LNCS 4583

Simona Ronchi Della Rocca (Ed.)

Typed Lambda Calculi and Applications

8th International Conference, TLCA 2007
Paris, France, June 2007
Proceedings

Lecture Notes in Computer Science 4583

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Simona Ronchi Della Rocca (Ed.)

Typed Lambda Calculi and Applications

8th International Conference, TLCA 2007
Paris, France, June 26-28, 2007
Proceedings

Springer

Volume Editor

Simona Ronchi Della Rocca
Università di Torino
Dipartimento di Informatica
corso Svizzera 185, 10149 Torino, Italy
E-mail: ronchi@di.unito.it

Library of Congress Control Number: 2007929283

CR Subject Classification (1998): F.4.1, F.3, D.1.1, D.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-73227-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-73227-3 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12080779 06/3180 5 4 3 2 1 0

Preface

This volume represents the proceedings of the Eighth International Conference on Typed Lambda Calculi and Applications, TLCA 2007, held in Paris, France during 26–28 June 2007, in conjunction with RTA. It contains the abstracts of the invited talks by Frank Pfenning and Patrick Baillot, plus 25 contributed papers. The contributed papers were selected from a total of 52 submissions. The conference program included an invited talk by Greg Morrisett and a special evening talk by Henk Barendregt, on "Diamond Anniversary of Lambda Calculus." I wish to express my gratitude to the members of the Program Committee and to all the referees for their contribution in preparing a very interesting scientific program. Moreover, I thank, the members of the Organizing Committee for their hard work and the sponsoring institutions.

April 2007 Simona Ronchi Della Rocca

Organization

Program Committee

Chantal Berline (CNRS, France)
Peter Dybjer (Chalmers, Sweden)
Healfdene Goguen (Google, USA)
Robert Harper (Carnegie Mellon University, USA)
Olivier Laurent (CNRS, France)
Simone Martini (University of Bologna, Italy)
Simona Ronchi Della Rocca (University of Torino, Italy), Chair
Peter Selinger (Dalhousie University, Canada)
Paula Severi (University of Leicester, UK)
Kazushige Terui (University of Sokendai, Japan)
Pawel Urzyczyn (University of Warsaw, Poland)

Steering Committee

Samson Abramsky (University of Oxford, UK)
Henk Barendregt (Katholieke Universitet Nijmegen, The Netherlands)
Mariangiola Dezani-Ciancaglini (University of Torino, Italy), Chair
Roger Hindley (University of Swansea, UK)
Martin Hofmann (University of Munich, Germany)
Pawel Urzyczyn (University of Warsaw, Poland)

Organizing Committee

Antonio Bucciarelli (PPS, University of Paris 7)
Vincent Padovani (PPS, University of Paris 7)
Ralf Treinen (LSV, ENS de Cachan, CNRS, and INRIA Futurs)
Xavier Urbain (Cedric, IIE-CNAM)

Referees

Samson Abramsky
Robin Adams
Thorsten Altenkirch
Zena Ariola
Patrick Baillot
Vincent Balat
Mutsunori Banbara
Henk Barendregt
Marcin Benke
Stefano Berardi
Martin Berger
Frederic Blanqui
Viviana Bono
Bjrn Bringert
Claudio Sacerdoti Coen
Mario Coppo
Paolo Coppola
Roy Crole
Ugo Dal Lago
Ferruccio Damiani
Mariangiola Dezani

Manfred Droste
Ken-etsu Fujita
Lorenzo Tortora de Falco
Maribel Fernandez
Jean-Christophe Filliatre
Melvin Fitting
Luca Fossati
Murdoch Gabbay
Maurizio Gabbrielli
Marco Gaboardi
Simon Gay
Silvia Ghilezan
Pietro Di Gianantonio
Eduardo Gimenez
Matthew Goldfield
Stefano Guerrini
Masahiro Hamano
Makoto Hamana
Russ Harmer
Michael Hedberg
Hugo Herbelin
Martin Hyland
Pierre Hyvernat
Patricia Johann
Thierry Joly
Steffen Jost
Shin-ya Katsumata
Kentaro Kikuchi
Daisuke Kimura
Naoki Kobayashi
Satoshi Kobayashi
Jean-Louis Krivine
Alexander Kurz
James Leifer
Stephane Lengrand
Pierre Letouzey
Paul Levy
Dan Licata
John Longley
William Lovas
Zhaohui Luo
Christoph Lth
Andrea Masini
Ralph Matthes
Damiano Mazza
Conor McBride
Guy McCusker
Dale Miller
Alexandre Miquel
Georg Moser
Jean-Yves Moyen
Andrzej Murawski
Koji Nakazawa
Ulf Norell
Vincent van Oomstrom
Luca Paolini
Brigitte Pientka
Randy Pollack
John Power
Christophe Raffalli
Jason Reed
Luca Roversi
Paul Rozire
Davide Sangiorgi
Sam Sanjabi
Aleksy Schubert
Carsten Schuermann
Ian Stark
Sam Staton
Morten Heine Srensen
Makoto Tatsuta
Tarmo Uustalu
Lionel Vaux
Betti Venneri
Fer-Jan de Vries
David Walker
Pawel Waszkiewicz
Hongwei Xi
Yoriyuki Yamagata
Noam Zeilberger

Sponsoring Institutions

Conservatoire des Arts et Métiers (CNAM), Centre National de la Recherche Scientifique (CNRS), École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIEE), GDR Informatique Mathématique, Institut National de Recherche en Informatique et Automatique (INRIA) unit Futurs, Région Île de France.

Table of Contents

On a Logical Foundation for Explicit Substitutions 1
Frank Pfenning

From Proof-Nets to Linear Logic Type Systems for Polynomial Time Computing 2
Patrick Baillot

Strong Normalization and Equi-(Co)Inductive Types 8
Andreas Abel

Semantics for Intuitionistic Arithmetic Based on Tarski Games with Retractable Moves 23
Stefano Berardi

The Safe Lambda Calculus 39
William Blum and C.-H. Luke Ong

Intuitionistic Refinement Calculus 54
Sylvain Boulmé

Computation by Prophecy 70
Ana Bove and Venanzio Capretta

An Arithmetical Proof of the Strong Normalization for the λ-Calculus with Recursive Equations on Types 84
René David and Karim Nour

Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo 102
Denis Cousineau and Gilles Dowek

Completing Herbelin's Programme 118
José Espírito Santo

Continuation-Passing Style and Strong Normalisation for Intuitionistic Sequent Calculi 133
José Espírito Santo, Ralph Matthes, and Luís Pinto

Ludics is a Model for the Finitary Linear Pi-Calculus 148
Claudia Faggian and Mauro Piccolo

Differential Structure in Models of Multiplicative Biadditive Intuitionistic Linear Logic 163
Marcelo P. Fiore

The Omega Rule is $\mathbf{\Pi^1_1}$-Complete in the $\lambda\beta$-Calculus 178
Benedetto Intrigila and Richard Statman

Weakly Distributive Domains 194
Ying Jiang and Guo-Qiang Zhang

Initial Algebra Semantics Is Enough! 207
Patricia Johann and Neil Ghani

A Substructural Type System for Delimited Continuations 223
Oleg Kiselyov and Chung-chieh Shan

The Inhabitation Problem for Rank Two Intersection Types 240
Dariusz Kuśmierek

Extensional Rewriting with Sums 255
Sam Lindley

Higher-Order Logic Programming Languages with Constraints: A Semantics 272
James Lipton and Susana Nieva

Predicative Analysis of Feasibility and Diagonalization 290
Jean-Yves Marion

Edifices and Full Abstraction for the Symmetric Interaction Combinators 305
Damiano Mazza

Two Session Typing Systems for Higher-Order Mobile Processes 321
Dimitris Mostrous and Nobuko Yoshida

An Isomorphism Between Cut-Elimination Procedure and Proof Reduction 336
Koji Nakazawa

Polynomial Size Analysis of First-Order Functions 351
Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen

Simple Saturated Sets for Disjunction and Second-Order Existential Quantification 366
Makoto Tatsuta

Convolution $\bar{\lambda}\mu$-Calculus 381
Lionel Vaux

Author Index 397

On a Logical Foundation for Explicit Substitutions

Frank Pfenning

Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu
http://www.cs.cmu.edu/~fp

Traditionally, calculi of explicit substitution [1] have been conceived as an implementation technique for β-reduction and studied with the tools of rewriting theory. This computational view has been extremely fruitful (see [2] for a recent survey) and raises the question if there may also be a more abstract underlying logical foundation.

Some forms of explicit substitution have been related to cut in the intuitionistic sequent calculus [3]. While making a connection to logic, the interpretation of explicit substitutions remains primarily computational since they do not have a reflection at the level of propositions, only at the level of proofs.

In recent joint work [4], we have shown how explicit substitutions naturally arise in the study of intuitionistic modal logic. Their logical meaning is embodied by a contextual modality which captures all assumptions a proof of a proposition may rely on. Explicit substitutions mediate between such contexts and therefore, intuitively, between worlds in a Kripke-style interpretation of modal logic.

In this talk we review this basic observation about the logical origin of explicit substitutions and generalize it to a multi-level modal logic. Returning to the computational meaning, we see that explicit substitutions are the key to a λ-calculus where variables, meta-variables, meta-meta-variables, etc. can be unified without the usual paradoxes such as lack of α-conversion. We conclude with some speculation on potential applications of this calculus in logical frameworks or proof assistants.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming 1(4), 375–416 (1991)
2. Kesner, D.: The theory of calculi with explicit substitutions revisited. Unpublished manuscript (October (2006)
3. Herbelin, H.: A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 61–75. Springer, Heidelberg (1995)
4. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. Transactions on Computational Logic (To appear, 2007)

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, p. 1, 2007.
© Springer-Verlag Berlin Heidelberg 2007

From Proof-Nets to Linear Logic Type Systems for Polynomial Time Computing

Patrick Baillot*

LIPN, CNRS & Université Paris 13,
99 av. J-B. Clément, 93430 Villetaneuse, France
patrick.baillot@lipn.univ-paris13.fr

Abstract. In this presentation we give an overview of Dual Light Affine Logic (DLAL), a polymorphic type system for lambda calculus ensuring that typable lambda terms are executable in polynomial time. We stress the importance of proof-nets from Light linear logic for the design of this type system and for a result establishing that typable lambda-terms can be evaluated efficiently with optimal reduction. We also discuss the issue of DLAL type inference, which can be performed in polynomial time for system F terms. These results have been obtained in collaborations with Terui [1], Atassi and Terui [2], and Coppola and Dal Lago [3].

Implicit Computational Complexity (ICC) is concerned with the study of computation with bounded time or memory. It has emerged from early works such as those of Leivant [4], Bellantoni and Cook [5], Jones [6], Leivant and Marion [7]. Lambda calculus and functional programming play a key role in this field. A particular interest is attached to *feasible computing*, by which we mean computing in polynomial time in the size of the input (PTIME).

Instead of seeing execution time simply as a result of observation, the driving motivation here is to shed some light on the nature of feasible computing, by unveiling some invariants or some programming methodologies which can in a modular way ensure that the resulting programs remain in the feasible class. Challenging goals in this area are to obtain manageable programming languages for feasible computing or to delineate some constructive proof systems in which extracted programs would be certified to be of polynomial time complexity.

An important issue is that of *intensional expressivity*: even if all polynomial time *functions* are representable, not all ICC systems have the same expressive power when it comes to implement concrete *algorithms*.

Linear Logic. Here we focus our attention on the *linear logic* approach to ICC, which is based on the *proofs-as-programs* correspondence. By giving a logical status to the operation of duplication linear logic provides a fine-grained way to study and control the dynamics of evaluation. Indeed various choices of rules for the modalities (*exponential* connectives), regulating duplication, result in variants of linear logic with different bounds on proof normalization (cut elimination).

* Partially supported by project NO-CoST (ANR, JC05_43380).

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 2–7, 2007.
© Springer-Verlag Berlin Heidelberg 2007

This approach lead in particular to *Bounded* (BLL) [8], *Soft* (SLL) [9] and *Light* (LLL) linear logics [10] (or its variant Light Affine Logic LAL [11]), which correspond to PTIME computation. The language used is that of *proof-nets*, a graph syntax for proofs. Using the Curry-Howard correspondence the system LAL for instance gives a calculus for polynomial time computing in the ICC approach (see *e.g.* [12]).

Types. We will describe a type system for lambda calculus obtained from Light linear logic, called *Dual Light Affine Logic* (DLAL). If a term acting say, on binary lists, is well typed in DLAL, then it runs in polynomial time. The advantage here with respect to LAL is that the source language, lambda calculus, is a standard one and the discipline for ensuring polynomial time bounds is managed by the type system. A nice aspect also w.r.t. other type-based ICC systems such as *e.g.* [13] is that the lambda calculus does not contain constants and recursor, but instead the data types and the corresponding iteration schemes are definable, as in system F. Indeed, DLAL, as other type systems from Light logics can be seen as a refinement of system F types.

Proof-Nets and Boxes. Essentially a DLAL type derivation corresponds to an LAL proof-net. The main extra information with respect to the underlying system F term lies in the presence of *boxes*, corresponding to the use of modalities. Boxes are a standard notion in the proof-net technology. They are usually needed to perform proof-net normalization. We emphasize here a double aspect of boxes in Light logics:

- from a methodological point of view: boxes are a key feature in Light logics (and thus in the design of Light type systems) because they allow to enforce some invariants which guarantee the complexity bound;
- from an operational point of view: boxes can somehow be forgotten for evaluation of (typable) terms; this can be achieved either by using the DLAL type system and β reduction, or by using *optimal reduction*, that is to say graph rewriting.

We will illustrate this double aspect of boxes and the interplay between lambda terms and proof-nets (Fig 1) by discussing the DLAL type assignment system, optimal reduction of typable terms and finally DLAL type inference.

1 Type System DLAL and Proof-Nets

Dual Light Affine Logic (DLAL) is a type system derived from Light Affine Logic [1]. Its type language is defined by:

$$A, B ::= \alpha \mid A \multimap B \mid A \Rightarrow B \mid \S A \mid \forall \alpha . A \ ,$$

where $\multimap$ (resp. $\Rightarrow$) is a linear (resp. non-linear) arrow connective. An integer called *depth* is attached to each derivation. The main property of DLAL is:

Theorem 1. *If a lambda term t is typable in DLAL with a type derivation of depth d, then any β reduction sequence of t has length bounded by $O(|t|^{2^d})$.*

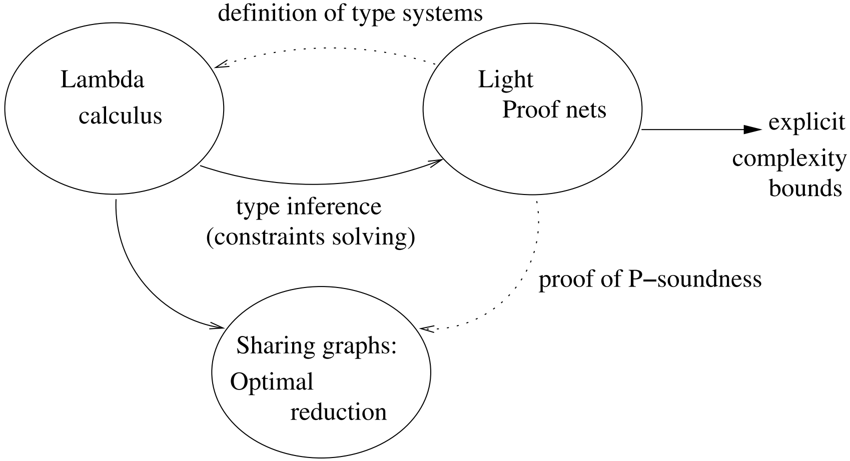


Fig. 1. Lambda terms and Light proof-nets

DLAL can be translated in LAL using $(A \Rightarrow B)^* = !A^* \multimap B^*$. Consequently any DLAL type derivation corresponds to an LAL proof-net. These proof-nets have two kinds of boxes: !-boxes, which are duplicable, and §-boxes which are not. The depth of a DLAL derivation corresponds to the maximal nesting of boxes in the corresponding proof-net.

All polynomial time *functions* can be represented in DLAL. However its intensional expressivity, as that of other Light logic systems, is actually quite weak (some simple PTIME system F terms are non typable). On the other hand testing if a term is typable can be done efficiently, as we will see in Section 3. This illustrates a kind of trade-off in ICC languages between the algorithmic expressivity on the one hand and the easiness of verifying if a program belongs to the language on the other.

2 Evaluating Without Boxes: Optimal Reduction

Boxes are an important information in proof-nets and are needed for their normalization (see *e.g.* [14]). However it turns out that DLAL *typable* lambda terms can be evaluated with a local graph-rewriting procedure (without the boxes): Lamping's abstract algorithm for optimal reduction. The advantage of this abstract algorithm with respect to Lamping's general algorithm [15,16] is that no bookkeeping is needed for managing the indices, which makes it particularly simple.

The fact that Lamping's algorithm is correct for LAL or EAL (Elementary Affine Logic) typed terms was actually a main motivation for the study of these systems for quite a while [17]. However a recent result [3] is that Lamping's abstract algorithm applied to DLAL or LAL typable terms is of similar complexity as proof-net reduction, that is to say polynomial in the size, if the

depth is fixed. This has been achieved by using as tools proof-nets and *context semantics* ([18]).

3 Recovering Boxes: Type Inference by Linear Programming

We stressed that typable lambda terms can be evaluated efficiently *without* the typing information. So, why would we need types anyway then? Actually the typing can be important: (i) to get an explicit complexity bound for the reduction (Theorem 1); (ii) for modularity: to be able to combine terms together and stay in the typable class.

The specificity of DLAL type inference is the inference of modalities, so we consider as input a system F typed lambda term. Concretely type inference in DLAL for this term can then be performed by decorating the syntax tree of the term with boxes and the types with modalities, that is to say by *constructing a proof-net*. An algorithm searching for such decorations by using constraints generation and solving has been given in [2] (Fig. 2), after works on related systems, *e.g.* in [19,20,21].

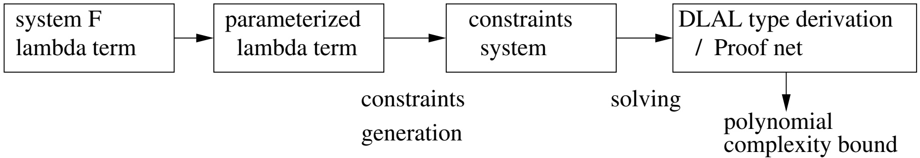


Fig. 2. DLAL type inference

The algorithm relies on two ingredients:

- analysing where non-linear application (and hence !-box) is needed: this is expressed by *boolean constraints*;
- searching for a suitable distribution of boxes (! or §-boxes): the key point here is to assign integer parameters for the number of (box) *doors* on each edge, and to search for instantiations of these parameters for which valid boxes can be reconstructed.

In this way a set of constraints on boolean ($\mathbf{b_i}$) and integer ($\mathbf{n_i}$) parameters is associated to a term, expressing its typability. It contains:

boolean constraints,	*e.g.*	$\mathbf{b_1} = \mathbf{b_2}$, $\mathbf{b_1} = \mathbf{0}$
linear constraints,	*e.g.*	$\sum_i \mathbf{n_i} \leq 0$
mixed boolean/linear constraints,	*e.g.*	$\mathbf{b_1} = 1 \Rightarrow \sum_i \mathbf{n_i} \leq 0$.

A resolution method for solving the constraints system is given by the following two-step procedure:

1. boolean phase: search for the *minimal* solution to the boolean constraints. This corresponds to doing a linearity analysis (determine which applications are linear and which ones are non-linear).
2. linear programming phase: once the constraints system is instantiated with the boolean solution, we get a linear constraints system, that can be solved with linear programming methods. This corresponds to finding a concrete distribution of boxes satisfying all the conditions.

This resolution procedure is correct and complete and it can be performed in polynomial time w.r.t. the size of the original system F term. Any solution of the constraints system gives a valid DLAL type derivation.

References

1. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: Proceedings of LICS'04, IEEE Computer Society, pp. 266–275. IEEE Computer Society Press, Los Alamitos (2004)
2. Atassi, V., Baillot, P., Terui, K.: Verification of Ptime reducibility for system F terms via Dual Light Affine Logic. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 150–166. Springer, Heidelberg (2006)
3. Baillot, P., Coppola, P., Dal Lago, U.: Light logics and optimal reduction: Completeness and complexity. In: Proceedings of LICS'07, IEEE Computer Society Press, Los Alamitos (to appear, 2007)
4. Leivant, D.: Predicative recurrence and computational complexity I: word recurrence and poly-time. In: Feasible Mathematics II, Birkhauser, pp. 320–343 (1994)
5. Bellantoni, S., Cook, S.: New recursion-theoretic characterization of the polytime functions. Computational Complexity 2, 97–110 (1992)
6. Jones, N.: LOGSPACE and PTIME characterized by programming languages. Theoretical Computer Science 228(1-2), 151–174 (1999)
7. Leivant, D., Marion, J.Y.: Lambda-calculus characterisations of polytime. Fundamenta Informaticae 19, 167–184 (1993)
8. Girard, J.Y., Scedrov, A., Scott, P.: Bounded linear logic: A modular approach to polynomial time computability. Theoretical Computer Science 97, 1–66 (1992)
9. Lafont, Y.: Soft linear logic and polynomial time. Theoretical Computer Science 318(1–2), 163–180 (2004)
10. Girard, J.Y.: Light linear logic. Information and Computation 143, 175–204 (1998)
11. Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM Transactions on Computational Logic 3(1), 1–39 (2002)
12. Terui, K.: Light affine lambda calculus and polynomial time strong normalization. Archive for Mathematical Logic 46(3-4), 253–280 (2007)
13. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. Information and Computation 183(1), 57–85 (2003)
14. Mazza, D.: Linear logic and polynomial time. Mathematical Structures in Computer Science 16(6), 947–988 (2006)
15. Lamping, J.: An algorithm for optimal lambda calculus reduction. In: Proceedings of POPL'90, pp. 16–30. ACM, New York (1990)

16. Gonthier, G., Abadi, M., Lévy, J.J.: The geometry of optimal lambda reduction. In: Proceedings of POPL'92, pp. 15–26. ACM, New York (1992)
17. Asperti, A.: Light affine logic. In: Proceedings of LICS'98, IEEE Computer Society, Los Alamitos (1998)
18. Dal Lago, U.: Context semantics, linear logic and computational complexity. In: Proceedings of LICS'06, pp. 169–178. IEEE Computer Society Press, Los Alamitos (2006)
19. Coppola, P., Martini, S.: Optimizing optimal reduction: A type inference algorithm for elementary affine logic. ACM Trans. Comput. Log 7(2), 219–260 (2006)
20. Coppola, P., Ronchi Della Rocca, S.: Principal typing for lambda calculus in elementary affine logic. Fundam. Inform. 65(1-2), 87–112 (2005)
21. Coppola, P., Dal Lago, U., Ronchi Della Rocca, S.: Elementary affine logic and the call-by-value lambda calculus. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 131–145. Springer, Heidelberg (2005)

Strong Normalization and Equi-(Co)Inductive Types

Andreas Abel*

Department of Computer Science, University of Munich
Oettingenstr.67, D-80538 München, Germany
abel@tcs.ifi.lmu.de

Abstract. A type system for the lambda-calculus enriched with recursive and corecursive functions over equi-inductive and -coinductive types is presented in which all well-typed programs are strongly normalizing. The choice of equi-inductive types, instead of the more common iso-inductive types, influences both reduction rules and the strong normalization proof. By embedding iso- into equi-types, the latter ones are recognized as more fundamental. A model based on orthogonality is constructed where a semantical type corresponds to a set of observations, and soundness of the type system is proven.

1 Introduction

Theorem provers based on the Curry-Howard-Isomorphism, such as Agda, Coq, Epigram, or LEGO are built on dependent types and use inductive and coinductive types to formalize data structures, object languages, logics, judgments, derivations, etc. Proofs by induction or coinduction are represented as recursive or corecursive programs, where only total programs represent valid proofs. As a consequence, only total programs, which are defined on all well-typed inputs, are accepted, and totality is checked by some static analysis (in case of Coq), or ensured by construction (in case of Epigram), or simply assumed (in case of LEGO and the current version of Agda).

Hughes, Pareto, and Sabry [16] have put forth a totality check based on sized types, such that each well-typed program is already total. Designed originally for embedded systems it has become attractive for theorem provers because of several advantages: First of all, its soundness can be proven by an interpretation of types as sets of total programs, as noted by Giménez [13]. Since soundness proofs for dependent types are delicate, the clarity that sized types offer should not be underestimated. Secondly, checking termination through types integrates the features of advanced type systems, most notably higher-order functions, higher-order types, polymorphism, and subtyping, into the termination check

* Research supported by the coordination action *TYPES* (510996) and thematic network *Applied Semantics II* (IST-2001-38957) of the European Union and the project *Cover* of the Swedish Foundation of Strategic Research (SSF).

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 8–22, 2007.
© Springer-Verlag Berlin Heidelberg 2007

without extra effort. Some advanced examples of what one can do with type-based termination, but not with syntactical, "term-based" termination checks, are given in other works of the author [4,3,2]. And last, type-based termination is just (a) sized inductive and coinductive types with subtyping induced by the sizes plus (b) typing rules for recursive and corecursive functions which ensure well-foundedness by checking that sizes are decreased in recursive instances. Due to this conceptual simplicity it is planned to replace the current term-based termination check in Coq by a type-based one; in recent works, sized types have been integrated with dependent types [9,10].

Dependently typed languages, such as the languages of the theorem provers listed above, need to compare terms for equality during type-checking. In the presence of recursion, this equality test is necessarily incomplete.[1] A common heuristics is to normalize the terms and compare them for syntactic equality. In general, these terms are open, i. e., have free variables, and normalization takes place in all term positions, also under binders. This complicates matters considerably: Although a function is total in a semantical sense and terminates on all closed terms under lazy evaluation, it will probably diverge on open terms under full evaluation.[2] Hence, unfolding recursion has to be sensibly restricted during normalization. In related work [13,8], inductive types are given by constructors, and a recursive function is only unfolded if its "recursive" argument, i. e., the argument that gets smaller in recursive calls, is a constructor.

We take a more foundational approach and consider a language, $\mathsf{F}_\omega^{\widehat{}}$, without constructors. Programs of $\mathsf{F}_\omega^{\widehat{}}$ are just λ-terms over constants fix_n^μ and fix_n^ν which introduce recursive and corecursive functions with n leading "parametric", i. e., non-recursive arguments. A recursive function $\mathsf{fix}_n^\mu\, s\, t_1\, \dots\, t_n\, v$ with body s, parametric arguments t_i and recursive argument v is unfolded if v is a value, i. e., a λ-abstraction or an under-applied, meaning not fully applied, (co)recursive function. A corecursive function $\mathsf{fix}_n^\nu\, s\, t_1 \dots t_n$ is unfolded if it is in evaluation position, e. g., applied to some term. In this article, we prove that this strategy is strongly normalizing for programs which are accepted by the sized type system.

For now, $\mathsf{F}_\omega^{\widehat{}}$ does not feature dependent types—they are not essential to studying the operational semantics, but cause considerable complications in the normalization proof. However, $\mathsf{F}_\omega^{\widehat{}}$ has arbitrary-rank polymorphism, thus, elementary data types like unit type, product type and disjoint sum can be defined by the usual Church-encodings. Inductive types are not given by constructors; instead we have least fixed-point types $\mu^a F$ which denote the ath iteration of type constructor F. Semantically $\mu^0 F$ is empty, $\mu^{a+1} F = F\,(\mu^a F)$, and for limit ordinals λ, $\mu^\lambda F$ denotes the union of all $\mu^a F$ for $a < \lambda$. It may help to think of the size index a as an strict upper bound for the height of the inhabitants of $\mu^a F$, viewed as trees. Dually, we have sized coinductive types $\nu^a F$, and a denotes

---

[1] And one would not expect that this test succeeds for the equation $f\,(\mathsf{fix}\,(g \circ f)) = \mathsf{fix}\,(f \circ g)$ given arbitrary (well-typed programs) f and g.

[2] Consider the recursive zero-function $\mathsf{zero}\, x = \mathsf{match}\ x\ \mathsf{with}\ 0 \mapsto 0 \mid y + 1 \mapsto \mathsf{zero}\, y$. If applying zero to a variable triggers unfolding of recursion, normalization of zero will diverge.

the minimum number of elementary destructions one can safely perform on an element of $\nu^a F$, which is, in case of streams, the minimum number of elements one can read off this stream.

With sum and product types, common inductive types can be expressed as $\mu^a F$ for a suitable F; their constructors are then simply λ-terms. However, there is a design choice: *equi-inductive* types have $\mu^{a+1} F = F\,(\mu^a F)$ as a type equation in the system; *iso-inductive* types stipulate that $\mu^{a+1} F$ and $F\,(\mu^a F)$ are only isomorphic, witnessed by a folding operation $\mathsf{in} : F\,(\mu^a F) \to \mu^{a+1} F$ and an unfolding operation $\mathsf{out} : \mu^{a+1} F \to F\,(\mu^a F)$. The iso-approach has been taken in previous work by the author [3] and seems to be more common [12,6,19,7], since it has a simpler theory. We go the foundational path and choose the "equi" flavor, which has consequences for the operational semantics and the normalization proof: since there are less syntactical "clutches" to hold on, more structure has to be built in the semantics.

Overview. In Section 2 we present System $\mathsf{F}_{\widehat{\omega}}$ with typing rules which only accept strongly normalizing functions. In Section 3 we motivate the reduction rules of $\mathsf{F}_{\widehat{\omega}}$ which are affected by *equi*-(co)inductive types. By embedding iso- into equi-inductive types in Section 4, we justify that equi-types are more fundamental than iso-types. We then proceed to develop a semantical notion of type, based on strong normalization and orthogonality (Section 5). Finally, we sketch the soundness proof for $\mathsf{F}_{\widehat{\omega}}$ in Section 6 and discuss some related work in Section 7.

2 System $\mathsf{F}_{\widehat{\omega}}$

Like in System F^ω, expressions of $\mathsf{F}_{\widehat{\omega}}$ are separated into three levels: kinds, type constructors, and terms (objects). Figure 1 summarizes the first two levels. In contrast to the standard presentation, we have a second base kind, ord, whose inhabitants are syntactic ordinals. Canonical ordinals are either $\mathsf{s}^n \imath = \mathsf{s}\,(\mathsf{s} \ldots (\mathsf{s}\,\imath))$ (notation: $\imath + n$), the nth successor of an ordinal variable ι, or ∞, the closure ordinal of all inductive and coinductive types of $\mathsf{F}_{\widehat{\omega}}$. In spite of the economic syntax, expressions of kind ord, which we will refer to as *size expressions*, semantically denote ordinals up to the ωth uncountable (see Sect. 5.3). We use the metavariable a to range over size expressions and the metavariable $\imath$ to range over size variables. The metavariable X ranges over type constructor variables, which includes size variables.

Another feature of $\mathsf{F}_{\widehat{\omega}}$ are polarized kinds [27,5,1]. Function kinds are labeled with a polarity p that classifies the inhabiting type constructors as covariant ($p = +$), contravariant ($p = -$), or non-variant ($p = \circ$), the last meaning mixed or unknown variance. Inductive types are introduced using the type constructor constant

$$\mu_\kappa : \mathsf{ord} \xrightarrow{+} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa.$$

We write $\mu_\kappa\, a\, F$ usually as $\mu^a_\kappa F$ and drop index kind κ if clear from the context. The underlying type constructor $F : \kappa \xrightarrow{+} \kappa$ must be covariant—otherwise divergence is possible even without recursion (Mendler [21])—and κ must be a

Syntactic categories.

$$
\begin{array}{lll}
p & ::= + \mid - \mid \circ & \text{polarity} \\
\kappa & ::= * \mid \mathsf{ord} \mid p\kappa \to \kappa' & \text{kind} \\
\kappa_* & ::= * \mid p\kappa_* \to \kappa'_* & \text{pure kind} \\
a, b, A, B, F, G & ::= C \mid X \mid \lambda X\!:\!\kappa.\, F \mid F\, G & \text{(type) constructor} \\
C & ::= \to \mid \forall_\kappa \mid \mu_{\kappa_*} \mid \nu_{\kappa_*} \mid \mathsf{s} \mid \infty & \text{constructor constant} \\
\Delta & ::= \diamond \mid \Delta, X\!:\!p\kappa & \text{polarized context}
\end{array}
$$

The signature Σ assigns kinds to constants ($\kappa \xrightarrow{p} \kappa'$ means $p\kappa \to \kappa'$).

$$
\begin{array}{lll}
\to & : * \xrightarrow{-} * \xrightarrow{+} * & \text{function space} \\
\forall_\kappa & : (\kappa \xrightarrow{\circ} *) \xrightarrow{+} * & \text{quantification} \\
\mu_{\kappa_*} & : \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* & \text{inductive constructors} \\
\nu_{\kappa_*} & : \mathsf{ord} \xrightarrow{-} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* & \text{coinductive constructors} \\
\mathsf{s} & : \mathsf{ord} \xrightarrow{+} \mathsf{ord} & \text{successor of ordinal} \\
\infty & : \mathsf{ord} & \text{infinity ordinal}
\end{array}
$$

Notation.

$$
\begin{array}{rll}
\nabla & \text{for} & \mu \text{ or } \nu \\
\nabla^a & \text{for} & \nabla a \\
\forall X\!:\!\kappa.\, A & \text{for} & \forall_\kappa(\lambda X\!:\!\kappa.\, A) \\
\forall X A & \text{for} & \forall X\!:\!\kappa.\, A \\
\lambda X F & \text{for} & \lambda X\!:\!\kappa.\, F
\end{array}
$$

Ordering and composition of polarities.

$$
p \le p \qquad \circ \le p \qquad +p = p \qquad -- = + \qquad \circ p = \circ \qquad pp' = p'p
$$

Inverse application of a polarity to a context.

$$
\begin{array}{ll}
p^{-1}\diamond = \diamond & \circ^{-1}(\Delta, X\!:\!\circ\kappa) = (\circ^{-1}\Delta), X\!:\!\circ\kappa \\
+^{-1}\Delta = \Delta & \circ^{-1}(\Delta, X\!:\!+\kappa) = \circ^{-1}\Delta \\
-^{-1}(\Delta, X\!:\!p\kappa) = (-^{-1}\Delta), X\!:\!(-p)\kappa & \circ^{-1}(\Delta, X\!:\!-\kappa) = \circ^{-1}\Delta
\end{array}
$$

Kinding $\Delta \vdash F : \kappa$.

$$
\textsc{kind-c}\ \frac{C\!:\!\kappa \in \Sigma}{\Delta \vdash C : \kappa} \qquad \textsc{kind-var}\ \frac{X\!:\!p\kappa \in \Delta \qquad p \le +}{\Delta \vdash X : \kappa}
$$

$$
\textsc{kind-abs}\ \frac{\Delta, X\!:\!p\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X\!:\!\kappa.\, F : p\kappa \to \kappa'} \qquad \textsc{kind-app}\ \frac{\Delta \vdash F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\, G : \kappa'}
$$

Fig. 1. $\mathsf{F}_{\widehat{\omega}}$: Kinds and constructors

pure kind, i. e., not mention ord. The last condition seems necessary to define a uniform closure ordinal for all inductive types, meaning an ordinal ∞ such that $\mu^\infty F = F\,(\mu^\infty F)$. Inductive types are covariant in their size argument; the subtyping chain $\mu^a F \leq \mu^{a+1} F \leq \mu^{a+2} F \leq \cdots \leq \mu^\infty F$ holds. Dually, coinductive types, which are introduced by the constant

$$\nu_\kappa : \mathsf{ord} \xrightarrow{-} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa,$$

are contravariant, and we have the chain $\nu^\infty F \leq \cdots \leq \nu^{a+2} F \leq \nu^{a+1} F \leq \nu^a F$. Type constructors are identified modulo $\beta\eta$ and the laws $\mathsf{s}\,\infty = \infty$ and $\nabla^{a+1} F = F\,(\nabla^a F)$, where ∇ is a placeholder for μ or ν, here and in the following. Type constructor equality is kinded and given by the judgement $\Delta \vdash F = F' : \kappa$ for a kinding context Δ. Exept β and η, we have the axioms $\Delta \vdash \mathsf{s}\,\infty = \infty : \mathsf{ord}$ and $\Delta \vdash \nabla_\kappa^{\mathsf{s}a} F = F\,(\nabla_\kappa^a F) : \kappa$ for wellkinded F and $a : \mathsf{ord}$. Similarly, we have kinded higher-order subtyping $\Delta \vdash F \leq F' : \kappa$ induced by $\Delta \vdash a \leq \mathsf{s}\,a : \mathsf{ord}$ and $\Delta \vdash a \leq \infty : \mathsf{ord}$. Due to space restrictions, the rules have to be omitted, please find them in the author's thesis [2, Sect. 2.2].

Figure 2 displays terms and typing rules of $\mathsf{F}_\omega^{\frown}$. Besides λ-terms, there are constants fix_n^μ to introduce functions that are recursive in the $n+1$st argument, and constants fix_n^ν to introduce corecursive functions with n arguments. The type $A(\imath)$ of a recursive function introduced by fix_n^μ must be of the shape

$$\forall \boldsymbol{X}.\, A_1 \to \cdots \to A_n \to \mu^\imath F^0\, \boldsymbol{G}^0 \to \cdots \to \mu^\imath F^m\, \boldsymbol{G}^m \to C,$$

where the A_i are contravariant in the size index $\imath$, C is covariant, and the F^j and $\boldsymbol{G}^j$ do not mention $\imath$. This criterion is written as $A\ \mathsf{fix}_n^\mu\text{-}\mathsf{adm}$. (More precisely, a function of this type is simultaneously recursive in the arguments $n+1$ to $n+m+1$, but we are only interested in the first recursive argument.) Note that if the variance conditions were ignored, non-terminating functions would be accepted [3]. The type of a corecursive function $\mathsf{fix}_n^\nu\, s$ with n arguments has to be of the form

$$\forall \boldsymbol{X}.\, A_1 \to \cdots \to A_n \to \nu^\imath F\, \boldsymbol{G}$$

where the A_i are again contravariant in $\imath$ and F and $\boldsymbol{G}$ do not mention $\imath$ (criterion $A\ \mathsf{fix}_n^\nu\text{-}\mathsf{adm}$).

Basic data types like unit, product, and sum can be added to the system, but we define them impredicatively (see Figure 2) since minimality of the system is a stronger concern in this work than efficiency. Some examples for sized types are:

$$\begin{array}{llll}
\mathsf{Nat} : & \mathsf{ord} \xrightarrow{+} * & \mathsf{Tree} & : \ \mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{Nat} := & \lambda\imath.\ \mu^\imath \lambda X.\, 1 + X & \mathsf{Tree} & := \lambda\imath\lambda B\lambda A.\, \mu^\imath\lambda X.\, 1 + A \times (B \to X) \\[1ex]
\mathsf{List} : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * & \mathsf{Stream} & : \ \mathsf{ord} \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{List} := & \lambda\imath\lambda A.\ \mu^\imath\lambda X.\, 1 + A \times X & \mathsf{Stream} & := \lambda\imath\lambda A.\, \nu^\imath \lambda X.\, A \times X
\end{array}$$

A rich collection of examples is provided in the author's thesis [2, Sect. 3.2].

Syntactic categories.

$\mathsf{Var} \ni x$		variable
$\mathsf{Tm} \ni r, s, t$	$::= x \mid \lambda x t \mid r\, s \mid \mathsf{fix}^{\mu}_{n} \mid \mathsf{fix}^{\nu}_{n}$	term ($n \in \mathbb{N}$)
$\mathsf{Val} \ni v$	$::= \lambda x t \mid \mathsf{fix}^{\nabla}_{n} \mid \mathsf{fix}^{\nabla}_{n} s\, \boldsymbol{t}$ (where $\lvert\boldsymbol{t}\rvert \leq n$)	value ($\nabla \in \{\mu, \nu\}$)
$\mathsf{Eframe} \ni e(_)$	$::= _\, s \mid \mathsf{fix}^{\mu}_{n}\ s\ t_{1..n}\ _$	evaluation frame
$\mathsf{Ecxt} \ni E$	$::= \mathsf{Id} \mid E \circ e \quad [\mathsf{Id}(r) = r, (E \circ e)(r) = E(e(r))]$	evaluation context
$\mathsf{Cxt} \ni \Gamma$	$::= \diamond \mid \Gamma, x : A \mid \Gamma, X : p\kappa$	typing context

Well-formed typing contexts.

$$\text{CXT-EMPTY}\ \frac{}{\diamond\ \mathsf{cxt}} \qquad \text{CXT-TYVAR}\ \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X : \circ\kappa\ \mathsf{cxt}} \qquad \text{CXT-VAR}\ \frac{\Gamma\ \mathsf{cxt} \qquad \Gamma \vdash A : *}{\Gamma, x : A\ \mathsf{cxt}}$$

Typing $\Gamma \vdash t : A$.

$$\text{TY-VAR}\ \frac{(x : A) \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash x : A} \qquad \text{TY-ABS}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x t : A \to B}$$

$$\text{TY-APP}\ \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B} \qquad \text{TY-SUB}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash t : B}$$

$$\text{TY-GEN}\ \frac{\Gamma, X : \circ\kappa \vdash t : F\, X}{\Gamma \vdash t : \forall_{\kappa} F} \qquad \text{TY-INST}\ \frac{\Gamma \vdash t\ :\ \forall_{\kappa} F \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash t : F\, G}$$

$$\text{TY-REC}\ \frac{\Gamma \vdash A : \mathsf{ord} \to * \qquad A\ \mathsf{fix}^{\nabla}_{n}\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}^{\nabla}_{n} : (\forall \imath : \mathsf{ord}.\, A\, \imath \to A\, (\imath + 1)) \to A\, a} \quad \nabla \in \{\mu, \nu\}$$

Impredicative definition of unit, product, and sum type.

$$\begin{array}{lll} 1 & := \forall C.\, C \to C & : * \\ \times & := \lambda A \lambda B \forall C.\, (A \to B \to C) \to C & : * \xrightarrow{+} * \xrightarrow{+} * \\ + & := \lambda A \lambda B \forall C.\, (A \to C) \to (B \to C) \to C & : * \xrightarrow{+} * \xrightarrow{+} * \end{array}$$

Reduction $t \longrightarrow t'$: Closure of the following axioms under all term constructors:

$$\begin{array}{lll} (\lambda x t)\, s & \longrightarrow [s/x]t & \\ \mathsf{fix}^{\mu}_{n} s\ t_{1..n}\, v & \longrightarrow s\, (\mathsf{fix}^{\mu}_{n} s)\, t_{1..n}\, v & \text{if } v \neq \mathsf{fix}^{\nu}_{n'} s'\, t_{1..n'} \\ e\, (\mathsf{fix}^{\nu}_{n} s\, t_{1..n}) & \longrightarrow e\, (s\, (\mathsf{fix}^{\nu}_{n} s)\, t_{1..n}) & \text{if } e \neq \mathsf{fix}^{\mu}_{n'} s'\, t_{1..n'}\ _ \end{array}$$

Fig. 2. $\mathsf{F}_{\widehat{\omega}}$: Terms and type assignment

3 Operational Semantics

In this section, the reduction rules for recursive and corecursive functions are developed. It is clear that unrestricted unfolding of fixed points $\mathsf{fix}\, s \longrightarrow s\, (\mathsf{fix}\, s)$

leads immediately to divergence. In the literature on type-based termination with iso-inductive types one finds the sound reduction rule $\mathsf{fix}^{\mu}_{n} s\, t_{1..n}\, (\mathsf{in}\, r) \longrightarrow s\, (\mathsf{fix}^{\mu}_{n} s)\, t_{1..n}\, (\mathsf{in}\, r)$, which requires the recursive argument to be a canonical inhabitant of the inductive type. Since the canonical inhabitants for equi-inductive types can be of any shape, we liberalize this rule to

$$\mathsf{fix}^{\mu}_{n} s\, t_{1..n}\, v \longrightarrow s\, (\mathsf{fix}^{\mu}_{n} s)\, t_{1..n}\, v, \tag{1}$$

where v is a value; in our case a λ-abstraction, or an under-applied (co)recursive function.

Elements of a coinductive type should be *delayed* by default, they should only be evaluated when they are *observed*, or *forced*, i. e., when they are surrounded by a non-empty evaluation context e. A candidate for a reduction rule is

$$e\, (\mathsf{fix}^{\nu}_{n} s\, t_{1..n}) \longrightarrow e\, (s\, (\mathsf{fix}^{\nu}_{n} s)\, t_{1..n}). \tag{2}$$

It is easy to find well-typed diverging terms if *less* than n arguments $t_{1..n}$ are required before the fixed-point can be unfolded.

Evaluation contexts $e(_)$ are either applications $_\, s$ or recursive functions $\mathsf{fix}^{\mu}_{n}\, s\, t_{1..n}\, _$. The second form is necessary because, before reduction (1) can be performed, the recursive argument has to be evaluated, hence, must be in evaluation position. However, we run into problems if a corecursive value is in a recursive evaluation context, e. g., $\mathsf{fix}^{\mu}_{0}\, (\lambda x x)\, (\mathsf{fix}^{\nu}_{0}\, (\lambda z y))$. Such a term can be well-typed[3] if we use types like $\mu\lambda X.\, \nu\lambda Y.\, A$. Depending on which fixed-point we unfold we get completely different behavior: the recursion fix^{μ}_{0} can be unfolded ad infinitum, the term diverges. If we unfold the corecursion fix^{ν}_{0}, we arrive at $\mathsf{fix}^{\mu}_{0}\, (\lambda x x)\, y$, which is blocked. Another bad example is $\mathsf{fix}^{\mu}_{0} s\, (\mathsf{fix}^{\nu}_{0} s)$ with $s = \lambda z \lambda x x$. If we unfold recursion, we arrive at the normal form $\mathsf{fix}^{\nu}_{0} s$. Otherwise, if we first unfold corecursion, we obtain $\mathsf{fix}^{\mu}_{0} s\, (\lambda x x)$ which has normal form $\lambda x x$; the calculus is not locally confluent.

In this article, we restore acceptable behavior in the following way: A corecursive value inside a recursive evaluation context should block reduction, terms like $\mathsf{fix}^{\mu}_{0}\, s\, (\mathsf{fix}^{\nu}_{0}\, s')$ should be considered neutral, like variables. The drawback of this decision is that types like $\nu^{\imath} \lambda X.\, \mathsf{List}^{\jmath}\, X$ (non-wellfounded, but finitely branching trees) are not well-supported by the system: Applying the List-length function to such a tree, like $\mathsf{fix}^{\nu}_{0} \lambda x.\mathsf{singletonList}(x)$, will not reduce. This seems to be a high price to pay for equi-(co)inductive types; in the iso-version, such problems do not arise. However, as we will see in the next section, even with these blocked terms, the equi-version is able to *completely simulate* reduction of the iso-version, so we have not lost anything in comparison with the iso-version, but we can gain something by improving the current reduction strategy in the equi-version.

4 Embedding Iso- into Equi-(Co)Inductive Types

Why are we so interested in equi-inductive types, if they cause us trouble? Because they are the more primitive notion. Strong normalization for iso-inductive

[3] Note that $\mathsf{fix}^{\mu}_{0} \lambda x x : (\mu^{a} \lambda X X) \rightarrow C$ and $\mathsf{fix}^{\nu}_{0} \lambda x x : \nu^{a} \lambda X X$.

types can be directly obtained from the result for equi-inductive types, since there exists a trivial type and reduction preserving embedding. Let Delay_κ be defined by recursion on the pure kind κ as follows:

$$\begin{array}{ll}\mathsf{Delay}_* & (A) := 1 \to A \\ \mathsf{Delay}_{p\kappa \to \kappa'} & (F) := \lambda X\!:\!\kappa.\, \mathsf{Delay}_{\kappa'}(F X)\end{array}$$

Then we can define iso-inductive $\overline{\mu}_\kappa$ and iso-coinductive $\overline{\nu}_\kappa$ types in $\mathsf{F}^{\frown}_\omega$ as follows:

$$\begin{array}{l}\overline{\nabla}_\kappa \quad := \lambda\imath\lambda F\!:\!\kappa.\, \nabla^\imath_\kappa\, \mathsf{Delay}_\kappa(F) \\ \mathsf{in}^\nabla(t) := \lambda z t \qquad \text{where } z \notin \mathsf{FV}(t) \qquad \mathsf{out}^\nabla(r) := r\,()\end{array}$$

Now $\mathsf{in}^\mu(t)$ is a non-corecursive *value* for each term t, and $\mathsf{out}^\nu(_)$ is an applicative *evaluation context*, so we obtain in $\mathsf{F}^{\frown}_\omega$ the reductions typical for iso-types:

$$\begin{array}{ll}\mathsf{fix}^\mu_n s\, t_{1..n}\, (\mathsf{in}^\mu(r)) & \longrightarrow s\,(\mathsf{fix}^\mu_n s)\, t_{1..n}\, (\mathsf{in}^\mu(r)) \\ \mathsf{out}^\nu(\mathsf{fix}^\nu_n s\, t_{1..n}) & \longrightarrow \mathsf{out}^\nu(s\,(\mathsf{fix}^\nu_n s)\, t_{1..n}).\end{array}$$

The reverse embedding, however, is *not* trivial. Since in the equi-system, folding and unfolding of inductive types can happen deep inside a type, equi-programs are not typable in the iso-system without major modifications. Only *typing derivations* of the equi-system can be translated into typing derivations of the iso-system. Thus, we consider equi-systems as more fundamental than iso-systems.

5 Semantical Types

A *strongly normalizing* term $t \in \mathsf{SN}$ is a term for which each reduction sequence ends in a value or a neutral term. A *neutral* term has either a variable in head position, or, in our case, a blocking fix^μ-fix^ν combination. We define SN inductively, extending previous works [15,28,17] by rules for (co)recursive terms (see Figure 3). Rule SN-ROLL is sound, but not strictly necessary; however, it simplifies the proof of extensionality (see lemma).

Safe reduction $t \rhd t'$ is a variant of weak head reduction which preserves strong normalization in both directions. In particular, SN is closed under safe expansion (rule SN-EXP). This works because we require $s \in \mathsf{SN}$ in rule SHR-β.

Lemma 1 (Properties of SN)

1. *Extensionality: If* $r\,x \in \mathsf{SN}$ *then* $r \in \mathsf{SN}$.
2. *Closure: If* $r \in \mathsf{SN}$ *and* $r \rhd r'$ *or* $r \lhd r'$ *then* $r' \in \mathsf{SN}$.
3. *Strong normalization: If* $r \in \mathsf{SN}$ *then there are no infinite reduction sequences* $r \longrightarrow r_1 \longrightarrow r_2 \longrightarrow \dots$.
4. *Weak head normalization: If* $r \in \mathsf{SN}$ *then* $r \rhd r'$ *and* $r' \in \mathsf{SNe} \cup \mathsf{Val}$.

Alternatively, one can take 3. as the defining property of SN and from this prove 1., 2., and the SN- and SNE-rules in Figure 3. Property 4. holds then also, but

Strongly normalizing evaluation contexts $E \in \mathsf{Scxt}$.

$$\text{SC-ID}\ \frac{}{\mathsf{Id} \in \mathsf{Scxt}} \qquad \text{SC-APP}\ \frac{E \in \mathsf{Scxt} \qquad s \in \mathsf{SN}}{E \circ (_\, s) \in \mathsf{Scxt}} \qquad \text{SC-REC}\ \frac{E \in \mathsf{Scxt} \qquad s, t_{1..n} \in \mathsf{SN}}{E \circ (\mathsf{fix}^{\mu}_{n} s\, t_{1..n}\ _) \in \mathsf{Scxt}}$$

Strongly normalizing neutral terms $r \in \mathsf{SNe}$.

$$\text{SNE-VAR}\ \frac{E \in \mathsf{Scxt}}{E(x) \in \mathsf{SNe}} \qquad \text{SNE-FIX}^{\mu}\text{FIX}^{\nu}\ \frac{E \in \mathsf{Scxt} \qquad s, \boldsymbol{t}, s', \boldsymbol{t}' \in \mathsf{SN}}{E(\mathsf{fix}^{\mu}_{n} s\, t_{1..n}\, (\mathsf{fix}^{\nu}_{n'} s'\, t'_{1..n'})) \in \mathsf{SNe}}$$

Strongly normalizing terms $t \in \mathsf{SN}$.

$$\text{SN-SNE}\ \frac{r \in \mathsf{SNe}}{r \in \mathsf{SN}} \qquad \text{SN-ABS}\ \frac{t \in \mathsf{SN}}{\lambda x t \in \mathsf{SN}} \qquad \text{SN-FIX}\ \frac{\boldsymbol{t} \in \mathsf{SN}}{\mathsf{fix}^{\nabla}_{n} \boldsymbol{t} \in \mathsf{SN}}\ |\boldsymbol{t}| \leq n+1$$

$$\text{SN-EXP}\ \frac{r \rhd r' \qquad r' \in \mathsf{SN}}{r \in \mathsf{SN}} \qquad \text{SN-ROLL}\ \frac{s\, (\mathsf{fix}^{\nu}_{n} s)\, \boldsymbol{t} \in \mathsf{SN}}{\mathsf{fix}^{\nu}_{n} s\, \boldsymbol{t} \in \mathsf{SN}}\ |\boldsymbol{t}| \leq n$$

Safe reduction $t \rhd t'$ (plus reflexivity and transitivity).

$$\text{SHR-}\beta\ \frac{s \in \mathsf{SN}}{E((\lambda x t)\, s) \rhd E([s/x]t)} \qquad \text{SHR-REC}\ \frac{v \neq \mathsf{fix}^{\nu}_{n'} s'\, t'_{1..n'}}{E(\mathsf{fix}^{\mu}_{n} s\, t_{1..n}\, v) \rhd E(s\, (\mathsf{fix}^{\mu}_{n} s)\, t_{1..n}\, v)}$$

$$\text{SHR-COREC}\ \frac{e \neq \mathsf{fix}^{\mu}_{n'} s'\, t'_{1..n'}\ _}{E(e(\mathsf{fix}^{\nu}_{n} s\, t_{1..n})) \rhd E(e(s\, (\mathsf{fix}^{\nu}_{n} s)\, t_{1..n}))}$$

Fig. 3. Strongly normalizing terms

only because there are no "junk terms" like $0\,(\lambda xx)$ in our language which block reduction but are neither neutral nor values.

In the remainder of this section, we prepare for the model construction for $\mathsf{F}^{\frown}_{\omega}$ that will verify strong normalization. As usual, we interpret types as sets $\mathcal{A}$ of strongly normalizing terms, where $\mathcal{A}$ is closed under safe expansion. In the iso-case, we could interpret a coinductive type $\mathcal{C} := [\![\nu^{a+1} F]\!]$ as $\{r \mid \mathsf{out}\, r \in [\![F\,(\nu^{a} F)]\!]\}$, or in words, as these terms r whose *canonical observation* $\mathsf{out}\, r$ is already well-behaved. A corecursive object, say $\mathsf{fix}^{\nu}_{0} s$ can enter $\mathcal{C}$ by the safe expansion $\mathsf{out}\,(\mathsf{fix}^{\nu}_{0} s) \rhd \mathsf{out}\,(s\,(\mathsf{fix}^{\nu}_{0} s))$ provided that $s\,(\mathsf{fix}^{\nu}_{0} s) \in \mathcal{C}$ already. In the equi-case, however, a canonical observation is not available, we have no choice than to set the interpretation of $\mathcal{C}$ to the semantical type $[\![F\,(\nu^{a} F)]\!]$. How can now $\mathsf{fix}^{\nu}_{0} s$ enter $\mathcal{C}$? The solution is that each semantical type $\mathcal{A}$ is characterized by a set of evaluation contexts, $\mathcal{E}$, such that $t \in \mathcal{A}$ iff $E(t) \in \mathsf{SN}$ for all $E \in \mathcal{E}$. This characterization automatically ensures that $\mathcal{A}$ is closed under safe reduction and expansion. Now $\mathsf{fix}^{\nu}_{0}\, s$ enters $\mathcal{C}$ through the safe expansion $E(\mathsf{fix}^{\nu}_{0} s) \rhd E(s\,(\mathsf{fix}^{\nu}_{0} s))$. Formally, this will be proven in Lemma 5. In the following, we give constructions and properties of semantical types. Due to lack of space, the presentation is rather dense, more details can be found in the author's thesis [2].

5.1 Orthogonality

We say that term t is *orthogonal* to evaluation context E,

$$t \perp E \;:\Longleftrightarrow\; E(t) \in \mathsf{SN}.$$

We could also say t *behaves well* in E. A *semantical type* $\mathcal{A}$ is the set of terms which behave well in all $E \in \mathcal{E}$, where $\mathcal{E}$ is some set of strongly normalizing evaluation contexts. The space of semantical types is called SAT.

$\mathcal{E}^\perp$	$:= \{t \mid t \perp E \text{ for all } E \in \mathcal{E}\}$		
$\mathcal{A}^\perp$	$:= \{E \mid t \perp E \text{ for all } t \in \mathcal{A}\}$		
SAT	$:= \{\mathcal{E}^\perp \mid \{\mathsf{Id}\} \subseteq \mathcal{E} \subseteq \mathsf{Scxt}\}$		saturated sets
$\mathcal{N}$	$:= \mathsf{Scxt}^\perp \supset \mathsf{SNe}$		neutral terms
$\mathcal{S}$	$:= \{\mathsf{Id}\}^\perp$		s.n. terms
$\overline{\mathcal{A}}$	$:= \mathcal{A}^{\perp\perp}$		closure
$\mathcal{A} \boxed{\to} \mathcal{E}^\perp$	$:= \{\mathsf{Id}, E \circ (_\, s) \mid E \in \mathcal{E}, s \in \mathcal{A}\}^\perp$		function space
$\boxed{\bigcap}\mathfrak{A}$	$:= \bigcap \mathfrak{A}$	for $\mathfrak{A} \subseteq \mathsf{SAT}$	infimum
$\boxed{\bigcup}\mathfrak{A}$	$:= \overline{\bigcup \mathfrak{A}}$	for $\mathfrak{A} \subseteq \mathsf{SAT}$	supremum

The greatest semantical type is $\mathcal{S} = \mathsf{SN}$; the least semantical type $\mathcal{N}$ contains all terms which behave well in all good contexts, including the variables and even more, all safe expansions of strongly normalizing neutral terms. But due to rule SNE-FIX$^\mu$FIX$^\nu$, also some corecursive values inhabit $\mathcal{N}$, e. g., $\mathsf{fix}^\nu_0 \lambda zy$.

Lemma 2 (Properties of saturated sets)

1. *Galois connection:* $\mathcal{A}^\perp \supseteq \mathcal{E} \iff \mathcal{A} \subseteq \mathcal{E}^\perp$. *This implies* $\mathcal{A} \subseteq \mathcal{A}^{\perp\perp}$, $A \subseteq \mathcal{B} \Longrightarrow \mathcal{A}^\perp \supseteq \mathcal{B}^\perp$, *and* $\mathcal{A}^{\perp\perp\perp} = \mathcal{A}^\perp$, *and the same laws for* $\mathcal{E}$*s.*
2. *Biorthogonal closure: If* $\mathcal{A} \subseteq \mathcal{S}$ *then* $\{\mathsf{Id}\} \subseteq \mathcal{A}^\perp \subseteq \mathsf{Scxt}$ *and* $\mathcal{A}^{\perp\perp} \in \mathsf{SAT}$.
3. *De Morgan 1:* $\bigcap_{i\in I} \mathcal{E}_i^\perp = (\bigcup_{i\in I} \mathcal{E}_i)^\perp$.
4. *De Morgan 2:* $\bigcup_{i\in I} \mathcal{E}_i^\perp \subseteq (\bigcap_{i\in I} \mathcal{E}_i)^\perp$.
5. *Reduction/expansion closure: If* $t \in \mathcal{E}^\perp$ *and* $t \rhd t'$ *or* $t \lhd t'$ *then* $t' \in \mathcal{E}^\perp$.
6. *Normalization: If* $t \in \mathcal{A} \in \mathsf{SAT}$ *then either* $t \in \mathcal{N}$ *or* $t \rhd v$.
7. *Function space: If* $\mathcal{A} \subseteq \mathcal{S}$ *and* $\mathcal{B} \in \mathsf{SAT}$ *then* $\mathcal{A} \boxed{\to} \mathcal{B} \in \mathsf{SAT}$.
8. *Infimum and supremum: If* $\mathfrak{A} \subseteq \mathsf{SAT}$ *then* $\boxed{\bigcap}\mathfrak{A} \in \mathsf{SAT}$ *and* $\boxed{\bigcup}\mathfrak{A} \in \mathsf{SAT}$.

In general, the inclusion in law De Morgan 2 is strict; thus, taking the orthogonal seems to be an intuitionistic rather than a classical negation.

Lemma 3 (Abstraction and application). *Let* $\mathcal{B} \in \mathsf{SAT}$.

1. *If* $\mathsf{Var} \subseteq \mathcal{A}$ *and* $r\,s, [s/x]t \in \mathcal{B}$ *for all* $s \in \mathcal{A}$, *then* $r, \lambda xt \in \mathcal{A} \boxed{\to} \mathcal{B}$.
2. *If* $r \in \mathcal{A} \boxed{\to} \mathcal{B}$ *and* $s \in \mathcal{A}$ *then* $r\,s \in \mathcal{B}$.

The proof of 1. uses extensionality (Lemma 1) to show $r \perp \mathsf{Id}$ from $r\,x \in \mathcal{B}$.

5.2 Recursion and Corecursion, Semantically

In this section, we characterize admissible types for recursion and corecursion in our semantics and prove semantical soundness of type-based termination. Let O denote some initial segment of the set-theoretic ordinals.

The semantic type family $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ is *admissible for recursion on the $n+1$st argument* if

ADM-μ-SHAPE	there is an index set K and there are $\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{I}, \mathcal{C} \in K \times \mathsf{O} \to \mathsf{SAT}$ such that for all $\alpha \in \mathsf{O}$, $\mathcal{A}(\alpha) = \bigcap_{k \in K} (\mathcal{B}_{1..n}(k, \alpha) \boxed{\to} \mathcal{I}(k, \alpha) \boxed{\to} \mathcal{C}(k, \alpha))$,
ADM-μ-START	$\mathcal{I}(k, 0) \subseteq \mathcal{N}$ for all $k \in K$, and
ADM-μ-LIMIT	$\bigcap_{\alpha < \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limits $\lambda \in \mathsf{O} \setminus \{0\}$.

In ADM-μ-SHAPE, the intersection $\bigcap$ stands for a quantification over types, the $\mathcal{B}_i$ for non-recursive arguments, the $\mathcal{I}$ for the recursive argument of inductive type, and $\mathcal{C}$ for the result type.

Lemma 4 (Recursion is a function). *Let $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ be admissible for recursion on the $n+1$st argument. If $s \in \mathcal{A}(\alpha) \boxed{\to} \mathcal{A}(\alpha + 1)$ for all $\alpha + 1 \in \mathsf{O}$, then $\mathsf{fix}^{\mu}_{n} s \in \mathcal{A}(\beta)$ for all $\beta \in \mathsf{O}$.*

Proof. By transfinite induction on $\beta \in \mathsf{O}$ [2, Lemma 3.32].

The soundness of corecursion makes crucial use of our definition of a semantical type by a set of evaluation contexts. It also requires that coinductive types denote the whole term universe $\mathcal{S}$ in the 0th iteration (ADM-ν-START).

The semantic type family $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ is *admissible for corecursion with n arguments* if

ADM-ν-SHAPE	for some index set K and $\mathcal{B}_{1..n}, \mathcal{C} \in K \times \mathsf{O} \to \mathsf{SAT}$, $\mathcal{A}(\alpha) = \bigcap_{k \in K} (\mathcal{B}_{1..n}(k, \alpha) \boxed{\to} \mathcal{C}(k, \alpha))$ for all $\alpha \in \mathsf{O}$,
ADM-ν-START	$\mathcal{S} \subseteq \mathcal{C}(k, 0)$ for all $k \in K$, and
ADM-ν-LIMIT	$\bigcap_{\alpha < \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limits $\lambda \in \mathsf{O} \setminus \{0\}$.

Lemma 5 (Corecursion is a function). *Let $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ be admissible for corecursion with n arguments. If $s \in \mathcal{A}(\alpha) \boxed{\to} \mathcal{A}(\alpha + 1)$ for all $\alpha + 1 \in \mathsf{O}$, then $\mathsf{fix}^{\nu}_{n} s \in \mathcal{A}(\beta)$ for all $\beta \in \mathsf{O}$.*

Proof. By transfinite induction on $\beta \in \mathsf{O}$ [2, Lemma 3.37].

5.3 Lattices and Iteration

The saturated sets form a complete lattice $[\![*]\!] = \mathsf{SAT}$ with least element $\bot^* := \mathcal{N}$ and greatest element $\top^* := \mathcal{S}$. It is ordered by inclusion $\sqsubseteq^* := \subseteq$ and has arbitrary infima $\inf^* := \boxed{\bigcap}$ and suprema $\sup^* := \boxed{\bigcup}$. Let $[\![\mathsf{ord}]\!] := [0; \top^{\mathsf{ord}}]$ be an initial segment of the set-theoretic ordinals which is closed under suprema, such that all (co)inductive types reach their fixpoint at ordinal $\top^{\mathsf{ord}}$. An upper

bound for the $\top^{\mathsf{ord}}$ is the ωth uncountable [3], although the true closure ordinal is probably much smaller, and it would be interesting to find out more about it. With the usual ordering on ordinals, $[\![\mathsf{ord}]\!]$ constitutes a complete lattice as well. Function kinds $[\![\circ\kappa \to \kappa']\!] := [\![\kappa]\!] \to [\![\kappa']\!]$ are interpreted as set-theoretic function spaces; a covariant function kind denotes just the monotonic functions and a contravariant kind the antitonic ones. For all function kinds, ordering is defined pointwise: $\mathcal{F} \sqsubseteq^{p\kappa\to\kappa'} \mathcal{F}' :\Longleftrightarrow \mathcal{F}(\mathcal{G}) \sqsubseteq^{\kappa'} \mathcal{F}'(\mathcal{G})$ for all $\mathcal{G} \in [\![\kappa]\!]$. Similarly, $\bot^{p\kappa\to\kappa'}(\mathcal{G}) := \bot^{\kappa'}$ is defined pointwise, and so are $\top^{p\kappa\to\kappa'}$, $\inf^{p\kappa\to\kappa'}$, and $\sup^{p\kappa\to\kappa'}$.

For monotone $\mathcal{F} \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]$ we define iteration from below and above as usual:

$$
\begin{array}{llll}
\boldsymbol{\mu}^0\mathcal{F} & = \bot^\kappa & \boldsymbol{\nu}^0\mathcal{F} & = \top^\kappa \\
\boldsymbol{\mu}^{\alpha+1}\mathcal{F} & = \mathcal{F}(\boldsymbol{\mu}^\alpha\mathcal{F}) & \boldsymbol{\nu}^{\alpha+1}\mathcal{F} & = \mathcal{F}(\boldsymbol{\nu}^\alpha\mathcal{F}) \\
\boldsymbol{\mu}^\lambda\mathcal{F} & = \sup^\kappa_{\alpha<\lambda} \boldsymbol{\mu}^\alpha\mathcal{F} & \boldsymbol{\nu}^\lambda\mathcal{F} & = \inf^\kappa_{\alpha<\lambda} \boldsymbol{\nu}^\alpha\mathcal{F}
\end{array}
$$

For fixed $\mathcal{F}$, $\boldsymbol{\mu}^\alpha\mathcal{F}$ is monotonic in α and $\boldsymbol{\nu}^\alpha\mathcal{F}$ is antitonic in α.

6 Soundness

For a constructor constant $C\!:\!\kappa$, the semantics $[\![C]\!] \in [\![\kappa]\!]$ is defined as follows:

$$
\begin{array}{llll}
[\![\to]\!](\mathcal{A},\mathcal{B} \in [\![*]\!]) & := \mathcal{A} \boxed{\to} \mathcal{B} & [\![\infty]\!] & := \top^{\mathsf{ord}} \\
[\![\mu_\kappa]\!](\alpha)(\mathcal{F} \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]) & := \boldsymbol{\mu}^\alpha\mathcal{F} & [\![\mathsf{s}]\!](\top^{\mathsf{ord}}) & := \top^{\mathsf{ord}} \\
[\![\nu_\kappa]\!](\alpha)(\mathcal{F} \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]) & := \boldsymbol{\nu}^\alpha\mathcal{F} & [\![\mathsf{s}]\!](\alpha < \top^{\mathsf{ord}}) & := \alpha + 1 \\
[\![\forall_\kappa]\!](\mathcal{F} \in [\![\kappa]\!] \to [\![*]\!]) & := \bigcap_{\mathcal{G}\in[\![\kappa]\!]} \mathcal{F}(\mathcal{G}) & &
\end{array}
$$

We extend this semantics to constructors F in the usual way.

Let θ be a partial mapping from constructor variables to sets. We say $\theta \in [\![\Delta]\!]$ if $\theta(X) \in [\![\kappa]\!]$ for all $(X : p\kappa) \in \Delta$. A partial order on valuations is defined by $\theta \sqsubseteq \theta' \in [\![\Delta]\!] \;:\Longleftrightarrow\; \theta(X) \sqsubseteq^p \theta'(X) \in [\![\kappa]\!]$ for all $(X : p\kappa) \in \Delta$. Herein, we have used $\sqsubseteq^-$ for $\sqsupseteq$, and $\sqsubseteq^\circ$ for $=$, and $\sqsubseteq^+$ as synonym for $\sqsubseteq$.

Theorem 1 (Soundness of type-related judgements). *Let $\theta, \theta' \in [\![\Delta]\!]$.*

1. *If $\Delta \vdash F : \kappa$ then $[\![F]\!]_\theta \in [\![\kappa]\!]$.*
2. *If $\Delta \vdash F = F' : \kappa$ and $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$, then $[\![F]\!]_\theta \sqsubseteq [\![F']\!]_{\theta'} \in [\![\kappa]\!]$.*
3. *If $\Delta \vdash F \leq F' : \kappa$ and $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$, then $[\![F]\!]_\theta \sqsubseteq [\![F']\!]_{\theta'} \in [\![\kappa]\!]$.*
4. *If $\Delta \vdash A$ fix^μ_n-adm, then $[\![A]\!]_\theta$ is admissible for recursion on the $n+1$st arg.*
5. *If $\Delta \vdash A$ fix^ν_n-adm, then $[\![A]\!]_\theta$ is admissible for corecursion with n arguments.*

We extend valuations θ to term variables and say $\theta \in [\![\Gamma]\!]$ if $\theta(X) \in [\![\kappa]\!]$ for all $(X : p\kappa) \in \Gamma$ and $\theta(x) \in [\![A]\!]_\theta$ for all $(x : A) \in \Gamma$. Let $(\!|t|\!)_\theta$ denote the capture-avoiding substitution of $\theta(x)$ for x in t, simultaneously for all $x \in \mathsf{FV}(t)$.

Theorem 2 (Soundness of $\mathsf{F}_{\widehat{\omega}}$). *If $\Gamma \vdash t : A$ and $\theta \in [\![\Gamma]\!]$ then $(\!|t|\!)_\theta \in [\![A]\!]_\theta$.*

The theorem is proved by induction on the typing derivation [2, Thm. 3.49]. As a consequence, taking $\theta(x) = x$ for all $(x : A) \in \Gamma$ and $\theta(X) = \top^\kappa$ for all $(X : p\kappa) \in \Gamma$, we get $t = (\!|t|\!)_\theta \in [\![A]\!]_\theta \subseteq \mathsf{SN}$.

7 Conclusion

We have presented a type system for termination of recursive functions over equi-inductive and -coinductive types and shown its soundness by a model based on orthogonality. All reductions of the corresponding iso-system are simulated, hence, termination of the iso-system follows as a special case.

Parigot [24] already introduces equi-inductive types to model efficient recursion schemes in system AF_2, second order functional arithmetic. Raffalli [26] considers also equi-coinductive types. However, recursion is limited to Mendler-style (co)iteration [22], preventing a direct implementation of primitive recursive programs such as factorial. Iteration is but a special case of the recursion scheme of the present work, which generalizes course-of-value recursion.

Orthogonality has been introduced by Girard for the semantics of linear logic; it pops up again in Ludics [14]. Parigot has implicitly used orthogonality to prove strong normalization of second-order classical natural deduction [25]. His work has been extended by Matthes to positive fixed-point types [20]. Lindley and Stark [18] use orthogonality to show strong normalization of the monadic lambda-calculus and give credit to Pitts. Vouillon and Melliès [30] model recursive types with orthogonality, Vouillon [29] bases subtyping rules for union types on orthogonality.

Related works on type-based termination include: Hughes, Pareto, and Sabry [16], who treat first-order inductive and coinductive types that close at iteration ω. Their system is also *equi* in spirit [23, Ch. 3.10], however, they do not give reduction rules but construct a denotational model. Barthe et al. [8] prove strong normalization for recursive functions over sized inductive types of kind $*$. Although there are no explicit (un)folding operations in and out, the only way to generate inductive data is via constructors for labeled sums, which is crucially in the reduction rule for recursion. Thus, the system is *iso* in disguise, in is merged into the constructors, and out into case distinction. Blanqui [10] considers type-based termination for his Calculus of Algebraic Constructions—iso-inductive in spirit—which subsumes the Calculus of Inductive Constructions (CIC). Barthe, Gregoire, and Pastawski [9] have extended type-based termination to the CIC. Xi [31] bases termination on dependent types, albeit only dependencies on integer expressions, which gives him a great flexibility in termination measures. Since in his system a typing context can become unsatisfiable, he only shows call-by-value normalization of closed programs. Blanqui and Riba [11] manage to avoid unsatisfiable contexts, and thus, recover strong normalization.

In our treatment of equi-(co)inductive types, it is a bit unsatisfactory that terms like $\mathsf{fix}_0^\mu s\,(\mathsf{fix}_0^\nu\, s')$ are blocked. One could think of allowing both unfoldings, arriving at a non-confluent calculus. The techniques described in this paper are then no longer sufficient to prove strong normalization, but maybe methods used for normalization of classical logic could be employed.

Acknowledgments. Thanks to my supervisor, Martin Hofmann, and to Ralph Matthes for supporting my thesis work. Thanks to Lennart Beringer for proof reading and to the anonymous referees for their insightful and helpful comments.

Jérôme Vouillon's excellent article on semantics of union types [29] inspired me to build a model based on orthogonality. Thanks to Frédéric Blanqui and Colin Riba for their invitation to LORIA in February 2007 and for discussions which deepened my understanding of orthogonality, strong normalization, and equi-inductive types.

References

1. Abel, A.: Polarized subtyping for sized types. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 381–392. Springer, Heidelberg (2006)
2. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Ph.D. thesis, Ludwig-Maximilians-Universität München (2006)
3. Abel, A.: Semi-continuous sized types and termination. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 72–88. Springer, Heidelberg (2006)
4. Abel, A.: Towards generic programming with sized types. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 10–28. Springer, Heidelberg (2006)
5. Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 190–204. Springer, Heidelberg (2004)
6. Altenkirch, T.: Constructions, Inductive Types and Strong Normalization. Ph.D. thesis, University of Edinburgh (1993)
7. Altenkirch, T.: Logical relations and inductive/coinductive types. In: Gottlob, G., Grandjean, E., Seyr, K. (eds.) CSL 1999. LNCS, vol. 1683, pp. 343–354. Springer, Heidelberg (1999)
8. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Math. Struct. in Comput. Sci. 14, 1–45 (2004)
9. Barthe, G., Grégoire, B., Pastawski, F.: CIC$\hat{~}$: Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 257–271. Springer, Heidelberg (2006)
10. Blanqui, F.: A type-based termination criterion for dependently-typed higher-order rewrite systems. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 24–39. Springer, Heidelberg (2004)
11. Blanqui, F., Riba, C.: Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 105–119. Springer, Heidelberg (2006)
12. Geuvers, H.: Inductive and coinductive types with iteration and recursion. In: Nordström, B., Pettersson, K., Plotkin, G. (eds.) Types for Proofs and Programs (TYPES'92), Båstad, Sweden, pp. 193–217 (1992)
13. Giménez, E.: Structural recursive definitions in type theory. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 397–408. Springer, Heidelberg (1998)
14. Girard, J.-Y.: Locus solum: From the rules of logic to the logic of rules. Math. Struct. in Comput. Sci. 11, 301–506 (2001)
15. Goguen, H.: Typed operational semantics. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 186–200. Springer, Heidelberg (1995)
16. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL'96, pp. 410–423 (1996)

17. Joachimski, F., Matthes, R.: Short proofs of normalization. Archive of Mathematical Logic 42, 59–87 (2003)
18. Lindley, S., Stark, I.: Reducibility and $\top\top$-lifting for computation types. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, Springer, Heidelberg (2005)
19. Matthes, R.: Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types. Ph.D. thesis, Ludwig-Maximilians-University (1998)
20. Matthes, R.: Non-strictly positive fixed-points for classical natural deduction. Ann. Pure Appl. Logic 133, 205–230 (2005)
21. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: Proc. of the 2nd IEEE Symp. on Logic in Computer Science (LICS'87), pp. 30–36. IEEE Computer Society Press, Los Alamitos (1987)
22. Mendler, N.P.: Inductive types and type constraints in the second-order lambda calculus. Annals of Pure. and Applied Logic 51, 159–172 (1991)
23. Pareto, L.: Types for Crash Prevention. Ph.D. thesis, Chalmers University of Technology (2000)
24. Parigot, M.: Recursive programming with proofs. Theor. Comput. Sci. 94, 335–356 (1992)
25. Parigot, M.: Proofs of strong normalization for second order classical natural deduction. The. Journal of Symbolic Logic 62, 1461–1479 (1997)
26. Raffalli, C.: Data types, infinity and equality in system af_2. In: Meinke, K., Börger, E., Gurevich, Y. (eds.) CSL 1993. LNCS, vol. 832, pp. 280–294. Springer, Heidelberg (1994)
27. Steffen, M.: Polarized Higher-Order Subtyping. Ph.D. thesis, Technische Fakultät, Universität Erlangen (1998)
28. van Raamsdonk, F., Severi, P., Sørensen, M.H., Xi, H.: Perpetual reductions in lambda calculus. Inf. Comput. 149, 173–225 (1999)
29. Vouillon, J.: Subtyping union types. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 415–429. Springer, Heidelberg (2004)
30. Vouillon, J., Melliès, P.-A.: Semantic types: A fresh look at the ideal model for types. In: Jones, N.D., Leroy, X. (eds.) Proc. of the 31st ACM Symp. on Principles of Programming Languages, POPL 2004, pp. 52–63. ACM Press, New York (2004)
31. Xi, H.: Dependent types for program termination verification. J. Higher-Order and Symb. Comput. 15, 91–131 (2002)

Semantics for Intuitionistic Arithmetic Based on Tarski Games with Retractable Moves

Stefano Berardi

C.S. Dept., University of Torino, Italy
http://www.di.unito.it/∼stefano

Abstract. We define an effective, sound and complete game semantics for $\mathtt{HA}_{\mathrm{inf}}$, Intuitionistic Arithmetic with ω-rule. Our semantics is equivalent to the original semantics proposed by Lorentzen [6], but it is based on the more recent notions of "backtracking" ([5], [2]) and of isomorphism between proofs and strategies ([8]). We prove that winning strategies in our game semantics are tree-isomorphic to the set of proofs of some variant of $\mathtt{HA}_{\mathrm{inf}}$, and that they are a sound and complete interpretation of $\mathtt{HA}_{\mathrm{inf}}$.

1 Why Game Semantics of Intuitionistic Arithmetic?

In [7], S.Hayashi proposed the use of an effective game semantics in his Proof Animation project. The goal of the project is "animating" (turning into algorithms) proofs of program specifications, in order to find bugs in the way a specification is formalized. Proofs are formalized in classical Arithmetic, and the method chosen for "animating" proofs is a simplified version of Coquand's game interpretation ([4], [5]) of $\mathtt{PA}_{\mathrm{inf}}$, classical arithmetic with ω-rule. The interest of the game interpretation is that it interprets rules of classical arithmetic by very simple operations, like arithmetical operation, reference to a pointer, adding and removing elements to a stack.

Coquand, however, defined implication $A \rightarrow B$ as classical implication, as "A is false or B is true". In real proofs, instead, we often use the constructive definition of implication $A \Rightarrow B$, which is: "assume A in order to prove B". $A \Rightarrow B$ is classically equivalent to "A is false or B is true", but this means that in order to interpret a proof in Coquand's semantics we have first to modify it. If we want some control and understanding of the algorithm we extract from a proof, instead, it is crucial to animate the *original* proof.

In this paper we adapt Coquand's game semantics of $\mathtt{PA}_{\mathrm{inf}}$ to game semantics of Intuitionistic Arithmetic $\mathtt{HA}_{\mathrm{inf}}$ with intuitionistic implication $\Rightarrow$. Our semantics is equivalent to the original Lorentzen's game semantics [6], and also bears some similarity with Hyland-Ong game semantics for simply typed lambda terms [10]. The main difference between our semantics and Lorentzen's semantics is that we do no not add dummy moves when interpreting connectives, but when interpreting implication (see [1], §4.4 for a discussion). Reducing the number of dummy moves is crucial in order to make evident the relationship between a

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 23–38, 2007.
© Springer-Verlag Berlin Heidelberg 2007

game strategy and the intuitionistic proof interpreted by the strategy. The main difference between our semantics and Hyland-Ong's semantics is, instead, that we consider all connectives as Lorentzen did. In this way the difference between the game interpretation for implication and the game interpretation for all other connectives becomes evident.

The game interpretation of the intuitionistic implication introduced in this paper aims to be one step in the Proof Animation project. What is still missing are game semantics combining all features of real proofs: classical logic, intuitionistic implication, cut rule, induction rule, and so forth. We claim that the semantics introduced in this paper can also be used to interpret Cut rule through the notion of dialogue, as it was done by Coquand, Hyland-Ong and Herbelin. We did not include Cut rule for reason of space. Even without Cut rule, we can use our semantic to interpret the evidence provided by an intuitionistic proof in term of very simple operations, without blurring the relation with the proof structure during the interpretation.

1.1 The Plan of the Paper

This is the plan of the paper. In §2 we introduce the language of arithmetic. In §3 we introduce our game semantics. In §4 we introduce $\mathtt{HA}_{\mathrm{inf}}$, intuitionistic arithmetic with ω-rule. In §5, 6 we prove that winning strategies of our semantics are tree-isomorphic the proofs of some variant of $\mathtt{HA}_{\mathrm{inf}}$, and that our game semantics is sound and complete for $\mathtt{HA}_{\mathrm{inf}}$. For a discussion of our definition of game, for one example of winning strategy and for one example of play we refer to [1], §4, 9, 10.

2 The Language of Arithmetic

In this section we introduce a language $\mathtt{L}_{\mathtt{HA}}$ for first order arithmetic, the notion of judgement, and the notion of sequent. In the next section we define our game semantics.

$\mathtt{L}_{\mathtt{HA}}$ has a connective $A \Rightarrow B$ denoting intuitionistic implication. We also introduce a "game language" $\mathtt{L}_{\mathtt{G}} \supset \mathtt{L}_{\mathtt{HA}}$ for game semantics. The formulas of $\mathtt{L}_{\mathtt{G}}$ denote games interpreting formulas of $\mathtt{L}_{\mathtt{HA}}$. Each connective of $\mathtt{L}_{\mathtt{HA}}$ corresponds to some operator defining games in $\mathtt{L}_{\mathtt{G}}$, operator which we will denote with the same symbol. In this way each formula of $\mathtt{L}_{\mathtt{HA}}$ will be also considered as a denotation for some game in $\mathtt{L}_{\mathtt{HA}}$ interpreting it. The only difference between $\mathtt{L}_{\mathtt{G}}$ and $\mathtt{L}_{\mathtt{HA}}$ is that $\mathtt{HA}$ has one extra connective $\rightarrow$. The connective $\rightarrow$ denotes one binary operator on games, used as an intermediate step in the interpretation of intuitionistic implication $\Rightarrow$.

We divide the formulas of the game language $\mathtt{L}_{\mathtt{G}}$ into disjunctive and conjunctive, by generalizing the usual distinction between disjunctive and conjunctive formulas we have in Logic. We consider $A \Rightarrow B$ a conjunction, and $A \rightarrow B$ a disjunction. The language $\mathtt{L}_{\mathtt{HA}}$, for intuitionistic arithmetic, consists of all formulas of $\mathtt{L}_{\mathtt{G}}$ which are $\rightarrow$-free.

Definition 1

- $\mathtt{L_G}$ *is the first order language including a r.e. set of function symbols for recursive functions (at least* $0, S, +, *$*), and a r.e. set of predicate symbols for recursive predicates (at least* $<, =$*), and the connectives* $T, F, \wedge, \vee, \neg, \Rightarrow, \forall, \exists$ *and* $\rightarrow$*. We call* T, F *"true" and "false", and we consider them different from all atomic formulas.*
- $\mathtt{L_{HA}}$*, the language of intuitionistic arithmetic, is the sub-language of* $\mathtt{L_G}$ *consisting of all formulas which are* $\rightarrow$*-free.* $\mathtt{L}^+$*, the language of positive formulas, is the sub-language of* $\mathtt{L_{HA}}$ *consisting of all formulas which are* $\neg, \Rightarrow, \rightarrow$*-free.* $\mathtt{L_{G0}}, \mathtt{L_{HA0}}, \mathtt{L}_0^+$ *are the sets of closed formulas of* $\mathtt{L_G}, \mathtt{L_{HA}}, \mathtt{L}^+$*.*
- *If* $A \in \mathtt{L_G}$ *is* F*, or is atomic, or starts with* $\vee, \rightarrow, \exists$*, we say that* A *is disjunctive. If* $A \in \mathtt{L_{G0}}$ *is* T*, or starts with* $\wedge, \Rightarrow, \neg, \forall$*, we say that* A *is conjunctive.*

We consider T conjunctive because the constant true is equivalent to an empty conjunction. We consider F disjunctive because the constant false is equivalent to an empty disjunction. We consider (arbitrarily) all atomic formulas disjunctive. The only non-trivial choice is considering $A \Rightarrow B$ conjunctive and $A \rightarrow B$ disjunctive: we discuss this choice in [1], §4.4. $\neg A$ is taken conjunctive by analogy with $A \Rightarrow F$, which is equivalent to $\neg A$. We use $t, u, v, \ldots$ to denote closed terms of $\mathtt{L_{G0}}$. We use $A, B, C, D,\ A_1, B_1, C_1, D_1,\ \ldots$ to denote closed formulas of $\mathtt{L_{G0}}$, and $a, b, c, \ldots$ for atomic closed formulas.

We consider the usual subformula relation between closed formulas, with the additional clauses: the only immediate subformula of $A \Rightarrow B$ is $A \rightarrow B$, and: the only subformula of a atomic is T or F, according if a is true or false.

Definition 2 (*Immediate Subformula relation* $<_1$ *over* $\mathtt{L_{G0}}$)

- T, F *have* no *immediate subformula. If* $A \in \mathtt{L_{G0}}$ *is atomic, the only immediate subformula of* A *is* T *if* A *is true, and* F *if* A *is false. The immediate subformulas of* $A \vee B, A \wedge B, A \rightarrow B$ *are* A, B*. The only immediate subformula of* $A \Rightarrow B$ *is* $A \rightarrow B$*. The only immediate subformula of* $\neg A$ *is* A*. The immediate subformulas of* $\forall x.A[x], \exists x.A[x]$ *are all* $A[t]$ *for* t *closed term of* $\mathtt{L_G}$*. If* A *is an immediate subformula of* B*, we write* $A <_1 B$ *(or* $B >_1 A$*). The subformula relation* $<$ *is the transitive closure of* $<_1$*.*
- *Assume* $A <_1 C$*. If* $C = \neg A, A \rightarrow B$*, we say that* A *is* negative *in* C*. In all other cases, we say that* A *is* positive *in* C*.*
- *An occurrence of a subformula* B *in* A *is any sequence* $A_0, \ldots, A_n$ *with* $A_0 = A$*, and* $A_i >_1 A_{i+1}$ *for all* $i < n$*, and* $A_n = B$*. The occurrence is* positive *if* A_{i+1} *is negative in* A_i *for an even number of* i*, the occurrence is* negative*, if* A_{i+1} *is negative in* A_i *for an odd number of* i*.*

We use the informal notion of tree and all tree terminology (children, father, ascendant, descendant, branch, leaf). The subformula tree of $C \in \mathtt{L_{G0}}$ is the tree of all subformula occurrences of C, ordered by the subformula relation $<$. If $C \in \mathtt{L_{HA}}$, then C by definition is $\rightarrow$-free. Yet, the subformula tree of C in $\mathtt{L_{G0}}$ can include occurrences of some $A \rightarrow B$, the children of the occurrences

$A \Rightarrow B$ of the tree. The only difference between an occurrence of $A \Rightarrow B$ and an occurrences of $A \rightarrow B$ is that the former is conjunctive while the latter is disjunctive. This duplication of nodes looks useless, but it will play a crucial role in the completeness result. If we restrict the subformula relation to $\mathtt{L_{HA}}$, then we skip $A \rightarrow B$. In this case the immediate subformulas of $A \Rightarrow B$ are A, B, as expected. We now formally define the notion of judgement and sign of a judgement.

Definition 3 (*Judgement and Sign of a Judgement*). *Let* $A \in \mathtt{L_{GO}}$.

1. *We call* $\mathtt{t}.A$ *and* $\mathtt{f}.A$ judgements, *and we read them "A is true", "A is false". We call s the sign of sA, and we say the sign is* positive *if* $s = \mathtt{t}.$, *is* negative *if* $s = \mathtt{f}.$. *We denote by* $s^{\perp}$ *the opposite sign of s:* $s^{\perp} = \mathtt{f}.$ *if* $s = \mathtt{t}.$, *and* $s^{\perp} = \mathtt{t}.$ *if* $s = \mathtt{f}.$.
2. *We say that* $\mathtt{t}.A$ *is a disjunctive judgement (respectively, a conjunctive judgement) if A is a disjunctive formula (respectively, a conjunctive formula).*
3. *We say that* $\mathtt{f}.A$ *is a disjunctive judgement (respectively, a conjunctive judgement) in the opposite case, namely, if A is a* conjunctive *formula (respectively, a* disjunctive *formula).*

By switching the sign of a formula we switch conjunctive and disjunctive judgements. For instance, $\mathtt{t}.A \Rightarrow B$ is conjunctive, while $\mathtt{f}.A \Rightarrow B$ is disjunctive. There is a sub-judgement relation analogous to the sub-formula relation, except that the judgements have opposite signs whenever the subformula is negative.

Definition 4 (*Immediate sub-judgement relation*). *Let* $A \in \mathtt{L_{GO}}$. *We say that* $s'A'$ *is an immediate sub-judgement of* sA, *and we write* $s'A' <_1 sA$, *if* $A' <_1 A$, *and:* $s' = s$ *if* A' *is positive in* A, *and* $s' = s^{\perp}$ *if* A' *is negative in* A. *The sub-judgement relation* $<$ *is the transitive closure of* $<_1$.

For instance, the immediate sub-judgements of $\mathtt{t}.A \rightarrow B$ are $\mathtt{f}.A$ and $\mathtt{t}.B$, while the immediate sub-judgements of $\mathtt{f}.A \rightarrow B$ are $\mathtt{t}.A$ and $\mathtt{f}.B$. If we restrict the sub-judgement relation to judgements of $\mathtt{L_{HA}}$, then the immediate sub-judgements of $\mathtt{t}.A \Rightarrow B$ are $\mathtt{f}.A, \mathtt{t}.B$, as expected.

We will now introduce a notion of pointed sequent: sequents having one "active formula" in evidence. We first recall what (finite) multisets are. A multiset is a set with possibly repetitions, in which elements $x_1, \ldots, x_n$ are distinguished through the use of labels. We use positive integers as labels. We formalize a multiset X with the set of its indexes paired with the corresponding elements: $X = \{\langle i_1, x_1\rangle, \ldots, \langle i_n, x_n\rangle\}$, with $0 < i_1 < \ldots < i_n$. We often write $X = x_1, \ldots, x_n$, leaving $i_1, \ldots, i_n$ implicit: the actual indexing is irrelevant. We use $\Gamma, \Delta, \Gamma', \Delta', \ldots$ to denote multisets of closed formulas in $\mathtt{L_{GO}}$. $\{A\}$ is the multiset consisting of one pair $\langle j, A\rangle$ for some $j > 0$. We denote the disjoint union of two multisets Γ and Δ with Γ, Δ. By renaming indexing, we can always assume two multisets are disjoint. We denote $\Gamma, \{A\}$ with Γ, A.

Definition 5 (*Intuitionistic Sequents and occurrences*). *Let* $\Gamma = \{\langle j_1, A_1\rangle, \ldots, \langle j_n, A_n\rangle\}$ *and* $\{\langle j, D\rangle\}$ *be two multisets over* $\mathtt{L_{GO}}$, *with disjoint set of indexes, and* $0 < j_1 < \ldots < j_n$.

- *An intuitionistic sequent on* $\mathtt{L_{GO}}$ *is any pair* $\Gamma \vdash \{D\}$*. A sequent is on* $\mathtt{L_{HAO}}$ *if all its formulas are.*
- *The indexing of* $\Gamma \vdash \{D\}$ *is the union of indexing of* Γ *and* $\{D\}$*. We call the index* $i > 0$ *of* A *in* $\Gamma \vdash D$ *an* occurrence *of* A *in* $\Gamma \vdash D$*.*
- *A pointed sequent is a pair* $\langle \Gamma \vdash D, i\rangle$ *of an intuitionistic sequent on* $\mathtt{L_{GO}}$*, and an occurrence* $i > 0$ *of some* A *in* $\Gamma \vdash D$*. We call* i *the* active occurrence *of the sequent.*
- $\langle \Gamma \vdash D, i\rangle$ *has canonical indexing if* $j = 1, j_1 = 2, \ldots, j_n = n+1$*. The canonical version* $\langle \Gamma' \vdash D', i'\rangle$ *of* $\Gamma \vdash D$ *is obtained by replacing* j *with* 1*,* j_1 *with* 2*, …,* j_n *with* $n+1$*, and defining* $i' = 1$ *if* $i = j$*, and* $i' = k+1$ *if* $i = j_k$ *for some* k*.*

From now on, we usually consider intuitionistic pointed sequents, and we call them just "sequents", for short. The canonical indexing $\langle \Gamma' \vdash D', i'\rangle$ of $\langle \Gamma \vdash D, i\rangle$ is obtained by renaming the indexes of $\langle \Gamma \vdash D, i\rangle$. The two sequents will be equivalent in our semantics. Also any two sequents $\langle \Gamma \vdash D, i\rangle$ and $\langle \Gamma \vdash D, j\rangle$, having different active formulas, will be equivalent. In logic, the indexing of Γ, and the active formula of a sequent are an irrelevant information. We can drop them and we can write a sequent just as $\Gamma \vdash D$. However, we need the indexing of Γ and the active formula when defining the game interpretation.

Occurrences of a formula in a sequent have a sign, according to the side of the sequent they belong to. The sign of an occurrence, like the sign of a judgement, switches conjunctive and disjunctive formulas.

Definition 6 (*Sign of a formula in a sequent*)

- *A formula occurrence in* $\Gamma \vdash D$ *has sign* `t.` *(is positive) if it is in the right-hand-side. It has sign* `f.` *(is negative) if it is in the left-hand-side.*
- *Assume* A *is a positive occurrence. Then* A *is a disjunctive occurrence if* A *is a disjunctive formula, and a conjunctive occurrence if* A *is a conjunctive formula.*
- *Assume* A *is a negative occurrence. Then* A *is a disjunctive occurrence if* A *is a conjunctive formula, and a conjunctive occurrence if* A *is a disjunctive formula.*

We introduce two operations on sequents. The operation $\langle \Gamma \vdash D, i\rangle + sA$ adds an active formula A of sign s to the sequent $\langle \Gamma \vdash D, i\rangle$ (and removes D, if we add A to the right-hand-side), and chooses some index $n+1$ for A. The operation $\langle \Gamma \vdash D, i\rangle + \mathtt{bck}(j)$ changes the active formula of $\langle \Gamma \vdash D, i\rangle$ from i to j (provided j is an index of $\Gamma \vdash D$).

Definition 7

1. *For all positive integers* j *we introduce the notation* $\mathtt{bck}(j)$*.*
2. `Moves` *is a set of operators on sequents, consisting of all judgements* sA *and all notations* $\mathtt{bck}(j)$*.*
3. *Assume* $\langle \Gamma \vdash D, i\rangle$ *is any sequent, having maximum index* n*. Let* $A \in \mathtt{L_{GO}}$*,* $j > 0$*. We define a sequent* $\langle \Gamma \vdash D, i\rangle + m$ *for any operator* $m \in$ `Moves`*, by cases over* m*, as follows.*

  - $\langle \Gamma \vdash D, i\rangle + \mathtt{f}.A = \langle \Gamma, A \vdash D, n+1\rangle$ *(the sequent has active formula A, of sign* $\mathtt{f}$.*, and index $n+1$).*
  - $\langle \Gamma \vdash D, i\rangle + \mathtt{t}.A = \langle \Gamma \vdash A, n+1\rangle$ *(the sequent has active formula A, of sign* $\mathtt{t}$.*, and index $n+1$).*
  - $\langle \Gamma \vdash D, i\rangle + \mathtt{bck}(j) = \langle \Gamma \vdash D, j\rangle$ *if j is an index of $\Gamma \vdash D$, and* $= \langle \Gamma \vdash D, n\rangle$ *o.w..*

  The sequent $\langle \Gamma \vdash D, i\rangle + sA$ has the same indexing of $\langle \Gamma \vdash D, i\rangle$, plus the index $n+1$ for A, and minus the index of D if $s =$ $\mathtt{t}$.*. The sequent $\langle \Gamma \vdash D, i\rangle + \mathtt{bck}(j)$ has the same indexing of $\langle \Gamma \vdash D, i\rangle$*

4. $\langle \Gamma \vdash D, i\rangle + \langle m_1, \ldots, m_k\rangle = (\ldots(\langle \Gamma \vdash D, i\rangle + m_1) + \ldots) + m_k$.

We denote the active formula of a sequent boldface, for instance: $\Gamma, \mathbf{A} \vdash D$ for a negative occurrence of A, and $\Gamma, A \vdash \mathbf{D}$ for a positive occurrence of D. Any two sequents over $\mathtt{L}_{\mathtt{HA0}}$, differing only for the active occurrence, like $\Gamma, \mathbf{A} \vdash D$ and $\Gamma, A \vdash \mathbf{D}$, will be equivalent in our game semantics (by Theorem 3). This is not always the case for a sequent of $\mathtt{L}_{\mathtt{G0}}$.

3 Game Semantics for Arithmetical Formulas

In this section we define our game semantics for formulas of $\mathtt{L}_{\mathtt{G0}}$. In the next section we unfold our definition and discuss its consequences.

We interpret (pointed) sequents $\langle \Gamma \vdash D, i\rangle$ as games are between two sides, $\mathcal{E}$ and $\mathcal{A}$. $\mathcal{E}$ is a single, finite and fallible being we call Eloise, able to learn from her mistakes. $\mathcal{A}$ is a potentially infinite array of omniscient, infallible beings we call the Abelard's, one for each move of the play. The omniscience of the Abelard's compensates the ability of Eloise to learn from her mistakes. Having one Abelard for each move of the game, instead, is just a colorful way of saying that Abelard decides his next move by considering only the previous move of the play. Therefore the replies of Abelard to two different moves are independent each other, and we can imagine they are made by two different individuals. Eloise can use all previous moves of the play to decide her next move, therefore her moves are related each other and we image them as made by a single individual.

