{\Gamma \vdash_\Sigma \pi_i(a, b) : U_i}$$

The following equations also need to be satisfied:
$T_{i+1}(t_{i+1}(a)) = T_i(a){:}Type$

$\Gamma \vdash_\Sigma T_i(\pi_i(a,b)) = \Pi(T_i(a), [x{:}T_i(a)]T_i(b(x))) : \mathit{Type}$
$\Gamma \vdash_\Sigma t_{i+1}(\pi_i(a,b)) = \pi_{i+1}(t_{i+1}(a), [x{:}T_i(a)]t_{i+1}(b(x))) : U_{i+1}$
Furthermore, crucially, the lifting operators t_{i+1} are now declared as coercions by asking that all the signatures start with the prefix $\Sigma_i \equiv U_0 \le_{t_0} U_1,\ \ \ldots,\ U_{i-1} \le_{t_i} U_i$ where i is bigger than the largest universe index that is used in an application.

Use of Coercion-based Tarski-style Universes. If universes are specified in the Tarski-style as above with the lifting operators declared as coercions, together with several notational conventions (eg, T_i is omitted, u_i is identified with U_i, etc.), they can now be used easily in Russell-style. The lifting operators are not seen (implicit) by the users. In particular, in this setting, all the Russell-style universe rules become derivable. Theorem 34 can now be extended in such a way that the Russell-style universes are faithfully emulated by the Tarski-style universes with coercive subtyping.

4.2 Injectivity and Constructor Subtyping

In subsumptive subtyping, $A \le B$ means that A is directly embedded in B. Intuitively, this may imply that, for a and a' in A, if the images of them are not equal in B, then they are not equal in A, either. If we consider coercive subtyping $A \le_c B$, this would imply that c is injective in the sense that $c(a) = c(a')$ implies that $a = a'$. In this section, we shall formally discuss this issue in the context of representing intuitive subtyping notions by means of coercions.

We shall consider constructor subtyping, studied by [4], in which an (inductive) type is considered to be a subtype of another if the latter has more constructors than the former. More precisely we shall discuss the example they start from, namely Even Numbers(Even) being a subtype of Natural Numbers (Nat) with the argument that the constructors of Even are 0 and successor of Odd, where Odd is given by the constructor successor of Even. Then, in Nat the successor constructor is overloaded to a lifting of these constructors as well. Formally they write:

```
datatype Odd = S of Even and Even = 0
     |S of Odd
datatype Nat  = 0
     |S of Nat
     |S of Odd
     |S of Even
```

The phenomenon we want to discuss here is injectivity, in particular the one related to Leibnitz equality. Leibnitz equality is defined as follows: $x = y$ if for any predicate P, $P(x) \iff P(y)$. We denote by $x =_A y$ for some type A the Leibnitz equality between x and y related to a certain domain. Then, we have injectivity of subtyping if, given $x =_{Nat} y$, with $x, y{:}\mathit{Even}$ it is the case that $x =_{Even} y$. Namely, whether for any predicate $Q{:}\mathit{Even} \longrightarrow \mathit{Prop}$, it is the case that $Q(x) \iff Q(y)$. For this it is enough to show that any predicate $Q{:}\mathit{Even} \longrightarrow \mathit{Prop}$ admits a lifting $Q'{:}\mathit{Nat} \longrightarrow \mathit{Prop}$ s.t. for any $x{:}\mathit{Even}, Q'(x) \Longrightarrow Q(x)$. We can easily define such a Q' as follows:

```
Q'(x) =  Q(0) if x = 0
         Q(S(n)) if x = S of n:Odd
         true if x = S of n:Even
         true if x = S of n:Nat
```

Injectivity of the embedding holds here but it is not granted in coercive subtyping. For functions $f{:}(x{:}A)B$ we denote $injective(f) = \forall x, y{:}A.f(x) =_B f(y) \longrightarrow x =_A y$. A function f is then injective if $\exists p{:}injective(f)$.

▶ **Definition 35.** We say a coercion $\vdash_{\Sigma_0,A\leq_c B,\Sigma_1} A \leq_c B$ is **injective with respect to $=_B$** if there exist p s.t. $\vdash_\Sigma p{:}injective(c)$ is derivable.

For a constant coercions (namely of the form $\vdash_{\Sigma_0,c:(A)B,\Sigma_1,A\leq_c B,\Sigma_2,\Sigma_3} A \leq_c B$) we can add the assumption that they are injective $\vdash_{\Sigma_0,c:(A)B,\Sigma_1,A\leq_c B,\Sigma_2,p:injective(c),\Sigma_3} A \leq_c B$. If we embed a subsumptive subtyping that propagates an equality from a type throughout its subtypes, we represent it as a constant coercion, thus, all we need to do is add the assumption that a coercion is injective. It is obvious that the transitivity and congruence preserve the injectivity property.

An example of noninjective coercions is if we think of Nat and $Even$ as follows

```
Inductive Nat : Type :=
    | O : Nat
    | S : Nat -> Nat.
Inductive even : Nat -> Prop :=
    | O1 : even O
    | O2 : even O
    | S1 : forall n1, even n1 -> even (S (S n1)).
Inductive Even := pair{n:Nat; e:even n}. Definition proj1(ev:Even) :=
    match
        ev with pair n e => n
        end.
Coercion proj1 : Even >-> Nat.
```

Note that the definition of $Even$ changed and we refer to it as a feature of the natural numbers rather than as a subset. In order for a natural number to be even we require a proof of that.

The reason this coercion is not injective is that we can have two different proofs that 4 is even $p_1, p_2{:}even4$, and hence, two different pairs $(4, p_1), (4, p_2){:}Even$, both of them being mapped to the same $4{:}Nat$. Enforcing injectivity here is similar to enforcing proof irrelevance.

5 Conclusion and Future Work

In this paper, we have developed a new calculus of coercive subtyping and shown that subsumptive subtyping can be faithfully embedded or represented in the calculus. The idea of representing coercive subtyping relations in signatures has achieved a balance between obtaining a powerful (and practical) calculus to capture intuitive notions of subtyping and keeping the resulting calculus simple enough for meta-theoretic studies.

We intend to extend the calculus to a richer type theory like Martin-Löf's type theory or UTT where you have rich inductive types. We do not see any difficulty in doing so, but of course, studies are needed to confirm this.

Specifying subtyping relations in signatures has changed the nature of 'basic subtyping relations' as studied in the earlier setting of coercive subtyping. The earlier setting allows parameterised coercions such as $n{:}Nat \vdash Vect(Nat, n) \leq_{c(n)} List(Nat)$, which instantiates, in particular, to $\vdash Vect(Nat, 3) \leq_{c(3)} List(Nat)$. Note that here we don't use *parameterised* in the sense of Coq Proof Assistant. This new system does not cover this kind of coercions at this point. It would be interesting to study a new mechanism to introduce parameterised coercions by means of entries in signatures.

References

1 The Agda proof assistant (version 2), 2008. URL: `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

2 D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266:273–309, 2001.

3 A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1999.

4 G. Barthe and M. J. Frade. Constructor subtyping. *Lecture Notes in Computer Science*, 1576:109–127, 1999.

5 Gustavo Betarte and Alvaro Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. *Twenty-five Years of Constructive Type Theory*, 1998.

6 V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. *Information and Computation*, 93, 1991.

7 P. Callaghan and Z. Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

8 S. Chatzikyriakidis and Z. Luo. Natural language inference in Coq. *J. of Logic, Language and Information.*, 23(4):441–480, 2014.

9 S. Chatzikyriakidis and Z. Luo. *Formal Semantics in Modern Type Theories*. ISTE/Wiley, 2018. (to appear).

10 The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.3), INRIA*, 2010.

11 Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

12 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:143–184, 1993.

13 Y. Luo. *Coherence and Transitivity in Coercive Subtyping*. PhD thesis, University of Durham, 2005.

14 Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

15 Z. Luo. Coercive subtyping in type theory. In *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258*, page draft., 1996.

16 Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9, 1999.

17 Z. Luo. Formal semantics in modern type theories with coercive subtyping. *Linguistics and Philosophy*, 35(6):491–513, 2012.

18 Z. Luo. Notes on Universes in Type Theory (for a talk given at Institute of Advanced Studies), 2012. URL: `https://uf-ias-2012.wikispaces.com/file/view/LuoUniverse.pdf`.

19 Z. Luo. Formal semantics in modern type theories: Is it model-theoretic, proof-theoretic, or both? (invited talk). In Nicholas Asher and Sergei Soloviev, editors, *Logical Aspects of Computational Linguistics*, volume 8535 of *Lecture Notes in Computer Science*, pages 177–188. Springer Berlin Heidelberg, 2014.

20 Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Dept of Computer Science, Univ of Edinburgh, 1992.

21 Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: theory and implementation. information and computation. *Information and Computation*, 223:18–42, 2013.

22 Zhaohui Luo and Fjodor Part. Subtyping in type theory: Coercion contexts and local coercions. In *TYPES 2013, Toulouse*, 2013.

23 P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

24 The Matita proof assistant. Available from: `http://matita.cs.unibo.it/`, 2008.

25 J. C. Mitchell. Coercion and type inference. In *POPL'83*, 1983.

26 Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Clarendon Press, New York, NY, USA, 1990.

27 Benjamin C. Pierce. Bounded quantification is undecidable. In *Information and Computation*, pages 305–315, 1993.

28 J. Reynolds. The meaning of types: From intrinsic to extrinsic semantics. *BRICS Report Series RS-00-32*, 2000.

29 John C. Reynolds. Using category theory to design implicit conversions and generic operators. *Semantics-Directed Compiler Generation 1980*, Lecture Notes in Computer Science 94, 1980.

30 A. Saïbi. Typing algorithm in type theory with inheritance. *POPL'97*, 1997.

31 S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113(1-3):297–322, 2002.

32 Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study, 2013.

33 Tao Xue. *Theory and Implementation of Coercive Subtyping.* PhD thesis, Royal Holloway University of London, 2013.

A Rules of $\Pi[C]^{;}$

The rules of $\Pi[C]^{;}$ consists of those in Figures 8, Figure 9, Figure 10, Figure 11 and Figure 12.

Validity of Signature/Contexts, Assumptions

$$\frac{}{\vdash \langle\rangle} \qquad \frac{\Sigma; <>\vdash K\ kind \quad c \notin dom(\Sigma)}{\vdash \Sigma, c{:}K} \qquad \frac{\vdash \Sigma, c{:}K, \Sigma'; \Gamma}{\Sigma, c{:}K, \Sigma'; \Gamma \vdash c{:}K}$$

$$\frac{\vdash \Sigma}{\vdash \Sigma; \langle\rangle} \qquad \frac{\Sigma; \Gamma \vdash K\ kind \quad x \notin dom(\Sigma) \cup dom(\Gamma)}{\vdash \Sigma; \Gamma, x{:}K} \qquad \frac{\vdash \Sigma; \Gamma, x{:}K, \Gamma'}{\Sigma; \Gamma, x{:}K, \Gamma' \vdash x{:}K}$$

Weakening

$$\frac{\Sigma, \Sigma'; \Gamma \vdash J \quad \Sigma; <>\vdash K\ kind \quad c \notin dom(\Sigma, \Sigma')}{\Sigma,\ c{:}K,\ \Sigma'; \Gamma \vdash J}$$

$$\frac{\Sigma; \Gamma, \Gamma' \vdash J \quad \Sigma; \Gamma \vdash K\ kind \quad x \notin dom(\Gamma, \Gamma')}{\Sigma; \Gamma, x{:}K, \Gamma' \vdash J}$$

Equality Rules

$$\frac{\Sigma; \Gamma \vdash K\ \ kind}{\Sigma; \Gamma \vdash K = K} \qquad \frac{\Sigma; \Gamma \vdash K = K'}{\Sigma; \Gamma \vdash K' = K} \qquad \frac{\Sigma; \Gamma \vdash K = K' \quad \Sigma; \Gamma \vdash K' = K''}{\Sigma; \Gamma \vdash K = K''}$$

$$\frac{\Sigma; \Gamma \vdash k{:}K}{\Sigma; \Gamma \vdash k = k{:}K} \qquad \frac{\Sigma; \Gamma \vdash k = k'{:}K}{\Sigma; \Gamma \vdash k' = k{:}K} \qquad \frac{\Sigma; \Gamma \vdash k = k'{:}K \quad \Sigma; \Gamma \vdash k' = k''{:}K}{\Sigma; \Gamma \vdash k = k''{:}K}$$

$$\frac{\Sigma; \Gamma \vdash k{:}K \quad \Sigma; \Gamma \vdash K = K'}{\Sigma; \Gamma \vdash k{:}K'} \qquad \frac{\Sigma; \Gamma \vdash k = k'{:}K \quad \Sigma; \Gamma \vdash K = K'}{\Sigma; \Gamma \vdash k = k'{:}K'}$$

Context Replacement

$$\frac{\Sigma_0, c{:}L, \Sigma_1; \Gamma \vdash J \quad \Sigma_0 \vdash L = L'}{\Sigma_0, c{:}L', \Sigma_1; \Gamma \vdash J} \qquad \frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash J \quad \Sigma; \Gamma_0 \vdash K = K'}{\Sigma; \Gamma_0, x{:}K', \Gamma_1 \vdash J}$$

Substitution Rules

$$\frac{\vdash \Sigma; \Gamma_0, x{:}K, \Gamma_1 \quad \Sigma; \Gamma_0 \vdash k{:}K}{\vdash \Sigma; \Gamma_0, [k/x]\Gamma_1}$$

$$\frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash K'\ \ kind \quad \Sigma; \Gamma_0 \vdash k{:}K}{\Sigma; \Gamma_0, [k/x]\Gamma_1 \vdash [k/x]K'\ \ kind} \qquad \frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash L = L' \quad \Sigma; \Gamma_0 \vdash k{:}K}{\Sigma; \Gamma_0, [k/x]\Gamma_1 \vdash [k/x]L = [k/x]L'}$$

$$\frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash k'{:}K' \quad \Sigma; \Gamma_0 \vdash k{:}K}{\Sigma; \Gamma_0, [k/x]\Gamma_1 \vdash [k/x]k'{:}[k/x]K'} \qquad \frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash l = l'{:}K' \quad \Sigma; \Gamma_0 \vdash k{:}K}{\Sigma; \Gamma_0, [k/x]\Gamma_1 \vdash [k/x]l = [k/x]l'{:}[k/x]K'}$$

$$\frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash K'\ \ kind \quad \Sigma; \Gamma_0 \vdash k = k'{:}K}{\Sigma; \Gamma_0, [k/x]\Gamma_1 \vdash [k/x]K' = [k'/x]K'} \qquad \frac{\Sigma; \Gamma_0, x{:}K, \Gamma_1 \vdash l{:}K' \quad \Sigma; \Gamma_0 \vdash k = k'{:}K}{\Sigma; \Gamma_0, [k/x]\Gamma_1 \vdash [k/x]l = [k'/x]l{:}[k/x]K'}$$

Dependent Product Kinds

$$\frac{\Sigma; \Gamma \vdash K\ kind \quad \Sigma; \Gamma, x{:}K \vdash K'\ kind}{\Sigma; \Gamma \vdash (x{:}K)K'\ kind} \qquad \frac{\Sigma; \Gamma \vdash K_1 = K_2 \quad \Sigma; \Gamma, x{:}K_1 \vdash K_1' = K_2'}{\Sigma; \Gamma \vdash (x{:}K_1)K_1' = (x{:}K_2)K_2'}$$

$$\frac{\Sigma; \Gamma, x{:}K \vdash y{:}K'}{\Sigma; \Gamma \vdash [x{:}K]y{:}(x{:}K)K'} \qquad \frac{\Sigma; \Gamma \vdash K_1 = K_2 \quad \Sigma; \Gamma, x{:}K_1 \vdash k_1 = k_2{:}K}{\Sigma; \Gamma \vdash [x{:}K_1]k_1 = [x{:}K_2]k_2{:}(x{:}K_1)K}$$

$$\frac{\Sigma; \Gamma \vdash f{:}(x{:}K)K' \quad \Sigma; \Gamma \vdash k{:}K}{\Sigma; \Gamma \vdash f(k){:}[k/x]K'} \qquad \frac{\Sigma; \Gamma \vdash f = f'{:}(x{:}K)K' \quad \Sigma; \Gamma \vdash k_1 = k_2{:}K}{\Sigma; \Gamma \vdash f(k_1) = f'(k_2){:}[k_1/x]K'}$$

$$\frac{\Sigma; \Gamma, x{:}K \vdash k'{:}K' \quad \Sigma; \Gamma \vdash k{:}K}{\Sigma; \Gamma \vdash ([x{:}K]k')(k) = [k/x]k'{:}[k/x]K'} \qquad \frac{\Sigma; \Gamma \vdash f{:}(x{:}K)K' \quad x \notin FV(f)}{\Sigma; \Gamma \vdash [x{:}K]f(x) = f{:}(x{:}K)K'}$$

The kind Type

$$\frac{\vdash \Sigma; \Gamma}{\Sigma; \Gamma \vdash Type\ \ kind} \qquad \frac{\Sigma; \Gamma \vdash A{:}Type}{\Sigma; \Gamma \vdash El(A)\ \ kind} \qquad \frac{\Sigma; \Gamma \vdash A = B{:}Type}{\Sigma; \Gamma \vdash El(A) = El(B)}$$

Figure 8 Inference Rules for $LF^{;}$.

$$\frac{\Sigma;\Gamma\vdash A:\mathit{Type}\quad \Sigma;\Gamma,x{:}A\vdash B(x):\mathit{Type}}{\Sigma;\Gamma\vdash \Pi(A,B):\mathit{Type}}$$

$$\frac{\Sigma;\Gamma\vdash A:\mathit{Type}\quad \Sigma;\Gamma\vdash B:(A)\mathit{Type}\quad \Sigma;\Gamma\vdash f:(x{:}A)B(x)}{\Sigma;\Gamma\vdash \lambda(A,B,f):\Pi(A,B)}$$

$$\frac{\Sigma;\Gamma\vdash g:\Pi(A,B)\quad \Sigma;\Gamma\vdash a:A}{\Sigma;\Gamma\vdash \mathit{app}(A,B,g,a):B(a)}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma\vdash A:\mathit{Type}\quad \Sigma;\Gamma\vdash B:(A)\mathit{Type}\\ \Sigma;\Gamma\vdash f:(x{:}A)B(x)\quad \Sigma;\Gamma\vdash a:A\end{array}}{\Sigma;\Gamma\vdash \mathit{app}(A,B,\lambda(A,B,f),a)=f(a):B(a)}$$

Figure 9 Inference Rules for $\Pi^{;}$.

Subtyping Rules

$$\frac{\Sigma;\Gamma\vdash A\le_c B\in\mathcal{C}}{\Sigma;\Gamma\vdash A\le_c B}$$

Congruence

$$\frac{\Sigma;\Gamma\vdash A\le_c B:\mathit{Type}\quad \Sigma;\Gamma\vdash A=A':\mathit{Type}\quad \Sigma;\Gamma\vdash B=B':\mathit{Type}\quad \Sigma;\Gamma\vdash c=c':(A)B}{\Sigma;\Gamma\vdash A'\le_{c'} B':\mathit{Type}}$$

Transitivity

$$\frac{\Sigma;\Gamma\vdash A\le_c A':\mathit{Type}\quad \Sigma;\Gamma\vdash A'\le_{c'} A'':\mathit{Type}}{\Sigma;\Gamma\vdash A\le_{c'\circ c} A'':\mathit{Type}}$$

Weakening

$$\frac{\Sigma,\Sigma';\Gamma\vdash A\le_d B:\mathit{Type}\quad \Sigma\vdash K\ \mathit{kind}}{\Sigma,\ c{:}K,\ \Sigma';\Gamma\vdash A\le_d B:\mathit{Type}}\quad (c\notin \mathit{dom}(\Sigma,\Sigma'))$$

$$\frac{\Sigma;\Gamma,\Gamma'\vdash A\le_d B:\mathit{Type}\quad \Sigma;\Gamma\vdash K\ \mathit{kind}}{\Sigma;\Gamma,x{:}K,\Gamma'\vdash A\le_d B:\mathit{Type}}\quad (x\notin \mathit{dom}(\Gamma,\Gamma'))$$

Context Replacement

$$\frac{\Sigma_0,c{:}L,\Sigma_1;\Gamma\vdash A\le_c B\quad \Sigma_0\vdash L=L'}{\Sigma_0,c{:}L',\Sigma_1;\Gamma\vdash A\le_c B}\qquad \frac{\Sigma;\Gamma_0,x{:}K,\Gamma_1\vdash A\le_c B\quad \Sigma;\Gamma_0\vdash K=K'}{\Sigma;\Gamma_0,x{:}K',\Gamma_1\vdash A\le_c B}$$

Substitution

$$\frac{\Sigma;\Gamma_0,x{:}K,\Gamma_1\vdash A\le_c B\quad \Sigma;\Gamma_0\vdash k{:}K}{\Sigma;\Gamma_0,[k/x]\Gamma_1\vdash [k/x]A\le_{[k/x]c}[k/x]B}$$

Identity Coercion

$$\frac{\Sigma;\Gamma\vdash A{:}\mathit{Type}}{\Sigma;\Gamma\vdash A\le_{[x:A]x} A{:}\mathit{Type}}$$

Dependent Product

$$\frac{\Sigma;\Gamma\vdash A'\le_{c_1} A:\mathit{Type}\quad \Sigma;\Gamma\vdash B,B':(A)\mathit{Type}\quad \Sigma;\Gamma,x{:}A\vdash B(x)\le_{c_2[x]} B'(x):\mathit{Type}}{\Sigma;\Gamma\vdash \Pi(A,B)\le_{[F:\Pi(A,B)]\lambda(A',B'\circ c_1,[x:A']c_2[x](\mathit{app}(A,B,F,c_1(x))))}\Pi(A',B'\circ c_1):\mathit{Type}}$$

Figure 10 Inference Rules for $\Pi[\mathcal{C}]^{;}_{0K}$ (1).

Basic Subkinding Rule and Identity

$$\frac{\Sigma;\Gamma\vdash A\leq_c B{:}Type}{\Sigma;\Gamma\vdash El(A)\leq_c El(B)} \qquad \frac{\Sigma;\Gamma\vdash K\ \ kind}{\Sigma;\Gamma\vdash K\leq_{[x:K]x} K}$$

Structural Subkinding Rules

$$\frac{\Sigma;\Gamma\vdash K_1\leq_c K_2 \quad \Sigma;\Gamma\vdash K_1=K_1' \quad \Sigma;\Gamma\vdash K_2=K_2' \quad \Sigma;\Gamma\vdash c=c'{:}(K_1)K_2}{\Sigma;\Gamma\vdash K_1'\leq_{c'} K_2'}$$

$$\frac{\Sigma;\Gamma\vdash K\leq_c K' \quad \Sigma;\Gamma\vdash K'\leq_{c'} K''}{\Sigma;\Gamma\vdash K\leq_{c'\circ c} K''}$$

$$\frac{\Sigma,\Sigma';\Gamma\vdash K\leq_d K' \quad \Sigma;<>\vdash K_0\ kind}{\Sigma,c{:}K_0,\Sigma';\Gamma\vdash K\leq_d K'} \quad (c\notin dom(\Sigma,\Sigma'))$$

$$\frac{\Sigma;\Gamma,\Gamma'\vdash K\leq_d K' \quad \Sigma;\Gamma\vdash K_0\ kind}{\Sigma;\Gamma,x{:}K_0,\Gamma'\vdash K\leq_d K'} \quad (x\notin dom(\Gamma,\Gamma'))$$

$$\frac{\Sigma_0,c{:}L,\Sigma_1;\Gamma\vdash K\leq_d K' \quad \Sigma_0;<>\vdash L=L'}{\Sigma_0,c{:}L',\Sigma_1;\Gamma\vdash K\leq_d K'} \qquad \frac{\Sigma;\Gamma_0,x{:}K,\Gamma_1\vdash L\leq_d L' \quad \Sigma;\Gamma_0\vdash K=K'}{\Sigma;\Gamma_0,x{:}K',\Gamma_1\vdash L\leq_d L'}$$

$$\frac{\Sigma;\Gamma_0,x{:}K,\Gamma_1\vdash K_1\leq_c K_2 \quad \Sigma;\Gamma_0\vdash k{:}K}{\Sigma;\Gamma_0,[k/x]\Gamma_1\vdash [k/x]K_1\leq_{[k/x]c}[k/x]K_2}$$

Subkinding for Dependent Product Kind

$$\frac{\Sigma;\Gamma\vdash K_1'\leq_{c_1} K_1\ \Sigma;\Gamma,x{:}K_1\vdash K_2\ kind\ \Sigma;\Gamma,x'{:}K_1'\vdash K_2'\ kind\ \Sigma;\Gamma,x{:}K_1\vdash [c_1(x')/x]K_2\leq_{c_2} K_2'}{\Sigma;\Gamma\vdash (x{:}K_1)K_2\leq_{[f:(x:K_1)K_2][x':K_1']c_2(f(c_1(x')))}(x{:}K_1')K_2'}$$

Figure 11 Inference Rules for $\Pi[\mathcal{C}]^{;}_{0K}$ (2).

Coercive Application

$$(CA_1) \qquad \frac{\Sigma;\Gamma\vdash f{:}(x{:}K)K' \quad \Sigma;\Gamma\vdash k_0{:}K_0 \quad \Sigma;\Gamma\vdash K_0\leq_c K}{\Sigma;\Gamma\vdash f(k_0){:}[c(k_0)/x]K'}$$

$$(CA_2) \qquad \frac{\Sigma;\Gamma\vdash f=f'{:}(x{:}K)K' \quad \Sigma;\Gamma\vdash k_0=k_0'{:}K_0 \quad \Sigma;\Gamma\vdash K_0\leq_c K}{\Sigma;\Gamma\vdash f(k_0)=f'(k_0'){:}[c(k_0)/x]K'}$$

Coercive Definition

$$(CD) \qquad \frac{\Sigma;\Gamma\vdash f{:}(x{:}K)K' \quad \Sigma;\Gamma\vdash k_0{:}K_0 \quad \Sigma;\Gamma\vdash K_0\leq_c K}{\Sigma;\Gamma\vdash f(k_0)=f(c(k_0)){:}[c(k_0)/x]K'}$$

Figure 12 The coercive application and definition rules in $\Pi[\mathcal{C}]^{;}$.