The play between $\mathcal{E}, \mathcal{A}$ is interpreted as a debate. The play between $\mathcal{E}$ and each Abelard is called a "thread" in the play (we use the word "thread" with its informal meaning in Computer Science). In each position of the play, $\mathcal{E}$ defends a thesis, and one Abelard attacks it, or vice versa. Moves done in defence of a thesis cannot be retracted, for all players. The weak player, $\mathcal{E}$, can retract finitely many times a move done while attacking a thesis of $\mathcal{A}$. The strong players, the Abelard's, can never retract a move (neither in attack, nor in defence). There is also a thesis for the whole play. $\mathcal{E}$ claims that some $\Gamma \vdash D$ is true, while the array of Abelard's claims that $\Gamma \vdash D$ is false. We interpret truth of $\Gamma \vdash D$ by the existence of a recursive winning strategy for $\mathcal{E}$ on the game associated to the pointed sequent $\langle \Gamma \vdash D, i\rangle$, for some i.

We first introduce plays and correct moves. Moves include moves of Tarski plays [9]. This kind of move is called a logical move, and denoted by a judgement sA. There is a new kind of moves, "backtracking", when $\mathcal{E}$ comes back to the

move number i of the play, retracts the move she did after it, and selects a new move, with the goal of learning better and better moves in this way. This kind of move is called a structural move, and denoted by $\mathtt{bck}(i)$. The idea of backtracking is taken from [4], [5], [2] (a similar idea can be found in [10], where it is used to interpret λ-terms). However, backtracking in our game semantics for intuitionism has a severe limitation: $\mathcal{E}$ can backtrack to any judgement having a negative sign, but only to the last judgement having a positive sign.

Definition 8 (*Moves, positions and plays*)

1. *The set of all moves is* `Moves` *(Def. 7.2). A logical move is any judgment* sA*. A structural move is any notation* $\mathtt{bck}(i)$*.*
2. $\mathtt{drop} \notin \mathtt{Moves}$ *is a special symbol, meaning: "I quit".*
3. *A position* Q *is any non-empty finite sequence over* `Moves`*, starting with some* $\mathtt{t}.A$*. We write* $P \leq Q$ *if* P *is a prefix of* Q*.*
4. *A finished play* P*, or just a "play" for short, is either a sequence* $Q, \mathtt{drop}$*, for some position* Q*, or an infinite sequence over* `Moves`*, starting with some* $\mathtt{t}.A$*.*
5. *If* Q *is any position, the indexing of logical moves (or judgements) in* Q *is:* 1 *for the first logical move,* 2 *for the second one, and so forth.*

Informally, we could describe a position as an "unfinished play". We define now the player $\mathcal{E}/\mathcal{A}$ moving from a given position Q, and the set of correct moves from Q. Only $\mathcal{E}$ can move $\mathtt{bck}(i)$, that is, only $\mathcal{E}$ can "backtrack" to the judgement number i of Q, with the further limitation that if i is the index of a positive judgement, this judgement has to be the last positive judgement of Q. Since $\mathcal{E}$ can come back to the logical move number i, the last logical move for $\mathcal{E}$ is not, necessarily, the last logical move of the sequence. We call the last logical move for $\mathcal{E}$: the *active move* of the position. For each position Q we define: the backtracking indexes, the active move and its index, the next player moving and its correct moves.

Definition 9 (*Correct moves*). *Let* P *be any position, having* $n > 0$ *logical moves.*

1. i *is a backtracking index of* P*, a bck-index for short, if: :* (i) $1 \leq i \leq n$*;* (ii) *if the* i*-th judgement is positive, then the* i*-th judgement is the* last *positive judgement of* P*.*
2. *If the last move of* P *is* sA*, the active move of* P *is* sA*, of index* n*.*
3. *If the last move of* P *is* $\mathtt{bck}(i)$*, the active move of* P *is the logical move of index* i *of* P *if* i *is a bck-index of* P*, the logical move of index* n *o.w..*
4. *If the active move of* P *is a disjunctive judgement, the player moving from* P *is* $\mathcal{E}$*.*
5. *If the active move of* P *is a conjunctive judgement, the player moving from* P *is* $\mathcal{A}$*.*
6. *If* p *is the player moving from* P*, and* sA *the active move, the correct moves of* p *are:*
   - $\mathtt{drop}$ *(to drop out from the game)*
   - *any* $s'A' <_1 sA$*,*
   - *if* $p = \mathcal{E}$*, also* $\mathtt{bck}(i)$*, for any* i *bck-index of* P*.*

7. *Q is a correct extension of P if $Q \geq P$, and for any $P \leq R@\langle m \rangle \leq Q$, the move m is correct from R.*

If the active move of P is $\mathtt{t}.A$, we say that $\mathcal{E}$ defends, in P, the thesis A, while some Abelard attacks the thesis A. If the active move of P is $\mathtt{f}.A$, we say that $\mathcal{E}$ attacks, in P, the thesis A while some Abelard defends the thesis A. We now define the winner and the loser for any (finished) play P.

Definition 10 (*loser and winner of a play*). *Let P be any play.*

1. *If $P = Q, \mathtt{drop}$, and p is the player moving from Q, then p loses in P, and its opponent wins.*
2. *If P is infinite, then $\mathcal{E}$ loses in P, and $\mathcal{A}$ wins.*

We allow a play to use any position P as initial segment. The game associated G_P to a position P is the set of all positions we can reach from P using only correct moves of the next player. Positions and sequents are interchangeable notions. Any position P is associated to some sequent $\mathtt{seq}(P) = \langle \Gamma \vdash D, i \rangle$ (see the definition below). $\mathtt{seq}(P)$ is defined by interpreting each move m in P as an operation building a sequent, by Def. 7.4. We imagine that, in the position P, $\mathcal{E}$ claims that $\langle \Gamma \vdash D, i \rangle$ is true, and $\mathcal{A}$ claims it is false. Conversely, any pointed sequent $\langle \Gamma \vdash D, i \rangle$ is associated to some canonical position $P = \mathtt{pos}(\Gamma \vdash D, i)$, and to some game $G(\Gamma \vdash D, i)$ having P as initial position.

Definition 11. *Assume $P = \mathtt{t}.A, m_1, \ldots, m_k$ is any position with $n > 0$ logical moves. Let $\langle \Gamma \vdash D, i \rangle$ be any pointed sequent, with canonical indexing $\langle \Gamma' \vdash D', i' \rangle$, and $\Gamma = A_1, \ldots, A_n$.*

1. *The* game G_P associated to P *is the set of all correct extensions of P.*
2. *The* sequent $\mathtt{seq}(P)$ associated to P *is $\langle \emptyset \vdash A, 1 \rangle + m_1 + \ldots + m_k$ (see Def. 7.4).*
3. *The* position $\mathtt{pos}(\Gamma \vdash D, i)$ associated to $\langle \Gamma \vdash D, i \rangle$ *is: $\mathtt{t}.D, \mathtt{f}.A_1, \ldots, \mathtt{f}.A_n$ if $i' = n + 1$, and $= \mathtt{t}.D, \mathtt{f}.A_1, \ldots, \mathtt{f}.A_n, \mathtt{bck}(i')$ if $1 \leq i' \leq n$.*
4. *If $P = \mathtt{pos}(\Gamma \vdash D, i)$, then $G(\Gamma \vdash D, i) = G_P$. $G(D) = G(\vdash D, 1)$.*

By definition unfolding, we can check that "taking the associated position" and "taking the associated sequent" are two operations inverse each other (up to index renaming).

Lemma 1. *Let $\langle \Gamma' \vdash D', i' \rangle$ be the canonical indexing of the sequent $\langle \Gamma \vdash D, i \rangle$, and $P = \mathtt{pos}(\Gamma \vdash D, i)$ be the position associated to $\langle \Gamma \vdash D, i \rangle$. Then $\mathtt{seq}(P) = \langle \Gamma' \vdash D', i' \rangle$ (i.e., "the sequent associated to the position associated to a sequent is the sequent itself", up to index renaming).*

The initial position of the game $G(\Gamma \vdash D, i)$ is $P = \mathtt{pos}(\Gamma \vdash D, i)$. In the initial position of $G(\Gamma \vdash D, i)$, $\mathcal{E}$ claims that $\mathtt{seq}(P) = \langle \Gamma' \vdash D', i' \rangle$ is true, while $\mathcal{A}$ claims it is false. The initial position of $G(A)$ is $\mathtt{pos}(\vdash A, 1) = \mathtt{t}.A$. We will now define winning strategies for $\mathcal{E}$ on a game as particular recursive trees. We first define a coding for recursive trees, and an indexing Children(x) for the children of a node x.

Definition 12 (*Coding recursive trees*)

- *A tree T over M is any set of finite lists over M, including the empty list $\langle\rangle$ and closed under prefix.*
- *If $x \in T$, then Children$(x) = \{m \in M | x@\langle m\rangle \in T\}$ is an indexing for the children of a node x.*
- *A tree is recursive if it is coded by a recursive set.*
- *A labeling over a tree is any map $\mathtt{l} : T \to I$ assigning some label $l(x) \in I$ to each $x \in T$.*

From now on, we always code trees by sets of lists. If x, y are lists, we denote the concatenation of x, y by $x@y$. We define winning strategies for $\mathcal{E}$ on a game G_P as particular recursive well-founded trees σ on the set `Moves`. Here is how σ works. During the play, the current position of the play is always $P@x$, for some $x \in \sigma$. σ is defined in such a way that, whenever $\mathcal{E}$ moves from $P@x$, there is exactly one child $x@\langle m\rangle \in \sigma$ of the current node $x \in \sigma$. $m \in$ `Moves` is the correct move $\neq$ `drop` suggested by σ to $\mathcal{E}$ from $P@x$. Since $m \in$ `Moves`, then $\mathcal{E}$ cannot play `drop`. Whenever $\mathcal{A}$ moves, instead, the children y of x are all $x@\langle m\rangle \in$ `Moves` which are correct moves $\neq$ `drop` of $\mathcal{A}$ from $P@x$. $\mathcal{E}$ chooses the child $x@\langle m\rangle$ corresponding to the actual move m by $\mathcal{A}$ (unless $\mathcal{A}$ plays `drop`, in which case play ends). Since σ is well-founded, after finitely many moves the play ends. The loser is necessarily $\mathcal{A}$, because $\mathcal{A}$ is the only player who can play `drop`.

Definition 13 (*Recursive winning strategies for $\mathcal{E}$ on G_P*). *Fix any position P. Let σ be any tree over the set* `Moves`.

1. *For any $x \in T$, we call $\mathtt{seq}_P(x) = \mathtt{seq}(P@x)$ the sequent labeling x.*
2. *σ is a strategy for $\mathcal{E}$ on G_P if for all $x \in T$, if p is the player moving from $P@x$:*
   - *if $p = \mathcal{E}$, then Children$(x) = \{m\}$, for some correct move $m \neq$ `drop` from $P@x$.*
   - *if $p = \mathcal{A}$, then Children(x) is the set of all correct moves $\neq$ `drop` from $P@x$.*
3. *σ is winning if it is a well-founded tree. σ is recursive if it is a recursive tree.*

We can now define validity in our game semantics.

Definition 14. *Let $\langle \Gamma \vdash D, i\rangle$ be any sequent of* $\mathtt{L_{G0}}$.

1. *$\sigma \models \langle \Gamma \vdash D, i\rangle$ if σ is a recursive winning strategy for $\mathcal{E}$ on $G(\Gamma \vdash D, i)$.*
2. *$\mathcal{G} \models \langle \Gamma \vdash D, i\rangle$ if $\exists\sigma.\sigma \models \langle \Gamma \vdash D, i\rangle$.*
3. *$\mathcal{G} \models D$ if $\mathcal{G} \models \langle \vdash D, 1\rangle$.*

We can characterize a strategy on any game G_P as follows:

Lemma 2. *σ is a recursive winning strategy for $\mathcal{E}$ on G_P if and only if σ is a recursive well-founded tree, and, for all $x \in \sigma$, if the active move of $P@x$ is sA, then:*

- *either sA is disjunctive and $Children(x) = \{\mathtt{bck}(j)\}$, for some j bck-index of $P@x$,*
- *or sA is disjunctive and $Children(x) = \{s'A'\}$, for some $s'A' <_1 sA$,*
- *or sA is conjunctive and $Children(x) = \{s'A' | s'A' <_1 sA\}$.*

Proof. By unfolding Def. 13, 9.

In §6 we will prove that for all $D \in \mathtt{L_{HA0}}$ we have $\mathcal{G} \models D$ if and only if D is a theorem of $\mathtt{HA}_{\mathrm{inf}}$, Infinitary Intuitionistic Arithmetic with ω-rule.

4 $\mathtt{HA}_{\mathrm{inf}}$, Intuitionistic Arithmetic with ω-Rule

In this section we introduce $\mathtt{HA}_{\mathrm{inf}}$, an infinitary sequent calculus (with pointed sequents) for Intuitionistic Arithmetic. We also introduce an infinitary logic $\mathcal{G}_{\mathrm{inf}}$ for deriving validity of formulas of $\mathtt{L_{G0}}$ in our game semantics. Proofs in $\mathcal{G}_{\mathrm{inf}}$ are isomorphic to winning strategies (Theorem 2). Eventually, in §6 we prove that $\mathcal{G}_{\mathrm{inf}}$ is a conservative extension of $\mathtt{HA}_{\mathrm{inf}}$, and we conclude that our game semantics is sound and complete for $\mathtt{HA}_{\mathrm{inf}}$.

The language of $\mathtt{HA}_{\mathrm{inf}}$ is $\mathtt{L_{HA0}}$ and the language for $\mathcal{G}_{\mathrm{inf}}$ is $\mathtt{L_{G0}}$. We first define logical rules of $\mathcal{G}_{\mathrm{inf}}$ and $\mathtt{HA}_{\mathrm{inf}}$ a synthetic way, using the operation $\langle \Gamma \vdash D, i\rangle + sA$ from Def. 7. Then we unfold our definition, in order to check that it is equivalent to the usual one for $\mathtt{HA}_{\mathrm{inf}}$.

Definition 15

- *(Logical Rules for $\mathtt{HA}_{inf}$) Assume $\langle \Gamma \vdash D, i\rangle$ is a sequent of $\mathtt{L_{HA0}}$, with active formula A of sign s. Let $s'A' <_1 sA$ in $\mathtt{L_{HA0}}$. Then a logical rule of conclusion $\langle \Gamma \vdash D, i\rangle$ has premise* one *$\langle \Gamma \vdash D, i\rangle + s'A'$ if sA is disjunctive, and* all *$\langle \Gamma \vdash D, i\rangle + s'A'$ if sA is conjunctive.*
- *(Logical Rules for $\mathcal{G}_{inf}$). The definition is obtained from the definition for $\mathtt{HA}_{inf}$, by replacing the language $\mathtt{L_{HA0}}$ with the language $\mathtt{L_{G0}}$.*

In $\mathcal{G}_{\mathrm{inf}}$, for each sequent there is (at most) one logical rule having conclusion this sequent. In the definition above, there are 11 possible cases for A: A true or false atomic formula, or A starting any of the 9 connectives of $\mathtt{L_G}$. There are 2 possible signs. Therefore there are at most $11 \times 2 = 22$ possible cases for logical rules. There is no logical rule, however, if sA is disjunctive and there is no $s'A' <_1 sA$, because a logical rule for a disjunctive sA requires one $s'A' <_1 sA$. The are only two cases of this kind: $sA = \mathtt{t}.F, \mathtt{f}.T$, both disjunctive and without immediate sub-judgements. Therefore, if we unfold the definition of logical rule, we obtain $22 - 2 = 20$ cases for a logical rule in $\mathcal{G}_{\mathrm{inf}}$[1]:

Definition 16 (*Logical Rules for $\mathcal{G}_{inf}$*). *We write the active occurrence of any sequent boldface.*

[1] In Def. 16 there are, in fact, 21 different patterns, because there are two patterns in the case A is an implication $B \to C$, and A is in the right-hand-side of a sequent.

– (Rules for F, T)

$$\frac{}{\Gamma, \mathbf{F} \vdash D} \qquad \frac{}{\Gamma \vdash \mathbf{T}}$$

– (Atomic logical rules) *For any a atomic false, and any b atomic true (true and false in the standard model N):*

$$\frac{\Gamma, a, \mathbf{F} \vdash D}{\Gamma, \mathbf{a} \vdash D} \qquad \frac{\Gamma \vdash \mathbf{F}}{\Gamma \vdash \mathbf{a}} \qquad \frac{\Gamma, b, \mathbf{T} \vdash D}{\Gamma, \mathbf{b} \vdash D} \qquad \frac{\Gamma \vdash \mathbf{T}}{\Gamma \vdash \mathbf{b}}$$

– (Conjunctive logical rules for $\wedge, \forall, \vee, \exists$)

$$\frac{\Gamma \vdash \mathbf{C_i} \ \textit{(for } i = 1, 2\textit{)}}{\Gamma \vdash \mathbf{C_1} \wedge \mathbf{C_2}} \qquad \frac{\Gamma \vdash \mathbf{A}[\mathbf{t}] \ \textit{(for all closed terms } t\textit{)}}{\Gamma \vdash \forall \mathbf{x}.\mathbf{A}}$$

$$\frac{\Gamma, C_1 \vee C_2, \mathbf{C_i} \vdash D \ \textit{(for } i = 1, 2\textit{)}}{\Gamma, \mathbf{C_1} \vee \mathbf{C_2} \vdash D} \qquad \frac{\Gamma, \exists x.A, \mathbf{A}[\mathbf{t}] \vdash D \ \textit{(for all closed terms } t\textit{)}}{\Gamma, \exists \mathbf{x}.\mathbf{A} \vdash D}$$

– (Disjunctive logical rules for $\wedge, \forall, \vee, \exists$). *Let $i = 1$ or $i = 2$, and t be any closed term.*

$$\frac{\Gamma \vdash \mathbf{C_i}}{\Gamma \vdash \mathbf{C_1} \vee \mathbf{C_2}} \qquad \frac{\Gamma \vdash \mathbf{A}[\mathbf{t}]}{\Gamma \vdash \exists \mathbf{x}.\mathbf{A}}$$

$$\frac{\Gamma, C_1 \wedge C_2, \mathbf{C_i} \vdash D}{\Gamma, \mathbf{C_1} \wedge \mathbf{C_2} \vdash D} \qquad \frac{\Gamma, \forall x.A, \mathbf{A}[\mathbf{t}] \vdash D}{\Gamma, \forall \mathbf{x}.\mathbf{A} \vdash D}$$

– (Logical rules of implication $\rightarrow$) *The logical rule for $A \rightarrow B$ in the right-hand-side has two sub-cases.*

$$\frac{\Gamma, \mathbf{A} \vdash A \rightarrow B}{\Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}} \qquad \frac{\Gamma \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}} \qquad \frac{\Gamma, A \rightarrow B \vdash \mathbf{A} \quad \Gamma, A \rightarrow B, \mathbf{B} \vdash D}{\Gamma, \mathbf{A} \rightarrow \mathbf{B} \vdash D}$$

– (Logical rules of intuitionistic implication $\Rightarrow$)

$$\frac{\Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}}{\Gamma \vdash \mathbf{A} \Rightarrow \mathbf{B}} \qquad \frac{\Gamma, A \Rightarrow B, \mathbf{A} \rightarrow \mathbf{B} \vdash D}{\Gamma, \mathbf{A} \Rightarrow \mathbf{B} \vdash D}$$

– (Logical rules of negation)

$$\frac{\Gamma, \mathbf{A} \vdash \neg A}{\Gamma \vdash \neg \mathbf{A}} \qquad \frac{\Gamma, \neg A \vdash \mathbf{A}}{\Gamma, \neg \mathbf{A} \vdash D}$$

We can unfold the logical rules for $\mathtt{HA}_{\text{inf}}$ in a similar way. The only difference between $\mathtt{HA}_{\text{inf}}$ and $\mathcal{G}_{\text{inf}}$ is that in $\mathtt{HA}_{\text{inf}}$ the rules for $A \rightarrow B$ are skipped, and the rules for $A \Rightarrow B$ have hypotheses with active formulas A, B, because the immediate subformulas of $A \Rightarrow B$ in $\mathtt{L}_{\mathtt{HA}}$ are A, B:

$$\frac{\Gamma, \mathbf{A} \vdash A \Rightarrow B}{\Gamma \vdash \mathbf{A} \Rightarrow \mathbf{B}} \qquad \frac{\Gamma \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \Rightarrow \mathbf{B}} \qquad \frac{\Gamma, A \Rightarrow B \vdash \mathbf{A} \quad \Gamma, A \Rightarrow B, \mathbf{B} \vdash D}{\Gamma, \mathbf{A} \Rightarrow \mathbf{B} \vdash D}$$

We consider only one structural rule both for $\mathtt{HA}_{\mathrm{inf}}$ and $\mathcal{G}_{\mathrm{inf}}$, Exchange, switching the active formula of a sequent.

Definition 17 (*Structural Rule for* $\mathtt{HA}_{inf}$ *and* $\mathcal{G}_{inf}$*: Exchange*). *Let* $\Gamma \vdash D$ *be any sequent, and* i, j *any two occurrences of formulas of* $\Gamma \vdash D$.

- *The* Exchange *rule, or* E*-rule for short, is the following: derive* $\langle \Gamma \vdash D, j\rangle$ *from* $\langle \Gamma \vdash D, j\rangle + \mathtt{bck}(i)$.
- DisjE, ConjE, $\mathrm{ConjE}_{\rightarrow}$*-rules are the restriction of the E-rule, when the occurrence* j *is, respectively: disjunctive, conjunctive, conjunctive and equal to* $B_1 \rightarrow B_2$ *for some* B_1, B_2.
- *The Exchange rules for* $\mathtt{HA}_{inf}, \mathcal{G}_{inf}$ *are E and ConjE.*

By unfolding the definition of $\langle \Gamma \vdash D, j\rangle + \mathtt{bck}(i)$, we can write the E-rule as follows: for any two occurrences i, j of $\Gamma \vdash D$,

$$\frac{\langle \Gamma \vdash D, i\rangle}{\langle \Gamma \vdash D, j\rangle}$$

E-rule says that two sequents differing only for the active formula are equivalent. In §5, we will prove that ConjE is conditionally derivable in $\mathcal{G}_{\mathrm{inf}}$ from DisjE for all sequents of $\mathtt{L}_{\mathtt{HA}}$ (i.e., for all sequents without $\rightarrow$). We will deduce that $\mathcal{G}_{\mathrm{inf}}$ is a conservative extension of $\mathtt{HA}_{\mathrm{inf}}$. Besides, strategies in our game semantics and proofs of $\mathcal{G}_{\mathrm{inf}}$ can be identified (see Theorem 2). By combining the two remarks, we will conclude that our game semantics are sound and complete for $\mathtt{HA}_{\mathrm{inf}}$.

Proofs of $\mathtt{HA}_{\mathrm{inf}}$ (of $\mathcal{G}_{\mathrm{inf}}$) are all well-founded recursive trees, labeled with sequents, and such that the sequent labeling each node is the conclusion of some rule of $\mathtt{HA}_{\mathrm{inf}}$ (of $\mathcal{G}_{\mathrm{inf}}$). We code proofs as trees over `Moves`, in order to stress the similarity between proofs of $\mathcal{G}_{\mathrm{inf}}$ and winning strategies.

Definition 18. *Fix any sequent* $S = \langle \Gamma \vdash D, i\rangle$ *of* $\mathtt{L}_{\mathtt{G0}}$ *(of* $\mathtt{L}_{\mathtt{HA0}}$*). Assume* Π *is any well-founded, recursive tree over* `Moves`.

1. *The labeling of* Π *with sequents is, for all* $x = \langle m_1, \ldots, m_k\rangle \in \Pi$*:* $\mathtt{seq}_S(x) = S + m_1 + \ldots + m_k$ *(see Def. 7.4).*
2. Π *is a proof of* S *in* $\mathtt{HA}_{inf}$ *(in* $\mathcal{G}_{inf}$*) if for all* $x \in \Pi$*, there is some rule of* $\mathtt{HA}_{inf}$ *(in* $\mathcal{G}_{inf}$*) whose conclusion is:* $\mathtt{seq}_S(x)$*, and whose assumptions are:* $\{\mathtt{seq}_S(x@\langle m\rangle) \mid x@\langle m\rangle \in \Pi\}$.
3. S *is provable in* $\mathtt{HA}_{inf}$ *(in* $\mathcal{G}_{inf}$*) if there is some proof* Π *of* S *in* $\mathtt{HA}_{inf}$ *(in* $\mathcal{G}_{inf}$*).*

We unfold the definition above, in order to explain how we code each rule of $\mathtt{HA}_{\mathrm{inf}}$ and $\mathcal{G}_{\mathrm{inf}}$. The root of Π is coded $\langle\rangle$. Assume $x \in \Pi$, and the sequent $\mathtt{seq}(x)$ labeling x has active move sA. If x is the conclusion of some logical rule, then the children of x in Π are coded by all (by some) $x@\langle s'A'\rangle$, for $s'A' <_1 sA$, according if x is conjunctive or disjunctive. If x is the conclusion of some structural rule, then the children of x in Π are coded by some $x@\langle \mathtt{bck}(i)\rangle$.

Introduction rule for $\rightarrow$ from Natural Deduction (if $\Gamma, A \vdash \mathbf{B}$, then $\Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}$) is conditionally derivable in $\mathtt{HA}_{\rm inf}$[2]. All other structural rules are derivable in $\mathtt{L_{HA}}$: identity, weakening, exchange (for all formulas), contraction in the left-hand side and cut. For instance, all previous rules in the left-hand side implicitly include contraction. Indeed, from $C_1 \wedge C_2, C_1 \vdash D$ we infer $C_1 \wedge C_2 \vdash D$, instead of $C_1 \wedge C_2, C_1 \wedge C_2 \vdash D$. We can characterize proofs of $\mathcal{G}_{\rm inf}$.

Lemma 3. *Let Π be any well-founded recursive tree over* `Moves`*. Assume $S = \langle \Gamma \vdash D, i\rangle$. Then Π is a proof of S in $\mathcal{G}_{inf}$ if and only if, for all $x \in \Pi$, if the active formula A of $\mathtt{seq}_S(x)$ has sign s, then:*

- *either sA is disjunctive and Children$(x) = \{\mathtt{bck}(j)\}$, for some index j of $\mathtt{seq}_S(x)$;*
- *or sA is disjunctive and Children$(x) = \{s'A'\}$, for some $s'A' <_1 sA$.*
- *or sA is conjunctive and Children$(x) = \{s'A' | s'A' <_1 sA\}$.*

Proof. By definition unfolding, and because the only structural rule of $\mathcal{G}_{\rm inf}$ is ConjE.

We will prove in Theorem 3 that $\mathcal{G}_{\rm inf}$ is a conservative extension of $\mathtt{HA}_{\rm inf}$. In §3, from this fact we will derive that our game semantic is sound and complete for $\mathtt{HA}_{\rm inf}$.

5 Conjunctive Structural Rule Is Conditionally Derivable

In this section we state the fact that ConjE (conjunctive structural rule, §4) is derivable from the other rules of $\mathcal{G}_{\rm inf}$, for all sequents of $\mathtt{L_{HA}}$ (i.e., $\rightarrow$-free). We will deduce that $\mathcal{G}_{\rm inf}$ is a conservative extension of $\mathtt{HA}_{\rm inf}$. For reason of spaces, proofs are omitted, but they can be found in [1], §7.

Theorem 1 (*ConjE is derivable from DisjE in $\mathcal{G}_{inf}$*). *If $\mathcal{G}_{inf}$ + ConjE proves $\langle \Gamma \vdash A, i\rangle$, and $\Gamma \vdash A$ is in $\mathtt{L_{HA0}}$, the $\mathcal{G}_{inf}$ proves $\langle \Gamma \vdash A, i\rangle$.*

We end this section with one corollary, $\mathcal{G}_{\rm inf}$ is a conservative extension of $\mathtt{HA}_{\rm inf}$, and one easy remark, provability and validity are invariant under index renaming.

Corollary 1

1. *$\mathcal{G}_{inf}$ + ConjE is a conservative extension of $\mathtt{HA}_{inf}$*
2. *$\mathcal{G}_{inf}$ is a conservative extension of $\mathtt{HA}_{inf}$.*

Lemma 4 (*Index renaming*). *Assume $\langle \Gamma \vdash D, i\rangle$ is any sequent of $\mathtt{L_{G0}}$, $\langle \Gamma' \vdash D', i'\rangle$ is its canonical indexing.*

1. *$\langle \Gamma \vdash D, i\rangle$ is provable in $\mathtt{HA}_{inf}$ if and only if $\langle \Gamma' \vdash D', i'\rangle$ is provable in $\mathtt{HA}_{inf}$*
2. *$\mathcal{G} \models \langle \Gamma \vdash D, i\rangle$ if and only if $\mathcal{G} \models \langle \Gamma' \vdash D', i'\rangle$.*

[2] *Proof.* If $\Gamma, A \vdash \mathbf{B}$, then $\Gamma, A \vdash \mathbf{A} \rightarrow \mathbf{B}$ by the right rule for $A \rightarrow B$. We obtain $\Gamma, \mathbf{A} \vdash A \rightarrow B$ by Exchange, and we conclude $\Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}$ again by the right rule for $A \rightarrow B$.

6 An Isomorphism Between Proofs and Strategies

In this section we prove, as Lorentzen did for his game semantics [6], that our game semantics for $\mathtt{HA}_{\mathrm{inf}}$ is sound and complete. We actually prove more, namely that winning strategies of the semantics and proofs of the variant $\mathcal{G}_{\mathrm{inf}}$ of $\mathtt{HA}_{\mathrm{inf}}$ can be identified. This improvement of Lorentzen's result is inspired by the isomorphism between classical proofs and strategies defined in ([8]).[3] A sample of this identification can be seen in [1], §9. We start proving a relation between positions of a game and the corresponding sequents.

Lemma 5. *Let P be any position. Let s_1A_1, ..., s_nA_n be the list of judgements in P. Assume $\langle \Gamma \vdash D, i\rangle = \mathtt{seq}(P)$. Then:*

1. *(j index of $\mathtt{seq}(P)$) $\Leftrightarrow$ (j backtracking position of P).*
2. *$P, \mathtt{seq}(P)$ have the same maximum index.*
3. *$\Gamma = \{\langle k, A_k\rangle | s_k = \mathtt{f}.\}$ (= all negative judgements, with the index they have in P).*
4. *$D = \{\langle k, A_k\rangle\}$, for the last k such that $s_k = \mathtt{t}$. (= the last positive judgement, with its index in P).*
5. *$i =$ the index of the active move of P.*

Proof. We will prove point $3, 4, 5$ by simultaneous induction over the length of P. Points $1, 2$ are implied by points $3, 4$. Indeed, the set of bck-indexes of P consists of all indexes of negative judgement, and of the last positive judgement, and by $3, 4$, of all indexes of $\mathtt{seq}(P)$. $P, \mathtt{seq}(P)$ have the same maximum index, because the largest index of P is equal to the largest bck-index of P.

If $P = \mathtt{t}.A$, then $3, 4, 5$ are immediate from the definition of $\mathtt{seq}$. Assume $P = Q, m$, for some position Q with $n > 0$ judgements, and some $m \in \mathtt{Moves}$. Let $n =$ the maximum index of $\mathtt{seq}(Q) =$ (by ind.hyp. on Q) the maximum index of Q.

Assume $m = sA$. Then sA has index $n + 1$ in P, $\mathtt{seq}(P)$. If $s = \mathtt{f}.$, then sA is added to the left-hand-side of $\mathtt{seq}(Q)$, and is added to the negative judgements of Q. If $s = \mathtt{t}.$, then sA is replaced to the right-hand-side of $\mathtt{seq}(Q)$, and is replaced to the last positive judgement of Q. We deduce $3, 4$. $n + 1$ is the active occurrence of $\mathtt{seq}(P)$ and the active move of P. We deduce 5.

Assume $m = \mathtt{bck}(j)$. Then $\mathtt{seq}(P), \mathtt{seq}(Q)$ have the same left- and right- hand side, and P, Q have the same sub-list of judgements. Since Q satisfies $3, 4$, then P satisfies $3, 4$. The active occurrence of $\mathtt{seq}(P)$ is j if and only if j is an index of $\mathtt{seq}(Q)$, it is n o.w.. The active move of P is j if and only if j is a bck-index of Q, it is n o.w.. By ind. hyp. on Q, we deduce that the active occurrence of $\mathtt{seq}(P)$ and the active move of P are the same. We conclude 5.

Isomorphism Theorem says that recursive winning strategies for $\mathcal{E}$ and proofs of $\mathcal{G}_{\mathrm{inf}}$ are isomorphic, and in fact, with the coding we have chosen, identical.

Theorem 2 (*Isomorphism theorem*). *Fix any pointed sequent $S = \langle \Gamma \vdash A, i\rangle$ having a canonical indexing and $n > 0$ formulas. Then:*

[3] Another proof-strategy isomorphism result of the same kind can be found in [3].

(Π is a proof of $\langle \Gamma \vdash A, i\rangle$ in $\mathcal{G}_{inf}$) $\Leftrightarrow$ (Π is a recursive winning strategy for $\mathcal{E}$ on $G(\Gamma \vdash A, i)$)

Proof. Let $P = \texttt{pos}(\Gamma \vdash D, i)$. By Lemma 1, and the fact that $S = \langle \Gamma \vdash D, i\rangle$ has a canonical indexing, we have $\texttt{seq}(P) = S$. By Lemma 3, Π is a proof of S in $\mathcal{G}_{\text{inf}}$ if and only if Π is a recursive well-founded tree, and, for all $x \in \Pi$, if $\texttt{seq}_S(x)$ has active formula A of sign s, then

1. either sA is disjunctive and Children$(x) = \{\texttt{bck}(j)\}$ for some j index of $\texttt{seq}_S(x)$;
2. or sA is disjunctive and Children$(x) = \{s'A'\}$, for some $s'A' <_1 sA$;
3. or sA is conjunctive and Children$(x) = \{s'A' | s'A' <_1 sA\}$.

By definition, $\texttt{seq}_S(x) = S + x = \texttt{seq}(P) + x = \texttt{seq}(P@x)$. Therefore the predicate "j index of $\texttt{seq}_S(x)$" in the first clause is equivalent to "j index of $\texttt{seq}(P@x)$". By Lemma 5.1 applied to (j index of $\texttt{seq}(P@x)$), clause 1 above is equivalent to:

1. Children$(x) = \{\texttt{bck}(j)\}$ for some j bck-index of $P@x$.

By Lemma 5.5, the hypothesis "A active formula of $\texttt{seq}(P@x)$, of sign s" is equivalent to: "sA active move of $P@x$". By Lemma 2, the assumption about Π, if reformulated in this way, is equivalent to the definition of recursive winning strategy for $\mathcal{E}$ on $G_P = G(\Gamma \vdash D, i)$.

Soundness and Completeness are immediate from the Isomorphism Theorem.

Theorem 3 (*Completeness Theorem*). *Let $\langle \Gamma \vdash A, i\rangle$ be any pointed sequent of $\texttt{L}_{\texttt{HA0}}$. Then:*
($\texttt{HA}_{inf}$ proves $\langle \Gamma \vdash A, i\rangle$) $\Leftrightarrow \mathcal{G} \models \langle \Gamma \vdash A, i\rangle$.

Proof. In view of Lemma 4, we can assume that $\langle \Gamma \vdash D, i\rangle$ has a canonical indexing, so we can apply Theorem 2. Now suppose $\langle \Gamma \vdash A, i\rangle$ has a proof Π in $\texttt{HA}_{\text{inf}}$. By Corollary 1, $\langle \Gamma \vdash A, i\rangle$ has a proof Π' in $\mathcal{G}_{\text{inf}}$. By Theorem 2, Π' is also a recursive winning strategy for $\mathcal{E}$ on $G(\Gamma \vdash A, i)$. Suppose the converse: $\mathcal{E}$ has some recursive winning strategy σ on $G(\Gamma \vdash A, i)$. By Theorem 2 again, σ is a proof of $\langle \Gamma \vdash A, i\rangle$ in $\mathcal{G}_{\text{inf}}$, and by Corollary 1 we have a proof in in $\texttt{HA}_{\text{inf}}$.

As a consequence of the main Theorem, we check that our game semantics does not validate some true positive formula, while it validates the negation of all false positive formulas.

Corollary 2. *Let $A \in \texttt{L}_0^+$ (i.e., A is $\neg, \Rightarrow, \rightarrow$-free).*

1. *For some true A we have $\neg(\mathcal{G} \models A)$.*
2. *$\mathcal{G} \models A \vdash F$ if and only if A is false (in the standard model of N).*

Proof

1. Let $\forall y.P(x, y)$ be any non-decidable predicate, and $P^{\perp}$ be the complement of P. Then $A = \forall x.(\forall y.P(x, y) \vee \exists y.P^{\perp}(x, y))$ is an instance of Excluded Middle. A is true, but it cannot be proved in $\texttt{HA}_{\text{inf}}$. By Theorem 3 we conclude $\neg(\mathcal{G} \models A)$.

2. $A \vdash F$ is a theorem of $\mathtt{HA}_{\mathrm{inf}}$ if and only if it is a theorem of $\mathtt{PA}_{\mathrm{inf}}$, classical arithmetic with ω-rule, because the rules of $\mathtt{HA}_{\mathrm{inf}}$, $\mathtt{PA}_{\mathrm{inf}}$ for positive formulas in the left-hand-side are the same. By completeness of $\mathtt{HA}_{\mathrm{inf}}$ w.r.t. game semantics (Theorem 3), and of $\mathtt{PA}_{\mathrm{inf}}$ w.r.t. first order semantics (folklore), we conclude $\mathcal{G} \models A \vdash F$ if and only if $A \vdash F$ is true, that is, if and only if A is false.

References

[1] Berardi, S.: Semantics for Intuitionistic Arithmetic based on Tarski Games with retractable moves Technical Report, Turin University (January 2007) http://www.di.unito.it/~stefano/Berardi-GamesForIntuitionisticLogic-2007-01.pdf

[2] Berardi, S., Coquand, Th., Hayashi, S.: Games with 1-backtracking. In: Proceedings of GaLop, Edinburgh (April 2005)

[3] Berardi, S., Yamagata, Y.: A sequent calculus for 1-backtracking, Technical Report, Turin University, CL&C 2006. Submitted to the special issue of APAL for the congress. (December 2005), http://www.di.unito.it/~stefano/Yamagata-Berardi-report.pdf

[4] Coquand, T.: A semantics of evidence for classical arithmetic (preliminary version). In: Huet, G., Plotkin, G., Jones, C. (eds.) Proceedings of the Second Workshop on Logical Frameworks, Edinburgh, Edinburgh, Edinburgh (1991), http://www.dcs.ed.ac.uk/home/lego/html/papers.html

[5] Coquand, T.: A semantics of evidence for classical arithmetic. Journal of Symbolic Logic 60, 325–337 (1995)

[6] Felscher, W.: Lorentzen's game semantics Handbook of Philosophical Logic, 2nd edn., vol. 5, pp. 115–145. Kluwer Academic Publisher, the Netherlands (2002)

[7] Hayashi, S.: Can proofs be animated by games. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 11–22. Springer, Heidelberg (2005) (invited paper)

[8] Herbelin, H.: Sequents qu'on calcule: de l'interpretation du calcul des sequents comme calcul de lambda-termes et comme calcul de strategies gagnantes, Ph. D. thesis, University of Paris VII (1995)

[9] Hodges, W.: Logic and Games. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy (Winter 2004) http://plato.stanford.edu/archives/win2004/entries/logic-games/

[10] Hyland, M., Ong, L.: On full abstraction for PCF. Information and Computation 163, 285–408 (2000)

The Safe Lambda Calculus

William Blum and C.-H. Luke Ong

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, England
{william.blum,luke.ong}@comlab.ox.ac.uk

Abstract. Safety is a syntactic condition of higher-order grammars that constrains occurrences of variables in the production rules according to their type-theoretic order. In this paper, we introduce the *safe lambda calculus*, which is obtained by transposing (and generalizing) the safety condition to the setting of the simply-typed lambda calculus. In contrast to the original definition of safety, our calculus does not constrain types (to be homogeneous). We show that in the safe lambda calculus, there is no need to rename bound variables when performing substitution, as variable capture is guaranteed not to happen. We also propose an adequate notion of β-reduction that preserves safety. In the same vein as Schwichtenberg's 1976 characterization of the simply-typed lambda calculus, we show that the numeric functions representable in the safe lambda calculus are exactly the multivariate polynomials; thus conditional is not definable. Finally we give a game-semantic analysis of safety: We show that safe terms are denoted by *P-incrementally justified strategies*. Consequently pointers in the game semantics of safe λ-terms are only necessary from order 4 onwards.

1 Introduction

Background

The *safety condition* was introduced by Knapik, Niwiński and Urzyczyn at FoSSaCS 2002 [12] in a seminal study of the algorithmics of infinite trees generated by higher-order grammars. The idea, however, goes back some twenty years to Damm [6] who introduced an essentially equivalent[1] syntactic restriction (for generators of word languages) in the form of *derived types*. A higher-order grammar (that is assumed to be *homogeneously typed*) is said to be *safe* if it obeys certain syntactic conditions that constrain the occurrences of variables in the production (or rewrite) rules according to their type-theoretic order. Though the formal definition of safety is somewhat intricate, the condition itself is manifestly important. As we survey in the following, higher-order *safe* grammars capture fundamental structures in computation, offer clear algorithmic advantages, and lend themselves to a number of compelling characterizations:

[1] See de Miranda's thesis [8] for a proof.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 39–53, 2007.
© Springer-Verlag Berlin Heidelberg 2007

– *Word languages*. Damm and Goerdt [7] have shown that the word languages generated by order-n *safe* grammars form an infinite hierarchy as n varies over the natural numbers. The hierarchy gives an attractive classification of the semi-decidable languages: Levels 0, 1 and 2 of the hierarchy are respectively the regular, context-free, and indexed languages (in the sense of Aho [4]), although little is known about higher orders.

  Remarkably, for generating word languages, order-n *safe* grammars are equivalent to order-n pushdown automata [7], which are in turn equivalent to order-n indexed grammars [14,15].
– *Trees*. Knapik *et al.* have shown that the Monadic Second Order (MSO) theories of trees generated by *safe* (deterministic) grammars of every finite order are decidable[2].

  They have also generalized the equi-expressivity result due to Damm and Goerdt [7] to an equivalence result with respect to generating trees: A ranked tree is generated by an order-n *safe* grammar if and only if it is generated by an order-n pushdown automaton.
– *Graphs*. Caucal [5] has shown that the MSO theories of graphs generated[3] by *safe* grammars of every finite order are decidable. However, in a recent preprint [10], Hague *et al.* have shown that the MSO theories of graphs generated by order-n *unsafe* grammars are undecidable, but deciding their modal mu-calculus theories is n-EXPTIME complete.

Overview

In this paper, we aim to understand the safety condition in the setting of the lambda calculus. Our first task is to transpose it to the lambda calculus and pin it down as an appropriate sub-system of the simply-typed theory. A first version of the *safe lambda calculus* has appeared in an unpublished technical report [3]. Here we propose a more general and cleaner version where terms are no longer required to be homogeneously typed (see Section 2 for a definition). The formation rules of the calculus are designed to maintain a simple invariant: Variables that occur free in a safe λ-term have orders no smaller than that of the term itself. We can now explain the sense in which the safe lambda calculus is safe by establishing its salient property: No variable capture can ever occur when substituting a safe term into another. In other words, in the safe lambda calculus, it is *safe* to use capture-*permitting* substitution when performing β-reduction.

There is no need for new names when computing β-reductions of safe λ-terms, because one can safely "reuse" variable names in the input term. Safe lambda calculus is thus cheaper to compute in this naïve sense. Intuitively one would expect the safety constraint to lower the expressivity of the simply-typed lambda calculus. Our next contribution is to give a precise measure of the expressivity

[2] It has been recently been shown [19] that trees generated by *unsafe* deterministic grammars (of every finite order) also have decidable MSO theories.

[3] These are precisely the configuration graphs of higher-order pushdown systems.

deficit of the safe lambda calculus. An old result of Schwichtenberg [21] says that the numeric functions representable in the simply-typed lambda calculus are exactly the multivariate polynomials *extended with the conditional function*. In the same vein, we show that the numeric functions representable in the safe lambda calculus are exactly the multivariate polynomials.

Our last contribution is to give a game-semantic account of the safe lambda calculus. Using a correspondence result relating the game semantics of a λ-term M to a set of *traversals* [19] over a certain abstract syntax tree of the η-long form of M (called *computation tree*), we show that safe terms are denoted by *P-incrementally justified strategies*. In such a strategy, pointers emanating from the P-moves of a play are uniquely reconstructible from the underlying sequence of moves and the pointers associated to the O-moves therein: Specifically, a P-question always points to the last pending O-question (in the P-view) of a greater order. Consequently pointers in the game semantics of safe λ-terms are only necessary from order 4 onwards. Finally we prove that a η-long β-normal λ-term is *safe* if and only if its strategy denotation is (innocent and) *P-incrementally justified*.

2 The Safe Lambda Calculus

Higher-Order Safe Grammars

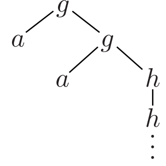
We first present the safety restriction as it was originally defined [12]. We consider simple types generated by the grammar $A ::= o \mid A \to A$. By convention, $\to$ associates to the right. Thus every type can be written as $A_1 \to \cdots \to A_n \to o$, which we shall abbreviate to $(A_1, \cdots, A_n, o)$ (in case $n = 0$, we identify (o) with o). The *order* of a type is given by $\mathsf{ord}(o) = 0$ and $\mathsf{ord}(A \to B) = \max(\mathsf{ord}(A) + 1, \mathsf{ord}(B))$. We assume an infinite set of typed variables. The order of a typed term or symbol is defined to be the order of its type.

A (higher-order) ***grammar*** is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, where Σ is a ranked alphabet (in the sense that each symbol $f \in \Sigma$ has an arity $ar(f) \geq 0$) of *terminals*[4]; $\mathcal{N}$ is a finite set of typed *non-terminals*; S is a distinguished ground-type symbol of $\mathcal{N}$, called the start symbol; $\mathcal{R}$ is a finite set of production (or rewrite) rules, one for each non-terminal $F : (A_1, \ldots, A_n, o) \in \mathcal{N}$, of the form $F z_1 \ldots z_m \to e$ where each z_i (called *parameter*) is a variable of type A_i and e is an applicative term of type o generated from the typed symbols in $\Sigma \cup \mathcal{N} \cup \{z_1, \ldots, z_m\}$. We say that the grammar is *order-n* just in case the order of the highest-order non-terminal is n.

The ***tree generated by a recursion scheme*** G is a possibly infinite applicative term, but viewed as a Σ-labelled tree; it is *constructed from the terminals in* Σ, and is obtained by unfolding the rewrite rules of G *ad infinitum*, replacing formal by actual parameters each time, starting from the start symbol S. See e.g. [12] for a formal definition.

[4] Each $f \in \Sigma$ of arity $r \geq 0$ is assumed to have type $(\underbrace{o, \cdots, o}_{r}, o)$.

Example 1. Let G be the following order-2 recursion scheme:



$$
\begin{aligned}
S &\rightarrow H\,a \\
H\,z^{o} &\rightarrow F\,(g\,z) \\
F\,\phi^{(o,o)} &\rightarrow \phi\,(\phi\,(F\,h))
\end{aligned}
$$

where the arities of the terminals g, h, a are $2, 1, 0$ respectively.
The tree generated by G is defined by the infinite term $g\,a\,(g\,a\,(h\,(h\,(h\,\cdots))))$.

A type $(A_1, \cdots, A_n, o)$ is said to be ***homogeneous*** if $\mathsf{ord}(A_1) \geq \mathsf{ord}(A_2) \geq \cdots \geq \mathsf{ord}(A_n)$, and each $A_1, \ldots, A_n$ is homogeneous [12]. We reproduce the following definition from [12].

Definition 1 (Safe grammar). (All types are assumed to be homogeneous.) A term of order $k > 0$ is *unsafe* if it contains an occurrence of a parameter of order strictly less than k, otherwise the term is *safe*. An occurrence of an unsafe term t as a subexpression of a term t' is *safe* if it is in the context $\cdots(ts)\cdots$, otherwise the occurrence is *unsafe*. A grammar is ***safe*** if no unsafe term has an unsafe occurrence at a right-hand side of any production.

Example 2. (i) Take $H : ((o,o),o)$, $f : (o,o,o)$; the following rewrite rules are unsafe (in each case we underline the unsafe subterm that occurs unsafely):

$$
\begin{aligned}
G^{(o,o)}\,x &\quad\rightarrow\quad H\,\underline{(f\,x)} \\
F^{((o,o),o,o,o)}\,z\,x\,y &\quad\rightarrow\quad f\,(F\,\underline{(F\,z\,y)}\,y\,(z\,x))\,x
\end{aligned}
$$

(ii) The order-2 grammar defined in Example 1 is unsafe.

Safety Adapted to the Lambda Calculus

We assume a set Ξ of higher-order constants. We use sequents of the form $\Gamma \vdash_{\Xi} M : A$ to represent terms-in-context where Γ is the context and A is the type of M. For simplicity we write $(A_1, \cdots, A_n, B)$ to mean $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$, where B is not necessarily ground.

Definition 2. (i) The ***safe lambda calculus*** is a sub-system of the simply-typed lambda calculus defined by induction over the following rules:

$$
(\mathsf{var})\ \frac{}{x : A \vdash_{\Xi} x : A} \qquad (\mathsf{const})\ \frac{}{\vdash_{\Xi} f : A}\ \ f \in \Xi \qquad (\mathsf{wk})\ \frac{\Gamma \vdash_{\Xi} s : A}{\Delta \vdash_{\Xi} s : A}\ \ \Gamma \subset \Delta
$$

$$
(\mathsf{app})\ \frac{\Gamma \vdash_{\Xi} s : (A_1, \ldots, A_n, B)\ \ \Gamma \vdash_{\Xi} t_1 : A_1\ \ \ldots\ \ \Gamma \vdash_{\Xi} t_n : A_n}{\Gamma \vdash_{\Xi} s t_1 \ldots t_n : B}\ \mathsf{ord}(B) \sqsubseteq \mathsf{ord}(\Gamma)
$$

$$
(\mathsf{abs})\ \frac{\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash_{\Xi} s : B}{\Gamma \vdash_{\Xi} \lambda x_1 \ldots x_n.s : (A_1, \ldots, A_n, B)}\ \mathsf{ord}(A_1, \ldots, A_n, B) \sqsubseteq \mathsf{ord}(\Gamma)
$$

where $\mathsf{ord}(\Gamma)$ denotes the set $\{\mathsf{ord}(y) : y \in \Gamma\}$ and "$c \sqsubseteq S$" means that c is a lower-bound of the set S. For convenience, we shall omit the subscript from $\vdash_{\Xi}$ whenever the generator-set Ξ is clear from the context.
(ii) The sub-system that is defined by the same rules in (i), such that all types that occur in them are homogeneous, is called the ***homogeneous safe lambda calculus***.

The safe lambda calculus deviates from the standard definition of the simply-typed lambda calculus in a number of ways. First the rules (app) and (abs) respectively can perform multiple applications and abstract several variables at once. (Of course this feature alone does not alter expressivity.) Crucially, the side-conditions in the application rule and abstraction rules require that variables in the typing context have order no smaller than that of the term being formed. We do not impose any constraint on types. In particular, type-homogeneity as used originally to define safe grammars [12] is not required here. Another difference is that we allow Ξ-constants to have arbitrary higher-order types.

Example 3 (Kierstead terms). Consider the terms $M_1 = \lambda f.f(\lambda x.f(\lambda y.y))$ and $M_2 = \lambda f.f(\lambda x.f(\lambda y.x))$ where $x, y : o$ and $f : ((o,o),o)$. The term M_2 is not safe because in the subterm $f(\lambda y.x)$, the free variable x has order 0 which is smaller than $\mathsf{ord}(\lambda y.x) = 1$. On the other hand, M_1 is safe.

It is easy to see that valid typing judgements of the safe lambda calculus satisfy the following simple invariant:

Lemma 1. *If $\Gamma \vdash M : A$ then every variable in Γ occurring free in M has order at least $ord(M)$.*

When restricted to the homogeneously-typed sub-system, the safe lambda calculus captures the original notion of safety due to Knapik *et al.* in the context of higher-order grammars:

Proposition 1. *Let $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ be a grammar and let e be an applicative term generated from the symbols in $\mathcal{N} \cup \Sigma \cup \{ z_1^{A_1}, \cdots, z_m^{A_m} \}$. A rule $F z_1 \ldots z_m \to e$ in $\mathcal{R}$ is safe if and only if $z_1 : A_1, \cdots, z_m : A_m \vdash_{\Sigma \cup \mathcal{N}} e : o$ is a valid typing judgement of the* homogeneous *safe lambda calculus.*

In what sense is the safe lambda calculus safe? A basic idea in the lambda calculus is that when performing β-reduction, one must use capture-*avoiding* substitution, which is standardly implemented by renaming bound variables afresh upon each substitution. In the safe lambda calculus, however, variable capture can never happen (as the following lemma shows). Substitution can therefore be implemented simply by capture-*permitting* replacement, without any need for variable renaming. In the following, we write $M\{N/x\}$ to denote the capture-*permitting* substitution[5] of N for x in M.

Lemma 2 (No variable capture). *There is no variable capture when performing capture-permitting substitution of N for x in M provided that $\Gamma, x : B \vdash M : A$ and $\Gamma \vdash N : B$ are valid judgments of the safe lambda calculus.*

Proof. We proceed by structural induction. The variable, constant and application cases are trivial. For the abstraction case, suppose $M = \lambda \overline{y}.R$ where $\overline{y} = y_1 \ldots y_p$. If $x \in \overline{y}$ then $M\{N/x\} = M$ and there is no variable capture.

[5] This substitution is done by textually replacing all free occurrences of x in M by N without performing variable renaming. In particular for the abstraction case we have $(\lambda y_1 \ldots y_n.M)\{N/x\} = \lambda y_1 \ldots y_n.M\{N/x\}$ when $x \notin \{y_1 \ldots y_n\}$.

If $x \notin \overline{y}$ then we have $M\{N/x\} = \lambda\overline{y}.R\{N/x\}$. By the induction hypothesis there is no variable capture in $R\{N/x\}$. Thus variable capture can only happen if the following two conditions are met: x occurs freely in R, and some variable y_i for $1 \leq i \leq p$ occurs freely in N. By Lemma 1, the latter condition implies $\mathsf{ord}(y_i) \geq \mathsf{ord}(N) = \mathsf{ord}(x)$. Since $x \notin \overline{y}$, the former condition implies that x occurs freely in the safe term $\lambda\overline{y}.R$ therefore Lemma 1 gives $\mathsf{ord}(x) \geq \mathsf{ord}(\lambda\overline{y}.R) \geq 1 + \mathsf{ord}(y_i) > \mathsf{ord}(y_i)$ which gives a contradiction. □

Remark 1. A version of the No-variable-capture Lemma also holds in safe grammars, as is implicit in (for example Lemma 3.2 of) the original paper [12].

Example 4. In order to contract the β-redex in the term

$$f : (o,o,o), x : o \vdash (\lambda\varphi^{(o,o)} x^o.\varphi\, x)(\underline{f\, x}) : (o,o)$$

one should rename the bound variable x with a fresh name to prevent the capture of the free occurrence of x in the underlined term during substitution. Consequently, by the previous lemma, the term is not safe. Indeed, it cannot be because $\mathsf{ord}(x) = 0 < 1 = \mathsf{ord}(f x)$.

Note that it is not the case that λ-terms that satisfy the No-variable-capture Lemma are necessarily safe. For instance the β-redex in $\lambda y^o z^o.(\lambda x^o.y)z$ can be contracted using capture-permitting substitution, even though the term is not safe.

Reductions and Transformations Preserving Safety

From now on we will use the standard notation $M\,[N/x]$ to denote the substitution of N for x in M. It is understood that, provided that M and N are safe, this substitution is capture-permitting.

Lemma 3 (Substitution preserves safety). *If $\Gamma, x : B \vdash M : A$ and $\Gamma \vdash N : B$ then $\Gamma \vdash M[N/x] : A$.*

This is proved by an easy induction on the structure of the safe term M.

It is desirable to have an appropriate notion of reduction for our calculus. However the standard β-reduction rule is not adequate. Indeed, safety is not preserved by β-reduction as the following example shows. Suppose that $w, x, y, z : o$ and $f : (o,o,o) \in \Sigma$ then the safe term $(\lambda xy.fxy)zw$ β-reduces to $(\underline{\lambda y.fzy})w$ which is unsafe since the underlined order-1 subterm contains a free occurrence of the ground-type z. However if we perform one more reduction we obtain the safe term fzw. This suggests an alternative notion of reduction that performs simultaneous reduction of "consecutive" β-redexes. In order to define this reduction we first introduce the appropriate notion of redex.

In the simply-typed lambda calculus a redex is a term of the form $(\lambda x.M)N$. In the safe lambda calculus, a redex is a succession of several standard redexes:

Definition 3. Let $l \geq 1$ and $n \geq 1$. We use the abbreviations $\overline{x}$ and $\overline{x} : \overline{A}$ for $x_1 \dots x_n$ and $x_1 : A_1, \dots, x_n : A_n$ respectively.

A ***safe redex*** is a safe term of the form $(\lambda \overline{x}.M)N_1 \dots N_l$ such that the variables $\overline{x}$ are abstracted altogether by one instance of the (abs) rule and the term $(\lambda \overline{x}.M)$ is applied to $N_1, \dots, N_l$ by one instance of the (app) rule.

Thus M, the N_i's and the redex itself are all safe terms. For instance, in the case $n < l$, a safe redex has a derivation tree of the following form:

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma, \overline{x} : \overline{A} \vdash M : (A_{n+1}, \dots, A_l, B)}}{\Gamma \vdash \lambda \overline{x}.M : (A_1, \dots, A_l, B)}(\mathsf{abs}) \quad \dfrac{\dots}{\Gamma \vdash N_1 : A_1} \quad \dots \quad \dfrac{\dots}{\Gamma \vdash N_l : A_l}}{\Gamma \vdash (\lambda \overline{x}.M)N_1 \dots N_l : B}(\mathsf{app})$$

We are now in a position to define a notion of reduction for safe terms.

Definition 4. We use the abbreviations $\overline{x} = x_1 \dots x_n$, $\overline{N} = N_1 \dots N_l$. The relation β_s is defined on the set of safe redexes as:

$$\begin{aligned} \beta_s = \{ & \ (\lambda \overline{x}.M)N_1 \dots N_l \mapsto \lambda x_{l+1} \dots x_n.M \left[\overline{N}/x_1 \dots x_l\right], \text{ for } n > l\} \\ \cup \{ & \ (\lambda \overline{x}.M)N_1 \dots N_l \mapsto M \left[N_1 \dots N_n/\overline{x}\right] N_{n+1} \dots N_l, \text{ for } n \leq l\} \ . \end{aligned}$$

where $M\left[R_1 \dots R_k/z_1 \dots z_k\right]$ denotes the simultaneous substitution of $R_1, \dots, R_k$ for $z_1, \dots, z_k$ in M. The ***safe β-reduction***, written $\rightarrow_{\beta_s}$, is the compatible closure of the relation β_s with respect to the formation rules of the safe lambda calculus.

Remark: The β_s-reduction is a multi-step β-reduction *i.e.* $\rightarrow_\beta \subset \rightarrow_{\beta_s} \subset \twoheadrightarrow_\beta$.

Lemma 4 (β_s-reduction preserves safety). *If $\Gamma \vdash s : A$ and $s \rightarrow_{\beta_s} t$ then $\Gamma \vdash t : A$.*

Proof. It suffices to show that the relation β_s preserves safety. Suppose that $s \ \beta_s \ t$ where s is the safe-redex $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ with $x_1 : B_1, \dots, x_n : B_n$ and M of type C. W.l.o.g we can assume that the last rule used to form the term s is (app) *i.e.* not the weakening rule (wk), thus we have $\Gamma = fv(s)$.

Suppose $n > l$ then $A = (B_{l+1}, \dots, B_n, C)$. By Lemma 3 we can form the safe term $\Gamma, x_{l+1} : B_{l+1}, \dots x_n : B_n \vdash M\left[\overline{N}/x_1 \dots x_l\right] : C$. By Lemma 1, since s is safe, all the variables in Γ have order $\geq \mathsf{ord}(A)$. This ensures that the side-condition of the (abs) rule is verified if we abstract the variables $x_{l+1} \dots x_n$, which gives us the judgement $\Gamma \vdash t : A$.

Suppose $n \leq l$. The substitution lemma gives $\Gamma \vdash M\left[N_1 \dots N_n/\overline{x}\right] : C$ and using (app) we form $\Gamma \vdash t : A$. □

In general, safety is not preserved by η-expansion; for instance we have $\vdash \lambda y^o z^o.y : (o, o, o)$ but $\not\vdash \lambda x^o.(\lambda y^o z^o.y)x : (o, o, o)$. However safety is preserved by η-reduction:

Lemma 5 (η-reduction preserves safety). *$\Gamma \vdash \lambda \varphi.s\varphi : A$ with φ not occurring free in s implies $\Gamma \vdash s : A$.*

Proof. Suppose $\Gamma \vdash \lambda\varphi.s\varphi : A$. If s is an abstraction then by construction of the safe term $\lambda\varphi.s\varphi$, s is necessarily safe. If $s = N_0 \dots N_p$ with $p \geq 1$ then again, since $\lambda\varphi.N_0 \dots N_p\varphi$ is safe, each of the N_i is safe for $0 \leq i \leq p$ and for any $z \in fv(\lambda\varphi.s\varphi)$, $\mathsf{ord}(z) \geq \mathsf{ord}(\lambda\varphi.s\varphi) = \mathsf{ord}(s)$. Since φ does not occur free in s we have $fv(s) = fv(\lambda\varphi.s\varphi)$, thus we can use the application rule to form $fv(s) \vdash N_0 \dots N_p : A$. The weakening rules permits us to conclude $\Gamma \vdash s : A$. $\square$

The η-long normal form (or simply η-long form) of a term is obtained by hereditarily η-expanding every subterm occurring at an operand position. Formally the ***η-long form*** $\lceil t \rceil$ of a term $t : (A_1, \dots, A_n, o)$ with $n \geq 0$ is defined by cases according to the syntactic shape of t:

$$\begin{aligned} \lceil \lambda x.s \rceil &= \lambda x.\lceil s \rceil \\ \lceil x s_1 \dots s_m \rceil &= \lambda\overline{\varphi}.x\lceil s_1 \rceil \dots \lceil s_m \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil \\ \lceil (\lambda x.s) s_1 \dots s_p \rceil &= \lambda\overline{\varphi}.(\lambda x.\lceil s \rceil)\lceil s_1 \rceil \dots \lceil s_p \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil \end{aligned}$$

where $m \geq 0$, $p \geq 1$, x is a variable or constant, $\overline{\varphi} = \varphi_1 \dots \varphi_n$ and each $\varphi_i : A_i$ is a fresh variable.

Lemma 6 (η-long normalization preserves safety). *If $\Gamma \vdash s : A$ then $\Gamma \vdash \lceil s \rceil : A$.*

Proof. First we observe that for any variable or constant $x : A$ we have $x : A \vdash \lceil x \rceil : A$. We show this by induction on $\mathsf{ord}(x)$. It is verified for any ground type variable x since $x = \lceil x \rceil$. Step case: $x : A$ with $A = (A_1, \dots, A_n, o)$ and $n > 0$. Let $\varphi_i : A_i$ be fresh variables for $1 \leq i \leq n$. Since $\mathsf{ord}(A_i) < \mathsf{ord}(x)$ the induction hypothesis gives $\varphi_i : A_i \vdash \lceil \varphi_i \rceil : A_i$. Using (wk) we obtain $x : A, \overline{\varphi} : \overline{A} \vdash \lceil \varphi_i \rceil : A_i$. The application rule gives $x : A, \overline{\varphi} : \overline{A} \vdash x\lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil : o$ and the abstraction rule gives $x : A \vdash \lambda\overline{\varphi}.x\lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil = \lceil x \rceil : A$.

We now prove the lemma by induction on s. The base case is covered by the previous observation. *Step case:*

- $s = x s_1 \dots s_m$ with $x : (B_1, \dots, B_m, A)$, $A = (A_1, \dots, A_n, o)$ for some $m \geq 0$, $n > 0$ and $s_i : B_i$ for $1 \leq i \leq m$. Let $\varphi_i : A_i$ be fresh variables for $1 \leq i \leq n$. By the previous observation we have $\varphi_i : A_i \vdash \lceil \varphi_i \rceil : A_i$, the weakening rule then gives us $\Gamma, \overline{\varphi} : \overline{A} \vdash \lceil \varphi_i \rceil : A_i$. Since the judgement $\Gamma \vdash x s_1 \dots s_m : A$ is formed using the (app) rule, each s_j must be safe for $1 \leq j \leq m$, thus by the induction hypothesis we have $\Gamma \vdash \lceil s_j \rceil : B_j$ and by weakening we get $\Gamma, \overline{\varphi} : \overline{A} \vdash \lceil s_j \rceil : B_j$. The (app) rule then gives $\Gamma, \overline{\varphi} : \overline{A} \vdash x\lceil s_1 \rceil \dots \lceil s_m \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil : o$. Finally the (abs) rule gives $\Gamma \vdash \lambda\overline{\varphi}.x\lceil s_1 \rceil \dots \lceil s_m \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil = \lceil s \rceil : A$, the side-condition of (abs) being verified since $\mathsf{ord}(\lceil s \rceil) = \mathsf{ord}(s)$.
- $s = t s_0 \dots s_m$ where t is an abstraction. For some fresh variables $\varphi_1, \dots, \varphi_n$ we have $\lceil s \rceil = \lambda\overline{\varphi}.\lceil t \rceil \lceil s_0 \rceil \dots \lceil s_m \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil$. Again, using the induction hypothesis we can easily derive $\Gamma \vdash \lambda\overline{\varphi}.\lceil t \rceil \lceil s_0 \rceil \dots \lceil s_m \rceil \lceil \varphi_1 \rceil \dots \lceil \varphi_n \rceil : A$.
- $s = \lambda\overline{\eta}.t$ where $\overline{\eta} : \overline{B}$ and $t : C$ is not an abstraction. The induction hypothesis gives $\Gamma, \overline{\eta} : \overline{B} \vdash \lceil t \rceil : C$ and using (abs) we get $\Gamma \vdash \lambda\overline{\eta}.\lceil t \rceil = \lceil s \rceil : A$. $\square$

Note that the converse does not hold in general, for instance $\lambda x^o.f^{(o,o,o)}x^o$ is unsafe although $\lceil \lambda x.fx \rceil = \lambda x^o y^o.fxy$ is safe.

Numeric Functions Representable in the Safe Lambda Calculus

Natural numbers can be encoded into the simply-typed lambda calculus using the Church Numerals: each $n \in \mathbb{N}$ is encoded into the term $\overline{n} = \lambda sz.s^n z$ of type $I = ((o,o),o,o)$ where o is a ground type. In 1976 Schwichtenberg [21] showed the following:

Theorem 1 (Schwichtenberg 1976). *The numeric functions representable by simply-typed λ-terms of type $I \to \ldots \to I$ using the Church Numeral encoding are exactly the multivariate polynomials* extended with the conditional function.

If we restrict ourselves to safe terms, the representable functions are exactly the multivariate polynomials:

Theorem 2. *The functions representable by safe λ-expressions of type $I \to \ldots \to I$ are exactly the multivariate polynomials.*

Corollary 1. *The conditional operator $C : I \to I \to I \to I$ verifying $Ctyz \to_\beta y$ if $t \to_\beta \overline{0}$ and $Ctyz \to_\beta z$ if $t \to_\beta \overline{n+1}$ is not definable in the safe simply-typed lambda calculus.*

Proof. Natural numbers are encoded using Church Numerals: $\overline{n} = \lambda sz.s^n z$. Addition: For $n,m \in \mathbb{N}$, $\overline{n+m} = \lambda \alpha^{(o,o)} x^o.(\overline{n}\alpha)(\overline{m}\alpha x)$. Multiplication: $\overline{n.m} = \lambda \alpha^{(o,o)}.\overline{n}(\overline{m}\alpha)$. All these terms are safe and clearly any multivariate polynomial $P(n_1,\ldots,n_k)$ can be computed by composing the addition and multiplication terms as appropriate.

For the converse, let U be a safe λ-term of type $I \to I \to I$. The generalization to terms of type $I^n \to I$ for $n > 2$ is immediate (they correspond to polynomials with n variables). W.l.o.g we can assume that $U = \lambda xy\alpha z.u$ where u is a safe term of ground type in β-normal form with $fv(u) \subseteq \{x,y : I, z : o, \alpha : o \to o\}$.

Notation: Let T be a set of terms of type $\tau \to \tau$ and T' be a set of terms of type τ then $T \cdot T'$ denotes the set of terms $\{ss' : \tau \mid s \in T \wedge s' \in T'\}$. We also define $T^k \cdot T'$ recursively as follows: $T^0 \cdot T' = T'$ and for $k \geq 0$, $T^{k+1} \cdot T' = T \cdot (T^k \cdot T')$ (*i.e.* $T^k \cdot T'$ denotes $\{s_1(\ldots(s_k s')) \mid s_1,\ldots,s_k \in T \wedge s' \in T'\}$). We define $T^+ \cdot T' = \bigcup_{k>0} T^k \cdot T'$ and $T^* \cdot T' = (T^+ \cdot T') \cup T'$. For two sets of terms T and T', we write $T =_\beta T'$ to express that any term of T is β-convertible to some term t' of T' and reciprocally.

Let us write $\mathcal{N}^\tau$ for the set of β-normal terms of type τ where τ ranges in $\{o, o \to o, I\}$ and with free variables in $\{x,y : I, z : o, \alpha : o \to o\}$. We write $\mathcal{A}^\tau$ for the subset of $\mathcal{N}^\tau$ consisting of applications only (*i.e.* not abstractions). Let B be the set of terms of type (o,o) defined by $B = \{\alpha\} \cup \{\lambda a.b \mid b \in \{a,z\}, a \neq z\}$. It is easy to see that the following equations hold:

$$\begin{aligned}
\mathcal{A}^I &= \{x,y\} \\
\mathcal{N}^{(o,o)} &= B \cup \mathcal{A}^I \cdot \mathcal{N}^{(o,o)} = (\mathcal{A}^I)^* \cdot B \\
\mathcal{A}^{(o,o)} &= \{\alpha\} \cup (\mathcal{A}^I)^+ \cdot B \\
\mathcal{A}^o = \mathcal{N}^o &= \{z\} \cup \mathcal{A}^{(o,o)} \cdot \mathcal{N}^o = (\mathcal{A}^{(o,o)})^* \cdot \{z\}
\end{aligned}$$

Hence $\mathcal{A}^o = (\{\alpha\} \cup \{x,y\}^+ \cdot (\{\alpha\} \cup \{\lambda a.b \mid b \in \{a,z\}, a \neq z\}))^* \cdot \{z\}$. Since u is safe, it cannot contain terms of the form $\lambda a.z$ with $a \neq z$ occurring at an operand position, therefore since u belongs to $\mathcal{A}^o$ we have:

$$u \in \left(\{\alpha\} \cup \{x,y\}^+ \cdot \{\alpha, \underline{i}\}\right)^* \cdot \{z\} \tag{1}$$

where $\underline{i}$ is the identity term of type $o \to o$.

We observe that $\overline{k}\underline{i} =_\beta \underline{i}$ for all $k \in \mathbb{N}$ and for $l \geq 1$, for all $k_1, \ldots k_l \in \mathbb{N}$, $\overline{k_1} \ldots \overline{k_l}\alpha =_\beta \overline{k_1 \times \ldots \times k_l}\alpha$. Hence for all $m,n \in \mathbb{N}$ we have:

$$\begin{aligned} \{\overline{m},\overline{n}\}^+ \cdot \{\alpha, \underline{i}\} &=_\beta \{\underline{i}\} \cup \{\overline{m^i n^j}\alpha \mid i+j \geq 1\} \\ &= \{\overline{m^i n^j}\alpha \mid i,j \geq 0\} \qquad \text{(since } \underline{i} = \overline{0}\alpha) \end{aligned} \tag{2}$$

therefore:

$$\begin{aligned} u[\overline{m},\overline{n}/x,y] &\in (\{\alpha\} \cup \{\overline{m},\overline{n}\}^+ \cdot \{\alpha, \underline{i}\})^* \cdot \{z\} && \text{(by eq. 1)} \\ &=_\beta \left(\{\alpha\} \cup \{\overline{m^i n^j}\alpha \mid i,j \geq 0\}\right)^* \cdot \{z\} && \text{(by eq. 2)} \\ &=_\beta \left\{\overline{m^i n^j}\alpha \mid i,j \geq 0\right\}^* \cdot \{z\} && (\alpha z =_\beta \overline{1}\alpha z). \end{aligned}$$

Furthermore, for all $m,n,r,i,j \in \mathbb{N}$ we have $\overline{m^i n^j}\alpha(\alpha^r z) =_\beta \alpha^{r+m^i n^j} z$, hence $u[\overline{mn}/x,y] =_\beta \alpha^{p(m,n)} z$ where $p(m,n) = \sum_{0 \leq k \leq d} m^{i_k} n^{j_k}$ for some $i_k, j_k \geq 0$, $k \in \{0,..,d\}$ and $d \geq 0$. Thus $U\overline{mn} =_\beta \overline{p(m,n)}$. $\square$

For instance, the term $C = \lambda FGH\alpha x.H(\underline{\lambda y.G\alpha x})(F\alpha x)$ used by Schwichtenberg [21] to define the conditional operator is unsafe since the underlined subterm is of order 1, occurs at an operand position and contains an occurrence of x of order 0.

3 A Game-Semantic Account of Safety

Our aim here is to characterize safety, which is a syntactic property, by game semantics. Because of length restriction, we shall assume that the reader is familiar with the basics of game semantics. (For an introduction, we recommend [2]). Recall that a *justified sequence* over an arena is an alternating sequence of O-moves and P-moves such that every move m, except the opening move, has a pointer to some earlier occurrence of the move m_0 such that m_0 enables m in the arena. A *play* is just a justified sequence that satisfies Visibility and Well-Bracketing. A basic result in game semantics is that λ-terms are denoted by *innocent strategies*, which are strategies that depends only on the *P-view* of a play. The main result (Theorem 3) of this section is that if a λ-term is safe, then its game semantics (is an innocent strategy that) is *P-incrementally justified*. In such a strategy, pointers emanating from the P-moves of a play are uniquely reconstructible from the underlying sequence of moves and pointers from the O-moves therein: Specifically a P-question always points to the last pending O-question (in the P-view) of a greater order.

The proof of Theorem 3 depends on a Correspondence Theorem (see the long version of this paper) that relates the strategy denotation of a λ-term M to the set of *traversals* over a certain abstract syntax tree of the η-long form of M. In the language of game semantics, traversals are just (concrete representations of) the *uncovering* (in the sense of Hyland and Ong [11]) of plays in the strategy denotation.

The useful transference technique between plays and traversals was originally introduced by one of us [19] for studying the decidability of MSO theories of infinite structures generated by higher-order grammars (in which the Σ-constants are at most order 1, and *uninterpreted*). The long version of the paper presents an extension of this framework to the general case of the simply-typed lambda calculus with free variables of any order. A new traversal rule is introduced to handle nodes labelled with free variables. Also new nodes are added to the computation tree to account for the answer moves of the game semantics, thus enabling the framework to model languages with interpreted constants such as PCF (by adding traversal rules to handle constant nodes).

Incrementally-Bound Computation Tree

In [19] the computation tree of a grammar is defined as the unravelling of a finite graph representing the long transform of a grammar. Similarly we define the computation tree of a λ-term as an abstract syntax tree of its η-long normal form. We write $l(t_1, \ldots, t_n)$ with $n \geq 0$ to denote the tree with a root labelled l with n children subtrees $t_1, \ldots, t_n$. In the following, judgements of the form $\Gamma \vdash M : T$ refer to simply-typed terms not necessarily safe unless mentioned.

Definition 5. The ***computation tree*** $\tau(M)$ of a simply-typed term $\Gamma \vdash M : T$ with variable names in a countable set $\mathcal{V}$ is a tree with labels in $\{@\} \cup \mathcal{V} \cup \{\lambda x_1 \ldots x_n \mid x_1, \ldots, x_n \in \mathcal{V}\}$ defined from its η-long form as follows:

$$\text{for } n \geq 0 \text{ and } s : o,\ \tau(\lambda x_1 \ldots x_n . s) = \lambda x_1 \ldots x_n(t) \quad \text{where } \tau(s) = \lambda(t)$$
$$\text{for } m \geq 0 \text{ and } x \in \mathcal{V},\ \tau(x s_1 \ldots s_m : o) = \lambda(x(\tau(s_1), \ldots, \tau(s_m)))$$
$$\text{for } m \geq 1,\ \tau((\lambda x.t) s_1 \ldots s_m : o) = \lambda(@(\tau(\lambda x.t), \tau(s_1), \ldots, \tau(s_m)))\ .$$

Even-level nodes are λ-nodes (the root is on level 0). A single λ-node can represent several consecutive variable abstractions or it can just be a *dummy lambda* if the corresponding subterm is of ground type. Odd-level nodes are variable or application nodes.

The ***order*** of a node n, written $\mathsf{ord}(n)$, is defined as follows: @-nodes have order 0. The order of a variable-node is the type-order of the variable labelling it. The order of the root node is the type-order of $(A_1, \ldots, A_p, T)$ where $A_1, \ldots, A_p$ are the types of the variables in the context Γ. Finally, the order of a lambda node different from the root is the type-order of the term represented by the sub-tree rooted at that node.

We say that a variable node n labelled x is ***bound*** by a node m, and m is called the ***binder*** of n, if m is the closest node in the path from n to the root such that m is labelled $\lambda\overline{\xi}$ with $x \in \overline{\xi}$. We introduce a class of computation trees in which the binder node is uniquely determined by the nodes' orders:

Definition 6. A computation tree is ***incrementally-bound*** if for all variable node x, either x is *bound* by the first λ-node in the path to the root with order $> \mathsf{ord}(x)$ or x is a *free variable* and all the λ-nodes in the path to the root except the root have order $\leq \mathsf{ord}(x)$.

Proposition 2

(i) If M is safe then $\tau(M)$ is incrementally-bound.
(ii) Conversely, if M is a closed *simply-typed term and $\tau(M)$ is incrementally-bound then the η-long form of M is safe.*

The assumption that M is closed is necessary. For instance for $x, y : o$, the two identical computation trees $\tau(\lambda xy.x)$ and $\tau(\lambda y.x)$ are incrementally-bound but $\lambda xy.x$ is safe and $\lambda y.x$ is not.

P-Incrementally Justified Strategy

We now consider the game-semantic model of the simply-typed lambda calculus. The strategy denotation of a term is written $[\![\Gamma \vdash M : T]\!]$. We define the ***order*** of a move m, written $\mathsf{ord}(m)$, to be the length of the path from m to its furthest leaf in the arena minus 1. (There are several ways to define the order of a move; the definition chosen here is sound in the current setting where each question move in the arena enables at least one answer move.)

Definition 7. A strategy σ is said to be ***P-incrementally justified*** if for any play $s\, q \in \sigma$ where q is a P-question, q points to the last unanswered O-question in $\ulcorner s \urcorner$ with order strictly greater than $\mathsf{ord}(q)$.

Note that although the pointer is determined by the P-view, the choice of the move itself can be based on the whole history of the play. Thus P-incremental justification does not imply innocence.

The definition suggests an algorithm that, given a play of a P-incrementally justified denotation, uniquely recovers the pointers from the underlying sequence of moves and from the pointers associated to the O-moves therein. Hence:

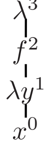
Lemma 7. *In P-incrementally justified strategies, pointers emanating from P-moves are superfluous.*

Example 5. Copycat strategies, such as the identity strategy id_A on game A or the evaluation map $ev_{A,B}$ of type $(A \Rightarrow B) \times A \rightarrow B$, are all P-incrementally justified.[6]

[6] In such strategies, a P-move m is justified as follows: either m points to the preceding move in the P-view or the preceding move is of smaller order and m is justified by the second last O-move in the P-view.

The Correspondence Theorem gives us the following equivalence:

Proposition 3. *For a β-normal term $\Gamma \vdash M : T$, $\tau(M)$ is incrementally-bound if and only if $[\![\Gamma \vdash M : T]\!]$ is P-incrementally justified.*



Example: Consider the β-normal term $\Gamma \vdash f(\lambda y.x) : o$ where $y : o$ and $\Gamma = f : ((o,o),o),\ x : o$. The figure on the right represents its computation tree with the node orders given as superscripts. Node x is not incrementally-bound therefore $\tau(f(\lambda y.x))$ is not incrementally-bound and by Proposition 3, $[\![\Gamma \vdash f(\lambda y.x) : o]\!]$ is not incrementally-justified (although $[\![\Gamma \vdash f : ((o,o),o)]\!]$ and $[\![\Gamma \vdash \lambda y.x : (o,o)]\!]$ are).

Propositions 2 and 3 allow us to show the following:

Theorem 3 (Safety and P-incremental justification)

(i) If $\Gamma \vdash M : T$ is safe then $[\![\Gamma \vdash M : T]\!]$ is P-incrementally justified.
(ii) If $\vdash M : T$ is a closed simply-typed term and $[\![\vdash M : T]\!]$ is P-incrementally justified then the η-long form of the β-normal form of M is safe.

Putting Theorem 3(i) and Lemma 7 together gives:

Proposition 4. *In the game semantics of safe λ-terms, pointers emanating from P-moves are unnecessary i.e. they are uniquely recoverable from the underlying sequences of moves and from O-moves' pointers.*

In fact, as the last example highlights, pointers are entirely superfluous at order 3 for safe terms. This is because for question moves in the first two levels of an arena, the associated pointers are uniquely recoverable thanks to the visibility condition. At the third level, the question moves are all P-moves therefore their associated pointers are uniquely recoverable by P-incremental justification. This is not true anymore at order 4: Take the safe term $\psi : (((o^4,o^3),o^2),o^1) \vdash \psi(\lambda\varphi.\varphi a) : o^0$ for some constant $a : o$, where $\varphi : (o,o)$. Its strategy denotation contains plays whose underlying sequence of moves is $q_0\, q_1\, q_2\, q_3\, q_2\, q_3\, q_4$. Since q_4 is an O-move, it is not constrained by P-incremental justification and thus it can point to any of the two occurrences of q_3.[7]

Safe PCF and Safe Idealised Algol

PCF is the simply-typed lambda calculus augmented with basic arithmetic operators, if-then-else branching and a family of recursion combinator $Y_A : ((A,A),A)$ for any type A. We define *safe* PCF to be PCF where the application and abstraction rules are constrained in the same way as the safe lambda calculus. This

[7] More generally, a P-incrementally justified strategy can contain plays that are not "O-incrementally justified" since it must take into account any possible strategy incarnating its context, including those that are not P-incrementally justified. In the given example, the version of the play that is not O-incrementally justified is involved in the strategy composition $[\![\vdash M_2 : (((o,o),o),o)]\!];[\![\psi : (((o,o),o),o) \vdash \psi(\lambda\varphi.\varphi a) : o]\!]$ where M_2 denotes the unsafe Kierstead term.

language inherits the good properties of the safe lambda calculus: No variable capture occurs when performing substitution and safety is preserved by the reduction rules of the small-step semantics of PCF. Using a PCF version of the Correspondence Theorem we can prove the following:

Theorem 4. *Safe PCF terms have P-incrementally justified denotations.*

Similarly, we can define safe IA to be safe PCF augmented with the imperative features of Idealized Algol (IA for short) [20]. Adapting the game-semantic correspondence and safety characterization to IA seems feasible although the presence of the base type var, whose game arena $\mathtt{com}^{\mathbb{N}} \times \mathtt{exp}$ has infinitely many initial moves, causes a mismatch between the simple tree representation of the term and its game arena. It may be possible to overcome this problem by replacing the notion of computation tree by a "computation directed acyclic graph".

The possibility of representing plays *without some or all of their pointers* under the safety assumption suggests potential applications in algorithmic game semantics. Ghica and McCusker [9] were the first to observe that pointers are unnecessary for representing plays in the game semantics of the second-order finitary fragment of Idealized Algol (IA_2 for short). Consequently observational equivalence for this fragment can be reduced to the problem of equivalence of regular expressions. At order 3, although pointers are necessary, deciding observational equivalence of IA_3 is EXPTIME-complete [18,17]. Restricting the problem to the safe fragment of IA_3 may lead to a lower complexity.

4 Further Work and Open Problems

The safe lambda calculus is still not well understood. Many basic questions remain. What is a (categorical) model of the safe lambda calculus? Does the calculus have interesting models? What kind of reasoning principles does the safe lambda calculus support, via the Curry-Howard Isomorphism? Does the safe lambda calculus characterize a complexity class, in the same way that the simply-typed lambda calculus characterizes the polytime-computable numeric functions [13]? Do incrementally-justified strategies compose? Is the addition of unsafe contexts to safe ones conservative with respect to observational (or contextual) equivalence?

With a view to algorithmic game semantics and its applications, it would be interesting to identify sublanguages of Idealised Algol whose game semantics enjoy the property that pointers in a play are uniquely recoverable from the underlying sequence of moves. We name this class PUR. IA_2 is the paradigmatic example of a PUR-language. Another example is *Serially Re-entrant Idealized Algol* [1], a version of IA where multiple uses of arguments are allowed only if they do not "overlap in time". We believe that a PUR language can be obtained by imposing the *safety condition* on IA_3. Murawski [16] has shown that observational equivalence for IA_4 is undecidable; is observational equivalence for *safe* IA_4 decidable?

References

1. Abramsky, S.: Semantics via game theory. In: Marktoberdorf International Summer School, Lecture slides (2001)
2. Abramsky, S., McCusker, G.: Game semantics. In: Schwichtenberg, H., Berger, U. (eds.) Logic and Computation. LNCS, Springer, Heidelberg (1998)
3. Aehlig, K., de Miranda, J.G., Ong, C.-H.L.: Safety is not a restriction at level 2 for string languages. Technical report, University of Oxford (2004)
4. Aho, A.V.: Indexed grammars – an extension of context-free grammars. J. ACM 15(4), 647–671 (1968)
5. Caucal, D.: On infinite terms having a decidable monadic theory, pp. 165–176 (2002)
6. Damm, W.: The IO- and OI-hierarchy. TCS 20, 95–207 (1982)
7. Damm, W., Goerdt, A.: An automata-theoretical characterization of the OI-hierarchy. Information and Control 71(1-2), 1–32 (1986)
8. de Miranda, J.G.: Structures generated by higher-order grammars and the safety constraint. Dphil thesis, University of Oxford (2006)
9. Ghica, D.R., McCusker, G.: Reasoning about idealized algol using regular languages. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 103–116. Springer, Heidelberg (2000)
10. Hague, M., Murawski, A.S., Ong, C.-H.L., Serre, O.: Collapsible pushdown automata and recursive schemes, p. 13, preprint (November 2006)
11. Hyland, J.M.E., Ong, C.-H.L.: On full abstraction for PCF: I, II, and III. Information and Computation 163(2), 285–408 (2000)
12. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) ETAPS 2002 and FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
13. Leivant, D., Marion, J.-Y.: Lambda calculus characterizations of poly-time. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 274–288. Springer, Heidelberg (1993)
14. Maslov, A.N.: The hierarchy of indexed languages of an arbitrary level. Soviet Math. Dokl. 15, 1170–1174 (1974)
15. Maslov, A.N.: Multilevel stack automata. Problems of Information Transmission 12, 38–43 (1976)
16. Murawski, A.: On program equivalence in languages with ground-type references. In: Proceedings 18th Annual IEEE Symposium on Logic in Computer Science, June 22-25, 2003, pp. 108–117 (2003)
17. Murawski, A.S., Walukiewicz, I.: Third-order idealized algol with iteration is decidable. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 202–218. Springer, Heidelberg (2005)
18. Ong, C.-H.L.: An approach to deciding observational equivalence of algol-like languages. Ann. Pure Appl. Logic 130(1-3), 125–171 (2004)
19. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: Proceedings of IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos, Extended abstract (2006)
20. Reynolds, J.C.: The essence of algol. In: de Bakker, J.W., van Vliet, J.C. (eds.) Algorithmic Languages. IFIP, pp. 345–372. North-Holland, Amsterdam (1981)
21. Schwichtenberg, H.: Definierbare funktionen im lambda-kalkul mit typen. Archiv Logik Grundlagenforsch 17, 113–114 (1976)

Intuitionistic Refinement Calculus

Sylvain Boulmé

Laboratoire d'Informatique de Grenoble
681, rue de la Passerelle BP-72 – 38402 St-Martin D'Hères – France
Sylvain.Boulme@imag.fr

Abstract. Refinement calculi are program logics which formalize the "top-down" methodology of software development promoted by Dijkstra and Wirth in the early days of structured programming. I present here the *shallow* embedding of a refinement calculus into Coq constructive type theory. This embedding involves monad transformers and the computational reflexion of weakest-preconditions, using a continuation passing style. It should allow to reason about many ML programs combining non-functional features (state, exceptions, etc) with purely functional ones (higher-order functions, structural recursion, etc).

1 Introduction

The refinement calculus of [Mor90] is the kernel of the B method [Abr96] which has been successfully applied in large industrial projects [Beh99]. This paper presents the marriage of this refinement calculus with the Calculus of Inductive Constructions [Coq88, PM93], the constructive type theory of Coq [Coq04]. This marriage is interesting for both formalisms. On the Coq side, refinement calculus provides a simple and efficient embedding of computational behaviors which are not natively available in Coq: side-effects and partial functions. Here, partial functions may involve undefined behaviors or non-termination, interpreted in partial or total correctness. On the other side, this marriage shows that all kinds of Coq computation can be fully integrated into refinement calculus: pattern-matching, structural recursion over inductive types, and higher-order functions. Moreover, because Coq is a typed lambda-calculus with types and propositions as *first-class citizens* (see Figure 1), it is more than a higher-order logic: it is also a programming language where propositions and proofs are first-class values. Hence, programming computations of WP (Weakest-Preconditions) in Coq is very easy because substitutions of variable are expressed at an abstract level. This allows to use Coq as the kernel of a refinement prover, with two benefits. First, all libraries and tools developed for Coq can be reused in refinement proofs. Second, refinement rules are formally proved. In particular, the reflexion of WP-computations ensures their soundness w.r.t. purely deductive rules.

Section 2 motivates this work by the formalization of reasoning about higher-order imperative functions in Coq. It illustrates that this reasoning may combine equational reasoning in monads with Hoare logic. This leads me to introduce *Dijkstra Specification Monads* (abbreviated DSM). Section 3 defines DSM as a

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 54–69, 2007.
© Springer-Verlag Berlin Heidelberg 2007

"Type" is the type of types. In order to avoid logical paradoxes, each of its occurrence is implicitly indexed with a natural, such that in **Type**:**Type**, the left index is strictly lower than the right one.
"Prop" is the type of logical propositions. In Curry-Howard style, propositions are represented as types, and proofs as lambda-terms. Hence, if `A`:**Prop** then `A`:**Type**.
"forall `x:A,B`**"** is the type of functions "**fun** `x:A => b`" when if `x:A` then `b:B`. This type is also logically interpreted as universal quantification or as implication.
"`A -> B`**"** is a synonym of **forall** `(x:A),B` when `x` does not occur free in `B`.
"`A*B`**"** is the type of pairs "`(x,y)`" where `x:A` and `y:B`.
"`A /\ B`**"** represents conjunction and is like `A*B` at **Prop** level.
"exists `x:A,B`**"** represents existential quantification (defined as a dependent-pair).
"`x=y`**"** means that every property satisfied by `x` is satisfied by `y` (Leibniz's equality).

Fig. 1. A very short description of Coq syntax

very simple combination of monads and lattices in Coq, where the order relation is *refinement*. Then, it presents a Tarski fixpoint theorem, which makes DSM adapted to reason about non-terminating programs. Section 4 gives the modular construction of the state DSM. This construction uses a WP calculus embedded in Coq as a CPS (Continuation-Passing Style) semantics of DSM. It is also based on the state monad transformer. Hence, it could be easily adapted for other monad transformers like exception monad transformer. With an example combining partial functions and structural recursion, Section 5 illustrates that refinement formulae may be simplified by computing WP. Then, it shows how interactive refinement proofs mix deduction and WP-computation. Finally, Section 6 explains how the state DSM is used to prove higher-order imperative programs, and in particular how Hoare logic is expressed in the state DSM.

2 Reasoning on Higher-Order Imperative Functions in Coq

Let me first introduce a higher-order imperative example in Ocaml syntax. Given `nat` the type of natural numbers and `bintree` the type of binary trees:

```
type nat = O | S of nat;;    type bintree = Leaf | Node of bintree*bintree;;
```

Given `n: nat`, and `f: bintree->unit` (`f` is an action parametrized by a binary tree), I consider below a function `enumBT` such that `(enumBT n f)` calls successively `f` over all and only binary trees of height `n`, but only once for a given tree.

```
let rec enumBT n f = match n with
  | O -> f Leaf
  | (S p) -> enumBT p (fun l ->
                       enumBT p (fun r -> f(Node(l,r)))   ;
                       enumlt p (fun r -> f(Node(l,r)) ; f(Node(r,l))))
and enumlt n f = match n with
  | O -> ()     | (S p) -> enumBT p f ; enumlt p f
```

The main advantage of this CPS-like implementation is to call `f` as soon as a tree is computed, before computing the next tree. Function `enumBT` is defined mutually recursively over `n` with `enumlt` which enumerates binary trees with a height strictly lower than `n`. Then, it uses the fact that in a tree of height `(S n)`, either its two children have a height equal to `n`, or one of them has a height equals to `n` and the other has a height strictly lower than `n`.

In order to reason about this simple function in Coq, we may use a style inspired by Hoare logic. Unfortunately, a simple Hoare logic does not allow us to reason on such a higher-order imperative function. Actually, we need here extensions of Hoare logic as those recently proposed in [Hon05]. Alternatively, I simply propose here to specify this program using an equality on programs. Hence, I have proved in Coq (see [Bou06]) that for a given `n`, there exists `l` of type `(list bintree)` such that `l` contains only all binary trees of height `n` without duplicates, and such that `(enumBT n f)`≡`(iter l f)` where "`iter l f`" calls `f` on successive elements in `l`, and relation ≡ is an observational equivalence on expressions. Here, the expression language is formalized as a monad.

2.1 Axioms of Monads in Coq

A monad is a categorical structure expressing non-purely functional features like state or exception handling, finite non-determinism and environment interactions [Mog91, Pey93]. My Coq axiomatization of monads is given Figure 2:

- `K` is a parametrized type such that `(K A)` represents the type of an expression returning a value of type `A`. Hence, values of the monad are any Coq values.
- `equiv` is an "observational" equivalence on expressions.
- `val` is side-effect free operator to lift a value into an expression.
- `bind` corresponds to a "let-in" construct in ML: it generalizes sequence of imperative languages. The power of this operator is to allow *any* Coq function as second argument.

Using the predefined type `unit` of single value `tt`, we say a monadic expression is an instruction if its type is `(K unit)`. The `skip` instruction is thus `(val tt)`. Sequencing instructions is just applying `bind`:

```
Definition seq (A:Type) (i1:K unit) (i2:K A): K A := bind i1 (fun _ => i2).
Implicit Arguments seq [A].
```

In the following, I use the notation `If cond Then i1 Else i2` as a macro for `bind cond (fun b:bool => if b then i1 else i2)`. This lifts the if-then-else construction of Coq into monad expressions. It is easy to prove that the monadic if-then-else is compatible with equivalence of expressions. More generally, all purely functional constructions are lifted to monad expressions, such that the lifted construction is compatible with equivalence. Hence, monads support naturally pattern-matching, structural recursion, and higher-order functions. Here, I lift functions at monad expressions using call-by-value evaluation, like in ML. For example, the Coq type for `enumBT` is `nat -> (bintree -> K unit) -> K unit`.

2.2 Marrying Monads with Hoare Logic

The previous example is rather simple because `enumBT` do not modify the state directly. On other examples, we may need to specify a modification of the state

Constants of monads are:

```
K: Type -> Type.
equiv: forall (A:Type), (K A) -> (K A) -> Prop.
val: forall (A:Type), A -> (K A).
bind: forall (A B:Type), (K A) -> (A -> (K B)) -> (K B).
```

To have a lighter syntax, I require (with the commands below) that Coq infers type parameters (like `A`) of `equiv`, `val` and `bind` like a ML compiler:

```
Implicit Arguments equiv [A].
Implicit Arguments val [A].
Implicit Arguments bind [A B].
```

Hence, axioms of monads are expressed as:

```
equiv_refl: forall (A:Type) (x:(K A)), equiv x x.
equiv_sym: forall (A:Type) (x y:(K A)), (equiv x y)->(equiv y x).
equiv_trans: forall (A:Type) (x y z:(K A)),
  (equiv x y) -> (equiv y z) -> (equiv x z).
bind_compat: forall (A B:Type) (k1 k2:(K A)) (f1 f2: A -> (K B)),
  (forall (x:A), equiv (f1 x) (f2 x)) ->
     (equiv k1 k2) -> (equiv (bind k1 f1) (bind k2 f2)).
bind_val_l: forall (A B:Type) (a:A) (f:A->(K B)),
              equiv (bind (val a) f) (f a).
bind_val_r: forall (A:Type) (k:(K A)), equiv (bind k (fun a => val a)) k.
bind_assoc: forall (A B C:Type) (k:(K A)) (f: A->(K B)) (g: B -> (K C)),
    equiv (bind (bind k f) g) (bind k (fun a => bind (f a) g)).
```

Fig. 2. Coq interface for monads

as in Hoare logic. In particular, let me consider an expression `e` of type `K A` in a state monad: `e` works on a global state of type `St` (hence, the type of `St` is **Type**) and returns a result of type `A`. The state monad provides two specific operators `set: St->(K unit)`, to update the current value of the global state, and `get: (K St)`, to read the current value of the global state. These operators satisfy the following three axioms:

```
get_set: equiv (bind get set) skip.
set_get: forall (st:St), equiv (seq (set st) get) (seq (set st) (val st)).
set_set: forall (st1 st2:St), equiv (seq (set st1) (set st2)) (set st2).
```

A Hoare specification of `e` can be seen as the pair of two predicates: a *precondition* `P: St->Prop` on the initial state, and a *postcondition* `Q: St->St->A->Prop` on the initial state, the final state and the result. In total-correctness semantics, such a specification is interpreted through the following formula:

```
forall (sti:St), (P sti) -> exists stf:St, exists r:A,
     (Q sti stf r) /\ (equiv (seq (set sti) e) (seq (set stf) (val r))).
```

Fortunately, Dijkstra has invented a weakest-precondition calculus to simplify this tedious formula. Hence, there is a function `wp` of type:

```
wp: forall (A:Type),(K A) -> (St -> A -> Prop) -> St -> Prop.
Implicit Arguments wp [A].
```

such that the preceding tedious formula is equivalent to

```
forall (sti:St), (P sti) -> (wp e (Q sti) sti)
```

and such that `wp` is a CPS semantics of the state monad where continuations are predicates on the final state and the result (postconditions). In other words, `wp` discharges the user to infer manually the two existential connectors of the tedious formula.

This paper shows that refinement calculus is a marriage of monads and Hoare logic such that WP are hidden in the refinement order. Hence, from the user point of view, reasoning with refinement is very natural, because this order generalizes the equality of monads. Formally, my refinement calculus is based on *Dijkstra Specification Monads*: a combination of monads and lattices in Coq.

3 Lattice Theory of Dijkstra Specification Monads

Given a particular monad M, let me explain informally how to define the *Dijkstra Specification Monad* (or DSM in short) of M. It is an extension of M with two non-deterministic operators that transform the "programming language" of M into a "specification language". DSM are special cases of monads. Thus, to distinguish expressions of M and expressions of its DSM, the former are called *programs* and the latter are called *specifications*.

Very roughly, a specification describes *a set of programs that implement it*. This set is closed under observational equivalence. A specification S_1 *refines* a specification S_2, *if and only if every implementation of* S_1 *also implements* S_2. A program is a special case of specification which is only implemented by itself (modulo observational equivalence). More generally, composition operators of M are lifted in its DSM as the closure (under observational equivalence) of their pointwise extension. The DSM of M extends its expression language with two operators called `sync` and `choice` corresponding respectively to the intersection and the union of a family of specifications.