A Formal Study of Boolean Games with Random Formulas as Payoff Functions

Érik Martin-Dorel
Lab. IRIT, Univ. of Toulouse, CNRS, IRIT Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse Cedex 9, France
erik.martin-dorel@irit.fr
https://orcid.org/0000-0001-9716-9491

Sergei Soloviev[1]
Lab. IRIT, Univ. of Toulouse, CNRS, IRIT Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse Cedex 9, France
sergei.soloviev@irit.fr

Abstract

In this paper, we present a probabilistic analysis of Boolean games. We consider the class of Boolean games where payoff functions are given by random Boolean formulas. This permits to study certain properties of this class in its totality, such as the probability of existence of a winning strategy, including its asymptotic behaviour. With the help of the Coq proof assistant, we develop a Coq library of Boolean games, to provide a formal proof of our results, and a basis for further developments.

2012 ACM Subject Classification Theory of computation → Higher order logic, Theory of computation → Algorithmic game theory, Mathematics of computing → Stochastic processes

Keywords and phrases Boolean games, Random process, Coq formal proofs

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.14

Supplement Material `https://doi.org/10.5281/zenodo.1317609`

Funding This work was partly supported by the FAGames project of LabEx CIMI.

Acknowledgements The authors would like to thank Evgeny Dantsin and Jan-Georg Smaus for fruitful discussions on algorithmic games and other topics, as well as the anonymous referees for their remarks and feedback on our article. The first author also wishes to thank Cyril Cohen and Enrico Tassi for advice and enhancement suggestions for our `under` tactic.

1 Introduction

One of the main motivations to consider the classes of games with random parameters is that it is a good method to explore these classes in their totality and to understand the relative importance of various properties of games (such as simultaneous or alternating moves, different assumptions concerning the access to information, etc.)

The situation when, for a given game, only its type can be known in advance but its parameters cannot, is common when the game-theoretic approach is used to study the behaviour of embedded systems, i.e., when at least some of the players are programs and the

[1] S. Soloviev was partly supported by the Government of Russian Federation Grant 08-08 at ITMO University, Saint-Petersburg, Russia (associated researcher).

parameters are not fully controlled. The augmented frequency of interaction that usually surpasses any conceivable capacity of human players, and rapid evolution of parameters of interaction makes the use of probabilistic methods quite natural.

In this paper we present the first results obtained via this approach applied to Boolean Games [11, 10, 7, 3]. More precisely, we limit our study to Boolean Games with random formulas that represent payoff functions. This model is naturally related to the situation when games between automated systems (e.g., embedded systems in computer networks) are considered. Indeed, assuming that the players are finite non-deterministic machines, they can be simulated by a family of Boolean formulas.

Before the exploration may start, several choices have to be made concerning the probabilistic model.

Regarding elementary events: we have chosen Boolean functions as elementary events. Another possible choice would be to consider formulas as syntactic objects, but the former choice makes it easier to define probability distributions that are naturally related to the properties on the associated Boolean games (while the latter choice would require to cope with the complex behaviour of the logical equivalence of formulas).

It seems natural also to consider a probability space for each n, where n is the number of variables. Indeed, n is one of the key parameters involved when considering the complexity of Boolean functions, and if we shall need to consider different values of n, it is possible to combine in some way the spaces built for each n.

Let us recall some basic properties of Boolean functions.

(i) The domain of these functions is $\mathbf{2}^n = \{\mathsf{false}, \mathsf{true}\}^n$, the set of all Boolean vectors of length n (which contains 2^n elements).

(ii) Each Boolean function can be identified with a characteristic function of a subset of $\mathbf{2}^n$ and thus with the subset itself, so the set of all elementary events is $\Omega = \mathbf{2}^{\mathbf{2}^n}$.

(iii) A subset of $\mathbf{2}^n$ may be identified with a formula of n variables in the full disjunctive normal form[2] that is satisfied by exactly these vectors (to each vector corresponds the conjunction of variables and their negations: to true at the i-th place corresponds v_i and to false corresponds $\neg v_i$).

(iv) The set Ω is a complete Boolean algebra, and logical operators on elements of Ω correspond to the set-theoretic operators on $\mathbf{2}^n$. The top element of this algebra is the set $\mathbf{2}^n \in \Omega$ (it represents the Boolean function true) and the bottom element of this algebra is the set $\emptyset \in \Omega$ (it represents the Boolean function false). How these logical operators "interact" with probability on Ω is a separate question, for example, $\mathbb{P}(\mathsf{true})$ needs not of course be equal to 1.

(v) There is a natural partial order on the elements of Ω, that is defined by the inclusion of subsets of $\mathbf{2}^n$ and at the same time by Boolean implication (see Definition 1 below).

Since the elements of Ω may be seen at the same time as Boolean functions and as sets of Boolean vectors, we pose the following definition:

► **Definition 1.** Let ω_1, ω_2 be two Boolean functions ($\omega_1, \omega_2 \in \Omega := \mathbf{2}^{\mathbf{2}^n}$). We shall write $\omega_1 \Rightarrow_0 \omega_2$ and say that "ω_2 is true on ω_1" if for each vector $v \in \mathbf{2}^n$, $\omega_1(v) = \mathsf{true}$ implies $\omega_2(v) = \mathsf{true}$. Also, this is equivalent to the inclusion of ω_1 in ω_2, seen as subsets of $\mathbf{2}^n$:

$$\forall \omega_1, \omega_2 \in \mathbf{2}^{\mathbf{2}^n}.\ (\omega_1 \Rightarrow_0 \omega_2) \iff (\omega_1 \subseteq \omega_2)\,.$$

[2] In particular, the empty subset may be identified with the empty DNF, that is the constant false (the neutral element of the disjunction).

▶ Remark. We may see random Boolean functions as (not necessarily independent) vectors of 2^n random variables with values in $\mathbf{2} = \{\mathsf{false}, \mathsf{true}\}$.

Regarding the sigma-algebra of events: as usual for finite probability spaces, we consider the sigma-algebra $\mathcal{S}$ of all subsets of Ω:

$$\mathcal{S} = \mathbf{2}^{2^{2^n}}.$$

Regarding the probability distributions on the spaces of Boolean formulas: as it is noticed in [8], it is often assumed that all Boolean functions on a given number of variables have the same probability (see also [16]). In this paper where we start our study, we decided to consider a slightly more general class of probability distributions, where Boolean functions are generated by a Bernoulli scheme on Boolean vectors, with any probability p as parameter. Some more sophisticated ways to define probability distributions on Boolean expressions are discussed in [8] and we plan to explore them in a near future.

In Section 2, we prove several general results on the probability of winning strategies assuming an arbitrary probability distribution. Then in Section 3, we study the case of Boolean functions constructed through a finite Bernoulli process, specialise our results in this simpler setting, then discuss the relevance of these results. In Section 4, we further study the probability that a winning strategy exists for player A, with the new assumption that player A knows s bits of the opponent player. Then in Section 5, we study the growth rate of the aforementioned probability, with respect to the knowledge of the second player's choices. Section 6 is devoted to technical remarks about the formalisation of our results within the Coq [5] formal proof assistant. Finally, a discussion on the notion of "non-guaranteed win" and its relationship with the order of moves is presented in Appendix A.

All the results of the paper have been formally verified within Coq.[3]

The Coq code is available online at `https://github.com/erikmd/coq-bool-games` and it also has been archived, see [14].

Beyond the fact that formal certification is interesting in itself to the TYPES community, it is nowadays common in the development and characterisation of the behaviour of autonomous programs, which is one of the subjects of this study.

2 Probability of Winning Strategies

Building upon the material of the previous section, we can consider any probability $\mathbb{P}$ defined on the sigma-algebra $\mathcal{S} = \mathbf{2}^{2^{2^n}}$, and thus obtain a probability space $(\Omega, \mathcal{S}, \mathbb{P})$. We shall show in this section that several results can be derived in this setting, however general as it may sound.

▶ **Example 2** (using Definition 1)**.** The probability of the event "ω is true on ω_0", with fixed $\omega_0 \in \Omega$, is

$$\mathbb{P}(\omega_0 \Rightarrow_0 \omega) = \sum_{\substack{\omega \\ \omega_0 \Rightarrow_0 \omega}} \mathbb{P}(\{\omega\}).$$

Below we shall consider the Boolean Games of two players A and B with a random Boolean function F of n variables as the payoff function of A, and its negation as the payoff

[3] In the sequel of the paper, all definitions and theorems will be stated in mathematical syntax, and the corresponding Coq identifier will be given between brackets.

function for B. We shall assume that A controls the first k variables, and B the remaining $n-k$ variables.

The strategy of A is any vector that belongs to $\mathbf{2}^k$ (valuation of the first k variables) and the strategy of B any valuation of the remaining $n-k$ variables (a vector of $\mathbf{2}^{n-k}$).

The outcome of the game is given by player A's payoff function $F : \mathbf{2}^n \to \mathbf{2}$, which can thereby be viewed as a function $F : \mathbf{2}^k \times \mathbf{2}^{n-k} \to \mathbf{2}$ (mapping a profile strategy to a Boolean outcome). In the sequel of the paper, we shall identify these two possible types for the function F – while in the formal development they will be encoded respectively as `(bool_fun` n`)` and `(bool_game` n k`)`.

▶ **Definition 3** (`winA`)**.** For any game $F : \mathbf{2}^k \times \mathbf{2}^{n-k} \to \mathbf{2}$, a strategy $a = (a_1, \dots, a_k)$ of player A is winning if it wins against any strategy $b \in \mathbf{2}^{n-k}$ of B:

$$\mathrm{win}_A[F](a) := \forall b \in \mathbf{2}^{n-k}.\ F(a,b) = \mathsf{true}.$$

If there is no ambiguity, we shall omit the name of the game and simply write $\mathrm{win}_A(a)$.

In other words, a is winning if the payoff function is equal to `true` on all vectors of length n that "extend" a. This led us to introduce the following

▶ **Definition 4** (`w_`, `W_`)**.** For any $a \in \mathbf{2}^k$, let ω_a be the set of vectors in $\mathbf{2}^n$ that extend a:

$$\omega_a := \{v \in \mathbf{2}^n \mid v_1 = a_1 \wedge \dots \wedge v_k = a_k\} \in \Omega$$

and W_a be the set of all Boolean functions that are true on ω_a:

$$W_a := \{\omega \in \Omega \mid \omega_a \Rightarrow_0 \omega\} \in \mathcal{S}.$$

These definitions straightforwardly imply the following lemma:

▶ **Lemma 5** (`winA_eq`)**.** *For any Boolean function $F : \mathbf{2}^n \to \mathbf{2}$ and any strategy $a \in \mathbf{2}^k$ of player A in the associated Boolean game, we have:*

$$\mathrm{win}_A(a) \iff F \in W_a.$$

Lemma 5 implies that the probability that a winning strategy exists satisfies:

$$\mathbb{P}(\exists a : \mathbf{2}^k.\, \mathrm{win}_A(a)) = \mathbb{P}\left(\bigcup_{a \in \mathbf{2}^k} W_a\right). \tag{1}$$

Then we shall rely on the inclusion-exclusion formula, which we proved in full generality as follows:

▶ **Theorem 6** (`Pr_bigcup_incl_excl`)**.** *For any finite probability space $(\Omega, \mathcal{S}, \mathbb{P})$ and any sequence of events $(S_i)_{0 \le i < n}$, we have:*

$$\mathbb{P}\left(\bigcup_{0 \le i < n} S_i\right) = \sum_{m=1}^{n} (-1)^{m-1} \sum_{\substack{J \subseteq \mathbb{N} \cap [0,n) \\ \mathrm{Card}\, J = m}} \mathbb{P}\left(\bigcap_{j \in J} S_j\right). \tag{2}$$

Proof. For proving this theorem in Coq we formalise a small theory of indicator functions $\mathrm{Ind}_S : \Omega \to \{0,1\}$ for any finite set $S \subseteq \Omega$, including the fact that the expectation satisfies

$\mathbb{E}(\mathrm{Ind}_S) = \mathbb{P}(S)$, then formalise an algebraic proof of the inclusion-exclusion formula.[4] These proofs strongly rely on the `bigop` theory of the `MathComp` library, as well as on the tactic "`under`" that we developed in Ltac to easily "rewrite under lambdas" (e.g., under the $\sum$ symbol). These tactics facilities will be further detailed in Section 6. ◀

Hence the following result:

▶ **Theorem 7** (`Pr_ex_winA`). *For any finite probability space $(\Omega, \mathcal{S}, \mathbb{P})$, the probability that there exists some strategy $a = (a_1, \dots, a_k)$ of A that is winning satisfies:*

$$\begin{aligned}\mathbb{P}(\exists a : \mathbf{2}^k.\, \mathrm{win}_A(a)) &= \sum_{a\in\mathbf{2}^k} \mathbb{P}(W_a) - \sum_{\substack{a,a'\in\mathbf{2}^k\\ a\neq a'}} \mathbb{P}(W_a \cap W_{a'}) + \cdots \\ &= \sum_{m=1}^{2^k} (-1)^{m-1} \sum_{\substack{J\subseteq\mathbf{2}^k\\ \mathrm{Card}\, J=m}} \mathbb{P}\left(\bigcap_{a\in J} W_a\right).\end{aligned}$$

Proof. The result follows from (1) and (2), after reindexing all big-operators $\bigcup$, $\sum$, $\bigcap$ by natural numbers instead of Boolean vectors $a \in \mathbf{2}^k$, or conversely. ◀

Theorem 7 is applicable with *any* probability $\mathbb{P}$, but it is not easy to handle. In the upcoming section, we shall investigate in more detail the case when $\mathbb{P}$ is relatively simple.

Before refining Theorem 7 with specific definitions of $\mathbb{P}$, we formally study the dual case (i.e., the existence of a winning strategy from the point of view of player B).

▶ **Definition 8** (`winB`). For any game $F : \mathbf{2}^k \times \mathbf{2}^{n-k} \to \mathbf{2}$, a strategy $b = (b_1, \dots, b_{n-k})$ of player B is winning if it wins against any strategy $a \in \mathbf{2}^k$ of A:

$$\mathrm{win}_B[F](b) := \forall a \in \mathbf{2}^k.\ F(a, b) = \mathsf{false}.$$

If there is no ambiguity, we shall omit the name of the game and simply write $\mathrm{win}_B(b)$.

A first result consists in showing that player B wins in a given game if and only if player A wins in the "dual game".

▶ **Lemma 9** (`winB_eq`). *Any Boolean game $F : \mathbf{2}^k \times \mathbf{2}^{n-k} \to \mathbf{2}$ (with n variables, k of which are controlled by player A) can be associated with a dual Boolean game $F' : \mathbf{2}^{n-k} \times \mathbf{2}^k \to \mathbf{2}$ such that*

$$\mathrm{win}_B[F](b) \iff \mathrm{win}_A[F'](b)$$

Proof. First, we define the dual game $F' :=$ `bool_game_sym`(F) associated with F as:

$$F' := (b, a) \mapsto \neg F(a, b).$$

Then, we define `bool_game_sym'` (the inverse of function `bool_game_sym`) and show that both functions are bijections. In the formal development, the related lemmas are named `bool_game_sym_bij` and `bool_game_sym'_bij`. ◀

We then deduce the following result, which relates the probability of existence of a winning strategy for player B with respect to that of player A.

[4] taking inspiration from the proof path presented at `https://en.wikipedia.org/wiki/Inclusion-exclusion_principle#Algebraic_proof`

▶ **Theorem 10** (`Pr_ex_winB`). *For any finite probability space $(\Omega, \mathcal{S}, \mathbb{P})$, the probability that there exists some strategy $b = (b_1, \ldots, b_{n-k})$ of B that is winning satisfies:*

$$\mathbb{P}\left(\exists b : \mathbf{2}^{n-k}.\, \mathrm{win}_B[F](b)\right) = \mathbb{P}\left(\exists a : \mathbf{2}^{n-k}.\, \mathrm{win}_A[F'](a)\right),$$

where $F' :=$ `bool_game_sym`(F).

Proof. The proof straightforwardly derives from Lemma 9. ◀

Finally, we prove the intuitive fact that the events "$\exists a.\, \mathrm{win}_A(a)$" and "$\exists b.\, \mathrm{win}_B(b)$" are disjoint, and thereby their probability adds up:

▶ **Lemma 11** (`Pr_ex_winA_winB_disj`). *For any finite probability space $(\Omega, \mathcal{S}, \mathbb{P})$, we have:*

$$\mathbb{P}\left(\exists a.\, \mathrm{win}_A(a) \;\vee\; \exists b.\, \mathrm{win}_B(b)\right) = \mathbb{P}\left(\exists a.\, \mathrm{win}_A(a)\right) + \mathbb{P}\left(\exists b.\, \mathrm{win}_B(b)\right).$$

Proof. Given the definitions of win_A and win_B, for a given game F and any strategies a and b, the events "$\mathrm{win}_A(a)$" and "$\mathrm{win}_B(b)$" are disjoint. So the proof path just amounts to lift this fact (considering existence) and use the additivity of $\mathbb{P}$. ◀

3 Bernoulli Process and Winning Strategies

In this section we still consider the space $\Omega = \mathbf{2}^{\mathbf{2}^n}$ of random Boolean formulas of n variables endowed with the discrete σ-algebra $\mathcal{S} = \mathbf{2}^{\mathbf{2}^{\mathbf{2}^n}}$, and the associated Boolean games with parameter $0 \leq k \leq n$. But now, we assume that the Boolean formulas (in DNF) are determined by a random choice of the Boolean vectors that satisfy the formulas.

To be more precise, we assume the probability that each vector $v \in \mathbf{2}^n$ belongs to the truth-set of the formula F is equal to p, $(0 \leq p \leq 1)$. As usual, we write $q = 1 - p$.

In the sequel, we shall often identify Boolean functions $F : \mathbf{2}^n \to \mathbf{2}$ and their truth-set $F^{-1}(\{\mathsf{true}\}) \in \mathbf{2}^{\mathbf{2}^n}$. In the Coq formalisation, the distinction between the two is always made explicit, and the function that gives the truth-set of a Boolean function is implemented by a function

```
finset_of_bool_fun : ∀ n : nat, bool_fun n -> {set bool_vec n}
```

and the inverse of this function is formalised as a function `DNF_of` (disjunctive normal form).

Our setup amounts to constructing a Bernoulli process, that is a series of independent Bernoulli trials, to decide whether each vector $v \in \mathbf{2}^n$ belongs to the truth-set of F or not. We obtain the following result:

▶ **Lemma 12** (`dist_BernoulliE`). *For any $F \in \Omega$, the probability of an elementary event $\{F\}$ with respect to the considered probability $\mathbb{P}_{n;\,p}$ (modelling a series of 2^n independent Bernoulli trials of parameter p) is:*

$$\mathbb{P}_{n;\,p}(\{F\}) = p^m(1-p)^{2^n - m}$$

where m denotes the number of vectors in the truth-set of F, and $2^n - m$ denotes the number of vectors in the truth-set of the negation of F.

Proof. The proof (and its formal counterpart in Coq) straightforwardly derives from the definitions. ◀

For now, we assume that the choices of player A and B are done simultaneously. A wins if the value of F is `true`, otherwise B wins. What is the probability that A has a winning strategy?

First, suppose that the strategy a of A is fixed, and let us compute the probability that it is winning. We first prove the following

▶ **Lemma 13** (`Pr_implies0_Bern`). *Let $S \subseteq \mathbf{2}^n$, and let us write $m := \operatorname{Card} S$. Then the probability that F is true on S satisfies: $\mathbb{P}_{n;\,p}(S \Rightarrow_0 F) = p^m$.*

Proof. We follow the following proof path:

$$\begin{aligned}
\mathbb{P}_{n;\,p}(S \Rightarrow_0 F) &= \sum_{\substack{F \\ S\subseteq F}} \mathbb{P}_{n;\,p}(\{F\}) \\
&= \sum_{\substack{S'\subseteq \mathbf{2}^n\setminus S \\ F=S\cup S'}} \mathbb{P}_{n;\,p}(\{F\}) \\
&= \sum_{S'\subseteq \mathbf{2}^n\setminus S} p^{\operatorname{Card}(S\cup S')} q^{2^n-\operatorname{Card}(S\cup S')} \text{ by Lemma 12} \\
&= \sum_{m'=0}^{2^n-m} \binom{2^n-m}{m'} p^{m+m'} q^{2^n-m-m'} \\
&= p^m \sum_{m'=0}^{2^n-m} \binom{2^n-m}{m'} p^{m'} q^{(2^n-m)-m'} \\
&= p^m (p+q)^{2^n-m} \\
&= p^m.
\end{aligned}$$ ◀

▶ **Lemma 14** (`card_w_a_Bern`). *For any strategy a of player A, we have*

$$\operatorname{Card} w_a = 2^{n-k}.$$

Proof. This lemma easily follows from the fact that w_a is the image of the strategy space $\mathbf{2}^{n-k}$ of B by an injective function. ◀

Hence the following theorem, which gives the probability that a fixed strategy of A is winning:

▶ **Theorem 15** (`Pr_winA_Bern`). *For any strategy a of player A, we have*

$$\mathbb{P}_{n;\,p}(\mathrm{win}_A(a)) = p^{2^{n-k}}.$$

Proof. This result is an immediate consequence of Lemmas 13 and 14. ◀

Now, let us determine what is the probability that A has at least one winning strategy. One may first notice the following

▶ **Lemma 16** (`w_trivIset`). *The truth-sets of w_a (for $a \in J \subset \mathbf{2}^k$) are pairwise disjoint.*

Proof. By contradiction: if we had $a, a' \in \mathbf{2}^k$ such that $w_a \neq w_{a'}$ and $w_a \cap w_{a'} \neq \emptyset$, then let us pose $x \in w_a \cap w_{a'}$. By unfolding Definition 4, this means that the first k bits of x coincides with all bits of a, and likewise for a'. This implies that $a = a'$ and thereby $w_a = w_{a'}$, which contradicts the initial hypothesis. ◀

Lemma 16 implies that we have

$$\operatorname{Card}\left(\bigcup_{a\in J} w_a\right) = \sum_{a\in J} \operatorname{Card} w_a = \operatorname{Card} J \cdot 2^{n-k}. \tag{3}$$

We can now prove the following

▶ **Theorem 17** (`Pr_ex_winA_Bern`). *For any n and k, if $\mathbb{P}_{n;\,p}$ follows the Bernoulli scheme that we previously constructed, the probability that player A has a winning strategy is:*

$$\mathbb{P}_{n;\,p}(\exists a.\, \mathrm{win}_A(a)) = 1 - \left(1 - p^{2^{n-k}}\right)^{2^k}.$$

Proof. Thanks to Theorem 7, we can write:

$$\begin{aligned}
\mathbb{P}_{n;\,p}(\exists a : \mathbf{2}^k.\, \mathrm{win}_A(a)) &= \sum_{m=1}^{2^k} (-1)^{m-1} \sum_{\substack{J\subseteq \mathbf{2}^k\\ \operatorname{Card} J = m}} \mathbb{P}_{n;\,p}\left(\bigcap_{a\in J} W_a\right)\\
&= \sum_{m=1}^{2^k} (-1)^{m-1} \sum_{\substack{J\subseteq \mathbf{2}^k\\ \operatorname{Card} J = m}} \mathbb{P}_{n;\,p}\left(\bigcap_{a\in J} [w_a \Rightarrow_0 F]\right)\\
&= \sum_{m=1}^{2^k} (-1)^{m-1} \sum_{\substack{J\subseteq \mathbf{2}^k\\ \operatorname{Card} J = m}} \mathbb{P}_{n;\,p}\left[\left(\bigcup_{a\in J} w_a\right) \Rightarrow_0 F\right]\\
&= \sum_{m=1}^{2^k} (-1)^{m-1} \sum_{\substack{J\subseteq \mathbf{2}^k\\ \operatorname{Card} J = m}} p^{\left(\operatorname{Card}\left(\bigcup_{a\in J} w_a\right)\right)} \text{ by Lemma 13}\\
&= \sum_{m=1}^{2^k} (-1)^{m-1} \sum_{\substack{J\subseteq \mathbf{2}^k\\ \operatorname{Card} J = m}} p^{m\cdot 2^{n-k}} \text{ by using (3) and Lemma 14}\\
&= \sum_{m=1}^{2^k} (-1)^{m-1} \binom{2^k}{m} p^{m\cdot 2^{n-k}}\\
&= 1 - \sum_{m=0}^{2^k} \binom{2^k}{m} (-p^{2^{n-k}})^m 1^{2^k-m}\\
&= 1 - \left(1 - p^{2^{n-k}}\right)^{2^k}.
\end{aligned}$$

◀

By duality, one can derive the existence of a winning strategy for player B:

▶ **Corollary 18** (`Pr_ex_winB_Bern`). *For any p, n, k, if $\mathbb{P}_{n;\,p}$ denotes the considered Bernoulli scheme (with parameters $0 \le p \le 1$ and $n \in \mathbb{N}$) and if k denotes the number of variables controlled by player A, then the probability that player B has a winning strategy is:*

$$\mathbb{P}_{n;\,p}(\exists b : \mathbf{2}^{n-k}.\, \mathrm{win}_B(b)) = 1 - \left(1 - (1-p)^{2^k}\right)^{2^{n-k}}.$$

Proof. The result follows from Theorems 10 and 17. Also, the proof makes use of our `under` tactic for rewriting under lambdas (it will be presented in Section 6). ◀

Table 1 The probability that a winning strategy exists neither for A nor for B ($n = 10$).

p\k	1	2	3	4	5	6	7	8	9
0.25	1.52e-184	5.11e-43	1.37e-6	0.525	0.997	1	0.998	0.367	4.46e-15
0.5	1.07e-64	6.68e-8	0.606	0.999	1	0.999	0.606	6.68e-8	1.07e-64
0.75	4.46e-15	0.367	0.998	1	0.997	0.525	1.37e-6	5.11e-43	1.52e-184

Table 2 The probability that a winning strategy exists neither for A nor for B ($n = 20$).

p\k	1	2	3	4	5	6	7	8	9	10
0.25	1.27e-188231	2.04e-43307	2.15e-6005	1.99e-287	3.72e-2	1	1	1	1	1
0.5	1.32e-65504	2.74e-7348	1.61e-223	0.368	1	1	1	1	1	1
0.75	7.53e-14696	2.58e-446	0.135	1	1	1	1	1	1	1

p\k	11	12	13	14	15	16	17	18	19
0.25	1	1	1	1	1	1	0.135	2.58e-446	7.53e-14696
0.5	1	1	1	1	1	0.368	1.61e-223	2.74e-7348	1.32e-65504
0.75	1	1	1	1	3.72e-2	1.99e-287	2.15e-6005	2.04e-43307	1.27e-188231

▶ **Corollary 19** (`Pr_nex_winA_winB_Bern`)**.** *For any p, n, k, if $\mathbb{P}_{n;\,p}$ denotes the considered Bernoulli scheme (with parameters $0 \le p \le 1$ and $n \in \mathbb{N}$) and if k denotes the number of variables controlled by player A, then the probability that no player has a winning strategy is:*

$$\mathbb{P}_{n;\,p}\Big(\neg\big((\exists a.\,\mathrm{win}_A(a)) \vee (\exists b.\,\mathrm{win}_B(b))\big)\Big) = \Big(1 - p^{2^{n-k}}\Big)^{2^k} + \Big(1 - (1-p)^{2^k}\Big)^{2^{n-k}} - 1. \quad (4)$$

Proof. The result follows from Lemma 11, Theorem 17 and Corollary 18. ◀

3.1 Discussion

The computations above may seem elementary, but lead to some observations that are less trivial. As we may see, there is a considerable probability that there is no winning strategy at all. For example, if $p \in \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$, $n \in \{10, 20\}$, $0 < k < n$, the probability that a winning strategy exists neither for A nor for B (cf. Equation (4)) is given in Tables 1 and 2 (the values were computed using Sollya[5] with 3-digit decimal output). In both tables, it should be noted that 1 actually means a value extremely close to 1, not 1 exactly.