Below, I define DSM *axiomatically*. This axiomatization is sufficient to develop Tarski's fixpoint theory. Hence, even if all computations of the original monad M terminate, its associated DSM is expressive enough to represent non-terminating expressions. It can thus be used to reason about programs of an extension of M with fixpoint operators. This idea is illustrated in Section 6.2.

3.1 Axioms of DSM

A DSM is a monad, that provides a preorder `refines` (reflexive and transitive) such that its associated equivalence is `equiv`. In other words, `equiv s1 s2` must be equivalent to `(refines s1 s2) /\ (refines s2 s1)`.

```
refines: forall (A:Type), (K A)->(K A)->Prop.
Implicit Arguments refines [A].
```

Furthermore, the operator `bind` must be monotonic (increasing) for this preorder. Hence, all functional features (pattern-matching, structural recursion, ...) are lifted as monotonic constructions. The expression language of monads is extended with two primitive operators:

– `any`: **forall** `(A:`**Type**`), (K A)` such that `any A` is implemented by any concrete value of the type `A`. If `A` is empty, then `any A` has no implementation.
– `sync`: **forall** `(A B:`**Type**`), (A -> (K B)) -> (K B)` (with `A` and `B` as implicit arguments) such that for any `a:A`, every implementation of `sync s` must be an implementation of `s a`. If `A` is empty, we say that `sync s` *aborts*: it is refined by any specification and refines only aborting specifications.

More formally, these operators must satisfy the following five axioms:

```
any_refined: forall (A:Type) (a:A), refines (val a) (any A).
any_refines: forall (A B:Type) (s1:A -> K B) (s2:K B),
  (forall (a:A), refines (s1 a) s2) -> refines (bind (any A) s1) s2.

sync_refines: forall (A B:Type) (s:A -> K B) (a:A), refines (sync s) (s a).
sync_refined: forall (A B:Type) (s1:(K B)) (s2:A -> (K B)),
  (forall (a:A), refines s1 (s2 a)) -> (refines s1 (sync s2)).

bind_sync_indep: forall (A B C:Type) (s1:K C) (s2:K B) (s3:B -> K C),
  (A -> refines s1 (bind s2 s3))
     -> refines s1 (bind (sync (fun _:A => s2)) s3).
```

Axioms `sync_refines` and `sync_refined` express that `sync s` is the greatest lower bound[1] of `s` (where `s` is a family of specifications indexed over `A`). Let me now explain the interest and the meaning of other axioms. First, I need to define `choice` from `bind` and `any`:

```
Definition choice (A B:Type) (s:A -> K B): (K B) := bind (any A) s.
Implicit Arguments choice [A B].
```

Combining axioms of `bind` with respectively `any_refined` and `any_refines`, I derive the following two lemmas:

```
Lemma choice_refined: forall (A B:Type) (s:A -> K B) (a:A),
  (refines (s a) (choice s)).
Lemma choice_refines: forall (A B:Type) (s1:A -> K B) (s2:K B),
  (forall (a:A), refines (s1 a) s2) -> (refines (choice s1) s2).
```

These two properties express that `choice s` is the least upper bound of `s`. There is thus a relative symmetry between `choice` and `sync` with respect to `refines`. However, `choice` distributes over `bind` whereas `sync` does not. Indeed, the following lemma is a consequence of `bind` associativity:

```
Lemma choice_bind_distr: forall (A B C:Type) (s1:A -> K B) (s2:B -> K C),
    equiv (bind (choice s1) s2) (choice (fun x:A => bind (s1 x) s2)).
```

But, this property is not satisfied by `sync`. For example, let me define `s1` as **fun** `b:bool => val b` and `s2` as **fun** `x:bool => skip`. Applying our intuitive model, `sync s1` is the intersection of `(val true)` and `(val false)`: it represents the empty set. Thus, the left side of the refinement goal, `bind (sync s1) s2`, is empty. On the right side, **fun** `x => bind (s1 x) s2` is equivalent to **fun** `x:bool => skip`. Hence, its intersection is equivalent to `skip` and thus non-empty.

[1] If S_1 refines S_2, I consider that S_1 is *smaller* than S_2, according to their intuitive meaning as sets of implementations. But, in the literature about refinement, the dual view is also often considered.

Actually, axiom `bind_sync_indep` is a weak form of distributivity, when the intersection ranges over an empty family or a singleton. In particular, the following definition uses `sync` to derive a `require` operator that sets a precondition (I use here the fact that **Prop** is a subtype of **Type**). The proof of `bind_require` below uses axiom `bind_sync_indep`.

```
Definition require (P:Prop):(K unit) := sync (fun _:P => skip).
```

Instruction `require P` expresses that `P` is assumed by implementations (any implementation is authorized if `P` does not hold). Using `sync` and `bind` axioms, we show that such preconditions can only be weakened in refinement.

```
Lemma bind_require: forall (A:Type) (p1 p2:Prop) (s1 s2:K A),
   (p2 -> (p1 /\ refines s1 s2)) ->
     refines (seq (require p1) s1) (seq (require p2) s2).
Lemma require_True: equiv (require True) skip.
```

Symmetrically, I introduce an operator `ensure` to set a postcondition. Instruction `ensure P` expresses that `P` is guaranteed by implementations.

```
Definition ensure (P:Prop): (K unit) := choice (fun _:P => skip).
Lemma bind_ensure: forall (A:Type) (p1 p2:Prop) (s1 s2:K A),
  (p1 -> (p2 /\ refines s1 s2)) ->
     refines (seq (ensure p1) s1) (seq (ensure p2) s2).
Lemma ensure_True: equiv (ensure True) skip.
```

3.2 Fixpoint Theory of DSM

Using `sync`, I adapt here in Coq the Tarski's proof of existence of a smallest fixpoint for any monotonic function in a complete lattice. The main trick of this proof is the higher-order scheme (i.e. a specification "computing" a specification) in the definition of the smallest fixpoint operator below. Indeed, `sfix F` is defined as the intersection of all prefixpoints of `F` (i.e. `sp` such that `(F sp)` refines `sp`):

```
Definition sfix (A:Type) (F:(K A) -> (K A)) : (K A) :=
  sync (fun sp:(K A) => seq (require (refines (F sp) sp)) sp).
Implicit Arguments sfix [A].
```

I first prove that `sfix` refines every prefixpoint of `F`.

```
Theorem sfix_smallest: forall (A:Type) (F:(K A) -> (K A)) (sp:K A),
    (refines (F sp) sp) -> (refines (sfix F) sp).
```

Then, I prove that under the assumption that `F` is monotonic, then `sfixF F` is a fixpoint (and it is the smallest by `sfix_smallest`):

```
Theorem sfix_fix: forall (A:Type) (F:(K A) -> (K A)),
 (forall (x y:K A), refines x y -> refines (F x) (F y))
  -> equiv (sfix F) (F (sfix F)).
```

Actually the proofs of these two theorems are very simple applications of the axioms and the hypotheses.

The greatest fixpoint operator follows a symmetric construction, and has symmetric properties. These fixpoint operators are monotonic (see [Bou06]). I have also defined a notion of well-founded fixpoints for recursive functions returning a specification. These well-founded fixpoints are unique and preserve such properties as determinism and weak-termination (see [Bou06]).

4 Modular Construction of the State DSM

In the previous section, DSM are defined *axiomatically*. On the contrary, this section gives particular *models* of DSM. They are used in next sections to simplify reasoning about refinement formulae.

Let me start with an intuitive model of the pure DSM, the DSM of the pure monad (i.e. the monad of purely functional expressions). In this model, a specification is defined as a pair of a *precondition* (i.e. a proposition assumed by implementations) and a *postcondition* (i.e. a predicate on the result computed by implementations). Formally, the definition below makes `K A` a product type with `Build_K` as constructor, and `pre` and `post` as projections (parameter `A` being implicit):

```
Record K (A:Type): Type := { pre: Prop ; post: A -> Prop }.

Definition refines (A:Type) (s1 s2:K A): Prop
 := pre s2 -> ( pre s1 /\ (forall a:A, post s1 a -> post s2 a) ).

Definition val (A:Type) (a:A): K A := Build_K True (fun b => a=b).

Definition bind (A B:Type) (s1:K A) (s2:A -> K B): (K B) :=
  Build_K (pre s1 /\ (forall a:A, post s1 a -> pre (s2 a)))
          (fun b => exists a:A, post s1 a /\ post (s2 a) b).

Definition any (A:Type): K A := Build_K True (fun a:A => True).

Definition sync (A B:Type) (s:A -> K B) : K B :=
  Build_K (exists a:A, pre (s a))
          (fun b => forall a:A, pre (s a) -> post (s a) b).
```

It is straightforward to show that the previous definitions satisfy DSM axioms. Moreover, they are fully compatible with higher-order schemes like the one used in `sfix` definition. In particular, I have carefully avoided to define `K` as an inductive type that encodes the abstract syntax of specifications. Indeed, such an inductive definition would introduce universe constraints forbidding higher-order schemes.

In conclusion, this model is intuitive and simple, but not very interesting in practice: specifications are interpreted as huge formulae, because of `bind` definition. Now, I first present a second model which is logically equivalent to the previous one. It is based on WP and produces simpler formulae. Then, I give a model for the state DSM, the DSM of the state monad.

4.1 Construction of the Pure DSM from Weakest-Preconditions

The WP semantics improves the previous one by translating computational contents of specifications into Coq computations. Due to the presence of nondeterminism, there are two notions of weakest-preconditions. They are expressed below using the pre/post semantics of specifications. Hence, let us assume a type `A`, and a specification `s` of type `(K A)`.

Given a postcondition `R` on the result of `s`, the *strong demonic weakest-precondition* of `s` for `R`, noted `(sdemon s R)`, is the necessary and sufficient condition which must hold for `R` to be satisfied however `s` is implemented. Hence, `(sdemon s)` has type `(A->Prop)->Prop` and satisfies:

```
forall R:A->Prop, (sdemon s R) <-> ((pre s) /\ forall a:A, (post s a)->(R a))
```

On the contrary, assuming that `(pre s)` holds, the *angelic weakest-precondition* of `s` for `R` is the necessary and sufficient condition which ensures that there exists an implementation of `s` satisfying `R`.

```
(pre s) -> forall R:A->Prop, (angel s R)<->(exists a:A, (post s a) /\ (R a))
```

In classical logic, these two notions can be defined dually using Excluded-Middle. For instance, `(angel s R)` could be defined as `not (sdemon s (fun a => not (R a)))`. In Coq which is an intuitionistic logic, these two notions are not dual. But, a big interest of Coq is that WP are *computed* inside the logic.

Actually, to simplify the definition of `angel` and have better properties, I impose `angel` to satisfy the following property:

```
forall R:A->Prop, (angel s R) <-> (exists a:A, (post s a) /\ (R a))
```

In particular, this implies that `angel` is monotonic (like `sdemon`):

```
forall (R1 R2:A->Prop), (forall a, R1 a -> R2 a) -> (angel s R1)->(angel s R2)
```

Moreover, using the following two properties (which derive from previous definitions), `pre` and `post` can now be defined from `sdemon` and `angel`:

```
(pre s) <-> (sdemon s (fun _ => True)).
forall (a:A), (post s a) <-> (angel s (fun b => a=b)).
```

Hence, instead of defining specifications using a `pre`/`post` pair, I now define them as the following triple:

```
Record K (A:Type) : Type := {
   sdemon: (A -> Prop) -> Prop;
   angel: (A -> Prop) -> Prop;
   WPcorrect: forall R : A -> Prop,
     ( sdemon R <-> ( sdemon (fun _ => True)
                       /\ forall a:A, (angel (fun b => a=b)) -> R a )
     ) /\ (angel R <-> (exists a:A, (angel (fun b => a=b)) /\ R a))
  }.
```

Actually, with this definition, I proved all standard properties of the WP-calculus without introducing abstract syntax for specifications.

In the following, I still continue to use `pre` and `post`, but these constructions are now derived from `sdemon` and `angel` using the previous equivalences. Refinement is directly defined by translating the pre/post semantics:

```
Definition refines (A:Type) (s1 s2:K A): Prop
 := (pre s2) -> (sdemon s1 (fun a => post s2 a)).
```

At last, I present below the definition of pure DSM operators through Coq formulae. Indeed, their `WPcorrect` component is not very human-friendly (see the real Coq code in [Bou06]). Hence, `val`, `bind` and `any` are defined such that

```
forall (A:Type) (a:A) (R:A->Prop),
  (sdemon (val a) R = R a) /\ (angel (val a) R = R a)
```

```
forall (A B:Type) (s1:(K A)) (s2:A->(K B)) (R:B->Prop),
   (sdemon (bind s1 s2) R = sdemon s1 (fun a => sdemon (s2 a) R))
/\  (angel (bind s1 s2) R = angel s1 (fun a => angel (s2 a) R))

forall (A:Type) (R:A->Prop),
  (sdemon (any A) R = forall a:A,R a) /\ (angel (any A) R = exists a:A,R a)
```

Now, I must define `sync`. But, WP of `sync` are not very simple. Thus, I derive `sync` from `require`, where `require` is satisfying:

```
forall (P:Prop) (R:unit->Prop),
  (sdemon (require P) R = (P /\ R tt)) /\ (angel (require P) R = R tt)
```

Below, `sync` is defined from other operators, assuming that `choice` and `ensure` are defined according to Section 3.1 (their definitions depend only on `bind`, `val` and `any`). This definition of `sync` is only valid in the pure DSM.

```
Definition sync (A B:Type) (s:A -> K B) :=
  seq (require (exists a:A, pre (s a)))
      (choice (fun b =>  seq (ensure (forall a, pre (s a) -> post (s a) b))
                             (val b))).
```

These definitions satisfy the DSM axioms. Moreover, on the programming part of the language (here `val` and `bind`), `sdemon` and `angel` perform CPS computations. Examples of WP computed by Coq are given in Section 5.

4.2 Construction of the State DSM from the Pure DSM

This section presents how the state DSM is derived from the pure DSM, by applying the state monad transformer. Intuitively, a monad transformer is a function from monads to monads that extends the input monad with specific constructions (see [Lia96, Ben02]). A monad transformer is simply given by a transformation on the type constructor `K`, by an embedding of the expressions of the input monads, and by a generic transformation on composition operators of the input monad. However, the properties of the specific operators of the input monad are not necessarily fully preserved in the output monad.

Hence, in our case, we have to prove that applying the state monad transformer on the pure DSM, we obtain a DSM (it is a state monad by construction). From now, constructions of the pure DSM are prefixed by "`F.`": the non-prefixed names are only used to denote constructions of the state DSM. It is easy to prove that the following definitions satisfy axioms of DSM (and of state monads):

```
Definition K (A:Type) := St -> F.K (St*A).
Definition refines (A:Type) (s1 s2:St -> F.K (St*A))
 := forall st:St, F.refines (s1 st) (s2 st).

Definition val (A:Type) (a:A): K A := fun st => F.val (st,a).
Definition bind (A B:Type) (s:K A) (f:A -> K B): K B
 := fun st => F.bind (s st) (fun p => let (stf,a):=p in (f a stf)).
Definition set (st:St): K unit := fun _ => F.val (st,tt).
Definition get: K St := fun st => F.val (st,st).
```

```
Definition any (A:Type): K A
 := fun st => F.bind (F.any A) (fun a => F.val (st,a)).
Definition sync (A B:Type) (s:A->K B) := fun st => F.sync (fun x => s x st).
```

Weakest-preconditions on the state DSM are also derived from the pure DSM:

```
Definition sdemon (A:Type) (s:K A) (R:St->A->Prop) (st:St)
  := F.sdemon (s st) (fun p => let (stf,a):=p in (R stf a)).
```

This method allows me also to extend the state DSM with exception-handling, using the exception monad transformers. It could also probably be applied with many other monad transformers.

5 WP-Computations in Interactive Refinement Proofs

In a refinement prover like B, refinement formulae are automatically translated into first-order "proof obligations" through WP-computations. This is expressed in the pure DSM by the following rule:

```
Lemma wp_refines: forall (A:Type) (s1 s2:F.K A),
 (F.pre s2 -> F.sdemon s1 (fun a => F.post s2 a)) -> F.refines s1 s2.
```

Indeed, application of this lemma replaces the current refinement goal by the formula in hypothesis, that Coq can then simplify by computing effectively "`F.pre s2`", "`F.sdemon s1`" and "`F.post s2`".

Let me run this rule on an example. First, I introduce an operator `abort` that sets a false precondition, and thus, behaves like an error-raising operator. Given a type `A`, `abort A` could be defined as `seq (require False) (any A)`. Here, in the pure DSM, I use an (observationally) equivalent definition, but with optimized WP. Hence, `F.abort` is defined such that

```
forall (A:Type) (R:A->Prop),
  (F.sdemon (F.abort A) R = False) /\ (F.angel (F.abort A) R = False).
```

Second, I introduce two function definitions:

– Function `pred` computes the predecessor of a natural n, or aborts if n is zero.

```
Definition pred (n:nat): F.K nat :=    match n with | 0 =>   F.abort nat
                                                    | (S p) => F.val p
                                       end.
```

– Function `minus` computes $n - m$ when $n \geq m$. If $n < m$, it aborts. It is defined by structural recursion over parameter m as indicated by **struct** keyword.

```
Fixpoint minus (n m:nat) {struct m}: F.K nat :=
      match m with  | 0 => F.val n
                    | (S p) => F.bind (pred n) (fun n' => minus n' p)
      end.
```

At last, let me consider the following three goals (below literals are expanded in Peano numbers by the Coq parser). The first goal does not hold, because the left side aborts, whereas the right side does not. The second goal holds, because the right side aborts. The third goal is an expected property of `minus`.

```
forall (n:nat), F.refines (minus 100 (500+n)) (F.any nat).
forall (n:nat) (sp:F.K nat), F.refines sp (minus 100 (500+n)).
forall (n:nat), F.refines (minus (500+n) 500) (F.val n).
```

After introduction of variables and application of `wp_refines`, the first goal is reduced to `(nat -> True) -> False`. In the same way, the second goal is reduced to `False -> F.sdemon sp (fun _:nat => False)` and the third goal is reduced to `True -> n = n`. Here, we benefit from the fact that `500+n` is reduced by Coq to $\texttt{S}^{500}$ `n`. Of course, Coq can discharge automatically the two last formulae.

Moreover, combining induction over `m`, transitivity of refinement and lemma `wp_refines`, I have established the correctness of `minus` in Coq interactive prover:

```
Lemma minus_correctness: forall (m n:nat), m <= n ->
  F.refines (minus n m)
            (F.choice (fun k => F.seq (F.ensure (n=m+k)) (F.val k))).
```

This small example illustrates that refinement is very convenient to handle partial functions in Coq and that computations of WP can involve structural recursion and pattern-matching.

5.1 Mixing Deductions and WP-Computations in the Pure DSM

In the presence of higher-order functions, `wp_refines` rule does not suffice. For example, because of higher-order parameters, `sdemon` and `angel` are not necessarily eliminated from the resulting formula. Moreover, reasoning about higher-order functions often requires to find a good instantiation of a higher-order lemma. Hence, the user needs to control WP-computations such that in refinement proofs, deductions and WP-computations may be interlaced. The deduction rules in refinement proofs are: reflexivity, transitivity of refinement, associativity, monotonicity of `bind`, and other lemmas derived from DSM axioms.

In the pure DSM, three simplifying rules involving `bind` operator are given below. Property `bind_simpl_left` indicates that "`(F.refines (F.bind s1 s2) s3)`" can be deduced from "`(F.sdemon s1 (fun a => F.refines (s2 a) s3))`". If `s1` is simple enough, this last formula is simplified such that `s1` and `sdemon` do not appear any more. Hence, this lemma allows to perform a kind of partial evaluation of `bind` into the left side of the refinement goal. The two others lemmas correspond respectively to a simplification of `bind` in the right part of the refinement goal, or in both part of the refinement goal.

```
Lemma bind_simpl_left: forall (A B:Type)(s1:F.K A)(s2:A->F.K B)(s3:F.K B),
   F.sdemon s1 (fun a=>F.refines (s2 a) s3) -> F.refines (F.bind s1 s2) s3.

Lemma bind_simpl_right: forall (A B:Type)(s1:F.K A)(s2:A->F.K B)(s3:F.K B),
   F.angel s1 (fun a=>F.refines s3 (s2 a)) -> F.refines s3 (F.bind s1 s2).

Lemma bind_simpl_both: forall (A B:Type) (s1 s3:F.K A) (s2 s4:A -> F.K B),
  F.sdemon s1 (fun a => F.refines (s2 a) (s4 a))
   -> F.refines s1 s3 -> F.refines (F.bind s1 s2) (F.bind s3 s4).
```

5.2 Mixing Deductions and WP-Computations in the State DSM

I now briefly explain how the previous ideas are extended to reason in the state DSM. In the state DSM, it is convenient to reason with a restricted form of

refinement formulae that compares specifications for a given initial state. Below, I define `refInEnv` relation from `refines`:

```
Definition refInEnv (st:St) (A:Type) (s1 s2: K A)
 :=  (refines (seq (set st) s1) (seq (set st) s2)).
Implicit Arguments refInEnv [A].
```

Actually, refinement proofs can use `refInEnv` instead of `refines`, because of the following property:

```
forall (A:Type)(s1 s2:K A), refines s1 s2 <-> forall st, refInEnv st s1 s2.
```

The interest of `refInEnv` is that conditions over the initial state are propagated from the hypotheses without a loss of information, as exemplified below on the state DSM version of `bind_simpl_both`.

```
Lemma bind_simpl_both: forall (A B:Type)(s1 s3:K A)(s2 s4:A->K B)(st:St),
  sdemon s1 (fun stf a => refInEnv stf (s2 a) (s4 a)) st
    -> refInEnv st s1 s3 -> refInEnv st (bind s1 s2) (bind s3 s4).
```

6 Proof of Programs with the State DSM

As DSM are monads, equational reasoning presented Section 2 has actually been performed in the state DSM (see [Bou07]). I have also proved other small higher-order imperative functions which have led me to design the simplification rules given in the previous section (see [Bou06]).

Thus, here, I first show how Hoare logic reasoning on first order imperative programs is encoded into the state DSM. Then, I explain how while-loops are expressed in the state DSM, for partial or total correctness. These ideas have been experimented with the total-correctness proof of a sort on arrays [Bou06].

6.1 Hoare Specifications

On the state DSM, given a predicate `q: St -> A -> Prop`, I define `absPost q` as the specification of imperative expressions that admits `q` as postcondition:

```
Definition absPost (A:Type) (q:St -> A -> Prop) : K A :=
   choice (fun stf =>
     choice (fun a => seq (ensure (q stf a)) (seq (set stf) (val a)))).
Implicit Arguments absPost [A].
```

Given `e` of type `(K A)` in the state DSM, the Hoare specification of `e` (see Section 2.2) is now expressed by the following formula:

```
forall sti:St, (P sti) -> refInEnv sti e (absPost (Q sti))
```

The conclusion of this formula can be simplified using the following rule:

```
Lemma absPost2wp: forall (A:Type) (s:K A) (q:St->A->Prop) (st:St),
    (sdemon s q st) -> (refInEnv st s (absPost q)).
```

Applying this rule generates proof obligations corresponding to those of Hoare logic, except for function calls. Indeed, in a function call, my WP-computations unfold the function definition (functions are used as white-boxes), whereas standard WP-computations use the specification of the function (functions are used as black-boxes). Hence, before to apply my WP-simplifications, the user has to replace function calls by their specification using monotonicity rules: currently, the user-control on WP computations comes at this price.

6.2 Non-terminating Expressions

I illustrate here on a while-loop operator that non-terminating expressions can be represented by smallest fixpoints in partial correctness semantics, or by greatest fixpoints in total correctness semantics. First, I define the "unfolding" of a while-loop computation `w` where `cond` and `body` represent respectively the condition and the body of this computation.

```
Definition unfoldW (cond:K bool) (body w:K unit): K unit :=
  If cond Then (seq body w) Else skip.
```

Then, `while cond body` is defined as the smallest fixpoint of `unfoldW cond body`.

```
Definition while (cond:K bool) (body:K unit) := sfix (unfoldW cond body).
```

Operator `while` corresponds to a while-loop in partial correctness: termination is not guaranteed. In the state DSM, I have defined from `while`, a `whileWF` operator corresponding to total correctness. Actually, `whileWF cond body` calls `while cond body` under a *precondition*, which expresses that, in the state transformation induced by the sequence of `cond` and `body`, the state strictly decreases with respect to a well-founded relation. In other words, `whileWF` requires a property implying its termination. When `whileWF` appears in the left side of a refinement goal, this property must be proved. Moreover, `whileWF cond body` is refined by any fixpoint of `unfoldW cond body`, because under its assumption of termination, there is a unique fixpoint. Hence, `whileWF` is a greatest fixpoint (see [Bou06]).

7 Conclusion

WP have been invented in [Dij75], inspiring the *refinement calculus* of [Bac78]. Then, the calculus has been developed by many other authors. In particular, [Bun97] presents a refinement for a higher-order expressions language which shares many aspects with my DSM theory. In parallel, [Gor88] promoted the use of higher-order logics to formalize particular programming logics like Hoare logic. It inspired many works. Among them, [Bod99] formalizes B in Coq using a pre/post semantics. Actually, [vW94] inspired my first attempts to formalize refinement calculus in Coq. I was also interested in combining monads with Hoare logic [Fil03, Sch03, Nan06] and encoding WP as CPS [Aud99].

With respect to all these works, the main contributions of my formalization seem to be: to extend refinement calculus with higher-order functions and structural recursion, to propose its modular construction using monad transformers, to embed its fixpoint theory into constructive type theory, to program WP as continuations in an *intuitionistic* logic, and to define simplification rules of refinement formulae in interactive proofs. As a result, Coq now embeds the refinement calculus of [Mor90]. The refinement calculus of [Bac98] has stronger properties, but its perfect symmetry seems to deeply rely on the axiom of Excluded-Middle.

In the future, refinement calculus could be for Coq what monads have been to Haskell. But there is a long road ahead: for instance, to address the frame problem and data refinement. A long-term goal is to provide a notion of "abstract machines" like in B [Abr96]: combining abstract machines and higher-order functions would give an object-oriented flavor to the specification language.

References

[Abr96] Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)

[Aud99] Audebaud, P., Zucca, E.: Deriving Proof Rules from Continuation Semantics. Formal Aspects of Computing 11(4), 426–447 (1999)

[Bac78] Back, R.-J.: On the correctness of refinement steps in program development. Ph.D. thesis, University of Helsinki (1978)

[Bac98] Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)

[Beh99] Behm, P., Benoit, P., et al.: Météor: A Successful Application of B in a Large Project. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)

[Ben02] Benton, N., Hughes, J., et al.: Monads and Effects. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 42–122. Springer, Heidelberg (2002)

[Bod99] Bodeveix, J.-P., Filali, M., et al.: A Formalization of the B-Method in Coq and PVS. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, Springer, Heidelberg (1999)

[Bou06] Boulmé, S.: Higher-Order Refinement In Coq (reports and Coq files). (2006) Web page: http://www-lsr.imag.fr/users/Sylvain.Boulme/horefinement/

[Bou07] Boulmé, S.: Higher-Order imperative enumeration of binary trees in Coq. (2007) Online paper: http://.../Sylvain.Boulme/horefinement/enumBTdesc.pdf

[Bun97] Bunkenburg, A.: Expression Refinement. Ph.D. thesis, Computing Science Department, University of Glasgow (1997)

[Coq88] Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation 76, 95–120 (1988)

[Coq04] Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.0. Logical Project, INRIA-Rocquencourt (2004)

[Dij75] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)

[Fil03] Filliâtre, J.-C.: Verification of Non-Functional Programs using Interpretations in Type Theory. Journal of Functional Programming 13(4), 709–745 (2003)

[Gor88] Gordon, M.J.C.: Mechanizing Programming Logics in Higher-Order Logic. In: Current Trends in Hardware Verification and Automatic Theorem Proving, Springer, Heidelberg (1988)

[Hon05] Honda, K., Berger, M., et al.: An Observationally Complete Program Logic for Imperative Higher-Order Functions. In: IEEE Symp. on LICS'05 (2005)

[Lia96] Liang, S., Hudak, P.: Modular Denotational Semantics for Compiler Construction. In: Nielson, H.R. (ed.) ESOP 1996. LNCS, vol. 1058, pp. 219–234. Springer, Heidelberg (1996)

[Mog91] Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)

[Mor87] Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Science of Computer Programming 9(3), 287–306 (1987)

[Mor90] Morgan, C.: Programming from Specifications. Prentice-Hall, Englewood Cliffs (1990)

[Nan06] Nanevski, A., Morrisett, G., et al.: Polymorphism and separation in Hoare Type Theory. In: Nanevski, A., Morrisett, G. (eds.) Proc. of ICFP'06, ACM Press, New York (2006)

[Pey93] Jones, S.L.P., Wadler, P.: Imperative functional programming. In: Proc. of the 20th symposium on POPL, ACM Press, New York (1993)

[PM93] Paulin-Mohring, C.: Inductive Definitions in the System Coq - Rules and Properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, Springer, Heidelberg (1993)

[Sch03] Schröder, L., Mossakowski, T.: Monad-independent Hoare logic in HasCasl. In: Pezzé, M. (ed.) ETAPS 2003 and FASE 2003. LNCS, vol. 2621, Springer, Heidelberg (2003)

[vW94] von Wright, J.: Program refinement by theorem prover. In: 6th Refinement Workshop, Springer, Heidelberg (1994)

Computation by Prophecy

Ana Bove[1] and Venanzio Capretta[2]

[1] Department of Computer Science and Engineering
Chalmers University of Technology,
412 96 Göteborg, Sweden
Tel.: +46-31-7721020, Fax: +46-31-7723663
bove@chalmers.se
[2] Computer Science Institute (iCIS),
Radboud University Nijmegen,
Tel.: +31-24-3652631, Fax: +31-24-3652525
venanzio@cs.ru.nl

Abstract. We describe a new method to represent (partial) recursive functions in type theory. For every recursive definition, we define a co-inductive type of *prophecies* that characterises the traces of the computation of the function. The structure of a prophecy is a possibly infinite tree, which is coerced by linearisation to a type of partial results defined by applying the delay monad to the co-domain of the function. Using induction on a weight relation defined on the prophecies, we can reason about them and prove that the formal type-theoretic version of the recursive function, resulting from the present method, satisfies the recursive equations of the original function. The advantages of this technique over the method previously developed by the authors via a special-purpose accessibility (domain) predicate are: there is no need of extra logical arguments in the definition of the recursive function; the function can be applied to any element in its domain, regardless of termination properties; we obtain a type of partial recursive functions between any two given types; and composition of recursive functions can be easily defined.

1 Introduction

The implementation of general recursive functions in type theory has received wide attention in the last decade, and several methods to implement recursive algorithms and reason about them have been described in the literature.

We give a survey of different approaches in the *related work* section of a previous article [6]. After the publication of that paper, the type-theory based proof assistant *Coq* [14,3] has been extended with a new feature to define total recursive functions that expands the native structural recursion. The feature `Function` (based on the work by Balaa and Bertot [1]) facilitates the definition of total functions that have a well-founded relation associated to them. In addition, the tactic `functional induction` (based on the work by Barthe and Courtieu [2]) has been added to the system, providing induction principles that follow the definition of structurally recursive functions. These contributions enlarge

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 70–83, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the class of total recursive functions that can be studied in *Coq*. There are, however, functions that lay outside the reach of these new features, for example, all strictly partial recursive functions. Our main goal in this work is to provide a good general type-theoretic treatment of recursive computations (partial or not).

We have previously worked on two methods to tackle this issue. The first method [6,4,7] consists in characterising the inputs on which the function terminates by an inductive (domain) predicate easily defined from the recursive equations, and then defining the function itself by recursion on the proof that the input argument satisfies this domain predicate. The second method [8] consists in defining co-inductive types of partial elements and implementing general recursive functions by co-recursion on these types.

Both methods have pros and cons.

The first method has two main advantages. First, the type-theoretic equations defining the function are almost identical to the recursive equations in a functional programming language, except that the former require proof terms for the domain predicate as additional arguments. Second, it is easy to reason about a function formalised with this method by induction on its domain predicate. A disadvantage is that the application of a function to a certain input always requires a proof that the input satisfies the domain predicate defined for the function; as a consequence, the function can only be applied to arguments for which we know how to construct such a proof.

On the other hand, a function formalised with the second method requires no additional logical arguments in its definition and, hence, the function can be applied even to arguments for which it might not terminate. The main disadvantage is that reasoning about functions formalised in this way is much more involved than with the first method.

Here, we show a new way of representing general recursive functions in intensional type theory, where we adapt some of the ideas of the first method to facilitate the definition of functions with the second one and the subsequent reasoning about their formalisation. In other words, we propose a co-inductive version of the Bove/Capretta method.

Throughout this paper we will use a generalisation of the Fibonacci function as a running example to illustrate the notions we are presenting. This algorithm is defined as follows:

$$
\begin{aligned}
&\mathsf{F}\colon \mathbb{N} \to \mathbb{N}\\
&\quad \mathsf{F}\ 0 = \mathsf{a}\\
&\quad \mathsf{F}\ 1 = \mathsf{b} + \mathsf{c} * \mathsf{F}\ (g\ 0)\\
&\quad \mathsf{F}\ (\mathsf{S}\ (\mathsf{S}\ n)) = \mathsf{F}\ (g\ n) + \mathsf{F}\ (g\ (\mathsf{S}\ n)).
\end{aligned}
$$

where a, b and c are natural numbers and $g\colon \mathbb{N} \to \mathbb{N}$. We can get the standard Fibonacci sequence by letting $\mathsf{a} = 1, \mathsf{b} = 1, \mathsf{c} = 0$, and letting g be the identity function.

Observe that the totality of F depends on the definition of g: with the same choices of $\mathsf{a}, \mathsf{b}, \mathsf{c}$, but choosing for g the successor function, we obtain a function that is defined in 0 but diverges for any other value.

This paper is organised as follows. In Section 2, we recall how to define a recursive function with the inductive Bove/Capretta method. Sections 3 and 4 present, respectively, the *prophecies*, that is, the co-inductive version of the Bove/Capretta method, and their evaluation procedure. In Section 5, we show the validity of this new method. Finally, Section 6 presents some conclusions.

We have formalised the running example in the proof assistant *Coq*. As an additional example, we have also formalised the *quicksort* algorithm using the method we present here. The files of both formalisations are available on the web at the address: `http://www.cs.ru.nl/~venanzio/Coq/prophecy.html`.

2 Overview of the Bove/Capretta Method

We outline the general steps of the inductive Bove/Capretta method by showing how an algorithm defined by general recursive equations can be formalised in type theory.

Let f be a recursive function defined by a series of (non-overlapping) equations. We assume that the informal definition of the function has the following form:

$$\begin{aligned} &f\colon A \to B \\ &\quad \cdots \\ &\quad f\ p = e[(f\ p_1), \ldots, (f\ p_n)] \\ &\quad \cdots \end{aligned}$$

where "$f\ p = \cdots$" is one of the recursive equations defining f. The term p is a pattern, possibly containing variables that can occur in the right-hand side of the equation. The right-hand side is an expression e, recursively calling f on the arguments $p_1, \ldots, p_n$.

For an argument a matching the pattern p, there are three phases in the computation of $(f\ a)$: first, from the argument a, the recursive arguments $a_1, \ldots, a_n$ are computed; then, the program f is recursively applied to these arguments; and finally, the results of the recursive calls are fed into the operator e to obtain the final result. This three-steps process is general and can be used to give a very abstract notion of computable function [9].

We recall that the Bove/Capretta method consists in characterising the domain of a function by an inductive predicate with (in principle) one constructor for each of the equations defining the function[1]. The constructor corresponding to each equation takes as parameters assumptions stating that the recursive arguments in the equation satisfy the domain predicate. The general form of the domain predicate for the function f above is as follows:

$$\begin{aligned} &\mathsf{D}_f\colon A \to \mathsf{Prop} \\ &\quad \cdots \\ &\quad \mathsf{d}_p\colon (\Gamma_p)(\mathsf{D}_f\ p_1) \to \cdots \to (\mathsf{D}_f\ p_n) \to (\mathsf{D}_f\ p) \\ &\quad \cdots \end{aligned}$$

[1] Equations with a case-expression on their right-hand side can give raise to several constructors. See [6] for a more detailed explanation on how to handle these equations.

where Γ_p is a context local to the equation, comprising the variables that occur in p.

For our example we have

$$\begin{aligned}
&\mathsf{D_F} \colon \mathbb{N} \to \mathsf{Prop}\\
&\quad \mathsf{d_0} \colon (\mathsf{D_F}\ 0)\\
&\quad \mathsf{d_1} \colon (\mathsf{D_F}\ (g\ 0)) \to (\mathsf{D_F}\ 1)\\
&\quad \mathsf{d_S} \colon (n \colon \mathbb{N})(\mathsf{D_F}\ (g\ n)) \to (\mathsf{D_F}\ (g\ (\mathsf{S}\ n))) \to (\mathsf{D_F}\ (\mathsf{S}\ (\mathsf{S}\ n)))
\end{aligned}$$

The type-theoretic version of f takes as an extra argument a proof that the input satisfies the domain predicate, and it is defined by structural recursion on this extra argument:

$$\begin{aligned}
&f \colon (y \colon A)(\mathsf{D}_f\ y) \to B\\
&\quad \cdots\\
&\quad f\ p\ (\mathsf{d}_p\ \overrightarrow{x}\ h_1\ \cdots\ h_n) = e[(f\ p_1\ h_1), \ldots, (f\ p_n\ h_n)]\\
&\quad \cdots
\end{aligned}$$

For the F function we get:

$$\begin{aligned}
&\mathsf{F} \colon (m \colon \mathbb{N})(\mathsf{D_F}\ m) \to \mathbb{N}\\
&\quad \mathsf{F}\ 0\ \mathsf{d_0} = \mathsf{a}\\
&\quad \mathsf{F}\ 1\ (\mathsf{d_1}\ h) = \mathsf{b} + \mathsf{c} * (\mathsf{F}\ (g\ 0)\ h)\\
&\quad \mathsf{F}\ (\mathsf{S}\ (\mathsf{S}\ n))\ (\mathsf{d_S}\ n\ h_1\ h_2) = \mathsf{F}\ (g\ n)\ h_1 + \mathsf{F}\ (g\ (\mathsf{S}\ n))\ h_2
\end{aligned}$$

For a complete description of this method and for more examples of its application, the reader is referred to [6,4,7].

3 Views and Prophecies

We can consider an alternative representation of the domain D_f of f where we ignore the elements of A altogether. Below we present the general form of this new domain type which we call A_f, and its corresponding instance for the F function which we call $\mathbb{N}_\mathsf{F}$.

$$\begin{aligned}
&A_f \colon \mathsf{Type}\\
&\quad \cdots\\
&\quad \mathsf{c}_p \colon \Gamma_p \to \underbrace{A_f \to \cdots \to A_f}_{n \text{ times}} \to A_f\\
&\quad \cdots
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathbb{N}_\mathsf{F} \colon \mathsf{Type}\\
&\quad \mathsf{c_0} \colon \mathbb{N}_\mathsf{F}\\
&\quad \mathsf{c_1} \colon \mathbb{N}_\mathsf{F} \to \mathbb{N}_\mathsf{F}\\
&\quad \mathsf{c_S} \colon \mathbb{N} \to \mathbb{N}_\mathsf{F} \to \mathbb{N}_\mathsf{F} \to \mathbb{N}_\mathsf{F}
\end{aligned}$$

Using the terminology of [12], we call A_f a *view* of the domain.

We now formalise f as a function on this new type:

$$\begin{aligned}
&f_\star \colon A_f \to B\\
&\quad \cdots\\
&\quad f_\star\ (\mathsf{c}_p\ \overrightarrow{x}\ t_1\ \cdots\ t_n) = e[(f_\star\ t_1), \ldots, (f_\star\ t_n)]\\
&\quad \cdots
\end{aligned}$$

The variables in Γ_p are still required as parameters of the constructor c_p, even if they do not occur in the arguments of the branches, since they may occur in the operator e.

For our example we obtain

$$\begin{aligned}
&\mathsf{F}_\star \colon \mathbb{N}_\mathsf{F} \to \mathbb{N}\\
&\quad \mathsf{F}_\star\ \mathsf{c}_0 = \mathsf{a}\\
&\quad \mathsf{F}_\star\ (\mathsf{c}_1\ h) = \mathsf{b} + \mathsf{c} * (\mathsf{F}_\star\ h)\\
&\quad \mathsf{F}_\star\ (\mathsf{c}_\mathsf{S}\ n\ h_1\ h_2) = \mathsf{F}_\star\ h_1 + \mathsf{F}_\star\ h_2
\end{aligned}$$

We can see A_f as the type of *abstract inputs* for the function f. It is easy to obtain the elements in A_f from the elements of A satisfying D_f:

$$\begin{aligned}
&\imath_f \colon (y\colon A)(\mathsf{D}_f\ y) \to A_f\\
&\quad \cdots\\
&\quad \imath_f\ p\ (\mathsf{d}_p\ \overrightarrow{x}\ h_1 \cdots h_n) = \mathsf{c}_p\ \overrightarrow{x}\ (\imath_f\ p_1\ h_1) \cdots (\imath_f\ p_n\ h_n)\\
&\quad \cdots
\end{aligned}$$

The reverse direction is however not possible, since elements in A_f can be constructed in an arbitrary way, completely disconnected from the behaviour of the function f. The possible projections of those elements in A have absolutely no reason to satisfy D_f. Consider for example the element $(\mathsf{c}_\mathsf{S}\ 10\ \mathsf{c}_0\ \mathsf{c}_0)$ for our example of the function F.

We now try to dualise the inductive Bove/Capretta method by using a co-inductive approach. The idea is that, instead of defining an inductive characterisation of the domain, we define a co-inductive characterisation of the co-domain of the function:

$$\begin{aligned}
&\mathsf{CoInductive}\ B^f \colon \mathsf{Type}\\
&\qquad \cdots\\
&\qquad \mathsf{b}_e \colon \Gamma_e \to \underbrace{B^f \to \cdots \to B^f}_{n \text{ times}} \to B^f\\
&\qquad \cdots
\end{aligned}$$

where Γ_e is the context comprising the variables occurring free in e. In principle, Γ_e could coincide with Γ_p; however, some of the variables in the pattern p may not be needed for the computation of e and hence, they could be omitted in Γ_e[2].

Observe that the definition of B^f is identical to that of A_f except for the contexts appearing in the equations of both definitions, and for the fact that it is a co-inductive definition instead of an inductive one.

The co-inductive characterisation of the co-domain of F is defined as

$$\begin{aligned}
&\mathsf{CoInductive}\ \mathbb{N}^\mathsf{F} \colon \mathsf{Type}\\
&\qquad \mathsf{f}_0 \colon \mathbb{N}^\mathsf{F}\\
&\qquad \mathsf{f}_1 \colon \mathbb{N}^\mathsf{F} \to \mathbb{N}^\mathsf{F}\\
&\qquad \mathsf{f}_\mathsf{S} \colon \mathbb{N}^\mathsf{F} \to \mathbb{N}^\mathsf{F} \to \mathbb{N}^\mathsf{F}.
\end{aligned}$$

[2] Actually, in some cases, not even all the variables that are free in e need to be included in Γ_e; for example, the Natural argument n is omitted from the constructor f_S in the definition of $\mathbb{N}^\mathsf{F}$.

We now give a version of f that has B^f as co-domain:

$$
\begin{aligned}
&f^\star\colon A \to B^f \\
&\quad \cdots \\
&\quad f^\star\ p = \mathsf{b}_e\ \overrightarrow{x'}\ (f^\star\ p_1)\cdots(f^\star\ p_n) \\
&\quad \cdots
\end{aligned}
$$

This definition is sound because the co-recursive calls $(f^\star\ p_1),\ldots,(f^\star\ p_n)$ are guarded by the constructor b_e (see [11] for further reading on this issue).

We can see B^f as the type of *abstract outputs* in the same way we saw A_f as the type of abstract inputs. In a certain sense, the elements of B^f are a prediction of the structure of the result. For this reason we will call them *prophecies*.

The version of F using a co-inductive co-domain is

$$
\begin{aligned}
&\mathsf{F}^\star\colon \mathbb{N} \to \mathbb{N}^{\mathsf{F}} \\
&\quad \mathsf{F}^\star\ 0 = \mathsf{f}_0 \\
&\quad \mathsf{F}^\star\ 1 = \mathsf{f}_1\ (\mathsf{F}^\star\ (g\ 0)) \\
&\quad \mathsf{F}^\star\ (\mathsf{S}\ (\mathsf{S}\ n)) = \mathsf{f}_{\mathsf{S}}\ (\mathsf{F}^\star\ (g\ n))\ (\mathsf{F}^\star\ (g\ (\mathsf{S}\ n))).
\end{aligned}
$$

4 Evaluating the Prophecies

The relation between B^f and B is not very clear: since the elements of B^f may represent infinite computations, they do not always correspond to elements of B. Instead, we can find a correspondence between B^f and the type of partial elements of B defined in [8] as:

$$
\begin{aligned}
\mathsf{CoInductive}\ B^\nu &: \mathsf{Type} \\
\mathsf{return} &: B \to B^\nu \\
\mathsf{step} &: B^\nu \to B^\nu
\end{aligned}
$$

Following [8], we use the notations $\ulcorner b \urcorner$ for $(\mathsf{return}\ b)$ and $\triangleright x$ for $(\mathsf{step}\ x)$. The definition above means that an element of B^ν can be either a finite sequence of $\triangleright$ steps followed by the return of a value of B, or an infinite sequence of $\triangleright$ steps. So B^ν represents the partial elements of B.

Our goal is then to represent the program f in type theory as a function with type $A \to B^\nu$. To accomplish this, we need to define an *evaluation* operator $\mathsf{evaluate}_f\colon B^f \to B^\nu$. Notice that B^f has a tree structure, since elements constructed by b_e correspond to nodes of branching degree n, while B^ν has a linear structure given by a sequence of $\triangleright$ steps. The problem consists in linearising a tree or, in computational terms, sequentialising a parallel computation. To achieve this, we create a stack of calls and we execute them sequentially. Every call, when evaluated, can in turn generate new calls that are added to the stack. The stack is represented by a vector. The empty vector is denoted by $\langle\ \rangle$. We use the notation $\langle x_1,\ldots,x_n;\overrightarrow{v}\rangle$ to denote the vector whose first n elements are $x_1,\ldots,x_n$ and whose elements from the $(n+1)$th on are given by the vector $\overrightarrow{v}$. In the case where $\overrightarrow{v}$ is the empty vector, we simply write $\langle x_1,\ldots,x_n\rangle$. In what follows, A^k represents the type of vectors of elements in A with length k.

The evaluation operator also has an extra parameter: a function that will compute the final result from the results of the recursive calls on the elements in the stack. This is similar to continuation-passing programming [13].

The co-recursive evaluation function θ_f, defined below, performs the sequentialisation we just mentioned. The function is defined by cases on the length of the vector and, when the vector is not empty, by cases on its first element.

$$\begin{array}{l}
\theta_f : (k : \mathbb{N})(B^f)^k \to (B^k \to B) \to B^\nu \\
\quad \theta_f\ 0\ \langle\ \rangle\ h = \ulcorner h\ \langle\ \rangle \urcorner \\
\quad \cdots \\
\quad \theta_f\ (\mathsf{S}\ k)\ \langle (\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n); \overrightarrow{v} \rangle\ h = \triangleright(\theta_f\ (n+k)\ \langle y_1, \ldots, y_n; \overrightarrow{v} \rangle\ h') \\
\qquad \text{where } h'\ \langle z_1, \ldots, z_n; \overrightarrow{u} \rangle = h\ \langle e[z_1, \ldots, z_n]; \overrightarrow{u} \rangle \\
\quad \cdots
\end{array}$$

Notice that the recursive call in the function θ_f are guarded by the constructor $\triangleright$, hence this is a valid co-fixpoint definition.

In our running example for the function F we obtain:

$$\begin{array}{l}
\theta_\mathsf{F} : (k : \mathbb{N})(\mathbb{N}^\mathsf{F})^k \to (\mathbb{N}^k \to \mathbb{N}) \to \mathbb{N}^\nu \\
\theta_\mathsf{F}\ 0\ \langle\ \rangle\ h = \ulcorner h\ \langle\ \rangle \urcorner \\
\theta_\mathsf{F}\ (\mathsf{S}\ k)\ \langle \mathsf{f_0}; \overrightarrow{v} \rangle\ h = \triangleright(\theta_\mathsf{F}\ k\ \overrightarrow{v}\ h') \\
\qquad \text{where } h'\ \overrightarrow{u} = h\ \langle \mathsf{a}; \overrightarrow{u} \rangle \\
\theta_\mathsf{F}\ (\mathsf{S}\ k)\ \langle (\mathsf{f_1}\ y); \overrightarrow{v} \rangle\ h = \triangleright(\theta_\mathsf{F}\ (\mathsf{S}\ k)\ \langle y; \overrightarrow{v} \rangle\ h') \\
\qquad \text{where } h'\ \langle z; \overrightarrow{u} \rangle = h\ \langle (\mathsf{b} + \mathsf{c} * z); \overrightarrow{u} \rangle \\
\theta_\mathsf{F}\ (\mathsf{S}\ k)\ \langle (\mathsf{f_S}\ y_1\ y_2); \overrightarrow{v} \rangle\ h = \triangleright(\theta_\mathsf{F}\ (2+k)\ \langle y_1, y_2; \overrightarrow{v} \rangle\ h') \\
\qquad \text{where } h'\ \langle z_1, z_2; \overrightarrow{u} \rangle = h\ \langle (z_1 + z_2); \overrightarrow{u} \rangle
\end{array}$$

The evaluation function, both in its general form and its instantiation for our example, is defined as follows:

$$\begin{array}{ll}
\mathsf{evaluate}_f : B^f \to B^\nu & \mathsf{evaluate}_\mathsf{F} : \mathbb{N}^\mathsf{F} \to \mathbb{N}^\nu \\
\quad \mathsf{evaluate}_f\ y = \theta_f\ 1\ \langle y \rangle\ (\lambda\langle z \rangle.z) & \quad \mathsf{evaluate}_\mathsf{F}\ y = \theta_\mathsf{F}\ 1\ \langle y \rangle\ (\lambda\langle z \rangle.z)
\end{array}$$

where $\lambda\langle z \rangle.z$ denotes the function giving the only element of a vector of length one.

Finally, we define the desired functions f and F:

$$\begin{array}{ll}
f : A \to B^\nu & \mathsf{F} : \mathbb{N} \to \mathbb{N}^\nu \\
\quad f\ x = \mathsf{evaluate}_f\ (f^\star\ x) & \quad \mathsf{F}\ n = \mathsf{evaluate}_\mathsf{F}\ (\mathsf{F}^\star\ n).
\end{array}$$

5 Validity of the Prophecy Method

We want to prove that the formal version of the function f defined with the prophecy method is a correct implementation of the informal recursive function. Specifically, we want to prove the validity of the equations defining the function.

Remember that f may return an element consisting of infinite computation steps when applied to certain inputs. The inductive relation (defined in [8])

$\mathsf{Value}\colon A^\nu \to A \to \mathsf{Prop}$, for which we use the notation $(_ \downarrow _)$, characterises terminating computations. The expression $(x \downarrow a)$ states that the element x of A^ν converges to the value a in A. Its inductive definition has two rules:

$$\frac{}{\ulcorner a \urcorner \downarrow a} \qquad\qquad \frac{x \downarrow a}{\triangleright x \downarrow a}.$$

We now formulate the recursive equations in terms of $(_ \downarrow _)$ and we prove that our implementation of the function satisfies them.

For each non-recursive equation $f\ p = a$ in the informal definition of f, where a may contain occurrences of the variables in Γ but no recursive calls to f, we would like to show that the formal version of f satisfies

$$\forall \overrightarrow{x} : \Gamma, f\ p \downarrow a;$$

where $\overrightarrow{x}$ are the variables defined in the context Γ, and for each recursive equation of the form $f\ p = e[(f\ p_1), \ldots, (f\ p_n)]$ in the informal definition of f, we would like to show that its formalisation is such that

$$\forall \overrightarrow{x} : \Gamma, \forall r_1, \ldots, r_n : B, (f\ p_1) \downarrow r_1 \to \cdots \to (f\ p_n) \downarrow r_n \to (f\ p) \downarrow e[r_1, \ldots, r_n].$$

For our example function F, we want to prove the following three statements

$$\begin{aligned}
&(\mathsf{F}\ 0) \downarrow \mathsf{a}, \\
&\forall m\colon \mathbb{N}, (\mathsf{F}\ (g\ 0)) \downarrow m \to (\mathsf{F}\ 1) \downarrow (\mathsf{b} + \mathsf{c} * m), \\
&\forall n, m_1, m_2\colon \mathbb{N}, (\mathsf{F}\ (g\ n)) \downarrow m_1 \to (\mathsf{F}\ (g\ (\mathsf{S}\ n))) \downarrow m_2 \to \\
&\qquad\qquad (\mathsf{F}\ (\mathsf{S}\ (\mathsf{S}\ n)))) \downarrow (m_1 + m_2).
\end{aligned}$$

On the road to proving these results, let us consider more closely the meaning of prophecies. A prophecy can be seen as the tree representation of the computation of the result of an expression. That is, it would be the computation trace, if parallel evaluation were allowed. For example the prophecy

$$\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n$$

specifies a parallel computation in which we first evaluate the subtrees $y_1, \ldots, y_n$ and, if all these computations terminate giving $r_1, \ldots, r_n$ as result, respectively, then we obtain the output by computing the expression $e[r_1, \ldots, r_n]$.

Recall that, since types of partial elements like B^ν represent computations in a sequential model, we could not directly define the evaluation of a prophecy following the above intuition, but we needed to use the sequentialising operator θ_f.

We can characterise the behaviour of $(\theta_f\ k\ \overrightarrow{v}\ h)$ as follows:

> $(\mathsf{evaluate}_f\ v_i)$ converges for every prophecy v_i in the vector $\overrightarrow{v} : (B^f)^k$ if and only if $(\theta_f\ k\ \overrightarrow{v}\ h)$ converges; moreover, in case they converge, if $z_i\colon B$ is the value of $(\mathsf{evaluate}_f\ v_i)$ for $0 \leqslant i \leqslant k$, then $(h\ \langle z_1, \ldots, z_k \rangle)$ is the value of $(\theta_f\ k\ \overrightarrow{v}\ h)$.

The characterisation of θ_f is described in the following lemma, where the inductive relation $(\overrightarrow{v} \Downarrow \overrightarrow{u})$ expresses that $(\mathsf{evaluate}_f\ v_i) \downarrow u_i$ for $0 \leqslant i \leqslant k$, where v_i is the ith element of $\overrightarrow{v} \colon (B^f)^k$ and u_i is the ith element of $\overrightarrow{u} \colon B^k$.

Lemma 1

$$\begin{array}{l}\forall k \colon \mathbb{N}, \forall \overrightarrow{v} \colon (B^f)^k, \forall h \colon B^k \to B, \forall b \colon B, \\ \qquad (\theta_f\ k\ \overrightarrow{v}\ h) \downarrow b \longleftrightarrow \exists \overrightarrow{u} \colon B^k, (\overrightarrow{v} \Downarrow \overrightarrow{u}) \wedge (h\ \overrightarrow{u} = b).\end{array}$$

In order to prove Lemma 1 we define a *weight* on prophecies and vectors of prophecies. Intuitively, the weight of a finite prophecy indicates the size of the prophecy. The weight of a prophecy, when defined, must be a positive natural number. Moreover, the weight of a tree-structured prophecy y will only be defined if the weights of all its children are defined. In addition, the weight of y has to be strictly greater than the sum of the weights of its children.

Since prophecies need not be well-founded trees, it is not possible to define their weights by a total function. Instead, the weight of a prophecy is defined as an inductive relation $\mathsf{Wght} \colon B^f \to \mathbb{N} \to \mathsf{Prop}$ with a constructor for each constructor in the set of prophecies. For each constant constructor $\mathsf{b} \colon B^f$, there is a weight constructor of the form

$$\frac{}{\mathsf{wght_b} \colon \mathsf{Wght}\ \mathsf{b}\ 1},$$

and for each non-constant constructor $\mathsf{b}_e \colon \Gamma_e \to \overbrace{B^f \to \cdots \to B^f}^{n \text{ times}} \to B^f$, there is a weight constructor of the form

$$\frac{\overrightarrow{x'} \colon \Gamma_e \qquad h_1 \colon \mathsf{Wght}\ y_1\ w_1 \quad \cdots \quad h_n \colon \mathsf{Wght}\ y_n\ w_n}{\mathsf{wght}_{\mathsf{b}_e}\ \overrightarrow{x'}\ h_1 \cdots h_n \colon \mathsf{Wght}\ (\mathsf{b}_e\ \overrightarrow{x'}\ y_1\ \cdots y_n)\ (\mathsf{S}\ (w_1 + \cdots + w_n))}.$$

The weight of a vector of prophecies is the sum of the weights of its elements plus its length, and it is given by an inductive relation $\mathsf{Weight} \colon (k \colon \mathbb{N})(B^f)^k \to \mathbb{N} \to \mathsf{Prop}$ defined as follows:

$$\frac{}{\mathsf{weight_0} \colon \mathsf{Weight}\ 0\ \langle\rangle\ 0} \qquad \frac{h_y \colon \mathsf{Wght}\ y\ w_y \qquad h_v \colon \mathsf{Weight}\ k\ \overrightarrow{v}\ w_v}{\mathsf{weight_S}\ k\ h_y\ h_v \colon \mathsf{Weight}\ (\mathsf{S}\ k)\ \langle y; \overrightarrow{v}\rangle\ (\mathsf{S}\ (w_y + w_v))}.$$

The statement of Lemma 1 can now be restrained to those vectors of prophecies that have a weight. This makes it easier to prove the lemma by applying course-of-value induction on the weight of the vector. Later, in lemmas 3 and 5, we show that the restriction can be relaxed because all converging prophecies have a weight.

Lemma 2

$$\begin{array}{l}\forall\ w \colon \mathbb{N}, \forall k \colon \mathbb{N}, \forall \overrightarrow{v} \colon (B^f)^k, \mathsf{Weight}\ k\ \overrightarrow{v}\ w \to \\ \qquad \forall h \colon B^k \to B, \forall b \colon B, (\theta_f\ k\ \overrightarrow{v}\ h) \downarrow b \longleftrightarrow \exists \overrightarrow{u} \colon B^k, (\overrightarrow{v} \Downarrow \overrightarrow{u}) \wedge (h\ \overrightarrow{u} = b).\end{array}$$

Proof. By course-of-value induction on the weight w and cases on k. Recall that the value of k determines the structure of the vector v. When v is not empty, we also perform cases on its first element.

If $k = 0$, then $\overrightarrow{v} = \langle\ \rangle$. By definition of θ_f, we have $(\theta_f\ 0\ \langle\ \rangle\ h) = \ulcorner h\ \langle\ \rangle \urcorner$ and, therefore, it must be that $b = (h\ \langle\ \rangle)$ and $\overrightarrow{u} = \langle\ \rangle$. Hence, the statement is true.

If $k = (\mathsf{S}\ k')$ then $\overrightarrow{v} = \langle y; \overrightarrow{v'}\rangle$. We now perform case analysis on y.

Let y be a leaf. We know that the vector $\overrightarrow{v'}$ must have a weight w' and moreover, w' must be strictly smaller than w, $w' < w$. Both directions of the lemma can now be easily proved by induction hypothesis on the weight w', and by definition of the functions θ_f and $\mathsf{evaluate}_f$.

Let y have a tree structure, that is, $y = (\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n)$. Given h, by definition of θ_f we get

$$\theta_f\ (\mathsf{S}\ k')\ \langle(\mathsf{b}_e\ \overrightarrow{x'}\ y_1 \cdots y_n); \overrightarrow{v'}\rangle\ h = \triangleright(\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h')$$

with $(h'\ \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle) = h\ \langle e[z_1, \ldots, z_n]; \overrightarrow{u}\rangle$. Hence, for any given b, we have the equivalence

$$(\theta_f\ (\mathsf{S}\ k')\ \overrightarrow{v}\ h) \downarrow b \longleftrightarrow (\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h') \downarrow b. \tag{1}$$

The new vector of prophecies has a smaller weight than the original one, since we replaced the first element in the vector by all its children. That is, there is a weight w' such that $(\mathsf{Weight}\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ w')$ and $w' < w$. Therefore we can apply the induction hypothesis to w', h' and b and obtain that

$$\begin{aligned}&(\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h') \downarrow b \longleftrightarrow \\ &\exists \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle \colon B^{n+k'}, (\langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle \downdownarrows \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle) \wedge \\ &\qquad\qquad (h'\ \langle z_1, \ldots, z_n; \overrightarrow{u}\rangle = b).\end{aligned} \tag{2}$$

In addition, the weight of $\langle y_1, \ldots, y_n\rangle$ is strictly smaller than the weight of the vector $\overrightarrow{v}$, so we can apply the inductive hypothesis again with the continuation $h = \lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n]$ to obtain

$$\begin{aligned}&\forall b \colon B, (\theta_f\ n\ \langle y_1, \ldots, y_n\rangle\ (\lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n])) \downarrow b \longleftrightarrow \\ &\qquad \exists \overrightarrow{z} \colon B^n, (\langle y_1, \ldots, y_n\rangle \downdownarrows \overrightarrow{z}) \wedge ((\lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n])\ \overrightarrow{z} = b).\end{aligned} \tag{3}$$

Now we prove the main statement.

In the direction from left to right, let us assume $(\theta_f\ (\mathsf{S}\ k')\ \overrightarrow{v}\ h) \downarrow b$. By the equivalence in (1), we have that $(\theta_f\ (n + k')\ \langle y_1, \ldots, y_n; \overrightarrow{v'}\rangle\ h') \downarrow b$. Now, by the instantiated induction hypothesis in (2), we know that there exists a vector $\langle z_1, \ldots, z_n; \overrightarrow{u}\rangle$ with the stated properties. In particular, $(\mathsf{evaluate}_f\ y_i) \downarrow z_i$ for $0 \leqslant i \leqslant n$, and hence $\langle y_1, \ldots, y_n\rangle \downdownarrows \langle z_1, \ldots, z_n\rangle$.

Let $z = e[z_1 \cdots z_n]$; we claim that $\langle z; \overrightarrow{u}\rangle$ is the vector satisfying the conclusion of the the lemma. We need to prove that

$$(\langle y; \overrightarrow{v'}\rangle \downdownarrows \langle z; \overrightarrow{u}\rangle) \wedge (h\ \langle z; \overrightarrow{u}\rangle = b),$$

which amounts to proving that $(\mathsf{evaluate}_f\ y) \downarrow z$, since the rest follows from the equivalence in (2). By definition of $\mathsf{evaluate}_f$, we need to prove that

$$(\theta_f\ 1\ \langle(\mathsf{b}_e\ \overrightarrow{x}\ y_1 \cdots y_n)\rangle\ (\lambda\langle z'\rangle.z')) \downarrow e[z_1 \cdots z_n].$$

By unfolding the definition of θ_f, we obtain that this statement is equivalent to the following:

$$(\theta_f\ n\ \langle y_1, \ldots, y_n\rangle\ (\lambda\langle z_1, \ldots, z_n\rangle.e[z_1, \ldots, z_n])) \downarrow e[z_1 \cdots z_n]. \qquad (4)$$

We now instantiate (3) with $b = e[z_1 \cdots z_n]$, and we apply the right-to-left direction of the resulting equivalence with the vector $\langle z_1, \ldots, z_n\rangle$ and the corresponding proofs of the needed hypotheses. We obtain then a proof of the claim in (4).

In the direction from right to left, assume that

$$\exists\langle z; \overrightarrow{u}\rangle \colon B^k, (\langle y; \overrightarrow{v'}\rangle \Downarrow \langle z; \overrightarrow{u}\rangle) \wedge (h\ \langle z; \overrightarrow{u}\rangle = b).$$

Then $(\mathsf{evaluate}_f\ y) \downarrow z$ and, since $y = (b_e\ \overrightarrow{x}\ y_1 \cdots y_n)$, by the definitions of $\mathsf{evaluate}_f$ and θ_f, we can apply the left-to-right direction of (3) with $b = z$. Thus, there exists a vector $\langle z_1, \ldots, z_n\rangle$ such that $(\mathsf{evaluate}_f\ y_i) \downarrow z_i$ for $0 \leqslant i \leqslant n$, and $e[z_1 \cdots z_n] = z$.

We can now use the right-to-left direction of (2) with the vector $\langle z_1, \ldots, z_n; \overrightarrow{u}\rangle$ and the corresponding proofs of the needed hypotheses.

Finally, the equivalence in (1) allows us to conclude that $(\theta_f\ (\mathsf{S}\ k')\ \langle y; \overrightarrow{v}\rangle\ h) \downarrow b$. □

In the next three lemmas, we show that all converging prophecies have a weight. This allows us to eliminate the weight constraint in Lemma 2, which in turn, easily allows us to obtain a proof of Lemma 1.

Lemma 3

$$\begin{aligned}\forall k\colon \mathbb{N}, \forall \overrightarrow{v}\colon (B^f)^k, \forall h\colon B^k \to B, \forall b\colon B,\\ (\theta_f\ k\ \overrightarrow{v}\ h \downarrow b) \to \exists\ w, \mathsf{Weight}\ k\ \overrightarrow{v}\ w.\end{aligned}$$

Proof. By induction on the structure of the proof of $(\theta_f\ k\ \overrightarrow{v}\ h) \downarrow b$, and by cases on the vector and, when the vector is not empty, on its first element. □

Lemma 4

$$\forall y\colon B^f, \forall b\colon B, (\mathsf{evaluate}_f\ y) \downarrow b \to \exists w, \mathsf{Wght}\ y\ w.$$

Proof. By definition of the operator $\mathsf{evaluate}_f$ and Lemma 3. □

Lemma 5

$$\forall k\colon \mathbb{N}, \forall \overrightarrow{v}\colon (B^f)^k, \forall \overrightarrow{u}\colon (B)^k, (\overrightarrow{v} \Downarrow \overrightarrow{u}) \to \exists\ w, \mathsf{Weight}\ k\ \overrightarrow{v}\ w.$$

Proof. By induction on k, using Lemma 4 on each of the elements in the vector $\overrightarrow{v}$. □

Finally, we are in the position of proving the validity of the equations defining the function f. Proving the validity of non-recursive equations is immediate: we only need to reduce the functions f and then $\mathsf{evaluate}_f$ to obtain the desired result. The validity of the recursive equations can be proved by using Lemma 1.

Theorem 1 (Validity of Recursive Equations)

$$\forall \overrightarrow{x} : \Gamma, \forall r_1, \ldots, r_n : B, (f\ p_1) \downarrow r_1 \rightarrow \cdots \rightarrow (f\ p_n) \downarrow r_n \rightarrow (f\ p) \downarrow e[r_1, \ldots, r_n].$$

Proof. Assume that $(f\ p_i) \downarrow r_i$, for $1 \leqslant i \leqslant n$. By definition of f this means that $\mathsf{evaluate}_f\ (f^\star\ p_i) \downarrow r_i$. Unfolding definitions, we have that:

$$\begin{aligned} f\ p &= \mathsf{evaluate}_f\ (f^\star\ p) \\ &= \theta_f\ 1\ \langle (f^\star\ p) \rangle\ (\lambda \langle z \rangle . z) \\ &= \theta_f\ 1\ \langle \mathsf{b}_e\ \overrightarrow{x'}\ (f^\star\ p_1) \cdots (f^\star\ p_n) \rangle\ (\lambda \langle z \rangle . z) \\ &= \triangleright(\theta_f\ n\ \langle (f^\star\ p_1), \cdots, (f^\star\ p_n) \rangle\ \lambda \langle z_1, \ldots, z_n \rangle . e[z_1, \ldots, z_n]). \end{aligned}$$

The conclusion easily follows now by applying the second constructor of the inductive relation $(_ \downarrow _)$ and the right-to-left direction of Lemma 1. □

In the specific case of the function F, Theorem 1 gives the validity of the equations presented in page 77.

When reasoning about recursive functions, an inversion principle given by the converse of Theorem 1 may be useful.

Theorem 2 (Inversion Principle for Recursive Equations)

$$\begin{aligned} &\forall \overrightarrow{x} : \Gamma, \forall b \colon B, (f\ p) \downarrow b \rightarrow \\ &\exists r_1, \ldots, r_n \colon B, (f\ p_1) \downarrow r_1 \wedge \cdots \wedge (f\ p_n) \downarrow r_n \wedge b = e[r_1, \ldots, r_n]. \end{aligned}$$

Proof. By the left-to-right direction of Lemma 1. □

6 Conclusions

This article describes a new method to represent (partial) recursive functions in type theory. It combines ideas from our previous work on the subject, namely, the one characterising the inputs on which a function terminates by an inductive (domain) predicate [6,4,7], and the one implementing recursive functions by co-recursion on co-inductive types of partial elements [8].

Given the recursive equations for a computable function $f \colon A \rightarrow B$, we define a co-inductive set B^f of *prophecies* representing the traces of the computation of the function. This set is the dual of the predicate defining the domain of the function as described in [6]. It is easy to define a formal recursive function $f^\star \colon A \rightarrow B^f$ returning prophecies as output. The type-theoretic version of the

original function is then a function whose co-domain is the set B^ν of partial elements as studied in [8]. This function is defined by *linearising* the prophecy obtained from the formal recursive function $f^\star$. The linearisation is done by an operator θ_f with two parameters: a stack of recursive calls and a continuation function that will compute, when possible, the final result.

We prove that a function formalised in type theory with this new method satisfies all the equations (recursive or not) of its informal version.

We illustrate the method on a toy example, a generalisation F of the Fibonacci function. The development for F has been fully formalised in the proof assistant *Coq* [14,3]. In addition, we also performed a complete formalisation of the *quick-sort* algorithm following this method, and of the proofs stating the validity of the equations for the algorithm. The files of both formalisations are available on the web at the address: `http://www.cs.ru.nl/~venanzio/Coq/prophecy.html`.

At the moment, the method has been tested just on simple recursive programs, that is, not nested or mutually recursive. The formalisation of mutually recursive functions usually presents no major problems in systems like *Coq*, but on the other hand, nested functions are not trivial to formalise. Already our previous work (using the domain predicates) on nested recursive functions [5] could not directly be translated into *Coq* because the proof assistant lacks support for inductive-recursive definitions, as described by Dybjer in [10].

As can be seen from the proofs, the method is general and can be adapted to every simple recursive program. However, we have yet to formalise the mechanisation process and therefore, the user must go through the tedious but trivial process of adapting definitions and proofs. It would be desirable, in a future stage, to fully automatise the definitions of the set of prophecies, of the function θ_f, and the related proofs from the set of (recursive) equations given by the user.

As mentioned before, this technique has several advantages over the method we have previously developed via a special-purpose accessibility (domain) predicate [6,4,7]; namely, there is no need of extra logical arguments in the definition of the recursive function; the function can be applied to any element in its domain, regardless of termination properties; we obtain a type of partial recursive functions between any two given types; and composition of recursive functions can be easily defined through the usual composition of monadic function (see [8] for further reading on this point).

References

1. Balaa, A., Bertot, Y.: Fonctions récursives générales par itération en théorie des types. Journées Francophones des Langages Applicatifs - JFLA02, INRIA (January 2002)
2. Barthe, G., Courtieu, P.: Efficient reasoning about executable specifications in Coq. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 20–23. Springer, Heidelberg (2002)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Berlin Heidelberg (2004)

4. Bove, A.: General recursion in type theory. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 39–58. Springer, Heidelberg (2003)
5. Bove, A., Capretta, V.: Nested general recursion and partiality in type theory. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 121–135. Springer, Heidelberg (2001)
6. Bove, A., Capretta, V.: Modelling general recursion in type theory. Mathematical Structures in Computer Science 15(4), 671–708 (2005)
7. Bove, A., Capretta, V.: Recursive functions with higher order domains. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 116–130. Springer, Heidelberg (2005)
8. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1(2), 1–18 (2005)
9. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437–468 (2006)
10. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. Journal of Symbolic Logic, vol. 65(2) (2000)
11. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Smith, J., Dybjer, P., Nordström, B. (eds.) TYPES 1994. LNCS, vol. 996, pp. 39–59. Springer, Heidelberg (1995)
12. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1), 69–111 (2004)
13. Reynolds, J.C.: The discoveries of continuations. Lisp. and Symbolic Computation 6(3–4), 233–248 (1993)
14. The Coq Development Team. LogiCal Project. The Coq Proof Assistant. Reference Manual. Version 8. INRIA (2004) Available at the web page http://pauillac.inria.fr/coq/coq-eng.html

An Arithmetical Proof of the Strong Normalization for the λ-Calculus with Recursive Equations on Types

René David and Karim Nour

Université de Savoie, Campus Scientifique, 73376 Le Bourget du Lac, France
{david, nour}@univ-savoie.fr

Abstract. We give an arithmetical proof of the strong normalization of the λ-calculus (and also of the $\lambda\mu$-calculus) where the type system is the one of simple types with recursive equations on types.

The proof using candidates of reducibility is an easy extension of the one without equations but this proof cannot be formalized in Peano arithmetic. The strength of the system needed for such a proof was not known. Our proof shows that it is not more than Peano arithmetic.

1 Introduction

The λ-calculus is a powerful model for representing functions. In its un-typed version, every recursive function can be represented. But, in this model, a term can be applied to itself and a computation may not terminate. To avoid this problem, types are used. In the simplest case, they are built from atomic types with the arrow and the typing rules say that a function of type $U \to V$ may only be applied to an argument of type U. This discipline ensures that every typed term is strongly normalizing, i.e. a computation always terminate.

In this system (the simply typed λ-calculus), Church numerals, i.e. the terms of the form $\lambda f \lambda x(f\ (f\ \ldots\ (f\ x)))$, are codes for the integers. They are the only terms (in normal form) of type $(o \to o) \to (o \to o)$. Thus, functions on the integers can be represented but Schwichtenberg [38] has shown that very few functions are so. He showed that the extended polynomials (i.e. polynomials with positive coefficients together with a conditional operator) are the only functions that can be represented there. Other type systems were then designed to allow the representation of more functions. They are built in different ways.

The first one consists in extending the set of terms. For example, in Gödel system T, the terms use the usual constructions of the λ-calculus, the constant 0, the constructor S and an operator for recursion. The types are built from the atomic type N with the arrow. This system represents exactly the functions whose totality can be shown in Peano first order arithmetic.

The second one consists in keeping the same terms but extending the type system. This is, for example, the case of Girard system F where the types can use a second order universal quantifier. There, the type of the integers is given

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 84–101, 2007.
© Springer-Verlag Berlin Heidelberg 2007

by $\forall X\,((X \to X) \to (X \to X))$. This system represents exactly the functions whose totality can be shown in Peano second order arithmetic.

A third way consists in extending the *logic*. In the Curry-Howard correspondence, the previous systems correspond to *intuitionistic* logic. Other systems correspond to *classical* logic. There, again, new constructors for terms are introduced. This is, for example, the case of Parigot's $\lambda\mu$-calculus [35].

Since the introduction of Girard system F for intuitionistic logic and Parigot's $\lambda\mu$-calculus for classical logic, many others, more and more powerful, type systems were introduced. For example, the calculus of constructions (Coquand & Huet [7]) and, more generally, the Pure Type Systems.

It is also worth here to mention the system TTR of Parigot [33] where some types are defined as the least fixed point of an operator. This system was introduced, not to represent more functions, but to represent more *algorithms*. For example, to be able to represent the integers in such a way that the predecessor can be computed in constant time, which is not the case for the previous systems.

These systems all satisfy the subject reduction (i.e. the fact that the type is preserved by reduction), the strong normalization (i.e. every computation terminates) and, for the systems based on simple types, the decidability of type assignment.

We study here other kinds of extension of the simply typed λ-calculus, i.e. systems where *equations* on types are allowed. These types are usually called *recursive types*. For more details see, for example, [3]. They are present in many languages and are intended to be able to be *unfolded* recursively to match other types. The subject reduction and the decidability of type assignment are preserved but the strong normalization may be lost. For example, with the equation $X = X \to T$, the term $(\delta\ \delta)$ where $\delta = \lambda x\ (x\,x)$ is typable but is not strongly normalizing. With the equation $X = X \to X$, every term can be typed.

By making some natural assumptions on the recursive equations the strong normalization can be preserved. The simplest condition is to accept the equation $X = F$ (where F is a type containing the variable X) only when the variable X is positive in F. For a set $\{X_i = F_i \ / \ i \in I\}$ of mutually recursive equations, Mendler [29] has given a very simple and natural condition that ensures the strong normalization of the system. He also showed that the given condition is necessary to have the strong normalization. His proof is based on the reducibility method. The condition ensures enough monotonicity to have fixed point on the candidates. But this proof (using candidates of reducibility) cannot be formalized in Peano arithmetic and the strength of the system needed for a proof of the strong normalization of such systems was not known.

In this paper, we give an *arithmetical* proof of the strong normalization of the simply typed λ-calculus (and also of the $\lambda\mu$-calculus) with recursive equations on types satisfying Mendler's condition.

This proof is an extension of the one given by the first author for the simply typed λ-calculus. It can be found either in [8] (where it appears among many other things) or as a simple unpublished note on the web page of the first author

[9]. Apparently, proof methods similar to that used here were independently invented by several authors (Levy, van Daalen, Valentini and others). The proof for the $\lambda\mu$-calculus is an extension of the ones given in [11] or [12].

The paper is organized as follows. In section 2 we define the simply typed λ-calculus with recursive equations on types. To help the reader and show the main ideas, we first give, in section 3, the proof of strong normalization for the λ-calculus. We generalize this proof to the $\lambda\mu$-calculus in section 4. In section 5, we give two examples of applications of systems with recursive types. We conclude in section 6 with some open questions.

2 The Typed λ-Calculus

Definition 1. *Let $\mathcal{V}$ be an infinite set of variables.*

1. *The set $\mathcal{M}$ of λ-terms is defined by the following grammar*

$$\mathcal{M} ::= \mathcal{V} \mid \lambda\mathcal{V}\ \mathcal{M} \mid (\mathcal{M}\ \mathcal{M})$$

2. *The relation $\triangleright$ on $\mathcal{M}$ is defined as the least relation (compatible with the context) containing the rule $(\lambda x\ M\ N) \triangleright M[x := N]$. As usual, $\triangleright^*$ (resp. $\triangleright^+$) denotes the reflexive and transitive (resp. transitive) closure of $\triangleright$.*

Definition 2. *Let $\mathcal{A}$ be a set of atomic constants and $\mathcal{X} = \{X_i \ / \ i \in I\}$ be a set of type variables.*

1. *The set $\mathcal{T}$ of types is defined by the following grammar*

$$\mathcal{T} ::= \mathcal{A} \mid \mathcal{X} \mid \mathcal{T} \to \mathcal{T}$$

2. *When $E = \{F_i \ / \ i \in I\}$ is a set of types, the congruence $\approx$ generated by E is the least congruence on $\mathcal{T}$ such that $X_i \approx F_i$ for each $i \in I$.*

Definition 3. *Let $\approx$ be a congruence on $\mathcal{T}$. The typing rules of the typed system are given below where Γ is a context, i.e. a set of declarations of the form $x : U$ where $x \in \mathcal{V}$ and $U \in \mathcal{T}$.*

$$\frac{}{\Gamma, x : U \vdash x : U}\ ax \qquad \frac{\Gamma \vdash M : U \quad U \approx V}{\Gamma \vdash M : V}\ \approx$$

$$\frac{\Gamma, x : U \vdash M : V}{\Gamma \vdash \lambda x\ M : U \to V}\ \to_i \qquad \frac{\Gamma \vdash M_1 : U \to V \quad \Gamma \vdash M_2 : U}{\Gamma \vdash (M_1\ M_2) : V}\ \to_e$$

Lemma 1. *Let $\approx$ be a congruence generated by a set of types.*

1. *If $U \approx V_1 \to V_2$, then $U \in \mathcal{X}$ or $U = U_1 \to U_2$.*
2. *If $U_1 \to V_1 \approx U_2 \to V_2$, then $U_1 \approx U_2$ and $V_1 \approx V_2$.*
3. *If $\Gamma \vdash x : T$, then $x : U$ occurs in Γ for some $U \approx T$.*
4. *If $\Gamma \vdash \lambda x\ M : T$, then $\Gamma, x : U \vdash M : V$ for some U, V such that $U \to V \approx T$.*
5. *If $\Gamma \vdash (M\ N) : T$, then $\Gamma \vdash M : U \to V$, $\Gamma \vdash N : U$ for some $V \approx T$ and U.*
6. *If $\Gamma, x : U \vdash M : T$ and $U \approx V$, then $\Gamma, x : V \vdash M : T$.*
7. *If $\Gamma, x : U \vdash M : T$ and $\Gamma \vdash N : U$, then $\Gamma \vdash M[x := N] : T$.*

Proof. Easy. □

Theorem 1. *If $\Gamma \vdash M : T$ and $M \rhd^* M'$, then $\Gamma \vdash M' : T$.*

Proof. It is enough to show that if $\Gamma \vdash (\lambda x\ M\ N) : T$, then $\Gamma \vdash M[x := N] : T$. Assume $\Gamma \vdash (\lambda x\ M\ N) : T$. By lemma 1, $\Gamma \vdash \lambda x\ M : U \to V$, $\Gamma \vdash N : U$ and $V \approx T$. Thus, $\Gamma, x : U' \vdash M : V'$ and $U' \to V' \approx U \to V$. By lemma 1, we have $U' \approx U$ and $V' \approx V$. Thus, $\Gamma, x : U \vdash M : V$. Since $\Gamma \vdash N : U$ and $V \approx T$, the result follows immediately. □

Definition 4. *Let $X \in \mathcal{X}$. We define the subsets $\mathcal{T}^+(X)$ and $\mathcal{T}^-(X)$ of $\mathcal{T}$ as follows.*

- $X \in \mathcal{T}^+(X)$
- *If $U \in (\mathcal{X} - \{X\}) \cup \mathcal{A}$, then $U \in \mathcal{T}^+(X) \cap \mathcal{T}^-(X)$.*
- *If $U \in \mathcal{T}^-(X)$ and $V \in \mathcal{T}^+(X)$, then $U \to V \in \mathcal{T}^+(X)$ and $V \to U \in \mathcal{T}^-(X)$.*

Definition 5. *We say that a congruence $\approx$ is good if the following property holds: for each $X \in \mathcal{X}$, if $X \approx T$, then $T \in \mathcal{T}^+(X)$.*

Examples

In each of the following cases, the congruence generated by the given equations is good.

1. $X_1 \approx (X_1 \to X_2 \to Y) \to Y$ and $X_2 \approx (X_2 \to X_1 \to Y) \to Y$.
2. $X_1 \approx X_2 \to X_1$ and $X_2 \approx X_1 \to X_2$.
3. The same equations as in case 2 and $X_3 \approx F(X_1, X_2) \to X_3$ where F is any type using only the variables X_1, X_2.
4. The same equations as in case 3 and $X_4 \approx X_5 \to G(X_1, X_2, X_3) \to X_4$, $X_5 \approx X_4 \to H(X_1, X_2, X_3) \to X_5$ where G, H are any types using only the variables X_1, X_2, X_3.

In the rest of the paper, we fix a finite set $E = \{F_i \ / \ i \in I\}$ of types and we denote by $\approx$ the congruence generated by E. We assume that $\approx$ is good.

Notations and Remarks

- We have assumed that the set of equations that we consider is finite. This is to ensure that the order on I given by definition 6 below is well founded. It should be clear that this is not a real constraint. Since to type a term, only a finite number of equations is used, we may consider that the other variables are constant and thus the general result follows immediately from the finite case.
- If M is a term, $cxty(M)$ will denote the structural complexity of M.
- We denote by SN the set of strongly normalizing terms. If $M \in SN$, we denote by $\eta(M)$ the length of the longest reduction of M and by $\eta c(M)$ the pair $\langle \eta(M), cxty(M) \rangle$.
- We denote by $M \preceq N$ the fact that M is a sub-term of a reduct of N.
- As usual, some parentheses are omitted and, for example, we write $(M\ P\ Q)$ instead of $((M\ P)\ Q)$. More generally, if $\overrightarrow{O}$ is a finite sequence $O_1, \ldots, O_n$ of terms, we denote by $(M\ \overrightarrow{O})$ the term $((\ldots(M\ O_1)\ldots\ O_{n-1})\ O_n)$ and by $\overrightarrow{O} \in SN$ the fact that $O_1, \ldots, O_n \in SN$.

- If σ is the substitution $[x_1 := N_1, \ldots, x_n := N_n]$, we denote by $dom(\sigma)$ the set $\{x_1, \ldots, x_n\}$, by $Im(\sigma)$ the set $\{N_1, \ldots, N_n\}$ and by $\sigma \in SN$ the fact that $Im(\sigma) \subset SN$.
- If σ is a substitution, $z \notin dom(\sigma)$ and M is a term, we denote by $[\sigma + z := M]$ the substitution σ' defined by $\sigma'(x) = \sigma(x)$ for $x \in dom(\sigma)$ and $\sigma'(z) = M$.
- In a proof by induction, *IH* will denote the induction hypothesis. When the induction is done on a tuple of integers, the order always is the lexicographic order.

3 Proof of the Strong Normalization

3.1 The Idea of the Proof

We give the idea for one equation $X \approx F$. The extension for the general case is given at the beginning of section 3.4.

It is enough to show that, if M, N are in SN, then $M[x := N] \in SN$. Assuming it is not the case, the interesting case is $M = (x\ P)$ with $(N\ P_1) \notin SN$ where $P_1 = P[x := N] \in SN$. This implies that $N \triangleright^* \lambda y N_1$ and $N_1[y = P_1] \notin SN$. If we know that the type of N is an arrow type, we get a similar situation to the one we started with, but where the type of the substituted variable has decreased. Repeating the same argument, we get the desired result, at least for N whose type does not contain X. If it is not the case, since, by repeating the same argument, we cannot come to a constant type (because such a term cannot be applied to something), we come to X. Thus, it remains to show that, if M, N are in SN and the type of x is X, then $M[x := N] \in SN$.

To prove this, we prove something a bit more general. We prove that, if $M, \sigma \in SN$ where σ is a substitution such that the types of its image are in $\mathcal{T}^+(X)$, then $M[\sigma] \in SN$. The proof is done, by induction on $\eta c(M)$ as follows. As before, the interesting case is $M = (x\ P), \sigma(x) = N \triangleright^* \lambda y N_1$, $P_1 = P[\sigma] \in SN$ and $N_1[y = P_1] \notin SN$. Thus, there is a sub-term of a reduct of N_1 of the form $(y\ N_2)$ such that $(P_1\ N_2[y := P_1]) \notin SN$ but $N_2[y := P_1] \in SN$. Thus P_1 must reduce to a λ.

This λ cannot come from some $x' \in dom(\sigma)$, i.e. $P \triangleright^* (x'\ \overrightarrow{Q})$. Otherwise, the type of P would be both positive (since $P \triangleright^* (x'\ \overrightarrow{Q})$ and the type of x' is positive) and negative (since, in M, P is an argument of x whose type also is positive). Thus the type of P_1 (the same as the one of P) does not contain X. But since N_1, P_1 are in SN, we already know that $N_1[y = P_1]$ must be in SN. A contradiction. Thus, $P \triangleright^* \lambda x_1 M_1$ and we get a contradiction from the induction hypothesis since we have $M_1[\sigma'] \notin SN$ for M_1 strictly less than M. The case when y has more than one argument is intuitively treated by "repeat the same argument" or, more formally, by lemma 8 below.

As a final remark, note that many lemmas are stated in a negative style and thus may seem to hold only classically. This has been done in this way because we believe that this presentation is closer to the intuition. However, it is not difficult to check that the whole proof can be presented and done in a constructive way.

3.2 Some Useful Lemmas on the Un-Typed Calculus

Lemma 2. *Assume* $M, N, \overrightarrow{O} \in SN$ *and* $(M\ N\ \overrightarrow{O}) \notin SN$*. Then, for some term* M'*,* $M \rhd^* \lambda x\ M'$ *and* $(M'[x := N]\ \overrightarrow{O}) \notin SN$*.*

Proof. Since $M, N, \overrightarrow{O} \in SN$, an infinite reduction of $P = (M\ N\ \overrightarrow{O})$ looks like $P \rhd^* (\lambda x\ M'\ N'\ \overrightarrow{O'}) \rhd (M'[x := N']\ \overrightarrow{O'}) \rhd \ldots$ and the result immediately follows from the fact that $(M'[x := N]\ \overrightarrow{O}) \rhd^* (M'[x := N']\ \overrightarrow{O'})$. □

Lemma 3. *Let* M *be a term and* σ *be a substitution. Assume* $M, \sigma \in SN$ *and* $M[\sigma] \notin SN$*. Then* $(\sigma(x)\ \overrightarrow{P[\sigma]}) \notin SN$ *for some* $(x\ \overrightarrow{P}) \preceq M$ *such that* $\overrightarrow{P[\sigma]} \in SN$*.*

Proof. A sub-term M' of a reduct of M such that $\eta c(M')$ is minimum and $M'[\sigma] \notin SN$ has the desired form. □

Lemma 4. *Let* M *be a term and* σ *be a substitution such that* $M[\sigma] \rhd^* \lambda z M_1$*. Then*
- either $M \rhd^* \lambda z M_2$ *and* $M_2[\sigma] \rhd^* M_1$
- or $M \rhd^* (x\ \overrightarrow{N})$ *for some* $x \in dom(\sigma)$ *and* $(\sigma(x)\ \overrightarrow{N[\sigma]}) \rhd^* \lambda z M_1$*.*

Proof. This is a classical (though not completely trivial) result in λ-calculus. Note that, in case $M \in SN$ (and we will only use the lemma in this case), it becomes easier. The proof can be done by induction on $\eta c(M)$ by considering the possibility for M: either $\lambda y M_1$ or $(\lambda y M_1\ P\ \overrightarrow{Q})$ or $(x\ \overrightarrow{N})$ (for x in $dom(\sigma)$ or not). □

3.3 Some Useful Lemmas on the Congruence

Definition 6. *We define on* I *the following relations*

– $i \leq j$ *iff* $X_i \in var(T)$ *for some* T *such that* $X_j \approx T$*.*
– $i \sim j$ *iff* $i \leq j$ *and* $j \leq i$*.*
– $i < j$ *iff* $i \leq j$ *and* $j \not\leq i$

It is clear that $\sim$ is an equivalence on I.

Definition 7

1. *Let* $\mathcal{X}_i = \{X_j \ /\ j \leq i\}$ *and* $\mathcal{X}'_i = \{X_j \ /\ j < i\}$*.*
2. *For* $\mathcal{Y} \subseteq \mathcal{X}$*, let* $\mathcal{T}(\mathcal{Y}) = \{T \in \mathcal{T} \ /\ var(T) \subseteq \mathcal{Y}\}$ *where* $var(T)$ *is the set of type variables occurring in* T*.*
3. *For* $i \in I$*, we will abbreviate by* $\mathcal{T}_i$ *the set* $\mathcal{T}(\mathcal{X}_i)$ *and by* $\mathcal{T}'_i$ *the set* $\mathcal{T}(\mathcal{X}'_i)$*.*
4. *If* $\varepsilon \in \{+, -\}$*,* $\overline{\varepsilon}$ *will denote the opposite of* ε*. The opposite of + is - and conversely.*

Lemma 5. *Let* $i \in I$*. The class of* i *can be partitioned into two disjoint sets* i^+ *and* i^- *satisfying the following properties.*

1. *If* $\varepsilon \in \{+,-\}$, $j \in i^\varepsilon$ *and* $X_j \approx T$, *then for each* $k \in i^\varepsilon$, $T \in \mathcal{T}^\varepsilon(X_k)$ *and for each* $k \in i^{\overline{\varepsilon}}$, $T \in \mathcal{T}^{\overline{\varepsilon}}(X_k)$.
2. *Let* $j \sim i$. *Then, if* $j \in i^+$, $j^+ = i^+$ *and* $j^- = i^-$ *and if* $j \in i^-$, $j^+ = i^-$ *and* $j^- = i^+$.

Proof. This follows immediately from the following observation. Let $i \sim j$ and $X_i \approx T \approx U$. Choose an occurrence of X_j in T and in U. Then, these occurrences have the same polarity. This is because, otherwise, since $i \leq j$, there is a V such that $X_j \approx V$ and X_i occurs in V. But then, replacing the mentioned occurrences of X_j by V in T and U will contradict the fact that $\approx$ is good. □

Definition 8. *Let* $i \in I$ *and* $\varepsilon \in \{+,-\}$. *We denote* $\mathcal{T}_i^\varepsilon = \{T \in \mathcal{T}_i$ / *for each* $j \in i^\varepsilon$, $T \in \mathcal{T}^\varepsilon(X_j)$ *and for each* $j \in i^{\overline{\varepsilon}}$, $T \in \mathcal{T}^{\overline{\varepsilon}}(X_j)\}$.

Lemma 6. *Let* $i \in I$ *and* $\varepsilon \in \{+,-\}$.

1. $\mathcal{T}_i^\varepsilon \cap \mathcal{T}_i^{\overline{\varepsilon}} \subseteq \mathcal{T}_i'$.
2. *If* $U \in \mathcal{T}_i^\varepsilon$ *and* $U \approx V$, *then* $V \in \mathcal{T}_i^\varepsilon$.
3. *If* $U \in \mathcal{T}_i^\varepsilon$ *and* $U \approx U_1 \to U_2$, *then* $U_1 \in \mathcal{T}_i^{\overline{\varepsilon}}$ *and* $U_2 \in \mathcal{T}_i^\varepsilon$.

Proof. Immediate. □

Notations, Remarks and Examples

- If the equations are those of the case 4 of the examples given above, we have $1 \sim 2 < 3 < 4 \sim 5$ and, for example, $1^+ = \{1\}$ and $1^- = \{2\}$, $3^+ = \{3\}$, $3^- = \emptyset$, $4^+ = \{4\}$ and $4^- = \{5\}$.
- If T is a type, we denote by $lg(T)$ the size of T. Note that the size of a type is, of course, not preserved by the congruence. The size of a type will only be used in lemma 7 and the only property that we will use is that $lg(U_1)$ and $lg(U_2)$ are less than $lg(U_1 \to U_2)$.
- By the typing rules, the type of a term can be freely replaced by an equivalent one. However, for $i \in I$ and $\varepsilon \in \{+,-\}$, the fact that $U \in \mathcal{T}_i^\varepsilon$ does not change when U is replaced by V for some $V \approx U$. This will be used extensively in the proofs of the next sections.

3.4 Proof of the Strong Normalization

To give the idea of the proof, we first need a definition.

Definition 9. *Let* $\mathcal{E}$ *be a set of types. Denote by* $H[\mathcal{E}]$ *the following property:*
Let $M, N \in SN$. *Assume* $\Gamma, x : U \vdash M : V$ *and* $\Gamma \vdash N : U$ *for some* Γ, U, V *such that* $U \in \mathcal{E}$. *Then* $M[x := N] \in SN$.

To get the result, it is enough to show $H[\mathcal{T}]$. The proof that any typed term is in SN is then done by induction on $cxty(M)$. The only non trivial case is $M = (M_1\, M_2)$. But $M = (x\, M_2)[x := M_1]$ and the result follows from $H[\mathcal{T}]$ and the IH.

We first show the following (see lemma 7). Let $\mathcal{Y} \subseteq \mathcal{X}$. To prove $H[\mathcal{T}(\mathcal{Y})]$, it is enough to prove $H[\{X\}]$ for each $X \in \mathcal{Y}$.

It is thus enough to prove of $H[\{X_i\}]$ for each $i \in I$. This is done by induction on i. Assume $H[\{X_j\}]$ for each $j < i$. Thus, by the previous property, we know $H[\mathcal{T}_i']$. We show $H[\{X_i\}]$ essentially as we said in section 3.1. The only difference is that, what was called there " X is both positive and negative in T" here means T is both in $\mathcal{T}_i^+$ and $\mathcal{T}_i^-$. There we deduced that X does not occur in T. Here we deduce $T \in \mathcal{T}_i'$ and we are done since we know the result for this set.

Lemma 7. *Let $\mathcal{Y} \subseteq \mathcal{X}$ be such that $H[\{X\}]$ holds for each $X \in \mathcal{Y}$. Then $H[\mathcal{T}(\mathcal{Y})]$ holds.*

Proof. Let M, N be terms in SN. Assume $\Gamma, x : U \vdash M : V$ and $\Gamma \vdash N : U$ and $U \in \mathcal{T}(\mathcal{Y})$. We have to show $M[x := N] \in SN$.

This is done by induction on $lg(U)$. Assume $M[x := N] \notin SN$. By lemma 3, let $(x\,P\,\overrightarrow{Q}) \preceq M$ be such that $P_1, \overrightarrow{Q_1} \in SN$ and $(N\,P_1\,\overrightarrow{Q_1}) \notin SN$ where $P_1 = P[x := N]$ and $\overrightarrow{Q_1} = \overrightarrow{Q[x := N]}$. By lemma 2, $N \rhd^* \lambda x_1 N_1$ and $(N_1[x_1 := P_1]\ \overrightarrow{Q_1}) \notin SN$.

If U is a variable (which is in $\mathcal{Y}$ since $U \in \mathcal{T}(\mathcal{Y})$), we get a contradiction since we have assumed that $H[\{X\}]$ holds for each $X \in \mathcal{Y}$.

The type U cannot be a constant since, otherwise x could not be applied to some arguments.

Thus $U = U_1 \to U_2$. In the typing of $(N\,P_1\,\overrightarrow{Q_1})$, the congruence may have been used and thus, by lemma 1, there are $W_1 \approx U_1$, $W_2 \approx U_2$, $U \approx W_1 \to W_2$ and $\Gamma, x_1 : W_1 \vdash N_1 : W_2$ and $\Gamma \vdash P_1 : W_1$. But then, we also have $\Gamma, x_1 : U_1 \vdash N_1 : U_2$ and $\Gamma \vdash P_1 : U_1$. Now, by the *IH*, we have $N_1[x_1 := P_1] \in SN$ since $lg(U_1) < lg(U)$. Since $\Gamma, z : U_2 \vdash (z\,\overrightarrow{Q_1}) : V'$ for some V' and $\Gamma \vdash N_1[x_1 := P_1] : U_2$, by the *IH* since $lg(U_2) < lg(U)$, we have $(N_1[x_1 := P_1]\ \overrightarrow{Q_1}) = (z\,\overrightarrow{Q_1})[z = N_1[x_1 := P_1]] \in SN$. Contradiction. □

For now on, we fix some i and we assume $H[\{X_j\}]$ for each $j < i$. Thus, by lemma 7, we know that $H[\mathcal{T}_i']$ holds. It remains to prove $H[\{X_i\}]$ i.e. proposition 1.

Definition 10. *Let M be a term, σ be a substitution, Γ be a context and U be a type. Say that (σ, Γ, M, U) is adequate if the following holds.*

- *$\Gamma \vdash M[\sigma] : U$ and $M, \sigma \in SN$.*
- *For each $x \in dom(\sigma)$, $\Gamma \vdash \sigma(x) : V_x$ and $V_x \in \mathcal{T}_i^+$.*

Lemma 8. *Let n, m be integers, $\overrightarrow{S}$ be a sequence of terms and (δ, Δ, P, B) be adequate. Assume that*

1. *$B \in \mathcal{T}_i^- - \mathcal{T}_i'$ and $\Delta \vdash (P[\delta]\ \overrightarrow{S}) : W$ for some W.*
2. *$\overrightarrow{S} \in SN$, $P \in SN$ and $\eta c(P) < \langle n, m\rangle$.*
3. *$M[\sigma] \in SN$ for every adequate (σ, Γ, M, U) such that $\eta c(M) < \langle n, m\rangle$.*

Then $(P[\delta]\ \overrightarrow{S}) \in SN$.

Proof. By induction on the length of $\overrightarrow{S}$. If $\overrightarrow{S}$ is empty, the result follows from (3) since $\eta c(P) < \langle n, m\rangle$. Otherwise, let $\overrightarrow{S} = S_1\overrightarrow{S_2}$ and assume that $P[\delta] \triangleright^* \lambda z\ R$. By lemma 4, there are two cases to consider:

– $P \triangleright^* \lambda z\ R'$. We have to show that $Q = (R'[\delta + z := S_1]\ \overrightarrow{S_2}) \in SN$. Since $B \in \mathcal{T}_i^-$, by lemmas 1 and 6, there are types B_1, B_2 such that $B \approx B_1 \to B_2$ and $\Delta, z : B_1 \vdash R' : B_2$ and $\Delta \vdash S_1 : B_1$ and $B_1 \in \mathcal{T}_i^+$ and $B_2 \in \mathcal{T}_i^-$. Since $\eta c(R') < \langle n, m\rangle$ and $([\delta + z = S_1], \Delta \cup \{z : B_1\}, R', B_2)$ is adequate, it follows from (3) that $R'[\delta + z := S_1]] \in SN$.
  - Assume first $B_2 \in \mathcal{T}_i'$. Since $(z'\ \overrightarrow{S_2}) \in SN$ and $Q = (z'\ \overrightarrow{S_2})[z' := R'[\delta + z := S_1]]$, the result follows from $H[\mathcal{T}_i']$.
  - Otherwise, the result follows from the *IH* since $([\delta + z = S_1], \Delta \cup \{z : B_1\}, R', B_2)$ is adequate and the length of $\overrightarrow{S_2}$ is less than the one of $\overrightarrow{S}$.

– If $P \triangleright^* (y\ \overrightarrow{T})$ for some $y \in dom(\delta)$. Then $\Delta \vdash (\delta(y)\ \overrightarrow{T[\delta]}) : B$. By the definition of adequacy, the type of y is in $\mathcal{T}_i^+$ and $B \in \mathcal{T}_i^- \cap \mathcal{T}_i^+ \subseteq \mathcal{T}_i'$. Contradiction. □

Lemma 9. *Assume (σ, Γ, M, A) is adequate. Then $M[\sigma] \in SN$.*

Proof. By induction on $\eta c(M)$. The only non trivial case is $M = (x\ Q\ \overrightarrow{O})$ for some $x \in dom(\sigma)$. Let $N = \sigma(x)$.

By the *IH*, $Q[\sigma], \overrightarrow{O[\sigma]} \in SN$. By lemma 1, we have $V_x \approx W_1 \to W_2$, $\Gamma \vdash Q[\sigma] : W_1$ and $\Gamma \vdash (N\ Q[\sigma]) : W_2$. Moreover, by lemma 6, $W_1 \in \mathcal{T}_i^-$ and $W_2 \in \mathcal{T}_i^+$. Since $M[\sigma] = (z\ \overrightarrow{O})[\sigma + z := (N\,Q[\sigma])]$, $\eta((z\ \overrightarrow{O})) \leq \eta(M)$, $cxty((z\ \overrightarrow{O})) < cxty(M)$ and $W_2 \in \mathcal{T}_i^+$, it is enough, by the *IH*, to show that $(N\,Q[\sigma]) \in SN$. Assume that $N \triangleright^* \lambda y\ N'$. We have to show that $N'[y := Q[\sigma]] \in SN$.

- Assume first $W_1 \in \mathcal{T}_i'$. The result follows from $H[\mathcal{T}_i']$.

- Otherwise, assume $N'[y := Q[\sigma]] \notin SN$. Since $N', Q[\sigma] \in SN$, by lemma 3, $(y\ \overrightarrow{L}) \preceq N'$ for some $\overrightarrow{L}$ such that $\overrightarrow{L[y := Q[\sigma]]} \in SN$ and $(Q[\sigma]\ \overrightarrow{L[y := Q[\sigma]]})$ $\notin SN$. But this contradicts lemma 8. Note that, by the *IH*, condition (3) of this lemma is satisfied. □

Proposition 1. *Assume $\Gamma, x : X_i \vdash M : U$ and $\Gamma \vdash N : X_i$ and $M, N \in SN$. Then $M[x := N] \in SN$.*

Proof. This follows from lemma 9 since $([x := N], \Gamma, M, U)$ is adequate. □

4 The Typed $\lambda\mu$-Calculus

Definition 11

1. *Let $\mathcal{W}$ be an infinite set of variables such that $\mathcal{V} \cap \mathcal{W} = \emptyset$. An element of $\mathcal{V}$ (resp. $\mathcal{W}$) is said to be a λ-variable (resp. a μ-variable). We extend the set of terms by the following rules*

$$\mathcal{M} ::= \ldots \mid \mu\mathcal{W}\,\mathcal{M} \mid (\mathcal{W}\ \mathcal{M})$$

2. *We add to the set* $\mathcal{A}$ *the constant symbol* $\perp$ *and we denote by* $\neg U$ *the type* $U \to \perp$.
3. *We extend the typing rules by*

$$\frac{\Gamma, \alpha : \neg U \vdash M : \perp}{\Gamma \vdash \mu\alpha M : U} \perp_e \qquad \frac{\Gamma, \alpha : \neg U \vdash M : U}{\Gamma, \alpha : \neg U \vdash (\alpha\ M) : \perp} \perp_i$$

where Γ *is now a set of declarations of the form* $x : U$ *and* $\alpha : \neg U$ *where* x *is a* λ*-variable and* α *is a* μ*-variable.*
4. *We add to* $\triangleright$ *the following reduction rule* $(\mu\alpha M\ N) \triangleright \mu\alpha M[\alpha = N]$ *where* $M[\alpha = N]$ *is obtained by replacing each sub-term of* M *of the form* $(\alpha\ P)$ *by* $(\alpha\ (P\ N))$*. This substitution will be called a* μ*-substitution whereas the (usual) substitution* $M[x := N]$ *will be called a* λ*-substitution.*

Remarks

- Note that we adopt here a more liberal syntax (also called de Groote's calculus [13]) than in the original calculus since we do not ask that a $\mu\alpha$ is immediately followed by a $(\beta\ M)$ (denoted $[\beta]M$ in Parigot's notation).
- We also have changed Parigot's typing notations. Instead of writing $M : (A_1^{x_1}, \dots, A_n^{x_n} \vdash B, C_1^{\alpha_1}, \dots, C_m^{\alpha_m})$ we have written $x_1 : A_1, \dots, x_n : A_n, \alpha_1 : \neg C_1, \dots, \alpha_m : \neg C_m \vdash M : B$ but, since the first introduction of the $\lambda\mu$-calculus, this is now quite common.
- Unlike for a λ-substitution where, in $M[x := N]$, the variable x has disappeared it is important to note that, in a μ-substitution, the variable α has not disappeared. Moreover its type has changed. If the type of N is U and, in M, the type of α is $\neg(U \to V)$ it becomes $\neg V$ in $M[\alpha = N]$.
- The definition of good congruence is the same as before. As a consequence, we now have the following facts. If $U \approx \perp$, then $U = \perp$ and, if $\neg U \approx \neg V$, then $U \approx V$.
- We also extend all the notations given in section 2. Finally note that lemma 1 remains valid. Moreover, they are easily extended by lemma 10 below.

Lemma 10

1. *If* $\Gamma \vdash \mu\alpha\ M : U$*, then* $\Gamma, \alpha : \neg V \vdash M : \perp$ *for some* V *such that* $U \approx V$.
2. *If* $\Gamma, \alpha : \neg U \vdash (\alpha\, M) : T$*, then* $\Gamma, \alpha : \neg U \vdash M : U$ *and* $T = \perp$.
3. *If* $\Gamma, \alpha : \neg(U \to V) \vdash M : T$ *and* $\Gamma \vdash N : U$*, then* $\Gamma, \alpha : \neg V \vdash M[\alpha = N] : T$.

Theorem 2. *If* $\Gamma \vdash M : T$ *and* $M \triangleright^* M'$*, then* $\Gamma \vdash M' : T$.

Proof. It is enough to show that, if $\Gamma \vdash (\mu\alpha\ M\, N) : T$, then $\Gamma \vdash \mu\alpha\, M[\alpha = N] : T$. Assume $\Gamma \vdash (\mu\alpha\ M\, N) : T$. By lemma 1, $\Gamma \vdash \mu\alpha\ M : U \to V$, $\Gamma \vdash N : U$ and $V \approx T$. Thus, $\Gamma, \alpha : \neg T' \vdash M : \perp$ and $T' \approx U \to V$. By lemma 1, we have $\Gamma, \alpha : \neg(U \to V) \vdash M : \perp$. Since $\Gamma \vdash N : U$ and $V \approx T$, $\Gamma, \alpha : \neg V \vdash M[\alpha = N] : \perp$. Then $\Gamma \vdash \mu\alpha\, M[\alpha = N] : V$ and $\Gamma \vdash \mu\alpha\, M[\alpha = N] : T$. □

4.1 Some Useful Lemmas on the Un-Typed Calculus

Lemma 11. *Let M be a term and $\sigma = \sigma_1 \cup \sigma_2$ where σ_1 (resp. σ_2) is λ (resp. μ) substitution. Assume $M[\sigma] \triangleright^* \mu\alpha M_1$ (resp. $\lambda y M_1$). Then*
- either $M \triangleright^* \mu\alpha M_2$ (resp. $\lambda y M_2$) and $M_2[\sigma] \triangleright^* M_1$
- or $(M \triangleright^* (x\ \overrightarrow{N})$ for some $x \in dom(\sigma_1)$ and $(\sigma(x)\ \overrightarrow{N[\sigma]}) \triangleright^* \mu\alpha M_1$ (resp. $\lambda y M_1$).

Proof. A μ-substitution cannot create a λ or a μ (see, for example, [11]) and thus, the proof is as in lemma 4. □

Lemma 12. *Assume $M, P, \overrightarrow{Q} \in SN$ and $(M\ P\ \overrightarrow{Q}) \notin SN$. Then either ($M \triangleright^* \lambda x M_1$ and $(M_1[x := P]\ \overrightarrow{Q}) \notin SN$) or ($M \triangleright^* \mu\alpha M_1$ and $(\mu\alpha M_1[\alpha = P]\ \overrightarrow{Q}) \notin SN$).*

Proof. As in lemma 2. □

Lemma 13. *Let M be a term and σ be a λ-substitution. Assume $M, \sigma \in SN$ and $M[\sigma] \notin SN$. Then $(\sigma(x)\ \overrightarrow{P[\sigma]}) \notin SN$ for some $(x\ \overrightarrow{P}) \preceq M$ such that $\overrightarrow{P[\sigma]} \in SN$.*

Proof. As in lemma 3. □

Definition 12. *A μ-substitution σ is said to be fair if, for each $\alpha \in dom(\sigma)$, $\alpha \notin Fv(\sigma)$ where $x \in Fv(\sigma)$ (resp. $\beta \in Fv(\sigma)$) means that $x \in Fv(N)$ (resp. $\beta \in Fv(N)$) for some $N \in Im(\sigma)$.*

Lemma 14. *Let σ be is a fair μ-substitution, $\alpha \in dom(\sigma)$ and $x \notin Fv(\sigma)$ (resp. $\beta \notin Fv(\sigma)$), then $M[\sigma][x := \sigma(\alpha)] = M[x := \sigma(\alpha)][\sigma]$ (resp. $M[\sigma][\beta = \sigma(\alpha)] = M[\beta = \sigma(\alpha)][\sigma]$).*

Proof. Immediate. □

Lemma 15. *Let M, N be terms and σ be a fair μ-substitution. Assume $M[\sigma]$, $N \in SN$ but $(M[\sigma]\ N) \notin SN$. Assume moreover that $M[\sigma] \triangleright^* \mu\alpha M_1$. Then, for some $(\alpha\ M_2) \preceq M$, we have $(M_2[\sigma']\ N) \notin SN$ and $M_2[\sigma'] \in SN$ where $\sigma' = [\sigma + \alpha = N]$.*

Proof. By lemma 11, we know that $M \triangleright^* \mu\alpha M_1'$ for some M_1' such that $M_1'[\sigma] \triangleright^* M_1$. Let M' be a sub-term of a reduct of M such that $\langle \eta(M'[\sigma]), cxty(M') \rangle$ is minimum and $M'[\sigma'] \notin SN$. We show that $M' = (\alpha\ M_2)$ and has the desired properties. By minimality, M' cannot be of the form $\lambda x P$, $\mu\beta P$ nor $(\beta\ P)$ for $\beta \neq \alpha$ or $\beta \notin dom(\sigma)$.

If $M' = (P_1\ P_2)$. By the minimality of M', $P_1[\sigma'], P_2[\sigma'] \in SN$. Thus, by lemma 11 and 12, $P_1 \triangleright^* \lambda x Q$ (resp. $P_1 \triangleright^* \mu\beta Q$) such that $Q[\sigma'][x := P_2[\sigma']] = Q[x := P_2][\sigma'] \notin SN$ (resp. $Q[\sigma'][\beta = P_2[\sigma']] = Q[\beta = P_2][\sigma'] \notin SN$) and this contradicts the minimality of M'.

If $M' = (\beta \ P)$ for some $\beta \in dom(\sigma)$. Then $(P[\sigma'] \ \sigma(\beta)) \notin SN$ and, by the minimality of M', $P[\sigma'] \in SN$. Thus, by lemmas 11, 12 and 14, $P \triangleright^* \lambda x Q$ (resp. $P \ \triangleright^* \ \mu\gamma Q$) such that $Q[\sigma'][x := \sigma(\beta)] = Q[x := \sigma(\beta)][\sigma'] \notin SN$ (resp. $Q[\sigma'][\gamma = \sigma(\beta)] = Q[\gamma = \sigma(\beta)][\sigma'] \notin SN$) and this contradicts the minimality of M'.

Thus $M' = (\alpha \ M_2)$ and its minimality implies $M_2[\sigma'] \in SN$. □

4.2 Proof of the Strong Normalization

We use the same notations as in section 3.

Lemma 16. *Let $\mathcal{Y} \subseteq \mathcal{X}$ be such that $H[\{X\}]$ holds for each $X \in \mathcal{Y}$. Then $H[\mathcal{T}(\mathcal{Y})]$ holds.*

Proof. Assume that $H[\{X\}]$ holds for each $X \in \mathcal{Y}$. The result is a special case of the following claim.

Claim : Let M be a term, U, V be types such that $U \in \mathcal{T}(\mathcal{Y})$ and σ be a λ-substitution such that, for each x, $\sigma(x) = N_x[\tau_x]$ where τ_x is a fair μ-substitution such that $dom(\tau_x) \cap Fv(M[\sigma]) = \emptyset$. Assume $\Gamma \vdash M : V$ and for each $x \in dom(\sigma)$, $x : U \in \Gamma$. Assume finally that M and the $N_x[\tau_x]$ are in SN. Then, $M[\sigma] \in SN$.

Proof. By induction on $\langle lg(U), \eta c(M), \eta c(\sigma)\rangle$ where $\eta(\sigma) = \sum \eta(N_x)$ and $cxty(\sigma) = \sum cxty(N_x)$ and, in the sums, each occurrence of a variable counts for one. For example, if there are two occurrences of x_1 and three occurrences of x_2, $cxty(\sigma) = 2\ cxty(N_1) + 3\ cxty(N_2)$. Note that we really mean $cxty(N_x)$ and not $cxty(N_x[\tau_x])$ and similarly for η.

The only non trivial case is when $M = (x \ Q \ \overrightarrow{O})$ for $x \in dom(\sigma)$. By the *IH*, $Q[\sigma], \overrightarrow{O[\sigma]} \in SN$. It is enough to show that $(N_x[\tau_x] \ Q[\sigma]) \in SN$ since $M[\sigma]$ can be written as $M'[\sigma']$ where $M' = (z \ \overrightarrow{O[\sigma]})$ and $\sigma'(z) = (N_x[\tau_x] \ Q[\sigma])$ and (since the size of the type of z is less than the one of U) the *IH* gives the result. By lemma 12, we have two cases to consider.

- $N_x[\tau_x] \triangleright^* \lambda y N_1$. By lemma 11, $N_x \triangleright^* \lambda y N_2$ and the proof is exactly the same as in lemma 7.
- $N_x[\tau_x] \triangleright^* \mu\alpha N_1$. By lemma 15, let $(\alpha \ N_2) \preceq N_x$ be such that $N_2[\tau'] \in SN$ and $R = (N_2[\tau'] \ Q[\sigma]) \notin SN$ where $\tau' = [\tau_x + \alpha = Q[\sigma]]$. But R can be written as $(y \ Q)[\sigma']$ where σ' is the same as σ except that $\sigma'(y) = N_2[\tau']$. Note that $(y \ Q)$ is the same as (or less than) M but one occurrence of x has been replaced by the fresh variable y. The substitution τ' is fair and $dom(\tau') \cap Fv((y \ Q)) = \emptyset$. The *IH* gives a contradiction since $\eta c(\sigma') < \eta c(\sigma)$. Note that the type condition on σ' is satisfied since N_x has type U, thus α has type $\neg U$ and thus N_2 also has type U. □

For now on, we fix some i and we assume $H[\{X_j\}]$ for each $j < i$. Thus, by lemma 16, we know that $H[\mathcal{T}'_i]$ holds. It remains to prove $H[\{X_i\}]$ i.e. proposition 2.

Definition 13. *Let M be a term, $\sigma = \sigma_1 \cup \sigma_2$ where σ_1 (resp. σ_2) is a λ (resp. μ) substitution, $\mathit{\Gamma}$ be a context and U be a type. Say that $(\sigma, \mathit{\Gamma}, M, U)$ is adequate if the following holds:*

- *$\mathit{\Gamma} \vdash M[\sigma] : U$ and $M, \sigma \in SN$.*
- *For each $x \in dom(\sigma_1)$, $\mathit{\Gamma} \vdash \sigma(x) : V_x$ and $V_x \in \mathcal{T}_i^+$.*

Note that nothing is asked on the types of the μ-variables.

Lemma 17. *Let n, m be integers, $\overrightarrow{S}$ be a sequence of terms and $(\delta, \mathit{\Delta}, P, B)$ be adequate. Assume that*

1. *$B \in \mathcal{T}_i^- - \mathcal{T}_i'$ and $\mathit{\Delta} \vdash (P[\delta]\ \overrightarrow{S}) : W$ for some W.*
2. *$\overrightarrow{S} \in SN$, $P \in SN$ and $\eta c(P) < \langle n, m \rangle$.*
3. *$M[\sigma] \in SN$ for every adequate $(\sigma, \mathit{\Gamma}, M, U)$ such that $\eta c(M) < \langle n, m \rangle$.*

Then $(P[\delta]\ \overrightarrow{S}) \in SN$.

Proof. By induction on the length of $\overrightarrow{S}$. The proof is as in lemma 8. The new case is $P[\delta] \triangleright^* \mu\alpha R$ (when $\overrightarrow{S} = S_1 \overrightarrow{S_2}$). By lemma 11, we have two cases to consider.

- $P \triangleright^* \mu\alpha R'$. We have to show that $Q = (\mu\alpha R'[\delta + \alpha = S_1]\ \overrightarrow{S_2}) \in SN$. By lemma 10, the properties of $\approx$ and since $B \in \mathcal{T}_i^-$, there are types B_1, B_2 such that $B \approx B_1 \to B_2$ and $\mathit{\Delta} \vdash \mu\alpha R'[\delta + \alpha = S_1] : B_2$ and $B_2 \in \mathcal{T}_i^-$. Since $\eta c(R') < \langle n, m \rangle$ and $([\delta + \alpha = S_1], \mathit{\Delta} \cup \{\alpha : \neg B_2\}, \mu\alpha R', B_2)$ is adequate, it follows from (3) that $R'[\delta + \alpha = S_1] \in SN$.
  - Assume first $B_2 \in \mathcal{T}_i'$. Since $(z'\ \overrightarrow{S_2}) \in SN$ and $Q = (z'\ \overrightarrow{S_2})[z' := \mu\alpha R'[\delta + \alpha = S_1]]$, the result follows from $H[\mathcal{T}_i']$.
  - Otherwise, the result follows from the *IH* since $([\delta + \alpha = S_1], \mathit{\Delta} \cup \{\alpha : \neg B_2\}, \mu\alpha R', B_2)$ is adequate and the length of $\overrightarrow{S_2}$ is less than the one of $\overrightarrow{S}$.
- $P \triangleright^* (y\ \overrightarrow{T})$ for some λ-variable $y \in dom(\delta)$. As in lemma 8. □

Lemma 18. *Assume $(\sigma, \mathit{\Gamma}, M, A)$ is adequate. Then $M[\sigma] \in SN$.*

Proof. As in the proof of the lemma 16, we prove a more general result. Assume that, for each $x \in dom(\sigma_1)$, $\sigma_1(x) = N_x[\tau_x]$ where τ_x is a fair μ-substitution such that $dom(\tau_x) \cap Fv(M[\sigma]) = \emptyset$. We prove that $M[\sigma] \in SN$.

By induction on $\eta c(M)$ and, by secondary induction, on $\eta c(\sigma_1)$ where $\eta(\sigma_1)$ and $cxty(\sigma_1)$ are defined as in lemma 16. The proof is as in lemma 16. The interesting case is $M = (x\ Q\ \overrightarrow{O})$ for some $x \in dom(\sigma_1)$. The case when $N_x[\tau_x] \triangleright^* \lambda y N'$ is as in lemma 9. The new case is when $N_x[\tau_x] \triangleright^* \mu\alpha N'$. This is done as in lemma 16. Note that, for this point, the type was not used. □

Proposition 2. *Assume $\mathit{\Gamma}, x : X_i \vdash M : U$ and $\mathit{\Gamma} \vdash N : X_i$ and $M, N \in SN$. Then $M[x := N] \in SN$.*

Proof. This follows from lemma 18 since $([x := N], \mathit{\Gamma}, M, U)$ is adequate. □

5 Some Applications

5.1 Representing More Functions

By using recursive types, some terms that cannot be typed in the simply typed λ-calculus become typable. For example, by using the equation $X \approx (X \to T) \to T$, it is possible to type terms containing both $(x\ y)$ and $(y\ x)$ as sub-terms. Just take $x : X$ and $y : X \to T$. By using the equation $X \approx T \to X$, it is possible to apply an unbounded number of arguments to a term.

It is thus natural to try to extend Schwichtenberg's result and to determine the class of functions that are represented in such systems and, in particular, to see whether or not they allow to represent more functions. Note that Doyen [15] and Fortune & all [16] have given extensions of Schwichtenberg's result.

Here is an example of function that cannot be typed (of the good type) in the simply typed λ-calculus.

Let $Nat = (X \to X) \to (X \to X)$ and $Bool = Y \to (Y \to Y)$ where X, Y are type variables. Let $\tilde{n} = \lambda f \lambda x\ (f\ (f\ \ldots\ x)\ \ldots)$ be the church numeral representing n and $\mathbf{0} = \lambda x \lambda y\ y$, $\mathbf{1} = \lambda x \lambda y\ x$ be the terms representing *false* and *true*. Note that $\tilde{n}$ has type Nat and $\mathbf{0}$, $\mathbf{1}$ have type $Bool$.

The term $Inf = \lambda x \lambda y\ (x\, M\, \lambda z \mathbf{1}\, (y\, M\, \lambda z \mathbf{0}))$ where $M = \lambda x\ \lambda y\ (y\, x)$ has been introduced by B.Maurey. It is easy to see that, for every $n, m \in \mathbb{N}$, the term $(\mathrm{Inf}\, \widetilde{m}\, \widetilde{n})$ reduces to $\mathbf{1}$ if $m \leq n$ and to $\mathbf{0}$ otherwise. Krivine has shown in [24] that the type $Nat \to Nat \to Bool$ cannot be given to Inf in system F but, by adding the equation $X \approx (X \to Bool) \to Bool$, it becomes typable. Our example uses the same ideas.

Let $\approx$ be the congruence generated by $X \approx (X \to Bool) \to Bool$. For each $n \in \mathbb{N}^*$, let $Inf_n = \lambda x\ (x\, M\, \lambda y \mathbf{1}\, (M^{n-1}\ \lambda y \mathbf{0}))$ where $(M^k\ P) = (M\ (M\ \ldots\ (M\ P)))$.

Proposition 3. *For each $n \in \mathbb{N}^*$ we have $\vdash Inf_n : Nat \to Bool$.*

Proof. We have $x : X \to Bool, y : X \vdash (y\, x) : Bool$, then $\vdash M : (X \to Bool) \to (X \to Bool)$, thus $\vdash (\widetilde{n}\, M) : (X \to Bool) \to (X \to Bool)$. But $\vdash \lambda y \mathbf{0} : X \to Bool$, therefore $\vdash (\widetilde{n}\, M\, \lambda y \mathbf{0}) : X \to Bool$.

We have $x : X, y : X \to Bool \vdash (y\, x) : Bool$, then $\vdash M : X \to X$, thus $x : Nat \vdash (x\, M) : X \to X$. But $\vdash \lambda y \mathbf{1} : (X \to Bool) \to Bool$, therefore $x : Nat \vdash (x\, M\, \lambda y \mathbf{1}) : X$.

We deduce that $x : Nat \vdash ((\widetilde{n}\, M\, \lambda y \mathbf{0})\, (x\, M\, \lambda y \mathbf{1})) : Bool$, then $x : Nat \vdash (x\, M\, \lambda y \mathbf{1}\, (M^{n-1}\ \lambda y \mathbf{0})) : Bool$ and thus $\vdash Inf_n : Nat \to Bool$. □

Proposition 4. *For each $n \in \mathbb{N}^*$ and $m \in \mathbb{N}$, $(Inf_n\, \widetilde{m})$ reduces to $\mathbf{1}$ if $m \leq n$ and to $\mathbf{0}$ otherwise.*

Proof. $(Inf_n\, \widetilde{m})\ \rhd^*\ (M^m\ \lambda y \mathbf{1}\, (M^{n-1}\ \lambda y \mathbf{0}))\ \rhd^*\ (M^{n-1}\ \lambda y \mathbf{0}\, (M^{m-1}\ \lambda y \mathbf{1}))\ \rhd^*$ $(M^{m-1}\ \lambda y \mathbf{1}\, (M^{n-2}\ \lambda y \mathbf{0}))\ \rhd^*\ (M^{n-2}\ \lambda y \mathbf{0}\, (M^{m-2}\ \lambda y \mathbf{1}))\ \rhd^* \ldots$
$\rhd^* \mathbf{1}$ *if* $m \leq n$ *and* $\mathbf{0}$ *otherwise.* □

Remarks

Note that for the (usual) simply typed λ-calculus we could have taken for X and Y the same variable but, for propositions 3 and 4, we cannot assume that $X = Y$ because then the condition of positivity would not be satisfied. This example is thus not completely satisfactory and it actually shows that the precise meaning of the question "which functions can be represented in such systems" is not so clear.

5.2 A Translation of the $\lambda\mu$-Calculus into the λ-Calculus

The strong normalization of a typed $\lambda\mu$-calculus can be deduced from the one of the corresponding typed λ-calculus by using CPS translations. See, for example, [14] for such a translation. There is another, somehow simpler, way of doing such a translation. Add, for each atomic type X, a constant a_X of type $\neg\neg X \to X$. Using these constants, it is not difficult to get, for each type T, a λ-term M_T (depending on T) such that M_T has type $\neg\neg T \to T$. This gives a translation of the $\lambda\mu$-calculus into the λ-calculus from which the strong normalization of the $\lambda\mu$-calculus can be deduced from the one of the λ-calculus. This translation, quite different from the CPS translations, has been used by Krivine [26] to code the $\lambda\mu$-calculus with second order types in the $\lambda\mathcal{C}$-calculus.

With recursive equations, we do not have to add the constant a_X since we can use the equation $X \approx \neg\neg X$. We give here, without proof, the translation. We denote by $S_\approx$ the simply typed λ-calculus where $\approx$ is the congruence on $\mathcal{T}$ (where $\mathcal{A} = \{\perp\}$) generated by $X \approx \neg\neg X$ for each X and by $S_{\lambda\mu}$ the usual (i.e. without recursive types) $\lambda\mu$-calculus.

Definition 14

1. *We define, for each type T, a closed λ-term M_T such that $\vdash_\approx M_T : \neg\neg T \to T$ as follows. This is done by induction on T.*
   - $M_\perp = \lambda x\,(x\ I)$ *where* $I = \lambda x\ x$.
   - *If* $X \in \mathcal{X}$, $M_X = I$.
   - $M_{U\to V} = \lambda x \lambda y\ (M_V\ \lambda z(x\ \ \lambda t(z\ (t\ y))))$
2. *We define a translation from $S_{\lambda\mu}$ to $S_\approx$ as follows.*
   - $x^* = x$.
   - $(\lambda x\, M)^* = \lambda x\, M^*$.
   - $(M\, N)^* = (M^*\, N^*)$.
   - $(\mu\alpha\, M)^* = (M_U\ \lambda\alpha\, M^*)$ *if* α *has the type* $\neg U$.
   - $(\alpha\, M)^* = (\alpha\, M^*)$.

For a better understanding, in the translation of $\mu\alpha M$ and $(\alpha\, M)$, we have kept the same name to the variable α but it should be clear that the translated terms are λ-terms with only on kind of variables.

Lemma 19. *If* $\Gamma \vdash_{\lambda\mu} M : U$ *then* $\Gamma \vdash_\approx M^* : U$.

Lemma 20. *Let* M, N *be typed* $\lambda\mu$*-terms. If* $M \triangleright N$, *then* $M^* \triangleright^+ N^*$.

Proof. It is enough to check that $(\mu\alpha M\ N)^* \triangleright^+ (\mu\alpha M[\alpha = N])^*$. $\square$

Theorem 3. *The strong normalization of $S_{\approx}$ implies the one of $S_{\lambda\mu}$.*

Proof. By lemmas 19 and 20. □

Remark
Note that the previous translation cannot be used to show that the $\lambda\mu$-calculus with recursive types is strongly normalizing since having two equations (for example $X \approx \neg\neg X$ and $X \approx F$) is problematic.

6 Remarks and Open Questions

1. The proof of the strong normalization of the system D of intersection types [6] is exactly the same as the one for simple types. Is it possible to extend our proof to such systems with equations ? Note that the sort of constraints that must be given on the equations is not so clear. For example, what does that mean to be positive in $A \wedge B$? To be positive both in A and B ? in one of them ? It will be interesting to check precisely because, for example, it is known that the system[1] given by system D and the equations $X \approx (Y \to X) \wedge (X \to X)$ and $Y \approx X \to Y$ is strongly normalizing (but the proof again is not formalized in Peano arithmetic) though the positivity condition is violated.
2. We could add other typing rules and constructors to ensure that, intuitively, X represents the *least fixed point* of the equation $X \approx F$. This kind of thing is done, for example, in TTR. What can be said for such systems?
3. There are many translations from, for example, the $\lambda\mu$-calculus into the λ-calculus that allows to deduce the strong normalization of the former by the one of the latter. These CPS transformations differ from the one given in section 5.2 by the fact that the translation of a term does not depend on its type. What is the behavior of such translations with recursive equations ?

Acknowledgments

We would like to thank P Urzyczyn who has mentioned to us the question solved here and has also indicated some errors appearing in previous versions of our proofs. Thanks also to the referees and their valuable remarks.

References

1. Barendregt, H.P.: The Lambda Calculus, Its Syntax and Semantics. North-Holland, Amsterdam (1985)
2. Barendregt, H.P.: Lambda Calculi with types. In: Abramsky & al. pp. 117–309 (1992)
3. Barendregt, H.P., Dekkers, W., Statman, R.: Typed lambda calculus (To appear)

[1] This example appears in a list of open problems of the working group Gentzen, Utrecht 1993.

4. Berger, U., Schwichtenberg, H.: An Inverse of the Evaluation Functional for Typed lambda-calculus. LICS, pp. 203–211 (1991)
5. Church, A.: A Formulation of the Simple Theory of Types. JSL 5 (1940)
6. Coppo, M., Dezani, M.: A new type assignment for lambda terms. Archiv. Math. Logik (19), 139–156 (1978)
7. Coquand, T., Huet, G.: A calculus of constructions. Information and Computation (76), 95–120 (1988)
8. David, R.: Normalization without reducibility. Annals of Pure. and Applied Logic (107), 121–130 (2001)
9. David, R.: A short proof of the strong normalization of the simply typed lambda calculus (www.lama.univ-savoie.fr/~david)
10. David, R., Nour, K.: A short proof of the strong normalization of the simply typed $\lambda\mu$-calculus. Schedae Informaticae (12), 27–34 (2003)
11. David, R., Nour, K.: Arithmetical proofs of strong normalization results for the symmetric lambda-mu-calculus. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 162–178. Springer, Heidelberg (2005)
12. David, R., Nour, K.: Arithmetical proofs of strong normalization results for symmetric λ-calculi. Fundamenta Informaticae (To appear)
13. de Groote, P.: On the Relation between the Lambda-Mu-Calculus and the Syntactic Theory of Sequential Control. LPAR, pp. 31–43 (1994)
14. de Groote, P.: A CPS-Translation of the $\lambda\mu$-Calculus. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787, pp. 85–99. Springer, Heidelberg (1994)
15. Doyen, J.: Quelques propriétés du typage des fonctions des entiers dans les entiers. C.R. Acad. Sci. Paris, t.321, Série I, 663–665 (1995)
16. Fortune, S., Leivant, D., O'Donnell, M.: Simple and Second order Types Structures. JACM 30-1, 151–185 (1983)
17. Friedman, H.: Equality between functionals. Logic Coll'73, LNM 453 pp. 22–37 (1975)
18. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press, Cambridge (1989)
19. Gödel, K.: Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. Dialectica 12, 280–287 (1958)
20. Goldfarb, W.D.: The undecidability of the 2nd order unification problem. TCS (13), 225–230 (1981)
21. Huet, G.P.: The Undecidability of Unification in Third Order Logic. Information and Control 22(3), 257–267 (1973)
22. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Archive for Mathematical Logic 42(1), 59–87 (2003)
23. Jung, A., Tiuryn, J.A.: A New Characterization of Lambda Definability. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 245–257. Springer, Heidelberg (1993)
24. Krivine, J.-L.: Un algorithme non typable dans le système. F. C. R. Acad. Sci. Paris vol. 304(5) (1987)
25. Krivine, J.-L.: Lambda Calcul: types et modèles. Masson, Paris (1990)
26. Krivine, J.-L.: Classical Logic, Storage Operators and Second-Order lambda-Calculus. Ann. Pure Appl. Logic 68(1), 53–78 (1994)
27. Lambek, J.: Cartesian Closed Categories and Typed Lambda-calculi. Combinators and Functional Programming Languages, pp. 136-175 (1985)
28. Loader, R.: The Undecidability of λ-definability. In: Essays in memory of A. Church, pp. 331–342 (2001)

29. Mendler, N.P.: Recursive Types and Type Constraints in Second-Order Lambda Calculus. LICS, pp. 30–36 (1987)
30. Mendler, N.P.: Inductive Types and Type Constraints in the Second-Order Lambda Calculus. Ann. Pure Appl. Logic 51(1-2), 159–172 (1991)
31. Parigot, M.: Programming with proofs: a second order type theory. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 145–159. Springer, Heidelberg (1988)
32. Parigot, M.: On representation of data in lambda calculus. CSL, pp. 309–321 (1989)
33. Parigot, M.: Recursive programming with proofs. Theoritical Computer Science 94, 335–356 (1992)
34. Parigot, M.: Strong Normalization for Second Order Classical Natural Deduction. LICS, pp. 39–46 (1993)
35. Parigot, M.: $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. Journal of symbolic logic 62-4, 1461–1479 (1997)
36. Parigot, M.: Proofs of Strong Normalisation for Second Order Classical Natural Deduction. J. Symb. Log. 62(4), 1461–1479 (1997)
37. Plotkin, G.D.: Lambda-definability and logical relations. Technical report (1973)
38. Schwichtenberg, H.: Functions definable in the simply-typed lambda calculus. Arch. Math Logik 17, 113–114 (1976)
39. Statman, R.: The Typed lambda-Calculus is not Elementary Recursive. FOCS, pp. 90–94 (1977)
40. Statman, R.: λ-definable functionals and $\beta\eta$-conversion. Arch. Math. Logik 23, 21–26 (1983)
41. Statman, R.: Recursive types and the subject reduction theorem. Technical report, Carnegie Mellon University, pp. 94–164 (March 1994)
42. Tait, W.W.: Intensional Interpretations of Functionals of Finite Type I. JSL, vol. 32(2) (1967)
43. Weiermann, A.: A proof of strongly uniform termination for Gödel's T by methods from local predicativity. Archive fot Mathematical Logic 36, 445–460 (1997)

Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo

Denis Cousineau and Gilles Dowek

École polytechnique and INRIA
LIX, École polytechnique, 91128 Palaiseau Cedex, France
Cousineau@lix.polytechnique.fr
http://www.lix.polytechnique.fr/~cousineau
Gilles.Dowek@polytechnique.edu
http://www.lix.polytechnique.fr/~dowek

Abstract. The lambda-Pi-calculus allows to express proofs of minimal predicate logic. It can be extended, in a very simple way, by adding computation rules. This leads to the lambda-Pi-calculus modulo. We show in this paper that this simple extension is surprisingly expressive and, in particular, that all functional Pure Type Systems, such as the system F, or the Calculus of Constructions, can be embedded in it. And, moreover, that this embedding is conservative under termination hypothesis.

The $\lambda\Pi$*-calculus* is a dependently typed lambda-calculus that allows to express proofs of minimal predicate logic through the Brouwer-Heyting-Kolmogorov interpretation and the Curry-de Bruijn-Howard correspondence. It can be extended in several ways to express proofs of some theory. A first solution is to express the theory in Deduction modulo [7,9], *i.e.* to orient the axioms as rewrite rules and to extend the $\lambda\Pi$-calculus to express proofs in Deduction modulo [3]. We get this way the $\lambda\Pi$*-calculus modulo*. This idea of extending the dependently typed lambda-calculus with rewrite rules is also that of Intuitionistic type theory used as a logical framework [13].

A second way to extend the $\lambda\Pi$-calculus is to add typing rules, in particular to allow polymorphic typing. We get this way the *Calculus of Constructions* [4] that allows to express proofs of simple type theory and more generally the *Pure Type Systems* [2,15,1]. These two kinds of extensions of the $\lambda\Pi$-calculus are somewhat redundant. For instance, simple type theory can be expressed in deduction modulo [8], hence the proofs of this theory can be expressed in the $\lambda\Pi$-calculus modulo. But they can also be expressed in the Calculus of Constructions. This suggests to relate and compare these two ways to extend the $\lambda\Pi$-calculus.

We show in this paper that all functional Pure Type Systems can be embedded in the $\lambda\Pi$-calculus modulo using an appropriate rewrite system. This rewrite system is inspired both by the expression of simple type theory in Deduction modulo and by the mechanisms of universes *à la* Tarski [12] of Intuitionistic type theory. In particular, this work extends Palmgren's construction of an impredicative universe in type theory [14].

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 102–117, 2007.
© Springer-Verlag Berlin Heidelberg 2007

1 The $\lambda\Pi$-Calculus

The *λΠ-calculus* is a dependently typed lambda-calculus that permits to construct types depending on terms, for instance a type *array n*, of arrays of size n, that depends on a term n of type *nat*. It also permits to construct a function f taking a natural number n as an argument and returning an array of size n. Thus, the arrow type $nat \Rightarrow array$ of simply typed lambda-calculus must be extended to a dependent product type $\Pi x : nat\ (array\ x)$ where, in the expression $\Pi x : A\ B$, the occurrences of the variable x are bound in B by the symbol Π (the expression $A \Rightarrow B$ is used as a shorter notation for the expression $\Pi x : A\ B$ when x has no free occurrence in B). When we apply the function f to a term n, we do not get a term of type *array x* but of type *array n*. Thus, the application rule must include a substitution of the term n for the variable x. The symbol *array* itself takes a natural number as an argument and returns a type. Thus, its type is $nat \Rightarrow Type$, *i.e.* $\Pi x : nat\ Type$. The terms $Type$, $nat \Rightarrow Type$, ... cannot have type $Type$, because Girard's paradox [10] could then be expressed in the system, thus we introduce a new symbol $Kind$ to type such terms. To form terms, like $\Pi x : nat\ Type$, whose type is $Kind$, we need a rule expressing that the symbol $Type$ has type $Kind$ and a new product rule allowing to form the type $\Pi x : nat\ Type$, whose type is $Kind$. Besides the variables such as x whose type has type $Type$, we must permit the declaration of variables such as *nat* of type $Type$, and more generally, variables such as *array* whose type has type $Kind$. This leads to introduce the following syntax and typing rules.

Definition 1 (The syntax of $\lambda\Pi$). *The syntax of the λΠ-calculus is*

$$t = x \mid Type \mid Kind \mid \Pi x : t\ t \mid \lambda x : t\ t \mid t\ t$$

The α-equivalence and β-reduction relations are defined as usual and terms are identified modulo α-equivalence.

Definition 2 (The typing rules of $\lambda\Pi^-$).

$$\frac{}{[\,]\ \textit{well-formed}}\ \textbf{Empty}$$

$$\frac{\Gamma \vdash A : Type}{\Gamma[x : A]\ \ \textit{well-formed}}\ \textbf{Declaration}\ x\ \textit{not in}\ \Gamma$$

$$\frac{\Gamma \vdash A : Kind}{\Gamma[x : A]\ \ \textit{well-formed}}\ \textbf{Declaration2}\ x\ \textit{not in}\ \Gamma$$

$$\frac{\Gamma\ \ \textit{well-formed}}{\Gamma \vdash Type : Kind}\ \textbf{Sort}$$

$$\frac{\Gamma\ \ \textit{well-formed}\quad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textbf{Variable}$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Type}{\Gamma \vdash \Pi x : A\ B : Type} \textbf{Product}$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Kind}{\Gamma \vdash \Pi x : A\ B : Kind} \textbf{Product2}$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Type \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B} \textbf{Abstraction}$$

$$\frac{\Gamma \vdash t : \Pi x : A\ B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : (u/x)B} \textbf{Application}$$

It is useful, in some situations, to add a rule allowing to build type families by abstraction, for instance $\lambda x : nat\ (array\ (2 \times x))$ and rules asserting that a term of type $(\lambda x : nat\ (array\ (2 \times x))\ n)$ also has type $array\ (2 \times n)$. This leads to introduce the following extra typing rules.

Definition 3 (The typing rules of $\lambda\Pi$). *The typing rules of $\lambda\Pi$ are those of $\lambda\Pi^-$ and*

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Kind \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B} \textbf{Abstraction2}$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash t : A}{\Gamma \vdash t : B} \textbf{Conversion}\ A \equiv_\beta B$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind \quad \Gamma \vdash t : A}{\Gamma \vdash t : B} \textbf{Conversion2}\ A \equiv_\beta B$$

where $\equiv_\beta$ is the β-equivalence relation.

It can be proved that types are preserved by β-reduction, that β-reduction is confluent and strongly terminating and that each term has a unique type modulo β-equivalence.

The $\lambda\Pi$-calculus, and even the $\lambda\Pi^-$-calculus, can be used to express proofs of minimal predicate logic, following the Brouwer-Heyting-Kolmogorov interpretation and the Curry-de Bruijn-Howard correspondence. Let $\mathcal{L}$ be a language in predicate logic, we consider a context Γ formed with a variable ι of type $Type$ – or variables $\iota_1, \ldots, \iota_n$ of type $Type$ when $\mathcal{L}$ is many-sorted —, for each function symbol f of $\mathcal{L}$, a variable f of type $\iota \Rightarrow \ldots \Rightarrow \iota \Rightarrow \iota$ and for each predicate symbol P of $\mathcal{L}$, a variable P of type $\iota \Rightarrow \ldots \Rightarrow \iota \Rightarrow Type$.

To each formula P containing free variables $x_1, \ldots, x_p$ we associate a term P° of type $Type$ in the context $\Gamma, x_1 : \iota, \ldots, x_p : \iota$ translating each variable, function symbol and predicate symbol by itself and the implication symbol and the universal quantifier by a product.

To each proof π, in minimal natural deduction, of a sequent $A_1, \ldots, A_n \vdash B$ with free variables $x_1, \ldots, x_p$, we can associate a term π° of type B° in the context $\Gamma, x_1 : \iota, \ldots, x_p : \iota, \alpha_1 : A_1^\circ, \ldots, \alpha_n : A_n^\circ$. From the strong termination of the $\lambda\Pi$-calculus, we get cut elimination for minimal predicate logic. If B is an atomic formula, there is no cut free proof, hence no proof at all, of $\vdash B$.

2 The $\lambda\Pi$-Calculus Modulo

The $\lambda\Pi$-calculus allows to express proofs in pure minimal predicate logic. To express proofs in a theory $\mathcal{T}$, we can declare a variable for each axiom of $\mathcal{T}$ and consider proofs-terms containing such free variables, this is the idea of the Logical Framework [11]. However, when considering such open terms most benefits of termination, such as the existence of empty types, are lost.

An alternative is to replace axioms by rewrite rules, moving from predicate logic to *Deduction modulo* [7,9]. Such extensions of type systems with rewrite rules to express proofs in Deduction modulo have been defined in [3] and [13]. We shall present now an extension of the $\lambda\Pi$-calculus: the $\lambda\Pi$-calculus modulo.

Recall that if Σ, Γ and Δ are contexts, a substitution θ, binding the variables declared in Γ, is said to be *of type* $\Gamma \leadsto \Delta$ *in* Σ if for all x declared of type T in Γ, we have $\Sigma\Delta \vdash \theta x : \theta T$, and that, in this case, if $\Sigma\Gamma \vdash u : U$, then $\Sigma\Delta \vdash \theta u : \theta U$.

A *rewrite rule* is a quadruple $l \longrightarrow^{\Gamma,T} r$ where Γ is a context and l, r and T are β-normal terms. Such a rule is said to be *well-typed* in the context Σ if, in the $\lambda\Pi$-calculus, the context $\Sigma\Gamma$ is well-formed and the terms l and r have type T in this context.

If Σ is a context, $l \longrightarrow^{\Gamma,T} r$ is a rewrite rule well-typed in Σ and θ is a substitution of type $\Gamma \leadsto \Delta$ in Σ then the terms θl and θr both have type θT in the context $\Sigma\Delta$. We say that the term θl *rewrites* to the term θr.

If Σ is a context and $\mathcal{R}$ a set of rewrite rules well-typed in the $\lambda\Pi$-calculus in Σ, then the *congruence generated by* $\mathcal{R}$, $\equiv_{\mathcal{R}}$, is the smallest congruence such that if t rewrites to u then $t \equiv_{\mathcal{R}} u$.

Definition 4 ($\lambda\Pi$-modulo). *Let Σ be a context and $\mathcal{R}$ a rewrite system in Σ. Let $\equiv_{\beta\mathcal{R}}$ be the congruence of terms generated by the rules of $\mathcal{R}$ and the rule β.*

*The $\lambda\Pi$-*calculus modulo $\mathcal{R}$ *is the extension of the $\lambda\Pi$-calculus obtained by replacing the relation $\equiv_\beta$ by $\equiv_{\beta\mathcal{R}}$ in the conversion rules*

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}\ \textbf{Conversion}\ A \equiv_{\beta\mathcal{R}} B$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}\ \textbf{Conversion2}\ A \equiv_{\beta\mathcal{R}} B$$

Notice that we can also extend the $\lambda\Pi^-$-calculus with rewrite rules. In this case, we introduce conversion rules, using the congruence defined by the system $\mathcal{R}$ alone.

Example 1. Consider the context $\Sigma = [P : Type,\ Q : Type]$ and the rewrite system $\mathcal{R}$ formed with the rule $P \longrightarrow (Q \Rightarrow Q)$. The term $\lambda f : P\ \lambda x : Q\ (f\ x)$ is well-typed in the $\lambda\Pi$-calculus modulo $\mathcal{R}$.

3 The Pure Type Systems

There are several ways to extend the functional interpretation of proofs to simple type theory. The first is to use the fact that simple type theory can be

expressed in Deduction modulo with rewrite rules only [8]. Thus, the proofs of simple type theory can be expressed in the $\lambda\Pi$-calculus modulo, and even in the $\lambda\Pi^-$-calculus modulo. The second is to extend the $\lambda\Pi$-calculus by adding the following typing rules, allowing for instance the construction of the type $\Pi P : Type\ (P \Rightarrow P)$.

$$\frac{\Gamma \vdash A : Kind \quad \Gamma[x:A] \vdash B : Type}{\Gamma \vdash \Pi x : A\ B : Type}\ \textbf{Product3}$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma[x:A] \vdash B : Kind}{\Gamma \vdash \Pi x : A\ B : Kind}\ \textbf{Product4}$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma[x:A] \vdash B : Type \quad \Gamma[x:A] \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B}\ \textbf{Abstraction3}$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma[x:A] \vdash B : Kind \quad \Gamma[x:A] \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B}\ \textbf{Abstraction4}$$

We obtain the *Calculus of Constructions* [4].

The rules of the simply typed λ-calculus, the $\lambda\Pi$-calculus and of the Calculus of Constructions can be presented in a schematic way as follows.

Definition 5 (Pure type system). *A* Pure Type System *[2,15,1] P is defined by a set S, whose elements are called* sorts*, a subset A of $S \times S$, whose elements are called* axioms *and a subset R of $S \times S \times S$, whose elements are called* rules*. The typing rules of P are*

$$\frac{}{[\,]\ \textit{well-formed}}\ \textbf{Empty}$$

$$\frac{\Gamma \vdash A : s}{\Gamma[x:A]\ \ \textit{well-formed}}\ \textbf{Declaration}\ s \in S \textit{ and } x \textit{ not in } \Gamma$$

$$\frac{\Gamma\ \textit{well-formed}}{\Gamma \vdash s_1 : s_2}\ \textbf{Sort}\ \langle s_1, s_2 \rangle \in A$$

$$\frac{\Gamma\ \textit{well-formed} \quad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textbf{Variable}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma[x:A] \vdash B : s_2}{\Gamma \vdash \Pi x : A\ B : s_3}\ \textbf{Product}\ \langle s_1, s_2, s_3 \rangle \in R$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma[x:A] \vdash B : s_2 \quad \Gamma[x:A] \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B}\ \textbf{Abstraction}\ \langle s_1, s_2, s_3 \rangle \in R$$

$$\frac{\Gamma \vdash t : \Pi x : A\ B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : (u/x)B}\ \textbf{Application}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}\ \textbf{Conversion}\ s \in S \quad A \equiv_\beta B$$

The simply typed λ-calculus is the system defined by the sorts $Type$ and $Kind$, the axiom $\langle Type, Kind \rangle$ and the rule $\langle Type, Type, Type \rangle$. The $\lambda\Pi$-calculus is the system defined by the same sorts and axiom and the rules $\langle Type, Type, Type \rangle$ and $\langle Type, Kind, Kind \rangle$. The Calculus of Constructions is the system defined by the same sorts and axiom and the rules $\langle Type, Type, Type \rangle$, $\langle Type, Kind, Kind \rangle$, $\langle Kind, Type, Type \rangle$ and $\langle Kind, Kind, Kind \rangle$. Other examples of Pure Type Systems are Girard's systems F and Fω.

In all Pure Type Systems, types are preserved under reduction and the β-reduction relation is confluent. It terminates in some systems, such as the $\lambda\Pi$-calculus, the Calculus of Constructions, the system F and the system Fω. Uniqueness of types is lost in general, but it holds for the $\lambda\Pi$-calculus, the Calculus of Constructions, the system F and the system Fω, and more generally for all functional Pure Type Systems.

Definition 6 (Functional Type System). *A type system is said to be* functional *if*

$$\langle s_1, s_2 \rangle \in A \textit{ and } \langle s_1, s_3 \rangle \in A \textit{ implies } s_2 = s_3$$

$$\langle s_1, s_2, s_3 \rangle \in R \textit{ and } \langle s_1, s_2, s_4 \rangle \in R \textit{ implies } s_3 = s_4$$

4 Embedding Functional Pure Type Systems in the $\lambda\Pi$-Calculus Modulo

We have seen that the $\lambda\Pi$-calculus modulo and the Pure Type Systems are two extensions of the $\lambda\Pi$-calculus. At a first glance, they seem quite different as the latter adds more typing rules to the $\lambda\Pi$-calculus, while the former adds more computation rules. But they both allow to express proofs of simple type theory.

We show in this section that functional Pure Type Systems can, in fact, be embedded in the $\lambda\Pi$-calculus modulo with an appropriate rewrite system.

4.1 Definition

Consider a functional Pure Type System $P = \langle S, A, R \rangle$. We build the following context and rewrite system.

The context Σ_P contains, for each sort s, two variables

$$U_s : Type \quad \textit{and} \quad \varepsilon_s : U_s \Rightarrow Type$$

for each axiom $\langle s_1, s_2 \rangle$, a variable

$$\dot{s_1} : U_{s_2}$$

and for each rule $\langle s_1, s_2, s_3 \rangle$, a variable

$$\dot{\Pi}_{\langle s_1, s_2, s_3 \rangle} : \Pi X : U_{s_1} \ (((\varepsilon_{s_1} \ X) \Rightarrow U_{s_2}) \Rightarrow U_{s_3})$$

The type U_s is called the *universe* of s and the symbol ε_s the *decoding function* of s.

The rewrite rules are

$$\varepsilon_{s_2}(\dot{s}_1) \longrightarrow U_{s_1}$$

in the empty context and with the type $Type$, and

$$\varepsilon_{s_3}(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ X\ Y) \longrightarrow \Pi x : (\varepsilon_{s_1}\ X)\ (\varepsilon_{s_2}\ (Y\ x))$$

in the context $X : U_{s_1}, Y : (\varepsilon_{s_1}\ X) \Rightarrow U_{s_2}$ and with the type $Type$.

These rules are called the *universe-reduction rules*, we write $\equiv_P$ for the equivalence relation generated by these rules and the rule β and we call the $\lambda\Pi_P$-calculus the $\lambda\Pi$-calculus modulo these rewrite rules and the rule β. To ease notations, in the $\lambda\Pi_P$-calculus, we do not recall the context Σ_P in each sequent and write $\Gamma \vdash t : T$ for $\Sigma_P \Gamma \vdash t : T$, and we note $\equiv$ for $\equiv_P$ when there is no ambiguity.

Example 2. The embedding of the Calculus of Constructions is defined by the context

$$\dot{Type} : U_{Kind} \qquad U_{Type} : Type \qquad U_{Kind} : Type$$

$$\varepsilon_{Type} : U_{Type} \Rightarrow Type \qquad \varepsilon_{Kind} : U_{Kind} \Rightarrow Type$$

$$\dot{\Pi}_{\langle Type,Type,Type\rangle} : \Pi X : U_{Type}\ (((\varepsilon_{Type}\ X) \Rightarrow U_{Type}) \Rightarrow U_{Type})$$

$$\dot{\Pi}_{\langle Type,Kind,Kind\rangle} : \Pi X : U_{Type}\ (((\varepsilon_{Type}\ X) \Rightarrow U_{Kind}) \Rightarrow U_{Kind})$$

$$\dot{\Pi}_{\langle Kind,Type,Type\rangle} : \Pi X : U_{Kind}\ (((\varepsilon_{Kind}\ X) \Rightarrow U_{Type}) \Rightarrow U_{Type})$$

$$\dot{\Pi}_{\langle Kind,Kind,Kind\rangle} : \Pi X : U_{Kind}\ (((\varepsilon_{Kind}\ X) \Rightarrow U_{Kind}) \Rightarrow U_{Kind})$$

and the rules

$$\varepsilon_{Kind}(\dot{Type}) \longrightarrow U_{Type}$$

$$\varepsilon_{Type}(\dot{\Pi}_{\langle Type,Type,Type\rangle}\ X\ Y) \longrightarrow \Pi x : (\varepsilon_{Type}\ X)\ (\varepsilon_{Type}\ (Y\ x))$$

$$\varepsilon_{Kind}(\dot{\Pi}_{\langle Type,Kind,Kind\rangle}\ X\ Y) \longrightarrow \Pi x : (\varepsilon_{Type}\ X)\ (\varepsilon_{Kind}\ (Y\ x))$$

$$\varepsilon_{Type}(\dot{\Pi}_{\langle Kind,Type,Type\rangle}\ X\ Y) \longrightarrow \Pi x : (\varepsilon_{Kind}\ X)\ (\varepsilon_{Type}\ (Y\ x))$$

$$\varepsilon_{Kind}(\dot{\Pi}_{\langle Kind,Kind,Kind\rangle}\ X\ Y) \longrightarrow \Pi x : (\varepsilon_{Kind}\ X)\ (\varepsilon_{Kind}\ (Y\ x))$$

Definition 7 (Translation). *Let Γ be a context in a functional Pure Type System P and t a term well-typed in Γ, we defined the translation $|t|$ of t in Γ, that is a term in $\lambda\Pi_P$, as follows*

- $|x| = x$,
- $|s| = \dot{s}$,
- $|\Pi x : A\ B| = \dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ |A|\ (\lambda x : (\varepsilon_{s_1}\ |A|)\ |B|)$, *where s_1 is the type of A, s_2 is the type of B and s_3 the type of $\Pi x : A\ B$,*
- $|\lambda x : A\ t| = \lambda x : (\varepsilon_s\ |A|)\ |t|$,
- $|t\ u| = |t|\ |u|$.

Definition 8 (Translation as a type). *Consider a term A of type s for some sort s. The* translation of A as a type *is*

$$\|A\| = \varepsilon_s \ |A|.$$

Note that if A is a well-typed sort s' then

$$\|s'\| = \varepsilon_s \ \dot{s}' \equiv_P U_{s'}.$$

We extend this definition to non well-typed sorts, such as the sort $Kind$ in the Calculus of Constructions, by

$$\|s'\| = U_{s'}$$

The translation of a well formed context is defined by

$$\|[\,]\| = [\,] \quad \textit{and} \quad \|\Gamma[x : A]\| = \|\Gamma\|[x : \|A\|]$$

4.2 Soundness

Proposition 1

1. $|(u/x)t| = (|u|/x)|t|$, $\|(u/x)t\| = (|u|/x)\|t\|$.
2. *If* $t \longrightarrow_\beta u$ *then* $|t| \longrightarrow_\beta |u|$.

Proof

1. By induction on t.
2. Because a β-redex is translated as a β-redex.

Proposition 2. $\|\Pi x : A\ B\| \equiv_P \Pi x : \|A\|\ \|B\|$

Proof. Let s_1 be the type of A, s_2 that of B and s_3 that of $\Pi x : A\ B$. We have $\|\Pi x : A\ B\| = \varepsilon_{s_3}\ |\Pi x : A\ B| = \varepsilon_{s_3}\ (\dot{\Pi}_{\langle s_1, s_2, s_3\rangle}\ |A|\ (\lambda x : (\varepsilon_{s_1}\ |A|)\ |B|))$
$\equiv_P \Pi x : (\varepsilon_{s_1}\ |A|)\ (\varepsilon_{s_2}\ ((\lambda x : (\varepsilon_{s_1}\ |A|)\ |B|)\ x)) \equiv_P \Pi x : (\varepsilon_{s_1}\ |A|)\ (\varepsilon_{s_2}\ |B|)$
$= \Pi x : \|A\|\ \|B\|$.

Example 3. In the Calculus of Constructions, the translation as a type of $\Pi X : Type\ (X \Rightarrow X)$ is $\Pi X : U_{Type}\ ((\varepsilon_{Type}\ X) \Rightarrow (\varepsilon_{Type}\ X))$. The translation as a term of $\lambda X : Type\ \lambda x : X\ x$ is the term $\lambda X : U_{Type}\ \lambda x : (\varepsilon_{Type} X)\ x$. Notice that the former is the type of the latter. The generalization of this remark is the following proposition.

Proposition 3 (Soundness)

$$\textit{If} \ \ \Gamma \vdash t : B \ \ \textit{in } P \textit{ then} \ \ \|\Gamma\| \vdash |t| : \|B\| \ \ \textit{in } \lambda\Pi_P.$$

Proof. By induction on t.

- If t is a variable, this is trivial.
- If $t = s_1$ then $B = s_2$ (where $\langle s_1, s_2\rangle$ is an axiom), we have $\dot{s_1} : U_{s_2} = \|s_2\|$.

– If $t = \Pi x : C\ D$, let s_1 be the type of C, s_2 that of D and s_3 that of t. By induction hypothesis, we have

$$\|\Gamma\| \vdash |C| : U_{s_1} \quad \textit{and} \quad \|\Gamma\|, x : \|C\| \vdash |D| : U_{s_2}$$

i.e.

$$\|\Gamma\|, x : (\varepsilon_{s_1}\ |C|) \vdash |D| : U_{s_2}$$

Thus

$$\|\Gamma\| \vdash (\dot{\Pi}_{\langle s_1, s_2, s_3 \rangle}\ |C|\ \lambda x : (\varepsilon_{s_1}\ |C|)\ |D|) : U_{s_3}$$

i.e.

$$\|\Gamma\| \vdash |\Pi x : C\ D| : \|s_3\|$$

– If $t = \lambda x : C\ u$, then we have

$$\Gamma, x : C \vdash u : D$$

and $B = \Pi x : C\ D$. By induction hypothesis, we have

$$\|\Gamma\|, x : \|C\| \vdash |u| : \|D\|$$

i.e.

$$\|\Gamma\|, x : (\varepsilon_{s_1}\ |C|) \vdash |u| : \|D\| \quad \textit{then} \quad \|\Gamma\| \vdash \lambda x : (\varepsilon_{s_1}\ |C|)\ |u| : \Pi x : \|C\|\ \|D\|$$

i.e.

$$\|\Gamma\| \vdash |t| : \|\Pi x : C\ D\|$$

– If $t = u\ v$, then we have

$$\Gamma \vdash u : \Pi x : C\ D, \quad \Gamma \vdash v : C$$

and $B = (v/x)D$. By induction hypothesis, we get

$$\|\Gamma\| \vdash |u| : \|\Pi x : C\ D\| = \Pi x : \|C\|\ \|D\| \quad \textit{and} \quad \|\Gamma\| \vdash |v| : \|C\|$$

Thus

$$\|\Gamma\| \vdash |t| : (|v|/x)\|D\| = \|(v/x)D\|$$

4.3 Termination

Proposition 4. *If $\lambda\Pi_P$ terminates then P terminates.*

Proof. Let t_1 be a well-typed term in P and $t_1, t_2, \ldots$ be a reduction sequence of t_1 in P. By Proposition 3, the term $|t_1|$ is well-typed in $\lambda\Pi_P$ and, by Proposition 1, $|t_1|, |t_2|, \ldots$ is a reduction sequence of $|t_1|$ in $\lambda\Pi_P$. Hence it is finite.

4.4 Confluence

Proposition 5. *For any functional Pure Type System P, the relation $\longrightarrow$ is confluent in $\lambda\Pi_P$*

Like that of pure λ-calculus, the reduction relation of $\lambda\Pi_P$ is not strongly confluent. Thus, we introduce another reduction relation ($\twoheadrightarrow$) that can reduce, in one step, none to all the $\beta\mathcal{R}$-redices that appears in a term, that is strongly confluent and such that $\twoheadrightarrow^* = \longrightarrow^*$. Then, from the confluence of the relation $\twoheadrightarrow$, we deduce that of the relation $\longrightarrow$. See the long version of the paper for the full proof.

5 Conservativity

Let P be a functional Pure Type System. We could attempt to prove that if the type $\|A\|$ is inhabited in $\lambda\Pi_P$, then A is inhabited in P, and more precisely that if Γ is a context and A a term in P and t a term in $\lambda\Pi_P$, such that $\|\Gamma\| \vdash t : \|A\|$, then there exists a term u of P such that $|u| = t$ and $\Gamma \vdash u : A$. Unfortunately this property does not hold in general as shown by the following counterexamples.

Example 4. If P is the simply-typed lambda-calculus, then the polymorphic identity is not well-typed in P, in particular:

$nat : Type \nvdash ((\lambda X : Type\ \lambda x : X\ x)\ nat) : (nat \Rightarrow nat)$

however, in $\lambda\Pi$, we have

$nat : \|Type\| \vdash ((\lambda X : \|Type\|\ \lambda x : \|X\|\ x)\ |nat|) : \|nat \Rightarrow nat\|$.

Example 5. If $\langle s_1, s_2, s_3\rangle \in R$, $\Sigma_P \vdash \dot{\Pi}_{\langle s_1,s_2,s_3\rangle} : \|\Pi X : s_1\ ((X \Rightarrow s_2) \Rightarrow s_3)\|$ but the term $\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}$ is not the translation of any term of P.

Therefore, we shall prove a slightly weaker property: that if the type $\|A\|$ is inhabited *by a normal term* in $\lambda\Pi_P$, then A is inhabited in P. Notice that this restriction vanishes if $\lambda\Pi_P$ is terminating.

We shall prove, in a first step, that if $\|\Gamma\| \vdash t : \|A\|$, and t is a weak η-long normal term then there exists a term in u such that such that $|u| = t$ and $\Gamma \vdash u : A$. Then we shall get rid of this restriction on weak η-long forms.

Definition 9. *A term t of $\lambda\Pi_P$ is a weak η-long term if and only if each occurrence of $\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}$ in t, is in a subterm of the form $(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ t_1\ t_2)$ (*i.e. *each occurrence of $\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}$ is η-expanded).*

Definition 10 (Back translation). *We suppose that P contains at least one sort: s_0. Then we define a translation from $\lambda\Pi_P$ to P as follows:*

- $x^* = x$, $\quad s^* = s_0 \quad \dot{s}^* = s$, $\quad U_s^* = s$,
- $(\Pi x : A\ B)^* = \Pi x : A^*\ B^*$,
- $(\lambda x : A\ t)^* = \lambda x : A^*\ t^*$,

– $(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ A\ B)^* = \Pi x : A^*\ (B^*\ x)$,
– $(\varepsilon_s\ u)^* = u^*$,
– $(t\ u)^* = t^*\ u^*$ *otherwise.*

Proposition 6. *The back translation* $(.)^*$ *is a right inverse of* $|.|$ *and* $\|.\|$ i.e. *for all* t *such that* $|t|$ *(*resp. $\|t\|$*) is well defined,* $|t|^* = t$ *(*resp. $\|t\|^* = t$*).*

Proof. By induction on the structure of t.

Proposition 7. *For all* t, u *terms and* x *variable of* $\lambda\Pi_P$,

1. $((u/x)t)^* = (u^*/x)t^*$
2. *If* $t \longrightarrow u$ *then* $t^* \longrightarrow_\beta^* u^*$ *in* P.

Proof

1. By induction on t.
2. If $t \longrightarrow_\beta u$ then $t^* \longrightarrow_\beta u^*$, and if $t \longrightarrow_{\mathcal{R}} u$, then $t^* = u^*$.

Proposition 8. *For all terms* A, B *of* P *and* C, D *of* $\lambda\Pi_P$ *(such that* $\|A\|$ *and* $\|B\|$ *are well defined),*

1. *If* $A \equiv_\beta B$, *then* $\|A\| \equiv \|B\|$.
2. *If* $C \equiv D$, *then* $C^* \equiv_\beta D^*$.
3. *If* $\|A\| \equiv \|B\|$, *then* $A \equiv_\beta B$.
4. *If* $C \equiv \|A\|$, *then* $C \equiv \|C^*\|$.

Proof

1. By induction on the length of the path of β-reductions and β-expansions between A and B, and by Proposition 1.
2. By the same reasoning as for the first point, using Proposition 7.
3. By the second point and Proposition 6.
4. By the first and second points and Proposition 6.

Proposition 9 (Conservativity). *If there exists a context* Γ, *a term* A *of* P, *and a term* t, *in weak* η*-long normal form, of* $\lambda\Pi_P$, *such that* $\|\Gamma\| \vdash t : \|A\|$, *Then there exists a term* u *of* P *such that* $|u| \equiv t$ *and* $\Gamma \vdash u : A$.

Proof. By induction on t.

- If t is a well-typed product or sort, then it cannot be typed by a translated type (by confluence of $\lambda\Pi_P$).
- If $t = \lambda x : B\ t'$. The term t is well typed, thus there exists a term C of $\lambda\Pi_P$, such that $\|\Gamma\| \vdash t : \Pi x : B\ C$. Therefore $\|A\| \equiv \Pi x : B\ C$ (α).
  And $\Pi x : B\ C \equiv \|(\Pi x : B\ C)^*\| = \|\Pi x : B^*\ C^*\| \equiv \Pi x : \|B^*\|\ \|C^*\|$
  In particular (by confluence of $\lambda\Pi_P$),

$$B \equiv \|B^*\|, \quad C \equiv \|C^*\| \quad \text{and} \quad \|\Gamma\| \vdash \lambda x : B\ t'\ :\ \Pi x : \|B^*\|\ \|C^*\|$$

Therefore $\|\Gamma\|, x : \|B^*\| \vdash t' : \|C^*\|$. The term $\lambda x : B\ t'$ is in weak η-long normal form, thus t' is also in weak η-long normal form, and, by induction hypothesis, there exists a term u' of P, such that $|u'| \equiv t'$ and
$\Gamma, x : B^* \vdash u' : C^*$. Therefore $\Gamma \vdash \lambda x : B^*\ u' \ :\ \Pi x : B^*\ C^*$ (β). Moreover, $A \equiv_\beta \Pi x : B^*\ C^*$ by (α) and Proposition 8. Thus, by the conversion rule of P, we get $\Gamma \vdash \lambda x : B^*\ u' \ :\ A$.
And $|\lambda x : B^*\ u'| = \lambda x : \|B^*\|\ |u'| \equiv \lambda x : B\ t' = t$.

- If t is an application or a variable, as it is normal, it has the form $x\ t_1\ \dots\ t_n$ for some variable x and terms $t_1, \dots, t_n$. We have $\|\Gamma\| \vdash x\ t_1\ \dots\ t_n : \|A\|$ (α_0).
  - If x is a variable of the context Σ_P,
    - If $x = \dot{s_1}$ (where $\langle s_1, s_2\rangle$ is an axiom of P), then $n = 0$ (because t is well typed) and $\|A\| = U_{s_2}$. We have $\vdash s_1 : s_2$ in P, therefore $\Gamma \vdash s_1 : s_2$.
    - If $x = U_s$ (where s is a sort of P), then $n = 0$ and $\|A\| \equiv Type$. That's an absurdity by confluence of $\lambda\Pi_P$.
    - If $x = \varepsilon_s$ (where s is a sort of P), then, as t is well typed $n \leq 1$.
      - ⋆ If $n = 1$, then $\|\Gamma\| \vdash t_1 : U_s$, and $\|A\| \equiv Type$ (absurdity).
      - ⋆ If $n = 0$, then $\|A\| \equiv U_s \Rightarrow Type$, thus by Propositions 8 and 2, $U_s \Rightarrow Type \equiv \|(U_s \Rightarrow Type)^*\| = \|s \Rightarrow s_0\| \equiv \|s\| \Rightarrow \|s_0\|$. Therefore $Type \equiv \|s_0\|$ (absurdity).
    - If $x = \dot{\Pi}_{\langle s_1, s_2, s_3\rangle}$ (where $\langle s_1, s_2, s_3\rangle$ is a rule of P), then as t is well-typed and in weak η-long form, $n = 2$. We have $\|A\| \equiv U_{s_3}$ thus $A \equiv s_3$ by Proposition 8.
      And $\|\Gamma\| \vdash t_1 : U_{s_1}$ *i.e.* $\|\Gamma\| \vdash t_1 : \|s_1\|$.
      And $\|\Gamma\|, t_1 : U_{s_1} \vdash t_2 : ((\varepsilon_{s_1} t_1) \Rightarrow U_{s_2})$ (α_1)
      t_1 is also in weak η-long normal form, then, by induction hypothesis, there exists a term u_1 of P such that:
      $$|u_1| \equiv t_1 \quad and \quad \Gamma \vdash u_1 : s_1 \ \ (\beta_1)$$
      Then, by (α_1), $\|\Gamma\|, t_1 : \|s_1\| \vdash t_2 : \|u_1 \Rightarrow s_2\|$.
      In particular, $\|\Gamma\|, t_1 : \|s_1\| \vdash t_2 : \|u_1\| \Rightarrow \|s_2\|$.
      However t_2 is also in weak η-long normal form, then there exists a term t'_2 (in weak η-long normal form) of $\lambda\Pi_P$ such that
      $$t_2 = \lambda x : \|u_1\|\ t'_2 \quad and \quad \|\Gamma\|, x : \|u_1\| \vdash t'_2 : \|s_2\|$$
      By induction hypothesis, there exists a term u'_2 of P, such that
      $$|u'_2| \equiv t'_2 \quad and \quad \Gamma, x : u_1 \vdash u'_2 : s_2 \ \ (\beta_2)$$
      Then we choose $u = \Pi x : u_1\ u'_2$ that verifies $\Gamma \vdash u : s_3$, by (β_1), (β_2), and the fact that $\langle s_1, s_2, s_3\rangle$ is a rule of P. And, finally,
      $|u| = \dot{\Pi}_{\langle s_1, s_2, s_3\rangle} |u_1|\ (\lambda x : (\varepsilon_{s_1} |u_1|)\ |u'_2| \equiv \dot{\Pi}_{\langle s_1, s_2, s_3\rangle} t_1\ t_2 = t$
  - If x is a variable of the context Γ,
    For $k \in \{0, .., n\}$, let (H_k) be the statement: *"The term $x\ t_1\ \dots\ t_k$ is typable in $\|\Gamma\|$ and its type is in the image of $\|.\|$".*
    We first prove (H_0),. . . ,(H_n) by induction.
    - ⋆ $k = 0$: x is a variable of the context Γ, then there exists a well typed term or a sort T in P such that Γ contains $x : T$. Therefore $\|\Gamma\|$ contains $x : \|T\|$.

- ⋆ $0 \leq k \leq n-1$: We suppose (H_k).
  $x\ t_1\ \ldots\ t_{k+1}$ is well typed in Γ, then there exists terms D and E of $\lambda\Pi_P$ such that $\|\Gamma\| \vdash t_{k+1} : D\ \ (\delta_1)$,
  $\|\Gamma\| \vdash x\, t_1\ \ldots\ t_k : \Pi y : D\ E\ \ (\delta_2)$, and $\|\Gamma\| \vdash x\, t_1\ \ldots\ t_{k+1} : E\ \ (\delta_3)$.
  However, by (H_k), we can type $x\ t_1\ \ldots\ t_k$ by a translated type in $\|\Gamma\|$, then by (δ_2) and Proposition 8, $\Pi y : D\ E \equiv \Pi y : \|D^*\|\ \|E^*\|$.
  In particular, $E \equiv \|E^*\|\ \ (\eta_1)$,
  We conclude, by (δ_3), (η_1) and the conversion rule of $\lambda\Pi_P$.

Then, if $n = 0$, we take $u = x$ and Γ contains $x : T$ with $\|T\| \equiv \|A\|$.
And, if $n > 0$, then, by (α_0), there exists terms B and C of $\lambda\Pi_P$ such that $\|\Gamma\| \vdash t_n : B\ \ (\theta_1)$ and $\|\Gamma\| \vdash x\ t_1\ \ldots\ t_{n-1} : \Pi y : B\ C\ \ (\theta_2)$ with $\|A\| \equiv (t_n/y)C\ \ (\theta_3)$. Then, by (H_{n-1}), (θ_2), and Proposition 8.4, $\Pi y : B\ C \equiv \Pi y : \|B^*\|\ \|C^*\|$, therefore $B \equiv \|B^*\|$ and $C \equiv \|C^*\|$.
Thus, $\|\Gamma\| \vdash t_n : \|B^*\|$ and $\|\Gamma\| \vdash x\ t_1\ \ldots\ t_{n-1} : \|\Pi y : B^*\ C^*\|$.
t_n and $x\ t_1\ \ldots\ t_{n-1}$ are both in weak η-long normal form, then, by induction hypothesis, there exists terms w_1 and w_2 of P such that:

$$|w_1| \equiv x\ t_1\ \ldots t_{n-1} \text{ and } \Gamma \vdash w_1 : \Pi y : B^*\ C^*$$
$$|w_2| \equiv t_n \text{ and } \Gamma \vdash w_2 : B^*$$

Let $u = w_1\ w_2$, we have:

$$|u| = |w_1|\ |w_2| \equiv x\ t_1\ \ldots\ t_{n-1}\ t_n \text{ and } \Gamma \vdash u : (w_2/y)C^*.$$

However, by (θ_3) and Proposition 8, we have:
$A \equiv (t_n^*/y)C^* \equiv (w_2/y)C^*$, and, finally, $\Gamma \vdash u : A$.

Finally, we get rid of the weak η-long form restriction with the following Propositions.

Proposition 10. *For all terms A, B of $\lambda\Pi_P$, and for all well typed term or sort C of P,*

1. *If $A \longrightarrow B$ then $A" \longrightarrow^* B"$*
2. *If $A \equiv B$ then $A" \equiv B"$*
3. *If A is in weak η-long form, then $A" \longrightarrow_\beta^* A$, in particular $A" \equiv A$*
4. *$\|C\|" \equiv \|C\|$*
5. *If $A \equiv \|C\|$ then $A" \equiv A$*

Proof

1. If $A \longrightarrow_\beta B$, then $A" \longrightarrow_\beta B"$ (by induction on A).
   If $A \longrightarrow_{\mathcal{R}} B$,
   - for all axiom $\langle s_1, s_2\rangle$, $(\varepsilon_{s_2}\ (\dot{s_1}))" = \varepsilon_{s_2}\ (\dot{s_1}) \longrightarrow_{\mathcal{R}} U_{s_1} = (U_{s_1})"$.
   - for all rule $\langle s_1, s_2, s_3\rangle$, $(\varepsilon_{s_3}(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle} C\ D))" =$
     $\varepsilon_{s_3}((\lambda x : U_{s_1} \lambda y : ((\varepsilon_{s_1}\ x) \Rightarrow U_{s_2})\ (\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ x\ y))C"\ D")$
     $\longrightarrow_\beta^2 \varepsilon_{s_3}(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle} C"\ D") \longrightarrow_{\mathcal{R}} \Pi x : (\varepsilon_{s_1}\ C")\ (\varepsilon_{s_2}\ (D"\ x))$
     $= \Pi x : (\varepsilon_{s_1}\ C")\ (\varepsilon_{s_2}\ (D\ x)")$
2. By induction on the number of derivations and expansions from A to B.
3. By induction on A, remarking that $(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ t_1\ t_2)" \longrightarrow_\beta^2 \dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ t_1"\ t_2"$.
4. A translated term $\|C\|$ is in weak η-long form.
5. If $A \equiv \|C\|$ then $A" \equiv \|C\|" \equiv \|C\|$, by the the second and fourth points.

Proposition 11. *Let t be a normal term of $\lambda\Pi_P$,*

$$\text{if } \|\Gamma\| \vdash t : \|A\| \text{ then } \|\Gamma\| \vdash t" : \|A\|$$

Proof. By induction on t.

- If t is a well-typed product or sort, then it cannot be typed by a translated type (by confluence of $\lambda\Pi_P$).
- If $t = \lambda x : B\ u$, then there exists a term C of $\lambda\Pi_P$, such that $\|A\| \equiv \Pi x : B\ C\ (\alpha)$, with $\Gamma, x : B \vdash u : C$.
  By (α), we have $B \equiv \|B^*\|\ (\beta)$ and $C \equiv \|C^*\|$. Thus $\Gamma, x : \|B^*\| \vdash u : \|C^*\|$. Then, by induction hypothesis, we have $\Gamma, x : \|B^*\| \vdash u" : \|C^*\|$, therefore $\Gamma \vdash \lambda x : \|B^*\|\ u" : \Pi x : \|B^*\|\ \|C^*\| \equiv \|A\|$ thus $\Gamma \vdash \lambda x : B\ u" : \|A\|$, by (β). Finally, by (β) and the Proposition 10.5, $\lambda x : B\ u" \equiv \lambda x : B"\ u"$, therefore, by subject reduction, $\Gamma \vdash t" = \lambda x : B"\ u" : \|A\|$
- If t is an application or a variable, as it is normal, it has the form $x\ t_1\ \ldots\ t_n$ for some variable x and terms $t_1, \ldots, t_n$. We have $\|\Gamma\| \vdash x\ t_1\ \ldots\ t_n : \|A\|\quad (\alpha_0)$.
  - ⊸ If x is a variable of the context Σ_P,
    - ∗ If $x = \dot{s_1}$ (where $\langle s_1, s_2\rangle$ is an axiom of P), then $n = 0$ (because t is well typed) and we have $(\dot{s_1})" = \dot{s_1}$.
    - ∗ If $x = U_s$ (where s is a sort of P), then $n = 0$ and $\|A\| \equiv Type$. That's an absurdity by confluence of $\lambda\Pi_P$.
    - ∗ If $x = \varepsilon_s$ (where s is a sort of P), then, as t is well typed $n \leq 1$.
      - ⋆ If $n = 1$, then $\|\Gamma\| \vdash t_1 : U_s$, and $\|A\| \equiv Type$ (absurdity).
      - ⋆ If $n = 0$, we have $(\varepsilon_s)" = \varepsilon_s$
    - ∗ If $x = \dot{\Pi}_{\langle s_1,s_2,s_3\rangle}$ (where $\langle s_1, s_2, s_3\rangle$ is a rule of P), then as t is well-typed, $n \leq 2$. Moreover, $\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}$, $(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ t_1)$, and $(\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ t_1\ t_2)$ have the same types than their weak η-long forms.
  - ⊸ If x is a variable of the context Γ,
    - ∗ If $n = 0$, we have $x" = x$.
    - ∗ If $n > 0$, then there exists terms B and C of $\lambda\Pi_P$ such that $\|\Gamma\| \vdash t_n : B\quad (\alpha_1)$ and $\|\Gamma\| \vdash x\ t_1\ \ldots\ t_{n-1} : \Pi y : B\ C\quad (\alpha_2)$ with $\|A\| \equiv (t_n/y)C\quad (\alpha_3)$. As in the proof of Proposition 9, we can type $x\ t_1\ \ldots\ t_{n-1}$ by a translated type, then $\Pi y : B\ C \equiv \Pi y : \|B^*\|\ \|C^*\|$. In particular, $B \equiv \|B^*\|$ and $C \equiv \|C^*\|$.
      Thus, $\|\Gamma\| \vdash t_n : \|B^*\|$ and $\|\Gamma\| \vdash x\ t_1\ \ldots\ t_{n-1} : \|\Pi y : B^*\ C^*\|$.
      By induction hypothesis, we have $\|\Gamma\| \vdash t_n" : \|B^*\|$ and $\|\Gamma\| \vdash x\ t_1"\ \ldots\ t_{n-1}" : \Pi y : \|B^*\|\ \|C^*\|$. Finally, by (α_3) and Proposition 10.5, $\|\Gamma\| \vdash t" = x\ t_1"\ \ldots\ t_n" : (t_n"/y)C \equiv \|A\|$.

Theorem 1. *Let P be a functional Pure Type System, such that $\lambda\Pi_P$ is terminating. The type $\|A\|$ is inhabited by a closed term in $\lambda\Pi_P$ if and only if the type A is inhabited by a closed term in P.*

Proof. If A has a closed inhabitant in P, then by Proposition 3, $\|A\|$ has a closed inhabitant in $\lambda\Pi_P$. Conversely, if $\|A\|$ has a closed inhabitant then, by termination of $\lambda\Pi_P$ and Proposition 11, it has a closed inhabitant in weak η-long normal form and by Proposition 9, A has a closed inhabitant in P.

Remark 1. This conservativity property we have proved is similar to that of the Curry-de Bruijn-Howard correspondence. If the type A° is inhabited in $\lambda\Pi$-calculus, then the proposition A is provable in minimal predicate logic, but not all terms of type A° correspond to proofs of A. For instance, if A is the proposition $(\forall x\ P(x)) \Rightarrow P(c)$, then the normal term $\lambda\alpha : (\Pi x : \iota\ (P\ x))\ (\alpha\ c)$ corresponds to a proof of A but the term $\lambda\alpha : (\Pi x : \iota\ (P\ x))\ (\alpha\ ((\lambda y : \iota\ y)\ c))$ does not.

Remark 2. There are two ways to express proofs of simple type theory in the $\lambda\Pi$-calculus modulo. We can either use directly the fact that simple type theory can be expressed in Deduction modulo [8] or first express the proofs of simple type theory in the Calculus of Constructions and then embed the Calculus of Constructions in the $\lambda\Pi$-calculus modulo.

These two solutions have some similarities, in particular if we write o the symbol U_{Type}. But they have also some differences: the function $\lambda x\ x$ of simple type theory is translated as the symbol I — or as the term $\lambda 1$ — in the first case, using a symbol I — or the symbols λ and 1 — specially introduced in the context to express this particular theory, while it is expressed as $\lambda x\ x$ using the symbol λ of the $\lambda\Pi$-calculus modulo in the second.

More generally in the second case, we exploit the similarities of the $\lambda\Pi$-calculus modulo and simple type theory — the fact that they both allow to express functions — to simplify the expression while the first method is completely generic and uses no particularity of simple type theory. This explains why this first expression requires only the $\lambda\Pi^-$-calculus modulo, while the second requires the conversion rule to contain β-conversion.

References

1. Barendregt, H.: Lambda calculi with types. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, pp. 117–309. Oxford University Press, Oxford (1992)
2. Berardi, S.: Towards a mathematical analysis of the Coquand-Huet Calculus of Constructions and the other systems in Barendregt's cube, manuscript (1988)
3. Blanqui, F.: Definitions by rewriting in the Calculus of Constructions. Mathematical Structures in Computer Science 15(1), 37–92 (2005)
4. Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation 76, 95–120 (1988)
5. Cousineau, D.: Un plongement conservatif des Pure Type Systems dans le lambda Pi modulo, Master Parisien de Recherche en Informatique (2006)
6. Dougherty, D., Ghilezan, S., Lescanne, P., Likavec, S.: Strong normalization of the dual classical sequent calculus, LPAR-2005 (2005)
7. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. Journal of Automated Reasoning 31, 33–72 (2003)
8. Dowek, G., Hardin, T., Kirchner, C.: HOL-lambda-sigma: an intentional first-order expression of higher-order logic. Mathematical Structures in Computer Science 11, 1–25 (2001)
9. Dowek, G., Werner, B.: Proof normalization modulo. The. Journal of Symbolic Logic 68(4), 1289–1316 (2003)

10. Girard, J.Y.: Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur, Thèse de Doctorat, Université Paris VII (1972)
11. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)
12. Martin-Löf, P.: Intuitionistic Type Theory, Bibliopolis (1984)
13. Nordström, B., Petersson, K., Smith, J.M.: Martin-Löf's type theory. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, pp. 1–37. Clarendon Press, Oxford (2000)
14. Palmgren, E.: On universes in type theory. In: Twenty five years of constructive type theory. Oxford Logic Guides, vol. 36, pp. 191–204. Oxford University Press, New York (1998)
15. Terlouw, J.: Een nadere bewijstheoretische analyse van GSTT's, manuscript (1989)

Completing Herbelin's Programme

José Espírito Santo

Departamento de Matemática, Universidade do Minho, Portugal
jes@math.uminho.pt

Abstract. In 1994 Herbelin started and partially achieved the programme of showing that, for intuitionistic implicational logic, there is a Curry-Howard interpretation of sequent calculus into a variant of the λ-calculus, specifically a variant which manipulates formally "applicative contexts" and inverts the associativity of "applicative terms". Herbelin worked with a fragment of sequent calculus with constraints on left introduction. In this paper we complete Herbelin's programme for full sequent calculus, that is, sequent calculus without the mentioned constraints, but where permutative conversions necessarily show up. This requires the introduction of a lambda-like calculus for full sequent calculus and an extension of natural deduction that gives meaning to "applicative contexts" and "applicative terms". Such extension is a calculus with *modus ponens* and primitive substitution that refines von Plato's natural deduction; it is also a "coercion calculus", in the sense of Cervesato and Pfenning. The proof-theoretical outcome is noteworthy: the puzzling relationship between cut and substitution is settled; and cut-elimination in sequent calculus is proven isomorphic to normalisation in the proposed natural deduction system. The isomorphism is the mapping that inverts the associativity of applicative terms.

1 Introduction

Herbelin's CSL'94 paper [11] is an integrated contribution into two closely related subjects: structural proof theory and the study of the computational interpretation of sequent calculus. Here, structural proof theory is taken in the restricted sense of the study of the relationship between natural deduction and sequent calculus, the two kinds of proof systems introduced since the subject was born [10]. Such relationship is a puzzle that constantly attracted attention during the last 70 years [10,18,22,17,14,20]. The study of the computational interpretation of sequent calculus, with the purpose of extending the Curry-Howard correspondence, is a relatively recent topic, with the first explicit contributions starting in the early 1990's [9,21,13]. An integrated contribution to the two subjects is desirable: one should understand the differences and similarities between natural deduction and sequent calculus, if one wants to extend the Curry-Howard correspondence; and a way of expressing those differences and similarities is, precisely, via the corresponding computational interpretations.

Herbelin's paper initiates the programme of defining a λ-calculus (with a strongly normalising set of reduction rules) such that, by means of the calculus,

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 118–132, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the following two goals are achieved simultaneously: (1) to give a convincing computational interpretation of (a fragment of) sequent calculus, along the lines of the Curry-Howard correspondence; and (2) to express the difference between sequent calculus and natural deduction, reducing it to the mere inversion of the associativity of applicative terms.

Herbelin studied a fragment LJT of sequent calculus LJ and gave its computational interpretation in terms of the so-called $\overline{\lambda}$-calculus. Contrary to earlier contributions, whose focus was on the feature of pattern matching, in $\overline{\lambda}$ the novelty is the existence of an auxiliary syntactic class of *applicative contexts*. In the case of intuitionistic implication, an applicative context is simply a list of terms, understood as a "multiary" argument for functional application. Hence, "applicative terms" in $\overline{\lambda}$ have the form $t[u_1, ..., u_m]$. Herbelin concludes that the difference between sequent calculus and natural deduction resides in the organization of applicative terms: sequent calculus is right-associative $t(u_1 :: ...(u_n :: []))$, whereas natural deduction is left-associative $(...(MN_1)...N_m)$.

Herbelin's paper achieved (1) for LJT and has the merit of suggesting that (2) can be achieved. Verification of (2) happened in later papers. The mapping that inverts the associativity of applicative terms is proved in [3] to be a bijection between normal λ-terms and cut-free $\overline{\lambda}$-terms, in [5] to be an isomorphism between the λ-calculus and a fragment of $\overline{\lambda}$, and in [6] to be an isomorphism between an extension of the λ-calculus and a larger fragment of $\overline{\lambda}$. Fulfillment of (2) is useful for (1), because only an isomorphic natural deduction system gives rigorous meaning to "applicative context" and "applicative term".

Notwithstanding the parts of Herbelin's programme already completed (including the extension of (1) to classical logic in [2]), a lot remains unfinished. LJT is a permutation-free fragment, where only a restricted form of left introduction is available and where the computational meaning of permutation (so typical of sequent calculus) is absent. In addition, the fulfilment of (2), in connection with larger fragments of sequent calculus, requires the extension of the natural deduction system. One idea for this extension is in [6], and turns out to be the idea of defining natural deduction as a "coercion calculus", in the sense of Cervesato and Pfenning [1]. Another idea is that of generalised elimination rules, due to von Plato [20].

In the setting of intuitionistic implicational logic, we contribute to the completion of Herbelin's programme for full sequent calculus, that is, sequent calculus without constraints on left introductions (but where permutative conversions necessarily show up). The computational interpretation is in terms of a λ-calculus λ^{Gtz} with a primitive notion of applicative context, taken in a natural, generalised sense. In order to fulfil (2), a system of natural deduction λ_{Nat} is defined that extends and refines von Plato's natural deduction. It is a calculus with *modus ponens* and primitive substitution and it is also a coercion calculus. Then we prove that $\lambda^{\mathsf{Gtz}} \cong \lambda_{\mathsf{Nat}}$ in the fullest sense: the mapping that inverts the associativity of applicative terms is a sound bijection between the sets of terms of the two calculi and, in addition, establishes an isomorphism between cut-elimination in λ^{Gtz} and normalisation in λ_{Nat}. Strong cut-elimination for λ^{Gtz} is proved via

an interpretation into the calculus of "delayed substitutions" $\lambda\mathtt{s}$ of [7]; strong normalisation for λ_{Nat} follows by isomorphism. These results constitute, for the logic under analysis here, considerable improvements over [11,1,20,6].

The paper is organized as follows. Section 2 presents λ^{Gtz}. Section 3 presents λ_{Nat}. Sections 4 and 5 prove and analyze $\lambda^{\mathsf{Gtz}} \cong \lambda_{\mathsf{Nat}}$. Section 6 concludes.

Notations: Types (=formulas) are ranged over by A, B, C and generated from type variables using the "arrow type" (=implication), written $A \supset B$. Contexts Γ are consistent sets of declarations $x : A$. "Consistent" means that for each variable x there is at most one declaration in Γ. The notation $\Gamma, x : A$ always produces a consistent set. Meta-substitution is denoted with square brackets $[_/x]_$. All calculi in this paper assume Barendregt's variable convention (in particular we take renaming of bound variables for granted).

Naming of Systems: Sequent calculi are denoted λ^S (where S is some tag); natural deduction systems introduced here are denoted λ_S; more or less traditional systems of natural deduction are denoted λS.

2 Sequent Calculus

The sequent calculus we introduce is named λ^{Gtz} (read "λ-Gentzen").

Expressions and Typing Rules: There are two sorts of expressions in λ^{Gtz}: terms t, u, v and contexts k.

$$\begin{array}{ll} \text{(Terms)} & t, u, v ::= x \mid \lambda x.t \mid tk \\ \text{(Contexts)} & k ::= (x)v \mid u :: k \end{array}$$

Terms are either variables x, y, z, abstractions $\lambda x.t$ or *cuts* tk. Contexts are either a *selection* $(x)v$ or a *linear left introduction* $u :: k$, often called a *cons*. x is bound in $(x)v$. [1] A computational reading of contexts is as a prescription of what to do next (with some expression that has to be plugged in). A selection $(x)v$ says "substitute for x in v" and a cons $u :: k$ says "apply to u and proceed according to k". A cut tk is a plugging of a term t in the context k. We will use the following abbreviations: $[] = (x)x$, $[u_1, ..., u_n] = u_1 :: ...u_n :: []$, and $\langle u/x \rangle t = u(x)t$.

The typing rules of λ^{Gtz} are as follows:

$$\frac{}{\Gamma, x : A \vdash x : A}\ Axiom$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B}\ Right \qquad \frac{\Gamma \vdash u : A \quad \Gamma; B \vdash k : C}{\Gamma; A \supset B \vdash u :: k : C}\ Left$$

$$\frac{\Gamma \vdash t : A \quad \Gamma; A \vdash k : B}{\Gamma \vdash tk : B}\ Cut \qquad \frac{\Gamma, x : A \vdash v : B}{\Gamma; A \vdash (x)v : B}\ Selection$$

[1] In order to save parentheses, the scope of binders extends to the right as far as possible.

There are two sorts of sequents in λ^{Gtz}, namely $\Gamma \vdash t : A$ and $\Gamma; A \vdash k : B$. The distinguished position in the antecedent of sequents of the latter kind contains the *selected* formula. There is a typing rule $Selection$ that selects an antecedent formula. Besides this rule, there are the axiom rule, the introductions on the left(=antecedent) and on the right(=succedent) of sequents, and the cut.

The typing rules follow a reasonable discipline: active formulas in the antecedent of sequents have to be previously selected (the B in $Left$ and one A in Cut); and a formula introduced on the left is selected. The latter constraint implies that a left introduction $u :: k$ is a *linear* introduction, because there cannot be an implicit contraction. Full left introduction is recovered as a cut between an axiom and a linear left introduction, corresponding to $x(u :: k)$. The cut-elimination process will not touch these trivial cuts. More generally, given a context k, xk represents the inverse of a selection, that is, the operation that takes a formula out of the selection position and gives it name x. An implicit contraction may happen here.

Reduction Rules: The reduction rules of λ^{Gtz} are as follows:

$$\begin{array}{llll} (\beta) & (\lambda x.t)(u :: k) \to \langle u/x \rangle (tk) & (\sigma) & \langle t/x \rangle v \to [t/x]v \\ (\pi) & (tk)k' \to t(k@k') & (\mu) & (x)xk \to k, \text{ if } x \notin k \end{array}$$

where

$$(u :: k)@k' = u :: (k@k') \qquad ((x)v)@k' = (x)vk'$$

By *cut-elimination* we mean $\beta\pi\sigma$-reduction. Rules β, π and σ aim at eliminating all cuts that are not of the form $x(u :: k)$. The procedure is standard. If a cut is a key-cut (both cut-formulas main(=introduced) in the premisses) with cut-formula $A \supset B$, the cut is reduced to two cuts, with cut-formulas A and B. This is rule β. If a cut, not of the form $x(u :: k)$, is not a key cut, this means that it can be permuted to the right (rule σ) or to the left (rule π). The particular case of σ when $v = x$ is named ϵ and reads $\langle t/x \rangle x \to t$ or $t[] \to t$. A term t is a $\beta\pi\sigma$-normal form iff it is generated by the following grammar:

$$\begin{array}{c} t, u, v ::= x \,|\, \lambda x.t \,|\, x(u :: k) \\ k ::= (x)v \,|\, u :: k \end{array} \tag{1}$$

There is a further reduction rule, named μ, of a different nature. It undoes the sequence of inferences consisting of un-selecting and selecting the same formula, if no implicit contraction is involved. A similar rule has been defined for Parigot's $\lambda\mu$-calculus [16], but acting on the RHS of sequents.

Consider the term $(\lambda x.t)(u :: k)$. After a β-step, we get $v = \langle u/x \rangle (tk)$. If u is a cut $t'k'$, v is both a σ- and a π-redex. In this case, there is a choice as to how to continue evaluation. Opting for σ gives $([u/x]t)k$, whereas the π option gives $t'(k'@(x)tk)$. According to [2], this choice is a choice between a call-by-name and a call-by-value strategy of evaluation.

Strong Normalisation: We give a proof of strong normalisation for λ^{Gtz} by defining a reduction-preserving interpretation in the $\lambda\mathsf{s}$-calculus of [7].

The terms of $\lambda\mathsf{s}$ are given by:

$$M, N, P ::= x \mid \lambda x.M \mid MN \mid \langle N/x \rangle M$$

This set of terms is equipped with the following reduction rules:

$$\begin{array}{llll} (\beta) & (\lambda x.M)N \to \langle N/x \rangle M & (\pi_1) & (\langle P/x \rangle M)N \to \langle P/x \rangle (MN) \\ (\sigma) & \langle N/x \rangle M \to [N/x]M & (\pi_2) & \langle \langle P/y \rangle N/x \rangle M \to \langle P/y \rangle \langle N/x \rangle M \end{array}$$

where meta-substitution $[N/x]M$ is defined as expected. In particular

$$[N/x]\langle P/y \rangle M = \langle [N/x]P/y \rangle [N/x]M \ .$$

Let $\pi = \pi_1 \cup \pi_2$. We now define a mapping $(_)^* : \lambda^{\mathsf{Gtz}} \to \lambda\mathsf{s}$. More precisely, mappings $(_)^* : \lambda^{\mathsf{Gtz}} - Terms \to \lambda\mathsf{s} - Terms$ and $(_,_)^* : \lambda\mathsf{s} - Terms \times \lambda^{\mathsf{Gtz}} - Contexts \to \lambda\mathsf{s} - Terms$ are defined by simultaneous recursion as follows:

$$\begin{array}{rlrl} x^* &= x & (M,(x)v)^* &= \langle M/x \rangle v^* \\ (\lambda x.t)^* &= \lambda x.t^* & (M, u :: k)^* &= (Mu^*, k)^* \\ (tk)^* &= (t^*, k)^* & & \end{array}$$

The idea is that, if t, u_i and v are mapped by $(_)^*$ to M, N_i and P, respectively, then $t(u_1 :: \cdots u_m :: (x)v)$ is mapped to $\langle MN_1 \cdots N_m/x \rangle P$.