Also, one may notice that when $p \in (0, 1)$ is fixed, $k = c \cdot n$ for a given constant $0 < c < 1$, and n tending towards $+\infty$, the probability that a winning strategy exists neither for player A, nor for player B, tends to 1.

If (for some game F) a winning strategy exists neither for A nor for B, then the order of moves becomes important. Indeed, let a be an arbitrary strategy of A. Since it is not winning, there exists at least one b of B such that $F(a, b) = \mathsf{false}$. If B makes his choice after A, he may always win. Similarly, if A makes his choice after B, he may always win.

We shall elaborate on this observation and give a motivating example in Appendix A.

Bradfield, Gutierrez and Wooldridge notice [4] (as do some other authors): "As they are conventionally formulated, Boolean games assume that players make their choices in ignorance of the choices being made by other players – they are games of simultaneous moves.

[5] `http://sollya.gforge.inria.fr/`

For many settings, this is clearly unrealistic." Our simple probabilistic analysis provides a direct quantitative argument to support this general observation.

4 Partial Information on the Opponent's Choices

Now, let us consider the case when A may have partial information about the choices of B before making his own choice. Without loss of generality, we may assume that he knows the values of the first s variables among the variables $v_{k+1}, ..., v_n$ controlled by B. We shall consider the probability of existence of strategies of A such that for every vector $b_{1:s} = (b_1, ..., b_s) \in \mathbf{2}^s$ there exists a strategy $a \in \mathbf{2}^k$ that wins against any strategy $b \in \mathbf{2}^{n-k}$ where first s values coincide with $b_{1:s}$.

In other words, we are interested in the probability of guaranteed win by A when s choices by B among $n-k$ are known (assuming $0 \leq s \leq n-k$). We thus introduce the following predicate:

▶ **Definition 20** (`winA_knowing`)**.** For any game $F : \mathbf{2}^k \times \mathbf{2}^{n-k} \to \mathbf{2}$ and any $b_{1:s} \in \mathbf{2}^s$, we say that a strategy $a \in \mathbf{2}^k$ is winning under the knowledge of $b_{1:s}$ if it is winning against all strategy profile $(a, b) \in \mathbf{2}^k \times \mathbf{2}^{n-k}$ that is compatible with $b_{1:s}$:

$$\mathrm{win}_A(a \mid b_{1:s}) := \forall b \in \mathbf{2}^{n-k}.\ \texttt{compat\_knowing}(b_{1:s}, b) \implies F(a, b) = 1,$$

where

$$\texttt{compat\_knowing}(b_{1:s}, b) := \forall i \in \mathbf{2}^s.\ (b_{1:s})_i = b_i.$$

For relating this predicate with that of Definition 3, the proof of the following lemma is immediate:

▶ **Lemma 21** (`winA_knowingE`)**.** *For any game $F : \mathbf{2}^k \times \mathbf{2}^{n-k} \to \mathbf{2}$ and any bit-vectors $b_{1:s} \in \mathbf{2}^s$ and $a \in \mathbf{2}^k$, we have:*

$$\mathrm{win}_A[F](a \mid b_{1:s}) = \mathrm{win}_A[\mathrm{bgk}(F, b_{1:s})](a)$$

where $\mathrm{bgk}(F, b_{1:s}) : \mathbf{2}^k \times \mathbf{2}^{n-s-k}$ *is the Boolean game defined by:*

$$\mathrm{bgk}(F, b_{1:s})(a, b') = F(a, (b_{1:s}, b')).$$

Now, to compute the probability $\mathbb{P}_{n;p}\left(\forall b_{1:s} \in \mathbf{2}^s.\ \exists a \in \mathbf{2}^k.\ \mathrm{win}_A(a \mid b_{1:s})\right)$ in the space $(\Omega, \mathcal{S}, \mathbb{P}_{n;p})$ introduced in Section 3, we shall first construct a probability space $(\Omega', \mathcal{S}', \mathbb{P}')$ that is provably isomorphic to $(\Omega, \mathcal{S}, \mathbb{P}_{n;p})$, but which is simpler to handle.

First, we note that there are 2^s possible Boolean vectors $b_{1:s} = (b_1, ..., b_s)$ and for all $b_{1:s}$, we pose

$$B_{b_{1:s}} = \{v \in \mathbf{2}^n \mid v_{k+1} = b_1 \wedge \cdots \wedge v_{k+s} = b_s\}.$$

The family $(B_{b_{1:s}})_{b_{1:s} \in \mathbf{2}^s}$ constitutes a partition of $\mathbf{2}^n$ (we have $\mathbf{2}^n = \bigcup_{b_{1:s} \in \mathbf{2}^s} B_{b_{1:s}}$, intersections of $B_{b_{1:s}}$ for different $b_{1:s}$ are empty, and no set $B_{b_{1:s}}$ is empty).

Second, we define $\Omega_{b_{1:s}} := \mathbf{2}^{B_{b_{1:s}}}$ as the powerset of $B_{b_{1:s}}$ and show that there is a one-to-one correspondence between $\Omega_{b_{1:s}}$ and $\mathbf{2}^{\mathbf{2}^{n-s}}$. We shall denote the corresponding bijections by $g : \Omega_{b_{1:s}} \to \mathbf{2}^{\mathbf{2}^{n-s}}$ and $h : \mathbf{2}^{\mathbf{2}^{n-s}} \to \Omega_{b_{1:s}}$. In the formal development, the related lemmas are named `bool_fun_of_OmegaB_bij` and `OmegaB_of_bool_fun_bij`.

Next, we consider the probability $\mathbb{P}_{b_{1:s}} := \mathbb{P}_{n-s;\,p} \circ h^{-1}$ defined as the pushforward distribution (with respect to function h) of the Bernoulli process $\mathbb{P}_{n-s;\,p}$ with parameters $n-s$ and p.

We then consider the product space $(\Omega', \mathcal{S}', \mathbb{P}')$ defined by:

$$\begin{cases} \Omega' = \prod_{b_{1:s} \in \mathbf{2}^s} \Omega_{b_{1:s}} \\ \mathcal{S}' = \mathbf{2}^{\Omega'} \\ \mathbb{P}' = \bigotimes_{b_{1:s} \in \mathbf{2}^s} \mathbb{P}_{b_{1:s}} \end{cases}$$

Relying on functions g and h, we finally show that there is a one-to-one correspondence between Ω' and $\Omega = \mathbf{2}^{\mathbf{2}^n}$. We shall denote the corresponding bijections by $g' : \Omega' \to \Omega$ and $h' : \Omega \to \Omega'$. In the formal development, the related lemmas are named `bool_fun_of_Omega'_bij` and `Omega'_of_bool_fun_bij`.

We now prove that the spaces $(\Omega, \mathcal{S}, \mathbb{P}_{n;\,p})$ and $(\Omega', \mathcal{S}', \mathbb{P}')$ are isomorphic:

▶ **Lemma 22** (`isom_dist_Omega'`)**.** *The probability distribution $\mathbb{P}_{n;\,p}$ (defined in Section 3 as the Bernoulli process with parameters n and p) is extensionally equal to the pushforward distribution of $\mathbb{P}'$ with respect to function g'.*

Proof. In the Coq formal proof, this lemma amounts to splitting a big-operator expression with respect to the partition of $\mathbf{2}^n$, reindexing big-operator expressions half-a-dozen times, and rewriting "cancellation lemmas" for simplifying the composition of a bijection and its inverse function. Also, the use of our `under` tactic (see Section 6) contributed to simplify the mechanisation of this proof. ◀

A key ingredient for the sequel will be the following

▶ **Lemma 23** (`ProductDist.indep`)**.** *Given a finite type I and a family of finite probability spaces $(\Omega_i, \mathcal{S}_i{=}\mathbf{2}^{\Omega_i}, \mathbb{P}_i)_{i \in I}$, the product space defined by*

$$\begin{cases} \Omega_\Pi = \prod_{i \in I} \Omega_i \\ \mathcal{S}_\Pi = \mathbf{2}^{\Omega_\Pi} \\ \mathbb{P}_\Pi = \bigotimes_{i \in I} \mathbb{P}_i \end{cases}$$

is such that the projections $(\pi_i : \Omega_\Pi \to \Omega_i)_{i \in I}$ are independent random variables. In other words, for any family of events $(Q_i)_{i \in I} \in \prod_{i \in I} \mathcal{S}_i$, we have:

$$\mathbb{P}_\Pi \left(\bigcap_{i \in I} \pi_i^{-1}(Q_i) \right) = \prod_{i \in I} \mathbb{P}_i(Q_i).$$

We can now prove the following

▶ **Theorem 24** (`Pr_ex_winA_knowing_Bern`)**.** *For all $p \in [0,1]$ and for all integers n, k, s satisfying $0 \le s \le n-k \le n$, if $\mathbb{P}_{n;\,p}$ is the Bernoulli process with parameters n and p defined in Section 3, the probability of guaranteed win for player A knowing s choices of player B among his $n-k$ variables is:*

$$\mathbb{P}_{n;\,p} \left(\forall b_{1:s} \in \mathbf{2}^s.\ \exists a \in \mathbf{2}^k.\ \operatorname{win}_A(a \mid b_{1:s}) \right) = \left(1 - \left(1 - p^{2^{n-k-s}} \right)^{2^k} \right)^{2^s}. \tag{5}$$

Proof. We follow the following proof path:

$$\begin{aligned}&\mathbb{P}_{n;\,p}\left(\forall b_{1:s} \in \mathbf{2}^s.\ \exists a \in \mathbf{2}^k.\ \operatorname{win}_A(a \mid b_{1:s})\right)\\&= \mathbb{P}_{n;\,p}\left\{F \in \Omega \;\middle|\; \forall b_{1:s} \in \mathbf{2}^s.\ \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[F](a \mid b_{1:s})\right\}\end{aligned}$$

hence by using Lemma 21

$$= \mathbb{P}_{n;\,p}\left\{F \in \Omega \;\middle|\; \forall b_{1:s} \in \mathbf{2}^s.\ \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[\operatorname{bgk}(F, b_{1:s})](a)\right\}$$

hence by using Lemma 22

$$= \left(\mathbb{P}' \circ {g'}^{-1}\right)\left\{F \in \Omega \;\middle|\; \forall b_{1:s} \in \mathbf{2}^s.\ \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[\operatorname{bgk}(F, b_{1:s})](a)\right\}$$

hence by using elementary facts on g, g' and the bgk function defined in Lemma 21

$$= \mathbb{P}'\left\{f \in \Omega' \;\middle|\; \forall b_{1:s} \in \mathbf{2}^s.\ f(b_{1:s}) \in \left\{S \in \Omega_{b_{1:s}} \;\middle|\; \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[g(S)](a)\right\}\right\}$$

hence by using Lemma 23

$$= \prod_{b_{1:s} \in \mathbf{2}^s} \mathbb{P}_{b_{1:s}}\left\{S \in \Omega_{b_{1:s}} \;\middle|\; \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[g(S)](a)\right\}$$

hence by definition of $\mathbb{P}_{b_{1:s}}$

$$= \prod_{b_{1:s} \in \mathbf{2}^s} \left(\mathbb{P}_{n-s;\,p} \circ h^{-1}\right)\left\{S \in \Omega_{b_{1:s}} \;\middle|\; \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[g(S)](a)\right\}$$

hence by definition of g and h

$$= \prod_{b_{1:s} \in \mathbf{2}^s} \mathbb{P}_{n-s;\,p}\left\{F \in \mathbf{2}^{\mathbf{2}^{n-s}} \;\middle|\; \exists a \in \mathbf{2}^k.\ \operatorname{win}_A[F](a)\right\}$$

hence by using Theorem 17 in the case of random Boolean functions with $n - s$ variables

$$\begin{aligned}&= \prod_{b_{1:s} \in \mathbf{2}^s} \left(1 - \left(1 - p^{2^{n-s-k}}\right)^{2^k}\right)\\&= \left(1 - \left(1 - p^{2^{n-k-s}}\right)^{2^k}\right)^{2^s}.\end{aligned}$$ ◀

We may compare this probability with the probability of existence of unconditionally winning strategy studied in Section 3 (Theorem 17). The upcoming section will focus on this question.