Proposition 1. *Let $R \in \{\beta, \pi, \sigma, \mu\}$. If $t \to_R u$ in λ^{Gtz}, then $t^* \to^+_{\beta\pi\sigma} u^*$ in $\lambda\mathsf{s}$.*

Proof: Follows from the following four facts: (i) $(\langle N/x \rangle M, k)^* \to^+_\pi \langle N/x \rangle (M,k)^*$; (ii) $((M,k)^*, k')^* \to^+_\pi (M, append(k,k'))^*$; (iii) $([t/x]u)^* = [t^*/x]u^*$; and (iv) $\langle M/x \rangle (N,k)^* \to_\sigma ([M/x]N, k)^*$, if $x \notin k$. ∎

Theorem 1 (Strong cut-elim.). *Every typable $t \in \lambda^{\mathsf{Gtz}}$ is $\beta\pi\sigma\mu$-SN.*

Proof: [7] proves that every typable $t \in \lambda\mathsf{s}$ is $\beta\pi\sigma$-SN (if we use for $\lambda\mathsf{s}$ the obvious typing rules). The theorem follows from this fact, the previous proposition and the fact that $(_)^*$ preserves typability. ∎

Related Systems: We can easily embed LJ in λ^{Gtz}, if we define LJ as the typing system for some obvious term language. The embedding is given by:

$$\begin{array}{rlrl} \mathsf{Axiom}(x) &\rightsquigarrow x & \mathsf{Left}(x, L_1, (y)L_2) &\rightsquigarrow x(u_1 :: (y)u_2) \\ \mathsf{Right}((x)L) &\rightsquigarrow \lambda x.t & \mathsf{Cut}(L_1, (x)L_2) &\rightsquigarrow t_1(x)t_2 \end{array}$$

The cut-free LJ terms correspond to the sub-class of terms in (1) such that k in $x(u :: k)$ has to be a selection $(y)v$. These correspond also to von Plato's "fully normal" natural deductions. $\beta\pi\sigma$-normal forms correspond exactly to Schwichtenberg's *multiary* cut-free terms [19]. We refer to these as *Schwichtenberg nfs*.

A context $u_1 :: ... :: u_m :: (x)x$ ($m \geq 0$) is called an *applicative* context, and may be regarded as a list $[u_1, ..., u_m]$, if we think of $(x)x$ as []. If every context in a term t is applicative, t is a $\overline{\lambda}$-term. A term t is $\beta\pi\sigma$-normal and only contains applicative contexts iff t is a cut-free $\overline{\lambda}$-term, in the sense of [11]. We refer to

such terms as *Herbelin nfs*. They are given by $t, u ::= x \,|\, \lambda x.t \,|\, x(u :: k)$ and $k ::= [] \,|\, u :: k$. Another characterisation of this set is as the set of Schwichtenberg's terms (1) normal w.r.t. certain permutative conversions [19].

Every cut in λ^{Gtz} is of the form $t(u_1 :: ... :: u_m :: (x)v)$, with $m \geq 0$. Several interesting fragments of λ^{Gtz} may be obtained by placing restrictions on m. There is a $m \geq 1$-fragment, which gives a version of the system λJ^m studied in [8]. There is a $m \leq 1$-fragment, which gives a version λ^{gs} of the λg-calculus with explicit substitution λgs, to be defined in the next section. The $m \leq 1$-terms are the terms normal w.r.t. the following *permutation rule*

$$(\nu) \qquad t(u :: v :: k) \rightarrow t(u :: (z)z(v :: k)) \ ,$$

with $z \notin v, k$. Notice that $\rightarrow_\nu \subseteq \rightarrow_\mu^{-1}$. Clearly, ν is terminating and locally confluent. The ν-nf of t is written $\nu(t)$.

3 Natural Deduction

The natural deduction system we introduce is named λ_{Nat} (read "λ-natural"). It is an improvement of natural deduction with general elimination rules.

Natural Deduction with General Elimination Rules: This system [20] may be presented as a type system for the λ-calculus with generalized application. The latter is the system ΛJ of [12], which we rename here as λg, for the sake of uniformity with the names of other calculi. Terms of λg are given by $M, N, P ::= x \,|\, \lambda x.M \,|\, M(N, x.P)$. The typing rule for generalized application is

$$\frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A \quad \Gamma, x : B \vdash P : C}{\Gamma \vdash M(N, x.P) : C} \; gElim$$

The λg-calculus has two reduction rules:

$$\begin{array}{ll} (\beta) & (\lambda x.M)(N, y.P) \rightarrow [[N/x]M/y]P \\ (\pi) & M(N, x.P)(N', y.P') \rightarrow M(N, x.P(N', y.P')) \ . \end{array}$$

Rule π corresponds to the permutative conversion allowed by general eliminations. The $\beta\pi$-normal terms are given by $M, N, P ::= x \,|\, \lambda x.M \,|\, x(N, y.P)$ and correspond to von Plato's "fully normal" natural deductions. A $\beta\pi$-normal form M is called a *Mints normal form* if, for every application $x(N, y.P)$ in M, P is y-normal [4]. P is *y-normal* if $P = y$ or $P = y(N', y'.P')$ and $y \notin N', P'$ and P' is y'-normal. Another characterisation of Mints nfs is as $\beta\pi$-normal forms which are, in addition, normal w.r.t. a set of permutation rules given in [4].

The λgs-calculus is the following version of λg with *explicit* substitution. A new term constructor, explicit substitution $\langle N/x \rangle M$, is added. In rule β

$$(\beta) \qquad (\lambda x.M)(N, y.P) \rightarrow \langle N/x \rangle \langle M/y \rangle P \ ,$$

two explicit substitutions are generated, instead of two calls to the meta-substitution. π stays the same. Finally, the calculus contains a new reduction rule, named

σ, and defined by $\langle N/x\rangle M \to [N/x]M$. A λgs-term is in $\beta\pi\sigma$-normal form iff it is a λg-term in $\beta\pi$-normal form.[2]

The usual λ-calculus embeds in λg by setting $MN = M(N, x.x)$. Likewise, *modus ponens* (=Gentzen's elimination rule for implication) may be seen as the particular case of the $gElim$ where $B = C$ and the rightmost premiss is omitted. The set of β-normal λ-terms is in bijective correspondence with the set of Mints normal forms [14,4].

Motivation for λ_{Nat}: If one sees generalised application $M(N, x.P)$ as a substitution $subst(MN, x.P)$ (the notation here is not important), then one can say that in λg every ordinary application MN occurs as the actual parameter of a substitution. This situation has a defect: it is cumbersome to write iterated, ordinary applications. For instance, MNN' is written $subst(subst(MN, x.x)N', y.y)$, with x, y fresh. A solution is to allow $m \geq 0$ application as actual parameters of substitutions: $subst(MN_1...N_m, x.P)$. The particular case $m = 0$ encompasses explicit substitution. The usefulness of allowing $m > 1$ is precisely in having the alternative way $subst(MNN', x.x)$ of writing MNN'.

Expressions and Typing Rules: There are two syntactic classes in λ_{Nat}: terms M, N, P and *elimination expressions* E.

(Terms)	$M, N, P ::= x \mid \lambda x.M \mid \{E/x\}P$
(Elimination-Expressions)	$E ::= hd(M) \mid EN$

Terms are either variables x, y, z, abstractions $\lambda x.M$ or *(primitive) substitutions* $\{E/x\}P$. Elimination expressions (EEs, for short) are either *coercions* $hd(M)$ (a.k.a. *heads*) or eliminations EN. So an EE is a sequence of zero or more eliminations starting from a coerced term and ending as the *actual parameter* of a substitution. Hence, every substitution has the form $\{hd(M)N_1...N_m/x\}P$, with $m \geq 0$. Generalised elimination is recovered as $\{hd(M)N/x\}P$, that is the particular case $m = 1$. Ordinary elimination is $\{hd(M)N/x\}x$. We will use the following abbreviations: $ap(E) = \{E/z\}z$, $MN = ap(hd(M)N)$ and $\langle N/x\rangle M = \{hd(N)/x\}M$.

The typing rules of λ_{Nat} are as follows:

$$\frac{}{\Gamma, x : A \vdash x : A}\ Assumption$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \supset B}\ Intro \qquad \frac{\Gamma \rhd E : A \supset B \quad \Gamma \vdash N : A}{\Gamma \rhd EN : B}\ Elim$$

$$\frac{\Gamma \rhd E : A \quad \Gamma, x : A \vdash P : B}{\Gamma \vdash \{E/x\}P : B}\ Subst \qquad \frac{\Gamma \vdash M : A}{\Gamma \rhd hd(M) : A}\ Coercion$$

There are two sorts of sequents in λ_{Nat}, namely $\Gamma \vdash M : A$ and $\Gamma \rhd E : A$. The typing system contains an assumption rule, an introduction rule, an elimination

[2] For a slightly different definition of λgs see [7].

rule and a rule for typing substitution. These are standard, except for the use of two sorts of sequents. The coercion rule changes the kind of sequent. The displayed formula of the coercion rule is the *coercion formula*. The construction $ap(E)$ (=$\{E/x\}x$) represents the inverse of the coercion rule.

Reduction Rules: The reduction rules of λ_{Nat} will act on the head of substitutions $\{hd(M)N_1...N_m/x\}P$. In order to have access to such heads, it is convenient to introduce the following syntactic expressions:

$$\mathcal{C} ::= \{[]/x\}P \,|\, N \cdot \mathcal{C}$$

These expressions are called *meta-contexts* of λ_{Nat}. As opposed to the contexts of λ^{Gtz}, which are formal expressions of λ^{Gtz}, meta-contexts are not formal expressions of λ_{Nat}, but rather a device in the meta-language. Intuitively, a meta-context is a substitution with a "hole": $\{[]N_1...N_k/x\}P$. Formally, given E, we define $\mathcal{C}[E]$ (the result of filling E in the hole of $\mathcal{C}$) by recursion on $\mathcal{C}$: $(\{[]/x\}P)[E] = \{E/x\}P$ and $(N \cdot \mathcal{C})[E] = \mathcal{C}[EN]$. So $N \cdot \mathcal{C}$ can be thought of as the meta-context $\mathcal{C}[[]N]$.

The reduction rules of λ_{Nat} are as follows:

$$(\beta)\ \mathcal{C}[hd(\lambda x.M)N] \to \langle N/x\rangle(\mathcal{C}[hd(M)]) \qquad (\sigma)\ \langle M/x\rangle P \to [M/x]P$$
$$(\pi)\ \mathcal{C}[hd(\{E/x\}P)] \to \{E/x\}(\mathcal{C}[hd(P)]) \qquad (\mu)\ \{E/x\}(\mathcal{C}[hd(x)]) \to \mathcal{C}[E]$$

There are three reduction rules, β, π and σ, enforcing every head to be of the form $hd(x)$ and to be in the function position of some application (hence not in the actual-parameter position of some substitution). The $\beta\pi\sigma$-normal forms are given by:

$$M, N, P ::= x \,|\, \lambda x.M \,|\, \{EN/x\}P$$
$$E ::= hd(x) \,|\, EN$$

Later on, we will refer to this set as $\boxed{\mathrm{A}}$.

By *normalisation* we mean $\beta\pi\sigma$-reduction. At the level of derivations, the *normality criterion* is: a derivation in λ_{Nat} is $\beta\pi\sigma$-normal if every coercion formula occurring in it is an assumption and the main premiss of an elimination. This extends von Plato's criterion of normality. Indeed, if m is always 1 in $\{hd(M)N_1...N_m/x\}P$, coercion formula = main premiss of elimination, and the criterion boils down to: the main premiss of an elimination is an assumption.

The particular case $P = x$ of rule σ reads $ap(hd(M)) \to M$ and is named ϵ. There is a fourth reduction rule, named μ, which is a handy tool not available in λg. Consider the λ-term xN_1N_2, that is, $ap(hd(ap(hd(x)N_1))N_2)$. After a π step we get $\{hd(x)N_1/z_1\}\{hd(z_1)N_2/z_2\}z_2$ (z_i's fresh), which is a $\beta\pi\sigma$-normal form, if N_1, N_2 are. After a μ step one gets $ap(hd(x)N_1N_2)$, which is much simpler.

Related Systems: A term M is $\beta\pi\sigma$-normal and only contains substitutions of the form $ap(E)$ iff M is a normal term of Cervesato and Pfenning's coercion calculus in [1]. Later on, we will refer to the class of such terms as $\boxed{\mathrm{B}}$. They

are given by $M, N ::= x \,|\, \lambda x.M \,|\, ap(EN)$ and $E ::= hd(x) \,|\, EN$. Another characterisation of this set is as the set of β-normal forms of $\lambda\mathcal{N}$, a coercion calculus studied in [5].

Fragments of λ_{Nat} are determined by placing restrictions on the number m in $\{hd(M)N_1...N_m/x\}P$. There is a $m \leq 1$-fragment, which gives a version λ_{gs} of the λg-calculus with explicit substitution λgs. The β-rule of λgs is recovered as follows. Let $\mathcal{C} = \{[]/y\}P$. Then $\{hd(\lambda x.M)N/y\}P = \mathcal{C}[hd(\lambda x.M)N] \to_\beta \langle N/x\rangle\mathcal{C}[hd(M)] = \langle N/x\rangle\langle M/y\rangle P$. The π-rule of λgs is recovered as follows. Let $E = hd(M)N$ and $\mathcal{C} = N' \cdot \{[]/y\}P'$. Then $\{hd(\{hd(M)N/x\}P)N'/y\}P' = \mathcal{C}[hd(\{E/x\}P)] \to_\pi \{E/x\}\mathcal{C}[hd(P)] = \{hd(M)N/x\}\{hd(P)N'/y\}P'$.

The $m \leq 1$-terms are the terms normal w.r.t. the following *permutation rule*

$$(\nu) \qquad \{ENN'/y\}P \to \{EN/z\}\{hd(z)N'/y\}P \ ,$$

with $z \notin N', P$. Notice that $\nu \subseteq \mu^{-1}$. Clearly, ν is terminating and locally confluent. The ν-nf of M is written $\nu(M)$.

4 Isomorphism

Mappings Ψ and Θ: We start with a mapping $\Psi : \lambda_{\mathsf{Nat}} - Terms \longrightarrow \lambda^{\mathsf{Gtz}} - Terms$. Let $\Psi(M) = t$, $\Psi(N_i) = u_i$ and $\Psi(P) = v$. The idea is to map, say, $\{hd(M)N_1N_2N_3/x\}P$ to $t(u_1 :: u_2 :: u_3 :: (x)v)$. This is achieved with the help of an auxiliary function $\Psi' : \lambda_{\mathsf{Nat}} - EEs \times \lambda^{\mathsf{Gtz}} - Contexts \longrightarrow \lambda^{\mathsf{Gtz}} - Terms$ as follows:

$$\begin{array}{rlrl} \Psi(x) &= x & \Psi'(hd(M), k) &= (\Psi M)k \\ \Psi(\lambda x.M) &= \lambda x.\Psi M & \Psi'(EN, k) &= \Psi'(E, \Psi N :: k) \\ \Psi(\{E/x\}P) &= \Psi'(E, (x)\Psi P) & & \end{array}$$

Next we consider a mapping $\Theta : \lambda^{\mathsf{Gtz}} - Terms \longrightarrow \lambda_{\mathsf{Nat}} - Terms$. Let $\Theta(t) = M$, $\Theta(u_i) = N_i$ and $\Theta(v) = P$. The idea is to map, say, $t(u_1 :: u_2 :: u_3 :: (x)v)$ to $\{hd(M)N_1N_2N_3/x\}P$. This is achieved with the help of an auxiliary function $\Theta' : \lambda_{\mathsf{Nat}} - EEs \times \lambda^{\mathsf{Gtz}} - Contexts \longrightarrow \lambda_{\mathsf{Nat}} - Terms$ as follows:

$$\begin{array}{rlrl} \Theta(x) &= x & \Theta'(E, (x)v) &= \{E/x\}\Theta v \\ \Theta(\lambda x.t) &= \lambda x.\Theta t & \Theta'(E, u :: k) &= \Theta'(E\Theta u, k) \\ \Theta(tk) &= \Theta'(hd(\Theta t), k) & & \end{array}$$

Contexts vs Meta-contexts: Let $MetaContexts$ be the set of meta-contexts of λ_{Nat}. It is obvious that there is a connection between contexts of λ^{Gtz} and meta-contexts of λ_{Nat}. There is a function $\Theta_{_} : Contexts \to MetaContexts$ defined by $\Theta_{(x)v} = \{[]/x\}\Theta v$ and $\Theta_{u::k} = \Theta u \cdot \Theta_k$, and a function $\Psi_{_} : MetaContexts \to Contexts$ defined by $\Psi_{\{[]/x\}P} = (x)\Psi P$ and $\Psi_{N \cdot \mathcal{C}} = \Psi N :: \Psi_{\mathcal{C}}$.

We can identify each meta-context $\mathcal{C}$ of λ_{Nat} with a function of type $EEs \to Substs$, where $Substs$ is the set $\{M \in \lambda_{\mathsf{Nat}} : M$ is of the form $\{E/x\}P\}$; it is the function that sends E to $\mathcal{C}[E]$ (hence $\mathcal{C}(E) = \mathcal{C}[E]$). Now let k be a context of λ^{Gtz} and consider $\Theta'(_, k) : EEs \to Substs$. By induction on k one proves easily that $\Theta'(_, k)$ and Θ_k are the same function, *i.e.* $\Theta_k[E] = \Theta'(E, k)$.

Theorem 2 (Isomorphism). *Mappings Ψ and Θ are sound, mutually inverse bijections between the set of λ^{Gtz}-terms and the set of λ_{Nat}-terms. Moreover, for each $R \in \{\beta, \sigma, \pi, \mu\}$:*

1. *$t \to_R t'$ in λ^{Gtz} iff $\Theta t \to_R \Theta t'$ in λ_{Nat}.*
2. *$M \to_R M'$ in λ_{Nat} iff $\Psi M \to_R \Psi M'$ in λ^{Gtz}.*

Proof: For bijection, prove $\Theta\Psi M = M$ and $\Theta\Psi'(E,k) = \Theta'(E,k)$ by simultaneous induction on M and E, and prove $\Psi\Theta t = t$ and $\Psi\Theta'(E,k) = \Psi'(E,k)$, by simultaneous induction on t and k. It follows that $k = \Psi_{\mathcal{C}}$ iff $\mathcal{C} = \Theta_k$. As to isomorphism, the "if" statements follow from the "only if" statements and bijection. We just sketch the "only if" statement 1, which is proved together with the claim that, if $k \to_R k'$ in λ^{Gtz}, then, for all E, $\Theta_k[E] \to_R \Theta'_k[E]$ in λ_{Nat}. The proof is by simultaneous induction on $t \to_R t'$ and $k \to_R k'$, and uses the following properties of Θ: (i) if $\Theta'(E',k) = \Theta'(E,(x)v)$ then $\Theta'(E', k@k') = \{E/x\}\Theta'(hd(\Theta v), k')$; (ii) $\Theta(\langle u/x\rangle t) = \langle \Theta u/x\rangle\Theta t$; (iii) $\Theta([u/x]t) = [\Theta u/x]\Theta t$. Here are the base cases:

Case β.

$$\Theta((\lambda x.t)(u :: k)) = \Theta_{u::k}[hd(\lambda x.\Theta t)] = (\Theta u \cdot \Theta_k)[hd(\lambda x.\Theta t)] = \Theta_k[hd(\lambda x.\Theta t)\Theta u]$$
$$\downarrow$$
$$\Theta(\langle u/x\rangle(tk)) \stackrel{(ii)}{=} \langle \Theta u/x\rangle\Theta(tk) = \langle \Theta u/x\rangle\Theta_k[hd(\Theta t)]$$

Case π. Suppose $\Theta'(hd(\Theta t), k) = \Theta'(E,(x)v)$.

$$\Theta((tk)k') = \Theta_{k'}[hd(\Theta'(hd(\Theta t),k))] = \Theta_{k'}[hd(\Theta'(E,(x)v))] = \Theta_{k'}[hd(\{E/x\}\Theta v)]$$
$$\downarrow$$
$$\Theta(t(k@k')) = \Theta'(hd(\Theta t), k@k') \stackrel{(i)}{=} \{E/x\}\Theta'(hd(\Theta v), k') = \{E/x\}\Theta_{k'}[hd(\Theta v)]$$

Case σ: $\Theta(\langle t/x\rangle v) \stackrel{(ii)}{=} \langle \Theta t/x\rangle\Theta v \to_\sigma [\Theta t/x]\Theta v \stackrel{(iii)}{=} \Theta([t/x]v)$.
Case μ: $\Theta_{(x)xk}[E] = \{E/x\}\Theta(xk) = \{E/x\}\Theta_k[hd(x)] \to \Theta_k[E]$. ■

Corollary 1 (SN). *Every typable $t \in \lambda_{\mathsf{Nat}}$ is $\beta\pi\sigma\mu$-SN.*

Proof: From Theorems 1 and 2. ■

5 Analyzing the Isomorphism

Cut vs Substitution, Left Introduction vs Elimination, Cut-Elimination vs Normalisation: There is an entanglement in the traditional mappings between natural deduction and sequent calculus. An elimination is translated as a combination of cut and left introduction [10] and a left introduction is translated as a combination of elimination and meta-substitution [18]. With these mappings one proves that normalisation is a *"homomorphic"* image of cut-elimination [22,17].[3]

[3] For a study of the traditional mappings between sequent calculus and natural deduction, and some of their optimizations, see [7].

The typing system of λ_{Nat} clarifies the puzzling relation between cut and substitution. Consider rule *Cut* in λ^{Gtz} and rule *Subst* in λ_{Nat}. First, we observe, as Negri and von Plato in [15], that the right cut-formula of *Cut*, but not the right substitution formula in *Subst*, may be the conclusion of a sequence of left introductions. Second, and here comes the novelty, we may also observe that the left substitution formula in *Subst*, but not left cut-formula in *Cut*, may be the conclusion of a sequence of elimination rules. So, cut is more general on the right, whereas substitution is more general on the left.

Mapping Ψ establishes bijective correspondences between occurrences of elimination EN (resp. of substitution $\{E/x\}P$) in the source term and occurrences of left introduction $u :: k$ (resp. of cut tk) in the target term (inversely for Θ). So the entanglement of traditional mappings is solved, and the outcome is that normalization in λ_{Nat} becomes the *isomorphic* image, under Θ, of cut-elimination in λ^{Gtz}.

Applicative Terms: The presentation of sequent calculus and natural deduction as systems λ^{Gtz} and λ_{Nat}, respectively, reduces the difference between the two kinds of systems to the difference between two ways of organizing "applicative terms". By "applicative term" we mean the following data: a function (or head), m arguments ($m \geq 1$) and a continuation (or tail). The notion of applicative term is intended as a common abstraction to the notions of cut in λ^{Gtz}

$$t(u_1 :: ... :: u_m :: (x)v) \ , \tag{2}$$

and substitution in λ_{Nat}

$$\{hd(M)N_1...N_m/x\}P \ . \tag{3}$$

When (2) and (3) are regarded in the abstract way of just providing the data that constitutes an applicative term, the only difference that remains between the two expressions is that (2) associates to the right, so that the head t is at the surface and the continuation $(x)v$ is hidden at the bottom of the expression, whereas (3) associates to the left, so that the head $hd(M)$ is hidden at the bottom of the expression, and the continuation x, P is at the surface. The isomorphism $\lambda^{\mathsf{Gtz}} \cong \lambda_{\mathsf{Nat}}$ may, then, be described as a mere inversion of the *associativity* of applicative terms.

Interpretations of λ^{Gtz}: From the previous paragraph follows that an interpretation of λ^{Gtz} is as a λ-calculus with right associative applicative terms. Another interpretation is as a formalized meta-calculus for λ_{Nat} (and not for a smaller natural deduction system, like λg or λgs, let alone λ). Contexts in λ^{Gtz} are the formal counterpart to meta-contexts in λ_{Nat} and the interpretation of cut $\Theta(tk) = \Theta_k[hd(\Theta t)]$ is "fill Θt in the hole of Θ_k".

Variants of the Isomorphism: $\lambda^{\mathsf{Gtz}} \cong \lambda_{\mathsf{Nat}}$ is a particular manifestation of the isomorphism between sequent calculus and natural deduction. For instance, if rule π of λ^{Gtz} is taken in the call-by-name version $(tk)(u :: k') \to t(k@(u :: k'))$ [2], avoiding a critical pair with σ, then there is corresponding version for rule π of λ_{Nat} $\mathcal{C}[hd(\{E/x\}P)N] \to \{E/x\}(\mathcal{C}[hd(P)N])$.

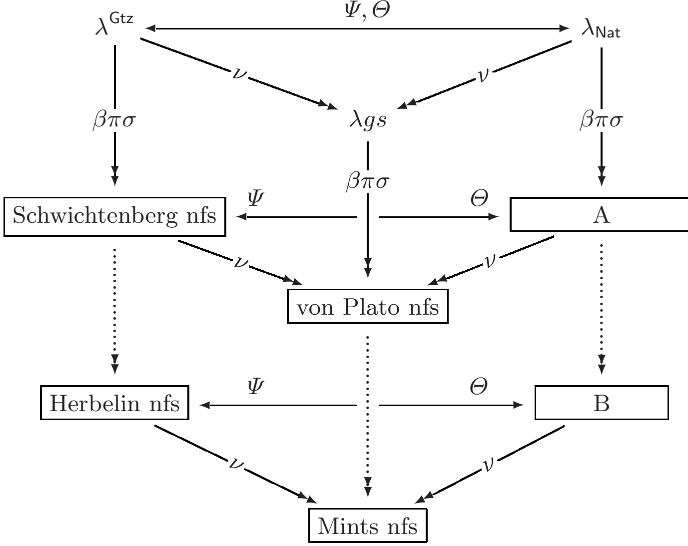


Fig. 1. Particular cases of the isomorphism and important classes of terms

Another variant of rule π is the "eager" variant, determined by a slight change in the definition of @: $((x)V)@k = (x)Vk$, if V is a value (*i.e.* variable or abstraction); and $((x)tk')@k = (x)t(k'@k)$. So, one keeps pushing k until a value is found.

Let $\{Es/xs\}P$ denote a sequence of substitutions $\{E_1/x_1\}...\{E_n/x_n\}P$. The eager variant of π for natural deduction is $\mathcal{C}[hd(\{Es/xs\}V)] \rightarrow \{Es/xs\}\mathcal{C}[hd(V)]$. So, the eager variant takes a sequence of substitutions out, as opposed to the lazy variant, which takes them one by one.

Theorem 2 still holds with eager π. In the proof fact (i) becomes slightly different: if $\Theta'(E', k) = \{Es/xs\}V$ then $\Theta'(E', k@k') = \{Es/xs\}\Theta'(hd(V), k')$.

Particular Cases of the Isomorphism: We now analyze the diagram in Figure 1. The $m \leq 1$-fragment λ^{gs} of λ^{Gtz} and the $m \leq 1$-fragment λ_{gs} of λ_{Nat} are two copies of λgs, hence isomorphic. They are identified in Figure 1. In both cases, the fragment consists of the ν-nfs. The isomorphism $\lambda^{gs} \cong \lambda_{gs}$ is a degenerate form of Theorem 2, with Θ and Ψ translating between $t(x)v$ and $\{hd(M)/x\}P$, and between $t(u :: (x)v)$ and $\{hd(M)N/x\}P$. The latter are two decompositions of generalised elimination:

$$\dfrac{\Gamma \vdash t : A \supset B \qquad \dfrac{\Gamma \vdash u : A \qquad \dfrac{\Gamma, x : B \vdash v : C}{\Gamma; B \vdash (x)v : C}\,Selection}{\Gamma; A \supset B \vdash u :: (x)v : C}\,Left}{\Gamma \vdash t(u :: (x)v) : C}\,Cut$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash M : A \supset B}{\Gamma \rhd hd(M) : A \supset B}\ Coercion \quad \Gamma \vdash N : A}{\Gamma \rhd hd(M)N : B}\ Elim \quad \Gamma, x : B \vdash P : C}{\Gamma \vdash \{hd(M)N/x\}P : C}\ Subst$$

The λ-calculus is absent from Figure 1 (λ-terms form a subset of λgs), but there are three sets in bijective correspondence with the set of β-normal λ-terms, namely [Herbelin nfs], [B] and [Mints nfs], the lower triangle. [Herbelin nfs] $\cong$ [Mints nfs] was known [4], the bijection being the restriction of ν to [Herbelin nfs]. A degenerate form of Theorem 2 is [Herbelin nfs] $\cong$ [B]. The latter bijection (but not the former) extends to another bijection, namely [Schwichtenberg nfs] $\cong$ [A] (the former bijection does not extend to another bijection because many "multiary" cut-free derivations in [Schwichtenberg nfs] have the same ν-normal form in [von Plato nfs]). The bijection [Schwichtenberg nfs] $\cong$ [A] is in turn the residue of the isomorphism $\lambda^{\mathsf{Gtz}} \cong \lambda_{\mathsf{Nat}}$, because it is the bijection between the sets of $\beta\pi\sigma$-nfs. The dotted arrows represent three reduction relations generated by permutative conversions. Two of such relations have been characterised [4,19].

6 Final Remarks

Contributions and Related Work: This paper completes Herbelin's programme, for the logic under analysis here. As compared to [11], we covered full sequent calculus, where the constraints on left introduction that define Herbelin's fragment LJT are dropped, but where the phenomenon of permutative conversions, typical of sequent calculus, shows up. In addition, we fully achieved the second goal of Herbelin's programme, residually present in [11], implicitly considered in [1] and already addressed in [5,6]. The improvement over [1] and [5,6] is that the spine calculus, when restricted to the logic of this paper, and the sequent systems in [5,6] are all fragments of Herbelin's LJT and, therefore, are under the restrictions already mentioned.

In order to fully achieve the second goal, one has to define an extension of natural deduction that combines the idea of coercion calculus with von Plato's idea of generalised elimination rule [20]. On the one hand, von Plato's work goes much farther than this paper, in that [20] covers the whole language of first order logic; on the other hand, it lacks an analysis of the correspondence between cut-elimination and normalisation, indispensable to attaining the second goal. This paper may then be seen as containing an extension of von Plato's work. Not only we extended and refined system λg (and here it is quite appealing that we end up in a system where generalised application is decomposed into *modus ponens* and substitution), but also we give the precise connection between generalised normalisation and cut-elimination, which is this: von Plato's normalisation, taken in the already slightly extended sense embodied in system λgs, is the common core of cut-elimination (in λ^{Gtz}) and normalisation (in λ_{Nat}) - in particular, it is a fragment of the former.

Once one has the natural deduction system λ_{Nat}, one can clarify the connection between cut and substitution, and translate between sequent calculus and natural deduction in a way that the classical mappings of Gentzen [10] and Prawitz [18] never could: elimination and substitution correspond to left introduction and cut, respectively. At the term calculi level, this mapping inverts the associativity of applicative terms, as envisaged by Herbelin. Then, such bijection at the level of proofs proves to be an isomorphism between cut-elimination and normalisation. This result improves, for the logic examined here, the classical results of Zucker and Pottinger [22,17].

Applications and Future Work: An issue that deserves further consideration is the use of languages λ^{Gtz} and λ_{Nat} in practice. As emphasized in [1], the spine calculus, Herbelin's $\overline{\lambda}$ and - we add - λ^{Gtz}, give a useful representation of λ-terms for procedures that act on the head of applicative terms, like normalisation or unification. It seems that the role of languages like λ^{Gtz} or λ_{Nat} is not as languages in which someone writes his programs, but either as internal languages for symbolic systems, like theorem provers, or as intermediate languages for compilers of functional languages. On the other hand, languages λ^{Gtz} and λ_{Nat} are good tools for doing proof theory efficiently, as this paper shows. We plan to keep using these languages in a more comprehensive study of permutative conversions. As the study of rule ν shows so far, calculus λ_{Nat} is no worse than calculus λ^{Gtz} for that purpose.

Conclusions: Herbelin's seminal suggestion in [11] is that the (computational) difference between sequent calculus and natural deduction may be reduced to a mere question of representation of λ-terms, when these are conceived in a sufficiently extended sense. We proposed an abstract, robust extension of the concept of λ-term, under two concrete representations (λ^{Gtz}-terms and λ_{Nat}-terms), and studied the languages where these representations live. Representation questions (like whether there is direct head access in applicative terms) prove to have impact in the real word [1]. But, as expected, they also impact on foundational matters. Indeed, they allow a radical answer to a long-standing problem of structural proof-theory: if normalisation is extended as we propose, then the meaning of $\lambda^{\mathsf{Gtz}} \cong \lambda_{\mathsf{Nat}}$ is that cut-elimination and normalisation are really the same process, they only look different because they operate with different representations of the same objects.

Acknowledgments. The author is supported by FCT, through Centro de Matemática, Universidade do Minho. We have used Paul Taylor's macros for typesetting Fig. 1.

References

1. Cervesato, I., Pfenning, F.: A linear spine calculus. Journal of Logic and Computation 13(5), 639–688 (2003)
2. Curien, P.-L., Herbelin, H.: The duality of computation. In: Proceedings of International Conference on Functional Programming 2000, IEEE (2000)

3. Dyckhoff, R., Pinto, L.: Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. Studia Logica 60, 107–118 (1998)
4. Dyckhoff, R., Pinto, L.: Permutability of proofs in intuitionistic sequent calculi. Theoretical Computer Science 212, 141–155 (1999)
5. Espírito Santo, J.: Conservative extensions of the λ-calculus for the computational interpretation of sequent calculus. PhD thesis, University of Edinburgh (2002) Available at http://www.lfcs.informatics.ed.ac.uk/reports/
6. Espírito Santo, J.: An isomorphism between a fragment of sequent calculus and an extension of natural deduction. In: Baaz, M., Voronkov, A. (eds.) LPAR 2002. LNCS (LNAI), vol. 2514, pp. 354–366. Springer, Heidelberg (2002)
7. Espírito Santo, J.: Delayed substitutions. In: Baader, F. (ed.) Proceedings of RTA'07. LNCS, Springer, Heidelberg (2007)
8. Espírito Santo, J., Pinto, L.: Permutative conversions in intuitionistic multiary sequent calculus with cuts. In: Hofmann, M.O. (ed.) TLCA 2003. LNCS, vol. 2701, pp. 286–300. Springer, Heidelberg (2003)
9. Gallier, J.: Constructive logics. Part I: A tutorial on proof systems and typed λ-calculi. Theoretical Computer Science 110, 248–339 (1993)
10. Gentzen, G.: Investigations into logical deduction. In: Szabo, M.E. (ed.) The collected papers of Gerhard Gentzen, North Holland (1969)
11. Herbelin, H.: A λ-calculus structure isomorphic to a Gentzen-style sequent calculus structure. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 61–75. Springer, Heidelberg (1995)
12. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Archive for Mathematical Logic 42, 59–87 (2003)
13. Kesner, D., Puel, L., Tannen, V.: A typed pattern calculus. Information and Computation, vol. 124(1) (1995)
14. Mints, G.: Normal forms for sequent derivations. In: Odifreddi, P. (ed.) Kreiseliana, pp. 469–492. A.K. Peters, Wellesley, Massachusetts (1996)
15. Negri, S., von Plato, J.: Structural Proof Theory. Cambridge (2001)
16. Parigot, M.: $\lambda\mu$-calculus: an algorithmic interpretation of classic natural deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, Springer, Heidelberg (1992)
17. Pottinger, G.: Normalization as a homomorphic image of cut-elimination. Annals of Mathematical Logic 12, 323–357 (1977)
18. Prawitz, D.: Natural Deduction. A Proof-Theoretical Study. Almquist and Wiksell, Stockholm (1965)
19. Schwichtenberg, H.: Termination of permutative conversions in intuitionistic gentzen calculi. Theoretical Computer Science, vol. 212 (1999)
20. von Plato, J.: Natural deduction with general elimination rules. Annals of Mathematical Logic 40(7), 541–567 (2001)
21. Wadler, P.: A Curry-Howard isomorphism for sequent calculus, Manuscript (1993)
22. Zucker, J.: The correspondence between cut-elimination and normalization. Annals of Mathematical Logic 7, 1–112 (1974)

Continuation-Passing Style and Strong Normalisation for Intuitionistic Sequent Calculi

José Espírito Santo[1], Ralph Matthes[2], and Luís Pinto[1]

[1] Departamento de Matemática, Universidade do Minho, Portugal
{jes,luis}@math.uminho.pt
[2] C.N.R.S. and University of Toulouse III, France
matthes@irit.fr

Abstract. The intuitionistic fragment of the call-by-name version of Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus is isolated and proved strongly normalising by means of an embedding into the simply-typed λ-calculus. Our embedding is a continuation-and-garbage-passing style translation, the inspiring idea coming from Ikeda and Nakazawa's translation of Parigot's $\lambda\mu$-calculus. The embedding simulates reductions while usual continuation-passing-style transformations erase permutative reduction steps. For our intuitionistic sequent calculus, we even only need "units of garbage" to be passed. We apply the same method to other calculi, namely successive extensions of the simply-typed λ-calculus leading to our intuitionistic system, and already for the simplest extension we consider (λ-calculus with generalised application), this yields the first proof of strong normalisation through a reduction-preserving embedding.

1 Introduction

CPS (continuation-passing style) translations are a tool with several theoretical uses. One of them is an interpretation between languages with different type systems or logical infra-structure, possibly with corresponding differences at the level of program constructors and computational behavior. Examples are when the source language (but not the target language): (i) allows permutative conversions, possibly related to connectives like disjunction [4]; (ii) is a language for classical logic, usually with control operators [9,10,13]; (iii) is a language for type theory [1,2] (extending (ii) to variants of pure type systems that have dependent types and polymorphism).

This article is about CPS translations for intuitionistic sequent calculi. The source and the target languages will differ neither in the reduction strategy (they will be both call-by-name) nor at the types/logic (they will be both based on intuitionistic implicational logic); instead, they will differ in the structural format of the type system: the source is in the sequent calculus format (with cut and left introduction) whereas the target is in the natural deduction format (with elimination/application). From a strictly logical point of view, this seems a new proof-theoretical use for double-negation translations.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 133–147, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Additionally, we insist that our translations simulate reduction. This is a strong requirement, not present, for instance in the concept of reflection of [23]. It seems to have been intended by [1], however does not show up in the journal version [2]. But it is, nevertheless, an eminently useful requirement if one wants to infer strong normalisation of the source calculus from strong normalisation of the simply-typed λ-calculus, as we do. In order to achieve simulation, we define continuation-and-garbage passing style (CGPS) translations, following an idea due to Ikeda and Nakazawa [13]. Garbage will provide room for observing reduction where continuation-passing alone would inevitably produce an identification, leading to failure of simulation in several published proofs for variants of operationalized classical logic, noted by [20] (the problem being β-reductions under vacuous μ-abstractions). As opposed to [13], in our intuitionistic setting garbage can be reduced to "units", and garbage reduction is simply erasing a garbage unit.

The main system we translate is the intuitionistic fragment of the call-by-name restriction of the $\overline{\lambda}\mu\tilde{\mu}$-calculus [3], here named $\lambda\mathbf{J}^{\mathbf{m}se}$. The elaboration of this system is interesting on its own. We provide a CPS and a CGPS translation for $\lambda\mathbf{J}^{\mathbf{m}se}$. We also consider other intuitionistic calculi, whose treatment can be easily derived from the results for $\lambda\mathbf{J}^{\mathbf{m}se}$. Among these is included, for instance, the λ-calculus with generalised application. For all these systems a proof of strong normalisation through a reduction-preserving embedding into the simply-typed λ-calculus is provided for the first time.

The article is organized as follows: Section 2 presents $\lambda\mathbf{J}^{\mathbf{m}se}$. Sections 3 and 4 deal with the CPS and the CGPS translation of $\lambda\mathbf{J}^{\mathbf{m}se}$, respectively. Section 5 considers other intuitionistic calculi. Section 6 compares this work with related work and concludes.

2 An Intuitionistic Sequent Calculus

In this section, we define and identify basic properties of the calculus $\lambda\mathbf{J}^{\mathbf{m}se}$. A detailed explanation of the connection between $\lambda\mathbf{J}^{\mathbf{m}se}$ and $\overline{\lambda}\mu\tilde{\mu}$ is left to the end of this section.

There are three classes of expressions in $\lambda\mathbf{J}^{\mathbf{m}se}$:

$$\begin{array}{lrl} \text{(Terms)} & t,u,v & ::= x \mid \lambda x.t \mid \{c\} \\ \text{(Co-terms)} & l & ::= [] \mid u :: l \mid (x)c \\ \text{(Commands)} & c & ::= tl \end{array}$$

An *evaluation context* E is a co-term of the form $[]$ or $u :: l$. Terms can be variables (of which we assume a denumerable set ranged over by letters x, y, w, z), lambda-abstractions $\lambda x.t$ or coercions $\{c\}$ from commands to terms[1]. A *value* is a term which is either a variable or a lambda-abstraction. We use letter V to range over values. Co-terms provide means of forming lists of arguments,

[1] A version of $\lambda\mathbf{J}^{\mathbf{m}se}$ with implicit coercions would be possible but to the detriment of the clarity, in particular, of the reduction rule ϵ below.

generalised arguments [14], or explicit substitutions. The latter two make use of the construction $(x)c$, a new binder that binds x in c. A command tl has a double role: if l is of the form $(x)c$, tl is an explicit substitution; otherwise, tl is a general form of application.

In writing expressions, sometimes we add parentheses to help their parsing. Also, we assume that the scope of binders λx and (x) extends as far as possible. Usually we write only one λ for multiple abstraction.

In what follows, we reserve letter T ("term in a large sense") for arbitrary expressions. We write $x \notin T$ if x does not occur free in T. Substitution $[t/x]T$ of a term t for all free occurrences of a variable x in T is defined as expected, where it is understood that bound variables are chosen so that no variable capture occurs. Evidently, syntactic classes are respected by substitution, i. e., $[t/x]u$ is a term, $[t/x]l$ is a co-term and $[t/x]c$ is a command.

The calculus $\lambda\mathbf{J}^{\mathbf{m}se}$ has a form of sequent for each class of expressions:

$$\Gamma \vdash t : A \qquad \Gamma | l : A \vdash B \qquad \Gamma \xrightarrow{c} B$$

Letters A, B, C are used to range over the set of types (=formulas), built from a base set of type variables (ranged over by X) using the function type (that we write $A \supset B$). In sequents, contexts Γ are viewed as finite sets of declarations $x : A$, where no variable x occurs twice. The context $\Gamma, x : A$ is obtained from Γ by adding the declaration $x : A$, and will only be written if this yields again a valid context, i. e., if x is not declared in Γ.

The typing rules of $\lambda\mathbf{J}^{\mathbf{m}se}$ can be presented as follows, stressing the parallel between left and right rules:

$$\frac{}{\Gamma | [] : A \vdash A}\ LAx \qquad \frac{}{\Gamma, x : A \vdash x : A}\ RAx$$

$$\frac{\Gamma \vdash u : A \quad \Gamma | l : B \vdash C}{\Gamma | u :: l : A \supset B \vdash C}\ LIntro \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B}\ RIntro$$

$$\frac{\Gamma, x : A \xrightarrow{c} B}{\Gamma | (x)c : A \vdash B}\ LSel \qquad \frac{\Gamma \xrightarrow{c} A}{\Gamma \vdash \{c\} : A}\ RSel$$

$$\frac{\Gamma \vdash t : A \quad \Gamma | l : A \vdash B}{\Gamma \xrightarrow{tl} B}\ Cut$$

Besides admissibility of usual weakening rules, other forms of cut are admissible as typing rules for substitution for each class of expressions.

We consider the following base reduction rules on expressions:

$$\begin{array}{llll} (\beta) & (\lambda x.t)(u :: l) \to u((x)tl) & (\mu) & (x)xl \to l, \text{ if } x \notin l \\ (\pi) & \{tl\}E \to t\,(l@E) & (\epsilon) & \{t[]\} \to t \\ (\sigma) & t(x)c \to [t/x]c, & & \end{array}$$

where, in general, $l@l'$ is a co-term that represents an "eager" concatenation of l and l', viewed as lists, and is defined as follows[2]:

$$[]@l' = l' \qquad (u :: l)@l' = u :: (l@l') \qquad ((x)tl)@l' = (x)t\,(l@l')$$

The one-step reduction relation $\rightarrow$ is inductively defined as the compatible closure of the reduction rules.

The reduction rules β, π and σ are relations on commands. The reduction rule μ (resp. ϵ) is a relation on co-terms (resp. terms). Rules β and σ generate and execute an explicit substitution, respectively. Rule π appends fragmented co-terms, bringing the term t of the π-redex $\{tl\}E$ closer to root position. Also, notice here the restricted form of the outer co-term E. This restriction characterizes call-by-name reduction [3]. A μ-reduction step that is not at the root has necessarily one of two forms: (i) $t(x)xl \rightarrow tl$, which is the execution of a linear substitution; (ii) $u :: (x)xl \rightarrow u :: l$, which is the simplification of a generalised argument. Finally, rule ϵ erases an empty list under $\{_\}$. Notice that empty lists are important under (x). Another view of ϵ is as a way of undoing a sequence of two coercions: the "coercion" of a term t to a command $t[]$, immediately followed by coercion to a term $\{t[]\}$. By the way, $\{c\}[] \rightarrow c$ is a π-reduction step. Most of these rules have genealogy: see Section 5.

The $\beta\pi\sigma$-normal forms are obtained by constraining commands to one of the two forms $V[]$ or $x(u :: l)$, where V, u, l are $\beta\pi\sigma$-normal values, terms and co-terms respectively. The $\beta\pi\sigma\epsilon$-normal forms are obtained by requiring additionally that, in coercions $\{c\}$, c is of the form $x(u :: l)$ (where u, l are $\beta\pi\sigma\epsilon$-normal terms and co-terms respectively). $\beta\pi\sigma\epsilon$-normal forms correspond to the multiary normal forms of [24]. If we further impose μ-normality as in [24], then co-terms of the form $(x)x(u :: l)$ obey to the additional restriction that x occurs either in u or l.

Subject reduction holds for $\rightarrow$. This fact is established with the help of the admissible rules for typing substitution and with the help of yet another admissible form of cut for typing the append operator.

We offer now a brief analysis of critical pairs in $\lambda\mathbf{J}^{\mathbf{m}se}$ [3]. There is a self-overlap of π ($\{\{tl\}E\}E'$), there are overlaps between π and any of β ($\{(\lambda x.t)(u :: l)\}E$), σ ($\{t(x)c\}E$) and ϵ (the latter in two different ways from $\{t[]\}E$ and $\{\{tl\}[]\}$). Finally, μ overlaps with σ ($t(x)xl$ for $x \notin l$). The last three critical pairs are trivial in the sense that both reducts are identical. Also the other critical pairs are joinable in the sense that both terms have a common $\rightarrow^*$-reduct. We only show this for the first case: $\{tl\}E \rightarrow t(l@E)$ by π, hence also $\{\{tl\}E\}E' \rightarrow \{t(l@E)\}E' =: L$. On the other hand, a direct application of π

[2] Concatenation is "eager" in the sense that, in the last case, the right-hand side is not $(x)\{tl\}l'$ but, in the only important case that l' is an evaluation context E, its π-reduct. One immediately verifies $l@[] = l$ and $(l@l')@l'' = l@(l'@l'')$ by induction on l. Associativity would not hold with the lazy version of @. Nevertheless, one would get that the respective left-hand side reduces in at most one π-step to the right-hand side.

[3] For higher-order rewrite systems, see the formal definition in [18].

yields $\{\{tl\}E\}E' \rightarrow \{tl\}(E@E') =: R$. Thus the critical pair consists of the terms L and R. $L \rightarrow t((l@E)@E')$ and $R \rightarrow t(l@(E@E'))$, hence L and R are joinable by associativity of @.

Since the critical pairs are joinable, the relation $\rightarrow$ is locally confluent [18]. Thus, from Corollary 1 below and Newman's Lemma, $\rightarrow$ is confluent on typable terms.

$\lambda\mathbf{J}^{\mathbf{m}se}$ as the Intuitionistic Fragment of CBN $\overline{\lambda}\mu\tilde{\mu}$. An appendix to this article recalls the (call-by-name) restriction of Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [3]. The reader should have in mind the non-standard naming of reduction rules.

Let $*$ be a fixed co-variable. The intuitionistic terms, co-terms and commands are generated by the grammar.

$$\begin{array}{lrl} \text{(Terms)} & t,u,v ::= & x \mid \lambda x.t \mid \mu *.c \\ \text{(Co-terms)} & e ::= & * \mid u :: e \mid \tilde{\mu}x.c \\ \text{(Commands)} & c ::= & \langle t|e\rangle \end{array}$$

Terms have no free occurrences of co-variables. Each co-term or command has exactly one free occurrence of $*$. Sequents are restricted to have exactly one formula in the RHS. Therefore, they have the particular forms $\Gamma \vdash t : A$, $\Gamma|e : A \vdash * : B$ and $c : (\Gamma \vdash * : B)$. We omit writing the intuitionistic typing rules. Reduction rules read as for $\overline{\lambda}\mu\tilde{\mu}$, except for π and $\tilde{\mu}$:

$$(\pi) \quad \langle \mu *.c|E\rangle \rightarrow [E/*]c \qquad\qquad (\tilde{\mu}) \quad \mu *.\langle t|*\rangle \rightarrow t$$

Since $* \notin t$, $[E/*]t = t$. Let us spell out $[E/*]c$ and $[E/*]e$.

$$\begin{array}{ll} [E/*]\langle t|e\rangle = \langle t|[E/*]e\rangle & [E/*](u :: e) = u :: [E/*]e \\ [E/*]* = E & [E/*](\tilde{\mu}x.c) = \tilde{\mu}x.[E/*]c \end{array}$$

If we define rule π as $\langle \mu *.\langle t|e\rangle|E\rangle \rightarrow \langle t|[E/*]e\rangle$ and $[E/*](\tilde{\mu}x.\langle t|e\rangle) = \tilde{\mu}x.\langle t|[E/*]e\rangle$ we can avoid using $[E/*]c$ altogether.

The $\lambda\mathbf{J}^{\mathbf{m}se}$-calculus is obtained from the intuitionistic fragment as a mere notational variant. The co-variable $*$ disappears from the syntax. The co-term $*$ is written $[]$. $\{c\}$ is the coercion of a command to a term, corresponding to $\mu *.c$. This coercion is what remains of the μ binder in the intuitionistic fragment. Since there is no μ, there is little sense for the notation $\tilde{\mu}$. So we write $(x)c$ instead of $\tilde{\mu}x.c$. Reduction rule $\tilde{\mu}$ now reads $\{t[]\} \rightarrow t$ and is renamed as ϵ. Sequents $\Gamma|e : A \vdash * : B$ and $c : (\Gamma \vdash * : B)$ are written $\Gamma|e : A \vdash B$ and $\Gamma \xrightarrow{c} B$. Co-terms are ranged over by l (instead of e) and thought of as generalised lists. Finally, $[E/*]l$ is written $l@E$.

3 CPS for $\lambda\mathbf{J}^{\mathbf{m}se}$

We fix a ground type (some type variable) $\perp$. Then, $\neg A := A \supset \perp$, as usual in intuitionistic logic. While our calculus is strictly intuitionistic in nature, a double-negation translation nevertheless proves useful for the purposes of establishing

strong normalisation, as has been shown by de Groote [4] for disjunction with its commuting conversions. A type A will be translated to $\overline{A} = \neg\neg A^*$, with the type A^* defined by recursion on A (where the definition of $\overline{A}$ is used as an abbreviation): $X^* = X$; $(A \supset B)^* = \neg\overline{B} \supset \neg\overline{A}$. This symmetrically-looking definition of $(A \supset B)^*$ is logically equivalent to $\overline{A} \supset \neg\neg\overline{B}$. The additional double negation of $\overline{B}$ is needed to treat cuts with co-terms ending in $(x)c$ (that are already present as generalised application in $\lambda\mathbf{J}$, see Section 5).

The translation of all syntactic elements T will be presented in Plotkin's [21] colon notation $(T : K)$ for some term K taken from simply-typed λ-calculus. A term t of $\lambda\mathbf{J}^{\mathbf{m}se}$ will then be translated into the simply-typed λ-term

$$\overline{t} = \lambda k.(t : k)$$

with a "new" variable k. The definition of $(T : K)$ uses the definition of $\overline{t}$ as an abbreviation (the variables m, w are supposed to be "fresh"):

$$\begin{array}{rclrcl}
(x : K) &=& xK & ([] : K) &=& \lambda w.wK \\
(\lambda x.t : K) &=& K(\lambda wx.w\overline{t}) & (u :: l : K) &=& \lambda w.w(\lambda m.m\,(l : K)\,\overline{u}) \\
(\{c\} : K) &=& (c : K) & ((x)c : K) &=& \lambda x.(c : K)
\end{array}$$

$$\begin{array}{rcl}
(t[] : K) &=& (t : K) \\
(t(u :: l) : K) &=& (t : \lambda m.m\,(l : K)\,\overline{u}) \\
(t(x)c : K) &=& (\lambda x.(c : K))\overline{t}
\end{array}$$

The translation obeys to the following typing:

$$\frac{\Gamma \vdash t : A \quad \overline{\Gamma} \vdash K : \neg A^*}{\overline{\Gamma} \vdash (t : K) : \bot} \qquad \frac{\Gamma \xrightarrow{c} A \quad \overline{\Gamma} \vdash K : \neg A^*}{\overline{\Gamma} \vdash (c : K) : \bot}$$

$$\frac{\Gamma | l : A \vdash B \quad \overline{\Gamma} \vdash K : \neg B^*}{\overline{\Gamma} \vdash (l : K) : \neg\overline{A}}$$

Only the first premise in all these three rules refers to $\lambda\mathbf{J}^{\mathbf{m}se}$, the other ones to simply-typed λ-calculus. $\overline{\Gamma}$ is derived from Γ by replacing every $x : C$ in Γ by $x : \overline{C}$. As a direct consequence (to be established during the proof of the above typings), type soundness of the CPS translation follows:

$$\Gamma \vdash_{\lambda\mathbf{J}^{\mathbf{m}se}} t : A \Longrightarrow \overline{\Gamma} \vdash_{\lambda} \overline{t} : \overline{A}$$

This CPS translation is also sound for reduction, in the sense that each reduction step in $\lambda\mathbf{J}^{\mathbf{m}se}$ translates to zero or more β-steps in λ-calculus. Because of the collapsing of some reductions, this result does not guarantee yet strong normalisation of $\lambda\mathbf{J}^{\mathbf{m}se}$.

Proposition 1. *If $t \to u$ in $\lambda\mathbf{J}^{\mathbf{m}se}$, then $\overline{t} \to_\beta^* \overline{u}$ in the λ-calculus.*

Proof. Simultaneously we prove $T \to T' \Longrightarrow (T : K) \to_\beta^* (T' : K)$ for T, T' terms, co-terms or commands. More specifically, at the base cases, the CPS translation does the following: identifies ϵ and π-steps; sends one μ-step into zero or more β-steps in λ-calculus; sends one β or σ-step into one or more β-steps in λ-calculus. □

4 CGPS for $\lambda\mathbf{J}^{\mathbf{m}se}$

This is the central mathematical finding of the present article. It is very much inspired from a "continuation and garbage passing style" translation for Parigot's $\lambda\mu$-calculus, proposed by Ikeda and Nakazawa [13]. While they use garbage to overcome the problems of earlier CPS translations that did not carry β-steps to at least one β-step if they were under a vacuous μ-binding, as reported in [20], we ensure proper simulation of ϵ, π and μ. Therefore, we can avoid the separate proof of strong normalisation of permutation steps alone that is used in addition to the CPS in [4] (there in order to treat disjunction and not for sequent calculi as we do).

We use the type $\top$ for "garbage", i. e., terms that are carried around for their operational properties, not for denotational purposes. We only require the following from $\top$: There is a term $\mathsf{s}(\cdot) : \top \to \top$ such that $\mathsf{s}(x) \to_\beta^+ x$. This can, e. g., be realised by $\top := \bot \to \bot$ and $\mathsf{s}(\cdot) := \lambda x.(\lambda y.x)(\lambda z.z)$. We abbreviate $[t;u] := (\lambda x.t)u$ for some $x \notin t$. Then, $[t;u] \to_\beta t$, and $\Gamma \vdash t : A$ and $\Gamma \vdash u : B$ together imply $\Gamma \vdash [t;u] : A$.

The only change w. r. t. the type translation in CPS is that, now,

$$\overline{A} = \top \supset \neg\neg A^*$$

is used throughout, hence, again, $X^* = X$ and $(A \supset B)^* = \neg\overline{B} \supset \neg\overline{A}$.

We define the simply-typed λ-term $(T : G, K)$ for every syntactic construct T of $\lambda\mathbf{J}^{\mathbf{m}se}$ and simply-typed λ-terms G (for "garbage") and K. Then, the translation of term t is defined to be

$$\overline{t} = \lambda gk.(t : g, k)$$

with "new" variables g, k, that is again used as an abbreviation inside the recursive definition of $(T : G, K)$ as follows (the variables m, w are again "fresh"):

$$\begin{aligned}(x : G, K) &= x\,\mathsf{s}(G)K & ([] : G, K) &= \lambda w.w\,\mathsf{s}(G)K\\ (\lambda x.t : G, K) &= [K(\lambda wx.w\overline{t}); G] & (u :: l : G, K) &= \lambda w.w\,\mathsf{s}(G)(\lambda m.m\,(l : G, K)\,\overline{u})\\ (\{c\} : G, K) &= (c : G, K) & ((x)c : G, K) &= \lambda x.(c : G, K)\end{aligned}$$

$$\begin{aligned}(t[] : G, K) &= (t : \mathsf{s}(G), K)\\ (t(u :: l) : G, K) &= (t : \mathsf{s}(G), \lambda m.m\,(l : G, K)\,\overline{u})\\ (t(x)c : G, K) &= (\lambda x.(c : G, K))\overline{t}\end{aligned}$$

If one removes the garbage argument, one precisely obtains the CPS translation.

The translation obeys to the following typing:

$$\frac{\Gamma \vdash t : A \quad \overline{\Gamma} \vdash G : \top \quad \overline{\Gamma} \vdash K : \neg A^*}{\overline{\Gamma} \vdash (t : G, K) : \bot} \qquad \frac{\Gamma|l : A \vdash B \quad \overline{\Gamma} \vdash G : \top \quad \overline{\Gamma} \vdash K : \neg B^*}{\overline{\Gamma} \vdash (l : G, K) : \neg\overline{A}}$$

$$\frac{\Gamma \xrightarrow{c} A \quad \overline{\Gamma} \vdash G : \top \quad \overline{\Gamma} \vdash K : \neg A^*}{\overline{\Gamma} \vdash (c : G, K) : \bot}$$

For $\overline{\Gamma}$ see the previous section. Therefore (and to be proven simultaneously), the CGPS translation satisfies type soundness, i. e., $\Gamma \vdash t : A$ implies $\overline{\Gamma} \vdash \overline{t} : \overline{A}$.

Lemma 1

1. $[t/x](T : G, K) = (T : [t/x]G, [t/x]K)$ *for* T *any* u, l *or* c *such that* $x \notin T$.
2. $[\overline{t}/x](T : G, K) \to^*_\beta ([t/x]T : [\overline{t}/x]G, [\overline{t}/x]K)$ *for* T *any* u, l *or* c.
3. G *and* K *are subterms of* $(T : G, K)$ *for* T *any* u, l *or* c.
4. $(l : G, K)\overline{t} \to^*_\beta (tl : G, K)$
5. $\lambda x.(xl : G, K) \to^+_\beta (l : G, K)$ *if* $x \notin l$.
6. (a) $(tl : \mathsf{s}(G), \lambda m.m(l' : G, K)\overline{u}) \to^+_\beta (t\,(l@(u :: l')) : G, K)$
   (b) $(l : \mathsf{s}(G), \lambda m.m(l' : G, K)\overline{u}) \to^+_\beta (l@(u :: l') : G, K)$

Proof. 1./2./3. Each one by simultaneous induction on terms, co-terms and commands. 4./5. Case analysis on l. 6. By simultaneous induction on l. □

If we remove the garbage argument in statements 6 and 5 of this lemma, we can no longer guarantee one or more β-steps in λ-calculus. In the first case we have identity, whereas in the second case we have zero or more β-steps in λ-calculus. These differences account for the gain of simulation of π and μ-steps, when moving from CPS to CGPS.

Theorem 1 (Simulation). *If* $t \to u$ *in* $\lambda\mathbf{J}^{\mathbf{m}se}$, *then* $\overline{t} \to^+_\beta \overline{u}$ *in the* λ*-calculus.*

Proof. Simultaneously we prove: $T \to T' \implies (T : G, K) \to^+_\beta (T' : G, K)$ for T, T' terms, co-terms or commands. We illustrate the cases of the base rules.

Case β: $(\lambda x.t)(u :: l) \to u(x)tl$.

$$\begin{array}{rcll} ((\lambda x.t)(u :: l) : G, K) & = & (\lambda x.t : \mathsf{s}(G), \lambda m.m(l : G, K)\overline{u}) & \\ & = & [(\lambda m.m(l : G, K)\overline{u})(\lambda wx.w\overline{t}); \mathsf{s}(G)] & \\ & \to^3_\beta & (\lambda x.(l : G, K)\overline{t})\overline{u} & \\ & \to^*_\beta & (\lambda x.(tl : G, K))\overline{u} & \text{(Lemma 1.4)} \\ & = & (u(x)tl : G, K) & \end{array}$$

Case π: $\{tl\}E \to t\,(l@E)$. Sub-case $E = []$.

$$\begin{array}{rcll} (\{tl\}[] : G, K) = (tl : \mathsf{s}(G), K) & \to^+_\beta & (tl : G, K) & \text{(Lemma 1.3/1.1)} \\ & = & (t\,(l@[]) : G, K). & \end{array}$$

Sub-case $E = u :: l'$.

$$\begin{array}{rcll} (\{tl\}(u :: l') : G, K) & = & (tl : \mathsf{s}(G), \lambda m.m(l' : G, K)\overline{u}) & \\ & \to^+_\beta & (t\,(l@(u :: l')) : G, K) & \text{(Lemma 1.6)} \end{array}$$

Case σ: $t(x)c \to [t/x]c$.

$$\begin{array}{rcll} (t(x)c : G, K) & = & (\lambda x.(c : G, K))\overline{t} & \\ & \to_\beta & [\overline{t}/x](c : G, K) & \\ & \to^*_\beta & ([t/x]c : G, K) & \text{(Lemma 1.2)} \end{array}$$

Case μ: $(x)xl \to l$, if $x \notin l$.

$$((x)xl : G, K) = \lambda x.(xl : G, K) \to^+_\beta (l : G, K) \quad \text{(Lemma 1.5)}$$

Case ϵ: $\{t[]\} \to t$.

$$(\{t[]\} : G, K) = (t[] : G, K) = (t : \mathsf{s}(G), K) \to^+_\beta (t : G, K)$$

The cases corresponding to the closure rule $t \to t' \Longrightarrow tl \to t'l$ (resp. $l \to l' \Longrightarrow tl \to tl'$) can be proved by case analysis on l (resp. $l \to l'$). The cases corresponding to the other closure rules follow by routine induction. □

Remark 1. Unlike the failed simulation by CPS reported in [20] that only occurred with the closure rules, the need for garbage in our translation is already clearly visible in the subcase $E = []$ for π and the case ϵ. But the garbage is also effective for the closure rules, where the most delicate rule is the translation of $t(u :: l)$ that mentions l and u only in the continuation argument K to t's translation. Lemma 1.3 is responsible for propagation of simulation. The structure of our garbage – essentially just "units of garbage" – can thus be easier than in the CGPS in [13] for $\lambda\mu$-calculus since there, K cannot be guaranteed to be a subterm of $(T : G, K)$, again because of the problem with void μ-abstractions. The solution of [13] for the most delicate case of application is to copy the K argument into the garbage. We do not need this in our intuitionistic calculi.

Corollary 1. *The typable terms of* $\lambda\mathbf{J}^{\mathbf{m}se}$ *are strongly normalising.*

Recalling our discussion in Section 2, we already could have inferred strong normalisation of $\lambda\mathbf{J}^{\mathbf{m}se}$ from that of $\overline{\lambda}\mu\tilde{\mu}$, which has been shown directly by Polonovski [22] using reducibility candidates and before by Lengrand's [16] embedding into a calculus by Urban that also has been proven strongly normalizing by the candidate method. Our proof is just by a syntactic transformation to simply-typed λ-calculus.

5 CGPS for Other Intuitionistic Calculi

As a consequence of the results of the previous section, we obtain in this section the embedding, by a CGPS translation, of several intuitionistic calculi into the simply-typed λ-calculus. These intuitionistic calculi are successive extensions of the simply-typed λ-calculus that lead to $\lambda\mathbf{J}^{\mathbf{m}se}$, as illustrated in the diagram below, and include both natural deduction systems and other sequent calculi.

Each extension step adds both a new feature and a reduction rule to the preceding calculus. The following table summarizes these extensions.

calculus	reduction rules	feature added
λ	β	
$\lambda\mathbf{J}$	β, π	generalised application
$\lambda\mathbf{J}^{\mathbf{m}}$	β, π, μ	multiarity
$\lambda\mathbf{J}^{\mathbf{m}s}$	β, π, μ, σ	explicit substitution
$\lambda\mathbf{J}^{\mathbf{m}se}$	$\beta, \pi, \mu, \sigma, \epsilon$	empty lists of arguments

The scheme for naming systems and reduction rules intends to be systematic (and in particular explains the name $\lambda\mathbf{J}^{\mathbf{m}se}$).

The path between the two end-points of this spectrum visits and organizes systems known from the literature. $\lambda\mathbf{J}$ is a variant of the calculus $\mathit{\Lambda J}$ of [14]. $\lambda\mathbf{J}^{\mathbf{m}}$ is a variant of the system in [8]. $\lambda\mathbf{J}^{\mathbf{m}se}$ is studied in [7] under the name λ^{Gtz}. This path is by no means unique. Other intermediate systems could have been visited (like the multiary λ-calculus $\lambda^{\mathbf{m}}$, named λPh in [8]), had the route been a different one, i. e., had the different new features been added in a different order.

Each system $\mathcal{L} \in \{\lambda\mathbf{J}, \lambda\mathbf{J}^{\mathbf{m}}, \lambda\mathbf{J}^{\mathbf{m}s}\}$ embeds in the system immediately after it in this spectrum, in the sense of existing a mapping simulating reduction. Hence, strong normalisation is inherited from $\lambda\mathbf{J}^{\mathbf{m}se}$ all the way down to $\lambda\mathbf{J}$. Also, each $\mathcal{L} \in \{\lambda\mathbf{J}, \lambda\mathbf{J}^{\mathbf{m}}, \lambda\mathbf{J}^{\mathbf{m}s}\}$ has, by composition, an embedding $g_{\mathcal{L}}$ in $\lambda\mathbf{J}^{\mathbf{m}se}$. It can be shown that there is a CGPS translation of each $\mathcal{L}$ so that this CGPS translation is the composition of $g_{\mathcal{L}}$ with the CGPS translation of $\lambda\mathbf{J}^{\mathbf{m}se}$. It follows that the CGPS translation of each $\mathcal{L}$ simulates reduction, that is, is an embedding of $\mathcal{L}$ in the λ-calculus. Let us see all this with some detail.

λJ-Calculus. The terms of $\lambda\mathbf{J}$ are generated by the grammar:

$$t, u, v ::= x \,|\, \lambda x.t \,|\, t(u, x.v)$$

Construction $t(u, x.v)$ is called generalised application. Following [14], $(u, x.v)$ is called a generalised argument; they will be denoted by the letters R and S. Typing rules for x and $\lambda x.t$ are as usual and omitted. The typing rule for generalised application is:

$$\frac{\Gamma \vdash t : A \supset B \quad \Gamma \vdash u : A \quad \Gamma, x : B \vdash v : C}{\Gamma \vdash t(u, x.v) : C}\ GApp$$

Reduction rules are as in [14], except that π is defined in the "eager" way:

$$(\beta)\ (\lambda x.t)(u, y.v) \to [[u/x]t/y]v \qquad (\pi)\ tRS \to t(R@S)$$

where the generalised argument $R@S$ is defined by recursion on R:

$$(u, x.V)@S = (u, x.VS) \qquad (u, x.tR')@S = (u, x.t(R'@S))\ ,$$

for V a value, i. e., a variable or a λ-abstraction. The operation @ is associative, which allows to join the critical pair of π with itself as before for $\lambda\mathbf{J}^{\mathbf{m}se}$. The other critical pair stems from the interaction of β and π and is joinable as well.

Strong normalisation of typable terms immediately follows from that of ΛJ in [15], but in the present article, we even get an embedding into λ.

In defining the embeddings m, s and e we omit the clauses for variables and λ-abstraction, because in these cases the embeddings are defined homomorphically. Although we won't use it, we recall the embedding $J : \lambda \to \lambda\mathbf{J}$ just for completeness: $J(tu) = J(t)(J(u), x.x)$.

λJ$^{\mathbf{m}}$-Calculus. We offer now a new, lighter, presentation of the system in [8]. The expressions of $\lambda\mathbf{J}^{\mathbf{m}}$ are given by the grammar:

$$\text{(Terms)}\ \ t, u, v ::= x \,|\, \lambda x.t \,|\, t(u, l) \qquad \text{(Co-terms)}\ \ l ::= u :: l \,|\, (x)v$$

The application $t(u,l)$ is both generalised and multiary. Multiarity is the ability of forming a chain of arguments, as in $t(u_1, u_2 :: u_3 :: (x)v)$. By the way, this term is written $t(u_1, u_2 :: u_3 :: [], (x)v)$ in the syntax of [8]. There are two kinds of sequents: $\Gamma \vdash t : A$ and $\Gamma | l : A \vdash B$. Typing rules are as follows:

$$\frac{\Gamma \vdash t : A \supset B \quad \Gamma \vdash u : A \quad \Gamma | l : B \vdash C}{\Gamma \vdash t(u,l) : C}\ GMApp$$

$$\frac{\Gamma, x : A \vdash v : B}{\Gamma | (x)v : A \vdash B}\ Sel \qquad \frac{\Gamma \vdash u : A \quad \Gamma | l : B \vdash C}{\Gamma | u :: l : A \supset B \vdash C}\ LIntro$$

We re-define reduction rules of [8] in this new syntax. Rule μ can now be defined as a relation on co-terms. Rule π is changed to the "eager" version, using letters R and S for generalised arguments, i. e., elements of the form (u,l).

$$\begin{array}{llll} (\beta_1) & (\lambda x.t)(u,(y)v) \to [[u/x]t/y]v & (\pi) & tRS \to t(R@S) \\ (\beta_2) & (\lambda x.t)(u, v :: l) \to ([u/x]t)(v,l) & (\mu) & (x)x(u,l) \to u :: l, \text{ if } x \notin u,l \end{array}$$

$\beta = \beta_1 \cup \beta_2$. The generalised argument $R@S$ is defined with the auxiliary notion of the co-term $l@S$ that is defined by recursion on l by $(u :: l)@S = u :: (l@S)$, $((x)V)@S = (x)VS$, for V a value, and $((x)t(u,l))@S = (x)t(u, l@S)$. Then, define $R@S$ by $(u,l)@S = (u, l@S)$. Since the auxiliary operation @ can be proven associative, this also holds for the operation @ on generalised arguments. Apart from the usual self-overlapping of π that is joinable by associativity of @, there are critical pairs between β_i and π that are joinable. The last critical pair is between β_1 and μ and needs a β_2-step to be joined.

The embedding $m : \lambda\mathbf{J} \to \lambda\mathbf{J^m}$ is given by $m(t(u,x.v)) = m(t)(m(u),(x)m(v))$.

$\lambda\mathbf{J}^{\mathbf{m}s}$-Calculus. The expressions of $\lambda\mathbf{J}^{\mathbf{m}s}$ are given by:

$$\text{(Terms)}\ \ t,u,v ::= x \,|\, \lambda x.t \,|\, tl \qquad \text{(Co-terms)}\ \ l ::= u :: l \,|\, (x)v$$

The construction tl has a double role: either it is a generalised and multiary application $t(u :: l)$ or it is an explicit substitution $t(x)v$. The typing rules for $u :: l$ and $(x)v$ are as in $\lambda\mathbf{J^m}$. Construction tl is typed by:

$$\frac{\Gamma \vdash t : A \quad \Gamma | l : A \vdash B}{\Gamma \vdash tl : B}\ Cut$$

The reduction rules are as follows:

$$\begin{array}{llll} (\beta) & (\lambda x.t)(u :: l) \to u((x)tl) & (\sigma) & t(x)v \to [t/x]v \\ (\pi) & (tl)(u :: l') \to t\,(l@(u :: l')) & (\mu) & (x)xl \to l, \text{ if } x \notin l \end{array}$$

where the co-term $l@l'$ is defined by $(u :: l)@l' = u :: (l@l')$, $((x)V)@l' = (x)Vl'$, for V a value, and $((x)tl)@l' = (x)t\,(l@l')$. Again, @ is associative and guarantees the joinability of the critical pair of π with itself. The critical pairs between β and π and between σ and μ are joinable as for $\lambda\mathbf{J}^{\mathbf{m}se}$. The overlap between σ

and π is bigger than in $\lambda\mathbf{J}^{\mathbf{m}se}$ since the divergence arises for $t((x)v)(u :: l)$ with v an arbitrary term whereas in $\lambda\mathbf{J}^{\mathbf{m}se}$, there is only a command at that place. Joinability is nevertheless easily established.

Comparing these reduction rules with those of $\lambda\mathbf{J}^{\mathbf{m}}$, there is only one β-rule, whose effect is to generate a substitution. There is a separate rule σ for substitution execution. The embedding $s : \lambda\mathbf{J}^{\mathbf{m}} \to \lambda\mathbf{J}^{\mathbf{m}s}$ is characterized by $s(t(u, l)) = s(t)(s(u) :: s(l))$.

Finally, let us compare $\lambda\mathbf{J}^{\mathbf{m}s}$ and $\lambda\mathbf{J}^{\mathbf{m}se}$. In the former, any term can be in the scope of a selection (x), whereas in the latter the scope of a selection is a command. But in the latter we have a new form of co-term $[]$. Since in $\lambda\mathbf{J}^{\mathbf{m}se}$ we can coerce any term t to a command $t[]$, we can translate $\lambda\mathbf{J}^{\mathbf{m}s}$ into $\lambda\mathbf{J}^{\mathbf{m}se}$, by defining $e((x)t) = (x)e(t)[]$. In fact, one has to refine this idea in order to get simulation of reduction. The embedding $e : \lambda\mathbf{J}^{\mathbf{m}s} \to \lambda\mathbf{J}^{\mathbf{m}se}$ obeys the following:

$$e(tl) = \{e(t)e(l)\} \qquad e((x)V) = (x)e(V)[] \qquad e((x)tl) = (x)e(t)e(l)$$

Proposition 2. *Each of the embeddings m, s and e simulates reduction.*

Proof. We just sketch the proof for e (the others are easier). We prove

$$t \to t' \Longrightarrow e(t) \to^+ e(t') \,\text{and}\, e((x)t) \to^+ e((x)t'), \;\; \text{for any}\, t, t' \in \lambda\mathbf{J}^{\mathbf{m}s}$$

simultaneously with $l \to l' \Longrightarrow e(l) \to^+ e(l')$. In particular, the following fact is used: $[e(t)/x]e(T) \to_\epsilon^* e([t/x]T)$, for T a term or a co-term. □

Since each of m, s and e preserves typability, it follows from Corollary 1 that:

Corollary 2. *The typable terms of $\lambda\mathbf{J}^{\mathbf{m}s}$, $\lambda\mathbf{J}^{\mathbf{m}}$ and $\lambda\mathbf{J}$ are strongly normalising.*

CGPS Translations. We define CGPS translations for $\lambda\mathbf{J}^{\mathbf{m}s}$, $\lambda\mathbf{J}^{\mathbf{m}}$ and $\lambda\mathbf{J}$. The translation of types is unchanged. In each translation, we just show the clauses that are new.

1. For $\lambda\mathbf{J}^{\mathbf{m}s}$ one has (the first rule just replaces c by vl in the rule for $\lambda\mathbf{J}^{\mathbf{m}se}$):

$$\begin{aligned}(t(x)vl : G, K) &= (\lambda x.(vl : G, K))\overline{t}\\ (t(x)V : G, K) &= (\lambda x.(V : \mathsf{s}(G), K))\overline{t}\\ ((x)v : G, K) &= \lambda x.(v : \mathsf{s}(G), K)\end{aligned}$$

2. For $\lambda\mathbf{J}^{\mathbf{m}}$: $(t(u, l) : G, K) = (t : \mathsf{s}(G), \lambda m.m\,(l : G, K)\,\overline{u})$.
3. Finally, for $\lambda\mathbf{J}$: $(t(u, x.v) : G, K) = (t : \mathsf{s}(G), \lambda m.m\,(\lambda x.(v : \mathsf{s}(G), K))\,\overline{u})$.

These translations are coherent with the CGPS translation for $\lambda\mathbf{J}^{\mathbf{m}se}$:

Proposition 3. *Let $\mathcal{L} \in \{\lambda\mathbf{J}^{\mathbf{m}s}, \lambda\mathbf{J}^{\mathbf{m}}, \lambda\mathbf{J}\}$. Let $f_\mathcal{L}$ be the embedding of $\mathcal{L}$ in its immediate extension and let $g_\mathcal{L}$ be the embedding of $\mathcal{L}$ in $\lambda\mathbf{J}^{\mathbf{m}se}$. Then, for all $t \in \mathcal{L}$, $\overline{t} = \overline{f_\mathcal{L}(t)}$. Hence, for all $t \in \mathcal{L}$, $\overline{t} = \overline{g_\mathcal{L}(t)}$.*

Theorem 2 (Simulation). *Let $\mathcal{L} \in \{\lambda\mathbf{J}^{\mathbf{m}s}, \lambda\mathbf{J}^{\mathbf{m}}, \lambda\mathbf{J}\}$. If $t \to u$ in $\mathcal{L}$, then $\overline{t} \to_\beta^+ \overline{u}$ in the λ-calculus.*

Proof. By Propositions 2 and 3 and Theorem 1. □

6 Further Remarks

This article provides reduction-preserving CGPS translations of $\lambda\mathbf{J}^{\mathbf{m}se}$ and other intuitionistic calculi, hence obtaining embeddings into the simply-typed λ-calculus and proving strong normalisation. As a by-product, the connections between systems like $\lambda\mathbf{J}$ and $\lambda\mathbf{J}^{\mathbf{m}}$ and the intuitionistic fragment of $\overline{\lambda}\mu\tilde{\mu}$ are detailed.

In the literature one finds strong normalisation proofs for sequent calculi [5,6,16,17,22,25], but not by means of CPS translations; or CPS translations for natural deduction systems [1,2,4,13,19].

This article provides, in particular, a reduction-preserving CGPS translation for the lambda-calculus with generalised applications $\lambda\mathbf{J}$. [19] covers full propositional classical logic with general elimination rules and its intuitionistic implicational fragment corresponds to $\lambda\mathbf{J}$. However, [19] does not prove a simulation by CPS in our sense (permutative conversions are collapsed), so an auxiliary argument in the style of de Groote [4], involving a proof in isolation of SN for permutative conversions, is used.

In Curien and Herbelin's work [3,11] one finds a CPS translation $(_)^n$ of the call-by-name restriction of $\overline{\lambda}\mu\tilde{\mu}$. We compare $(_)^n$ with our $\overline{(_)}$. (i) $(_)^n$ generalises Hofmann-Streicher translation [12]; $\overline{(_)}$ generalises Plotkin's call-by-name CPS translation [21]. (ii) $(_)^n$ does not employ the colon operator; $\overline{(_)}$ does employ (we suspect that doing administrative reductions at compile time is necessary to achieve simulation of reduction); (iii) $(_)^n$ is defined for expressions where every occurrence of $u :: l$ is of the particular form $u :: E$; no such restriction is imposed in the definition of $\overline{(_)}$. (iv) at some points it is unclear what the properties of $(_)^n$ are, but no proof of strong normalisation is claimed; the CGPS $\overline{(_)}$ simulates reduction and thus achieves a proof of strong normalisation.

The results obtained extend to second-order calculi (this extension is omitted for space reasons). We plan to extend the technique of continuation-and-garbage passing to $\overline{\lambda}\mu\tilde{\mu}$ and to dependently-typed systems. We tried to extend the CGPS to CBN $\overline{\lambda}\mu\tilde{\mu}$, described in the appendix, but already for a CPS translation, we do not see how to profit from the continuation argument for the translation of co-terms and commands. Moreover, a special case of the rule we call π corresponds to the renaming rule $a(\mu b.M) \rightarrow [a/b]M$ of $\lambda\mu$-calculus. This rule is evidently not respected by the CGPS translation by Ikeda and Nakazawa [13] (nor by the CPS they recall) since the continuation argument K is omitted in the interpretation of the left-hand side but not in the right-hand side. So, new ideas or new restrictions will be needed.

Acknowledgements. We thank the referees for pointing out the work of Nakazawa and Tatsuta, whom we thank for an advanced copy of [19]. The first and third authors are supported by FCT through the Centro de Matemática da Universidade do Minho. The second author thanks for an invitation by that institution to Braga in October 2006. All authors are also supported by the European project TYPES.

References

1. Barthe, G., Hatcliff, J., Sørensen, M.: A notion of classical pure type system (preliminary version). In: Brookes, S., Mislove, M. (eds.) Proc. of the 30th Conf. on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science, vol. 6, p. 56. Elsevier, Amsterdam (1997)
2. Barthe, G., Hatcliff, J., Sørensen, M.: Cps translations and applications: The cube and beyond. Higher-Order and Symbolic Computation 12(2), 125–170 (1999)
3. Curien, P.-L., Herbelin, H.: The duality of computation. In: Proc. of 5th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP '00), Montréal, pp. 233–243. IEEE (2000)
4. de Groote, P.: On the strong normalisation of intuitionistic natural deduction with permutation-conversions. Information and Computation 178, 441–464 (2002)
5. Dragalin, A.: Mathematical Intuitionism. Translations of Mathematical Monographs, vol. 67. AMS (1988)
6. Dyckhoff, R., Urban, C.: Strong normalisation of Herbelin's explicit substitution calculus with substitution propagation. Journal of Logic and Computation 13(5), 689–706 (2003)
7. Espírito Santo, J.: Completing Herbelin's programme (in this volume)
8. Espírito Santo, J., Pinto, L.: Permutative conversions in intuitionistic multiary sequent calculus with cuts. In: Hofmann, M.O. (ed.) TLCA 2003. LNCS, vol. 2701, pp. 286–300. Springer, Heidelberg (2003)
9. Felleisen, M., Friedman, D., Kohlbecker, E., Duba, B.: Reasoning with continuations. In: 1st Symposium on Logic and Computer Science, pp. 131–141. IEEE (1986)
10. Griffin, T.: A formulae-as-types notion of control. In: ACM Conf. Principles of Programming Languages, pp. 47–58. ACM Press, New York (1990)
11. Herbelin, H.: C'est maintenant qu'on calcule, Habilitation Thesis, Paris XI (2005)
12. Hofmann, M., Streicher, T.: Continuation models are universal for lambda-mu-calculus. In: Proc. of LICS 1997, pp. 387–395. IEEE Computer Society Press, Los Alamitos (1997)
13. Ikeda, S., Nakazawa, K.: Strong normalization proofs by CPS-translations. Information Processing Letters 99, 163–170 (2006)
14. Joachimski, F., Matthes, R.: Standardization and confluence for a lambda-calculus with generalized applications. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 141–155. Springer, Heidelberg (2000)
15. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Archive for Mathematical Logic 42(1), 59–87 (2003)
16. Lengrand, S.: Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In: Gramlich, B., Lucas, S. (eds.) Post-proc. of the 3rd Workshop on Reduction Strategies in Rewriting and Programming (WRS'03). Electronic Notes in Theoretical Computer Science, vol. 86, Elsevier, Amsterdam (2003)
17. Lengrand, S., Dyckhoff, R., McKinna, J.: A sequent calculus for type theory. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 441–455. Springer, Heidelberg (2006)
18. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical Computer Science 192, 3–29 (1998)
19. Nakazawa, K., Tatsuta, M.: Strong normalization of classical natural deduction with disjunctions (Submitted)

20. Nakazawa, K., Tatsuta, M.: Strong normalization proof with CPS-translation for second order classical natural deduction. Journal of Symbolic Logic 68(3), 851–859 (2003), Corrigendum: 68(4), 1415–1416 (2003)
21. Plotkin, G.: Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science 1, 125–159 (1975)
22. Polonovski, E.: Strong normalization of lambda-mu-mu-tilde with explicit substitutions. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 423–437. Springer, Heidelberg (2004)
23. Sabry, A., Wadler, P.: A reflection on call-by-value. In: Proc. of ACM SIGPLAN Int. Conf. on Functional Programming ICFP 1996, pp. 13–24. ACM Press, New York (1996)
24. Schwichtenberg, H.: Termination of permutative conversions in intuitionistic Gentzen calculi. Theoretical Computer Science 212(1–2), 247–260 (1999)
25. Urban, C., Bierman, G.: Strong normalisation of cut-elimination in classical logic. In: Girard, J.-Y. (ed.) TLCA 1999. LNCS, vol. 1581, pp. 365–380. Springer, Heidelberg (1999)

A The Call-by-Name $\overline{\lambda}\mu\tilde{\mu}$-Calculus

In this appendix we recall the call-by-name restriction of $\overline{\lambda}\mu\tilde{\mu}$-calculus [3] (with the subtraction connective left out). Expressions are either terms, co-terms or commands and are defined by:

$$t, u, v ::= x \,|\, \lambda x.t \,|\, \mu a.c \qquad e ::= a \,|\, u :: e \,|\, \tilde{\mu} x.c \qquad c ::= \langle t|e\rangle$$

Variables (resp. co-variables) are ranged over by x, y, z (resp. a, b, c). An *evaluation context* E is a co-term of the form a or $u :: e$.

There is one kind of sequent per each syntactic class

$$\Gamma \vdash t : A|\Delta \qquad \Gamma|e : A \vdash \Delta \qquad c : (\Gamma \vdash \Delta)$$

Typing rules are as in [3]. We consider 5 reduction rules:

$$\begin{array}{llll}(\beta) & \langle \lambda x.t|u :: e\rangle \to \langle u|\tilde{\mu} x.\langle t|e\rangle\rangle & (\mu) & \tilde{\mu} x.\langle x|e\rangle \to e, \text{ if } x \notin e \\ (\pi) & \langle \mu a.c|E\rangle \to [E/a]c & (\tilde{\mu}) & \mu a.\langle t|a\rangle \to t, \text{ if } a \notin t \\ (\sigma) & \langle t|\tilde{\mu} x.c\rangle \to [t/x]c & & \end{array}$$

These are the reductions considered by Polonovski in [22], with three provisos. First, the β-rule for the subtraction connective is not included. Second, in the π-rule, the co-term involved is an evaluation context E; this is exactly what characterizes the call-by-name restriction of $\overline{\lambda}\mu\tilde{\mu}$ [3]. Third, the naming of the rules is non-standard. Curien and Herbelin (and Polonovski as well) name rules π and σ as μ, $\tilde{\mu}$, respectively. The name μ has moved to the rule called *se* in [22]. By symmetry, the rule called *sv* by Polonovski is now called $\tilde{\mu}$. The reason for this change is explained by the spectrum of systems in Section 5: the rule we now call π (resp. μ) is the most general form of the rule with the same name in the system $\lambda\mathbf{J}$ (resp. $\lambda\mathbf{J^m}$), and therefore its name goes back to [14] (resp. [8], actually back to [24]).

Ludics is a Model for the Finitary Linear Pi-Calculus

Claudia Faggian[1] and Mauro Piccolo[2]

[1] PPS, CNRS-Paris 7
faggian@pps.jussieu.fr
[2] Universitá di Torino - PPS
piccolo@di.unito.it

Abstract. We analyze in game-semantical terms the finitary fragment of the linear π-calculus. This calculus was introduced by Yoshida, Honda, and Berger [NYB01], and then refined by Honda and Laurent [LH06].

The features of this calculus - asynchrony and locality in particular - have a precise correspondence in Game Semantics. Building on work by Varacca and Yoshida [VY06], we interpret π-processes in linear strategies, that is the strategies introduced by Girard within the setting of Ludics [Gir01]. We prove that the model is fully complete and fully abstract w.r.t. the calculus.

1 Introduction

In this paper we show a precise correspondence between the strategies of Ludics [Gir01] and the linear π-calculus [NYB01]. Ludics has been introduced by Girard as an abstract game semantical model; the strategies, which here we call linear strategies, can be seen as a refinement of Hyland-Ong innocent strategies [HO00]. The linear π-calculus is a typed version of asynchronous π-calculus introduced by Yoshida, Honda and Berger. We interpret π-processes of the finitary fragment of the calculus into linear strategies. We prove that the model is fully complete w.r.t. the calculus.

Our analysis makes explicit an exact correspondence between process calculi features and game-semantical notions, in particular between asynchrony and innocence. Moreover, the names discipline of Ludics exactly matches that of the internal π-calculus.

The Linear π-Calculus: The linear π-calculus has been introduced by N. Yoshida and K. Honda in order to study strong normalization [NYB03], information flow security [KHY00, YH05], and other interesting properties of the calculus. The typing is based on Linear Logic.

The typing has recently be refined by Honda and Laurent [LH06], which establish a precise correspondence with polarized proof-nets. Hence typed $\pi-$calculus, Linear Logic, proof-nets, linear strategies fit together as aspects of the same broader picture.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 148–162, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The main features of the linear π-calculus are the following (for an in-depth discussion of these aspects, and their significance in concurrent and distributed computation, we refer to [SW01]):

Asynchrony: the act of sending a datum and the act of receiving it are separate. In particular no process need to wait to send a datum.
Internal mobility: only private (*fresh*) names can be communicated (by an output action).
Locality: names received in input are only used as output (and dually names sent in output are only used for input).
Linearity: The linearity constraint gives a discipline over the use of names. In particular it states that each linear name must be used at most once in the process.

In this paper, we study the **finitary** fragment of linear π-calculus, i.e. the fragment of linear π that does not contain the recursion and replication operators.

Ludics is a Game model, developed as an abstraction of Linear Logic. The strategies are a linear version of Hyland-Ong innocent strategies.

A key role is played by the notion of name (address), which play the same role as that of process calculus channel in an internal calculus (Sangiorgi's $\pi - I$). As we observed in previous work [FP06], the discipline on names imposed in [Gir01] is closely related to that of internal π-calculus.

Some other features which are specific to the Ludics setting - and fundamental for our interpretation of πprocesses - is the existence of 'incomplete' strategies, which terminate with an error state, and the treatment of the additives (corresponding to a prefixed summation).

Here we only use the basic level of the Ludics setting. Here we do not explore the full architecture, which however appears of great interest in the study of process calculi. In fact, Ludics: (i) comes equipped with a build-in notion of observational equivalence, (ii) has an interactive definition of types.

We expect that our work can open the way for applying the general setting of Ludics to a semantical analysis of process calculi. On the other direction, we hope to have an insight, making possible to import techniques concerning parallel execution and non-determinism which are well developed in the study of concurrency. In current work (see Section 5), we are exploring a possible extension of linear strategies with non-determinism.

Strategies and Processes. In this paper, we highlight the following correspondence:
asynchrony – innocence (which we discuss in Section 4.1)
internal mobility – names discipline in Ludics
locality – alternating arena.

2 The Calculus

In this section we describe the finitary fragment of the linear π-calculus ([NYB01], [LH06]). We first discuss the untyped syntax and then introduce the typed setting.

2.1 Syntax and Reduction Rules

The syntax of the finitary linear π-calculus is given by the following grammar:

$$P ::= \; u(\boldsymbol{x}).P \mid \overline{u}(\boldsymbol{x})\, P \mid P|P \mid \nu\boldsymbol{x}\, P \mid 0$$

where is imposed that

internal mobility only private (fresh) names can be passed by an output action.

locality in $a(\boldsymbol{x}).P$ the names $\boldsymbol{x}$ are distinct and cannot be used as subject of an input action (and dually for $\overline{a}(\boldsymbol{x})\, P$).

Observe the two constructs (blocking input and asynchronous output) that mark the *asynchrony* of the calculus. The input $u(\boldsymbol{x}).P$ is blocking, i.e. the process waits for some input from the environment. The output $\overline{u}(\boldsymbol{x})\, P$ is the binding *asynchronous output* construct, defined in [NYB01]. The encoding in the standard π-calculus is the following:

$$\overline{u}(\boldsymbol{x})\, P =_{def} \nu\boldsymbol{x}(\overline{u}\langle\boldsymbol{x}\rangle|P)$$

This means that the continuation P can evolve in an independent way with respect to the output. If the output has no continuation (since the environment does not answer to it), we write $\overline{u}(\boldsymbol{x})$.

Operational Semantics. Reduction rules and structural congruence are those defined in [NYB03]. The main reduction rule is the following:

$$u(\boldsymbol{x}).P|\overline{u}(\boldsymbol{x})\, Q \longrightarrow \nu\boldsymbol{x}\,(P|Q) \tag{1}$$

which corresponds to the consumption of an asynchronous message by a receptor. To ensure the asynchrony of the output, the following rule is also added

$$P \longrightarrow P' \Rightarrow \overline{u}(\boldsymbol{x})\, P \longrightarrow \overline{u}(\boldsymbol{x})\, P' \tag{2}$$

Structural congruence is defined in term of the standard rules, extended with the following axioms (which allow to infer interaction under a prefixing output)

$$\overline{x}(\boldsymbol{u})\,\overline{y}(\boldsymbol{v})\, P \equiv \overline{y}(\boldsymbol{v})\,\overline{x}(\boldsymbol{u})\, P \;\; \text{if } x \notin \boldsymbol{v} \;\text{ and }\; y \notin \boldsymbol{u} \tag{3}$$

$$\overline{x}(\boldsymbol{u})\,(P|Q) \equiv (\overline{x}(\boldsymbol{u})\, P)|Q \;\; \text{if } \boldsymbol{u} \notin Q \tag{4}$$

$$\nu y\,\overline{x}(\boldsymbol{u})\, P \equiv \overline{x}(\boldsymbol{u})\,\nu y\, P \;\; \text{if } y \notin \{x, \boldsymbol{u}\} \tag{5}$$

2.2 The Typing System

We assume a countable set of names, ranged over by $u, v, x, y \ldots$.

A **type** is assigned to a name in order to specify its use (the assignment is written $a : T$ where a is a name and T is a type). In particular, the capability a name can have, i.e. input, output or match, is specified by the polarity of the type: negative

(T^-), positive (T^+), or neutral ($\updownarrow$). Moreover, the type of a name u disciplines also the data (names) which can be delivered using u. For example we write $u : \bigotimes_{i\in I} T_i$ to express that the channel u can be used in output mode to send an n-upla of names $\boldsymbol{x}$ where each x_i has type T_i. The syntax for the types is the following:

$$\begin{array}{rllll} T & ::= T^+ \text{ positive} & \mid T^- & \text{negative} & \mid \updownarrow \text{ neutral} \\ T^+ & ::= \mathbf{0} \text{ send error} & \mid \bigotimes_{i\in I} T_i^- & \text{output channel} & \\ T^- & ::= \top \text{ non-reception} & \mid \invamp_{i\in I} T_i^+ & \text{input channel} & \end{array}$$

Given a type T, its **dual** ($T^\perp$) is defined as $\top^\perp = 0$ and $(\invamp_{i\in I} T_i)^\perp = \bigotimes_{i\in I}(T_i)^\perp$.

A **type environment** (denoted by the letters $\Gamma, \Delta, \ldots$) gives a well-formed judgement on processes. It is a list of *distinct* names, each with a type assignment:

$$\Gamma = x_1 : T_1, \ldots, x_n : T_n$$

A type environment Γ can be thought of as a partial function from names to types. Hence we write $Dom(\Gamma)$ for the set of names that occur in Γ; if $\Gamma = x : T, \Delta$, we have $\Gamma(x) = T$. With a slight abuse of notation, given a type environment Γ we denote by $\Gamma = \Gamma^+, \Gamma^\updownarrow, \Gamma^-$ its partition in positive, negative and neutral types.

The following operation on type environments is introduced in order to put the environments together when performing parallel composition of processes. Intuitively, this operation takes the union of two type environment and matches the names which appear in both environments.

Definition 2.1 ($\odot$-operator). *Let Γ and Δ be two type environments such that, for all $x \in Dom(\Gamma) \cap Dom(\Delta)$ we have $\Gamma(x) = \Delta(x)^\perp$. $\Gamma \odot \Delta$ is the environment Ξ such that $Dom(\Xi) = Dom(\Gamma) \cup Dom(\Delta)$ and*

$$\Xi(x) = \begin{cases} \Gamma(x) & \text{if } x \notin Dom(\Delta) \\ \Delta(x) & \text{if } x \notin Dom(\Gamma) \\ \updownarrow & \text{otherwise} \end{cases}$$

Lemma 2.1. *$\odot$ is a partial commutative associative operator on type environments.*

Typing Rules. Typing judgements are in the form $P \triangleright \Gamma$ and the corresponding typing rules are given in 1. The (ZERO)-rule types 0, that is the termination signal given by the process to the environment: it can be viewed as an error state or a lack of answer by the process to a question given by the environment. The (TOP)-rule, on the other hand, types the environment inaction, i.e a lack of answer by the environment to a question given by the process: note that in the process $\overline{u}(x)$ we have the continuation empty (the empty space), and it corresponds to the environment inaction on the channel x or a lack of answer on channel x). (NEG) and (POS) ensure the linearity constraint on names. Note the polarity constraint given in the rule (NEG): this ensures the decomposition of a given process into sub-processes with one negative port. The (PAR)-rule ensures the well-definiteness properties of the parallel composition. The (RES)-rule

Table 1. Typing rules for the finitary π-calculus

$$\frac{}{0\rhd}\ (\text{ZERO}) \quad \frac{}{\rhd a:\top}\ (\text{TOP}) \quad \frac{P\rhd\Gamma \quad \pi(T)\in\{+,\updownarrow\}}{P\rhd\Gamma,x:T}\ (\text{WEAK})$$

$$\frac{P\rhd x_1:T_1^-,\dots,x_n:T_n^-,\Gamma}{\bar{u}(\boldsymbol{x})\ P\rhd u:\bigotimes_{i\in I}T_i^-,\Gamma}\ (\text{POS}) \quad \frac{P\rhd x_1:T_1^+,\dots,x_n:T_n^+,\Gamma^+}{u(\boldsymbol{x}).P\rhd u:\bigparr_{i\in I}T_i^+,\Gamma^+}\ (\text{NEG})$$

$$\frac{P\rhd\Gamma \quad Q\rhd\Delta}{P|Q\rhd\Gamma\odot\Delta}\ (\text{PAR}) \quad \frac{P\rhd\Gamma,x:T \quad \pi(T)\in\{-,\updownarrow\}}{\nu x\,P\rhd\Gamma}\ (\text{RES})$$

allows that only negative and neutral name can be restricted since they carry actions which expect their dual action to exist in the environment.

With respect to the given typing rules, we have the following results:

Proposition 2.1 (subject congruence). *If $P\rhd\Gamma$ and $P\equiv Q$, then $Q\rhd\Gamma$.*

Proposition 2.2 (subject reduction). *If $P\rhd\Gamma$ and $P\longrightarrow Q$, then $Q\rhd\Gamma$.*

Note. The purpose of the rule (TOP) is to state the name a to which the type $\top$ is associated.

In this sense, it is a way to report the environment inaction on a given channel x: it means that the environment knows the channel x but it does not use it. A more rigorous way to express this would be to annotate the negative inaction type with a name ($\top_x$), and making the rule (POS) more complicated.

The environment inaction should not be confused with process termination, which we denoted with 0.

Properties of the Typing System. The typing system guarantee the following properties:

Proposition 2.3. *Let $P\rhd\Gamma$. Then*

linearity: *for all $x\in Dom(\Gamma)$ such that $\Gamma(x)\neq\updownarrow$, x occurs at most once in P.*
subject reduction: *$P\longrightarrow^* Q$ then $Q\rhd\Gamma$*
strong confluence: *$P\longrightarrow Q_i$ ($i=1,2$) then either $Q_1\equiv Q_2$ or there exists R such that $Q_i\longrightarrow R$*

2.3 The Additive Structure

We (sketch how to) extend the calculus with two new constructs: the branching input and the selective output[1] (in Linear Logic, these constructs correspond to

[1] Our approach is closely inspired by [NYB03, VY06].

the additive connectives; in a functional programming language, they correspond respectively to the case and to the select constructs).

The *syntax* is the following:

$$P ::= \ a \bigwith_{i \in I} in_i(\boldsymbol{x}_i).P_i \mid \overline{a} in_j(\boldsymbol{x}_j)\, P_j \mid P|P \mid \nu \boldsymbol{x}\, P \mid 0$$

The construct $a \,\Phi_{i\in I}\, in_i(\boldsymbol{x}_i).P_i$ is the **branching input**. It corresponds to an external choice (a choice by the environment). We may see a as a multi-port channel: the process waits the environment to choose one of the components, and then evolves in the corresponding branch P_i. The reduction rule is

$$a \bigwith_{i \in I} in_i(\boldsymbol{x}_i).P_i | \bar{a} in_j(\boldsymbol{x}_j)\, Q_j \rightarrow \nu \boldsymbol{x}_j P_j | Q_j$$

Two different branches P_i and P_j represent different evolutions of the system, which are in *conflict*.

The construct $\overline{a} in_j(\boldsymbol{x}_j)\, P_j$ is the **selective output,** which corresponds to an internal choice. If we want to have determinism, we need to impose that this choice is unique.

The positive and negative *types* are now:

$T^+ ::= \mathbf{0}$ zero $\mid \bigoplus_{i\in I} \bigotimes_{k\in K_i} T_k^-$ selective output

$T^- ::= \top$ inaction $\mid \Phi_{i\in I} \bigparr_{k\in K_i} T_k^+$ branching input

The *typing rules* are as in table (1), where we replace the (NEG) and (POS) rules with the following ones:

$$\frac{P_j \rhd x_{1j} : T_{1j}, \ldots, x_{nj} : T_{nj}, \Gamma \quad j \in I}{\overline{u} in_j(\boldsymbol{x}_j)\; P_j \rhd u : \bigoplus_{i\in I} \bigotimes_{k\in K_i} T_{ki}, \Gamma}\ pos \qquad \frac{\forall i.P_i \rhd x_{1i} : T_{1i}, \ldots, x_{ni} : T_{ni}, \Gamma^+}{u \bigwith_{i\in I} in_i(\boldsymbol{x}_i).P_i \rhd u : \bigwith_{i\in I} \bigparr_{k\in K_i} T_{ki}, \Gamma^+}\ neg$$

All previous results (subject reduction, confluence, etc.) can be extended.

2.4 Refinements of the Typing System

The typing system can be refined by means of several constraints. In particular, Honda and Laurent establishes a hierarchy of classes of processes by adding constraints. The most relevant ones are acyclicity and sequentiality. We refer to [NYB01, NYB03, LH06] for the technical details.

Acyclicity: a process is deadlock-free if it never reduces to stuck configurations (an example of stuck configuration is given by the process $\nu a, b(a.\overline{b}|b.\overline{a})$: here we have a *cyclic* dependency between a and b that blocks further reductions). The acyclicity constraint is added to control the way the resources are used by processes in order to avoid deadlocks. In [NYB03] acyclic processes are proved to be strongly normalizing.

Sequentiality: a process is sequential if at each step of reduction has at most one active output i.e. if there are multiple open inputs, the process can answer to only one of them. In [NYB01] sequential processes are used to provide a fully abstract encoding of PCF and in [LH06] sequential process are used to characterize polarized proof nets.

Important Note. From now on *we assume the acyclicity constraint* in our typing rules since it guarantees the good behaviour of parallel composition. However, we *do not assume sequentiality.*

For this reason, in the model we use the variant of linear strategies which has been defined in [CF05] (there called **L-forests**) and which correspond to proves in Multiplicative Additive Linear Logic plus the Mix rule. If we add sequentiality, we exactly obtain the strategies defined in [Gir01] (there called **designs**).

3 The Model

In this section we reformulate the definitions given in [Gir01, CF05]. We use the same notion of strategy as in Ludics, but types are here directly interpreted as arenas (as in [Fag05]), to make more direct and explicit the construction defined in [Gir01]. The full interactive construction of types which belongs to Ludics would be possible -and very interesting- but it would be much less compact.

Our presentation is non-standard also in that: we use process calculus language to help the intuition. Moreover, we highlight the fact that a strategy is an event structure, and exploit the notion of conflict to describe the additive structure.

3.1 Name and Actions

Let us consider an action in the sense of π-calculus: $u(\boldsymbol{x})$. To specify an action we need three data: the name u used as a channel, the set $\boldsymbol{x}$ of names that can be communicated on that channel, and the way the channel u can be used (input, output, etc.), i.e. its polarity. The same three data are specified by actions as defined in [Gir01]. Moreover, there is a coding, which is in many respect similar to that of De Bruijn notation.

Names and Actions in Ludics. A notion which is crucial both in the π-calculus and in Ludics is that of *name* (in [Gir01] names are called *addresses*). A name can be seen as a channel, which is used to send or receive *data which are names themselves.*

Let the set $\mathcal{N}$ of the **names** (ranged over by $u, v, x, y, \dots$) be the strings of natural numbers. Given a name $u \in N$, the set $\{u.i | i \in \mathbb{N}\}$ corresponds to all data (names) that can be communicated using u.

This leads to an intuitive notion of action: an action is a pair (u, I), with $I \subset \mathbb{N}$, which specifies a name u used as channel and the set of names that will be communicated on that channel. By construction, to characterize such a set, it is enough to give the set of suffixes I that will be added to the name u.

Summing up, the action $a = (u, I)$ can communicate the names $u.i$, with $i \in I$. We write $name(a)$ for u.

Example. Consider the process $u(x, y).\bar{x}(z)$ We can use the following renaming: $x := u.1, y := u.2, z := u.1.1$ We write $u(x, y)$ as $(u, \{1, 2\})$, and $\bar{x}(z)$ as $(u.1, \{1\})$.

Polarities. A *polarized action* is an action a together with a polarity, positive (a^+) or negative (a^-), which specifies the capability of the name, that is if the channel is used for sending in output (positive) or receiving in input (negative).

Zero. The set of actions is extended with a special action, denoted by $\dagger$, which indicates termination with an error. By definition, the action $\dagger$ is positive.

We will interpret the zero of process calculus with $\dagger$, as 0 establishes the termination of the process. Intuitively, it states that player has no answer to an opponent move; in this senses it represents an *error* state.

Enabling Relation. We say that the action $a = (u, I)$ generates the names $u.i$, written $a \vdash u.i$. Given two actions a, b, we write $a \vdash b$ if $a \vdash name(b)$. We call the relation $a \vdash b$ between actions *enabling* relation.

The enabling relation establishes dependency between the actions. This leads us to the notion of arena.

3.2 Arenas

In this section we define the notion of arena; arenas will interpret types. An arena is given by an *interface* and a *forest of polarized actions*, where the forest is induced by the enabling relation, which establishes dependency between the actions.

The interface of an arena specifies the names on which a strategy on that arena can communicate with the rest of the world. An **interface** $\mathcal{I}$ is a set of initial names $\mathcal{I} = \{u, v, \ldots\}$ together with a polarity function $\pi : \mathcal{I} \to \{+, -\}$. We impose that the set of negative names is either empty or a singleton.

Given an interface $\mathcal{I}$, an **arena** on $\mathcal{I}$ is a set of polarized actions together with the enabling relation $b \vdash c$ (defined above), such that it satisfies the following conditions:

- the enabling relation is arborescent;
- The action c is minimal (denoted $\vdash c$) iff $name(c) \in \mathcal{I}$;
- the polarity of each minimal action c is that specified by the interface for $name(c)$; if $a \vdash b$ then the actions a and b have opposite polarity.

Given an arena of interface $\mathcal{I}$, we partition the trees according to the name of the root. If A is the set of all trees whose root uses the name u, we will write $u : A$. We will denote an arena of interface $\mathcal{I} = \{u_1, \ldots, u_k\}$ by $\Gamma = u_1 : A_1, \ldots, u_k : A_k$.

The polarity induces a partition of the names in the interface, and hence of the arena, into positive names (the outputs) and negative names (the inputs). With a slight abuse of notation, we write $\Gamma = \Gamma^+, \Gamma^-$. An arena is said *positive* if Γ^- is empty (all names are positive), negative otherwise.

Arena Constructions. We consider the following constructions on arenas.

Empty. The empty forest is an arena (positive or negative, according to the interface). We indicate the positive empty arena with $\Gamma : 0$ and the negative one with $u : \top$.

Dual. If $u : A$ is an arena, its dual $u : A^{\perp}$ is obtained by changing the polarity of u. Hence the actions are the same but the induced polarity is inverted.

Product. Let $\{u.i : A_i | i \in I\}$ be a family of arenas of negative (resp. positive) polarity. We define the arena $u : \prod_{i \in I} A_i$ by rooting all A_i on top of the action (u, I). If the polarity of u is positive (resp. negative), then we write $\prod_{i \in I} A_i = \bigotimes_{i \in I} A_i$ (resp. $\prod_{i \in I} A_i = \bigparr_{i \in I} A_i$).

Sum. Let $\{u : A_i | i \in I\}$ be a family of arenas on the same interface and such that all roots are pairwise disjoint (hence, if (u, I) and (u, J) are roots of two distinct arenas, then $I \neq J$). We define the arena $u : \sum_{i \in I} A_i$ as the union of all the forests. If the polarity of the root is positive (resp. negative), then we write $\sum_{i \in I} A_i = \&_{i \in I} A_i$ (resp. $\sum_{i \in I} A_i = \bigoplus_{i \in I} A_i$).

3.3 Strategies

A strategy is here a forest of occurrences of actions. On the occurrences of actions is defined an order, which we denote by $\leq$. We write $e <_1 e'$ if the node e is immediate predecessor of e'.

Definition 3.1 (strategy). *Let Γ be an arena. A strategy σ on the arena Γ (written $\sigma : \Gamma$) is given by a forest $\langle E, \leq \rangle$, where E is a set of nodes, and $\leq$ is an arborescent partial order; the nodes are labelled by polarized actions*[2] *in $\Gamma \cup \{\dagger\}$; the polarity and the name of a node (written $\pi(e)$ and $name(e)$) are those of the labelling action. Formally, there is a labelling function $\lambda : E \to \Gamma \cup \{\dagger\}$ which satisfies the following conditions.*

justification: *For each node e, its labelling action is either initial ($\vdash \lambda(e)$) or there exits a preceding node $e' < e$ such that $\lambda(e') \vdash \lambda(e)$.*

alternance: *If $e <_1 e'$ then they have opposite polarity*

innocence: *If $e <_1 e'$ and e is positive, then $\lambda(e) \vdash \lambda(e')$.*

positivity: *If e is maximal (i.e. there exists no e' such that $e < e'$), then e is positive. Moreover, if Γ is positive then the strategy is non empty.*

We denote the empty strategy with $\emptyset$ and the strategy whose unique action is a $\dagger$ with $\mathcal{D}ai$.

3.4 Linear Strategies

In this section we describe linear strategies (as defined in [CF05]) which can be seen as an abstraction of Multiplicative-Additive Linear Logic with Mix.

Let us first introduce apart the definition of multiplicative strategy. This is a special case of linear strategy (which is enough to understand most of this paper). A strategy is **multiplicative** if no two labels use the same name. The more general definition below takes into account the repetitions due to the additive structure.

[2] Hence nodes are occurences of actions.

Let $\sigma : \Gamma$ be a strategy. We call **cell** a set of nodes that have the same name and the same predecessor. We call **positive** (resp. negative) a cell whose nodes are occurrences of positive (resp. negative) actions.

Starting from the notion of cell, we define on the nodes of a strategy the relation # of **conflict** as the smallest binary symmetric relation which satisfy the following conditions:

immediate conflict $e_1 \# e_2$ if they are distinct and belongs to the same cell;
inheritance if $k_1 < k_2$ and $k_1 \# k_3$ then $k_2 \# k_3$.

Definition 3.2 (linear strategy). *A strategy $\sigma : \Gamma$ is linear if it satisfies the following conditions*

linearity *for all distinct occurrences of actions k_1, k_2, if $name(k_1) = name(k_2)$ then $k_1 \# k_2$*
determinism *All non singleton cells are negative (i.e., every time there is a choice -expressed by the conflict relation- the choice belongs to Opponent).*

Observe that a linear strategy is multiplicative if all its cells are singletons, i.e. the conflict relation # is empty.

3.5 Totality

Typed processes will be interpreted into linear strategies which are total.

Definition 3.3 (totality). *A linear strategy $\sigma : \Gamma$ is total if, for each* negative *action c in the arena:*

1. *if $\vdash c$, then it occurs in σ (i.e. there is a node $e \in \sigma s.t.$ $\lambda(e) = c$);*
2. *if $b \vdash c$, each occurrence of b in σ is followed by an occurrence of c (i.e., for each $e \in \sigma$, $\lambda(e) = b \Longrightarrow \exists e' \in sigma$ s.t. $e <_1 e'$ and $\lambda(e') = c$).*

3.6 Composition of Strategies

Given two strategies σ_1, σ_2, we can compose them if they have compatible interfaces i.e. there is a common name that appear in both interfaces with opposite polarity.

A **cut net** is a finite set $\mathcal{R} = \{\sigma_1, \ldots, \sigma_n\}$ of strategies such that (i) each name occurs at most in two interfaces, once as a positive name and once as a negative name, (ii) the graph which has as vertexes the interfaces and an edge connecting any two interfaces with a common name is acyclic.

The interface of the cut net is the interface induced by the names of the interfaces which are not a cut. For example, given a cut net whose strategies have interface u^+, a^+ and a^-, b^+, c^+ and b^-, d^+, the cut net has interface u^+, c^+, d^+.

We do not give details here on the composition of strategies, whose definition is given in [Gir01]. The result of composing the strategies in a cut-net $\mathcal{R}$ is called normal form of $\mathcal{R}$, which we denote by $\mathcal{R}^*$. This is a linear strategy having as interface the interface of the cut net.

3.7 Set of Independent Strategies

$\mathcal{S} = \{\sigma_i : \Gamma_i | i \in I\}$ is a set of **independent strategies** if (i) $\mathcal{D}ai \in \mathcal{S}$ and any two distinct strategies have disjoint interfaces. For example, two strategies $\sigma_1 : \{u^+, v^+\}$ and $\sigma_2 : \{z^-, t^+\}$ are independent, while two strategies with interfaces $\{u^+, v^+\}$ and $\{v^-, t^+\}$ are not.

A set of independent strategies $\mathcal{S} = \{\sigma_i : \Gamma_i | i \in I\}$ has interface $\Delta = \cup \Gamma_i$ and we write $\mathcal{S} : \Delta$.

Composition. Given two set $\Sigma = \{\sigma_i : \Gamma_i | i \in I\}, \Psi = \{\psi_j : \Delta_j | j \in J\}$ of independent strategies, we define their composition $\Sigma; \Psi$ as follows. Given $\mathcal{C} = \Sigma \cup \Psi$, we obtain a new set of independent strategies by partitioning the set $\mathcal{C}$ into cut-nets, according to the interfaces: we consider the graph whose vertices are the non-empty interfaces, and draw an edge between interfaces which contain dual names. The operation is only defined if the graph is acyclic, and each name appears at most in two distinct interfaces, with opposite polarity. The partition of the graph into connected components induces a partition of the strategies into cut nets $\mathcal{R}_1, \ldots, \mathcal{R}_n$; we have $\Sigma; \Psi = \{\mathcal{R}_1^*, \ldots, \mathcal{R}_n^*\}$, which is a set of independent strategies.

4 The Interpretation

In this section , we interpret types with arenas, type environments with sets of arenas, and processes with linear strategies. For clarity, here we only show how to deal with the multiplicative case. To deal with the additive case is a straightforward extension (but the syntax becomes harder to read).

Interpretation of Types

$$\begin{aligned} [\![u : \top]\!] &= u : \top \\ [\![\Gamma : \mathbf{0}]\!] &= \Gamma : 0 \\ [\![u : \bigotimes_{i \in I} T_i]\!] &= u : \bigotimes_{i \in I} [\![u.i : T_i]\!] \\ [\![u : \Omega_{i \in I} T_i]\!] &= u : \Omega_{i \in I} [\![u.i : T_i]\!] \end{aligned}$$

Interpretation of type environments. The interpretation of a type environment is the juxtaposition of the interpretation of each type whose polarity is non neutral.

Interpretation of Processes. A typed process is interpreted into a set of independent strategies. The most interesting case is that of a typed process $P \triangleright \Gamma$, where Γ has at most a negative type; this process will be interpreted into a strategy σ on the arena $[\![\Gamma]\!]$.

We use the following constructions on sets of independent strategies (see [CF05]) to give a semantics respectively to prefixed input, asynchronous output and the scope operator.

boxing: given a set Σ of independent strategies on $u.1 : T_1^+, \ldots, u.n : T_n^+, \Gamma^+$ and a negative action $k = (u, I)^-$, the strategy $k.\Sigma : u : \bigparr_i T_i^+, \Gamma^+$ is obtained by prefixing the union of all strategies in Σ with k.

rooting: given a set Σ of independent strategies and a positive action $k = (u, I)^+$, the set of strategies $k \circ \Sigma$ is that obtained by adding a node of label (u, I), and making it precede only the nodes whose name is generated by (u, I) (where $(u, I) \vdash u.i$). Observe that, since the nodes of name $u.i$ are negative, they are roots. Hence we are prefixing with a node of label (u, I) all strategies on $u.i : T_i^-$.

restriction given a set of independent strategies Σ we obtain $\Sigma \setminus u$ by erasing all trees whose root has name u.

Let $P \triangleright \Gamma$. The interpretation is then defined in the following way

1. $[\![0 \triangleright _]\!] = \{Dai\}$
2. $[\![_ \triangleright x : \top]\!] = \{\emptyset, Dai\}$
3. $[\![\overline{u}(\boldsymbol{x})\, P \triangleright u : \bigotimes_{i \in I} T_i^-, \Gamma]\!] = (u, I) \circ [\![P[x_i := u.i] \triangleright u.1 : T_1^- \ldots u.n : T_n^-, \Gamma]\!]$ where $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ and $I = \{1, \ldots, n\}$
4. $[\![u(\boldsymbol{x}).P \triangleright u : \bigparr_{i \in I} T_i^+, \Gamma^+]\!] = \{(u, I).[\![P[x_i := u.i] \triangleright u.1 : T_1^+ \ldots u.n : T_n^+, \Gamma^+]\!]$ $Dai\}$
5. $[\![\nu x\, P \triangleright \Gamma]\!] = [\![P \triangleright \Gamma, x : T]\!] \setminus x$
6. $[\![P_1 | P_2 \triangleright \Gamma \odot \Delta]\!] = [\![P_1 \triangleright \Gamma]\!]; [\![P_2 \triangleright \Delta]\!]$ supposing that $P_1 \triangleright \Gamma$ and $P_2 \triangleright \Delta$

4.1 Innocence and Asynchrony

An innocent strategy specifies what is Player answer to any Opponent move without having any information on the way in which Opponent plays. This means that after a Player move, we only know which Opponent moves are enabled, but we do not know if and in which order they will be played. Technically, each Opponent move immediately follows the Player move which enables it.

In our strategies, the causal order between actions takes into account the constraints which are given by the typing (the arena). Thus, for example, if the name x is generated by the action $u(x)$, any action using the channel x causally depends on $u(x)$ also in the strategy. Moreover, the strategy can introduce additional order. However, if the strategy is innocent, the extra order is only on pairs (Opponent move, Player move).

Innocence can be seen as the game-semantical counterpart of asynchrony and corresponds to the fact that in the asynchronous π-calculus, the input (Opponent move/negative action) is prefixing and blocking, while the output (Player move/positive action) is not. For output we use rooting: the only constraints we have are those fixed by the arena. Boxing instead introduces additional order.

The correspondence between innocent strategies and processes in an asynchronous π calculus appears clearly when analyzing the normal forms (section 4.4).

4.2 Full Completeness

Proposition 4.1 (Validity). *If $P \triangleright \Gamma$ is a process, its interpretation $[\![P]\!]$ on $[\![\Gamma]\!]$ is a set of independent strategies, where each strategy is total (on the corresponding arena).*

Proposition 4.2 (Correctness). *The interpretation satisfies the following. If $P \equiv Q$ then $[\![P]\!] = [\![Q]\!]$*
If $P \longrightarrow Q$ then $[\![P]\!] = [\![Q]\!]$

Proposition 4.3 (Completeness). *Let $\Gamma = \Gamma^-, \Gamma^+$ be the arena interpreting a typing environment Γ^-, Γ^+, where Γ^- is either empty or a singleton. If σ is a linear strategy on that arena, and σ is total, then σ is the interpretation of a process $P \triangleright \Gamma^-, \Gamma^+$.*

We sketch the proof, only dealing with the multiplicative case.

Proof. We associate to the strategy the typing derivation of a process. The proof is by induction on the size $|\sigma|$ of σ, where the size of a strategy is the number of its actions which are not $\dagger$.

$|\sigma| = 0$ **and** σ **negative.** Being negative, the interface contains a negative name, u, corresponding to the negative arena $u : A$. As σ is the empty strategy, for it to be total the arena $u : A$ must be empty. This case corresponds to the TOP rule (possibly followed by weakening).

$|\sigma| = 0$ **and** σ **positive.** We have that $\sigma = \mathfrak{D}ai$; this correspond to the zero rule (possibly followed by weakening).

$|\sigma| > 0$ **and** σ **negative.** This case correspond to the NEG rule. We have that $\sigma = (u, I).\sigma'$ is a total strategy on the arena $u : A, \Gamma^+$, where $A = \Gamma_I A_i$ and σ' is a total *positive* strategy on the arena $u.i : A_i, \ldots \Gamma^+$. To check totality, we observe that the Positivity condition implies that σ' is non empty.

$|\sigma| > 0$ **and** σ **positive.** We have that σ is a forest of positive strategies. Let us analyze each single connected component (each tree), on the opportune arena (the arena containing the names used by the strategy). Putting them together will correspond to the PAR rule (possibly followed by weakening).

Assume that $\sigma = (u, I). \cup_{i \in I} \sigma_i$. This is a strategy on the arena $u : A, \Gamma^+$, where $A = \otimes_I A_i$. All the addresses used in each σ_i are distinct, hence we can partition Γ^+ into $\Gamma_1^+, \ldots, \Gamma_k^+$ according to the names of the actions which are initial in each σ_i. Each σ_i is a total negative strategy on the arena $u.i : A_i, \Gamma_i^+$.

4.3 Full Abstraction

By using the same technique as in [NYB03], we have the following result of full abstraction, where the notion of operational equivalence is the *typed weak bisimilarity* ($\approx$) defined in [NYB03].

Proposition 4.4 (full abstraction). $P \triangleright \Gamma \approx Q \triangleright \Gamma \iff [\![P \triangleright \Gamma]\!] = [\![Q \triangleright \Gamma]\!]$

Let us sketch the proof. All typed processes are strongly normalizing, hence we can define two processes to be equivalent if they reduce to the same normal form, up to structural congruence. Such an equivalence is the same as $\approx$, and we can take the (unique) term in normal form as canonical representative in each class of π-terms. The key result is the following lemma.

Lemma 4.1. *Let $P \triangleright \Gamma$ and $Q \triangleright \Gamma$ be in normal form. $P \approx Q$ if and only if $P \equiv Q$.*

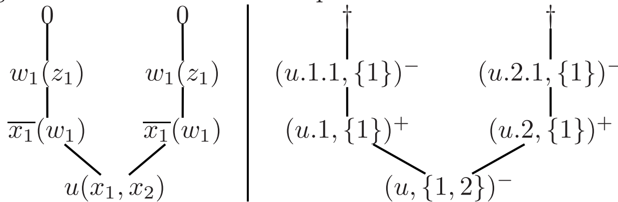
4.4 Normal Forms

In [NYB03] is given a canonical characterization of normal forms: each term in normal form can be written as a parallel composition of sub-processes which have a Bohm-tree like structure. These trees exactly correspond to Ludics strategies.

Example. Let us consider two normalized terms which are structurally equivalent (both are typed on $\Gamma = u : (\otimes(⅋T_1^+))\ ⅋\ (\otimes(⅋T_2^+)))$:

$$u(x_1, x_2).\overline{x_1}(w_1)\,\overline{x_2}(w_2)\,w_1(z_1).0|w_2(z_2).0 \equiv u(x_1, x_2).\overline{x_1}(w_1)\,w_1(z_1).0|\overline{x_2}(w_2)\,w_2(z_2).0$$

By reordering the actions performed by the two processes according to the Labelled Transition System, we obtain the tree on the left hand side below. On the right hand side, instead, we have the interpretation as linear strategy on the right: there is an exact correspondence.



5 Discussion and Future Work

We have shown a precise correspondence between the finitary fragment of the linear π-calculus [NYB01, NYB03] and the linear strategies introduced by Girard in the setting of ludics [Gir01].

Building on this core, we aim at extending the calculus and the model with non-determinism, recursion and replication, by following the approach proposed by [VY06].

Moreover, it is possible to use the full architecture of Ludics, and in particular the interactive constructions on types, and we are interested in exploring also this direction.

Non Determinism. In current work [FP07] we extend the calculus and the model with internal choice, i.e. with non determinism.

On one side, we add to the calculus a τ-prefixed sum ($\sum_i \tau.P_i$)

On the other side, the set of actions is extended with neutral actions, and a non deterministic choice is modeled by a cell of neutral actions. Non-determinism has the same behavious as the additive structure, but the actions labelling the cell are "silent".

Recursion and Replication. In this paper we have worked a finitary version of the linear π-calculus, in order to establish a correspondence with the existing setting of Ludics, as defined in [Gir01]. However, we expect to be able to extend the model both wiht replication (by using techniques similar to those which are developped in [VY06]) and with recursion.

References

[CF05] Curien, P.-L., Faggian, C.: L-nets, strategies and proof-nets. In: CSL 05 (Computer Science Logic). LNCS, Springer, Heidelberg (2005)

[Fag05] Faggian, C.: Linear logic games: Sequential and parallel. draft (2005)

[FP06] Faggian, C., Piccolo, M.: A graph abstract machine describing event structure composition. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, Springer, Heidelberg (2006) (short paper)

[FP07] Faggian, C., Piccolo. M.: Event structures and linear strategies. (submitted)

[Gir01] Girard, J.-Y.: Locus solum. Mathematical Structures in Computer Science 11, 301–506 (2001)

[HO00] Hyland, M., Ong, L.: On full abstraction for PCF. Information and Computation (2000)

[KHY00] Vasconcelos, V., Honda, K., Yoshida, N.: Secure information flow as typed process behaviour. In: Smolka, G. (ed.) ESOP 2000 and ETAPS 2000. LNCS, vol. 1782, Springer, Heidelberg, Extended abstract (2000)

[LH06] Laurent, O., Honda, K.: An exact correspondence between a typed pi-calculus and polarised proof-nets. draft (2006)

[NYB01] Honda, K., Yoshida, N., Berger, M.: Sequentiality and the pi-calculus. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, Springer, Heidelberg, Extended abstract (2001)

[NYB03] Honda, K., Yoshida, N., Berger, M.: Strong normalisation in the pi-calculus. Journal of Information and Computation, full version (2003)

[SW01] Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)

[VY06] Varacca, D., Yoshida, N.: Typed event structures and the pi-calculus. In: MFPS (2006)

[YH05] Yoshida, N., Honda, K.: Noninterference through flow analysis. Journal of Functional Programming, revised (2005)

Differential Structure in Models of Multiplicative Biadditive Intuitionistic Linear Logic

(Extended Abstract)

Marcelo P. Fiore

Computer Laboratory
University of Cambridge

Abstract. In the first part of the paper I investigate categorical models of multiplicative biadditive intuitionistic linear logic, and note that in them some surprising coherence laws arise. The thesis for the second part of the paper is that these models provide the right framework for investigating differential structure in the context of linear logic. Consequently, within this setting, I introduce a notion of creation operator (as considered by physicists for bosonic Fock space in the context of quantum field theory), provide an equivalent description of creation operators in terms of creation maps, and show that they induce a differential operator satisfying all the basic laws of differentiation (the product and chain rules, the commutation relations, *etc.*).

1 Introduction

Recent developments in the model theory of linear logic, starting with the work of Ehrhard [6,7], have uncovered a variety of models with differential structure. Examples include Köthe sequence spaces [6], finiteness spaces [7], the relational model, generalised species of structures [11,12], interaction systems [15], and complete semilattices [4]. This differential structure manifests itself as differential operators. In this context, a differential operator is a natural linear map

$$!A \multimap B \longrightarrow !A \multimap A \multimap B \tag{1}$$

that, when embedded as a map

$$A \Rightarrow B \longrightarrow A \Rightarrow (A \multimap B) \tag{2}$$

in the !-Kleisli category, enjoys the properties and satisfies the laws of differentiation. Intuitively, such an operator D provides a linear approximation $D[f]x : A \multimap B$ for every function $f : A \Rightarrow B$ at any point $x : A$.

The algebra underlying these models has also been investigated recently. Ehrhard and Regnier [8], isolated local-additive and commutative bialgebraic-exponential structure and explained, amongst other things, how they support

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 163–177, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the product rule. Blute, Cockett, and Seely [4], considered local-additive and exponential structure further supporting the chain rule. A common feature of these two approaches is that they take the local-additive structure, which allows morphisms to be added and is the minimal expression of linear-algebraic structure, as primitive. However, since local-additive structure in the presence of product structure is equivalent to biproduct structure, one may instead take as primitive the latter; which, furthermore, has the added bonus of inducing commutative bialgebraic-exponential structure. This is the viewpoint advocated here. It leads to the consideration of models of multiplicative biadditive intuitionistic linear logic, in which the additive structures (given by product and coproduct) coincide (as a biproduct), and to the thesis that these provide the right framework for investigating differential structure in the context of linear logic.

The present work is close in spirit to that of Blute, Cockett, and Seely [4] on differential categories, especially their Section 4. For latter comparison, I now highlight the relevant parts of their development. A notion of differential operator essentially as in (1) is introduced (see [4, Definition 2.3]). This is so that, for instance, the induced differential operator as in (2) satisfies the usual product and chain rules. Differential operators are shown to be in correspondence with so-called deriving transformations of the form

$$\partial : \; A \otimes {!A} \longrightarrow {!A} \tag{3}$$

(see [4, Definition 2.5 and Proposition 2.6]). Moreover, these are further seen to correspond to certain natural maps

$$\eta : \; A \longrightarrow {!A} \tag{4}$$

(see [4, Definition 4.11 and Theorem 4.12]).

Without loss of generality, my analysis of differential structure starts with the consideration of operators as in (3). These I call creation operators; as, interpreting the exponential as the bosonic Fock space construction [5], that models quantum systems of many identical non-interacting particles, they intuitively correspond to operators modelling particle creation. Indeed, categorical models of multiplicative intuitionistic linear logic come equipped with a canonical notion of annihilation operator

$$\propto : \; {!A} \longrightarrow A \otimes {!A}$$

with respect to which creation operators are shown to satisfy the commutation relations (see *e.g.* [14]). The above forms for creation and annihilation operators is non-standard; the standard forms are derivable.

The concept of creation operator given in this paper is novel and differs from that of deriving transformation mentioned above. This is clearly seen by comparing Theorem 4.1 below, which establishes a bijective correspondence between creation operators and certain natural maps as in (4), that I call creation maps, and [4, Corollary 4.13], which provides the corresponding result for deriving transformations. A crucial difference between the axiomatisations is that the one provided here, besides being sharper, involves an axiom describing the interaction between the differential structure and the monoidal strength of the

exponential. The present axiomatisation of creation maps has been directly influenced by and developed through a thorough analysis of the differential structure of generalised species of structures [10,11], which is a bicategorical generalisation of that of the relational model of linear logic.

Organisation and Contribution of the Paper. Section 2 provides basic background on biproduct structure. The emphasis there is on giving an algebraic presentation, analysing some of its consequences (importantly commutative bialgebraic structure), and then characterising it in terms of enrichment. I guess that these results are folklore. However, I do not know references for them. In Section 3, I define categorical models of multiplicative biadditive intuitionistic linear logic to be models of multiplicative intuitionistic linear logic, as have been considered in the literature, equipped with biproduct structure compatible with the monoidal structure. This directly induces commutative bialgebraic-exponential structure. More surprisingly, I note that in these models some unexpected coherence laws arise. These are important for the analysis of differential structure carried over in Section 4. As mentioned above, differential structure is first analysed in terms of creation operators, for which the commutation relations with respect to a canonical notion of annihilation operator hold. Subsequently, creation operators are characterised in terms of the simpler notion of creation maps. These are shown to induce differential operators satisfying all the basic laws of differentiation. Finally, Section 5 concludes with general remarks and prospects for further work.

2 Biproduct Structure and Enrichment

Biproduct Structure. I give an algebraic presentation of biproduct structure, both on categories and on monoidal categories. This is the key to the modelling of biadditive structure in models of linear logic.

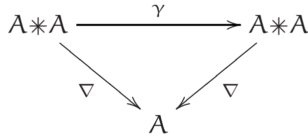
Definition 2.1. *A* biproduct structure *on a category is given by a symmetric monoidal structure* $(\mathbb{I}, \ast)$ *on it together with natural transformations*

$$\begin{array}{ccccc} \mathbb{I} & \xrightarrow{\;\mathsf{u}\;} & A & \xrightarrow{\;\mathsf{n}\;} & \mathbb{I} \\ A \ast A & \xrightarrow{\;\nabla\;} & A & \xrightarrow{\;\Delta\;} & A \ast A \end{array}$$

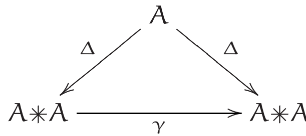
such that:

1. (A, u, ∇) *is a commutative monoid.*

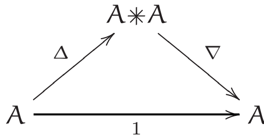
$$\begin{array}{ccccc} \mathbb{I} \ast A & \xrightarrow{\mathsf{u} \ast 1} & A \ast A & \xleftarrow{1 \ast \mathsf{u}} & A \ast \mathbb{I} \\ & \searrow^{\cong} & \downarrow{\scriptstyle \nabla} & {}^{\cong}\swarrow & \\ & & A & & \end{array} \qquad \begin{array}{ccc} A \ast A \ast A & \xrightarrow{\nabla \ast 1} & A \ast A \\ {\scriptstyle 1 \ast \nabla}\downarrow & & \downarrow{\scriptstyle \nabla} \\ A \ast A & \xrightarrow[\nabla]{} & A \end{array}$$



2. (A, n, Δ) *is a commutative comonoid.*



Definition 2.2. *A biproduct structure is* degenerate *whenever the following further law*



is satisfied.

The terminology of Definition 2.1 is justified by the following result.

Proposition 2.1. *In a category with biproduct structure* $(\mathbb{I}, *; u, \nabla; n, \Delta)$ *the following hold.*

1. $\mathbb{I}$ *is a zero object; that is, it is both initial and terminal.*
2. *The diagram*

$$A \xrightarrow{\cong} A * \mathbb{I} \xrightarrow{1 * u} A * B \xleftarrow{u * 1} \mathbb{I} * B \xleftarrow{\cong} B$$

   is a coproduct.
3. *The diagram*

$$A \xleftarrow{\cong} A * \mathbb{I} \xleftarrow{1 * n} A * B \xrightarrow{n * 1} \mathbb{I} * B \xrightarrow{\cong} B$$

   is a product.

That $\mathbb{I}$ is initial and Proposition 2.1(2) follow from Definition 2.1(1); dually, that $\mathbb{I}$ is terminal and Proposition 2.1(3) follow from Definition 2.1(2).

Corollary 2.1. *In a category with biproduct structure* $(\mathbb{I}, \circledast; u, \nabla; n, \Delta)$*, the natural transformations* u*,* ∇*,* n*,* Δ *are monoidal; that is,*

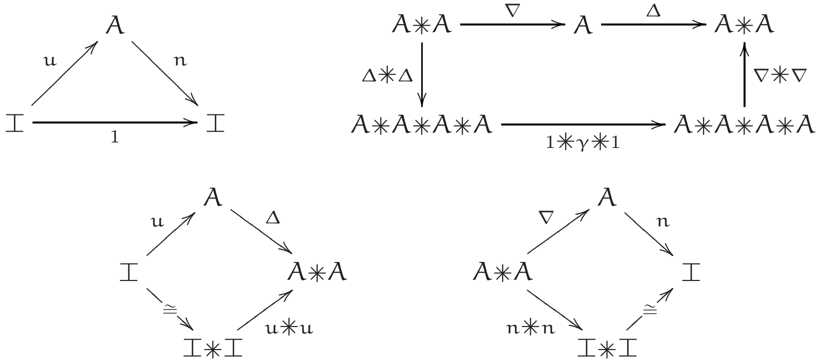
$$\begin{aligned} u_{\mathbb{I}} &= n_{\mathbb{I}} = 1_{\mathbb{I}} \\ u_{A \circledast B} &= \mathbb{I} \xrightarrow{\cong} \mathbb{I} \circledast \mathbb{I} \xrightarrow{u_A \circledast u_B} A \circledast B \\ n_{A \circledast B} &= A \circledast B \xrightarrow{n_A \circledast n_B} \mathbb{I} \circledast \mathbb{I} \xrightarrow{\cong} \mathbb{I} \end{aligned}$$

and

$$\begin{aligned} \nabla_{\mathbb{I} \circledast \mathbb{I}} &= \mathbb{I} \circledast \mathbb{I} \xrightarrow{\cong} \mathbb{I} \\ \nabla_{A \circledast B} &= A \circledast B \circledast A \circledast B \xrightarrow{1 \circledast \gamma \circledast 1} A \circledast A \circledast B \circledast B \xrightarrow{\nabla_A \circledast \nabla_B} A \circledast B \\ \Delta_{\mathbb{I} \circledast \mathbb{I}} &= \mathbb{I} \xrightarrow{\cong} \mathbb{I} \circledast \mathbb{I} \\ \Delta_{A \circledast B} &= A \circledast B \xrightarrow{\Delta_A \circledast \Delta_B} A \circledast A \circledast B \circledast B \xrightarrow{1 \circledast \gamma \circledast 1} A \circledast B \circledast A \circledast B \end{aligned}$$

It is important for our latter development to note that biproduct structure is equivalent to commutative bialgebraic structure.

Proposition 2.2. *In a category with biproduct structure* $(\mathbb{I}, \circledast; u, \nabla; n, \Delta)$*, the commutative monoid and comonoid structures* $(u, \nabla; n, \Delta)$ *form a commutative bialgebra; that is,* u *and* ∇ *are comonoid homomorphisms and, equivalently,* n *and* Δ *are monoid homomorphisms.*



Enrichment. I now recall the characterisation of biproduct structure in the context of enrichment.

Let **Mon** (**CMon**) be the symmetric monoidal category of (commutative) monoids with respect to the universal bilinear tensor product. Recall that **Mon**-categories (**CMon**-categories) are categories all of whose homs $[A, B]$ come equipped with a (commutative) monoid structure $(0_{A,B}, +_{A,B})$ such that composition is strict and bilinear; that is,

$$0_{B,C}\, f = 0_{A,C} \qquad \text{and} \qquad f\, 0_{C,A} = 0_{C,B}$$

for all $f : A \longrightarrow B$, and

$$g\,(f + f') = g\,f + g\,f' \qquad \text{and} \qquad (g + g')\,f = g\,f + g'\,f$$

for all $f, f' : A \longrightarrow B$ and $g, g' : B \longrightarrow C$.

Proposition 2.3. *The following are equivalent.*

1. *Categories with biproduct structure.*
2. **Mon**-*categories with (necessarily enriched) finite products.*
3. **CMon**-*categories with (necessarily enriched) finite products.*

The enrichment of categories with biproduct structure is given by *convolution* (see *e.g.* [21]) as follows:

$$0 = (A \xrightarrow{n} \mathbb{I} \xrightarrow{u} B)$$
$$f + g = (A \xrightarrow{\Delta} A {*} A \xrightarrow{f * g} B {*} B \xrightarrow{\nabla} B)$$

For **SLat** the symmetric monoidal category of semilattices with respect to the universal bilinear tensor product we have the following result, which justifies the terminology of Definition 2.2.

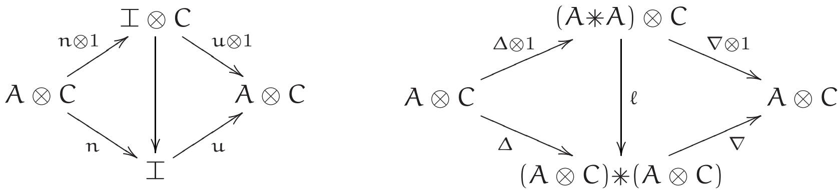
Proposition 2.4. *The following are equivalent.*

1. *Categories with degenerate biproduct structure.*
2. **SLat**-*categories with (necessarily enriched) finite products.*

Biproduct and Monoidal Structure. I further consider biproduct structure on symmetric monoidal categories. To this end, note that in a monoidal category with tensor $\otimes$ and binary products $\times$ there is a natural distributive law as follows:

$$\ell = \langle \pi_1 \otimes 1, \pi_2 \otimes 1 \rangle : (A \times B) \otimes C \longrightarrow (A \otimes C) \times (B \otimes C)$$

Definition 2.3. *A symmetric monoidal structure* $(\mathrm{I}, \otimes)$ *and a biproduct structure* $(\mathbb{I}, *; u, \nabla; n, \Delta)$ *on a category are* compatible *whenever the following hold:*



Recall that a **Mon**-*enriched (symmetric) monoidal category* is a (symmetric) monoidal category with a **Mon**-enrichment for which the tensor is strict and bilinear; that is, such that

$$0_{X,Y} \otimes f = 0_{X \otimes A, Y \otimes B} \qquad \text{and} \qquad f \otimes 0_{X,Y} = 0_{A \otimes X, B \otimes Y}$$

for all $f : A \longrightarrow B$, and

$$g \otimes (f + f') = g \otimes f + g \otimes f' \qquad \text{and} \qquad (g + g') \otimes f = g \otimes f + g' \otimes f$$

for all $f, f' : A \longrightarrow B$ and $g, g' : X \longrightarrow Y$.

Propositions 2.3 and 2.4 extend to the symmetric monoidal setting.

Proposition 2.5. *The following are equivalent.*

1. *Categories with compatible symmetric monoidal and biproduct structures.*
2. **Mon**-*enriched symmetric monoidal categories with (necessarily enriched) finite products.*
3. **CMon**-*enriched symmetric monoidal categories with (necessarily enriched) finite products.*

Corollary 2.2. *The following are equivalent.*

1. *Categories with compatible symmetric monoidal and degenerate biproduct structures.*
2. **SLat**-*enriched symmetric monoidal categories with (necessarily enriched) finite products.*

3 Models of Multiplicative Biadditive Intuitionistic Linear Logic

Models of Multiplicative Intuitionistic Linear Logic. I recall the definition of categorical model of multiplicative intuitionistic linear logic as it has been developed in the literature, see *e.g.* [17,20,2,3,1,18,19].

Definition 3.1. *An* $\mathcal{L}^!_\otimes$-model *is given by a category equipped with*

1. *a symmetric monoidal structure* $(\mathrm{I}, \otimes)$*;*
2. *a symmetric monoidal endofunctor* $\big(!, \varphi_\mathrm{I} : \mathrm{I} \longrightarrow !\mathrm{I}, \varphi : !A \otimes !B \longrightarrow !(A \otimes B)\big)$*;*
3. *a monoidal comonad structure* $A \xleftarrow{\epsilon} !A \xrightarrow{\delta} !!A$*;*
4. *a monoidal commutative comonoid structure* $\mathrm{I} \xleftarrow{e} !A \xrightarrow{d} !A \otimes !A$

subject to the following compatibility laws:

$$\begin{array}{ccc} !A & \xrightarrow{\delta} & !!A \\ {\scriptstyle e}\downarrow & & \downarrow{\scriptstyle !e} \\ \mathrm{I} & \xrightarrow[\varphi_\mathrm{I}]{} & !\mathrm{I} \end{array}$$

$$\begin{array}{ccccc} !A & & \xrightarrow{\delta} & & !!A \\ {\scriptstyle d}\downarrow & & & \swarrow{\scriptstyle d_!} & \downarrow{\scriptstyle !d} \\ !A \otimes !A & \xrightarrow[\delta\otimes\delta]{} & !!A \otimes !!A & \xrightarrow[\varphi_{!A,!A}]{} & !(!A \otimes !A) \end{array} \qquad (5)$$

Amongst the many coherence conditions imposed by the above definition on $\mathcal{L}^!_\otimes$-models (for which see *e.g.* [3]) note the following two:

$$\begin{array}{ccc} !A \otimes !B & \xrightarrow{\varphi} & !(A \otimes B) \\ {\scriptstyle e\otimes e}\downarrow & & \downarrow{\scriptstyle e_{A\otimes B}} \\ I \otimes I & \xrightarrow{\cong} & I \end{array} \tag{6}$$

$$\begin{array}{ccccc} !A \otimes !B & & \xrightarrow{\varphi} & & !(A \otimes B) \\ {\scriptstyle d\otimes d}\downarrow & & & & \downarrow{\scriptstyle d_{A\otimes B}} \\ !A \otimes !A \otimes !B \otimes !B & \xrightarrow[1\otimes\gamma\otimes 1]{} & !A \otimes !B \otimes !A \otimes !B & \xrightarrow[\varphi\otimes\varphi]{} & !(A \otimes B) \otimes !(A \otimes B) \end{array} \tag{7}$$

Definition 3.2. *An* $\mathcal{L}^!_{\otimes,\times}$-model *is an* $\mathcal{L}^!_\otimes$*-model on a category with finite products* $(\top, \times)$.

In this context, we obtain the *Seely* monoidal natural isomorphism

$$s : \ !A \otimes !B \xrightarrow{\cong} !(A \times B)$$

given by the composite

$$!A \otimes !B \xrightarrow{\delta\otimes\delta} !!A \otimes !!B \xrightarrow{\varphi} !(!A \otimes !B) \xrightarrow{!\langle \epsilon\otimes e, e\otimes\epsilon\rangle} !\big((A \otimes I) \times (I \otimes B)\big) \xrightarrow{\cong} !(A \times B)$$

with inverse

$$!(A \times B) \xrightarrow{d} !(A \times B) \otimes !(A \times B) \xrightarrow{!(1\times n)\otimes !(n\times 1)} !(A \times \top) \otimes !(\top \times B) \xrightarrow{\cong} !A \otimes !B$$

Also the map $s_I = I \xrightarrow{\varphi_I} !I \xrightarrow{!n} !\top$ is an isomorphism, with inverse $e : !\top \longrightarrow I$. It follows that the diagrams

$$\begin{array}{ccc} & !A & \\ {\scriptstyle d}\swarrow & & \searrow{\scriptstyle !\Delta} \\ !A \otimes !A & \xrightarrow[s_{A,A}]{\cong} & !(A \times A) \end{array} \tag{8}$$

$$\begin{array}{ccc} & !A & \\ {\scriptstyle e}\swarrow & & \searrow{\scriptstyle !n} \\ I & \xrightarrow[s_I]{\cong} & !\top \end{array} \tag{9}$$

commute; so that the *contraction* and *weakening* maps, d and e, arise from the product structure via the Seely isomorphims.

Proposition 3.1. *In an $\mathcal{L}^!_{\otimes,\times}$-model the following coherence law holds.*

$$\begin{array}{ccccc} !A \otimes !B & & \xrightarrow[\cong]{s} & & !(A \times B) \\ \downarrow{\scriptstyle \delta\otimes\delta} & & & & \downarrow{\scriptstyle \delta_{A\times B}} \\ !!A \otimes !!B & \xrightarrow[\varphi_{!A,!B}]{} & !(!A \otimes !B) & \xrightarrow[!s]{\cong} & !!(A \times B) \end{array} \tag{10}$$

The proof uses the definition of s, and the monoidality and associativity laws of δ.

Models of Multiplicative Biadditive Intuitionistic Linear Logic. I define categorical models of multiplicative biadditive intuitionistic linear logic to be models of multiplicative intuitionistic linear logic equipped with compatible biproduct structure. This is somewhat in the vein of Blute, Cockett, and Seely [4, Section 4].

Definition 3.3. *An* $\mathcal{L}^!_{\otimes,*}$-model *is an $\mathcal{L}^!_{\otimes}$-model equipped with a biproduct structure* $(\bot, *; u, \nabla; n, \Delta)$ *compatible with the symmetric monoidal structure* $(I, \otimes)$*.*

In this context, and via the monoidality of the Seely isomorphisms, the commutative bialgebra structure induced by the biproduct structure yields commutative bialgebraic-exponential structure.

Definition 3.4. *In $\mathcal{L}^!_{\otimes,*}$-models, the* coweakening *and* cocontraction *maps, $\imath$ and m, are defined as follows:*

$$\imath \;=\; I \xrightarrow[\cong]{s_I} !\bot \xrightarrow{!u} !A$$

$$m \;=\; !A \otimes !A \xrightarrow[\cong]{s_{A,A}} !(A * A) \xrightarrow{!\nabla} !A$$

Proposition 3.2. *In an $\mathcal{L}^!_{\otimes,*}$-model, the natural transformations*

$$I \xrightarrow{\imath} !A \xrightarrow{e} I \qquad !A \otimes !A \xrightarrow{m} !A \xrightarrow{d} !A \otimes !A$$

form a commutative bialgebra.

More surprisingly, the following result exhibits three coherence laws enjoyed by $\mathcal{L}^!_{\otimes,*}$-models that can respectively be thought of as a kind of unfolding of the coherence conditions (5), (6), (7).

Theorem 3.1. *In an $\mathcal{L}^{!}_{\otimes,\circledast}$-model the following coherence laws hold.*

1.

$$\begin{array}{ccccc} !A & & \xrightarrow{\quad\delta\quad} & & !!A \\ {\scriptstyle m}\uparrow & & & & \uparrow{\scriptstyle !m} \\ !A \otimes !A & \xrightarrow[\delta\otimes\delta]{} & !!A \otimes !!A & \xrightarrow[\varphi_{!A,!A}]{} & !(!A \otimes !A) \end{array} \tag{11}$$

2.

$$\begin{array}{ccccc} & & !A \otimes !B & \xrightarrow{\varphi} & !(A \otimes B) \\ & \nearrow{\scriptstyle \iota\otimes 1} & & & \uparrow{\scriptstyle \iota_{A\otimes B}} \\ I \otimes !B & & & & \\ & \searrow{\scriptstyle 1\otimes e} & & & \\ & & I \otimes I & \xrightarrow{\cong} & I \end{array} \tag{12}$$

3.

$$\begin{array}{ccccc} !A \otimes !B & & \xrightarrow{\qquad\varphi\qquad} & & !(A \otimes B) \\ {\scriptstyle m\otimes 1}\uparrow & & & & \\ !A \otimes !A \otimes !B & & & & \uparrow{\scriptstyle m} \\ {\scriptstyle 1\otimes 1\otimes d}\downarrow & & & & \\ !A \otimes !A \otimes !B \otimes !B & \xrightarrow[1\otimes\gamma\otimes 1]{} & !A \otimes !B \otimes !A \otimes !B & \xrightarrow[\varphi\otimes\varphi]{} & !(A \otimes B) \otimes !(A \otimes B) \end{array} \tag{13}$$

The proof of Theorem 3.1(1) uses the definition of m and the coherence law (10). The proof of Theorem 3.1(2) uses the coherence law (9), the definitions of s_I and ι, the monoidality of $(!, \varphi_I, \varphi)$, and the strictness of the tensor product to show that both composites are equal to the following one

$$I \otimes !B \overset{\cong}{\dashrightarrow} !B \xrightarrow{!0} !(A \otimes B)$$

Finally, the proof of Theorem 3.1(3) uses the product structure of $(\top, \circledast)$, the bilinearity of the tensor product, the definitions of m and s^{-1}, the coherence law (8), and the monoidality of d.

4 Differential Structure

The analysis of differential structure in $\mathcal{L}^{!}_{\otimes,\circledast}$-models follows.

Creation Operators. The starting point is the definition of annihilation and creation operators; the terminology for which is justified by Proposition 4.1.

Definition 4.1. *In an $\mathcal{L}^!_{\otimes}$-model, the* annihilation operator $\propto : !A \longrightarrow A \otimes !A$ *is the natural transformation given by the following composite*

$$!A \xrightarrow{d} !A \otimes !A \xrightarrow{\epsilon\otimes 1} A \otimes !A$$

Definition 4.2. *A* creation operator *in an $\mathcal{L}^!_{\otimes,\ast}$-model is a natural transformation*

$$\partial : A \otimes !A \longrightarrow !A$$

satisfying the following laws.

1. Strength.

$$\begin{array}{ccccc} A \otimes !A \otimes !B & \xrightarrow{\partial\otimes 1} & !A \otimes !B & \xrightarrow{\varphi} & !(A \otimes B) \\ \downarrow{\scriptstyle 1\otimes 1\otimes \propto} & & & & \uparrow{\scriptstyle \partial_{A\otimes B}} \\ A \otimes !A \otimes B \otimes !B & \xrightarrow[1\otimes\gamma\otimes 1]{} & A \otimes B \otimes !A \otimes !B & \xrightarrow[1\otimes 1\otimes\varphi]{} & A \otimes B \otimes !(A \otimes B) \end{array} \tag{14}$$

2. Comonad.

$$\begin{array}{ccccc} A \otimes !A & \xrightarrow{\partial} & !A & \xrightarrow{\epsilon} & A \\ & {\scriptstyle 1\otimes e}\searrow & & \nearrow{\scriptstyle \cong} & \\ & & A \otimes I & & \end{array} \qquad \begin{array}{ccccc} A \otimes !A & \xrightarrow{\partial} & !A & \xrightarrow{\delta} & !!A \\ \downarrow{\scriptstyle 1\otimes d} & & & & \uparrow{\scriptstyle \partial_!} \\ A \otimes !A \otimes !A & & \xrightarrow[\partial\otimes\delta]{} & & !A \otimes !!A \end{array} \tag{15}$$

3. Multiplication.

$$\begin{array}{ccccc} A \otimes !A \otimes !A & \xrightarrow{\partial\otimes 1} & !A \otimes !A & \xrightarrow{m} & !A \\ & {\scriptstyle 1\otimes m}\searrow & & \nearrow{\scriptstyle \partial} & \\ & & A \otimes !A & & \end{array} \tag{16}$$

The above form for creation and annihilation operators is non-standard. More commonly, see *e.g.* [14], the literature deals with creation operators $\partial_v : !A \longrightarrow !A$ for vectors $v : I \longrightarrow A$ and annihilation operators $\propto_{v'} : !A \longrightarrow !A$ for covectors $v' : A \longrightarrow I$. In the present setting, these are derived as follows:

$$\partial_v \;=\; !A \xrightarrow{\cong} I \otimes !A \xrightarrow{v\otimes 1} A \otimes !A \xrightarrow{\partial} !A$$

$$\propto_{v'} \;=\; !A \xrightarrow{\propto} A \otimes !A \xrightarrow{v'\otimes 1} I \otimes !A \xrightarrow{\cong} !A$$

Proposition 4.1. *Creation and annihilation operators in $\mathcal{L}^!_{\otimes,\ast}$-models satisfy the following commutation relations:*

1. $\propto \partial \;=\; 1 + (1 \otimes \partial)(\gamma \otimes 1)(1 \otimes \propto) \;\; : A \otimes !A \longrightarrow A \otimes !A$
2. $\partial\,(1 \otimes \partial) \;=\; \partial\,(1 \otimes \partial)\,(\gamma \otimes 1) \;\; : A \otimes A \otimes !A \longrightarrow !A$
3. $(1 \otimes \propto) \propto \;=\; (\gamma \otimes 1)(1 \otimes \propto) \propto \;\; : !A \longrightarrow A \otimes A \otimes !A$

It follows that

$$\begin{aligned} \propto_{v'} \partial_v &= \big(!A \overset{\cong}{\dashrightarrow} I \otimes !A \xrightarrow{(v'v)\otimes 1} I \otimes !A \overset{\cong}{\dashrightarrow} !A\big) + \big(!A \xrightarrow{\partial_v \propto_{v'}} !A\big) \\ \partial_u \partial_v &= \partial_v \partial_u \\ \propto_{u'} \propto_{v'} &= \propto_{v'} \propto_{u'} \end{aligned}$$

for all $u, v : I \longrightarrow A$ and $u', v' : A \longrightarrow I$.

For comparison with the work of Blute, Cockett, and Seely on deriving transformations, note that the laws of (15) are the *linearity* and the *chaining* conditions of [4, Definition 2.5] and that the law of (16) is the *multiplication rule* of [4, Definition 4.10]. The law of (14) is novel, and in its presence the *constant maps* and the *copying* conditions of [4, Definition 2.5] are derivable (see Proposition 4.2 below). Thus, creation operators are deriving transformations satisfying the multiplication rule.

Proposition 4.2. *Every creation operator ∂ in an $\mathcal{L}^!_{\otimes,*}$-model is such that*

1. $e\,\partial = 0 : A \otimes !A \longrightarrow I$, *and*
2. $d\,\partial = (\partial_1 + \partial_2)\,(1 \otimes d) : A \otimes !A \longrightarrow !A \otimes !A$ *where* $\partial_1 = A \otimes !A \otimes !A \xrightarrow{\partial \otimes 1} !A \otimes !A$ *and* $\partial_2 = A \otimes !A \otimes !A \xrightarrow{\gamma \otimes 1} !A \otimes A \otimes !A \xrightarrow{1 \otimes \partial} !A \otimes !A$.

Propositions 4.1 and 4.2 are better established using the representation of creation operators given in Theorem 4.1 below. The proofs of Propositions 4.1(1) and 4.2(2) use the biproduct structure, the strictness and bilinearity of the tensor product, the coherence of the Seely isomorphisms, and the bialgebraic-exponential structure; the proofs of Propositions 4.1(2&3) use the commutative of the bialgebraic-exponential structure; the proof of Proposition 4.2(1) uses the strictness of the tensor product.

Creation Maps. Creation operators have a simpler axiomatisation in terms of creation maps.

Definition 4.3. *A* creation map *in an $\mathcal{L}^!_{\otimes,*}$-model is a natural transformation $\eta : A \longrightarrow !A$ satisfying the following laws.*

1. *Strength.*

$$\begin{array}{ccccc} A \otimes !B & \xrightarrow{\eta \otimes 1} & !A \otimes !B & \xrightarrow{\varphi} & !(A \otimes B) \\ & {\scriptstyle 1\otimes\epsilon} \searrow & & \nearrow {\scriptstyle \eta_{A\otimes B}} & \\ & & A \otimes B & & \end{array}$$

2. *Comonad.*

$$\begin{array}{ccccc} & & !A & & \\ & {\scriptstyle \eta} \nearrow & & \searrow {\scriptstyle \epsilon} & \\ A & & \xrightarrow[1]{} & & A \end{array} \qquad \begin{array}{ccccc} A & \xrightarrow{\eta} & !A & \xrightarrow{\delta} & !!A \\ {\scriptstyle \cong}\downarrow & & & & \uparrow {\scriptstyle m_!} \\ A \otimes I & \xrightarrow[\eta \otimes \iota]{} & !A \otimes !A & \xrightarrow[\eta_! \otimes \delta]{} & !!A \otimes !!A \end{array}$$

As a direct consequence of the strength and first comonad law, creation maps are coherent with respect to the monoidal strength.

Proposition 4.3. *Every creation map* η *in an* $\mathcal{L}^!_{\otimes,*}$*-model satisfies the following coherence law:*

$$\begin{array}{ccc} & A \otimes B & \\ {\scriptstyle \eta\otimes\eta} \swarrow & & \searrow {\scriptstyle \eta_{A\otimes B}} \\ !A \otimes !B & \xrightarrow[\varphi]{} & !(A \otimes B) \end{array}$$

Theorem 4.1. *The mappings*

$$\partial : A \otimes !A \longrightarrow !A \;\longmapsto\; \eta = \; A \overset{\cong}{\dashrightarrow} A \otimes I \xrightarrow{1\otimes\iota} A \otimes !A \xrightarrow{\partial} !A$$

$$\eta : A \longrightarrow !A \;\longmapsto\; \partial = \; A \otimes !A \xrightarrow{\eta\otimes 1} !A \otimes !A \xrightarrow{m} !A$$

yield a bijection between creation operators and creation maps in $\mathcal{L}^!_{\otimes,*}$*-models.*

To show that the map η induced by a creation operator ∂ satisfies the strength law one uses the strength law for ∂ and the coherence law (12); to show that η satisfies the first comonad law one uses the first comonad law for ∂; to show that η satisfies the second comonad law one uses the second comonad law and the multiplication law for ∂. Conversely, to show that the operator ∂ induced by a creation map η satisfies the strength law one uses the strength law for η and the coherence law (13); to show that ∂ satisfies the first comonad law one uses the biproduct structure, the strictness of the tensor product, and the first comonad law for η; to show that ∂ satisfies the second comonad law one uses the strength law and second comonad law for η, the coherence condition (5), the coherence laws (11) and (13), and the comonad laws for $(!, \epsilon, \delta)$; to show that ∂ satisfies the multiplication law one uses the associativity of m.

Differentiation. In the presence of the above differential structure, one obtains a natural *differential operator*

$$D[-] = [\partial, -] : \; [!A, B] \longrightarrow [A \otimes !A, B]$$

such that the following rules hold.

1. Identity rule.
$$D[1] \;=\; \partial$$
2. Composition rule.
$$D[\ell\, f] \;=\; \ell\, D[f] \qquad (f : !A \longrightarrow B, \ell : B \longrightarrow C)$$
3. Constant rule.
$$D[e_A] \;=\; 0$$
4. Sum rule.
$$D[f + g] \;=\; D[f] + D[g] \qquad (f, g : !A \longrightarrow B)$$

5. Tensor rule.

$$D[(f{\otimes}g)\,d] = \big(D[f]{\otimes}g + (f{\otimes}D[g])(\gamma{\otimes}1)\big)\,(1{\otimes}d) \quad (f : !A \longrightarrow B, g : !A \longrightarrow C)$$

6. Linearity rule.

$$D[\ell\,\epsilon_A] \;=\; \ell\,(1 \otimes e_A) \qquad (\ell : A \longrightarrow B)$$

7. Chain rule.

$$D[g\,f^\dagger] \;=\; D[g]\,(D[f] \otimes f^\dagger)\,(1 \otimes d) \qquad (f : !A \longrightarrow B, g : !B \longrightarrow C)$$

where $f^\dagger = (!f)\,\delta : !A \longrightarrow !B$

Further, for $\mathcal{L}^{!,\multimap}_{\otimes,\ast}$-models, *i.e.* in the presence of closed structure ($\multimap$), one may internalise the differential operator as a *partial derivative operator*

$$D = \lambda u : A.\, \lambda f : !A \multimap B.\, \lambda x : !A.\, f\big(\partial(u \otimes x)\big) \;: A \multimap (!A \multimap B) \multimap !A \multimap B$$

for which, moreover, the following rules hold.

1. Symmetry rule.

$$u : A, v : A \;\vdash\; D_u \circ D_v = D_v \circ D_u \;: !A \multimap B$$

2. Strength rule.

$$f : !(A \otimes B) \multimap C, u : A, x : !A, y : !B$$
$$\vdash D_u\big[\lambda x : !A.\, f\big(\varphi(x \otimes y)\big)\big]x \;=\; \mathsf{let}\ v \otimes z = \propto(y) : B \otimes !B\ \mathsf{in}\ D_{u \otimes v}\big[f\big]\big(\varphi(x \otimes z)\big) : C$$

5 Concluding Remarks

The general theme of this paper has been the investigation of categorical models of multiplicative biadditive intuitionistic linear logic, and of differential structure therein. Within each of these two strands, various possibilities for research still remain. I mention a few here.

From the abstract theoretical viewpoint, the consideration of $\mathcal{L}^{!,\multimap}_{\otimes,\ast}$-models equipped with differential structure as categorical models of the differential λ-calculus of Ehrhard and Regnier [9] will be considered in the full version of the paper. A more important next step, however, is to work out the type and proof theory of $\mathcal{L}^{!,\multimap}_{\otimes,\ast}$-models, both as a term assignment system and a graphical calculus, and thereafter extend them to incorporate differential structure. In another direction, the relationship of our axiomatics with the earlier categorical axiomatic investigation of differential structure provided by Synthetic Differential Geometry (see *e.g.* [16, Part I]) should be addressed.

From the model-theoretic viewpoint, the discussion of concrete $\mathcal{L}^{!,\multimap}_{\otimes,\ast}$-models equipped with differential structure will be considered in the full version of the paper, where the diligent, but otherwise evident, verification that the models mentioned at the beginning of the Introduction are examples will be covered. More interestingly, I conjecture that the category of convenient vector spaces and linear maps of Frölicher and Kriegl [13] provides yet another example; as so may be the case, indicated to me by Anders Kock in correspondence, with the category of modules for the ring object of line type in some models of Synthetic Differential Geometry (see *e.g.* [16, Part III]).

References

1. Benton, N.: A mixed linear and non-linear logic: Proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, Springer, Heidelberg (1995)
2. Benton, N., Bierman, G., de Paiva, V., Hyland, M.: Linear λ-calculus and categorical models revisited. In: Martini, S., Börger, E., Kleine Büning, H., Jäger, G., Richter, M.M. (eds.) CSL 1992. LNCS, vol. 702, Springer, Heidelberg (1993)
3. Bierman, G.: What is a categorical model of intuitionistic linear logic? In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 78–93. Springer, Heidelberg (1995)
4. Blute, R., Cockett, J., Seely, R.: Differential categories. Mathematical Structures in Computer Science 16(6), 1049–1083 (2006)
5. Blute, R., Panangaden, P., Seely, R.: Fock space: A model of linear exponential types. Corrected version of Holomorphic models of exponential types in linear logic. In: Main, M.G., Melton, A.C., Mislove, M.W., Schmidt, D., Brookes, S.D. (eds.) Mathematical Foundations of Programming Semantics. LNCS, vol. 802, Springer, Heidelberg (1993)
6. Ehrhard, T.: On Köthe sequence spaces and linear logic. Mathematical Structures in Computer Science 12(5), 579–623 (2002)
7. Ehrhard, T.: Finiteness spaces. Mathematical Structures in Computer Science 15(4), 615–646 (2005)
8. Ehrhard, T., Regnier, L.: Differential interaction nets. Theoretical Computer Science 364(2), 166–195 (2006)
9. Ehrhard, T., Reigner, L.: The differential lambda-calculus. Theoretical Computer Science 309(1–3), 1–41 (2003)
10. Fiore, M.: Generalised species of structures: Cartesian closed and differential structure. Draft (2004)
11. Fiore, M.: Mathematical models of computational and combinatorial structures. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 25–46. Springer, Heidelberg (2005)
12. Fiore, M., Gambino, N., Hyland, M., Winskel, G.: The cartesian closed bicategory of generalised species of structures. Preprint (2006)
13. Frölicher, A., Kriegl, A.: Linear spaces and differentiation theory. Wiley Series in Pure and Applied Mathematics. Wiley-Interscience Publication, Chichester (1988)
14. Geroch, R.: Mathematical Physics. University of Chicago Press, Chicago (1985)
15. Hyvernat, P.: A logical investigation of interaction systems. PhD thesis, Université de la Mediterranée, Aix-Marseille 2 (2005)
16. Kock, A.: Synthetic Differential Geometry. London Mathematical Society Lecture Notes Series, vol. 333. Cambridge University Press, Cambridge (2006)
17. Lafont, Y.: Logiques, catégories et machines. PhD thesis, Université Paris, 7 (1988)
18. Melliès, P.-A.: Categorical models of linear logic revisited. To appear in Theoretical Computer Science
19. Schalk, A.: What is a categorical model of linear logic? Notes for research students (2004)
20. Seely, R.: Linear logic, *-autonomous categories and cofree coalgebras. In: Applications of Categories in Logic and Computer Science. Contemporary Mathematics, vol. 92 (1989)
21. Sweedler, M.: Hopf algebras. Benjamin, NY (1969)

The Omega Rule is $\mathbf{\Pi_1^1}$-Complete in the $\boldsymbol{\lambda\beta}$-Calculus

Benedetto Intrigila[1] and Richard Statman[2]

[1] Università di Roma "Tor Vergata", Rome, Italy
[2] Carnegie-Mellon University, Pittsburgh, PA, USA

Abstract. We give a many-one reduction of the set of true $\mathbf{\Pi_1^1}$ sentences to the set of consequences of the λ-calculus with the ω-rule. This solves in the affirmative a long-standing problem of H. Barendregt (1975).

1 Introduction

The present paper completes our analysis of the logical status of the so-called ω-rule in λ-theories.

We have first considered constructive forms of such rule in [9], obtaining r.e. λ-theories which are closed under the ω-rule. This gives the counterintuitive result that closure under the ω-rule does not necessarily give rise to non constructive λ-theories, thus solving a problem of A. Cantini.

Then we have considered the ω-rule with respect to the highly non constructive λ-theory $\mathcal{H}$. $\mathcal{H}$ is obtained extending β-conversion by identifying all closed unsolvables. $\mathcal{H}\omega$ is the closure of this theory under the ω-rule (and β-conversion). A long-standing conjecture of H. Barendregt ([4], Conjecture 17.4.15) stated that the provable equations of $\mathcal{H}\omega$ form a $\mathbf{\Pi_1^1}$-Complete set. In [10], we solved in the affirmative the problem.

Of course the most important problem is to determine the logical power of the ω-rule when added to the *pure* $\lambda\beta$-calculus.

This is also relevant for theorem provers such as Coq or Isabelle/HOL (see e.g. [1], [2], [3]), where it seems very hard to automatically set up inductive arguments to get universal conclusions. In this sense, the use of some (constructive) kind of ω-rule is very appealing since one could get a universal conclusion from, say, a finite number of cases. Typically, this happens when for every property P of interest, there exists a computable upper bound k such that if every ground term of complexity less than k satisfies P then $\forall x.P(x)$ holds, so that a universal conclusion can be obtained e.g. by a systematic search on a finite set of cases.

In [8], we showed that the resulting theory $\lambda\omega$ is *not* recursively enumerable, by giving a many-one reduction of the set of true $\mathbf{\Pi_2^0}$ sentences to the set of consequences of the λ-calculus with the omega rule, thus solving a problem originated with H. Barendregt and re-raised in [6].

The problem of the logical upper bound to $\lambda\omega$ remained open. That this bound is $\mathbf{\Pi_1^1}$ has been conjectured again by H. Barendregt in the well known Open Problems List, which ends the 1975 Conference on "λ-Calculus and Computer

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 178–193, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Science Theory", edited by C. Böhm [5]. Here we solve in the affirmative this conjecture. The celebrated *Plotkin terms* (introduced in [11]) furnish the main technical tool.

2 The $\boldsymbol{\omega}$-Rule

The present paper requires acquaintance with [8], although we shall adopt a somewhat different approach.

Notation will be standard and we refer to [4], for terminology and results on λ-calculus. In particular:

$\equiv$ denotes syntactical identity;
$\longrightarrow_\beta$, $\longrightarrow_\eta$ and $\longrightarrow_{\beta\eta}$ denote β-, η- and, respectively, $\beta\eta$-reduction;
$\longrightarrow^*_\beta$, $\longrightarrow^*_\eta$ and $\longrightarrow^*_{\beta\eta}$ their respective reflexive and transitive closures;
$=_\beta$ and $=_{\beta\eta}$ denote β- and, respectively, $\beta\eta$-conversion;
combinators (i.e. closed λ-terms) such e.g. $\mathbf{I}$ have the usual meaning;
$\underline{k}$ denotes the kth Church numeral.

We denote λ-terms by capital letters: in particular we adopt the convention that $H, J, M, N, P, Q, \ldots$ are *closed* terms and U, V, X, Y, W, Z are possibly *open* terms. For a λ-term the notions of *having order* 0 and *having positive order* have the usual meaning ([4] 17.3.2). In the sequel, we shall need the following notions. We define the notions of *trace* and *extended trace* (shortly *etrace*) as follows. Given the reduction $F \longrightarrow^*_{\beta\eta} G$ and the closed subterm M of F the *traces* of M in the terms of the reduction are simply the copies of M until each is either deleted by a contraction of a redex with a dummy λ or altered by a reduction internal to M or by a reduction with M at the head (when M begins with λ). The notion of *etrace* is the same except that we allow internal reductions, so that a copy of M altered by an internal reduction continues to be an etrace.

By $\lambda\beta$ we denote the theory of β-convertibility (see [4]). The theory $\lambda\omega$ is obtained by adding the so called ω*-rule* to $\lambda\beta$, see [4] 4.1.10.

We formulate $\lambda\omega$ slightly differently. In particular, we want a formulation of the theory such that only equalities between *closed* terms can be proven.

Definition 1. Equality in $\lambda\omega$ *(denoted by $=_\omega$) is defined by the following rules:*

- *$\beta\eta$-conversion:*
  if $M =_{\beta\eta} N$ then $M =_\omega N$
- *the rule of substituting equals for equals in the form:*
  if $M =_\omega N$ then $PM =_\omega PN$
- *transitivity and symmetry of equality,*
- *the ω-rule itself:*

$$\frac{\forall M,\ M\ closed,\ PM =_\omega QM}{P\ =_\omega\ Q}$$

We are working here with $\beta\eta$-conversion for convenience. The so called η-rule (that is $(\lambda x.Mx) =_\eta M$) obviously holds in $\lambda\omega$. Nevertheless it will be useful to have this rule at disposal to put proofs in some specified forms. We leave to the reader to check that the formulation above is equivalent to the standard one.

As usual proofs in $\lambda\omega$ can be thought of as (possibly infinite) well-founded trees. In particular the end piece of such a proof consists of a finite tree of equality inferences all of whose leaves are either $\beta\eta$ conversions or direct conclusions of the ω-rule. It is easy to see that each such end piece can be put in the form:

$$F =_{\beta\eta} G_1M_1 =_\omega G_1N_1 =_{\beta\eta} G_2M_2 =_\omega G_2N_2 =_{\beta\eta} \ldots G_tM_t =_\omega G_tN_t =_{\beta\eta} H$$

where $M_i =_\omega N_i$, for $1 \leq i \leq t$ are direct conclusions of the ω-rule. See [10], Section 5, for more details (in a slightly different context). This is a particular case of a general result due to the second author of the present paper, see [13]. Moreover, by the Church-Rosser Theorem this configuration of inferences can be put in the form

$$F \longrightarrow^*_{\beta\eta} J_1 \;{}^*_{\beta\eta}\!\longleftarrow G_1M_1 =_\omega G_1N_1 \longrightarrow^*_{\beta\eta} J_2 \;{}^*_{\beta\eta}\!\longleftarrow \qquad (1)$$

$${}^*_{\beta\eta}\!\longleftarrow G_2M_2 =_\omega G_2N_2 =_{\beta\eta}\longrightarrow^*_{\beta\eta} \ldots$$

$$\ldots \;{}^*_{\beta\eta}\!\longleftarrow G_tM_t =_\omega G_tN_t \longrightarrow^*_{\beta\eta} J_{t+1} \;{}^*_{\beta\eta}\!\longleftarrow H$$

where $M_i =_\omega N_i$, for $1 \leq i \leq t$, are as above. We shall call the sequence (1) *the standard form* for the end piece of a proof.

Since proofs are infinite trees $\mathcal{T}$ they can be described by countable ordinals. We shall need a few facts about countable ordinals, that we briefly mention in the following. All we need here is exposed in [10], Section 6. For more details on constructive ordinals, see e.g. [12].

(a) Cantor Normal Form to the Base Omega. (ω) Every countable ordinal α can be written uniquely in the form $\omega^{\alpha_1} * n_1 + \cdots + \omega^{\alpha_k} * n_k$ where $n_1, \ldots, n_k$ are positive integers and $\alpha_1 > \ldots > \alpha_k$ are ordinals.

(b) Hessenberg Sum. Write $\alpha = \omega^{\alpha_1} * n_1 + \cdots + \omega^{\alpha_k} * n_k$ and $\gamma = \omega^{\alpha_1} * m_1 + \cdots + \omega^{\alpha_k} * m_k$ where some of the n_i and m_j may be 0. Then the *Hessenberg Sum* is defined as follows: $\alpha \oplus \gamma =_{def} \omega^{\alpha_1} * (n_1 + m_1) + \cdots + \omega^{\alpha_k} * (n_k + m_k)$. Hessenberg sum is strictly increasing on both arguments. That is, for α, γ different from 0, we have: $\alpha, \gamma < \alpha \oplus \gamma$.

(c) Hessenberg Product. We only need this for product with an integer. We put: $\alpha \odot n =_{def} \alpha \oplus \cdots \oplus \alpha$ n-times.

Coming back to proofs, observe first that we can assume that if a proof has an endpiece, then this endpiece is in standard form (see above). The ordinal that we want to assign to a proof $\mathcal{T}$ (considered as a tree) is defined as follows:

Definition 2. *The transfinite ordinal* $ord(\mathcal{T})$, the order of $\mathcal{T}$, *is defined recursively by:*

– *If* $\mathcal{T}$ *ends in an endpiece computation of the form* (1) *with no instances of the* ω*-rule* ($t = 0$)*, that is consisting of a unique* $\beta\eta$*-conversion, then* $ord(\mathcal{T}) =_{def} 1$*;*
– *If* $\mathcal{T}$ *ends in an instance of the* ω*-rule whose premisses have trees resp.* $\mathcal{T}_1, \ldots \mathcal{T}_i, \ldots$ *then* $ord(\mathcal{T}) =_{def} \omega^\theta$*, with* $\theta = Sup\{ord(\mathcal{T}_1) \oplus \cdots \oplus ord(\mathcal{T}_i) : i = 1, 2, \ldots\}$*;*
– *If* $\mathcal{T}$ *ends in an endpiece computation of the form* (1)*, with* $t > 0$ *instances of the* ω*-rule, and the* t *premisses* $M_1 =_\omega N_1, \ldots, M_t =_\omega N_t$ *have resp. trees* $\mathcal{T}_1, \ldots, \mathcal{T}_t$ *then* $ord(\mathcal{T}) =_{def} 1 \oplus ord(\mathcal{T}_1) \oplus ord(\mathcal{T}_2) \cdots \oplus ord(\mathcal{T}_t)$*.*

where $\oplus$ *is the Hessenberg sum of ordinals defined above.*

A proof of the following facts can be found in [10], Section 6.

1. **Fact**.
   If $\mathcal{T}$ ends in an endpiece computation of the form (1), with $t > 0$, and the equations $M_1 =_\omega N_1, \ldots, M_t =_\omega N_t$, have resp. trees $\mathcal{T}_1, \ldots, \mathcal{T}_t$ then $ord(\mathcal{T}) > ord(\mathcal{T}_i)$, for each $i = 1, \ldots, t$.
2. **Fact**.
   Assume that $\mathcal{T}$ ends in an instance of the ω-rule whose premisses have, respectively, trees $\mathcal{T}_1, \ldots, \mathcal{T}_t, \ldots$ Then for any integers $t, n_1, \ldots, n_t$, $ord(\mathcal{T}) > ord(\mathcal{T}_1) \odot n_1 \oplus \cdots \oplus ord(\mathcal{T}_t) \odot n_t$.

3 Canonical Proofs

We want to show that proofs in $\lambda\omega$ can be set in a suitable form.

Definition 3. *We say that* M *has* the same form as N *iff:*

– *if* $N \equiv \lambda y_1 \ldots y_n.YL_1 \ldots L_m$*, with* Y *beginning with* λ*,*
  then $M \equiv \lambda y_1 \ldots y_n.ZP_1 \ldots P_m$*, with* Z *beginning with* λ *and*
  $\lambda y_1 \ldots y_n.Y =_\omega \lambda y_1 \ldots y_n.Z$*,*
  and for every i*, with* $1 \le i \le m$*,*
  $\lambda y_1 \ldots y_n.L_i =_\omega \lambda y_1 \ldots y_n.P_i$*;*
– *if* $N \equiv \lambda y_1 \ldots y_n.y_j L_1 \ldots L_m$*, then* $M \equiv \lambda y_1 \ldots y_n.y_j P_1 \ldots P_m$*,*
  and for every i*, with* $1 \le i \le m$*,*
  $\lambda y_1 \ldots y_n.L_i =_\omega \lambda y_1 \ldots y_n.P_i$*.*

Where possibly $n = 0$*.*

Assume now that a set $\mathcal{X}$ of closed terms, cofinal for $\beta\eta$-reductions, has been specified.

Definition 4. *An endpiece in standard form:*

$$F \longrightarrow^*_{\beta\eta} H_1 \ {}^*_{\beta\eta}\!\longleftarrow G_1M_1 =_\omega G_1N_1 \longrightarrow^*_{\beta\eta} H_2 \ {}^*_{\beta\eta}\!\longleftarrow \tag{2}$$
$${}^*_{\beta\eta}\!\longleftarrow G_2M_2 =_\omega G_2N_2 =_{\beta\eta}\longrightarrow^*_{\beta\eta} \cdots$$
$$\ldots \ {}^*_{\beta\eta}\!\longleftarrow G_tM_t =_\omega G_tN_t \longrightarrow^*_{\beta\eta} H_{t+1} \ {}^*_{\beta\eta}\!\longleftarrow F'$$

is called an $\mathcal{X}$-canonical endpiece *(or, when $\mathcal{X}$ is clear from the context, simply* a canonical endpiece*) iff*

1. *for every i, $i = 1, \ldots, t+1$, the confluence terms H_i belong to $\mathcal{X}$;*
2. *(Conditions on the Left Facing Arrows)*
   for every i, $i = 1, \ldots, t$, the sequence of reductions:
   $H_i \ {}^{*}_{\beta\eta}\!\longleftarrow G_i M_i$
   has the following structure:
   - *a one step β-reduction of the form $[M_i/x]X \ {}_{\beta}\!\longleftarrow (\lambda x.X)M_i$, for some X, where we moreover require that $\lambda x.X$ has* not *the form: $\lambda x z_1 \ldots z_n. x U_1 \ldots U_s$,*
   - *followed by a sequence of non-head β-reductions,*
   - *followed by a sequence of η-reductions.*
3. *(Condition on the Right Facing Arrows)*
   for $i = 1, \ldots, t$, assume that in the $\beta\eta$-reduction component:
   $G_i M_i =_\omega G_i N_i \longrightarrow^{*}_{\beta\eta} H_{i+1}$
   G_i has the form:
   $\lambda x.\lambda y_1 \ldots y_n.((\lambda y.Y)L_1 \ldots L_m)Z_1 \ldots Z_n$
   and $[N_i/x]Z_j \longrightarrow^{*}_{\beta\eta} y_j$, for every j, with $1 \leq j \leq n$,
   moreover let P_k, with $k = 0, \ldots, m$, range over the sequence:
   $[N_i/x]\lambda y.Y,\ [N_i/x]L_1, \ldots [N_i/x]L_m$*;*
   if P_k has a $\beta\eta$-normal form R_k let $J_k \equiv_{def} R_k$ and otherwise set:
   $J_k \equiv_{def} P_k$*;*
   finally let the term H^{+}_{i+1} be defined as follows:

$$H^{+}_{i+1} \equiv J_0 J_1 \ldots J_m \tag{3}$$

   we require that

$$G_i N_i \longrightarrow^{*}_{\beta\eta} H^{+}_{i+1} \longrightarrow^{*}_{\beta\eta} H_{i+1}.$$

Remark. Observe that, in a canonical endpiece of the form (2) above, each G_i is of the form $\lambda x.X$, for some X, and $[M_i/x]X$ reduces, by a (possibly empty) sequence of η-reductions, to a term *of the same form* as H_i.

Definition 5. *Given the cofinal set $\mathcal{X}$, the notion of $\mathcal{X}$-*canonical proof *is defined inductively as follows.*

- *A $\beta\eta$-conversion is $\mathcal{X}$-*canonical *if the confluence term belongs to $\mathcal{X}$.*
- *An instance of the ω-rule is $\mathcal{X}$-*canonical *if the proofs of the premisses of the instances are $\mathcal{X}$-canonical.*
- *Otherwise a proof is* canonical *if its endpiece is a $\mathcal{X}$-canonical endpiece and all the proofs of the leaves which are direct conclusions of the ω-rule are $\mathcal{X}$-canonical.*

Proposition 1. *For every cofinal set $\mathcal{X}$, every provable equality $M =_\omega N$ has a $\mathcal{X}$-canonical proof.*

Proof. Let $\mathcal{X}$ fixed. We prove this proposition by induction on the ordinal $ord(\mathcal{T})$ of a proof $\mathcal{T}$ of $M =_\omega N$. For the basis case just suppose that $M =_{\beta\eta} N$ and use the Church-Rosser theorem.

For the induction step we distinguish two cases.

Case 1: $M =_\omega N$ is the direct conclusion of the ω-rule. This follows directly from the induction hypothesis.

Case 2: $\mathcal{T}$ has an endpiece of the form:

$$M \longrightarrow^*_{\beta\eta} H_1 \;{}^*_{\beta\eta}\!\longleftarrow G_1M_1 =_\omega G_1N_1 \longrightarrow^*_{\beta\eta} H_2 \;{}^*_{\beta\eta}\!\longleftarrow \tag{4}$$

$${}^*_{\beta\eta}\!\longleftarrow G_2M_2 =_\omega G_2N_2 =_{\beta\eta}\longrightarrow^*_{\beta\eta} \cdots$$

$$\cdots \;{}^*_{\beta\eta}\!\longleftarrow G_tM_t =_\omega G_tN_t \longrightarrow^*_{\beta\eta} H_{t+1} \;{}^*_{\beta\eta}\!\longleftarrow N$$

where, for each $i = 1 \ldots t$, $M_i =_\omega N_i$ is the conclusion of an instance of the ω-rule.

Consider the first component of (4): $M \longrightarrow^*_{\beta\eta} H_1 \;{}^*_{\beta\eta}\!\longleftarrow G_1M_1 =_\omega G_1N_1$. Let σ be a standard $\beta\eta$-reduction $G_1M_1 \longrightarrow^*_{\beta\eta} H_1$, with all the η-reductions postponed. We have now different subcases.

First Subcase. No etrace of M_1 appears in functional position in a head redex neither in the head part of σ, nor in H_1 itself (that is H_1 has not a head redex of the form $(\lambda x.U)V$, with $\lambda x.U$ an etrace of M_1).

In this case, the same head reductions can be performed (up to a substitution of M_1 by N_1) in the G_1N_1 side. Thus simply replacing G_1, we may freely assume that this head part is missing at all and thus σ is composed only of non-head β-reductions followed by η-reductions. Moreover, by our hypothesis, we can also assume that G_1 has not the form:

$$\lambda xy_1 \ldots y_p.\; xY_1 \cdots Y_q.$$

On the G_1N_1 side, the *Conditions on the Right Facing Arrows* may require a reduction of G_1N_1 to a suitable term H^+.

By the Church-Rosser Theorem and the cofinality of $\mathcal{X}$, let $\overline{H}$ be a term in $\mathcal{X}$, which is a common reduct of H^+ and H_2. Now, there exists a proof $\mathcal{T}'$ of $\overline{H} =_\omega N$, with $ord(\mathcal{T}') < ord(\mathcal{T})$. Thus by induction hypothesis there exists a canonical proof $\mathcal{T}_1$ of $\overline{H} =_\omega N$. Now, the required canonical proof is obtained by concatenating the component:

$$M \longrightarrow^*_{\beta\eta} H_1 \;{}^*_{\beta\eta}\!\longleftarrow G_1M_1 =_\omega G_1N_1 \longrightarrow^*_{\beta\eta} \overline{H};$$

with $\mathcal{T}_1$.

That this concatenation results in a canonical proof can be easily checked in case $\mathcal{T}_1$ ends in an instance of the ω-rule as well as in case $\mathcal{T}_1$ ends in an endpiece.

Second Subcase. An etrace of M_1 appears in functional position in a head redex of the head part of σ, or in H_1 itself.

Assume moreover that in the head part of σ, a λ appears at the beginning of some term in this head reduction.

Thus we have, for some U, $G_1 M_1 \longrightarrow^*_{\beta\eta} \lambda u.U \longrightarrow^*_{\beta\eta} H_1$. For any closed term R, consider the reduction:

$$G_1 M_1 R \longrightarrow^*_{\beta\eta} (\lambda u.U)R \longrightarrow_{\beta\eta} [R/u]U \longrightarrow^*_{\beta\eta} H'.$$

Where H' is $[R/u]H_1$. This can be done for every λ appearing in the head part of σ. Thus for each choice of closed $R_1 \dots R_n$ we have a standard β-reduction σ' of $G_1 M_1 R_1 \dots R_n$ to a term H", which is H_1 with each abstracted variable u_j substituted by the corresponding closed term R_j.

Now in the head reduction part of σ', we come to a term V with a head redex of the form: $(\lambda u.W)U$, where $M_1 \longrightarrow^*_{\beta\eta} \lambda u.W$; let $V \equiv (\lambda u.W)UU_1 \cdots U_v$, we write V in the form: $(\lambda u.W)[V_1/x_1, \dots, V_{r_1}/x_{r_1}]\overrightarrow{X}_1$, showing all the etraces of M_1 in V. Then:

$$(*)\ M_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1 =_\omega N_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1$$

has a proof with ordinal (much) less than $ord(\mathcal{T})$. Now, consider the component:

$$MR_1 \cdots R_n \longrightarrow^*_{\beta\eta} H\text{"}\ {}^*_{\beta\eta}\!\longleftarrow M_1[M_1/x_1, \dots, M_1/x_{r_1}]\overrightarrow{X}_1 =_\omega$$

$$=_\omega M_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1$$

The reduction $M_1[M_1/x_1, \dots, M_1/x_{r_1}]\overrightarrow{X}_1 \longrightarrow^*_{\beta\eta} H$" has a head part shorter than σ'. Thus, iterating the previous transformation for each occurrence M_1 in functional position in the head reduction part of σ', we arrive to a final sequence of terms $\overrightarrow{X}_s$ such that $M_1[M_1/x_1, \dots, M_1/x_{r_s}]\overrightarrow{X}_s$ is the last such occurrence of M_1. Therefore, for what concerns the component:

$$MR_1 \dots R_n \longrightarrow^*_{\beta\eta} H\text{"}\ {}^*_{\beta\eta}\!\longleftarrow M_1[M_1/x_1, \dots, M_1/x_{r_s}]\overrightarrow{X}_s =_\omega$$

$$=_\omega M_1[N_1/x_1, \dots, N_1/x_{r_s}]\overrightarrow{X}_s$$

we can argue as in the First Subcase above.

On the right hand side, observe that the iteration of the previous argument gives rise to a chain of equalities (where for simplicity, we do not consider reduction internal to M_1; this does not affect the argument):

$$\begin{array}{l}
N_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1 =_\omega NR_1 \dots R_n \\
N_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1 =_\omega M_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1 \\
M_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1 =_\omega M_1[M_1/x_1, \dots, M_1/x_{r_1}]\overrightarrow{X}_1 \\
M_1[M_1/x_1, \dots, M_1/x_{r_1}]\overrightarrow{X}_1 \longrightarrow^*_{\beta\eta} M_1[M_1/x_1, \dots, M_1/x_{r_2}]\overrightarrow{X}_2 \\
M_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_1 \longrightarrow^*_{\beta\eta} N_1[N_1/x_1, \dots, N_1/x_{r_2}]\overrightarrow{X}_2 \\
M_1[M_1/x_1, \dots, M_1/x_{r_2}]\overrightarrow{X}_2 =_\omega M_1[N_1/x_1, \dots, N_1/x_{r_2}]\overrightarrow{X}_2 \\
M_1[N_1/x_1, \dots, N_1/x_{r_2}]\overrightarrow{X}_2 =_\omega N_1[N_1/x_1, \dots, N_1/x_{r_2}]\overrightarrow{X}_2 \\
\dots \\
M_1[N_1/x_1, \dots, N_1/x_{r_s}]\overrightarrow{X}_s =_\omega N_1[N_1/x_1, \dots, N_1/x_{r_1}]\overrightarrow{X}_s
\end{array}$$

from this chain, by Fact 2 of Section 2, one obtains a proof of $M_1[N_1/x_1, \ldots, N_1/x_{r_s}]\overrightarrow{X}_s =_\omega NR_1 \ldots R_n$, with an ordinal less than $ord(\mathcal{T})$. We can also substitute $M_1[N_1/x_1, \ldots, N_1/x_{r_s}]\overrightarrow{X}_s$ with a suitable reduct $\overline{H}$, meeting the *Conditions on the Right Facing Arrows* w.r.t. $M_1[N_1/x_1, \ldots, N_1/x_{r_s}]\overrightarrow{X}_s$ and the cofinality condition w.r.t. $\mathcal{X}$. Still, $\overline{H} =_\omega NR_1 \ldots R_n$ has a proof with ordinal less than $ord(\mathcal{T})$. Thus by induction hypothesis there exists a canonical proof $\mathcal{T}_1$ of $\overline{H} =_\omega NR_1 \ldots R_n$. Now, we can concatenate the component:

$$MR_1 \ldots R_n \longrightarrow^*_{\beta\eta} H" \ {}^*_{\beta\eta}\!\longleftarrow (\lambda x.M_1[x/x_1, \ldots, x/x_{r_s}]\overrightarrow{X}_s)M_1 =_\omega$$

$$=_\omega (\lambda x.M_1[x/x_1, \ldots, x/x_{r_s}]\overrightarrow{X}_s)N_1 \longrightarrow^*_{\beta\eta} \overline{H};$$

with $\mathcal{T}_1$.

That this concatenation results in a canonical proof can be easily checked in case $\mathcal{T}_1$ ends in an instance of the ω-rule as well as in case $\mathcal{T}_1$ ends in an endpiece.

Thus we have proved the following: for every $R_1 \ldots R_n$, there exists a canonical proof of $MR_1 \ldots R_n =_\omega NR_1 \ldots R_n$.

Now, n applications of the ω-rule gives the required canonical proof of $M =_\omega N$.

Third Subcase. An etrace of M_1 appears in functional position in a head redex of the head part of σ. Assume moreover that in the head part of σ, no λ appears at the beginning of any term in this head reduction. This case can be treated as the previous one, with the difference that the resulting canonical proof ends in a canonical endpiece, rather than in an instance of the ω-rule. *QED*

4 Plotkin Terms

Recall that H, M, N, P, Q always denote closed terms. Let $\lceil M \rceil$ denote the Church numeral corresponding to the Gödel number of the term M. By Kleene's enumerator construction ([4] 8.1.6) there exists a combinator J such that $J\lceil M \rceil$ β-converts to M, for every M. The combinator J can be used to enumerate various r.e. sets of closed terms. In particular, let $\mathcal{X}$ be a r.e. set of terms, and let $T_\mathcal{X}$ be a term representing the r.e. function that enumerates $\mathcal{X}$. Set $\mathbf{E} \equiv \lambda x.J(T_\mathcal{X} x)$. It is well known that we can assume that $\mathbf{E}$ is in $\beta\eta$-normal form. We call $\mathbf{E}$ a *generator of* $\mathcal{X}$. As usual we shorten $\mathbf{E}\underline{n}$ with $\mathbf{E}_n$. We also suppress the dependency of $\mathbf{E}$ from J and $\mathcal{X}$, when it is clear from the context.

Now, by the methods of proof used in [8], which make use of modified forms of the celebrated *Plotkin terms* ([4] 17.3.26), one can prove the following:

Lemma 1. *Given a r.e. set of terms $\mathcal{X}$ and a generator $\mathbf{E}$ of $\mathcal{X}$, there exists a term H such that for every M the following holds:*
$H\mathbf{E}_0 =_\omega HM$ iff for some k, $M =_\omega \mathbf{E}_k$.

An extension of this technique can be used to determine whether two sets $\mathcal{X}$ and $\mathcal{Y}$, with generators $\mathbf{E}^\mathcal{X}$ and, respectively, $\mathbf{E}^\mathcal{Y}$, have non empty intersection (up to equality in $\lambda\omega$):

Lemma 2. *Given two sets of terms $\mathcal{X}$ and $\mathcal{Y}$, with generators $\mathbf{E}^{\mathcal{X}}$ and, respectively, $\mathbf{E}^{\mathcal{Y}}$, there exist two terms P and Q such that :*
$P\mathbf{E}^{\mathcal{X}}\mathbf{E}^{\mathcal{Y}} =_\omega Q\mathbf{E}^{\mathcal{X}}\mathbf{E}^{\mathcal{Y}}$ *iff for some k and j, $\mathbf{E}_k^{\mathcal{X}} =_\omega \mathbf{E}_j^{\mathcal{Y}}$.*

Now, consider r.e. sets of terms $\mathcal{X}[x]$, all containing a specified variable x as unique free variable. Let the generator $\mathbf{E}[x]$ be constructed accordingly. Of course $\mathbf{E}[x]$ contains the variable x. By again exploiting the above mentioned methods of [8], we can prove:

Lemma 3. *There exists a term $H \equiv \lambda x.U$, for some term U containing the free variable x as unique free variable, such that for every M and for every N:*
$[N/x]U([N/x]\mathbf{E}_0[x]) =_\omega [N/x]U\ M$ *iff for some k, $M =_\omega [N/x]\mathbf{E}_k[x]$*

Proof (Sketch). The lemma follows applying Lemma 1 to the set $\mathcal{X}[N/x]$. *QED*

Lemma 4. *There exist terms P' and Q' such that for every M: $P'M =_\omega Q'M$ iff for all k, $M \neq_\omega \underline{k}$.*

Proof. In [8], we constructed Plotkin terms P and Q such that for every n: $P\underline{n} =_\omega Q\underline{n}$ iff n is the Gödel number of a closed term which does not $\beta\eta$-convert to a Church numeral. Consider now the following two sets of terms:

$$\mathcal{X}[x] \equiv \{< \underline{n}, x, P\underline{n} > | n \in \mathbf{N}\} \text{ and } \mathcal{Y} \equiv \{< \underline{n}, J\underline{n}, Q\underline{n} > | n \in \mathbf{N}\}.$$

Let $\mathbf{E}^{\mathcal{X}[x]}[x]$ and, respectively, $\mathbf{E}^{\mathcal{Y}}$ be the corresponding generators.

By Lemma 2, for every M there exist terms P_M and Q_M such that: $P_M([M/x]\mathbf{E}^{\mathcal{X}[x]}[x])\mathbf{E}^{\mathcal{Y}} =_\omega Q_M([M/x]\mathbf{E}^{\mathcal{X}[x]}[x])\mathbf{E}^{\mathcal{Y}}$ iff for some k and j:

$$[M/x]\mathbf{E}_k^{\mathcal{X}[x]}[x] =_\omega \mathbf{E}_j^{\mathcal{Y}}$$

that is iff M does not $\beta\eta$-convert to a Church numeral (recall that $J\lceil M\rceil$ β-converts to M).

Now the construction can be made uniform on the parameter M, giving two terms, P' and Q' such that for every M:

$P'M =_\omega Q'M$ iff M does not $\beta\eta$-convert to a Church numeral, and, by Proposition 4 in [8], iff M is not ω-equal to a Church numeral. *QED*

Lemma 5. *Let P and Q be closed terms. Then we can construct terms F and G such that for every M, $PM =_\omega QM$ iff $FM =_\omega G$.*

Proof. This is done by intersecting the ω-closure of the two sets: $\mathcal{X}$ the set of all pairs $< N, N >$, N a closed term; $\mathcal{Y}$ the set of all pairs $< PM, QM >$, M a closed term. *QED*

Summing up the result of all the previous lemmas, we have the following proposition:

Proposition 2. *There exist two terms $\mathbf{H_1}$ and $\mathbf{H_2}$ such that for every M:*

$$\mathbf{H_1}M =_\omega \mathbf{H_2} \text{ iff for all } k, \quad M \neq_\omega \underline{k}.$$

We shall make extensive use of terms $\mathbf{H_1}$ and $\mathbf{H_2}$ in the following Section.