▶ Remark. In Theorems 17 and 24, we formally studied the *probability of guaranteed win* (knowing partial information on the opponent), that is, the probability that for every value taken by the first s variables of B,[6] there exists a strategy for A that wins against all strategies of B given this fixed value of the first s variables of B. This problem is purely combinatorial and does not depend on the "preferences" of B (regarding the variables that he controls). So this probability will typically be different from the probability of non-guaranteed win for player A, as this latter probability could be influenced by the preferences of B for some choices, the dependency of these choices on F, and so on.

[6] Theorem 17 being a particular case of Theorem 24 ($s = 0$).

5 Probability of Guaranteed Win: Growth Rate

Using the result given by Theorem 24, we would like to study how the probability of guaranteed win grows with each bit of information concerning the choice of B.

For fixed values of $p \in (0,1)$, $n,k \in \mathbb{N}$ such that $0 < k < n$, and for $0 \leq s \leq n-k$, let us write $g(s)$ the quantity given in Equation (5).

First, we note that when s tends to $n-k$, the probability of guaranteed win for A tends to:

$$g(n-k) = \left(1-\left(1-p^{2^0}\right)^{2^k}\right)^{2^{n-k}} = \left(1-(1-p)^{2^k}\right)^{2^{n-k}} = 1 - \underbrace{\left[1-\left(1-(1-p)^{2^k}\right)^{2^{n-k}}\right]}_{\text{proba. of guaranteed win for } B}$$

Then, an interesting question may be: what is the order of growth of the difference

$$\phi(s) := g(s) - g(0) \quad (\in [0,1])$$

with respect to s? The following result is a first answer to this question:

▶ **Theorem 25** (`phi_ineq`). *For any $p \in (0,1)$, $n,k \in \mathbb{N}^*$ such that $0 \leq s \leq n-k$, if the following condition holds:*

$$2^k p^{2^{n-k-s}} < 1, \tag{6}$$

then we have

$$\phi(s) > \left(2^{(k-1)2^s} - 2^k\right) p^{2^{n-k}}, \tag{7}$$

where

$$\phi(s) = g(s) - g(0) = \left(1-\left(1-p^{2^{n-k-s}}\right)^{2^k}\right)^{2^s} - \left(1-\left(1-p^{2^{n-k}}\right)^{2^k}\right).$$

In particular, condition (6) is satisfied as soon as the following, stronger condition is satisfied:

$$s \leq (n-k) - \log_2(k+1) + \log_2(|\log_2 p|). \tag{8}$$

Proof. Let us write $t = p^{2^{n-k-s}}$. By the binomial formula, we have:

$$1-(1-t)^{2^k} = 2^k t - (2^k t)^2 \sum_{i=2}^{2^k} (-1)^i 2^{-2k} \binom{2^k}{i} t^{i-2}. \tag{9}$$

We notice that if (6) holds, that is if $2^k t < 1$, then the absolute value of the $(i+1)$th member of the sum $\sum$ in (9) is less than that of the i-th member because it is obtained by multiplication by $((2^k - i)/(i+1))t < 2^k t$. So, if (6) holds, then the sum (positive) is less than or equal to its first term. Moreover, the first term of the sum $\sum$ in (9) is $2^{-2k}\frac{2^k(2^k-1)}{2} < \frac{1}{2}$. So, if (6) is satisfied for some n, k, s, then from (9) we obtain

$$1-(1-t)^{2^k} \geq 2^k t - \frac{1}{2}(2^k t)^2 = 2^k t\left(1-\frac{1}{2}2^k t\right) > 2^{k-1} t. \tag{10}$$

Thus, under these conditions

$$g(s) = \left(1-\left(1-p^{2^{n-k-s}}\right)^{2^k}\right)^{2^s} > 2^{(k-1)2^s} p^{2^{n-k}}. \tag{11}$$

Next, a similar analysis applied to $\left(1-\left(1-p^{2^{n-k}}\right)^{2^k}\right)$ gives an estimation

$$g(0)=\left(1-\left(1-p^{2^{n-k}}\right)^{2^k}\right)=2^k p^{2^{n-k}}-\sum_{i=1}^{2^k}\binom{2^k}{i}\left(-p^{2^{n-k}}\right)^i\leq 2^k p^{2^{n-k}} \tag{12}$$

Combining (11) and (12) yields the following inequality:

$$g(s)-g(0)>2^{(k-1)2^s}p^{2^{n-k}}-2^k p^{2^{n-k}}=\left(2^{(k-1)2^s}-2^k\right)p^{2^{n-k}}. \tag{13}$$

Finally, the following condition is obviously stronger than (6):

$$2^k p^{2^{n-k-s}}\leq\frac{1}{2},$$

which is equivalent to

$$2^{n-k-s}\log_2 p\leq -(k+1).$$

Since $0<p<1$ we may write instead

$$2^{n-k-s}|\log_2 p|\geq (k+1).$$

Applying the log a second time, we obtain

$$s\leq (n-k)-\log_2(k+1)+\log_2(|\log_2 p|),$$

which is thereby a sufficient condition for (6). ◀

For example, if $p=\frac{1}{2}$, condition (8) becomes

$$s\leq (n-k)-\log_2(k+1). \tag{14}$$

And if $0<p<\frac{1}{2}$, $\log_2(|\log_2 p|)>0$ so we have $-\log_2(k+1)<-\log_2(k+1)+\log_2(|\log_2 p|)$, and thereby we can also rely on condition (14).

It can be noted that inequality (7) essentially gives an order of growth of $2^{(k-1)2^s}$ with respect to the quantity of information s (number of extra bits known by player A), which is much faster than usual orders of growth of s or 2^s.

Still, a more refined study of the behaviour of the function that describes the growth of $g(s)$ (the probability of a guaranteed win depending on s) requires much more effort and space that we could give to it in this exploratory paper. For example, it is intuitively clear that the graph of this function is a typical “S-form” curve (see Figure 1), but it is not easy to determine where the critical points are placed; and for small values of the parameter s, the inequality we get in (7) may be too rude to place with sufficient precision. However the behaviour of this function g may be of interest for strategic planning concerning both players. This remains subject of future work.

6 Remarks on the Formal Setup in the Coq Proof Assistant

6.1 Related Works on Formal Libraries of Probability

There have been several works focusing on the formalisation of measure theory or probability using interactive theorem proving. Some of these works only deal with discrete probability, or focus on the analysis of randomised algorithms; others formalise large fragments of measure theory up to Lebesgue’s integration theory.

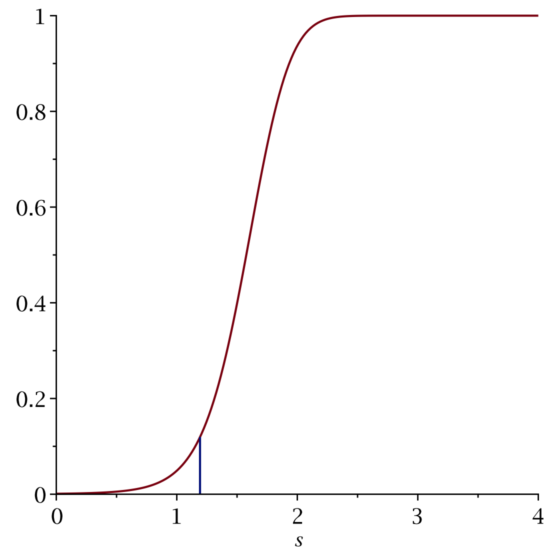


Figure 1 Graph of $g(s)$, for the parameters $p = \frac{1}{2}$, $n = 10$, and $k = 6$. The vertical line at $s \approx 1.19$ indicates the largest $s \in \mathbb{R}$ that satisfies (8).

Using the HOL proof assistant, Hurd [13] developed a framework for proving properties of randomised programs, relying on a formalisation of measure theory, and following a "monadic transformation" approach that provides the user with an infinite sequence of independent, identically distributed *Bernoulli*$(\frac{1}{2})$ random variables.

Still using the HOL proof assistant and building upon Hurd's work, Mhamdi, Hasan and Tahar [12, 15] developed a comprehensive formalisation of measure theory, including Lebesgue's integration theory.

Using the Coq proof assistant, Audebaud and Paulin-Mohring [2] developed the ALEA library[7] that provides a framework to reason about randomised functional programs. Unlike Hurd's approach, it does not require a complete formalisation of measure theory: it is built upon a Coq axiomatisation of the interval $[0, 1]$ and it interprets randomised programs as (discrete) probability distributions.

Still using the Coq formal proof assistant, Affeldt, Hagiwara and Sénizergues [1] developed the Infotheo library[8] that provides a formalisation of information theory. This library comes with a formalisation of finite probability theory and strongly relies on the theories of the MathComp library[9].

For developing our library on random Boolean games, we have chosen to rely on the Infotheo library. Even though it only deals with finite probability, this setting was sufficient for formalising our results and, further, it allowed us to benefit from the facilities of the SSReflect/MathComp library. In the rest of this section, we shall summarise the main notions that we used from the MathComp library and present our related contributions (in Section 6.2), then describe the overall setup of the Infotheo probability theory and present our related contributions (in Section 6.3).

[7] `https://www.lri.fr/~paulin/ALEA/`
[8] `https://staff.aist.go.jp/reynald.affeldt/shannon`
[9] `https://math-comp.github.io/math-comp/`

6.2 MathComp and Our Related Contributions

The MathComp library was born in the Mathematical Components project, which aimed at formalising the Odd Order Theorem in the Coq proof assistant [9], while organising formal proofs into components to get a reusable library of mathematical facts. It is built upon SSReflect, an extension of Coq's proof language that has a native support for the so-called small scale reflection (and in particular Boolean reflection) and often leads to concise proof scripts.

For our library of random Boolean games, we have been especially using the following libraries: (*i*) `fintype` for finite types with decidable equality, (*ii*) `finfun` for functions over finite domains, (*iii*) `finset` for finite sets, (*iv*) `bigop` for properties on "big-operators".

Big-operators and rewriting under lambdas

Regarding big-operators such as $\sum$, $\prod$, $\bigcap$ or $\bigcup$, they are formalised in MathComp as a higher-order function `bigop` that takes several arguments, including a function that specifies the "domain predicate" and the "general term". For example, the sum

$$\sum_{\substack{i=1\\ i \text{ odd}}}^{4} i^2$$

can be formally written as `\sum_(1 <= i < 5 | odd i) i^2`, which amounts to the following term if we get rid of the `\sum` notation:

```
bigop 0 (index_iota 1 5) (fun i:nat => BigBody i addn (odd i) (i^2))
```

If we want to transform such a big-operator expression by rewriting its domain predicate or general term, the following two MathComp lemmas on big-operators can be used.

```
eq_bigr :
  forall (R : Type) (idx : R) (op : R -> R -> R) (I : Type)
  (r : seq I) (P : pred I) (F1 F2 : I -> R),
  (forall i : I, P i -> F1 i = F2 i) ->
  \big[op/idx]_(i <- r | P i) F1 i = \big[op/idx]_(i <- r | P i) F2 i
```

```
eq_bigl :
  forall (R : Type) (idx : R) (op : R -> R -> R) (I : Type)
  (r : seq I) (P1 P2 : pred I) (F : I -> R),
  P1 =1 P2 ->
  \big[op/idx]_(i <- r | P1 i) F i = \big[op/idx]_(i <- r | P2 i) F i
```

Still, applying them directly would require to provide the entire term corresponding to the function we want to obtain.

We thus developed a Coq tactic "`under`" for *rewriting under the lambdas* of big-operators.

A generalised version of our tactic, also applicable for MathComp notions such as matrices, polynomials, and so on, is available online at `https://github.com/erikmd/ssr-under-tac` and we plan to submit it for possible inclusion in MathComp.

Below is a typical example of use for that generalised implementation of the `under` tactic.

For a goal that looks like

```
  A : finType
  n : nat
  F : A -> nat
========================================================
  0 <= \sum_(0 <= k < n)
       \sum_(J in {set A} | #|J :&: [set: A]| == k)
       \sum_(j in J) F j
```

the proof script

```
under eq_bigr [k Hk] under eq_bigl [J] rewrite setIT.
```

will yield the following goal:

```
  A : finType
  n : nat
  F : A -> nat
========================================================
  0 <= \sum_(0 <= k < n)
       \sum_(J in {set A} | #|J| == k)
       \sum_(j in J) F j
```

Dependent product of finTypes

MathComp has built-in support for finite functions: for any `(A:finType)` and `(T:Type)`, the notation `{ffun A -> T}` stands for the type of finite functions from `A` to `T`. If n denotes the cardinal of `A`, these functions are represented by a n-tuple of elements of `T`, which allows one to obtain convenient properties such as the extensionality of finite functions, which wouldn't hold otherwise in the constructive, intensional logic of Coq.

If `T` is also a finite type, then the MathComp library allows one to automatically retrieve (thanks to type inference and so-called canonical structures) a finite type structure for the type `{ffun A -> T}` itself. Thus, this construct amounts to the non-dependent product of a `finType`.