5 Barendregt Construction

We make the following definitions, which will hold in all the present and the next Section:

Definition 6

1. $\mathbf{\Theta} \equiv (\lambda ab.b(aab))(\lambda ab.b(aab))$ *(Turing's fixed point).*
2. $\mathbf{W} \equiv \lambda xy.xyy$
3. $J \equiv (\lambda xyz.\lambda abc.xy(z(yc))bac)$
4. $F \equiv \mathbf{\Theta} J \mathbf{H_1} \equiv (\lambda ab.b(aab))(\lambda ab.b(aab))(\lambda xyz.\lambda abc.xy(z(yc))bac)\mathbf{H_1}$
5. $G \equiv \mathbf{\Theta W H_2} \equiv (\lambda ab.b(aab))(\lambda ab.b(aab))(\lambda xy.xyy)\mathbf{H_2}$

Observe:
(i) $G \longrightarrow_{\beta\eta} \lambda b.b(\mathbf{\Theta} b)\mathbf{W H_2} \longrightarrow_{\beta\eta} \mathbf{W}(\mathbf{\Theta W})\mathbf{H_2} \longrightarrow_{\beta\eta} \mathbf{\Theta W H_2 H_2} \equiv G\mathbf{H_2}$
(ii) $FZABC \longrightarrow^*_{\beta\eta}$ (in 8 steps) $\mathbf{\Theta} J \mathbf{H_1}(Z(\mathbf{H_1} C))BAC \equiv FZ^*BAC$.

Let gk be the cofinal Gross-Knuth strategy defined in [4] 13.2.7. By writing $gk(M)$, we mean the term obtained by starting with the term M and applying (once) the gk strategy.

Then the reduction sequences:

(i) $G \longrightarrow^*_{\beta\eta} G(gk(\mathbf{H_2})) \longrightarrow^*_{\beta\eta} G\mathbf{H_2}(gk(\mathbf{H_2})) \longrightarrow^*_{\beta\eta}$
$\longrightarrow^*_{\beta\eta} G(gk(\mathbf{H_2}))(gk((gk(\mathbf{H_2})))) \longrightarrow^*_{\beta\eta} \cdots$
(ii) $FZABC \longrightarrow^*_{\beta\eta} F(gk(Z^*))(gk(B))(gk(A))(gk(C)) \longrightarrow^*_{\beta\eta}$
$F(gk((gk(Z^*))^*))(gk(gk(A)))(gk(gk(B)))(gk(gk(C))) \longrightarrow^*_{\beta\eta} \cdots$

are cofinal for $\beta\eta$-reductions starting with G and, respectively, with $FZABC$.

For the cofinal set of confluence terms we require only that in the reduction:

$$P \longrightarrow^*_{\beta\eta} H \; {}^*_{\beta\eta}\!\longleftarrow LQ =_\omega LR$$

if P $\beta\eta$-reduces to a term with F at the head then H begins with F and if P has positive order then H begins with λ.

Lemma 6. *If $GM_1 \ldots M_m =_\omega GN_1 \ldots N_m$ then, for each i, $1 \leq i \leq m$, $M_i =_\omega N_i$.*

Proof. By induction on the ordinal of a canonical proof $\mathcal{T}$ of $GM_1 \ldots M_m =_\omega GN_1 \ldots N_m$.

Basis. $ord(\mathcal{T})$ is 1. This case is clear since G is of order 0.

Induction step. Case 1. $\mathcal{T}$ ends in an application of the ω-rule. Apply the induction hypothesis to the subproof of $GM_1 \ldots M_m\mathbf{I} =_\omega GN_1 \ldots N_m\mathbf{I}$.

Induction step.Case 2. $\mathcal{T}$ has the endpiece $GM_1 \ldots M_m \longrightarrow^*_{\beta\eta} R_1 \; {}^*_{\beta\eta}\!\longleftarrow L_1P_1$ $=_\omega L_1Q_1 \longrightarrow^*_{\beta\eta} R_2 \; {}^*_{\beta\eta}\!\longleftarrow L_2P_2 =_\omega L_2Q_2 \longrightarrow^*_{\beta\eta} \cdots \longrightarrow^*_{\beta\eta} R_{t+1} \; {}^*_{\beta\eta}\!\longleftarrow$ $GN_1 \ldots N_m$.

Since $\mathcal{T}$ is canonical every term R_i has the form $\mathbf{\Theta W} H_1^* \ldots H_n^* M_1^* \ldots M_m^*$ where $H_j^* =_\omega \mathbf{H_2}$ and $M_i^* =_\omega M_i$ for $j = 1, \ldots, n$ and $i = 1, \ldots, m$. This completes the proof. *QED*

Lemma 7. *Suppose that:*

- $FL_1P_1Q_1R_1M_1 \ldots M_m =_\omega FL_2P_2Q_2R_2N_1 \ldots N_m$;
- $R_1 =_\omega R_2 =_\omega \underline{n}$, *for some* n;
- $L_1 =_\omega G(\mathbf{H_1}\underline{n}) \ldots (\mathbf{H_1}\underline{n})$, k *times;*
- $L_2 =_\omega G(\mathbf{H_1}\underline{n}) \ldots \mathbf{H_1}\underline{n})$, l *times.*

Then either $P_1 =_\omega P_2, Q_1 =_\omega Q_2$ *and* $k = l$ *mod 2 or* $P_1 =_\omega Q_2$, $Q_1 =_\omega P_2$ *and* $k = l{+}1$ *mod 2. (Where, possibly,* $k = 0$ *or* $l = 0$*.)*

Proof. By induction on the ordinal of a canonical proof of:
$FL_1P_1Q_1R_1M_1 \ldots M_m =_\omega FL_2P_2Q_2R_2N_1 \ldots N_m$.

Basis. The ordinal is 1 and we have a $\beta\eta$-conversion. Use a standard argument, taking into account that by Proposition 2 the copies of $\mathbf{H_2}$ are distinct, w.r.t. ω-equality, from the copies of $\mathbf{H_1}\underline{n}$. Therefore the β-reduction of G cannot affect the count of the copies of $\mathbf{H_1}\underline{n}$.

Induction step.

Case 1. The proof ends in an application of the ω-rule. Just apply the induction hypothesis to any of the premisses.

Case 2. The proof has a canonical endpiece beginning with a component:

$$FL_1P1Q_1R1M_1 \ldots M_m \longrightarrow^*_{\beta\eta} H \;{}^*_{\beta\eta}\!\longleftarrow LQ =_\omega LR \longrightarrow^*_{\beta\eta} H^+.$$

Now H has the same form as $FL_1P_1Q_1R_1M_1 \ldots M_m$ by the choice of the cofinal set. W.l.o.g. we can assume that the reduction from $FL_1P_1Q_1R_1M_1 \ldots M_m$ to H is a standard β-reduction followed by a sequence of η-reductions. The 8 term head reduction cycle of F with 4 arguments must be completed an integral number of times to result in a term which η-reduces to one with F at the head. Suppose that this cycle is completed s times and let $r = k + s$.

On the other hand, since the endpiece is canonical L, after a sequence (possibly empty) of η-reductions, reduces to a term of the form:

$$\lambda z.X_0X_2X_3X_4X_5X_6X_7X_8Y_1 \ldots Y_m$$

where $X_0 = \lambda x.X_1$.

This follows from the fact that we have to obtain H by internal reductions and Q is not substituted for a variable in functional position in a head redex.

It follows, using for some items Proposition 4 of [8], that:

- $[Q/z]X_0 \longrightarrow^*_{\beta\eta} \lambda ab.b(aab)$ and $[Q/z]X_2 \longrightarrow^*_{\beta\eta} \lambda ab.b(aab)$;
- $[Q/z]X_3 \longrightarrow^*_{\beta\eta} J$;
- $[Q/z]X_4 =_\omega \mathbf{H_1}$
- $[Q/z]X_5 =_\omega G\mathbf{H_2} \ldots \mathbf{H_2}(\mathbf{H_1}\underline{n}) \ldots (\mathbf{H_1}\underline{n})$,
  with t occurrences of $\mathbf{H_2}$, due to the possible β-reduction of G, and r occurrences of $\mathbf{H_1}\underline{n}$, since we have started with k copies of $\mathbf{H_1}\underline{n}$, and each reduction cycle of F adds a copy;

- $[Q/z]X_6 =_\omega P_1$ if $s \equiv 0$ *mod* 2 or $[Q/z]X_6 =_\omega Q_1$ if $s \equiv 1$ *mod* 2, this item, and the following one, results from the fact the each reduction cycle of F interchanges P_1 and Q_1;
- $[Q/z]X_7 =_\omega Q_1$ if $s \equiv 0$ *mod* 2 or $[Q/z]X_7 =_\omega P_1$ if $s \equiv 1$ *mod* 2;
- $[Q/z]X_8 \longrightarrow^*_{\beta\eta} \underline{n}$;
- $[Q/z]Y_i =_\omega M_i$, for every $1 \le i \le m$.

From the fact that $P =_\omega R$, and using again Proposition 4 of [8], we have:

- $[R/z]X_0 \longrightarrow^*_{\beta\eta} \lambda ab.b(aab)$ and $[R/z]X_2 \longrightarrow^*_{\beta\eta} \lambda ab.b(aab)$;
- $[R/z]X_3 \longrightarrow^*_{\beta\eta} J$;
- $[R/z]X_4 =_\omega \mathbf{H_1}$;
- $[R/z]X_5 =_\omega G\mathbf{H_2}\ldots\mathbf{H_2}(\mathbf{H_1}\underline{n})\ldots(\mathbf{H_1}\underline{n})$ with t occurrences of $\mathbf{H_2}$ and r occurrences of $\mathbf{H_1}\underline{n}$;
- $[R/z]X_6 =_\omega P_1$ if $s \equiv 0$ *mod* 2 or $[R/z]X_6 =_\omega Q_1$ if $s \equiv 1$ *mod* 2;
- $[R/z]X_7 =_\omega Q_1$ if $s \equiv 0$ *mod* 2 or $[R/z]X_7 =_\omega P_1$ if $s \equiv 1$ *mod* 2;
- $[R/z]X_8 \longrightarrow^*_{\beta\eta} \underline{n}$;
- $[R/z]Y_i =_\omega M_i$, for every $1 \le i \le m$.

Observe moreover that H^+, because of its construction, has the same form as H (up to some η-reductions). Say $H^+ \equiv FL_1^+P_1^+Q_1^+\underline{n}M_1^+\ldots M_m^+$. Moreover, we can freely assume that L_1^+ is $G(\mathbf{H_1}\underline{n})\ldots(\mathbf{H_1}\underline{n})$ with no occurrence of $\mathbf{H_2}$ and r occurrences of $\mathbf{H_1}\underline{n}$. This amounts to start with a different term and then perform t β-reductions of G. By Proposition 2 the copies of $\mathbf{H_2}$ are distinct, w.r.t. ω-equality, from the copies of $\mathbf{H_1}\underline{n}$. Therefore the β-reduction of G cannot affect the count of the copies of $\mathbf{H_1}\underline{n}$.

The part of the proof beginning with H^+ is a canonical proof that $H^+ =_\omega FL_2P_2Q_2R_2N_1\ldots N_m$ because the cofinality restriction met for LR also works for H^+ . Thus the induction hypothesis applies to this proof.

Now the idea is that r and l have "to be in accordance" by induction hypothesis. On the other hand k differs from r only for s cycles of F, and therefore they behave in the right way. So the required property is obtained by transitivity. Formally:

Subcase 2.1. $P_1^+ =_\omega P_2$, $Q_1^+ =_\omega Q_2$ and $r \equiv l$ *mod* 2.
In case s is even we have $P_1 =_\omega P_2$ and $Q_1 =_\omega Q_2$ and $k \equiv l$ *mod* 2. In case s is odd we have k and l with opposite parity and $Q_1 =_\omega P_1^+ =_\omega P_2$, $P_1 =_\omega Q_2^+ =_\omega Q_2$.

Subcase 2.2. $P_1^+ =_\omega Q_2$, $Q_1^+ =_\omega P_2$ and $r \equiv l+1$ *mod* 2.
In case s is even we have $P_1 =_\omega Q_2$ and $Q_1 =_\omega P_2$ and $k \equiv l+1$ *mod* 2. In case s is odd we have k and l with same parity and $P_1 =_\omega Q_1^+ =_\omega P_2$, $Q_1 =_\omega P_2^+ =_\omega Q_2$.
This completes the proof. *QED*

5.1 Well Founded Trees

We assume that we have encoded sequences of numbers as numbers, with 0 encoding the empty sequence. $< n >$ is the sequence consisting of n alone (singleton) and $*$ is the concatenation function. For simplicity, we shall use these

notations ambiguously for the corresponding λ-terms. We require only that the term $y * \ < z >$ is in $\beta\eta$-normal form with $z\mathbf{II}$ at its head (this construction can be obtained "making normal" a term representing $*$, see [14]).

Our proof of $\mathbf{\Pi^1_1}$-completeness of $\lambda\omega$ is inspired by the argument in Section 17.4 of [4] (however, we will substantially modify Barendregt's construction). The starting point is the following well known theorem (see [7] Ch.16 Th.20):

Theorem 1. *The set of (indices of) well founded recursive trees is* $\mathbf{\Pi^1_1}$*-complete.*

and the idea is to reduce the well-foundness of a recursive tree to the equality of two suitable terms in $\lambda\omega$.

Suppose that we have a primitive recursive tree $\mathbf{t}$ with a representing term $\mathbf{T}$ such that

$$\mathbf{T}\underline{n} \longrightarrow^*_{\beta\eta} \begin{cases} \mathbf{K} & \text{if } n \text{ is the number of a sequence in } \mathbf{t}; \\ \mathbf{K}^* & \text{otherwise.} \end{cases}$$

Define:

$$\begin{aligned} A \equiv_{def} & \ \mathbf{\Theta}(\lambda x.\lambda a.a(\lambda y.\mathbf{T}y(\lambda z.FG(x\mathbf{K}(y * \ < z >)) \\ & (x\mathbf{K}^*(y * \ < z >))z))(\lambda y.\mathbf{T}y(\lambda z.FG(x\mathbf{K}^*(y * \ < z >)) \\ & (x\mathbf{K}(y * \ < z >))z)))\mathbf{K} \end{aligned}$$

$$\begin{aligned} B \equiv_{def} & \ \mathbf{\Theta}(\lambda x.\lambda a.a(\lambda y.\mathbf{T}y(\lambda z.FG(x\mathbf{K}(y * \ < z >)) \\ & (x\mathbf{K}^*(y * \ < z >))z))(\lambda y.\mathbf{T}y(\lambda z.FG(x\mathbf{K}^*(y * \ < z >)) \\ & (x\mathbf{K}(y * \ < z >))z)))\mathbf{K}^* \end{aligned}$$

Clearly:

$$\begin{aligned} A &\longrightarrow^*_{\beta\eta} \lambda y.\mathbf{T}y(\lambda z.FG(A(y * \ < z >))(B(y * \ < z >))z) \\ B &\longrightarrow^*_{\beta\eta} \lambda y.\mathbf{T}y(\lambda z.FG(B(y * \ < z >))(A(y * \ < z >))z) \end{aligned}$$

Now we state a corollary to Lemma 7.

Corollary 1. *If* $FG(A\underline{n})(B\underline{n})\underline{n}M_1 \dots M_m \ =_\omega \ FG(B\underline{n})(A\underline{n})\underline{n}N_1 \dots N_m$ *then* $A\underline{n} =_\omega B\underline{n}$*.*

Lemma 8. *If the subtree* $\mathbf{t}(n)$ *of the tree* $\mathbf{t}$ *rooted at* n *is well-founded then* $A\underline{n} =_\omega B\underline{n}$*.*

Proof. By induction on the ordinal of the subtree $\mathbf{t}(n)$, which is defined in the natural way. Note that if n is not the number of a sequence in the tree then $\mathbf{T}\underline{n} \longrightarrow^*_{\beta\eta} \mathbf{K}^*$ so $A\underline{n} \longrightarrow^*_{\beta\eta} \mathbf{I} \ {}^*_{\beta\eta}\!\longleftarrow B\underline{n}$.

Basis. The ordinal is 0 so the tree $\mathbf{t}(n)$ contains only the empty sequence. Suppose that 0 is the number of the empty sequence. Then:

$$\begin{aligned} & A\underline{0} \longrightarrow^*_{\beta\eta} \lambda z FG(A(\underline{0} * \ < z >))(B(\underline{0} * \ < z >))z B\underline{0} \longrightarrow^*_{\beta\eta} \\ & \longrightarrow^*_{\beta\eta} \lambda z FG(B(\underline{0} * \ < z >))(A(\underline{0} * \ < z >))z \end{aligned}$$

and if N $\beta\eta$-converts to a Church numeral then:

$$A\underline{0}N \longrightarrow^*_{\beta\eta} FG\mathbf{II}N \;{}^*_{\beta\eta}\!\longleftarrow B\underline{0}N$$

and if N does not beta eta convert to a Church numeral then

$$\begin{array}{l} A\underline{0}N \longrightarrow^*_{\beta\eta} FG(A(\underline{0} * < N >))(B(\underline{0} * < N >))N \longrightarrow^*_{\beta\eta} \\ \longrightarrow^*_{\beta\eta} F(G(\mathbf{H_1}N))(B(\underline{0} * < N >))(A(\underline{0} * < N >))N =_\omega \\ =_\omega F(G\mathbf{H_2})(B(\underline{0} * < N >))(A(\underline{0} * < N >))N \;{}^*_{\beta\eta}\!\longleftarrow B\underline{0}N. \end{array}$$

Induction Step. The ordinal of the subtree rooted at n is larger than 0. We have:

$$\begin{array}{l} A\underline{n} \longrightarrow^*_{\beta\eta} \lambda z.FG(A(\underline{n} * < z >))(B(\underline{n} * < z >))z \\ B\underline{n} \longrightarrow^*_{\beta\eta} \lambda z.FG(B(\underline{n} * < z >))(A(\underline{n} * < z >))z \end{array}$$

now, if N $\beta\eta$-converts to a Church numeral, then:

$$A\underline{n}N \longrightarrow^*_{\beta\eta} FG(A(\underline{n} * < N >))(B(\underline{n} * < N >))N =_\omega$$

(by induction hypothesis)

$$=_\omega FG(B(\underline{n} * < N >))(A(\underline{n} * < N >))N \;{}^*_{\beta\eta}\!\longleftarrow B\underline{n}N$$

and if N does not $\beta\eta$-convert to a Church numeral, then:

$$\begin{array}{l} A\underline{n}N \longrightarrow^*_{\beta\eta} FG(A(\underline{n} * < N >))(B(\underline{n} * < N >))N \longrightarrow^*_{\beta\eta} \\ \longrightarrow^*_{\beta\eta} F(G(\mathbf{H_1}N)(B(\underline{n} * < N >))(A(\underline{n} * < N >))N =_\omega \\ F(G\mathbf{H_2})(B(\underline{n} * < N >))(A(\underline{n} * < N >))N \;{}^*_{\beta\eta}\!\longleftarrow B\underline{n}N. \end{array}$$

So by the ω-rule $A\underline{n} =_\omega B\underline{n}$. This completes the proof. *QED*

Proposition 3. *$A\underline{n} =_\omega B\underline{n}$ iff the subtree $\mathbf{t}(n)$ rooted at n is well-founded or n is not in the tree $\mathbf{t}$.*

Proof. Consider all canonical proofs of smallest ordinal of $A\underline{n} = B\underline{n}$ for n in the tree $\mathbf{t}$, and assume the subtree $\mathbf{t}(n)$ rooted at n is not well-founded. Let $\mathcal{T}$ be such a proof.

Case 1. $\mathcal{T}$ is a $\beta\eta$-conversion. It is easily seen that this is impossible.

Case 2. $\mathcal{T}$ ends in the ω-rule. Then for each m, $A\underline{nm} =_\omega B\underline{nm}$ has a canonical proof of smaller ordinal. Now:

$$\begin{array}{l} A\underline{nm} \longrightarrow^*_{\beta\eta} \lambda y.\mathbf{T}y(\lambda z.FG(A(y * < z >))(B(y * < z >))z)\underline{nm} \longrightarrow_{\beta\eta} \\ \mathbf{T}\underline{n}(\lambda z.FG(A(\underline{n} * < z >))(B(\underline{n} * < z >))z)\underline{m} \longrightarrow^*_{\beta\eta} \\ (\lambda z.FG(A(\underline{n} * < z >))(B(\underline{n} * < z >))z)\underline{m} \longrightarrow_{\beta\eta} \\ FG(A(\underline{n} * < \underline{m} >))(B(\underline{n} * < \underline{m} >))\underline{m} \end{array}$$

and reducing in a similar way $B\underline{nm}$, we see that

$$\begin{array}{l} FG(A(\underline{n} * < \underline{m} >))(B(\underline{n} * < \underline{m} >))\underline{m} =_\omega \\ =_\omega FG(B(\underline{n} * < \underline{m} >))(A(\underline{n} * < \underline{m} >))\underline{m} \end{array}$$

has a proof of the same (smaller) ordinal. Thus by Lemma 7, either

$$A(\underline{n} * < \underline{m} >) =_\omega B(\underline{n} * < \underline{m} >)$$

has a proof with the same or smaller ordinal or, by Lemma 6, $\mathbf{H_1}\underline{n} =_\omega \mathbf{H_2}$.

Now the second alternative is impossible by Proposition 2, thus by induction hypothesis, the extension of $n* < m >$ in the tree is well-founded.

So, every extension of n in the tree is well-founded. Thus the subtree rooted at n is well-founded. This contradicts the choice of n.

Case 3. $\mathcal{T}$ has an endpiece.

Now, a canonical proof with endpiece would have to begin:
$A\underline{n} \longrightarrow^*_{\beta\eta} (\lambda z.FG(A(\underline{n} * < z >))(B(\underline{n} * < z >))z) \ldots$, by choice of confluence terms.

Now for each m we have a proof with the same ordinal as:

$$A\underline{n}\underline{m} \longrightarrow^*_{\beta\eta} (\lambda z.FG(A(\underline{n} * < z >))(B(\underline{n} * < z >))z)\underline{m} \longrightarrow^*_{\beta\eta} \ldots \; {}^*_{\beta\eta}\!\longleftarrow B\underline{n}\underline{m}.$$

Now consider that the endpiece is separated so that each H_i has the same form as

$$[M_i/x]G_i, \text{ for each } i.$$

So, to equalize $FG(A(\underline{n} * < \underline{m} >))(B(\underline{n} * < \underline{m} >))\underline{m}$ with $FG(B(\underline{n} * < \underline{m} >))(A(\underline{n} * < \underline{m} >))\underline{m}$, it is necessary that some of instances of the ω-rule, occurring in the endpiece, supplies a proof of $A(\underline{n} * < \underline{m} >) =_\omega B(\underline{n} * < \underline{m} >)$.

So by the previous case, the extension of $n* < m >$ in the tree is well-founded. Thus every extension of n in the tree is well-founded and again we contradict the choice of n. This completes the proof. *QED*

Proposition 4. *The set* $\{(M,N)|M =_\omega N\}$ *is* $\mathbf{\Pi^1_1}$*-complete.*

Proof. It easy to see that equality in $\lambda\omega$ is $\mathbf{\Pi^1_1}$. On the other hand, given any recursive tree $\mathbf{t}$ construct the terms A and B (observe that the construction is effective and uniform on (the term $\mathbf{T}$ representing) $\mathbf{t}$). Then use Proposition 3 to determine (*via* equality in $\lambda\omega$) if $\mathbf{t}$ is well founded. *QED*

Acknowledgements

We thank an anonymous referee for her/his help in substantially improving a previous version of the paper.

References

1. URL: http://coq.inria.fr
2. URL: http://www.cl.cam.ac.uk/Research/HVG/Isabelle
3. URL: http://www.cl.cam.ac.uk/Research/HVG/HOL

4. Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. North-Holland, Amsterdam (1984)
5. Böhm, C. (ed.): Lambda-Calculus and Computer Science Theory. LNCS, vol. 37. Springer, Heidelberg (1975)
6. Flagg, R.C., Myhill, J.: Implication and Analysis in Classical Frege Structure. Annals of Pure. and Applied Logic 34, 33–85 (1987)
7. Rogers Jr., H.: Theory of Recursive Functions and Effective Computability. MacGraw Hill, New York (1967)
8. Intrigila, B., Statman, R.: The Omega Rule is Π^0_2-Hard in the $\lambda\beta$-Calculus. LICS 2004, pp. 202–210. IEEE Computer Society, Los Alamitos (2004)
9. Intrigila, B., Statman, R.: Some Results on Extensionality in Lambda Calculus. Annals of Pure. and Applied Logic 132(2-3), 109–125 (2005)
10. Intrigila, B., Statman, R.: Solution of a Problem of Barendregt on Sensible λ-Theories. Logical Methods in Computer Science, vol. 2(4) (2006)
11. Plotkin, G.: The λ-Calculus is ω-incomplete. J. Symbolic Logic, vol. 39, pp. 313–317
12. Schütte, K.: Proof Theory. Springer, Berlin Heidelberg (1977)
13. Statman, R.: Gentzen's Notion of a Direct Proof. In: Barwise, K.J. (ed.) Handbook of Mathematical Logic, North Holland, Amsterdam (1978)
14. Statman, R.: Normal Varieties of Combinators. In: Moschovakis, Y.N. (ed.) Logic from Computer Science, pp. 585–596. Springer, Berlin Heidelberg (1992)

Weakly Distributive Domains*

Ying Jiang[1] and Guo-Qiang Zhang[2]

[1] State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China
jy@ios.ac.cn
[2] Department of EECS, Case Western Reserve University
Cleveland, Ohio 44022, USA
gq@case.edu

Abstract. In our previous work [17] we have shown that for any ω-algebraic meet-cpo D, if all higher-order stable function spaces built from D are ω-algebraic, then D is finitary. This accomplishes the first of a possible, two-step process in solving the problem raised in [1,2]: whether the category of stable bifinite domains of Amadio-Droste-Göbel [1,6] is the largest cartesian closed full sub-category within the category of ω-algebraic meet-cpos with stable functions. This paper presents results on the second step, which is to show that for any ω-algebraic meet-cpo D satisfying axioms M and I to be contained in a cartesian closed full sub-category using ω-algebraic meet-cpos with stable functions, it must not violate MI^{∞} . We introduce a new class of domains called *weakly distributive domains* and show that for these domains to be in a cartesian closed category using ω-algebraic meet-cpos, property MI^{∞} must not be violated. We further demonstrate that principally distributive domains (those for which each principle ideal is distributive) form a proper subclass of weakly distributive domains, and Birkhoff's M_3 and N_5 [5] are weakly distributive (but non-distributive). We introduce also the notion of meet-generators in constructing stable functions and show that if an ω-algebraic meet-cpo D contains an infinite number of meet-generators, then $[D \to D]$ fails I. However, the original problem of Amadio and Curien remains open.

1 Introduction

Domains are order-theoretic structures initiated by Dana Scott in the late 1960s for suitable mathematical spaces to accommodate denotations of programs. Their rich structural properties are often manifested collectively as categorical properties such as cartesian closedness, with important computational consequences. The interplay between domain theory and denotational semantics of programming languages is much inspired by the pursue of "full abstraction" [9]. Full completeness addresses the related problem of ensuring that mathematical spaces naturally generated by a certain set of base domains (the interpretation of base types) using computationally meaningful categorical constructs do not contain computationally irrelevant elements.

* This work is partially supported by NSFC 60673045, NSFC 60373050, NSFC major research program 60496321 and NSFC 60421001.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 194–206, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Stable domain theory was initiated by Berry [3] in the late 1970s as an attempt to cut down computationally irrelevant elements typically found in function spaces based on Scott continuity alone. The hallmark of stable domain theory is that an element realizing finite computation never involves an approximation sequence beyond finitely many steps.

As with many scientific developments, although stable domain theory itself turned out not to be the solution to the full abstraction problem that motivated it, it has since played significant roles in linear logic (Girard [8]), concurrency (Winskel [13]), polymorphism in PCF (Coquand [4]), and object-oriented programming (Reddy [11]).

The existence of a variety of cartesian closed categories of domains motivated a systematic investigation of the question of "largest cartesian closed categories of domains", starting with the work of Smyth [12]. Similar development on stable domains has occurred only more recently. The second author showed [15] that Berry's category of dI-domains is the largest cartesian closed category inside the category of Scott domains (which are bounded complete) with stable functions. In [1,6], appropriate notions of stable domains beyond the bounded complete ones were investigated, in an effort to provide an understanding of how the stable order may be extended to SFP-like domains [10]. An interesting new category called *stable bifinite domains* was introduced in [1,6]. An important conceptual question (see Amadio and Curien [2], pages 287–291) is: whether the category of stable bifinite domains of Amadio-Droste-Göbel [1,6] is the largest cartesian closed full sub-category of the category of ω-algebraic meet-cpos with stable functions.

The paper by Amadio [1] presents a first explicit formulation and serious attack on this open problem. The main result of [1] is that the finite ascending chain condition and finite descending chain condition must be maintained in any stable cartesian closed category composed of ω-algebraic meet-cpos. These conditions account for two of the three cases for which a principal ideal determined by a compact element may violate axiom I (i.e., a compact element dominates only finitely many elements). The third case is the finite antichain condition: any principal ideal determined by a compact element must not contain an infinite antichain. In recent work [17] we solved the finite antichain case which leads to the immediate conclusion that axiom I must be maintained in any cartesian closed full sub-category composed of ω-algebraic meet-cpos with stable functions. This accomplishes the first of a two step process in solving the problem raised by Amadio and Curien. This paper presents results on the second step, which is to show that for any ω-algebraic meet-cpo D satisfying axioms M and I to be contained in a cartesian closed full sub-category using ω-algebraic meet-cpos with stable functions, it must not violate MI^{∞} .

In this paper we introduce a new class of domains called *weakly distributive domains* and show that for these domains to be in a cartesian closed full sub-category using ω-algebraic meet-cpos with stable functions, it must not violate MI^{∞} . We further demonstrate that principally distributive domains (those for which each principle ideal is distributive) form a proper subclass of weakly distributive domains, and Birkhoff's M_3 and N_5 [5] are weakly distributive (but non-distributive). We introduce also the notion of meet-generators in constructing stable functions and show that if an ω-algebraic

meet-cpo D contains an infinite number of generators, then $[D \to D]$ fails I. However, the original problem of Amadio and Curien remains open.

2 Preliminaries

We briefly summarize the relevant results in [17] in this section to fix notations and set the background for this paper.

By convention, we use $\downarrow x$ for the lower set $\{y \mid y \sqsubseteq x\}$ and $\uparrow x$ for the upper set $\{y \mid y \sqsupseteq x\}$. Also, by $a \uparrow b$ we mean that a and b are compatible, that is, there exists an element z such that $a \sqsubseteq z$ and $b \sqsubseteq z$, and $a \not\uparrow b$ denotes a and b are incompatible.

The basic property of a *conditional multiplicative* (cm) function is that it preserves the meet of any pair of compatible elements. Thus bounded meets should exist for stability to make sense (item (a) below). Meet should also interact smoothly with the join of any directed set (item (b) below). The stable order then arises naturally from the minimal requirement that the evaluation map (for cartesian closure) is stable [3].

Definition 1. *Let D be a dcpo (with bottom). It is called a meet-cpo if*

(a) for any $x, y \in D$, $x \sqcap y$ exists when $\{x, y\}$ is bounded above (or compatible),

(b) if $R \subseteq D$ is a directed set and x is compatible with the join of R, then

$$x \sqcap (\bigsqcup R) = \bigsqcup \{x \sqcap r \mid r \in R\}.$$

Beyond Scott domains and inside ω-algebraic domains there are the *stable bifinite domains* [1] which also form a cartesian closed category [6,7]. Stable bifinite domains are ω-algebraic meet-cpos for which the identity function can be expressed as the join (under the stable order) of a directed set of stable projections with finite images.

Some notational preparation is needed for the concept of stable bifinite domain. Let $\mathsf{mub}(X)$ be the set of minimal upper bounds (mubs) of X. Let $\bowtie(X) := \bigcup\{\mathsf{mub}(Y) \mid Y \subseteq_{\mathrm{fin}} X\}$, where $\subseteq_{\mathrm{fin}}$ denotes the "finite subset" relation. A set is called *mub-closed* if $\bowtie(X) = X$. An SFP domain, according to Plotkin, is an ω-algebraic cpo with property M, such that every finite set X of compact elements has a finite mub-closure. Stable bifinite domains are similar to SFP domains, but a stronger condition holds: for any finite set of compact elements, there is a finite superset, closed under the combination of down-closure and mub-closure. More precisely, let $(\mathsf{mub}, \mathsf{down})(X) := \downarrow(\bowtie(X))$. A set X is called *mub-down-closed* if $(\mathsf{mub}, \mathsf{down})(X) = X$.

Definition 2 (Stable Bifinite Domain). *An ω- algebraic meet-cpo is said to have property I and called finitary if every compact element dominates a finite number of elements. It is said to have property M if for every finite set X of compact elements, $\mathsf{mub}(X)$ is finite and complete – complete in the sense that each upper bound of X dominates some member of $\mathsf{mub}(X)$. It is called a stable bifnite domain if every finite set of compact elements is contained in a finite $(\mathsf{mub}, \mathsf{down})$-closed set. This last property is denoted as MI^{∞}.*

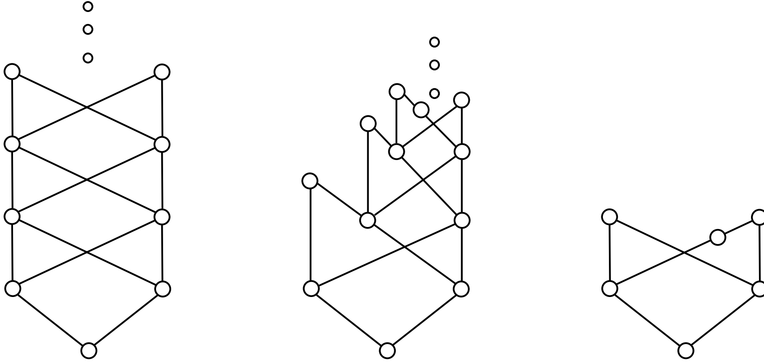


Fig. 1. Examples from left to right: a domain (not a meet-cpo) satisfying M but is not SFP; a SFP meet-cpo which is not stable bifinite; a stable bifinite domain

Definition 3. *Let D, E be meet-cpos. A Scott continuous function f from D to E is called* stable *if it preserves meets of compatible pairs,* i.e.*, for all x, y in D (where ↑ stands for compatibility or bounded-above),*

$$x \uparrow y \Rightarrow f(x \sqcap y) = f(x) \sqcap f(y).$$

The stable function space $[D \to E]$ consists of all stable functions from D to E under the Berry order: f is stably less than g, written $f \sqsubseteq_s g$, if for all x, y in D,

$$x \sqsubseteq y \Rightarrow f(x) = f(y) \sqcap g(x).$$

In the rest of the paper, we drop the subscript s when stable functions are compared, so $f \sqsubseteq g$ always means $f \sqsubseteq_s g$ unless stated otherwise.

Let **SB** be the category of stable bifinite domains with stable functions (under the Berry order for function space). We have the following [1,6].

Theorem 1. *The category* **SB** *is a cartesian closed category.*

We now recall the technical tool of (mub, meet)-closed sets which will be helpful to the understanding of the development in the rest of the paper.

Definition 4. *[17] Let D be an ω-algebraic meet-cpo. A set Y of compact elements in D is said to be a* (mub, meet)*-closed set if both of the following are true:*

(a) it is closed under minimal upper bounds of finite sets,
(b) it is closed under bounded meets of pairs of elements.

Clearly, every (mub, down)-closed set is (mub, meet)-closed. Moreover, for every X, $(\mathsf{mub},\mathsf{meet})(X) \subseteq (\mathsf{mub},\mathsf{down})(X)$. However, (mub, meet)-closed sets provide a more flexible and general way for constructing stable functions.

Lemma 1. *[17] Suppose D is an ω-algebraic meet-cpo with property* M*. Then every* (mub, meet)*-closed set A determines a stable function $\phi_A : D \to D$, given by*

$$\phi_A := \lambda x. \bigsqcup (\downarrow x \cap A).$$

An immediate consequence of this lemma is that $(\mathsf{mub}, \mathsf{down})$-closed sets determine stable functions. These are projections, dominated by the identity function under the stable order.

Lemma 2. *[17] Let ϕ_A be the stable function determined by a $(\mathsf{mub}, \mathsf{meet})$-closed set A as given in the previous lemma. Then*

(a) A is the set of compact fixed-points of ϕ_A, and

(b) if $f \sqsubseteq \phi_A$ and $f(x) = x$ for each $x \in A$, then $f = \phi_A$, where $\sqsubseteq$ denotes the extensional order.

The next lemma shows how stable functions determined by $(\mathsf{mub}, \mathsf{meet})$-closed sets can be compared.

Lemma 3. *[17] Suppose A, B are $(\mathsf{mub}, \mathsf{meet})$-closed sets. The following are equivalent:*

(a) $\phi_B \sqsubseteq \phi_A$;

(b) $\downarrow B \cap A = B$;

(c) $B \subseteq A$ and for each bounded $\{x, y\}$, if $x \in B$ and $y \in A$, then $x \sqcap y \in B$.

When a set X of compact elements is not already $(\mathsf{mub}, \mathsf{meet})$-closed, we can work with the $(\mathsf{mub}, \mathsf{meet})$-closed set *generated* by X, which is the smallest set of compact elements containing X and closed under minimal upper bounds of finite subsets and bounded meets. Such a generated set always exists in an ω-algebraic meet-cpo with both property M and the property that the meet of two compact elements is compact. In such case the closure exists and can be defined inductively:

$$
\begin{aligned}
&(\mathsf{mub}, \mathsf{meet})^0(X) := X \\
&(\mathsf{mub}, \mathsf{meet})^{(i+1)}(X) \\
&\qquad := (\mathsf{mub}, \mathsf{meet})((\mathsf{mub}, \mathsf{meet})^i(X)) \\
&(\mathsf{mub}, \mathsf{meet})^*(X) := \textstyle\bigcup_{i \geq 0} (\mathsf{mub}, \mathsf{meet})^i(X)
\end{aligned}
$$

Clearly, $(\mathsf{mub}, \mathsf{meet})^*(X)$ is the least $(\mathsf{mub}, \mathsf{meet})$-closed set containing X.

Lemma 4. *[17] Let Y be a $(\mathsf{mub}, \mathsf{meet})$-closed set generated by a finite set Y^0. Then the stable function ϕ_Y is compact.*

With respect to an ω-algebraic meet-cpo D, property I amounts to three more primitive ones. The most difficult case among the three is when D satisfies the finite descending chain condition and the finite ascending chain condition, but fails the finite antichain condition. This was resolved in [17].

Theorem 2. *[17] Suppose D is an ω-algebraic meet-cpo which satisfies the finite descending chain and finite ascending chain conditions, but fails the finite antichain condition. Then in the stable function space $[D \to D]$ there exists a finite set of compact stable functions with an infinite number of minimal upper bounds.*

3 Weak Distributivity

Theorem 2 shows that property I is maintained in any cartesian closed category within the space of ω-algebraic meet-cpos. A key technique used in [17] is $(\mathsf{mub}, \mathsf{meet})$-closed set, which gives a rich class of stable functions to work with. The remaining question is whether property MI^∞ is similarly maintained in any cartesian closed category within the space of ω-algebraic meet-cpos.

Property MI^∞ states that a finite set of compact elements has a finite $(\mathsf{mub}, \mathsf{down})$-closure. A reasonable strategy is to show that for any ω-algebraic meet-cpo D satisfying axioms M and I (assumed for the rest of the paper), if D violates MI^∞ , then $[D \to D]$ (or some even higher-order function space) violates either M or I. For this, we explore stable functions determined by the down-closure of a compact element. Of course, not every down-closure of a compact element in D determines a stable function in $[D \to D]$. For a compact element $c \in D$ to give rise to a stable function

$$\lambda x. \bigsqcup^{x} (\downarrow x \cap \downarrow c),$$

conditional multiplicity requires that for any $x \uparrow y$,

$$\bigsqcup^{x} (\downarrow x \cap \downarrow c) \sqcap \bigsqcup^{y} (\downarrow y \cap \downarrow c) = \bigsqcup^{x \sqcap y} (\downarrow (x \sqcap y) \cap \downarrow c),$$

and this leads to weak distributivity. To put this in context, recall that distributivity property states that

$$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$$

holds for all compatible triples x, y, z. Configurations violating distributivity include Birkhoff's famous M_3 and N_5 posets. We consider a weaker version of distributivity, in the next definition.

Definition 5 (Weakly distributive domains). *An ω-algebraic meet-cpo D with properties M and I is said to be weakly distributive if for any $z \in D^0$,*

$$x \sqcap \bigsqcup^{y} \{d \mid d \sqsubseteq y \ \& \ d \sqsubseteq z\} = \bigsqcup^{y} \{d \sqcap x \mid d \sqsubseteq y \ \& \ d \sqsubseteq z\}$$

for all compatible $x, y \in D$, where D^0 is the set of compact elements of D.

Lemma 5. *An ω-algebraic meet-cpo D with properties M and I is weakly distributive if and only if for any $x, y \in D$ and $z \in D^0$ with x compatible with y, we have*

$$x \sqcap \bigsqcup^{y} \downarrow y \cap \downarrow z = \bigsqcup^{y} \downarrow (x \sqcap y) \cap \downarrow z.$$

Lemma 6. *In reference to Def. 5, weak distributivity law holds for all of the following configurations: (1) $x \sqsubseteq z$; (2) $z \sqsubseteq x$; (3) $y \sqsubseteq z$; (4) $z \sqsubseteq y$; (5) $y \sqsubseteq x$.*

This lemma reduces the non-trivial configurations to check for weak distributivity to the cases when x and z are incomparable, y and z are incomparable, and y is not dominated by x.

Lemma 7. *An ω-algebraic meet-cpo D with properties* M *and* I *is weakly distributive if and only if for any $x \sqsubseteq y$ and $z \in D^0$, we have*

$$x \sqcap \bigsqcup^{y} \downarrow y \cap \downarrow z = \bigsqcup^{y} \downarrow x \cap \downarrow z.$$

Proof. It suffices to show that if the condition in the lemma holds, then for any compatible $x', y' \in D$ and any $z \in D^0$, we have

$$x' \sqcap \bigsqcup^{y'} \downarrow y' \cap \downarrow z = \bigsqcup^{y'} \downarrow (x' \sqcap y') \cap \downarrow z.$$

This is because

$$x' \sqcap \bigsqcup^{y'} \downarrow y' \cap \downarrow z = x' \sqcap y' \sqcap \bigsqcup^{y'} \downarrow y' \cap \downarrow z,$$

$x' \sqcap y' \sqsubseteq y'$, and one can take $x = x' \sqcap y'$, $y = y'$ and invoke the given assumption. □

With these in mind, one readily checks that *Birkhoff's M_3 and N_5 are both weakly distributive*. Therefore, weakly distributive domains need not be distributive. On the other hand, it is easy to see that all distributive domains (i.e., ω-algebraic meet-cpos satisfying M and I and distributivity) are weakly distributive.

It is important to note that there are domains that are not weakly distributive. Here are some examples.

Example 1. The meet-cpos below are not weakly distributive. Notice that they are all stable bifinite.

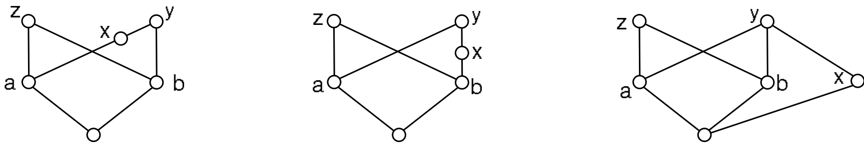


Fig. 2. Examples of stable bifinite domains that are not weakly distributive

To check the example on the left of Fig. 2, note that we have

$$\bigsqcup^{y} \downarrow (x \sqcap y) \cap \downarrow z = a,$$

but

$$x \sqcap \bigsqcup^{y} \downarrow y \cap \downarrow z = x \sqcap y = x.$$

Similarly one can check that the other two domains are not weakly distributive.

It is interesting to compare weak distributivity with *principal distributivity*, meet-cpos for which each principal ideal is distributive. We have the following result.

Lemma 8. *Any principally distributive domain is weakly distributive, but not vice versa.*

Proof. The second part has been demonstrated by Birkhoff's M_3 and N_5 earlier. For the first, we have

$$\begin{aligned} x \sqcap \bigsqcup^{y} {\downarrow}y \cap {\downarrow}z &= (x \sqcap y) \sqcap \bigsqcup^{y} {\downarrow}y \cap {\downarrow}z \\ (\,{\downarrow}y \text{ distributive}) &= \bigsqcup^{y} \{(x \sqcap y) \sqcap d \mid d \in {\downarrow}y \cap {\downarrow}z\} \\ &\sqsubseteq \bigsqcup^{y} {\downarrow}(x \sqcap y) \cap {\downarrow}z. \end{aligned}$$

□

The next definition allows us to consider weak distributivity locally, as needed for constructing stable functions.

Definition 6 (Weakly distributive element, generator). *A compact element $c \in D^0$ is said to be weakly distributive or called a generator, if for all compatible pairs $x, y \in D$, we have*

$$x \sqcap \bigsqcup^{y} {\downarrow}y \cap {\downarrow}c = \bigsqcup^{y} {\downarrow}(x \sqcap y) \cap {\downarrow}c.$$

Weakly distributive elements will be used to generate stable functions. In the actual applications of the weak distributivity property, we use the following equivalent version:

$$x \sqcap \bigsqcup^{y} {\downarrow}(x \sqcap y) \cap {\downarrow}z = x \sqcap \bigsqcup^{y} {\downarrow}y \cap {\downarrow}z.$$

Definition 7. *Let D be an ω-algebraic meet-cpo with properties* M *and* I *and $c \in D$ a generator. Define $\eta_c : D \to D$ as*

$$\eta_c := \lambda x. \bigsqcup^{x} (\,{\downarrow}x \cap {\downarrow}c).$$

Note that even though ${\downarrow}x \cap {\downarrow}c$ need not be a directed set, the least upper bound in the principle ideal ${\downarrow}x$ always exists due to the compactness of c and properties M *and* I *of D.*

Lemma 9. *η_c is a well-defined function. For a mub-down closed set A with $c \in A$, we have $\phi_A \circ \eta_c = \eta_c$.*

Proof. For the second conclusion, let $x \in D$. Note that $\{t \in D \mid t \sqsubseteq c \;\&\; t \sqsubseteq x\} \subseteq A$, since $t \sqsubseteq c \in A$ and A is down closed. Therefore, $\eta_c(x) \in A$, since A is moreover finite mub closed. By Lemma 2, we have $\phi_A(\eta_c(x)) = \eta_c(x)$. □

Lemma 10. *For any generator $c \in D$, η_c is a compact stable function.*

Proof. The monotonicity of η_c is straightforward. For continuity, suppose Y is a directed subset of D. We have

$$\begin{aligned} \eta_c(\bigsqcup Y) &= \bigsqcup^{\bigsqcup Y} (\,{\downarrow}(\bigsqcup Y) \cap {\downarrow}c) \\ &= \bigsqcup_{y \in Y} \bigsqcup^{y} (\,{\downarrow}y \cap {\downarrow}c) \\ &= \bigsqcup_{y \in Y} \eta_c(y). \end{aligned}$$

To check the stability of η_c, let $x, y \in D$ be such that $x \uparrow y$. Since D is a meet-cpo, $x \sqcap y$ exists. We need to show $\eta_c(x) \sqcap \eta_c(y) = \eta_c(x \sqcap y)$, which follows from the weakly distributive property of c:

$$\begin{aligned}
\eta_c(x \sqcap y) &= \bigsqcup^{x \sqcap y}(\downarrow(x \sqcap y) \cap \downarrow c) \\
&= x \sqcap y \sqcap \bigsqcup^{x}(\downarrow(x \sqcap y) \cap \downarrow c) \;\sqcap\; \bigsqcup^{y}(\downarrow(x \sqcap y) \cap \downarrow c) \\
&= x \sqcap \bigsqcup^{y}(\downarrow(x \sqcap y) \cap \downarrow c) \;\sqcap\; y \sqcap \bigsqcup^{x}(\downarrow(x \sqcap y) \cap \downarrow c) \\
&= x \sqcap \bigsqcup^{y}(\downarrow y \cap \downarrow c) \;\sqcap\; y \sqcap \bigsqcup^{x}(\downarrow x \cap \downarrow c) \\
&= \eta_c(x) \sqcap \eta_c(y).
\end{aligned}$$

To show that η_c is compact, note that $\downarrow c$ is a finite set of compact elements. The mub-down closure A of $\downarrow c$, though not necessarily finite, determines a compact stable function ϕ_A. We show that η_c is stably below ϕ_A, forcing η_c to be a compact element in the stable function space $[D \to D]$. For this purpose, let $x \sqsubseteq y$ in D. Then

$$\begin{aligned}
\eta_c(y) \sqcap \phi_A(x) &\sqsubseteq \phi_A(\eta_c(y)) \sqcap \phi_A(x) \\
&= \phi_A(x \sqcap \bigsqcup^{y}(\downarrow y \cap \downarrow c)) \\
&= \phi_A(x \sqcap \bigsqcup^{y}(\downarrow(x \sqcap y) \cap \downarrow c)) \\
&= \phi_A(x \sqcap \bigsqcup^{y}(\downarrow x \cap \downarrow c)) \\
&= \phi_A(\bigsqcup^{x}(\downarrow x \cap \downarrow c)) \\
&= \phi_A(\eta_c(x)) \\
&= \eta_c(x).
\end{aligned}$$

□

Theorem 3. *Let D be an ω-algebraic meet-cpo with properties* M *and* I*, but not* MI^∞ *. Let A be the infinite* (mub, down)*-closure of a finite subset of compact elements of D. If A contains an infinite number of generators, then $[D \to D]$ fails* I.

Proof. Suppose $C := \{c_i \mid i \geq 1\}$ is an infinite subset of A consisting of generators only. Note first that, for any $c \in C$, the range of η_c, written as $r(\eta_c)$, is a finite set. Suppose without loss of generality $c_{i+1} \notin r(\eta_{c_i})$ for any $i \geq 1$. For any $1 < i < j$, we have $\eta_{c_i}(c_j) \neq c_j = \eta_{c_j}(c_j)$. So $\eta_{c_i} \neq \eta_{c_j}$. By Lemma 10, all η_c are compact elements below ϕ_A, and so $[D \to D]$ fails property I. □

If D is a weakly distributive, ω-algebraic meet-cpo with properties M and I, but not MI^∞ , then an infinite A as mentioned in Theorem 3 exists. Moreover, since D is a weakly distributive, every element of A is a generator. Therefore, we have the following corollary.

Theorem 4. *Let D be a weakly distributive, ω-algebraic meet-cpo with properties* M *and* I*, but not* MI^∞ *. Then $[D \to D]$ fails* I.

Thus we have shown that a larger class of domains than principally distributive ones must not violate MI^∞ in the category **SB** of bifinite domains.

The next lemmas will be useful to work with examples below.

Lemma 11. *Let D and E be ω-algebraic meet-cpos. Let $f, g : D \to E$ be such that g is stable, f is continuous, and $f(x) = f(y) \sqcap g(x)$ for any $x \sqsubseteq y$ in D. Then f is also stable.*

Lemma 12. *[17] Let D, E be meet-cpos and f, g be compatible stable functions in $[D \to E]$. We have*
(a) if $f(x) = g(x)$, then $f(y) = g(y)$ for any $y \in {\downarrow}x$,
(b) if $a \uparrow b$ then $f(a) \sqcap g(b) = f(b) \sqcap g(a)$.

Remark. The stable function space construction does not preserve the weakly distributive law. Here is an example. Let D and E be weakly distributive domains as given on the left of Fig. 3. The stable function space $[D \to E]$ contains the structure on the right of Fig. 3 (among other things), which is not weakly distributive.

The functions labeled on the right of Fig. 3 are defined as follows:

$$f(x) = \begin{cases} u, & \text{if } x = c \\ \bot, & \text{otherwise} \end{cases} \quad g(x) = \begin{cases} v, & \text{if } x = c \\ \bot, & \text{otherwise} \end{cases}$$

$$h(x) = \begin{cases} z, & \text{if } x = c \\ w, & \text{if } x = a \\ \bot, & \text{otherwise} \end{cases} \quad j(x) = \begin{cases} z, & \text{if } x = c \\ w, & \text{if } x = b \\ \bot, & \text{otherwise} \end{cases} \quad k(x) = \begin{cases} z', & \text{if } x = c \\ w, & \text{if } x = a \\ \bot, & \text{otherwise} \end{cases}$$

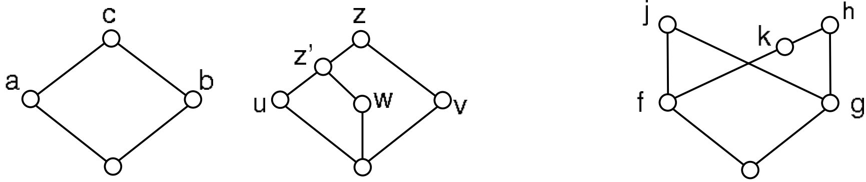


Fig. 3. Two weakly distributive domains whose stable function space is not weakly distributive

We have

1. f, g, h, j, k are compact stable functions.
2. h, j are minimal upper bounds of f, g.
3. $f \sqsubset k \sqsubset h$ and $k \not\sqsupseteq g$.

Therefore, f, g, h, j, k form a substructure as on the right of Fig. 3.

Checking all of these is a tedious task, but it should be helpful to note the following when doing so:

– Any upper bound of f, g must map c to z.
– Lemma 12 tells us that h, j are incompatible, since $h(c) = j(c)$, but $h(a) \neq j(a)$.
– To check the order relation holds as in Fig. 3, such as $f \sqsubset k$, note that $f(x) \sqcap t(y) \neq \bot$ for $t \in \{h, j, k\}$ only when $x = c$ and $y = c$.

Also note that the right side of Fig. 3 does not include all functions in the stable function space. For example, here are two more stable functions:

$$\alpha(x) = \begin{cases} z, & \text{if } x = c \\ \bot, & \text{otherwise} \end{cases} \qquad \beta(x) = \begin{cases} z', & \text{if } x = c \\ w, & \text{if } x = b \\ \bot, & \text{otherwise} \end{cases}$$

In fact, α is another minimal upper bound of f, g, and $f \sqsubset \beta \sqsubset j$.

4 Meet Generators

In this section we introduce another technique for generating stable functions.

Definition 8. *Let D be an ω-algebraic meet-cpo satisfying properties* M *and* I*. A compact element $a \in D$ is said to be a* meet-generator*, if for any $d \in D$, the meet $a \sqcap d$ exists.*

Theorem 5. *Let D be an ω-algebraic meet-cpo with properties* M *and* I*, but not* MI^∞ *. Let A be the infinite* (mub, down)*-closure of a finite subset of compact elements of D. If A contains an infinite number of meet-generators, then $[D \to D]$ fails property* I.

Proof. Let $B := \{a_i \in A \mid i \geq 1\}$ be an infinite subset of A consisting of meet-generators only. For each $i \geq 1$, define $\varphi_i : D \to D$ as

$$\varphi_i := \lambda x.\ a_i \sqcap x.$$

Then

(1) φ_i is a well-defined compact stable function;
(2) $\varphi_i \sqsubseteq \varphi_A$;
(3) $\varphi_i \neq \varphi_j$ for any distinct $i, j \geq 1$.

Item (3) is obvious. For (1), we need to show the continuity of φ_i. Let Y be a directed subset of D. Then $\varphi_i(\bigsqcup Y) = a_i \sqcap \bigsqcup Y = \bigsqcup_{y \in Y} a_i \sqcap y = \bigsqcup_{y \in Y} \varphi_i(y)$, because binary meet is continuous in both arguments in a meet-cpo. For stability of φ_i, let x, y be compatible elements in D. Then $\varphi_i(x \sqcap y) = a_i \sqcap x \sqcap y = (a_i \sqcap x) \sqcap (a_i \sqcap y) = \varphi_i(x) \sqcap \varphi_i(y)$. To show compactness of φ_i, let $\varphi_i = \bigsqcup_{j \in J} f_j$, where $\{f_j \mid j \in J\}$ is a directed family of stable functions. Here, we use equality by the meet-cpo property. We have, in particular, $a_i = \varphi_i(a_i) = \bigsqcup_{j \in J} f_j(a_i)$, and so $\varphi_i(a_i) = f_k(a_i) = a_i$ for some $k \in J$, by the compactness of a_i. By Lemma 2, item (a) of part I, $\varphi_i(x) = f_k(x)$ for all $x \sqsubseteq a_i$. Therefore $\varphi_i(x) = \varphi_i(\varphi_i(x)) = f_k(\varphi_i(x)) \sqsubseteq f_k(x)$ for any $x \in D$. Since φ_i and f_k are stably compatible, extensionally equal, we have $\varphi_i = f_k$, as needed.

For item (2), let $x \sqsubseteq y$ in D. Since A is down-closed and $a_i \in A$, $z = \varphi_i(z) = \varphi_A(z)$ for all $z \sqsubseteq a_i$. We have,

$$\begin{aligned}
\varphi_i(x) &= \varphi_i(y) \sqcap \varphi_i(\varphi_i(x)) \\
&= \varphi_i(y) \sqcap \varphi_A(\varphi_i(x)) \\
&= \varphi_i(y) \sqcap \varphi_A(a_i \sqcap x) \\
&= \varphi_i(y) \sqcap \varphi_A(a_i) \sqcap \varphi_A(x) \quad \text{(since } \varphi_A \text{ is stable)} \\
&= \varphi_i(y) \sqcap a_i \sqcap \varphi_A(x) \\
&= \varphi_i(y) \sqcap \varphi_A(x)
\end{aligned}$$

□

Theorem 6. *Let D be an ω-algebraic meet-cpo with properties* M *and* I*, but not* MI^∞. *Let A be the infinite* $(\mathsf{mub}, \mathsf{down})$*-closure of a finite set of compact elements of D. If A satisfies the condition that the principal ideals are distributive, then* $[D \to D]$ *fails* I.

Proof. For any $a \in A$, define $\eta_a : D \to D$ as

$$\eta_a := \lambda x. \bigsqcup^{x} \{t \mid t \sqsubseteq a \ \& \ t \sqsubseteq x\}$$

Then

(1) η_a is a well-defined compact stable function;
(2) $\eta_a \sqsubseteq \varphi_A$;
(3) there is an infinite subset B of A such that for any distinct $a, b \in B$, $\eta_a \neq \eta_b$.

Item (3) is obvious, since for any $a \in A$ the range of η_a is finite. For item (1), we need only to show the stability of η_a, let x, y be compatible elements in D. Then

$$\begin{aligned}
\eta_a(x) \sqcap \eta_a(y) &= \bigsqcup^{x}\{t \mid t \sqsubseteq a \ \& \ t \sqsubseteq x\} \sqcap \bigsqcup^{y}\{u \mid u \sqsubseteq a \ \& \ u \sqsubseteq y\} \\
&= \bigsqcup^{x \sqcap y}\{t \sqcap u \mid t \sqsubseteq x \ \& \ t \sqsubseteq a \ \& \ u \sqsubseteq y \ \& \ u \sqsubseteq a\} \quad \text{(by distributivity)} \\
&\sqsubseteq \bigsqcup^{x \sqcap y}\{v \mid v \sqsubseteq a \ \& \ v \sqsubseteq x \sqcap y\} \\
&= \eta_a(x \sqcap y)
\end{aligned}$$

For item (2), let $x \sqsubseteq y$ in D. We have,

$$\begin{aligned}
\eta_a(y) \sqcap \phi_A(x) &= \bigsqcup^{y}\{t \mid t \sqsubseteq a \ \& \ t \sqsubseteq y\} \sqcap \bigsqcup\{u \mid u \in A \ \& \ u \sqsubseteq x\} \\
&= \bigsqcup^{x}\{v \mid v \sqsubseteq a \ \& \ v \sqsubseteq x\} \\
&= \eta_a(x)
\end{aligned}$$

□

5 Conclusion

We introduced a new class of domains called weakly distributive domains within the category of ω-algebraic meet-cpos. The weak distributivity law allows us to construct stable functions based on the principal ideal generated by a single compact element – a generator. For weakly distributive domains to be included in any full stable cartesian closed category composed of ω-algebraic meet-cpos, they must satisfy axiom MI^∞ (Thm. 4). ω-algebraic meet-cpos satisfying M and I, which do not satisfy MI^∞ and are not weakly distributive, are abundant. The domain in the middle of Fig. 1 is one such example. Interestingly, all non-weakly distributive domains we have seen so far contain examples in Fig. 2 as substructures. Non-weak distributivity itself does not violate MI^∞ ; it is many of the non-weakly distributive configurations put together that creates a configuration violating MI^∞ . Thus, looking deeper into non-weakly distributive substructures in the context of configurations violating MI^∞ might lead to additional insight into the open problem of Amadio-Curien. The available ways to deal with a rich variety of configurations violating MI^∞ so far have helped keeping our bet on an affirmative solution alive.

References

1. Amadio, R.-M.: Bifinite domains: stable case. LNCS, vol. 530, pp. 16–33. Springer, Heidelberg (1991)
2. Amadio, R.-M., Curien, P.-L.: Domains and Lambda-Calculi. Cambridge Tracts in Theeoretical Computer Science, vol. 46. Cambridge University Press, Cambridge (1998)
3. Berry, G.: Modèles complètement adéquats et stables des lambda-calculs typés. Thèse de Doctorat d'Etat, Université Paris VII (1979)
4. Coquand, T., Gunter, C.A., Winskel, G.: DI-domains as a model of polymorphism. LNCS, vol. 298, pp. 344–363. Springer, Heidelberg (1987)
5. Davey, B.-A., Priestley, H.-A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (2002)
6. Droste, M.: On stable domains. Theoretical Computer Science 111, 89–101 (1993)
7. Droste, M.: Cartesian closed categories of stable domains for polymorphism. Preprint, Universität GHS Essen
8. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
9. Milner, R.: Fully abstract models of typed λ-calculi. Theoretical Computer Science 4, 1–22 (1977)
10. Plotkin, G.: A powerdomain construction. SIAM J. Computing 5, 452–487 (1976)
11. Reddy, U.: Global state considered unnecessary: An introduction to object-based semantics. J. Lisp and Symbolic Computation. 9, 7–76 (1996)
12. Smyth, M.B.: The largest cartesian closed category of domains. Theoretical Computer Science 27, 109–120 (1983)
13. Winskel, G.: An introduction to event structures. Lecture Notes in Computer Science 354, 364–399 (1988)
14. Zhang, G.-Q.: dI-domains as prime information systems. Information and Computation 100, 151–177 (1992)
15. Zhang, G.-Q.: The largest cartesian closed category of stable domains. Theoretical Computer Science. 166, 203–219 (1996)
16. Zhang, G.-Q.: Logic of Domains. Birkhauser (1991)
17. Zhang, G.-Q., Jiang, Y.: On a problem of Amadio and Curien: The finite antichain condition. Information and Computation 202, 87–103 (2005)

Initial Algebra Semantics Is Enough!

Patricia Johann[1] and Neil Ghani[2]

[1] Rutgers University, Camden, NJ, USA
pjohann@crab.rutgers.edu
[2] University of Nottingham, Nottingham, UK
nxg@cs.nott.ac.uk

Abstract. Initial algebra semantics is a cornerstone of the theory of modern functional programming languages. For each inductive data type, it provides a `fold` combinator encapsulating structured recursion over data of that type, a Church encoding, a `build` combinator which constructs data of that type, and a `fold`/`build` rule which optimises modular programs by eliminating intermediate data of that type. It has long been thought that initial algebra semantics is not expressive enough to provide a similar foundation for programming with nested types. Specifically, the `fold`s have been considered too weak to capture commonly occurring patterns of recursion, and no Church encodings, `build` combinators, or `fold`/`build` rules have been given for nested types. This paper overturns this conventional wisdom by solving all of these problems.

1 Introduction

Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. It provides support for `fold` combinators encapsulating structured recursion over data structures, thereby making it possible to write, reason about, and transform programs in principled ways. Recently, (13) extended the usual initial algebra semantics for inductive types to support not only standard `fold` combinators, but Church encodings and `build` combinators for them as well. In addition to being theoretically useful in ensuring that `build` is seen as a fundamental part of the basic infrastructure for programming with inductive types, this development has practical merit: the `fold` and `build` combinators can be used to define `fold`/`build` rules which optimise modular programs by eliminating intermediate inductive data structures. When applied to lists, this optimisation is known as *short cut fusion*.

Nested data types have become increasingly popular in recent years (1; 3; 5; 6; 7; 14; 15; 16; 17; 20). They have been used to implement a number of advanced data types in languages, such as Haskell, which support higher-kinded types. Among these data types are those with constraints, such as perfect trees (16); types with variable binding, such as untyped λ-terms (2; 5; 8); cyclic data structures (11); and certain dependent types (21). The expressiveness of nested types lies in their generalisation of the traditional treatment of types as free-standing individual entities to entire families of types. To illustrate this point, consider

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 207–222, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the type of lists of elements of type `a`. This type can be realised in Haskell via the declaration `data List a = Nil | Cons a (List a)`. As this declaration makes clear, the type `List a` can be defined independently of any type `List b` for `b` distinct from `a`. Moreover, since each type `List a` is, in isolation, an inductive type, the type constructor `List` is seen to define a *family of inductive types*. Compare the declaration for `List a` with the declaration

```
data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
```

defining the type `Lam a` of untyped λ-terms over variables of type `a` up to α-equivalence. By contrast with `List a`, the type `Lam a` cannot be defined in terms of only those elements of `Lam a` that have already been constructed. Indeed, elements of the type `Lam (Maybe a)` are needed to build elements of `Lam a` so that, in effect, the entire family of types determined by `Lam` has to be constructed simultaneously. Thus, rather than defining a family of inductive types as `List` does, `Lam` defines an *inductive family of types*.

Given the increased expressivity of nested types over inductive types, and the ensuing growth in their use, it is natural to ask whether initial algebra semantics can give a principled foundation for structured programming with nested types. Until now this has not been considered possible. In particular, `fold` combinators derived from initial algebra semantics for nested types have not been considered expressive enough to capture certain commonly occurring patterns of structured recursion over data of those types. This has led to a theory of *generalised* `fold`*s* for nested types (1; 3; 6). Moreover, no Church encodings, `build` combinators, or `fold`/`build` fusion rules have been proposed or defined for nested types.

This paper overturns this conventional wisdom and provides the ideal result, namely that *initial algebra semantics is enough to provide a principled foundation for programming with nested types.* Our major contributions are as follows:

- We define a generalised `fold` combinator `gfold` for *every* nested type and show it to be uniformly interdefinable with the corresponding `hfold` combinator derived from initial algebra semantics. Our `gfold` combinators coincide with the generalised `fold`s in the literature whenever the latter are defined. The `hfold` combinators provided by initial algebra semantics thus capture *exactly the same kinds of recursion* as the generalised `fold`s in the literature.
- We give the first-ever Church encodings for nested types. In addition to being interesting in their own right, these encodings are the key to defining the first-ever `build` combinators for nested types. Coupling each `hbuild` combinator with its corresponding `hfold` combinator in turn gives the first-ever `hfold`/`hbuild` rules for nested types, and thus extends short cut fusion to these types. A similar story holds for the `gfold` and `gbuild` combinators.

We make several other important contributions. First, we execute the above program in a generic style by providing *a single* generic `hfold` combinator, *a single* generic `hbuild` operator, and *a single* generic `hfold`/`hbuild` rule, each of which can be specialised to any particular nested type of interest — and similarly

for the generalised combinators. Secondly, while the theory of nested types has previously been developed only for limited classes of nested types arising from certain syntactically defined classes of rank-2 functors, our development handles *all* rank-2 functors. Finally, we give a complete implementation of our ideas in Haskell, available at `http://www.cs.nott.ac.uk/~nxg`. This demonstrates the practical applicability of our ideas, makes them more accessible, and provides a partial guarantee of their correctness via the Haskell type-checker. This paper can therefore be read both as abstract mathematics, and as providing the basis for experiments and practical applications. Past work on nested types did not come with full implementations, in part because essential features such as explicit and nested `forall`-types have only recently been added to Haskell.

Our result that initial algebra semantics is expressive enough to provide a foundation for programming with nested types allows us to capitalise on the increased expressiveness of nested types over inductive types without requiring the development of any fundamentally new theory. Moreover, this foundation is simple, clean, and accessible to anyone with an understanding of the basics of initial algebra semantics. This is important, since it guarantees that our results are immediately usable by functional programmers. Further, by closing the gap between initial algebra semantics and Haskell's data types, this paper clearly contributes to the foundations of functional programming. This paper also serves as a compelling demonstration of the practical applicability of left and right Kan extensions — which are the main technical tools used to define our `gfold`s and prove them interdefinable with the `hfold`s — and thus has the potential to render them mainstays of functional programming.

The paper is structured as follows. Section 2 recalls the initial algebra semantics of inductive types. Section 3 recalls the derivation of `fold` combinators from initial algebra semantics for nested types, and derives the first Church encodings, `build` combinators, and `fold`/`build` rules for them. Section 4 defines our `gfold` combinators for nested types and shows that they are interdefinable with their corresponding `hfold` combinators. It also derives our `gbuild` combinators and `gfold`/`gbuild` rules for nested types. Section 5 mentions the coalgebraic duals of our combinators and draws some conclusions.

2 Initial Algebra Semantics for Inductive Types

Inductive data types are fixed points of functors. Functors can be implemented in Haskell as type constructors supporting `fmap` functions as follows:

```
class Functor f where fmap :: (a -> b) -> f a -> f b
```

The function `fmap` is expected to satisfy the two semantic functor laws stating that `fmap` preserves identities and composition. As is well known (12; 13; 23), every inductive type has an associated `fold` and `build` combinator which can be implemented generically in Haskell as

```
newtype M f = Inn {unInn :: f (M f)}

ffold :: Functor f => (f a -> a) -> M f -> a
ffold h (Inn k) = h (fmap (ffold h) k)

fbuild :: Functor f => (forall b. (f b -> b) -> b) -> M f
fbuild g = g Inn
```

These `fbuild` and `ffold` combinators can be used to construct and eliminate inductive data structures of type `M f` from computations. Indeed, if `f` is any functor, `h` is any function of any type `f a -> a`, and `g` is any function of closed type `forall b. (f b -> b) -> b`, we have the `fold`/`build` rule:

```
ffold h (fbuild g) = g h                                    (1)
```

When specialised to lists, this gives the familiar combinators

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n [] = n
foldr c n (x:xs) = c x (foldr c n xs)

build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Intuitively, `foldr c n xs` produces a value by replacing all occurrences of `(:)` in `xs` by `c` and the occurrence of `[]` in `xs` by `n`. Thus, `sum xs = foldr (+) 0 xs` sums the (numeric) elements of the list `xs`. On the other hand, `build` takes as input a type-independent template for constructing "abstract" lists and produces a corresponding "concrete" list. Thus, `build (\c n -> c 4 (c 7 n))` produces the list `[4,7]`. List transformers can be written in terms of both `foldr` and `build`. For example, the standard `map` function for lists can be implemented as

```
map :: (a -> b) -> [a] -> [b]
map f xs = build (\c n -> foldr (c . f) n xs)
```

The function `build` is not just of theoretical interest as the producer counterpart to the list consumer `foldr`. In fact, `build` is an important ingredient in *short cut fusion* (9; 10), a widely-used program optimisation which capitalises on the uniform production and consumption of lists to improve the performance of list-manipulating programs. For example, if `sqr x = x * x`, then the specialisation of (1) to lists — i.e., the rule `fold c n (build g) = g c n` — can transform the modular function `sum (map sqr xs) :: [Int] -> Int` which produces an intermediate list into an optimised form which produces no such lists:

```
sum (map sqr xs) = foldr (+) 0
                             (build (\c n -> foldr (c . sqr) n xs))
                 = (\c n -> foldr (c . sqr) n xs) (+) 0
                 = foldr ((+) . sqr) 0 xs
```

If we are to generalise the treatment of inductive types given above to more advanced data types, we must ask ourselves why `fold` and `build` combinators exist for inductive types and why the associated `fold`/`build` rules are correct. One elegant answer is provided by *initial algebra semantics*. Within the paradigm of initial algebra semantics, every data type is the carrier of the initial algebra μF of a functor $F : \mathcal{C} \to \mathcal{C}$. If $\mathcal{C}$ has both an initial object and ω-colimits, and F preserves ω-colimits, then F does indeed have an initial algebra. Lambek's lemma ensures that the structure map in of an initial algebra is an isomorphism, and thus that the carrier of the initial algebra of a functor is a fixed point of that functor. The interpretation of a given data type as an initial algebra of a functor F ensures that there is a unique F-algebra homomorphism from this initial F-algebra to any other F-algebra. If (A, h) is an F-algebra, then $\texttt{fold}\, h : \mu F \to A$ is the map underlying this homomorphism and makes the following diagram commute:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(\texttt{fold}\, h)} & FA \\ {\scriptstyle in}\downarrow & & \downarrow{\scriptstyle h} \\ \mu F & \xrightarrow{\texttt{fold}\, h} & A \end{array}$$

From this diagram, we see that the type of `fold` is $(FA \to A) \to \mu F \to A$ and that $\texttt{fold}\, h$ satisfies $\texttt{fold}\, h\ (in\ t) = h\ (F\ (\texttt{fold}\, h)\ t)$. This justifies the definition of the `ffold` combinator given above. Also, the uniqueness of the mediating map ensures that, for every algebra h, the map $\texttt{fold}\, h$ is defined uniquely. This provides the basis for the correctness of `fold` fusion for inductive types, which states that if h and h' are F-algebras and ψ is an F-algebra homomorphism from h to h', then $\psi \,.\, \texttt{fold}\, h = \texttt{fold}\, h'$. But note that `fold` fusion (3; 5; 6; 7; 20), is completely different from, and inherently simpler than, the `fold`/`build` fusion which is central in this paper, and which we discuss next.

Although `fold` combinators for inductive types can be derived entirely from, and understood entirely in terms of, initial algebra semantics, regrettably the standard initial algebra semantics does not provide a similar principled derivation of the `build` combinators or the correctness of the `fold`/`build` rules. This situation was rectified in (13), which considered the initial F-algebra for a functor F to be not only the initial object of the category of F-algebras, but also the limit of the forgetful functor from the category of F-algebras to the underlying category $\mathcal{C}$ as well. When F has an initial algebra, no extra structure is required of either F or $\mathcal{C}$ for this limit to exist. This characterisation of initial algebras as both limits *and* colimits is what we call the *extended initial algebra semantics*. As shown in (13), an initial F-algebra has a different universal property as a limit from the one it inherits as a colimit. This alternate universal property ensures:

- The projection from the limit (the initial F-algebra) to the carrier of each algebra defines the `fold` combinator with type $(Fx \to x) \to \mu F \to x$.
- The mediating morphism maps a cone with arbitrary vertex c to a map from c to μF. Since a cone with vertex c has type $\forall x.(Fx \to x) \to c \to x$,

the mediating morphism defines the `build` combinator, which will thus have type $(\forall x. (Fx \to x) \to c \to x) \to c \to \mu F$.

- The correctness of the `fold`/`build` fusion rule `fold` h . `build` $g \;=\; g\,h$ then follows from the fact that `fold` after `build` is a projection after a mediating morphism, and thus is equal to the cone applied to a specific algebra.

The extended initial algebra semantics thus shows that, given a parametric interpretation of the quantifier `forall`, there is an isomorphism between the type `c -> M f` and the "generalised Church encoding" `forall x. (f x -> x) -> c -> x`. The term "generalised" reflects the presence of the parameter `c`, which is absent in other Church encodings ([23]), but is essential to the derivation of `build` combinators for nested types. Choosing `c` to be the unit type gives the usual isomorphism between an inductive type and its usual Church encoding.

3 Initial Algebra Semantics for Nested Types

Although many types of interest can be expressed as inductive types, these types are not expressive enough to capture all data structures of interest. Such structures can, however, often be expressed in terms of *nested types*.

Example 1. *The type of* perfect trees *over type* `a` *is given by*

```
data PTree a = PLeaf a | PNode (PTree (a,a))
```

The recursive constructor `PNode` stores not pairs of trees, but rather trees with data of pair types. Thus, `PTree a` is a nested type for each `a`. Perfect trees are easily seen to be in one-to-one correspondence with lists whose length is a power of two, and hence illustrate how nested types can be used to capture structural constraints on data types. Another example of nested types is given by

Example 2. *The type of (α-equivalence classes of)* untyped λ-terms *over variables of type* `a` *is given by*

```
data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
```

Elements of type `Lam a` include `Abs (Var Nothing)` and `Abs (Var (Just x))`, which represent $\lambda x.x$ and $\lambda y.x$, respectively. We observed above that each nested type constructor defines an inductive family of types. It is thus natural to model nested types as least fixed points of functors on the category of endofunctors on $\mathcal{C}$, written $[\mathcal{C}, \mathcal{C}]$. In this category, objects are functors and morphisms are natural transformations. We call such functors *higher-order functors*, and denote the fixed point of a higher-order functor `f` by `Mu f`. Our implementation cannot use the constructor `M` introduced above because Haskell lacks polymorphic kinding.

```
class HFunctor f where
  ffmap :: Functor g => (a -> b) -> f g a -> f g b
  hfmap :: Nat g h -> Nat (f g) (f h)

newtype Mu f a = In {unIn :: f (Mu f) a}
```

A higher-order functor thus maps functors to functors via the `ffmap` operation and natural transformations to natural transformations via the `hfmap` operation. While not explicit in the class definition above, the programmer is expected to verify that if `g` is a functor, then `f g` satisfies the functor laws. The type of natural transformations can be given in Haskell by `type Nat g h = forall a. g a -> h a`, since a parametric interpretation of the `forall` quantifier ensures that the naturality squares commute. Putting this all together, we have

Example 3. *The nested types of perfect trees and untyped λ-terms from Examples 1 and 2 arise as fixed points of the higher-order functors*

```
data HPTree f a = HPLeaf a | HPNode (f (a,a))


data HLam f a = HVar a | HApp (f a) (f a) | HAbs (f (Maybe a))
```

respectively. Indeed, the types `PTree a` *and* `Lam a` *are isomorphic to the types* `Mu HPTree a` *and* `Mu HLam a`*.*

Pleasingly, `fold` combinators for nested types can be derived by simply instantiating the ideas from Section 2 in a category of endofunctors. Of course, now the structure map of an algebra is a natural transformation, and the result of a `fold` is a natural transformation from a nested type to the carrier of the algebra. Using the synonym `type Alg f g = Nat (f g) g` for such algebras, we have

```
hfold :: HFunctor f => Alg f g -> Nat (Mu f) g
hfold m (In u) = m (hfmap (hfold m) u)
```

Example 4. *The* `fold` *combinator for perfect trees is*[1]

```
foldPTree :: (forall a. a -> f a) ->
                        (forall a. f (a,a) -> f a) -> PTree a -> f a
foldPTree f g (PLeaf x)  = f x
foldPTree f g (PNode xs) = g (foldPTree f g xs)
```

The uniqueness of `hfold`, guaranteed by its derivation from initial algebra semantics, provides the basis for the correctness of `fold` fusion for nested types ([7]). As mentioned above, `fold` fusion is not the same as `fold`/`build` fusion. In particular, the latter has not previously been considered for nested types.

Recall from Section 2 that Church encodings and `build` combinators for inductive types can be derived from the characterisation of the initial F-algebra as the limit of the forgetful functor from the category of F-algebras to the underlying category, and that this gives an isomorphism between types of the form `c -> M f` and generalised Church encodings `forall x. (f x -> x) -> c -> x`. Since this isomorphism holds for *all* functors, including higher-order ones, we should be able to instantiate it for higher-order functors to derive Church encodings and `build` combinators for nested types. And indeed we can. This gives the following Haskell code:

[1] Here we have used standard type isomorphisms to "unbundle" the input type `Alg HPTree f` for `foldPTree`. Such unbundling will be done without comment henceforth.

```
hbuild :: (HFunctor f, Functor c) =>
              (forall x. Alg f x -> Nat c x) -> Nat c (Mu f)
hbuild g = g In
```

It is worth noticing that each `hbuild` combinator follows the definitional format of the `build` combinators for inductive types: it applies its argument to the structure map `In` of the initial algebra of the higher-order functor `f` with which it is associated. For our running example of perfect trees, we have the following:

Example 5. *The* `hbuild` *combinator for perfect trees is given concretely by*

```
buildPTree :: (forall x. (forall a. a -> x a) ->
                         (forall a. x (a,a) -> x a) ->
                         (forall a. c a -> x a)) -> Nat c PTree
buildPTree g = g PLeaf PNode
```

The extended initial algebra semantics ensures that `hbuild` and (an argument-permuted version of) `hfold` are mutually inverse, and thus that the following `fold`/`build` rule holds for nested types:

Theorem 1. *If* `f` *is a higher-order functor,* `c` *and* `a` *are functors,* `h` *is the structure map of an algebra* `Alg f a`*, and* `g` *is any function of closed type* `forall x. Alg f x -> Nat c x`*, then*

```
hfold h . hbuild g = g h                                    (2)
```

Note that the *application* of `ffold h` to `fbuild g` in (1) has been generalised by the *composition* of `hfold h` and `hbuild g` in (2). This is because `c` remains uninstantiated in the nested setting, whereas it is specialised to the unit type in the inductive one. For our running example, we have the following:

Example 6. *The instantiation of (2) for perfect trees is*

```
foldPTree l n . buildPTree g = g l n
```

From Section 2, to ensure that a higher-order functor F on $\mathcal{C}$ has an initial algebra we need that the category $[\mathcal{C}, \mathcal{C}]$ has an initial object and ω-colimits, and that F preserves ω-colimits. But only the latter actually needs to be verified since the initial object and ω-colimits in $[\mathcal{C}, \mathcal{C}]$ are inherited from those in $\mathcal{C}$.

4 Generalised Folds, Builds, and Short Cut Fusion

In this section we recall the generalised `fold` combinators — here called `gfolds` — from the literature (1; 3; 6). We also introduce a corresponding generalised `build` combinator `gbuild` and a `gfold`/`gbuild` fusion rule for each nested type. We show that, just as the `gfold` combinators are instances of the `hfold` combinators, so the `gbuild` combinators are instances of the `hbuild` combinators, and the `gfold`/`gbuild` rules can be derived from the `hfold`/`hbuild` rules. These results are important because, until now, it has been unclear which general principles should underpin the definition of `gfold` combinators, and because `gbuild`

combinators and `gfold`/`gbuild` rules have not existed. Our rendering of the generalised combinators and fusion rules as instances of their counterparts from Section 3 shows that *the same principles of initial algebra semantics that govern the behaviour of* `hfold`*,* `hbuild`*, and* `hfold`*/*`hbuild` *fusion also govern the behaviour of* `gfold`*,* `gbuild`*, and* `gfold`*/*`gbuild` *fusion.* In particular, whereas `gfold`s have previously been defined only for certain syntactically defined classes of higher-order functors, initial algebra semantics allows us to define `gfold`s for *all* higher-order functors, and to do so in such a way that our `gfold`s coincide with the `gfold`s in the literature whenever the latter are defined. Our reduction of `gfold`s to `hfold`s can thus be seen as an extension of the results of (1).

Generalised folds arise when we want to consume a structure of type `Mu f a` for a *single* type `a`. The canonical example is the function `psum :: PTree Int -> Int` which sums the (integer) data in a perfect tree (16). It seems `psum` cannot be expressed in terms of `hfold` since `hfold` consumes expressions of polymorphic type, and `PTree Int` is not such a type. Naive attempts to define `psum` will fail because the recursive call to `psum` must consume a structure of type `PTree (Int,Int)` rather than `PTree Int`. These considerations have led to the development of *generalised* `fold` *combinators* for nested types (1; 3; 6). Like the `hfold` combinator for a nested type, the generalised `fold` takes as input an algebra of type `Alg f g` for a higher-order functor `f` whose fixed point the nested type constructor is. But while the `hfold` returns a result of type `Nat (Mu f) g`, the corresponding generalised `fold` returns a result of the more general type `` Nat (Mu f `Comp` g) h ``, where `Comp` represents the composition of functors:

```
newtype Comp g h a = Comp {icomp :: g (h a)}

instance (Functor g, Functor h) => Functor (g `Comp` h) where
    fmap k (Comp t) = Comp (fmap (fmap k) t)
```

However, `` Mu f `Comp` g `` is not necessarily an inductive type constructor, so there is no clear theory upon which the definition of `gfold`s can be based. Alternatively, `psum` can be defined using an accumulating parameter as follows:

```
psum :: PTree Int -> Int
psum xs = psumAux xs id

psumAux :: PTree a -> (a -> Int) -> Int
psumAux (PLeaf x) e  = e x
psumAux (PNode xs) e = psumAux xs (\(x,y) -> e x + e y)
```

Here, `psumAux` generalises `psum` to take as input an environment of type `a -> Int` which is updated to reflect the extra structure in the recursive calls. Thus, `psumAux` is a polymorphic function which returns a continuation of type `(a -> Int) -> Int`. To construct our generalised `fold`s, we will actually use a generalised form of continuation whose environment stores values parameterised by a functor `g`, and whose results are parameterised by a functor `h`. We have

```
newtype Ran g h a = Ran {iran :: forall b. (a -> g b) -> h b}
```

Categorically, these continuations are just right Kan extensions, which are defined as follows. Given a functor $G : \mathcal{A} \to \mathcal{B}$ and a category $\mathcal{C}$, precomposition with G defines a functor $_ \circ G : [\mathcal{B}, \mathcal{C}] \to [\mathcal{A}, \mathcal{C}]$. A *right Kan extension* is a right adjoint to $_ \circ G$. More concretely, given a functor $H : \mathcal{A} \to \mathcal{C}$, the right Kan extension of H along G, written $\mathsf{Ran}_G H$, is defined via the natural isomorphism $[\mathcal{A}, \mathcal{C}](F \circ G, H) \cong [\mathcal{B}, \mathcal{C}](F, \mathsf{Ran}_G H)$. The classic end formula (see [19]) for details) underlies the implementation of a right Kan extension in Haskell as a universally quantified type, with relational parametricity guaranteeing that we do get a proper end as opposed to simply a universally quantified formula.

We stress that no categorical knowledge of Kan extensions is needed to understand the remainder of this paper; indeed, the few concepts we use which involve them will be implemented in Haskell. However, we retain the terminology to highlight the mathematical underpinnings of generalised continuations, and to bring to a wider audience the computational usefulness of Kan extensions.

The bijection characterising right Kan extensions can be implemented as

```
toRan :: Functor k => Nat (k `Comp` g) h -> Nat k (Ran g h)
toRan s t = Ran (\env -> s (Comp (fmap env t)))

fromRan :: Nat k (Ran g h) -> Nat (k `Comp` g) h
fromRan s (Comp t) = iran (s t) id
```

The polymorphic function `psumAux` is a natural transformation from `PTree` to `Ran (Con Int) (Con Int)`, where `Con k` is the constantly `k`-valued functor defined by `newtype Con k a = Con {icon :: k}`.[2] This suggests that an alternative to inventing a generalised `fold` combinator to define `psumAux` is to first endow the functor `Ran (Con Int) (Con Int)` with an appropriate algebra structure and then define `psumAux` as the application of `hfold` to that algebra.

Giving a direct definition of an algebra structure for `Ran g h` turns out to be rather cumbersome. Instead, we circumvent this difficulty by drawing on the intuition inherent in the continuations metaphor for `Ran g h`. If `y` is a functor, then an *interpreter* for `y` with a polymorphic environment which stores values parameterised by `g` and whose results are parameterised by `h` is a function of type `type Interp y g h = Nat y (Ran g h)`. Such an interpreter takes as input a value of type `y a` and an environment of type `a -> g b`, and returns a result of type `h b`. Associated with the type synonym `Interp` is the function

```
runInterp :: Interp y g h -> y a -> (a -> g b) -> h b
runInterp k y e = iran (k y) e
```

An *interpreter transformer* can now be defined as a function which takes as input a higher-order functor `f` and functors `g` and `h`, and returns a map which takes as input an interpreter for any functor `y` and produces an interpreter for the functor `f y`. We can define a type of interpreter transformers in Haskell by

[2] The use of constructors such as `Con` and `Comp` is required by Haskell. Although the price of lengthier code and constructor pollution is unfortunate, we believe it is outweighed by the benefits of having an implementation.

```
type InterpT f g h = forall y. Functor y =>
                             Interp y g h -> Interp (f y) g h
```

We argue informally that interpreter transformers are relevant to the study of nested types. Recall that the `hfold` combinator for a higher-order functor `f` must compute a value for each value of type `Mu f a`, and the functor `Mu f` can be considered the colimit of the sequence of approximations `f^n 0`, where `0` is the functor whose value is constantly the empty type. We can define an interpreter for `0` since there is nothing to interpret. An interpreter transformer allows us to produce an interpreter for `f 0`, then for `f^2 0`, and so on, and thus contains all the information necessary to produce an interpreter for `Mu f`. This intuition can be formalised by showing that interpreter transformers are algebras. We have:

```
toAlg :: InterpT f g h -> Alg f (Ran g h)
toAlg interpT = interpT idNat

fromAlg :: HFunctor f => Alg f (Ran g h) -> InterpT f g h
fromAlg h interp = h . hfmap interp
```

where `idNat :: Nat f f` is the identity natural transformation defined by `idNat = id`. Parametricity and naturality guarantee that `toAlg` and `fromAlg` are mutually inverse. Thus, interpreter transformers are simply algebras over right Kan extensions presented in a more computationally intuitive manner. We now define

```
gfold :: HFunctor f => InterpT f g h -> Nat (Mu f) (Ran g h)
gfold interpT = hfold (toAlg interpT)
```

The function

```
rungfold :: HFunctor f =>
                 InterpT f g h -> Mu f a -> (a -> g b) -> h b
rungfold interpT   = iran . gfold interpT
```

removes the `Ran` constructor from the output of `gfold` to expose the underlying function. An alternative definition of `gfold` would have ``Nat (Mu f `Comp` g) h`` as its return type and use `toRan` to compute functions whose natural return types are of the form `Nat (Mu f) (Ran g h)`. But, contrary to expectation, `gfold` combinators defined in this way are not expressive enough to represent all uniform consumptions with return types of this form. For example, the function `fmap :: (a -> b) -> Mu f a -> Mu f b` in the `Functor` instance declaration for `Mu f` given at the end of this section is written using the `gfold` combinator defined above. However, defining `fmap` as the composition of `toRan` and a call to a `gfold` combinator with return type of the form ``Nat (Mu f `Comp` g) h`` is not possible. This is because the use of `toRan` assumes the functoriality of `Mu f` — which is precisely what defining `fmap` establishes.

We have thus defined the first-ever generalised `fold` combinators for *all* higher-order functors and done so *uniformly* in terms of their corresponding `hfold`

combinators. Our definition is different from, but, as noted above, provably equal to, the definition given in [1] for the class of functors treated there. It also differs from all definitions of generalised `fold`s appearing in the literature, since none of these establishes that the `gfold` combinator for any nested type can be defined in terms of its corresponding `hfold` combinator.

We come full circle by using the specialisation of the `gfold` combinator to the higher-order functor `HPTree` to define a function `sumPTree` which is equivalent to `psum`. We first define an auxiliary function `sumAuxPTree`, in terms of which `sumPTree` itself will be defined. To define `sumAuxPTree` we must define an interpreter transformer; we do this by giving its two unbundled components:

```
type PLeafT g h = forall y. forall a.
                       Nat y (Ran g h) -> a -> Ran g h a
type PNodeT g h = forall y. forall a.
                       Nat y (Ran g h) -> y (a,a) -> Ran g h a

gfoldPTree :: PLeafT g h -> PNodeT g h -> PTree a -> Ran g h a
gfoldPTree l n = foldPTree (l idNat) (n idNat)

psumL :: PLeafT (Con Int) (Con Int)
psumL pinterp x = Ran (\e -> e x)

psumN :: PNodeT (Con Int) (Con Int)
psumN pinterp x = Ran (\e -> runInterp pinterp x (update e))

update e (x,y) = e x `cplus` e y
                 where cplus (Con a) (Con b) = Con (a+b)

sumAuxPTree :: PTree a -> Ran (Con Int) (Cont Int) a
sumAuxPTree = gfoldPTree psumL psumN

sumPTree :: PTree Int -> Int
sumPTree = icon . fromRan sumAuxPTree . Comp . fmap Con
```

Thus, `sumPTree` is essentially `fromRan sumAuxPTree` — ignoring the constructor pollution introduced by Haskell, that is.

Our next example uses generalised `fold`s to show that untyped λ-terms are an instance of the monad class. Here, `gfold` is used to define the `bind` operation `>>=`, which captures substitution.

```
subAlg :: InterpT HLam (Mu HLam) (Mu HLam)
subAlg k (HVar x)   = Ran (\e -> e x)
subAlg k (HApp t u) = Ran (\e -> In (HApp (runInterp k t e)
                                          (runInterp k u e)))
subAlg k (HAbs t) = Ran (\e -> In (HAbs (runInterp k t (lift e))))
lift e (Just x) = fmap Just (e x)
lift e Nothing  = In (HVar Nothing)
```

```
instance Monad (Mu HLam) where
   return  = In . HVar
   t >>= f = rungfold subAlg t f
```

Finally, note that we can also put the generic form of generalised `fold`s to good use. We illustrate this by using `gfold` to establish that all nested types are functors as follows. Let `Id a = Id {unid :: a}`. Then

```
mapAlg :: HFunctor f => InterpT f Id (Mu f)
mapAlg k t = let k1 t = runInterp k t Id
             in Ran (\e -> In (hfmap k1 (ffmap (unid . e) t)))

instance HFunctor f => Functor (Mu f) where
      fmap k t = rungfold mapAlg t (Id . k)
```

It is natural to ask whether or not there exist generalised `build` combinators corresponding to our generalised `fold`s. Since the `gfold` combinators return results of type `Nat (Mu f) (Ran g h)`, their corresponding generalised `build`s should produce results with types of the form `Nat c (Mu f)`. But the fact that generalised `fold`s are representable as certain `hfold`s suggests that we should be able to define such generalised `build`s in terms of our `hbuild` combinators, rather then defining entirely new `build` combinators. Taking `c` to be the left Kan extension `Lan g h` dual to `Ran g h` (see [19] for details) and implemented in Haskell as

```
data Lan g h a = forall b. Lan (g b -> a, h b)
```

we have

```
gbuild :: HFunctor f => (forall x. Alg f x -> Nat (Lan g h) x)
                        -> Nat (Lan g h) (Mu f)
gbuild = hbuild
```

The Haskell functions

```
toLan :: Functor f => Nat h (f `Comp` g) -> Nat (Lan g h) f
toLan s (Lan (val, v)) =  fmap val (icomp (s v))

fromLan :: Nat (Lan g h) f -> Nat h (f `Comp` g)
fromLan s t = Comp (s (Lan (id, t)))
```

code the bijection between types of the form `Nat h (f `Comp` g)` and `Nat (Lan g h) f` characterising left Kan extensions. The simplicity of the definition of `gbuild` highlights the importance of choosing an appropriate formalism, here Kan extensions, to reflect inherent structure. While it appears that defining the `gbuild` combinators requires no effort at all once we have the `hbuild` combinators, the key insight lies in introducing the abstraction `Lan` and using the bijection between `Nat h (f `Comp` g)` and `Nat (Lan g h) f`.

As an immediate consequence of Theorem 1 we have

Theorem 2. *If* `f` *is a higher-order functor,* `g`*,* `h` *and* `h'` *are functors,* `k` *is an algebra presented as an interpreter transformer of type* `InterpT f g h'`*, and* `l` *is a function of closed type* `forall x. Alg f x -> Nat (Lan g h) x`*, then*

```
gfold k . (gbuild l) = l (toAlg k)                (3)
```

Examples of generalised short cut fusion in action will be given in a journal version of this paper.

5 Conclusion and Future Work

We have extended the standard initial algebra semantics for nested types to augment the standard `hfold` combinators for such types with the first-ever Church encodings, `hbuild` combinators, and `hfold`/`hbuild` rules for them. In fact, we have capitalised on the uniformity of the isomorphism between nested types and their Church encodings to derive a single generic standard `hfold` combinator, a single generic standard `hbuild` operator, and a single generic standard `hfold`/`hbuild` rule, each of which can be specialised to any particular nested type of interest. We have also defined a generic generalised `fold` combinator, a generic generalised `build` combinator, and a generic generalised `fold`/`build` rule, each of which is uniformly interdefinable with the corresponding standard construct for nested types. The uniformity of both the standard and generalised constructs derives from a technical approach based on initial algebras of functors. Our generalised `fold` combinators coincide with the generalised `fold`s in the literature when the latter are defined. Moreover, our approach is the first to apply to *all* nested types, and thus provides a principled and elegant foundation for programming with them. We also give the first (Haskell) implementation of these combinators, and illustrate their use in several examples. We believe this paper contributes to a settled foundation for programming with nested types.

In fact, our approach also straightforwardly dualises to the coinductive setting. Shortage of space prevents us from giving the corresponding constructs and results in detail in this paper, so we simply present their implementation:

```
type CoAlg f g  = Nat g (f g)

hunfold :: HFunctor f => CoAlg f g -> Nat g (Mu f)
hunfold k x = In (hfmap (hunfold k) (k x))

hdestroy :: (HFunctor f, Functor c) =>
                  (forall g. CoAlg f g -> Nat g c) -> Nat (Mu f) c
hdestroy g = g out
out :: Nat (Mu f) (f (Mu f))
out (In t) = t

-- fusion rule: hdestroy g . hunfold k = g k
```

The categorical semantics of ([13]) reduces correctness of `fold`/`build` rules to the problem of constructing a parametric model which respects that semantics. An alternative approach is taken in ([18]), where the operational semantics-based parametric model of ([22]) is used to validate the fusion rules for algebraic data types introduced in that paper. Extending these techniques to tie the correctness of `fold`/`build` rules into an operational semantics of the underlying functional language is one direction for future work. Finally, the techniques of this paper may provide insights into theories of `fold`s, `build`s, and fusion rules for advanced data types, such as mixed variance data types, GADTs, and dependent types.

References

[1] Abel, A., Matthes, R., Uustalu, T.: Iteration schemes for higher-order and nested datatypes. Theoretical Computer Science 333(1-2), 3–66 (2005)

[2] Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. Proc. Computer Science Logic, pp. 453–468 (1999)

[3] Bayley, I.: Generic Operations on Nested Datatypes. Ph.D. Dissertation, University of Oxford (2001)

[4] Bird, R., Meertens, L.: Nested datatypes. Proc. Mathematics of Program Construction, pp. 52–67 (1998)

[5] Bird, R., Paterson, R.: de Bruijn notation as a nested datatype. Journal of Functional Programming 9(1), 77–91 (1998)

[6] Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing 11(2), 200–222 (1999)

[7] Blampied, P.: Structured Recursion for Non-uniform Data-types. Ph.D. Dissertation, University of Nottingham (2000)

[8] Fiore, M., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. Proc. Logic in Computer Science, pp. 193–202 (1999)

[9] Gill, A.: Cheap Deforestation for Non-strict Functional Languages. Ph.D. Dissertation, Glasgow University (1996)

[10] Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. Proc. Functional Programming Languages and Computer Architecture, pp. 223–232 (1993)

[11] Ghani, N., Haman, M., Uustalu, T., Vene, V.: Representing cyclic structures as nested types. Presented at Trends in Functional Programming (2006)

[12] Ghani, N., Johann, P., Uustalu, T., Vene, V.: Monadic augment and generalised short cut fusion. Proc. International Conference on Functional Programming, pp. 294–305 (2005)

[13] Ghani, N., Uustalu, T., Vene, V.: Build, augment and destroy. Universally. Proc. Asian Symposium on Programming Languages, pp. 327–347 (2003)

[14] Hinze, R.: Polytypic functions over nested datatypes. Discrete Mathematics and Theoretical Computer Science 3(4), 193–214 (1999)

[15] Hinze, R.: Efficient generalized folds. Proc. Workshop on Generic Programming (2000)

[16] Hinze, R.: Functional Pearl: Perfect trees and bit-reversal permutations. Journal of Functional Programming 10(3), 305–317 (2000)

[17] Hinze, R.: Manufacturing datatypes. Journal of Functional Programming 11(5), 493–524 (2001)

[18] Johann, P.: A generalization of short-cut fusion and its correctness proof. Higher-order and Symbolic Computation 15, 273–300 (2002)
[19] MacLane, S.: Categories for the Working Mathematician. Springer, Heidelberg (1971)
[20] Martin, C., Gibbons, J., Bayley, I.: Disciplined efficient generalised folds for nested datatypes. Formal Aspects of Computing 16(1), 19–35 (2004)
[21] McBride, C., McKinna, J.: View from the left. Journal of Functional Programming 14(1), 69–111 (2004)
[22] Pitts, A.: Parametric polymorphism and operational equivalence. Mathematical Structures in Computer Science 10, 1–39 (2000)
[23] Takano, A., Meijer, E.: Shortcut deforestation in calculational form. Proc. Functional Programming Languages and Computer Architecture, pp. 306–313 (1995)

This research was supported in part by NSF award CCF-0429072.