However, for formalising our results and in particularly to construct the type Ω' that appears in Section 4, we have been led to formalise the dependent product of a finite family of finite types. This material is gathered in a file `fprod.v` which provides a type `fprod`, some notations in MathComp style and several support results such as lemmas `fprodP` and `fprodE`, whose signature is as follows:

```
fprod : forall I : finType, (I -> finType) -> finType

fprodP : forall (I : finType) (T_ : I -> finType) (f1 f2: fprod I T_),
         (forall x : I, f1 x = f2 x) <-> f1 = f2

fprodE : forall (I : finType) (T_ : I -> finType)
                (g : forall i : I, T_ i) (x : I),
         [fprod i => g i] x = g x
```

This theory involves proofs with dependent types, and to facilitate the formalisation process we tried to follow a MathComp formalisation style as much as possible, by using finite functions, records with Boolean conditions, and so on. This enabled us to rely on extensionality of functions, the Altenkirch-Streicher K axiom and proof irrelevance, which can be used "axiom-free" in the decidable fragment of MathComp `finTypes`.

6.3 Infotheo and Our Related Contributions

The Infotheo library relies on MathComp as well as the `Reals` theory from Coq's standard library. Among the Infotheo theories, the `proba` theory was the starting point of our formalisation. It first defines distributions as a dependent record `dist`, gathering a function `pmf` that gives the probability of each elementary event, and a proof that the sum of these probabilities is equal to 1:

```
Record dist (A : finType) :=
  mkDist { pmf :> A -> R+ ;
           pmf1 : \rsum_(a in A) pmf a = 1 }.
```

Then, it defines the probability of a subset of `A` as the sum of the probabilities of all elementary events in `A`:

```
Definition Pr (A : finType) (P : dist A) (E : {set A}) :=
  \rsum_(a in E) P a.
```

Then, basic properties of probability and expectation are provided in this setting.

On top of the Infotheo theories, we have developed the following contributions: (i) a formalisation of the pushforward distribution `dist_img` with the associated lemma

```
Lemma Pr_dist_img :
  forall {A B : finType} (X : A -> B) (PA : dist A) (E : {set B}),
  Pr (dist_img X PA) E = Pr PA (X @^-1: E).
```

(ii) a formal proof of a general version of the inclusion–exclusion theorem that we presented above in Theorem 6; (iii) the product distribution of a family of distributions, whose signature is as follows:

```
ProductDist.d :
  forall (I : finType) (T_ : I -> finType),
  (forall i : I, dist (T_ i)) -> dist (fprod I T_)
```

The associated independence result was presented above as Lemma 23.

7 Conclusion

In this work, we used the basics of the theory of Boolean games. In this sense, our work is obviously related to this area. But to the best of our knowledge, the idea of using probability theory applied to a certain class of Boolean games as a whole (in difference from merely random strategies) is new. The analysis of the whole class of games permits to discover some quantitative properties of these games that would be difficult to discover in the study of an individual game.

Furthermore, we used type theory and interactive theorem proving to formalise our results in order to give strong guarantees on their correctness as well as to extend existing formal libraries with new items.

In particular, we have proved a closed formula for the probability of existence of winning strategies in those random Boolean games. We specialised this result with a probability distribution on Boolean functions that are generated by a Bernoulli scheme on Boolean vectors with any probability p as parameter (it can be noted that this setting subsumes the simpler case where all Boolean functions have the same probability: this latter case corresponds to choosing $p = \frac{1}{2}$ in our setting).

In this paper our methods remained elementary, but they permitted to estimate the relative importance of the cases where the players use simultaneous and alternative moves. Another interesting phenomenon seems to us to be the growth of probability of the win as function of the information about the choices of the opponent. Essentially, it is much faster than usual 2^s where s is the quantity of information (number of extra bits) known by the player. This phenomenon emphasises the difference between the information that is required for winning and the "measure of knowledge" of the opponent and its strategies.

We already mentioned the interest of machine checked verification for the games between autonomous programs (embedded systems).

As a future work, we plan to consider more general classes of probability distributions and explore the "weight" of information with respect to winning in this more general setting.

We plan also to consider more closely the connection with algorithmic games [6].

References

1 Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. Formalization of Shannon's theorems. *J. Autom. Reasoning*, 53(1):63–103, 2014. `doi:10.1007/s10817-013-9298-1`.

2 Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009. `doi:10.1016/j.scico.2007.09.002`.

3 Élise Bonzon. *Modélisation des interactions entre agents rationnels : les jeux booléens.* PhD thesis, Université Toulouse III – Paul Sabatier, Toulouse, France, 2007.

4 Julian C. Bradfield, Julian Gutierrez, and Michael Wooldridge. Partial-order Boolean games: informational independence in a logic-based model of strategic interaction. *Synthese*, 193(3):781–811, 2016. `doi:10.1007/s11229-015-0991-y`.

5 The Coq Development Team. *The Coq Proof Assistant: Reference Manual: version 8.8*, 2018. URL: `https://coq.inria.fr/distrib/V8.8.0/refman/`.

6 Evgeny Dantsin, Jan-Georg Smaus, and Sergei Soloviev. Algorithms in Games Evolving in Time: Winning Strategies Based on Testing. In *Isabelle Users Workshop – ITP 2012*, 2012. 18 pages.

7 Paul E. Dunne and Wiebe van der Hoek. Representation and complexity in boolean games. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 347–359. Springer, 2004. `doi:10.1007/978-3-540-30227-8_30`.

8 Danièle Gardy. Random Boolean expressions. In René David, Danièle Gardy, Pierre Lescanne, and Marek Zaionc, editors, *Computational Logic and Applications, CLA '05*, volume AF of *DMTCS Proceedings*, pages 1–36, Chambéry, France, 2006. Discrete Mathematics and Theoretical Computer Science.

9 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. `doi:10.1007/978-3-642-39634-2_14`.

10 Paul Harrenstein. *Logic in Conflict. Logical Explorations in Strategic Equilibrium.* PhD thesis, Utrecht University, 2004.

11 Paul Harrenstein, Wiebe van der Hoek, John-Jules Meyer, and Cees Witteveen. Boolean Games. In J. van Benthem, editor, *Proceedings of the 8th International Conference on*

Theoretical Aspects of Rationality and Knowledge (TARK'01), pages 287–298, San Francisco, 2001. Morgan Kaufmann.

12 Osman Hasan and Sofiène Tahar. Using theorem proving to verify expectation and variance for discrete random variables. *J. Autom. Reasoning*, 41(3-4):295–323, 2008. doi:10.1007/s10817-008-9113-6.

13 Joe Hurd. *Formal verification of probabilistic algorithms.* PhD thesis, University of Cambridge, 2002.

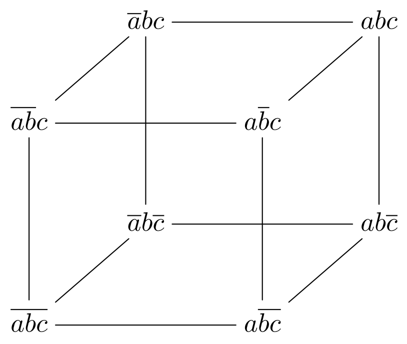
14 Erik Martin-Dorel and Sergei Soloviev. erikmd/coq-bool-games: BoolGames, 2018. doi:10.5281/zenodo.1317609.

15 Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. On the formalization of the Lebesgue integration theory in HOL. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2010. doi:10.1007/978-3-642-14052-5_27.

16 John Riordan and C. E. Shannon. The number of two-terminal series-parallel networks. *Journal of Mathematics and Physics*, 21:83–93, 1942.

A Non-Guaranteed Win: When the Order of Choices Matters

Let us consider an example of three variables a, b, c and two players, Alice who controls a and Bob who controls b, c. Let us consider all possible Boolean functions as payoff functions. There are 256 that may be identified with the subsets of the nodes of the cube below. Each subset is interpreted as the disjunction of the conjunctions in the nodes.



It makes sense to analyse this situation in a purely *combinatorial* way before we consider *randomly generated payoff functions*. We notice the following facts:

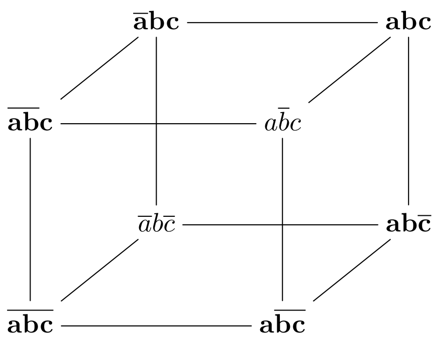
- Alice has an unconditionally winning strategy in 31 cases (these cases correspond to all subsets that contain all nodes of either the face with a or the face with $\overline{a}$; the number of subsets is easily counted by the formula of inclusions-exclusions).
- Bob has an unconditionally winning strategy in 175 cases (the cases correspond to the subsets that *do not* intersect with one of the four edges defined by the choice of two literals among $b, c, \overline{b}, \overline{c}$; the number is counted as above).
- There are 50 cases when neither Alice nor Bob has an unconditionally winning strategy. In these cases the order of choice matters:
  - If Alice chooses the value of a first, then Bob has a winning strategy (he may win in all these cases).
  - Similarly, if Bob chooses the values of b, c first then Alice may win in all these cases.

Now let us consider in more detail the case where the order of choices is $B - A - B$. In fact, here we need to distinguish three subcases:

1. Bob may give a value to any of b, c at his first step.
2. At his first step, Bob gives a value to b, and at his second to c.
3. At his first step, Bob gives a value to c, and at his second to b.

It can be noted that these three variants may correspond to preferences or to obligations (extra constraints) concerning Bob, in line with the remark at the end of Section 4 (page 12). We elaborate on these three subcases below.

1. The choice of Bob may be interpreted as the selection of one of the four faces of the cube that corresponds to b, $\overline{b}$, c, $\overline{c}$ respectively. There are four cases when Alice may win if she knows the first choice of Bob. One subset is shown below in bold, other are obtained by rotation. (We exclude the cases of unconditional win that were counted before.)


$\mathbf{\overline{a}bc}$
$\mathbf{abc}$
$\mathbf{\overline{ab}c}$
$a\overline{b}c$
$\overline{a}b\overline{c}$
$\mathbf{ab\overline{c}}$
$\mathbf{\overline{abc}}$
$\mathbf{a\overline{bc}}$

   In the case displayed above, if Bob has chosen $b =$ `true` then Alice has to choose $a =$ `true` and wins because the remaining formula will be $c \vee \overline{c}$.
2. If at the first step Bob must choose the value of b, then it may be seen as the choice of one of the *two* faces of the cube that correspond to b or $\overline{b}$. This gives Alice more possibilities to win. Indeed, she may win if the subset of nodes includes either $\overline{abc}, \overline{ab}c, ab\overline{c}, abc$ or $a\overline{bc}, a\overline{b}c, \overline{a}b\overline{c}, \overline{a}bc$. We may add one or more nodes to each subset of four, but if we exclude the previously considered cases, we shall have 12 more cases when Alice may win.
3. Similar analysis shows that it will be 12 cases (not considered previously) where Alice may win if Bob must choose the value of c first.

It is important to notice that the choices of Alice and Bob are not necessarily interpreted as the choices of logical values of a, b, c. This model may be used to model any binary choice. Indeed, let $a =$ `true` mean the choice of some value v_a and $a =$ `false` mean the choice of v'_a by Alice. Similarly, Bob may choose one of v_b, v'_b and one of v_c, v'_c. Instead of conjunction of literals (e.g., $a\overline{bc}$) let us take for each such conjunction a predicate[10] $P_{a\overline{bc}}(x, y, z)$ which is true if and only if $x = v_a, y = v'_b, z = v'_c$. Instead of considering the disjunction of these conjunctions, let us take the disjunction of corresponding predicates. It appears that the logical value of the result will exactly be the logical value of the payoff function represented by the DNF (or Boolean function).

The roles of Alice and Bob may be seen as the roles of "coaches" who choose the players for a series of matches. Alice wins if her "champion" wins at least one match. Also, to come back to the situation with random payoff functions, it makes perfect sense that in a real tournament the coach cannot know in advance which matches will be necessary to play.

[10] defined for $(x, y, z) \in \{v_a, v'_a\} \times \{v_b, v'_b\} \times \{v_c, v'_c\}$

The same idea may be used to model markets (the choice $a = \mathsf{true}$ may mean that Alice orders to buy a certain product a, $a = \mathsf{false}$ that she orders to sell, and the presence of $a\overline{bc}$ that she makes profit when she buys at the same time when Bob sells his two products).

This analysis also clearly shows what may be the role of introduction of random choice of payoff functions. It takes into account certain amount of unpredictability in a real situation. Notice that it does not eliminate some “geometric flavour” displayed in the above example.

However, as we emphasised before, we intend to use probability mostly for the analysis of the totality of games with all possible Boolean functions as payoff, rather than for considering one game with a randomly-chosen payoff function (though this may sometimes make sense).

The choice of probability distribution will influence the relative “weight” of the cases that we considered above in a purely combinatorial way, and has to be taken into account when additional conditions are considered, such as the order of moves or access to the information.

For example, if the probability parameter p takes a value of $\frac{1}{2}$ (i.e., if we focus on the instance $\mathbb{P}_{3;\,\frac{1}{2}}$ of the Bernoulli process presented in Section 3), this will give the uniform distribution on the 256 cases considered in the appendix.

The Completeness of BCD for an Operational Semantics

Richard Statman
Carnegie Mellon University, Department of Mathematical Sciences, Pittsburgh, PA 15213, USA
statman@cs.cmu.edu

Abstract

We give a completeness theorem for the BCD theory of intersection types in an operational semantics based on logical relations.

2012 ACM Subject Classification Theory of computation → Computability, Theory of computation → Lambda calculus

Keywords and phrases intersection types, operational semantics, Beth model, logical relations, forcing

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.15

1 Introduction

The theorem of Coppo, Dezani, and Pottinger ([3], [6])states that an untyped lambda term is strongly normalizable if and only if it provably has an intersection type. Here we consider which terms have which types.

We define an operational semantics for the collection of intersection types which assigns to every intersection type A a set of strongly normalizable terms [[A]]. We show that the theory of intersection types BCD (Barendregt, Coppo, Dezani) proves X : A for an untyped term X if and only if $X : [[A]]$ for all interpretations of $[[,]]$ in the operational semantics. Here we shall use the notation ":" for both the formal statement that X has type A, and also set theoretic membership.

Our view of what operational semantics should be begins with Tait style proofs ([8]) of strong normalization. These proofs consider a complete lattice of sets S of strongly normalizable untyped terms([2] 9.3). Not all such sets are considered but the lattice operations are union and intersection. We require that S is closed under reduction, and possibly some other conditions,such as head expansion with strong normalizable arguments, depending on the variant. The operation → is then introduced

$$S \to T = \{X | \text{for all } Y : S \Rightarrow (XY) : T\}.$$

This is certainly familiar from the theory of logical relations ([2] 3.3) for the simple typed case, positive recursive types, and our principal concern in this note; intersection types. Given an intersection type A, if the atoms (atomic types) of A are evaluated among the sets S then A has a value among the sets S. This will be the interpretation $[[A]]$.

2 Beth Models

SN is the set of strongly normalizable untyped terms. Here, we do not distinguish beta from beta-eta strong normalizability since they are equivalent. A Beth model consists of a pair (O, E) where O is a poset with partial order [, and E is a monotone map from O x Atoms into the subsets of SN closed under beta reduction and we shall assume that $E(p, a)$ is non empty except possibly when p is the [smallest element of O, should this exist.

For λ terms X we define the "forcing relation" $\models$ by

$$\begin{array}{lll} p & \models & X : a \text{ if for all } q]p \text{ there exists } r]q\ s.t.\ X : E(r, a) \\ p & \models & X : A/\backslash B \text{ if } p \models X : Ap \models X : B \\ p & \models & X : A \to B \text{ if whenever } q]p \text{ and } q \models U : A \text{ there exists } r]q \\ & & \text{such that } r \models (XU) : B \end{array}$$

and we assume that E satisfies the generalized monotonicity property if $[Y/x]X : E(p, a), q]p$, and $q \models Y : A$ then there exists $r]q$ such that $(\backslash xXY) : E(r, a)$ where $[Y/x]$ is the the substitution operation (the term Y for the variable x).

▶ **Definition 1.** An O chain (linearly ordered subset) W is generic if

(i) for any X and atom a there exists $p : W$ such that either $X : E(p, a)$ or there is no $q]p$ such that $q \models X : a$, and

(ii) for each $A \to B$ there exists $p : W$ such that either $p \models X : A \to B$ or there exists U and $q : W$ such that $q]p$ and $q \models U : A$ but there is no $r]q$ such that $r \models (XU) : B$. We could just as easily use directed subsets of O instead of chains but chains suffice.
For what follows the reader should consult the definition of BCD in [2] which appears on pages 582-583,but without the top element (U_{top}). When we wish to include the top element, U_{top}, together with its axiom ([2] page 583), we will write $BCD + U_{top}$. Especially, the reader should look at the definitions of equality and the ordering of types on page 582. These are reproduced in the appendix.

Facts

1. if $p \models X : A$ and $q]p$ then $q \models X : A$
2. $p \models X : A$ iff for each $q]p$ there exists $r]q$ $s.t.$ $r \models X : A$
3. if $p \models X : A$ and

   $A < B$ or $A = B$ in BCD (page 582)

   then $p \models X : B$
4. if W is generic then for any X and atom a there exists $p : W$ such that for all $q]p$ we have $q \models X : a$ or there is no $q]p$ $s.t.$ $q \models X : a$
5. if W is generic then for any X and $A/\backslash B$ there exists a $p : W$ such that $p \models X : A/\backslash B$ or there is no $q]p$ such that $q \models X : A/\backslash B$
6. if W is an O chain with a maximal element then there exists a generic O chain extending W.

▶ **Proposition 2.** *Let W be a generic O chain and set $R(A) = \{X|$ there exists $p : W$ such that $p \models X : A\}$. Let $X : SN$ then*

(i) $X : R(a)$ iff *there exists $p : W$ s.t. $X : E(p, a)$*

(ii) $X : R(A \to B)$ iff *for each*
$U : R(A)$ we have $(XU) :\ R\ (B)$

(iii) $X : R(A/\backslash B)$ iff $X : R(A)$ & $X : R(B)$

Proof. by induction on A. The basis case (i) is by definition. Induction step; Case (ii) $\Rightarrow$. Suppose that we have a $p : W$ such that $p \models X : A \Rightarrow B$ and we have $U : R(A)$. Thus there exists $q : W$ such that $q \models U : A$. By fact (1) we may assume that $q]p$. Now for any $r]q$

there exists $t]r$ such that $t \models (XU) : B$ but W is generic so there must be an $r : W$ such that $r \models (XU) : B$. That is $(XU) : R(B)$. $\Leftarrow$. Suppose that for each $U : R(A)$ we have $(XU) : R(B)$. Now if there is no $p : W$ such that $p \models X : A \to B$, since W is generic, there exists $p : W$ and aU such that $p \models U : A$ but there is no $q]p$ such that $q \models (XU) : B$. But by fact (1) this contradicts the hypothesis. Case (iii) similar to case (ii). ◀

The proposition clearly states that if the atoms a are evaluated $\{X|$ for some $p : W$ we have $X : E(p, a)\}$ then the value of the type A mentioned in the introduction is $R(A)$

▶ **Example 3** (finite sets). In this case we let O be the collection of finite sets of SN terms closed under beta-eta reduction and ordered by inclusion. We set $E(p, a) = p$. Suppose that $A = A(1) \to (...(A(t) \to)...)$ and $\sim (p \models X : A)$. Then there exists $q]p$ and $Y(1), ..., Y(t)$ s, t $q \models Y(i) : A(i)$ for $i = 1, ..., t$ but there is no $r]q$ with $XY(1)...Y(t) : r$. But this can only be the case if $XY(1)...Y(t)$ is not SN. Thus we can find a generic W such that $X : R(A)$ or there exists $Y(1), ..., Y(t)$ $s.t.$ $Y(i) : R(A(i))$ for $i = 1, ..., t$ and $XY(1)...Y(t)$ is not SN.

▶ **Example 4.** In this case we consider sets $S : O$ of closed beta-eta normal terms for which there exists an integer n such that $X : S$ iff every path in the Bohm tree of X has at most n lambdas and every node in the Bohm tree ([1] pg 212) of X has at most n descendants. Then for any partial recursive function f which is total on S and maps S to S there exists $M : R(S \to S)$ such that for any $N : S$ we have $MN = f(N)$ modulo beta-eta conversion.

▶ **Proposition 5.** *Suppose that O has a smallest element 0. Then, $0 \models X : A$ iff for every generic W we have $X : R(A)$.*

Proof. Immediate by facts (1)–(6). ◀

We next consider the theory BCD with its provability relation $\vdash$ as described in [2] and reproduced in the appendix. A basis F is a map from a finite set of lambda calculus variables, $dom(F)$, to the set of types. Below we shall often conflate F with the finite set

$$\{x : F(x)|x : dom(F)\}.$$

Let O, E be as above and W generic.

▶ **Proposition 6** (soundness). *Suppose that @ is a substitution and F is a base such that for all $x : dom(F), @(x) : R(F(x))$. Then if in BCD*

$$F \vdash X : A$$

we have $@(X) : R(A)$.

Now let O be the set of bases partially ordered by $F[G$ iff $dom(F)$ is contained in $dom(G)$ and for each $x : dom(F)$ we have $G(x)$ and $F(x)$ are equal types in BCD. Now define $E(a)$ by $X : E(F, a)$ if $FV(X)$ is contained in $dom(F)$ and $F \vdash X : a$. Clearly E is [monotone. In addition, $E(F, a)$ is closed under beta-eta reduction. However, generally $E(F, a)$ is not closed under beta head expansion for reasons similar to the case of BCD. In particular this happens when $(\backslash uUV)$ reduces to X, u is not free in U and there is an $x : FV(V)/\backslash FV(U)$ such that the basis entry $x : F(x)$ prevents V from having a BCD type. Thus we have to verify the generalized monotonicity property to insure soundness.First, we observe that there is no difference between E and $\models$ at atomic types.

Fact 7. $F \models X : a$ iff $X : E(F, a)$

Proof. If $FV(X)$ is contained in $dom(F)$ then the equivalence follows from the monotonicity of E and the weakening rule of BCD ([2] page 585). Otherwise suppose that $F \models X : a$. For each $x : FV(X) - dom(F)$ add a new atom $a(x)$ and extend F to G by $G(x) = a(x)$. Then $G \models X : a$ so by the previous argument $G \vdash X : a$ in BCD. But we may substitute U_{top} for each $a(x)$ and $(\backslash x.xx)(\backslash x.xx)$ for each x. So in $BCD + U_{top}$ we have

$$F \vdash [..., (\backslash x.xx)(\backslash x.xx)/x, ...]X : a$$

and this contradicts the fact that if a term has a BCD type (U_{top} free) in $BCD + U_{top}$ then it is strongly normalizable (theorem 17.2.15 (i) [2]). ◀

▶ **Lemma 7.** *Suppose that $FV(X)$ is contained in $dom(F)$.*

$$F \models X : A \text{ iff } F \vdash X : A \text{ in } BCD$$

Proof. this is proved by induction on A. The basis case is by fact 7. For the induction step the case $A = B/\backslash C$ is obvious. We consider the case $A = B \to C$. $\Rightarrow$. Suppose that $F \models X : B \to C$. Let z be a new variable and extend F by G by with $G(z) = B$. Since $G \models z : B$ by induction hypothesis $G \models z : B$ thus there exists $H]G$ such that $H \models Xz : C$. Again by induction hypothesis $H \vdash Xz : C$ in BCD. Reasoning in BCD, $H - \{z : B\} \vdash \backslash z(Xz) : B \to C$. Now by hypothesis $FV(X)$ is contained in $dom(F)$, so by weakening, $F \vdash \backslash z(Xz) : B \to C$. Hence by subject reduction for eta ([2] page 621)

$$F \vdash X : B \to C.$$

Conversely,suppose that $F \vdash X : A$. Let $G]F$ and $G \models U : B$. By induction hypothesis $G \vdash U : B$ in BCD Thus by induction hypothesis $G \models (XU) : C$. Hence

$$F \models X : B \to C.$$ ◀

▶ **Corollary 8.** *Generalized monotonicity holds and we have a Beth model.*

From the lemma we get the completeness theorem.

▶ **Theorem 9.** *Let M be closed. Then $BCD \vdash M : A$ iff for every Beth model (O, E) and generic $W, M : R(A)$.*

Proof. $\Rightarrow$. This is the soundness proposition. $\Leftarrow$. Consider the Beth model defined by the conditions above. By proposition 2 $0 \models M : A$. Hence by the lemma $\vdash M : A$ in BCD. ◀

References

1 Barendregt, The Lambda Calculus, North Holland (1981).

2 Barendregt, Dekkers and Statman, Lambda Calculus with Types, Cambridge University Press (2013).

3 Coppo and Dezani, A new type assignment for lambda terms, Archiv fur Math. Logik, 19, 139-156 (1978).

4 van Dalen, Intuitionistic Logic in The Blackwell Guide to Philosophical Logic, Gobble ed Blackwell (2001).

5 Plotkin, Lambda definability in the full type hierarchy in Essays to H.B. Curry, Hindley and Seldin eds, Academic Press, 363-373 (1980).

6 Pottinger, A type assignment for the strongly normalizable lambda terms in Essays to H.B. Curry, Hindley and Seldin eds, Academic Press, 561-577 (1980).

7 Statman, Logical relations and the typed lambda calculus, Information and Control, 165, 2/3, 85-97 (1985)

8 Tait, Constructive reasoning in Studies in Logic and the Foundations of Mathematics, 52, Van Rootselaar and Staal eds, Elsevier, 185-199 (1968).

A Appendix

(1) terms and types
variables $x, y, z, \ldots$ are terms
if X and Y are terms then so are (XY) and $\backslash xX$
atoms $a, b, c, \ldots$ are types
if A and B are types then so are $A/\backslash B$ and $A \to B$

(2) (quasi) order on types
A less than or equal A
$A/\backslash B$ less than or equal A
$A/\backslash B$ less than or equal B
$(A \to B)/\backslash(A \to C)$ less than or equal $A \to (B/\backslash C)$
if C less than or equal A and C less than or equal B
then C less than or equal $A/\backslash B$
if C less than or equal B and B less than or equal A
then C less than or equal A
if A less than or equal C and D less than or equal B
then $C \to D$ less than or equal $A \to B$
A equals B if A less than or equal B and
B less than or equal A

(3) axioms and rules of BCD
$F \vdash x : A$ if $(x : A)$ belongs to F
if $F, x : A \vdash X : B$ then $F \vdash \backslash xX : A \to B$
if $F \vdash X : A \to B$ and $F \vdash Y : A$ then $F \vdash (XY) : B$
if $F \vdash X : A$ and $F \vdash X : B$ then $F \vdash X : A/\backslash B$
if $F \vdash X : A$ and A less then or equal B in BCD
then $F \vdash X : B$