A Substructural Type System for Delimited Continuations*

Oleg Kiselyov[1] and Chung-chieh Shan[2]

[1] FNMOC
oleg@pobox.com
[2] Rutgers University
ccshan@rutgers.edu

Abstract. We propose type systems that *abstractly* interpret small-step rather than big-step operational semantics. We treat an expression or evaluation context as a structure in a linear logic with hypothetical reasoning. Evaluation order is not only regulated by familiar focusing rules in the operational semantics, but also expressed by structural rules in the type system, so the types track control flow more closely. Binding and evaluation contexts are related, but the latter are linear.

We use these ideas to build a type system for delimited continuations. It lets control operators change the answer type or act beyond the nearest dynamically-enclosing delimiter, yet needs no extra fields in judgments and arrow types to record answer types. The typing derivation of a direct-style program desugars it into continuation-passing style.

1 Introduction

Cousot and Cousot [14] originally presented abstract interpretation by starting with a small-step operational semantics. Nevertheless, the typical type system abstractly interprets [13] a denotational or big-step operational semantics, in that each typing rule is the abstract interpretation of a denotational equation or a big-step evaluation judgment. Besides simplicity, one reason to start with such a semantics coarser than a transition system is to make the type system *syntax-directed*: the type of each expression, like its denotation or its big-step evaluation result, is determined by structural induction over the expression. However, when the language involves effects (especially control effects), it can be easier to specify and reason with a small-step semantics (especially evaluation contexts) [68].

A canonical effect that makes semantics and types harder to determine inductively is *delimited control* [25, 26]. With this effect, an expression may access its *delimited continuation* [17, 18, 19] or *delimited evaluation context* as a first-class value. This ability is useful in backtracking search [12, 18, 44, 59], direct-style representations of monads [30, 31, 32], the continuation-passing-style (CPS)

* Thanks to Olivier Danvy, Andrzej Filinski, Michael Stone, Philip Wadler, and the anonymous referees. The appendices to this paper are online at http://okmij.org/ftp/papers/delim-control-logic.pdf

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 223–239, 2007.
© Springer-Verlag Berlin Heidelberg 2007

transformation [17, 18, 19], partial evaluation [6, 7, 10, 16, 23, 33, 37, 47, 64], Web interactions [35, 53], mobile code [50, 56, 60], and linguistics [9, 58].

This paper presents a new type system for delimited control as an example of typing by small-step abstract interpretation. Sect. 2 introduces delimited control, explains why answer types are crucial, and points out shortcomings in how the existing type systems track answer types. We then address the shortcomings in the rest of the paper. As a stepping stone, Sect. 3 introduces small-step typing using the familiar simply-typed λ-calculus. Sect. 4 then presents the $\lambda\xi_0$-calculus, a language with delimited control and small-step typing, and a type-checking algorithm for it. Our Twelf code online at `http://pobox.com/~oleg/ftp/packages/small-step-typechecking.tar.gz` implements type checking and contains numerous tests and sample derivations.

2 Answer Types

The intuition behind delimited control may be conveyed by the two programs below. They are written in the language with delimited control formally defined in Fig. 3, enriched with string "literals" and concatenation $\frown$. The first program shows that a *control delimiter* alone does not affect the evaluation result.

$$\#\,\$\,\text{“Goldilocks said: ”} \frown (\#\,\$\,\text{“This porridge is ”} \frown \text{“too hot”} \frown \text{“. ”}) \qquad (1)$$

This program contains two control delimiters, notated $\#\,\$\,\ldots$ where the subexpression $\ldots$ extends as far to the right as possible. (We pronounce # "reset" and \$ "plug".) The delimiter to the left surrounds the whole program, whereas the delimiter to the right surrounds the subexpression that computes what Goldilocks said. The program computes the string "Goldilocks said: This porridge is too hot. ". The delimiters affect the result only in the presence of a control operator that captures a delimited continuation, as in the following program.

$$\#\,\$\,\text{“Goldilocks said: ”} \frown \big(\#\,\$\,\text{“This porridge is ”} \frown \\ \big(\xi_0 k.(k\,\$\,\text{“too hot”}) \frown (k\,\$\,\text{“too cold”}) \frown (k\,\$\,\text{“just right”})\big) \frown \text{“. ”}\big) \qquad (2)$$

The control operator $\xi_0 k$ (pronounced "shift-zero k") removes, and binds k to, the current continuation up to the nearest dynamically-enclosing delimiter. Once this continuation is captured, an expression such as "too hot" can be plugged into it, notated $k\,\$$ "too hot" in the scope of k. In (2), k prepends "This porridge is " and appends ". " to any string plugged in, so the program computes "Goldilocks said: This porridge is too hot. This porridge is too cold. This porridge is just right. ". The prefix "Goldilocks said: " is not tripled because it is not captured in k.

These examples illustrate the distinction between delimited and undelimited control: a delimited continuation represents only a prefix of the default future of the computation. This prefix maps a subexpression's value (such as "too hot") to an intermediate result at the delimiter (such as "This porridge is too hot. "). A type system for delimited control must thus track these intermediate results' types as part of the effects of expressions. These types are called *answer types*.

The only answer type in (1) and (2) is that of strings, but real programs need different answer types at multiple delimiters. On one hand, it is useful for an expression to access its delimited continuation beyond the nearest dynamically-enclosing delimiter: to combine multiple monadic effects [18, 31, 32], to normalize λ-terms with sums [7], and to simulate exceptions and mutable references [39] and dynamic binding [45]. These uses motivate type systems [38, 39, 51] that maintain a stack or heap of answer types. On the other hand, it is also useful for an expression to change the answer type, that is, to capture one delimited continuation then install another with a different answer type: to create functions [18], to find list prefixes [10], to represent parameterized monads [5], and to analyze questions and polarity in natural language [58]. These uses motivate a type system [17] that is sensitive to evaluation order.

Unfortunately, no existing type system for delimited control subsumes all others, so no clear choice emerges for practical use. Moreover, the existing type systems attach answer types to judgments and arrows as effect annotations [32, 34, 48, 61, 62, 63, 65]. These annotations obscure any logical interpretation of the types via the Curry-Howard correspondence [4, 36, 42].

For example, Danvy and Filinski [17] uses typing judgments of the form $\rho, \alpha \vdash E : \tau, \beta$, where ρ is a typing environment, α and β are answer types, E is a term, and τ is its type. If E changes the answer type, then the answer types α and β may differ. If α, τ, β are atomic, then this judgment indicates that the CPS transformation of E has the type $(\tau \to \alpha) \to \beta$ in the simply-typed λ-calculus. The typing rule for λ-expressions reads

$$\frac{[x \mapsto \sigma]\rho, \alpha \vdash E : \tau, \beta}{\rho, \delta \vdash \lambda x.\, E : (\sigma/\alpha \to \tau/\beta), \delta.} \tag{3}$$

If $\sigma, \alpha, \tau, \beta$ are atomic, then the type $\sigma/\alpha \to \tau/\beta$ above indicates that the CPS transformation of $\lambda x.\, E$ has the simple type $\sigma \to (\tau \to \alpha) \to \beta$.

The extra fields for answer types in these judgments and arrow types still leave no room for delimiters beyond the nearest dynamically-enclosing one. Also, the comma and slash are not logical connectives in their own right, so the logical interpretation of the extra fields is unclear, unlike with undelimited control [36] or the simply-typed λ-calculus, which do not have varying answer types. Kameyama [42] logically interprets a static variant of Danvy ad Filinski's system that does not allow changing the answer type. Ariola et al. [4] embed Danvy ad Filinski's type system in subtractive logic, but the embedding is not full: the target includes undelimited control but the source does not.

In sum, we want a type system for delimited control that accommodates an arbitrary number of changing answer types. We achieve this goal by assigning types not just to expressions but also to evaluation contexts, as guided by CPS. For example, the delimited continuation k in (2) yields a string answer when a string is plugged into it; we write this type as $\mathrm{string} \uparrow \mathrm{string}$. The ξ_0-term in (2) is an expression that yields a string answer when it is plugged into such a delimited continuation; we write this type as $(\mathrm{string} \uparrow \mathrm{string}) \downarrow \mathrm{string}$. The return types to the right of the function-like *connectives* $\uparrow$ and $\downarrow$ are answer types.

Statements	$M ::= C \mathbin{\$} E$
Terms	$E, F ::= V \mid FE$
Values	$V ::= x \mid \lambda x.\, E$
Coterms	$C ::= \# \mid E, C \mid C; V$
Types	$T, U ::= U \to T \mid \text{string} \mid \text{int} \mid \cdots$
Cotypes	$S ::= U \uparrow T$
Statement contexts	$M[\ ] ::= C[\ ] \mathbin{\$} E \mid C \mathbin{\$} E[\ ]$
Term contexts	$E[\ ] ::= [\ ] \mid E[\ ]E \mid EE[\ ]$
Coterm contexts	$C[\ ] ::= E[\ ], C \mid E, C[\ ] \mid C[\ ]; V$

Statement equality

$$C \mathbin{\$} FE = E, C \mathbin{\$} F \qquad C \mathbin{\$} VE = C; V \mathbin{\$} E$$

Transitions

$$C \mathbin{\$} (\lambda x.\, E)V \rightsquigarrow C \mathbin{\$} E\{x \mapsto V\}$$

Typing

$$\frac{\begin{matrix}[x:U]\\ \vdots\\ \# \mathbin{\$} E : T\end{matrix}}{\lambda x.\, E : U \to T}\,\lambda \qquad \frac{F : U \quad \begin{matrix}[x:U]\\ \vdots\\ M[x] : T\end{matrix}}{M[F] : T}\,M[U] \qquad \frac{\begin{matrix}[x:U]\\ \vdots\\ Vx : T\end{matrix}}{V : U \to T}\,{\to}\mathrm{I} \qquad \frac{\begin{matrix}[x:U]\\ \vdots\\ C \mathbin{\$} x : T\end{matrix}}{C : U \uparrow T}\,{\uparrow}\mathrm{I}$$

$$\frac{V : U}{\# \mathbin{\$} V : U}\,\# \qquad \frac{F : U \to T \quad E : U}{FE : T}\,{\to}\mathrm{E} \qquad \frac{C : U \uparrow T \quad E : U}{C \mathbin{\$} E : T}\,{\uparrow}\mathrm{E}$$

Fig. 1. Warm-up: the simply typed λ-calculus with small-step typing. For uniformity with Fig. 3 below, the notation is somewhat unconventional: we use the metavariable E for terms and also $E[\ ]$ for a term with a term-hole. The only variable binder is λx. Following the Barendregt variable convention, the variable x in the $M[U]$, $\to$I, and $\uparrow$I rules is to be chosen fresh, not to occur free in the conclusion. The assumption $x : U$ discharged in these rules always occurs exactly once, because the hole $[\ ]$ appears linearly in a context and no rule duplicates subterms.

An evaluation context is not usually part of an expression. Thus, to assign types to evaluation contexts, we need to revise our notion of a syntax-directed type system. We do so first for the λ-calculus, then return to delimited control.

3 Warm-Up: Small-Step Typing

Fig. 1 shows a type system for the pure λ-calculus that includes small-step as well as big-step abstract interpretation. The purpose of this system is to prepare for the main development in Sect. 4. Many aspects of this system seem contrived and redundant when taken alone, but they are necessary for delimited control. The

accompanying Twelf code in `lfix-calc.elf` implements small-step abstract interpretation for this language and contains numerous sample derivations.

Besides *terms* E, this language defines two other syntactic categories: *coterms* C and *statements* M. Whereas a term can contain subterms, a statement is a complete program like a top-level term (a common notion in small-step semantics). A statement is formed by, and decomposes into, plugging a term into a coterm. This distinction between terms and statements is refined in Sect. 4.

A *statement context* $M[\,]$ (respectively *term context* $E[\,]$, *coterm context* $C[\,]$) is a statement (respectively term, coterm) with a hole that can be filled by any term, such that the hole is not under a binder λx. We write $M[E]$ for the context $M[\,]$ filled with the term E. Judgments of the form $M : T$, $E : T$, and $C : S$ assign types T and cotypes S to statements M, terms E, and coterms C.

A coterm is an evaluation context, that is, a defunctionalized continuation of a substitution-based evaluation function [1, 2, 3, 21]: the coterm $\#$ is the identity continuation; the coterm E, C means to apply to the argument term E then continue with the coterm C; the coterm $C; V$ means to apply the function value V then continue with the coterm C. Formally, a simple bijection maps coterms C to term contexts $E[\,]$ in which only values appear to the left of the hole.

Definition 1. *Associate with each coterm C a term context $C^\dagger[\,]$ by induction:*

$$\#^\dagger[\,] = [\,], \qquad (E, C)^\dagger[\,] = C^\dagger\big[[\,]E\big], \qquad (C; V)^\dagger[\,] = C^\dagger\big[V[\,]\big]. \tag{4}$$

Every coterm "comes with its own control delimiter", in that it always ends in the identity continuation $\#$. Hence a coterm represents a complete (delimited) continuation, not a list of stack frames. It makes no sense to "concatenate" coterms, for example to try to combine the coterms $E_1, \#$ and $E_2, \#$ into $E_1, (E_2, \#)$.

A statement, of the form $C \,\$\, E$ (pronounced "plug"), represents the term E plugged into the coterm C. It is a state of the CK machine [25, 27, 28]. A statement can also be understood as a zipper [41] over a term.

Among the binary constructors, \$ has the lowest precedence, and juxtaposition (for function application) the highest. All binary constructors associate to the right, except juxtaposition associates to the left.

3.1 A Substructural Logic for Expressions and Evaluation Contexts

Two *statement equality* rules enforce left-to-right, call-by-value evaluation, as evaluation contexts [25] and focusing [20] do in other accounts. Formally, our equality rules are equations in the multisorted algebra of statements, terms, and coterms, as well as the following reversible typing rules.

$$\frac{C \,\$\, F E : T}{E, C \,\$\, F : T} = \qquad\qquad \frac{C \,\$\, V E : T}{C; V \,\$\, E : T} = \tag{5}$$

These rules let us navigate around a term using a statement as a zipper.

Proposition 1. *The statement equality rules equate the statements* $C_1 \mathbin{\$} E_1$ *and* $C_2 \mathbin{\$} E_2$ *iff the terms* $C_1^\dagger[E_1]$ *and* $C_2^\dagger[E_2]$ *are equal.*

Because $C^\dagger[\,]$ is always an evaluation context for left-to-right, call-by-value evaluation, Prop. 1 ties evaluation order to transitions in the dynamic semantics as well as types in the static semantics. The # typing rule makes $\# \mathbin{\$} V$ effectively equivalent to V, so as to type # as the identity continuation.

The separator : in judgments is the turnstile in a substructural logic. This logic has four sorts (namely statements, terms, values, and coterms). It allows no exchange, associativity, weakening, or contraction except by the structural rules in (5). It builds structures from values using six multiplicative-conjunctive punctuation marks [54], or *modes*: four binary (juxtaposition and $, ;), one nullary (#), and one unary (the implicit coercion from a value to a term). This logic is thus a restricted *multimodal type-logical grammar* (TLG).

Multimodal TLG is a generalization of the Lambek calculus [46] whose proof theory and Kripke semantics are well-studied and well-behaved: there are sound and complete natural-deduction and sequent calculi with cut elimination [49, 52]. Our statements, terms, and coterms (to the left of the turnstile) are TLG structures, restricted to be sort-correct. Our types and cotypes (to the right of the turnstile) are TLG formulae, restricted to use only two implication connectives $\rightarrow$ and $\uparrow$ out of the four pairs available in TLG (one pair per binary mode).

Viewed as a substructural logic, this type system is mostly familiar. The $\rightarrow$I, $\rightarrow$E, $\uparrow$I, and $\uparrow$E rules establish $\rightarrow$ as the right-implication of juxtaposition and $\uparrow$ as the right-implication of $. As in the Lambek calculus, x occurs linearly in the premises of $\rightarrow$I and $\uparrow$I; these premises could be just $VU : T$ and $C \mathbin{\$} U : T$ if, in the spirit of abstract interpretation, types were values. Of these rules, only $\rightarrow$I is needed for delimited control in Sect. 4, but we include introduction and elimination rules for all binary connectives to relate them to TLG. Still, as in the original Lambek calculus, no binary mode comes with any product connective, such as any connective $*$ such that $F : T$ and $E : U$ justify $FE : T * U$. This distinction between modes and connectives is standard in substructural logic.

In contrast to the binary modes, the (implicit) unary mode for coercing values into terms does correspond to an (implicit) product connective. The $M[U]$ rule is the standard elimination rule for this connective in natural deduction. This rule lets us use any expression with a pure type—which in the pure λ-calculus is any type—as a value. This rule is more general than the $\uparrow$E rule in that it allows substituting a nonvalue F into an operand position in $M[\,]$ even if the corresponding (preceding) operator position contains a nonvalue as well. In particular, the equality rules can treat a term F of a pure type U as a value x.

Finally, the familiar λ rule creates a function value. Unlike in the $\rightarrow$I and $\uparrow$I rules, the bound variable x in the λ rule may appear multiple times, or not at all, in the body E of the abstraction $\lambda x.\, E$. In other words, a λ-bound variable is intuitionistic rather than substructural: it admits weakening and contraction.

The transition rule in Fig. 1 is β-reduction, restricted to when the argument V is a value. Transitions operate on statements, not terms: to run a term E as a complete program, we run the statement $\# \mathbin{\$} E$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{inc}:\mathrm{int}\to\mathrm{int}\quad [x:\mathrm{int}]^2}{\mathrm{inc}\,x:\mathrm{int}}{\to}\mathrm{E}\quad \dfrac{[y:\mathrm{int}]^1}{\#\,\$\,y:\mathrm{int}}\#}{\#\,\$\,\mathrm{inc}\,x:\mathrm{int}}M[U]^1}{\#;\mathrm{inc}\,\$\,x:\mathrm{int}}=}{\#;\mathrm{inc}:\mathrm{int}\uparrow\mathrm{int}}{\uparrow}\mathrm{I}^2\quad \dfrac{\mathrm{inc}:\mathrm{int}\to\mathrm{int}\quad 2:\mathrm{int}}{\mathrm{inc}\,2:\mathrm{int}}{\to}\mathrm{E}}{\#;\mathrm{inc}\,\$\,\mathrm{inc}\,2:\mathrm{int}}{\uparrow}\mathrm{E}}{\#\,\$\,\mathrm{inc}\,(\mathrm{inc}\,2):\mathrm{int}}=$$

$$\dfrac{\dfrac{\mathrm{inc}:\mathrm{int}\to\mathrm{int}\quad \dfrac{\mathrm{inc}:\mathrm{int}\to\mathrm{int}\quad 2:\mathrm{int}}{\mathrm{inc}\,2:\mathrm{int}}{\to}\mathrm{E}}{\mathrm{inc}\,(\mathrm{inc}\,2):\mathrm{int}}{\to}\mathrm{E}\quad \dfrac{[y:\mathrm{int}]^1}{\#\,\$\,y:\mathrm{int}}\#}{\#\,\$\,\mathrm{inc}\,(\mathrm{inc}\,2):\mathrm{int}}M[U]^1$$

Fig. 2. Two derivations of "$\#\,\$\,\mathrm{inc}\,(\mathrm{inc}\,2):\mathrm{int}$" from "$\mathrm{inc}:\mathrm{int}\to\mathrm{int}$" and "$2:\mathrm{int}$"

3.2 Normalizing Small-Step Derivations to Big-Step Derivations

Despite all these rules, the system is equivalent to the simply-typed λ-calculus.

Proposition 2. *Write $E :: T$ if the term E has the type T in the simply-typed λ-calculus. Then, under any typing assumptions $x_1 : T_1$, $x_1 :: T_1$, ..., $x_n : T_n$, $x_n :: T_n$: (a) $E : T$ iff $E :: T$. (b) $C\,\$\,E : T$ iff $C^\dagger[E] :: T$. (c) $C : U \uparrow T$ iff $\lambda x.\, C^\dagger[x] :: U \to T$.*

Proof. [$\Rightarrow$] By induction on a derivation in our system.

[$\Leftarrow$] (a) By induction on a simple-type derivation, using our $\to$E rule and
$$\dfrac{\dfrac{\begin{matrix}[x:U]^2\\ \vdots\\ E:T\end{matrix}\quad \dfrac{[y:T]^1}{\#\,\$\,y:T}\#}{\#\,\$\,E:T}M[U]}{\lambda x.\,E:U\to T}\lambda^2.$$
(b) Feed the conclusion of
$$\dfrac{\begin{matrix}\vdots\ \text{Use (a)}\\ C^\dagger[E]:T\end{matrix}\quad \dfrac{[y:T]^1}{\#\,\$\,y:T}\#}{\#\,\$\,C^\dagger[E]:T}M[U]^1$$
to Prop. 1. (c) Derive $C\,\$\,x : U$ by (b), then use $\uparrow$I. □

Because the simply-typed λ-calculus enjoys preservation, progress, and decidable type reconstruction, our system does as well.

Fig. 2 shows two typing derivations of the same statement $\#\$\mathrm{inc}\,(\mathrm{inc}\,2)$, where the value inc is the integer increment function. At the bottom is the result of converting the familiar derivation in the simply-typed λ-calculus to our system using part (b) of Prop. 2. We call this derivation *big-step* because it follows the applicative structure of the expression: it determines the type of $\mathrm{inc}\,(\mathrm{inc}\,2)$ from the type of its parts inc and $\mathrm{inc}\,2$. At the top is a *small-step* derivation, which separates the expression $\mathrm{inc}\,2$ from its evaluation context $\#;\mathrm{inc}$. This derivation represents the term $\mathrm{inc}\,2$ by the variable x in $\#;\mathrm{inc}\,\$\,x$. Thus x in the typing context is an abstract value, in the sense of abstract interpretation [13, 14].

Because all expressions in the λ-calculus are pure, they can be derived by both big-step and small-step. (In Sect. 4, impure expressions—which incur delimited-control effects—require small-step derivations.) These derivations are related by

a normalization process (not cut elimination, because our type system is based on natural deduction rather than sequents) detailed in Appendix A. There we normalize the small-step derivation in Fig. 2 to the big-step derivation below.

4 Delimited Control

Fig. 3 defines the static and dynamic semantics of the $\lambda\xi_0$-calculus, a new language with delimited control. The most prominent difference between this system and Fig. 1 is new non-value terms of the form $\xi_0 k.\, E$. These terms have *impure* types of the form $(U \uparrow T_1) \downarrow T_2$. As we discussed for the example (2) above, such a type means that, when the term is plugged into a coterm of cotype $U \uparrow T_1$ (in other words, a coterm which yields an answer of type T_1 when a value of type U is plugged into it), the combination yields an answer of type T_2. All other types are *pure*. We distinguish pure types by using the metavariable U rather than T.

Terms	$E, F ::= V \mid FE \mid C \,\$\, E \mid \xi_0 k.\, E$
Values	$V ::= x \mid \lambda x.\, E$
Coterms	$C ::= k \mid \# \mid E, C \mid C; V$
Types	$T ::= U \mid S \downarrow T$
Pure types	$U ::= U \to T \mid \text{string} \mid \text{int} \mid \cdots$
Cotypes	$S ::= U \uparrow T$
Term contexts	$E[\,] ::= [\,] \mid E[\,]E \mid EE[\,] \mid C[\,] \,\$\, E \mid C \,\$\, E[\,]$
Coterm contexts	$C[\,] ::= E[\,], C \mid E, C[\,] \mid C[\,]; V$

Term equality

$$C \,\$\, FE = E, C \,\$\, F \qquad C \,\$\, VE = C; V \,\$\, E \qquad \# \,\$\, V = V$$

Transitions

$$C_1 \,\$\, \cdots \,\$\, C_n \,\$\, (\lambda x.\, E)V \rightsquigarrow C_1 \,\$\, \cdots \,\$\, C_n \,\$\, E\{x \mapsto V\}$$
$$C_1 \,\$\, \cdots \,\$\, C_n \,\$\, C \,\$\, (\xi_0 k.\, E) \rightsquigarrow C_1 \,\$\, \cdots \,\$\, C_n \,\$\, E\{k \mapsto C\}$$

Typing

$$\dfrac{\begin{matrix}[x:U]\\ \vdots \\ E:T\end{matrix}}{\lambda x.\, E : U \to T}\,\lambda \qquad \dfrac{\begin{matrix}[k:S]\\ \vdots \\ E:T\end{matrix}}{\xi_0 k.\, E : S \downarrow T}\,\xi_0 \qquad \dfrac{F:U \quad \begin{matrix}[x:U]\\ \vdots \\ E[x]:T\end{matrix}}{E[F] : T}\,E[U]$$

$$\dfrac{\begin{matrix}[x:U]\\ \vdots \\ Vx:T\end{matrix}}{V : U \to T}\,{\to}\mathrm{I} \qquad \dfrac{\begin{matrix}[k:S]\\ \vdots \\ k \,\$\, E:T\end{matrix}}{E : S \downarrow T}\,{\downarrow}\mathrm{I} \qquad \dfrac{\begin{matrix}[x:U]\\ \vdots \\ C \,\$\, x:T\end{matrix}}{C : U \uparrow T}\,{\uparrow}\mathrm{I}$$

$$\dfrac{F : U \to T \quad E : U}{FE : T}\,{\to}\mathrm{E} \qquad \dfrac{C : S \quad E : S \downarrow T}{C \,\$\, E : T}\,{\downarrow}\mathrm{E} \qquad \dfrac{C : U \uparrow T \quad E : U}{C \,\$\, E : T}\,{\uparrow}\mathrm{E}$$

Fig. 3. The $\lambda\xi_0$-calculus: syntax and semantics

A term of the form $\xi_0 k.\, E$ in the $\lambda\xi_0$-calculus may capture not just its immediately surrounding delimited continuation in the covariable k but also delimited continuations beyond the nearest dynamically-enclosing delimiter, if the body E invokes another control operator $\xi_0 k.\, E'$. Hence our primitive control operator is *dynamic* [17, 18, 19]: the answer types T_1 and T_2 in the impure type $(U \uparrow T_1) \downarrow T_2$ may themselves be impure. We also allow changing the answer type, so T_1 and T_2 may differ. Thus our type system is the first to achieve both desiderata in Sect. 2: to reach beyond the nearest delimiter and to change the answer type.

More precisely, our ξ_0 is not Danvy and Filinski's *shift* but the variation in their Appendix C [17], which we pronounce "shift-zero". In the untyped setting, ξ_0, shift, *control* [25, 26], and their variants [38, 39, 40] are all macro-expressible in terms of each other [43, 57]. In the typed setting, ξ_0 easily emulates shift [17] (shift $k.\, E$ translates to $\xi_0 k.\, \# \mathbin{\$} E$), but it remains to relate ξ_0 to other type systems of control. In particular, unlike Gunter et al.'s system [38, 39], we assure that a program of a pure type never gets stuck due to a missing delimiter. We are also able to type more terms, for example $\xi_0 k.\, \lambda x.\, k \mathbin{\$} x$, which changes the answer type.

Existing languages with delimited control generally introduce a primitive expression form, called "reset" or "prompt", to insert a control delimiter. In contrast, our language includes terms of the form $C \mathbin{\$} E$, which means to plug the term E into the coterm C. We call these terms *statements*. Unlike in Fig. 1, a statement is a term. Because every coterm "comes with its own delimiter" in that it always ends in either the identity continuation $\#$ or a covariable k, our term $\# \mathbin{\$} E$ serves the purpose of "reset E" or "prompt E" in previous work, even though \$ alone is not a delimiter. Now that $\# \mathbin{\$} E$ is as much a term as E, we replace the typing rule $\#$ in Fig. 1 by a new term equality rule $\# \mathbin{\$} V = V$.

To evaluate programs that use delimited control, Fig. 3 defines two transition rules. The first rule substitutes an argument value V into the body E in $\lambda x.\, E$, whereas the second rule substitutes an argument coterm C into the body E in $\xi_0 k.\, E$. Both rules operate inside a term context $C_1 \mathbin{\$} \cdots \mathbin{\$} C_n \mathbin{\$} [\,]$, where $n \geq 0$. This term context is the *metacontinuation* [67] that appears in CPS semantics [17, 18, 19] and abstract machines [11, 24] for delimited control.

As in Sect. 3.1, the term equality rules in Fig. 3 are equations in the multi-sorted algebra of terms and coterms as well as the reversible typing rules

$$\frac{E'[C \mathbin{\$} F E] : T}{\overline{E'[E, C \mathbin{\$} F] : T}} = \qquad \frac{E'[C \mathbin{\$} V E] : T}{\overline{E'[C; V \mathbin{\$} E] : T}} = \qquad \frac{E'[V] : T}{\overline{E'[\# \mathbin{\$} V] : T}} = \tag{6}$$

and the corresponding rules replacing $E'[\,]$ and T by $C'[\,]$ and S.

As a substructural logic, the $\lambda\xi_0$-calculus has the same binary modes as Fig. 1, but allows not just the right-implication $\uparrow$ of the \$ mode but also its dual, the left-implication $\downarrow$. As before, this logic has neither negation nor multiple conclusions, so we interpret delimited control as multimodal *intuitionistic* logic via the Curry-Howard correspondence. The implication connectives each come with their own introduction and elimination rules, but $\downarrow$E and $\uparrow$E both conclude with $C \mathbin{\$} E : T$. This apparent ambiguity is standard in type systems descended from the Lambek

$$\dfrac{\dfrac{[k:\text{string}\uparrow\text{int}]^1 \quad x:\text{string}}{k\,\$\,x:\text{int}}\uparrow\text{E}}{x:(\text{string}\uparrow\text{int})\downarrow\text{int}}\downarrow\text{I}^1 \qquad \dfrac{\dfrac{\dfrac{[k':\text{int}\uparrow T]^2 \quad \dfrac{[k:\text{string}\uparrow\text{int}]^1 \quad x:\text{string}}{k\,\$\,x:\text{int}}\uparrow\text{E}}{k'\,\$\,k\,\$\,x:T}\uparrow\text{E}}{k\,\$\,x:(\text{int}\uparrow T)\downarrow T}\downarrow\text{I}^2}{x:(\text{string}\uparrow\text{int})\downarrow(\text{int}\uparrow T)\downarrow T}\downarrow\text{I}^1$$

Fig. 4. Two type derivations for a string x

calculus [46]. Indeed, if the first of the two transition rules in Fig. 3 did not restrict β-reductions to take place only when the argument is a value, then the term $\#\ \$\ (\lambda x.\ \text{"call by name"})(\xi_0 k.\ \text{"call by value"})$ would transition not only to "call by value" but also to $\#\ \$$ "call by name", which is equal to "call by name". Just as the dynamic semantics restricts argument terms to values, the static semantics restricts argument types to pure types.

With both $\uparrow$ and $\downarrow$ present, each term has an infinite number of types. For example, Fig. 4 shows that a string also has the types $(\text{string}\uparrow\text{int})\downarrow\text{int}$ and $(\text{string}\uparrow\text{int})\downarrow(\text{int}\uparrow T)\downarrow T$ for any T. In fact, the entailment relation of the logic is a partial order of subtyping, which we notate as $T_1 \le T_2$. This relation is generated by $U \le (U\uparrow T)\downarrow T$ along with congruences, covariance, and contravariance.

We show two example terms before stating formal properties. Appendix B gives the derivations. The term $(\xi_0 k.\,1)(\xi_0 k.\ \text{"x"})$ has the type $S\downarrow\text{int}$ for any cotype S. The derivation is CPS-like, even though the term is in direct style like all our programs. Since the type is impure, the term may (and does) get stuck if run alone, but can appear in safe programs such as $\lambda x.(\xi_0 k.\,1)(\xi_0 k.\ \text{"x"})$. As with (only) Danvy and Filinski's type system for shift [17], the answer type varies between int (for the subterm $\xi_0 k.\,1$) and string (for $\xi_0 k.\ \text{"x"}$), but the overall answer type is int rather than string due to (left-to-right) evaluation order.

The impure term $\text{inc}(\xi_0 k.\,\xi_0 k'.\ \text{"x"})$ reaches beyond the nearest enclosing delimiter, which shift does not allow. It has the type $(\text{int}\uparrow T)\downarrow S\downarrow\text{string}$, where int is the type of the result of inc.

Proposition 3 (Preservation). *If $E[E_1]:T$ and $E_1 \rightsquigarrow E_2$ then $E[E_2]:T$.*

We sketch the proof by stating three lemmas. The first lemma is termed *direct compositionality on demand* by Barker [8]: a subterm of a typed term is typed.

Lemma 1. *If $E[E_1]:T$, then there exists some type T_1 such that $E_1:T_1$ and whenever $E_1':T_1$ we have $E[E_1']:T$.*

The two remaining lemmas are less trivial than usual because the typing rules $\to$I, $\downarrow$I, and $\uparrow$I are not syntax-directed.

Lemma 2. *If $(\lambda x.\,E)V:T$ then $E\{x\mapsto V\}:T$.*

Lemma 3. *If $C\,\$\,(\xi_0 k.\,E):T$ then $E\{k\mapsto C\}:T$.*

Proposition 4 (Progress). *If $C\,\$\,E:U$, then either $C\,\$\,E$ is a value (that is, $C=\#$ and E is a value) or $C\,\$\,E \rightsquigarrow E'$ for some E'.*

The small-step type-checking algorithm in Sect. 4.1 offers an appealing proof of this proposition: on one hand, it is correct with respect to the static semantics (Corollary 1 below); on the other hand, it is sound as an abstract interpretation of the dynamic semantics.

Proposition 5 (Determinism). *If $E \rightsquigarrow E_1'$ and $E \rightsquigarrow E_2'$, then $E_1' = E_2'$.*

Proposition 6 (Termination). *If $E : T$, then there is no infinite transition sequence $E \rightsquigarrow E_1 \rightsquigarrow E_2 \rightsquigarrow \cdots$.*

4.1 Type-Checking Algorithm

Fig. 5 shows a type-checking algorithm, expressed as moded inference rules, for a variant of our language in which binders are annotated with types and cotypes. The accompanying Twelf code in `lxi0-calc.elf` implements this algorithm. It produces a CPS-like derivation from a direct-style program. For the term $(\xi_0 k.\, 1)$ $(\xi_0 k.\, \text{“x”})$ above, our algorithm returns its type and even the cotypes of k's.

Our type checker performs abstract interpretation by traversing the term, just as the focusing process of an evaluator would traverse the term in search of a *focus*, and replacing nonvariable subterms by typed variables one by one.

Definition 2. *A* focus *is a term of the form $V_1 V_2$, $C \,\$\, E$, or $\xi_0 k{:}S.\, E$.*

As explained in Sect. 3.2, each typed variable is an abstract value. In addition, our covariables are cotyped and can be thought of as abstract covalues.

Whereas the focusing process of an evaluator need only traverse a known term plugged into a known coterm, the type checker often needs to traverse a term without knowing what coterm it may be plugged into. For example, to check the function $\lambda x{:}\text{int}.\, \text{inc}(\xi_0 k{:}\text{int}{\uparrow}\text{string}.\, 3)$, the checker needs to visit the subterm $\xi_0 k{:}\text{int}{\uparrow}\text{string}.\, 3$ without knowing the context where the function will be applied and hence its body evaluated. In other words, the checker needs to use an equality rule $\dfrac{\langle\rangle; \text{inc} \,\$\, (\xi_0 k{:}\text{int}{\uparrow}\text{string}.\, 3) : T}{\langle\rangle \,\$\, \text{inc}(\xi_0 k{:}\text{int}{\uparrow}\text{string}.\, 3) : T}$ where $\langle\rangle$ represents an unknown coterm. To perform such traversals, we introduce the notion of an *incomplete coterm* $C\langle\rangle$, which is like a coterm but ends in $\langle\rangle$ rather than in $\#$ or k. Some of the checker's judgments use the notation $C\langle\rangle \,¢\, E$. Intuitively, $C\langle\rangle \,¢\, E$ means the term obtained by plugging E into the term context $C\langle\rangle^\dagger[\,]$ defined below.

Definition 3. *Associate with each incomplete coterm $C\langle\rangle$ a term context $C\langle\rangle^\dagger[\,]$:*

$$\langle\rangle^\dagger[\,] = [\,], \qquad (E, C\langle\rangle)^\dagger[\,] = C\langle\rangle^\dagger\big[[\,]E\big], \qquad (C\langle\rangle; V)^\dagger[\,] = C\langle\rangle^\dagger\big[V[\,]\big]. \tag{7}$$

The pair $C\langle\rangle$ and E is a zipper over the term $C\langle\rangle^\dagger[E]$, “unzipped” to display the subterm E. We also use $C \,¢\, E$ to mean the term $C \,\$\, E$, where C is a (complete) coterm. Whereas $C \,\$\, E$ is always a statement, the term $C\langle\rangle^\dagger[E]$ is only a statement when $C\langle\rangle$ is $\langle\rangle$ and E is a statement.

Unlike a ξ_0-bound covariable like k, this unknown coterm is not annotated with its cotype. Rather, the checker infers its (greatest) cotype, using a judgment

Terms (annotated)	$E, F ::= V \mid FE \mid C \mathbin{\$} E \mid \xi_0 k{:}S.\, E$
Values (annotated)	$V ::= x \mid \lambda x{:}U.\, E$
Incomplete coterms	$C\langle\rangle ::= \langle\rangle \mid E, C\langle\rangle \mid C\langle\rangle; V$
Possibly incomplete coterms	$C\langle?\rangle ::= C \mid C\langle\rangle$

Judgments (hats indicate output as opposed to input parameters)

$$T_1 \le T_2 \qquad E \Rightarrow \hat{T} \qquad C\langle?\rangle ¢ E \Rightarrow \hat{T} \qquad \hat{S} \Leftarrow C\langle\rangle ¢ E : T$$

$$V : \hat{U} \qquad k : \hat{S} \qquad C\langle?\rangle ¢ U \Rightarrow \hat{T} \qquad \hat{S} \Leftarrow C\langle\rangle ¢ U : T$$

Initial query for type inference $\quad \langle\rangle ¢ E \Rightarrow T$

Inference rules for $T_1 \le T_2$

$$\frac{U \text{ primitive}}{U \le U} \qquad \frac{U_1 \le U_2 \quad T_1 \le T_2}{U_1 \le (U_2 \uparrow T_1) \downarrow T_2} \qquad \frac{U_2 \le U_1 \quad T_1 \le T_2}{U_1 \to T_1 \le U_2 \to T_2} \qquad \frac{U_1 \le U_2 \quad T_2 \le T_1 \quad T_1' \le T_2'}{(U_1 \uparrow T_1) \downarrow T_1' \le (U_2 \uparrow T_2) \downarrow T_2'}$$

Inference rules for $E \Rightarrow \hat{T}$

$$\frac{V_1 : U_1 \to T \quad V_2 : U_2 \quad U_2 \le U_1}{V_1 V_2 \Rightarrow T} \qquad \frac{C ¢ E \Rightarrow T}{C \mathbin{\$} E \Rightarrow T} \qquad \frac{\begin{array}{c}[k:S] \\ \vdots \\ \langle\rangle ¢ E \Rightarrow T\end{array}}{\xi_0 k{:}S.\, E \Rightarrow S \downarrow T}$$

Inference rules for $C\langle?\rangle ¢ E \Rightarrow \hat{T}$

$$\frac{V : U}{\langle\rangle ¢ V \Rightarrow U} \qquad \frac{k : U_1 \uparrow T \quad V : U_2 \quad U_2 \le U_1}{k ¢ V \Rightarrow T} \qquad \frac{V : U}{\# ¢ V \Rightarrow U} \qquad \frac{C\langle?\rangle; V ¢ E \Rightarrow T}{E, C\langle?\rangle ¢ V \Rightarrow T}$$

$$\frac{C\langle?\rangle ¢ V_1 V_2 \Rightarrow T}{C\langle?\rangle; V_1 ¢ V_2 \Rightarrow T} \qquad \frac{F \text{ or } E \text{ is not a value} \quad E, C\langle?\rangle ¢ F \Rightarrow T}{C\langle?\rangle ¢ FE \Rightarrow T} \qquad \frac{E \Rightarrow U \quad C\langle?\rangle ¢ U \Rightarrow T}{C\langle?\rangle ¢ E \Rightarrow T}$$

$$\frac{E \Rightarrow (U \uparrow T_1) \downarrow T \quad C ¢ U \Rightarrow T_2 \quad T_2 \le T_1}{C ¢ E \Rightarrow T} \qquad \frac{E \Rightarrow (U \uparrow T_1) \downarrow T \quad S \Leftarrow C\langle\rangle ¢ U : T_1}{C\langle\rangle ¢ E \Rightarrow S \downarrow T}$$

Inference rules for $\hat{S} \Leftarrow C\langle\rangle ¢ E : T$

$$\frac{V : U}{U \uparrow T \Leftarrow \langle\rangle ¢ V : T} \qquad \frac{S \Leftarrow C\langle\rangle; V ¢ E : T}{S \Leftarrow E, C\langle\rangle ¢ V : T} \qquad \frac{S \Leftarrow C\langle\rangle ¢ V_1 V_2 : T}{S \Leftarrow C\langle\rangle; V_1 ¢ V_2 : T}$$

$$\frac{F \text{ or } E \text{ is not a value} \quad S \Leftarrow E, C\langle?\rangle ¢ F : T}{S \Leftarrow C\langle\rangle ¢ FE : T} \qquad \frac{E \Rightarrow U \quad S \Leftarrow C\langle\rangle ¢ U : T}{S \Leftarrow C\langle\rangle ¢ E : T}$$

$$\frac{E \Rightarrow (U \uparrow T_1) \downarrow T_2 \quad T_2 \le T \quad S \Leftarrow C\langle\rangle ¢ U : T_1}{S \Leftarrow C\langle\rangle ¢ E : T}$$

Other inference rules

$$\frac{\begin{array}{c}[x:U] \\ \vdots \\ \langle\rangle ¢ E \Rightarrow T\end{array}}{\lambda x{:}U.\, E : U \to T} \qquad \frac{\begin{array}{c}[x:U] \\ \vdots \\ C\langle?\rangle ¢ x \Rightarrow T\end{array}}{C\langle?\rangle ¢ U \Rightarrow T} \qquad \frac{\begin{array}{c}[x:U] \\ \vdots \\ S \Leftarrow C\langle\rangle ¢ x : T\end{array}}{S \Leftarrow C\langle\rangle ¢ U : T}$$

Fig. 5. Type-checking algorithm for delimited control

form $S \Leftarrow C\langle\rangle \cent E : T$ whose modes are rather special: the only output parameter is S. Any unknown coterm into which the term $C\langle\rangle^\dagger[E]$ is plugged for evaluation needs to have the cotype S in order to yield an answer of type T.

Our approach to "visit subterms in evaluation position before the context in which they occur" may be an instance of *tridirectional type-checking* [22]. The 4th–6th rules for $C\langle ? \rangle \cent E \Rightarrow \hat{T}$ and the 2nd–4th rules for $\hat{S} \Leftarrow C\langle\rangle \cent E : T$ are focusing rules: they traverse the applicative structure of E to find the next subterm to abstractly interpret according to the evaluation order.

Proposition 7 (Termination). *Under any typing assumptions $x_1 : U_1, \ldots,$ $k_1 : S_1, \ldots,$ any query using the inference rules in Fig. 5 terminates.*

Proposition 8. *Under any typing assumptions $x_1 : U_1, \ldots, k_1 : S_1, \ldots$:*

(a) $U_1 \leq U_2$ iff $E : U_1$ entails $E : U_2$.
(b) $E \Rightarrow T$ iff E is a focus and T is a least type such that $E : T$.
(c) $C \cent E \Rightarrow T$ iff T is a least type such that $C \,\$\, E : T$.
(d) $C\langle\rangle \cent E \Rightarrow T$ iff T is a least type such that $C\langle\rangle^\dagger[E] : T$.
(e) $S \Leftarrow C\langle\rangle \cent E : T$ iff S is a greatest cotype such that $C\langle\rangle^\dagger[E] : S \downarrow T$.

For T to be a least type such that $E : T$ means that $E : T$ and, for all T', if $E : T'$ then $T \leq T'$. For $U \uparrow T_1$ to be a greatest cotype such that $E : (U \uparrow T_1) \downarrow T_2$ means that $E : (U \uparrow T_1) \downarrow T_2$ and, for all U' and T_1', if $E : (U' \uparrow T_1') \downarrow T_2$ then $U \leq U'$ and $T_1' \leq T_1$.

Corollary 1 (Correctness). *$\langle\rangle \cent E \Rightarrow T$ iff T is a least type such that $E : T$.*

Given that we annotate binders with types, this corollary shows the least type is unique because the type-checking algorithm is deterministic.

5 Conclusion

We model syntax using a substructural logic, such that terms in the language are structures in the substructural logic. This approach is standard in logical analyses of natural language but less common for programming languages. Types as abstract interpretations fall out, because structures in logic naturally contain formulas, and formulas are types—or abstract values—in the language. Hypothetical reasoning and structural rules in the logic model small-step transitions. Thus our type systems embody small-step abstract interpretation.

Beyond reconstructing the simply-typed λ-calculus, the fruit of our approach is the $\lambda\xi_0$-calculus. It is the first type system for delimited continuations in which control effects may change the answer type as well as act beyond the nearest dynamically-enclosing delimiter. The types are built up from binary connectives, which can clearly be interpreted as implication in intuitionistic logic. This feature is enabled by small-step typing, which lets us assign cotypes to delimited evaluation contexts. We also presented and implemented a type-checking algorithm, which again operates by small-step abstract interpretation.

Modeling syntax using a substructural logic lets us take advantage of established results, such as proof rules and reductions. It further draws attention to the similarity between hypothetical reasoning in the binding context and in the evaluation context—for example, between the ξ_0 and $\downarrow$I rules in Fig. 3. The two kinds of contexts differ simply in that the binding context is intuitionistic whereas the evaluation context is substructural: while the former is usually written to the left of $\vdash$ and admits weakening and contraction, the latter is usually written between $\vdash$ and : and only allows structural rules that model focusing. Typing derivations thus follow evaluation order and control flow: the typing derivation of a direct-style term is essentially its CPS transformation.

Our work exhibits a duality closely related to that for undelimited continuations [15, 29, 55, 66], but investigating the delimited case remains future work. Given our analogy between small-step type-checking and small-step evaluation, we should relate our proof and term normalizations.

It remains to extend this work to other control operators, such as Felleisen's *control* [25, 26] and *named prompts* for delimiters, and to relate it to other substructural logics, such as those with additives and exponentials. We also look forward to mechanizing our proofs.

References

[1] Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proc. 5th intl. conf. principles & practice of declarative prog., pp. 8–19 (2003)
[2] Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Info. Proc. Lett. 90(5), 223–232 (2004)
[3] Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theor. Comp. Sci. 342(1), 149–172 (2005)
[4] Ariola, Z.M., Herbelin, H., Sabry, A.: A type-theoretic foundation of continuations and prompts. In: ICFP, pp. 40–53 (2004)
[5] Atkey, R.: Parameterised notions of computation. In: MSFP 2006, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society, Vancouver (2006)
[6] Balat, V., Danvy, O.: Memoization in type-directed partial evaluation. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 78–92. Springer, Heidelberg (2002)
[7] Balat, V., Di Cosmo, R., Fiore, M.: Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In: POPL, pp. 64–76 (2004)
[8] Barker, C.: Direct compositionality on demand. In: Barker, C., Jacobson, P. (eds.) Direct compositionality, pp. 102–131. Oxford University Press, New York (2007)
[9] Barker, C., Shan, C.-c.: Types as graphs: Continuations in type logical grammar. J. Logic, Lang. & Info. 15(4), 331–370 (2006)
[10] Biernacka, M., Biernacki, D., Danvy, O.: An operational foundation for delimited continuations in the CPS hierarchy. Logical Methods in Comp. Sci. 1(2:5) (2005)

[11] Biernacka, M., Danvy, O.: A syntactic correspondence between context-sensitive calculi and abstract machines. Theor. Comp. Sci. 375(1-3), 76–108 (2007)
[12] Biernacki, D., Danvy, O.: From interpreter to logic engine by defunctionalization. In: Bruynooghe, M. (ed.) LOPSTR 2003. LNCS, vol. 3018, pp. 143–159. Springer, Heidelberg (2004)
[13] Cousot, P.: Types as abstract interpretations. In POPL, pp. 316–331 (1997)
[14] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL, pp. 238–252 (1977)
[15] Curien, P-L., Herbelin, H.: The duality of computation. In: ICFP, pp. 233–243 (2000)
[16] Danvy, O.: Type-directed partial evaluation. In: POPL, pp. 242–257 (1996)
[17] Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU (1989) http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz
[18] Danvy, O.: Abstracting control. In: Proc. conf. Lisp & funct. prog., pp. 151–160 (1990)
[19] Danvy, O.: Representing control: A study of the CPS transformation. Math. Structures Comp. Sci. 2(4), 361–391 (1992)
[20] Danvy, O., Nielsen, L.R.: Syntactic theories in practice. In: van den Brand, M., Verma, R. (eds.) RULE 2001: 2nd intl. workshop on rule-based prog. Electron. Notes in Theor. Comp. Sci, vol. 59(4), pp. 358–374. Elsevier, Amsterdam (2001)
[21] Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Report RS-04-26, BRICS (2004)
[22] Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: POPL, pp. 281–292 (2004)
[23] Dybjer, P., Filinski, A.: Normalization and partial evaluation. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 137–192. Springer, Heidelberg (2002)
[24] Dybvig, R.K., Peyton Jones, S.L., Sabry, A.: A monadic framework for delimited continuations. Tech. Rep. 615, Indiana U (2005)
[25] Felleisen, M.: The calculi of λ_υ-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana U. (1987)
[26] Felleisen, M.: The theory and practice of first-class prompts. In: POPL, pp. 180–190 (1988)
[27] Felleisen, M., Flatt, M.: Programming languages and lambda calculi (2006) http://www.cs.utah.edu/plt/publications/pllc.pdf
[28] Felleisen, M., Friedman, D.P.: Control operators, the SECD machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal description of prog. concepts III, pp. 193–217. Elsevier, Amsterdam (1987)
[29] Filinski, A.: Declarative continuations: An investigation of duality in programming language semantics. In: Pitt, D.H., Rydeheard, D.E., Dybjer, P., Pitts, A.M., Poigné, A. (eds.) Category Theory and Computer Science. LNCS, vol. 389, pp. 224–249. Springer, Heidelberg (1989)
[30] Filinski, A.: Representing monads. In: POPL, pp. 446–457 (1994)
[31] Filinski, A.: Controlling effects. Ph.D. thesis, CMU (1996)
[32] Filinski, A.: Representing layered monads. In: POPL, pp. 175–188 (1999)
[33] Filinski, A.: Normalization by evaluation for the computational lambda-calculus. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 151–165. Springer, Heidelberg (2001)

[34] Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: Proc. conf. Lisp & funct. prog., pp. 28–38 (1986)
[35] Graunke, P.T.: Web interactions. Ph.D. thesis, Northeastern U (2003)
[36] Griffin, T.G.: A formulae-as-types notion of control. In: POPL, pp. 47–58 (1990)
[37] Grobauer, B., Yang, Z.: The second Futamura projection for type-directed partial evaluation. Higher-Order & Symbolic Comp. 14(2–3), 173–219 (2001)
[38] Gunter, C.A., Rémy, D., Riecke, J.G.: A generalization of exceptions and control in ML-like languages. In: Peyton Jones, S.L. (ed.) Funct. prog. lang. & comp. architecture: 7th conf., pp. 12–23 (1995)
[39] Gunter, C.A., Rémy, D., Riecke, J.G.: Return types for functional continuations (1998) http://pauillac.inria.fr/~remy/work/cupto/
[40] Hieb, R., Dybvig, R.K.: Continuations and concurrency. In: Proc. 2nd symposium on principles & practice of parallel prog., pp. 128–136 (1990)
[41] Huet, G.: The zipper. J. Funct. Prog. 7(5), 549–554 (1997)
[42] Kameyama, Y.: Towards logical understanding of delimited continuations. In: Amr, S. (ed.) Proc. 3rd workshop on continuations, pp. 27–33. Tech. Rep. 545, Indiana U (2001)
[43] Kiselyov, O.: How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Tech. Rep. 611, Indiana U (2005)
[44] Kiselyov, O., Shan, C.-c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers (functional pearl). In: ICFP, pp. 192–203 (2005)
[45] Kiselyov, O., Shan, C.-c., Sabry, A.: Delimited dynamic binding. In: ICFP, pp. 26–37 (2006)
[46] Lambek, J.: The mathematics of sentence structure. Amer. Math. Monthly 65(3), 154–170 (1958)
[47] Lawall, J.L., Danvy, O.: Continuation-based partial evaluation. In: Proc. conf. Lisp & funct. prog., pp. 227–238 (1994)
[48] Lucassen, J.M.: Types and effects: Towards the integration of functional and imperative programming. Ph.D. thesis, MIT (1987)
[49] Moortgat, M.: Categorial type logics. In: van Benthem, J.F.A.K., ter Meulen, A.G.B. (eds.) Handbook of logic and language, ch. 2, Elsevier, Amsterdam (1997)
[50] Murphy VII, T., Crary, K., Harper, R.: Distributed control flow with classical modal logic. In: Ong, C.-H.L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 51–69. Springer, Heidelberg (2005)
[51] Murthy, C.R.: Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In: Danvy, O., Talcott, C. (eds.) Proc. workshop on continuations, pp. 49–71 (1992) Tech. Rep. STAN-CS-92-1426, Stanford U.
[52] Polakow, J.: Ordered linear logic and applications. Ph.D. thesis, CMU (2001)
[53] Queinnec, C.: Continuations and web servers. Higher-Order & Symbolic Comp. 17(4), 277–295 (2004)
[54] Restall, G.: An introduction to substructural logics. Routledge, London (2000)
[55] Selinger, P.: Control categories and duality: On the categorical semantics of the lambda-mu calculus. Math. Structures Comp. Sci. 11, 207–260 (2001)
[56] Sewell, P., Leifer, J.J., Wansbrough, K., Nardelli, F.Z., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level programming language design for distributed computation. In: ICFP, pp. 15–26 (2005)
[57] Shan, C.-c.: Shift to control. In: Shivers, O., Waddell, O. (eds.) Proc. Scheme workshop, pp. 99–107, Tech. Rep. 600, Indiana U. (2004)
[58] Shan, C.-c.: Linguistic side effects. Ph.D. thesis, Harvard U. (2005)

[59] Sitaram, D.: Handling control. In: PLDI, pp. 147–155 (1993)
[60] Sumii, E.: An implementation of transparent migration on standard Scheme. In: Felleisen, M. (ed.) Proc. Scheme workshop, pp. 61–63, Tech. Rep. 00-368, Rice U (2000)
[61] Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. J. Funct. Prog. 2(3), 245–271 (1992)
[62] Talpin, J.-P., Jouvelot, P.: The type and effect discipline. Info. & Comp. 111(2), 245–296 (1994)
[63] Thielecke, H.: From control effects to typed continuation passing. In: POPL, pp. 139–149 (2003)
[64] Thiemann, P.: Combinators for program generation. J. Funct. Prog. 9(5), 483–525 (1999)
[65] Wadler, P.L.: The marriage of effects and monads. In: ICFP, pp. 63–74 (1998)
[66] Wadler, P.L.: Call-by-value is dual to call-by-name. In: ICFP (2003)
[67] Wand, M., Friedman, D.P.: The mystery of the tower revealed: A non-reflective description of the reflective tower. Lisp & Symbolic Comp. 1(1), 11–37 (1988)
[68] Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Info. & Comp. 115(1), 38–94 (1994)

The Inhabitation Problem for Rank Two Intersection Types*

Dariusz Kuśmierek

Warsaw University, Institute of Informatics
Banacha 2, 02-097 Warsaw, Poland
daku@mimuw.edu.pl

Abstract. We prove that the inhabitation problem for rank two intersection types is decidable, but (contrary to a common belief) EXPTIME-hard. The exponential time hardness is shown by reduction from the in-place acceptance problem for alternating Turing machines.

Keywords: lambda calculus, intersection types, type inhabitation problem, alternating Turing machine.

Introduction

Type inhabitation problem is usually defined as follows: "does there exist a closed term T of a given type τ (in an empty environment)".

In the simply typed system the inhabitation problem is PSPACE-complete (see [7]). The intersection types system studied in the current paper involves types of the form $\alpha \cap \beta$. Intuitively, a term can be assigned the type $\alpha \cap \beta$ if and only if it can be assigned the type α and also the type β.

Undecidability of the general inhabitation problem for intersection types was shown by P. Urzyczyn in [8].

Several weakened systems were studied, and proved to be decidable. T. Kurata and M. Takahashi in [2] proved the decidability of the problem in the system $\lambda(E\cap, \leq)$ which does not use the rule $(I\cap)$.

D. Leivant in [3] defines the rank of an intersection type. The notion of rank provides means for classification and a measure of complexity of the intersection types.

One can notice, that the construction in [8] uses types of rank four. The decidability of the inhabitation for rank three is still an open problem. The problem for rank two was so far believed to be decidable in polynomial space (see [8]).

Our result contradicts this belief. We prove the inhabitation problem for rank two to be EXPTIME-hard by a reduction from the halting problem for Alternating Linear Bounded Automata (ALBA in short). The idea of the reduction is as follows. For a given ALBA and a given word of length n we construct a type of the form $\alpha_1 \cap \ldots \cap \alpha_n \cap \alpha_{n+1} \cap \alpha_{n+2}$. For $i = 1 \ldots n$, the component α_i represents the behaviour of the i-th cell of the tape, α_{n+1} represents changes in the position of the head of the machine, and the last part α_{n+2} simulates changes of

* Partly suported by the Polish government grant 3 T11C 002 27.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 240–254, 2007.
© Springer-Verlag Berlin Heidelberg 2007

the machine state. The $\cap$ operator is used here to hold and process information about the whole configuration of the automaton.

The fact that the problem for rank two is EXPTIME-hard only demonstrates how difficult the still open problem for rank three may be.

1 Basics

We consider a lambda calculus with types defined by the following induction:

- Type variables are types;
- If α and β are types, then $\alpha \to \beta$ and $\alpha \cap \beta$ are also types.

We assume that the operator $\cap$ is associative, commutative and idempotent. The type inference rules for our system are as follows:

(VAR) $\quad \Gamma \vdash x{:}\,\sigma \quad$ if $(x{:}\,\sigma) \in \Gamma$

$$(E\to)\ \frac{\Gamma \vdash M{:}\,\alpha \to \beta \quad \Gamma \vdash N{:}\,\alpha}{\Gamma \vdash (MN){:}\,\beta} \qquad (I\to)\ \frac{\Gamma, (x{:}\,\alpha) \vdash M{:}\,\beta}{\Gamma \vdash \lambda x.M{:}\,\alpha \to \beta}$$

$$\frac{\Gamma \vdash M{:}\,\alpha \cap \beta}{\Gamma \vdash M{:}\,\alpha}\ (E\cap)\ \frac{\Gamma \vdash M{:}\,\alpha \cap \beta}{\Gamma \vdash M{:}\,\beta} \qquad (I\cap)\ \frac{\Gamma \vdash M{:}\,\alpha \quad \Gamma \vdash M{:}\,\beta}{\Gamma \vdash M{:}\,\alpha \cap \beta}$$

Definition 1. Following Leivant ([3]) we define the *rank* of a type τ:

$rank(\tau) = 0$, if τ is a simple type (without "$\cap$");
$rank(\tau \cap \sigma) = max(1, rank(\tau), rank(\sigma))$;
$rank(\tau \to \sigma) = max(1 + rank(\tau), rank(\sigma))$, when $rank(\tau) > 0$ or $rank(\sigma) > 0$.

2 Decidability of the Inhabitation Problem

2.1 The Algorithm

Definition 2. An *environment* Γ is a set of *declarations* of the form $(x{:}\,\alpha)$, where x is a variable and α is a type.

Definition 3. A variable x is *k-ary* in an environment Γ, if Γ includes a declaration $(x{:}\,\alpha)$, such that

$$\alpha = \beta_1 \to \cdots \to \beta_k \to \beta_{k+1}$$
or
$$\alpha = \gamma \cap (\beta_1 \to \cdots \to \beta_k \to \beta_{k+1}).$$

In other words, the variable is k-ary, if one can apply it to some k arguments.

Definition 4. A *constraint* is a pair (Γ, τ), denoted by $\Gamma \vdash X{:}\,\tau$, where Γ is an environment and τ is a type.

Definition 5. A *task* is a set of constraints

$$Z = [\Gamma_1 \vdash X{:}\,\tau_1,\ \ldots,\ \Gamma_n \vdash X{:}\,\tau_n],$$

where all the environments $\Gamma_1 \ldots \Gamma_n$ share the same domain of variables, and where the types $\tau_1 \ldots \tau_n$ are not intersections (meaning that $\tau_i \neq \alpha_i \cap \beta_i$).
A *solution* of the task Z is a term M such that for each $i = 1 \ldots n$ we have $\Gamma_i \vdash M{:}\,\tau_1$.

To ilustrate our algorithm, we first consider an example. Let us find an inhabitant in a β-normal form for a type $T = \tau_1 \cap \tau_2$, where

$$\tau_1 = ((((\alpha \to B) \to A) \cap ((\beta \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D)) \\ \to (\alpha \to \beta \to \alpha \to \beta \to E) \to A),$$

$$\tau_2 = ((((\beta \to B) \to A) \cap ((\alpha \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D)) \\ \to (\beta \to \alpha \to \alpha \to \beta \to E) \to A).$$

We have to find a term M which can be assigned both the type τ_1 and τ_2. The following must hold: $\emptyset \vdash M{:}\,\tau_1$; $\emptyset \vdash M{:}\,\tau_2$. From the structure of τ_1 and τ_2 we can see that M has to be an abstraction $M = \lambda XY.N$. Now we will try to find N such that: $\Gamma_1 \vdash N{:}\,A$; $\Gamma_2 \vdash N{:}\,A$, where

$$\Gamma_1 = \{X{:}\,(((\alpha \to B) \to A) \cap ((\beta \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D), \\ Y{:}\,\alpha \to \beta \to \alpha \to \beta \to E\},$$

$$\Gamma_2 = \{X{:}\,(((\beta \to B) \to A) \cap ((\alpha \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D), \\ Y{:}\,\beta \to \alpha \to \alpha \to \beta \to E\}.$$

We notice now, that N has to be X applied to some argument (of different types in different environments). We have $N = X(\lambda x.P)$. And we search now for P such that: $\Gamma_1, x{:}\,\alpha \vdash P{:}\,B$; $\Gamma_2, x{:}\,\beta \vdash P{:}\,B$. Then again $P = X(\lambda y.Q)$, where $\Gamma_1, x{:}\,\alpha, y{:}\,\beta \vdash Q{:}\,C$; $\Gamma_2, x{:}\,\beta, y{:}\,\alpha \vdash Q{:}\,C$. We continue with $Q = X(\lambda v.R)$, and $\Gamma_1, x{:}\,\alpha, y{:}\,\beta, v{:}\,\alpha \vdash R{:}\,D$; $\Gamma_2, x{:}\,\beta, y{:}\,\alpha, v{:}\,\alpha \vdash R{:}\,D$. Finally $R = X(\lambda z.S)$ and $\Gamma_1, x{:}\,\alpha, y{:}\,\beta, v{:}\,\alpha, z{:}\,\beta \vdash S{:}\,E$; $\Gamma_2, x{:}\,\beta, y{:}\,\alpha, v{:}\,\alpha, z{:}\,\beta \vdash S{:}\,E$. Now we see that $S = YS_1S_2S_3S_4$, and we have to solve four tasks:

$$\begin{array}{ll} \Gamma_1, x{:}\,\alpha, y{:}\,\beta, v{:}\,\alpha, z{:}\,\beta \vdash S_1{:}\,\alpha; & \Gamma_2, x{:}\,\beta, y{:}\,\alpha, v{:}\,\alpha, z{:}\,\beta \vdash S_1{:}\,\beta, \\ \Gamma_1, x{:}\,\alpha, y{:}\,\beta, v{:}\,\alpha, z{:}\,\beta \vdash S_2{:}\,\beta; & \Gamma_2, x{:}\,\beta, y{:}\,\alpha, v{:}\,\alpha, z{:}\,\beta \vdash S_2{:}\,\alpha, \\ \Gamma_1, x{:}\,\alpha, y{:}\,\beta, v{:}\,\alpha, z{:}\,\beta \vdash S_3{:}\,\alpha; & \Gamma_2, x{:}\,\beta, y{:}\,\alpha, v{:}\,\alpha, z{:}\,\beta \vdash S_3{:}\,\alpha, \\ \Gamma_1, x{:}\,\alpha, y{:}\,\beta, v{:}\,\alpha, z{:}\,\beta \vdash S_4{:}\,\beta; & \Gamma_2, x{:}\,\beta, y{:}\,\alpha, v{:}\,\alpha, z{:}\,\beta \vdash S_4{:}\,\beta. \end{array}$$

We see that the only possible assignment is $S_1 = x, S_2 = y, S_3 = v, S_4 = z$. Hence we found our inhabitant $M = \lambda XY.X(\lambda x.X(\lambda y.X(\lambda v.X(\lambda z.Yxyvz))))$.

Notice that, when searching for the inhabitants, we need always to consider simulteanously both environments. For each S_i we had always exactly two variables of the right type in each environment (because in every environment there

are two variables of the type α and two of the type β). But only one variable had the right type in both environments. While constructing an inhabitant for a type $\tau_1 \cap \tau_2$ we had to build a common solution for both parts.

We start the decidability proof by presenting the nondeterministic algorithm. This algorithm can be easily converted into a deterministic one, but at a considerable loss of clarity. Also we remind reader that the exact procedure described below does not have the termination property. Hovewer we prove later on (see Section 2.3), that for each input type of rank at most two, our algorithm (after a few simple modifications) must find a solution in a bounded number of steps or repeat a configuration.

Definition 6. *The Algorithm.*

Our algorithm uses the "intersection removal" operation *Rem* defined as follows:

$Rem(\Gamma \vdash X\!:\tau) = \{\Gamma \vdash X\!:\tau\}$ if τ is either a type variable or $\tau = \tau_1 \to \tau_2$;
$Rem(\Gamma \vdash X\!:\tau_1 \cap \tau_2) = Rem(\Gamma \vdash X\!:\tau_1) \cup Rem(\Gamma \vdash X\!:\tau_2)$.

The purpose of the *Rem* operation is to eliminate "$\cap$" and to convert a judgement $\Gamma \vdash X\!:\tau$ with τ being possibly an intersection type into a set of judgements where the types on the right side are not intersections.

For a given type τ the first task is:

$$Z_0 = Rem(\emptyset \vdash X\!:\tau)$$

Recall that the types on the right-hand sides of tasks are not intersections. Let the current task be:

$$Z = [\Gamma_1 \vdash X\!:\tau_1,\ \ldots,\ \Gamma_n \vdash X\!:\tau_n]$$

1. If each type τ_i is of the form $\alpha_i \to \beta_i$, then the next task processed recursively by the algorithm will be:

   $$Z' = Rem(\Gamma_1 \cup \{x\!:\alpha_1\} \vdash X'\!:\beta_1) \cup \ldots \cup Rem(\Gamma_n \cup \{x\!:\alpha_n\} \vdash X'\!:\beta_n),$$

   where x is a fresh variable not used in any of the Γ_i.
   If the recursive call for Z' returns M', then $M = \lambda x.M'$, if on the other hand the recursive call gives an answer "empty type", we shall give the same answer.
2. If at least one of the τ_i is a type variable, then the solution cannot be an abstraction, but must be an application or a variable. Note that all environments $\Gamma_1, \ldots, \Gamma_n$ have the same domain of variables. Suppose that there exists a number k and a variable x which is k-ary in each of the environments, and for all $j = 1 \ldots n$ we have that:

   $$\Gamma_j \vdash x\!:\beta_{j1} \to \cdots \to \beta_{jk} \to \tau_j$$

   (if there is more than one such a pair, we pick nondeterministically one of them). Then:

- If $k = 0$, then $M = x$,
- If $k > 0$, then $M = xM_1 \dots M_k$, where M_i are solutions of the k independent tasks:

$$Z_1 = Rem(\Gamma_1 \vdash X_1\colon \beta_{11}) \cup \ldots \cup Rem(\Gamma_n \vdash X_1\colon \beta_{n1}),$$
$$\ldots$$
$$Z_k = Rem(\Gamma_1 \vdash X_k\colon \beta_{1k}) \cup \ldots \cup Rem(\Gamma_n \vdash X_k\colon \beta_{nk}).$$

  If any of the k recursive calls gives the answer "empty type", we shall give the same answer.

If there is no such a number and a variable we give the answer "empty type".

2.2 Soundness and Completeness

Lemma 7. *If the above algorithm finds a term M for an input type τ, then $\vdash M\colon \tau$.*

Proof. By induction on M.

The algorithm proposed in Definition 6 is not capable of finding all the terms of a given type. Hence, to prove the correctness of the proposed procedure, we first define the notion of a *long* solution, then we show that every task, which has a solution, has also a long solution. Finally we complete the proof of the soundness of the algorithm by proving that every long solution can be found by the given procedure.

Definition 8. M is a *long* solution of the task $Z = [\Gamma_1 \vdash X\colon \tau_1, \ldots, \Gamma_n \vdash X\colon \tau_n]$, when one of the following holds:

- All types τ_i are of the form $\alpha_i \to \beta_i$ and $M = \lambda x.M'$, where M' is a long solution of the task
  $Z' = Rem(\Gamma_1 \cup \{x\colon \alpha_1\} \vdash X'\colon \beta_1) \cup \ldots \cup Rem(\Gamma_n \cup \{x\colon \alpha_n\} \vdash X'\colon \beta_n)$, or
- Some τ_i is a type variable and $M = xM_1 \dots M_k$ (possibly with $k = 0$), where for $i = 1 \dots n$ we have $\Gamma_i \vdash x\colon \alpha_{i1} \to \cdots \to \alpha_{ik} \to \tau_i$ and $M_1, \ldots, M_k$ are long solutions of tasks $Z_1, \ldots, Z_k$, where $Z_j = [\Gamma_1 \vdash X_j\colon \alpha_{1j}, \ldots, \Gamma_n \vdash X_j\colon \alpha_{nj}]$ for $j = 1 \dots k$.

Lemma 9. *If there exists a solution of a task Z, then there exists a long one.*

Proof. Assume that M is a solution of $Z = [\Gamma_1 \vdash X\colon \tau_1, \ldots, \Gamma_n \vdash X\colon \tau_n]$ and that M is in a normal form. We construct a long solution $A(M, Z)$ in the following way:

- If there is a τ_i which is a type variable, then M is not an abstraction and:
  - If $M = x$, then $A(M, Z) = M$, because in this case M is a long solution,
  - If $M = xM_1 \dots M_k$, then it must hold that $\Gamma_i \vdash x\colon \alpha_{i1} \to \cdots \to \alpha_{ik} \to \tau_i$ for $i = 1 \dots n$, so $A(M, Z) = xA(M_1, Z_1) \dots A(M_k, Z_k)$, where $Z_j = [\Gamma_1 \vdash X_j\colon \alpha_{1j}, \ldots, \Gamma_n \vdash X_j\colon \alpha_{nj}]$ for $j = 1 \dots k$.

– Otherwise (if all τ_i have the form of $\alpha_i \to \beta_i$):
  - If $M = xM_1 \dots M_k$ (possibly for $k = 0$) then again it must hold that $\Gamma_i \vdash x{:}\, \alpha_{i1} \to \dots \to \alpha_{ik} \to \tau_i$ for $i = 1 \dots n$, and also for $i = 1 \dots n$, $j = 1 \dots k$ it must hold that $\Gamma_i \vdash M_j{:}\, \alpha_{ij}$. Let r be the largest number, such that all the types τ_i are of the form $\beta_{1i} \to \dots \to \beta_{ri} \to \gamma_i$. Then $A(M, Z) = \lambda z_1 \dots z_r.xA(M_1, Z_1) \dots A(M_k, Z_k)A(z_1, Z'_1) \dots A(z_r, Z'_r)$, where

$$Z_j = [\Gamma_1, (z_1{:}\, \beta_{1j}), \dots, (z_r{:}\, \beta_{rj}) \vdash X_j{:}\, \alpha_{1j}, \dots, \Gamma_n, (z_1{:}\, \beta_{nj}), \dots, (z_r{:}\, \beta_{nj}) \vdash X_j{:}\, \alpha_{nj}],$$
$$Z'_j = [\Gamma_1, (z_1{:}\, \beta_{1j}), \dots, (z_r{:}\, \beta_{rj}) \vdash X'_j{:}\, \beta_{1j}, \dots, \Gamma_n, (z_1{:}\, \beta_{nj}), \dots, (z_r{:}\, \beta_{nj}) \vdash X'_j{:}\, \beta_{nj}].$$

  - If $M = \lambda x.M'$, then $A(M, Z) = \lambda x.A(M', Z')$, where $Z' = Rem([\Gamma_1, (x{:}\, \alpha_1) \vdash X'{:}\, \beta_1, \dots, \Gamma_n, (x{:}\, \alpha_n) \vdash X'{:}\, \beta_n])$.

Lemma 10. *Every long solution of the task $Z = [\Gamma_1 \vdash X{:}\, \tau_1, \dots, \Gamma_n \vdash X{:}\, \tau_n]$ can be found by the nondeterministic procedure described in Definition 6 in one of its possible runs.*

Proof. By induction on the structure of the long solution M.

– $M = x$. Since M is long, at least one of the τ_i must be a type variable. Hence the algorithm working on the task Z will search in the environments Γ_i for a variable of the right type (case 2 of the algorithm). One of these variables is x.
– $M = xM_1 \dots M_k$. Like before we can reason that the algorithm shall choose the case 2, and in one of its possible runs the algorithm will choose the variable x. After x is chosen, the procedure shall search for solutions of the tasks $Z_1, \dots, Z_k$. By the definition of a long solution, we have that $M_1, \dots, M_k$ are long solutions of the tasks $Z_1, \dots, Z_k$, and by the induction hypothesis, these solutions can be found by the recursive runs of our procedure. It follows that also M can be found.
– $M = \lambda x.M'$. Then of course all the types τ_i have to be of the form $\alpha_i \to \beta_i$. Hence for the task Z the procedure will choose case 1, and will search for a solution of the task $Z' = Rem([\Gamma_1, (x{:}\, \alpha_1) \vdash X'{:}\, \beta_1, \dots, \Gamma_n, (x{:}\, \alpha_n) \vdash X' : \beta_n])$. By the induction hypothesis, the long solution M' for Z' can be found by the algorithm. Hence also M can be found.

Corollary 11. *Our algorithm finds an inhabitant for every non-empty type, for which it terminates.*

Proof. A direct conclusion of Lemmas 9 and 10.

2.3 The Termination of the Algorithm

Let us consider the work of the algorithm for a type τ of rank two.

Fact 12. *Types of variables put in the environments during the work of the algorithm are of rank at most one.*

Proof. The environments are modified only in case 1. Since the type τ is of a rank at most two, the variables put in the environments have rank at most one.

Fact 13. *In every recursive run each task has at most $|\tau|$ constraints to solve.*

Proof. The number of the constraints increases only when the *Rem* operation is used. Let us denote the number of "$\cap$" operators in the type τ by $C(\tau)$. It is easy to notice that if $Rem(\Gamma \vdash X : \tau) = \{\Gamma \vdash X : \tau_1, \ldots, \Gamma \vdash X : \tau_k\}$, then the following holds:

$$C(\tau) = C(\tau_1) + \ldots + C(\tau_k) + k.$$

In other words creating new constraint always removes one "$\cap$". The recursive calls in case 2 do not increase the number of constraints. In these problems the procedure searches for terms which can serve as arguments for a variable taken from the environment. As stated in Fact 12, in environments there are only variables with types of rank zero and one, and such variables can only be given arguments with types of rank zero. And these types are simple (without intersections), so the *Rem* operation applied to them will not increase the number of constraints in a task.

The Decidability

Theorem 14 *The inhabitation problem for the types of rank two is decidable.*

Proof. The algorithm proposed in Definition 6 can be easily modified in a way which will prevent the environments from growing bigger infinitely during the work of the algorithm. Variables are added to the environments only when all currently examined types τ_i are of the form $\alpha_i \to \beta_i$. Then every environment Γ_i is expanded by a new variable of the type α_i. Note that there are only $O(|\tau|)$ types that can be assigned to a variable in one environment. Since we do not need to keep several variables of the same type (meaning of the same type in each of the environments) it follows that there is a bounded number of possible distinct environments that may occur during the work of the algorithm. Also the number of the types that may occur on the right hand side of each $\vdash$ is $O(|\tau|)$. Hence each branch of the procedure must finish or repeat a configuration after a bounded (although possibly exponential) number of steps.

3 The Lower Bound

3.1 Terms of Exponential Size

First we shall consider an instructive example. We propose a schema for creating instances of the inhabitation problem for which the above algorithm has to perform an exponential number of steps before finding the only inhabitant. The size of the inhabitant will also be exponential in the size of the type. Our example demonstrates a technique used in the construction to follow. Let $T(n) = \tau_1 \cap \ldots \cap \tau_n$, where

$$\tau_i = \alpha \to \underbrace{\Psi \to \cdots \to \Psi}_{i-1} \to (\alpha \to \beta) \to \underbrace{(\beta \to \alpha) \cdots \to (\beta \to \alpha)}_{n-i} \to \beta, \text{ and}$$
$$\Psi = (\alpha \to \alpha) \cap (\beta \to \beta).$$

For instance $T(3) =$

$$\begin{array}{l} (\alpha \to (\alpha \to \beta) \to (\beta \to \alpha) \to (\beta \to \alpha) \to \beta) \cap \\ (\alpha \to \quad \Psi \quad \to (\alpha \to \beta) \to (\beta \to \alpha) \to \beta) \cap \\ (\alpha \to \quad \Psi \quad \to \quad \Psi \quad \to (\alpha \to \beta) \to \beta) \end{array}$$

One can notice that a construction of an inhabitant for this type is similar to the rewriting from the word $\beta\beta\beta$ to the word $\alpha\alpha\alpha$ in the following way:

$$\beta\beta\beta \to \alpha\beta\beta \to \beta\alpha\beta \to \alpha\alpha\beta \to \beta\beta\alpha \to \alpha\beta\alpha \to \beta\alpha\alpha \to \alpha\alpha\alpha.$$

For $|T(n)| = O(n^2)$, there is only one (modulo α-equivalence) term t of type $T(n)$, and $|t| = O(2^n)$. For instance, the only term of type $T(3)$ is:

$$\lambda x_1 x_2 x_3 x_4 . x_2(x_3(x_2(x_4(x_2(x_3(x_2 x_1)))))),$$

while for $T(4)$ it is:

$$\lambda x_1 x_2 x_3 x_4 x_5 . x_2(x_3(x_2(x_4(x_2(x_3(x_2(x_5(x_2(x_3(x_2(x_4(x_2(x_3(x_2 x_1)))))))))))))).$$

In what follows, while proving EXPTIME-hardness of the inhabitation problem, we shall generate types of a similar form to $T(n)$. For this reason it is convenient to use $T(n)$ for introducing notions and notations, which we shall use later on. Because of the different role played by the "$\cap$" and "$\to$" it is convenient to consider the type as an object composed of columns and rows. The rows are connected with "$\cap$", and columns with "$\to$". According to this terminology $T(3)$ has three rows and five columns. We can think of α and β as of states of a certain object (e.g. a tape cell). Each row represents operations available for a given object and the initial and final state of the object. Each column represents a certain operation (that is a step of a certain automaton) which can modify the state of all objects. In type $T(3)$ there are three available operations. The i-th operation changes the i-th sign from β to α, and all earlier signs from α to β. More precisely each $(\alpha \to \beta)$ in the type $T(n)$ represents the change from β to α, and an occurence of Ψ represents no change of sign (changes α and β to themselves).

3.2 EXPTIME-Hardness

We shall show the lower bound for the complexity of the inhabitation problem by a reduction from the EXPTIME-complete problem of the in-place acceptance for alternating Turing machines.

We consider the *Alternating Linear Bound Automata* (ALBA) which are just alternating Turing machines which never leave the input word.

Definition 15. *An Alternating Linear Bound Automaton* is a sixtuple:

$$M = (Q, \Gamma, \delta, q_0, q_{acc}, g), \text{ where}$$

- Q is a non-empty, finite set of states;
- Γ is a non-empty, finite set of symbols. We shall assume that $\Gamma = \{0, 1\}$;
- $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$ - is a non-empty, finite transition relation;
- $q_0 \in Q$ is the initial state;
- $q_{acc} \in Q$ is the final state;
- $g : Q \to \{\wedge, \vee\}$ is a function which assigns a kind to every state.

Definition 16. A *configuration* of an ALBA is a triple:

$$C = (q, t, n), \text{ where}$$

- $q \in Q$ is a state;
- $t \in \Gamma^*$ is a tape content;
- $n \in N$ is a position of the head.

Definition 17. We shall say that a *transition* $p = ((q_1, s_1), (q_2, s_2, k)) \in \delta$ is *available* in a configuration $C_1 = (q_1, t, n)$, when the following conditions hold:

- $t(n) = s_1$;
- $(k = L$ and $n > 1)$ or $(k = R$ and $n < |t|)$.

In this case, we shall say that p *transforms* the configuration C_1 into the configuration $C_2 = (q_2, t_2, n_2)$, such that

- $t_2(m) = \begin{cases} s_2 & \text{if } m = n, \\ t(m) & \text{otherwise.} \end{cases}$
- $n_2 = \begin{cases} n + 1 \text{ if } k = R, \\ n - 1 \text{ otherwise.} \end{cases}$

It is worth noting that, in configurations in which the head scans the first symbol of the tape, the only available transitions are these which move the head to the right, and when the head reaches the end of the word, the only active transitions will move it to the left.

Definition 18. An ALBA *accepts* a configuration $C = (q, t, n)$, if

- $q = q_{acc}$ and $|t| = n$, or
- $g(q) = \vee$ and there exists a transition available in C, which transforms C into a configuration which the automaton accepts, or
- $g(q) = \wedge$ and every transition available in C, transforms C into a configuration which the automaton accepts.

Definition 19. *The problem of in-place acceptance* is defined as follows: does a given ALBA accept a given word t (meaning it accepts the configuration $C_0 = (q_0, t, 1)$).

It is known that APSPACE = EXPTIME (see Corollary 2 to Theorem 16.5 and Corollary 3 to Theorem 20.2 in [6]).

Lemma 20. *The problem of in-place acceptance for ALBA is* EXPTIME-complete (APSPACE-complete).

Proof. A simple modification of the proof of Theorem 19.9 in [6]. First we note that the in-place acceptance is in APSPACE. Consider a machine $M = (Q, \Gamma, \delta, q_0, q_{acc}, g)$. Keeping the counter of steps, we simulate the run of M on the input word t. We reject if M rejects, or if the machine makes $|t||Q||\Gamma|^{|t|} + 1$ steps, because after so many steps the machine has to repeat a configuration.

Let L be a language in APSPACE accepted in space n^k by a machine M. It means that M does not use in any of its parallel computations more than n^k cells of the tape (where n is the length of the input word). Let $\perp$ denote the blank symbol. Let us consider a modified machine M', which during its work performs the same moves as M, but when M reaches a final state, the head of M' makes $n^k - n$ steps to the right and also enters a final state. It is clear that M accepts t if and only if the machine M' accepts $t\perp^{n^k-n}$ without ever leaving this word (note that according to the definition, the machine M accepts in the final state only with the head at the rightmost symbol of the input word). Blank symbols $\perp$ at the very end of the word do not change the behaviour of the machine, and M does not use more than n^k cells of the tape. So t belongs to L if and only if M' accepts $t\perp^{n^k-n}$ in-place.

Definition 21. *The construction of the type.*

Let us consider the input word $t = t_1 t_2 \ldots t_{n-1} t_n$. We construct a type with $n+2$ rows and some number of columns (according to the terminology introduced in Section 3.1). The first n rows shall represent the contents of n tape cells. The second last row shall represent the position of the head (values $1 \ldots n$). The last row shall stand for the state of the machine.

Initial and Final State: We shall begin our construction with these two columns:

$$
\begin{array}{l}
(\ \ldots \to 0 \cap 1 \to t_1\ )\cap \\
\qquad\qquad \ldots \\
(\ \ldots \to 0 \cap 1 \to t_n\ )\cap \\
(\ \ldots \to \quad n \quad \to\ 1\ )\cap \\
(\ \ldots \to\ q_{acc}\ \to q_0\ ).
\end{array}
$$

The last column in the type represents the initial configuration: the variables $t_1 \ldots t_n$ represent the symbols of the input word, the head is at first position, and the machine is in state q_0. The second last column represents the final configuration: the head of the machine is on the last cell of the tape. The content of the tape is irrelevant. Each column of the type (except the last one) will be assigned to one variable in a term. The components of a column are different types that are assigned to the same variable in $n+2$ different environments.

In the further construction we shall add new columns on the left side.

States of Kind $\vee$: Let $Id(p) = (\overbrace{0 \to \cdots \to 0}^{p+1}) \cap (\overbrace{1 \to \cdots \to 1}^{p+1})$. For each element $((q_1, s_1), (q_2, s_2, k))$ of δ, such that $g(q_1) = \vee$, we add $n-1$ columns — one column for each position of the head.

If $k = L$, then the i-th added column is of the form:

$$\left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} i$$
$$(s_2 \rightarrow s_1) \rightarrow$$
$$\left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} n - i - 1$$
$$(i - 1 \rightarrow i) \rightarrow$$
$$(q_2 \rightarrow q_1) \rightarrow$$

And if $k = R$, then the i-th added column is:

$$\left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} i - 1$$
$$(s_2 \rightarrow s_1) \rightarrow$$
$$\left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} n - i$$
$$(i + 1 \rightarrow i) \rightarrow$$
$$(q_2 \rightarrow q_1) \rightarrow$$

States of Kind $\wedge$: For each state q, such that $g(q) = \wedge$, and for each sign $s \in \Gamma$ we add n columns (one for each position of the head). The i-th column is generated this way: let $((q, s), (q_1, s_1, k_1)), \dots, ((q, s), (q_p, s_p, k_p))$ be all transitions available in q, when head is at i-th position, which holds sign s. In this case, the i-th column has the form of:

$$\left.\begin{array}{l} Id(p) \rightarrow \\ \dots \\ Id(p) \rightarrow \end{array}\right\} i - 1$$
$$(s_1 \rightarrow \dots \rightarrow s_p \rightarrow s) \rightarrow$$
$$\left.\begin{array}{l} Id(p) \rightarrow \\ \dots \\ Id(p) \rightarrow \end{array}\right\} n - i$$
$$((i + r(k_1)) \rightarrow \dots \rightarrow (i + r(k_p)) \rightarrow i) \rightarrow$$
$$(q_1 \rightarrow \dots \rightarrow q_p \rightarrow q) \rightarrow$$

where

$$r(k) = \begin{cases} 1 & \text{if } k = R \\ -1 & \text{otherwise} \end{cases}$$

The above construction corresponds to the definition of the acceptance in a state of kind $\wedge$, when the automaton needs to accept in all the reachable configurations. The variable corresponding to the added column can be used in a term

(inhabitant) only when it is possible to find inhabitants for each of the arguments. Each such inhabitant represents a computation in one of the possible next configurations.

Note that, if there is no reachable configuration from a state of the kind $\wedge$, then the added column will not be of a functional type (it will not have any arrows except for the one on the right), and so it will not require any further searching for inhabitants. The computation will terminate successfully, which corresponds to the acceptation of a word in states of kind $\wedge$, from which the machine has nowhere to go.

3.3 Correctness of the Reduction

We shall consider the instances of the type inhabitation problem generated by the above construction. Notice that, for such types, the construction of the inhabitant according to the algorithm proposed in Definition 6 will go as follows: first the task shall be split into $n+2$ constraints by use of the *Rem* operator, then the algorithm will use case 1 several times, after which the current task will be

$$Z_0 = [\Gamma_1 \vdash X\colon t_1, \ldots, \Gamma_n \vdash X\colon t_n, \Gamma_{n+1} \ \vdash X\colon 1, \Gamma_{n+2} \vdash X\colon q_0]$$

From this moment the algorithm will use only the application case (case 2), since the types under consideration will always be type variables. In the following steps the only thing that will change will be $s_1, \ldots, s_n, k, q$, but the environments $\Gamma_1, \ldots, \Gamma_n$ will stay the same.

Lemma 22. *Let $s_1, \ldots, s_n \in \Gamma$, $1 \le k \le n$ and $q \in Q$.*
The task $Z = [\Gamma_1 \vdash X\colon s_1, \ldots, \Gamma_n \vdash X\colon s_n, \Gamma_{n+1} \vdash X\colon k, \Gamma_{n+2} \vdash X\colon q]$ has a solution if and only if M accepts the configuration $C = (q, s_1 \ldots s_n, k)$.

Proof
($\Rightarrow$) Induction with respect to the structure of the solution T of the task Z.

- T is a variable x. Then $\Gamma_{n+2} \vdash x\colon q$, and either q is the final state, and $k = n$ (see 3.2) or q is of the kind "$\wedge$", and in the configuration C there are no available transitions (because only in this case there was a variable of a type q added to the environment Γ_{n+2} (see 3.2)). In both cases M accepts the configuration C.
- T is an abstraction. Impossible, because the types, for which we seek an inhabitant in Z are type variables.
- T is an application. There are two possibilities.
  - $T = xT_1$ and $g(q) = \vee$. According to the type construction (see 3.2) it holds that:

$$\begin{gathered}\Gamma_1 \vdash x\colon s_1 \to s_1,\\ \ldots\\ \Gamma_k \vdash x\colon s'_k \to s_k,\\ \ldots\\ \Gamma_n \vdash x\colon s_n \to s_n,\\ \Gamma_{n+1} \vdash x\colon k + r(c) \to k,\\ \Gamma_{n+2} \vdash x\colon q' \to q,\end{gathered}$$

and $((q, s_k), (q', s'_k, c)) \in \delta$. Hence T_1 is a solution of the task

$$[\Gamma_1 \vdash X_1 : s_1, \ldots, \Gamma_k \vdash X_1 : s'_k, \ldots, \Gamma_n \vdash X_1 : s_n, \\ \Gamma_{n+1} \vdash X_1 : k + r(c), \Gamma_{n+2} \vdash X_1 : q'].$$

By the induction hypothesis (for T_1) the machine M accepts $C_1 = (q', s_1 \ldots s'_k \ldots s_n, k + r(c))$. However, since q is of the kind $\vee$ and there exists a transition from C to C_1, it follows that M accepts also C.

- $T = xT_1 \ldots T_m$, for some m and $g(q) = \wedge$. According to the type construction (see 3.2) it must hold that:

$$\begin{gathered} \Gamma_1 \vdash x : s_1 \to \cdots \to s_1 \to s_1, \\ \ldots \\ \Gamma_k \vdash x : s_{k1} \to \cdots \to s_{km} \to s_k, \\ \ldots \\ \Gamma_n \vdash x : s_n \to \cdots \to s_n \to s_n, \\ \Gamma_{n+1} \vdash x : k + r(c_1) \to \cdots \to k + r(c_m) \to k, \\ \Gamma_{n+2} \vdash x : q_1 \to \cdots \to q_m \to q, \end{gathered}$$

and the following transitions are all the transitions available in C: $((q, s_k), (q_1, s_{k1}, c_1)), \ldots, ((q, s_k), (q_m, s_{km}, c_m))$. Then of course each T_i is a solution of the task:

$$[\Gamma_1 \vdash X_i : s_1, \ldots, \Gamma_k \vdash X_i : s_{ki}, \ldots, \Gamma_n \vdash X_i : s_n, \\ \Gamma_{n+1} \vdash X_i : k + r(c_i), \Gamma_{n+2} \vdash X_i : q_i].$$

By the induction hypothesis for $T_1, \ldots, T_m$, the machine M accepts all the configurations $C_1, \ldots, C_m$, where $C_i = (q_i, s_1 \ldots s_{ki} \ldots s_n, k + r(c_i))$. It means that M accepts all configurations reachable from C in one step, so it accepts C.

($\Leftarrow$) Induction with respect to the definition of acceptance.

(*Base*) Let $q = q_{acc}$. Then $k = n$, because the machine accepts only with the head at the rightmost position. Then according to the construction for the final state (see 3.2), there exists a variable x, such that:
$\Gamma_1 \vdash x : s_1, \ldots, \Gamma_n \vdash x : s_n, \Gamma_{n+1} \vdash x : n, \Gamma_{n+2} \vdash x : q_{acc}$. So $T = x$ is a solution of Z.

(*Step*) Assume that M accepts $C = (q, s_1 \ldots s_n, k)$, where q is not the final state. There are two possibilities:

– Let $g(q) = \vee$. Since M accepts the configuration C it means that there exists a transition $((q, s_k), (q', s'_k, c))$, such that M accepts configuration $C_1 = (q', s_1 \ldots s'_k \ldots s_n, k + r(c))$. According to the induction hypothesis there exists a solution T_1 of the task

$$[\Gamma_1 \vdash X_1 : s_1, \ldots, \Gamma_k \vdash X_1 : s'_k, \ldots, \Gamma_n \vdash X_1 : s_n, \\ \Gamma_{n+1} \vdash X_1 : k + r(c), \Gamma_{n+2} \vdash X_1 : q'].$$

Since q is of the kind $\vee$, then according to the construction (see 3.2) there exists a variable x, such that

$$\begin{gathered} \Gamma_1 \vdash x\colon s_1 \to s_1, \\ \dots \\ \Gamma_k \vdash x\colon s'_k \to s_k, \\ \dots \\ \Gamma_n \vdash x\colon s_n \to s_n, \\ \Gamma_{n+1} \vdash x\colon k + r(c) \to k, \\ \Gamma_{n+2} \vdash x\colon q' \to q. \end{gathered}$$

So $T = xT_1$ is a solution of the task Z.

– $g(q) = \wedge$. Then for each transition $((q, s_k), (q_i, s_{ki}, c_i))$ available from C, machine M accepts configuration $C_i = (q_i, s_1 \dots s_{ki} \dots s_n, k + r(c_i))$. By the induction hypothesis there exist solutions $T_1, \dots, T_m$ of tasks $Z_1, \dots, Z_m$, where

$$Z_i = [\Gamma_1 \vdash X_i\colon s_1,\ \dots,\ \Gamma_k \vdash X_i\colon s_{ki},\ \dots,\ \Gamma_n \vdash X_i\colon s_n, \\ \Gamma_{n+1} \vdash X_i\colon k + r(c_i),\ \Gamma_{n+2} \vdash X_i\colon q_i].$$

Since q is of the kind $\wedge$, there must (see 3.2) exist a variable x, such that

$$\begin{gathered} \Gamma_1 \vdash x\colon s_1 \to \dots \to s_1 \to s_1, \\ \dots \\ \Gamma_k \vdash x\colon s_{k1} \to \dots \to s_{km} \to s_k, \\ \dots \\ \Gamma_n \vdash x\colon s_n \to \dots \to s_n \to s_n, \\ \Gamma_{n+1} \vdash x\colon k + r(c_1) \to \dots \to k + r(c_m) \to k, \\ \Gamma_{n+2} \vdash x\colon q_1 \to \dots \to q_m \to q. \end{gathered}$$

Then $T = xT_1 \dots T_m$ is a solution of Z.

Corollary 23. *The inhabitation problem for rank two intersection types is* EXPTIME-hard.

References

[1] Alessi, F., Barbanera, F., Dezani-Ciancanglini, M.: Intersection types and lambda models. Theoretical Computer Science 355(2), 108–126 (2006)
[2] Kurata, T., Takahashi, M.: Decidable properties of intersection type systems. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 297–311. Springer, Heidelberg (1995)
[3] Leivant, D.: Polymorphic type inference. Proceedings of the 10th ACM Symposium on Principles of Programing Languages, Austin, Texas, pp. 88–98 (1983)

[4] Lopez-Escobar, E.G.K.: Proof-Functional Connectives. LNCS, vol. 1130, pp. 208–221. Springer-Verlag, Heidelberg (1985)
[5] Mints, G.: The Completeness of Provable Realizability. Notre Dame. Journal of Formal Logic 30, 420–441 (1989)
[6] Papadimitriou, C.H.: Computational Complexity. Addison-Wesley Publishing Company, Reading, MA (1995)
[7] Statman, R.: Intuitionistic propositional logic is polynomial-space complete. TCS 9, 67–72 (1979)
[8] Urzyczyn, P.: The emptiness problem for intersection types. Journal of Symbolic Logic 64(3), 1195–1215 (1999)

Extensional Rewriting with Sums

Sam Lindley

Laboratory for Foundations of Computer Science,
School of Informatics, The University of Edinburgh
Sam.Lindley@ed.ac.uk*

Abstract. Inspired by recent work on normalisation by evaluation for sums, we propose a normalising and confluent extensional rewriting theory for the simply-typed λ-calculus extended with sum types. As a corollary of confluence we obtain decidability for the extensional equational theory of simply-typed λ-calculus extended with sum types. Unlike previous decidability results, which rely on advanced rewriting techniques or advanced category theory, we only use standard techniques.

1 Introduction

It is easy to add sum types to the equational theory of the simply-typed λ-calculus, in the presence of η-rules, or to add sum types to the rewriting theory of simply-typed λ-calculus, in the absence of η-rules. However, adding sum types to the rewriting theory is difficult in the presence of η-rules. Existing rewriting theories, with the exception of Ghani's [5], are either incomplete with respect to the equational theory or non-confluent. Quoting Altenkirch et al [1], Ghani's work involves 'intricate rewriting techniques whose details are daunting'. Our aim is to introduce a straightforward rewriting theory using standard techniques.

The essential reason why the problem with confluence arises is that reordering independent nested cases does not change the semantics of a term. For instance, let $=$ be equivalence in the equational theory, writing $\delta(p, x_1.n_2, x_2.n_2)$ for $\mathsf{case}\ p\ \mathsf{of}\ \mathsf{in}_1(x_1) \Rightarrow n_2 \mid \mathsf{in}_2(x_2) \Rightarrow n_2$, then

$$\begin{aligned}&\delta(p_1, x_1.\delta(p_2, y_1.n_1, y_2.n_2), x_2.\delta(p_2, y_1.m_1, y_2.m_2))\\ &\quad = \delta(p_2, y_1.\delta(p_1, x_1.n_1, x_2.m_1), y_2.\delta(p_1, x_1.n_2, x_2.m_2))\end{aligned}$$

where $x_1, x_2, y_1, y_2, m_1, m_2, n_1, n_2, p_1, p_2$ are distinct object variables. The structure of the terms on each side of the equation is identical, so it is not possible to capture the equivalence with a rewrite rule.

This article explores several solutions to the case ordering problem, all suggested by the work of Altenkirch et al [1] and Balat et al [2] on normalisation by evaluation for the simply-typed λ-calculus extended with sums. The goal of normalisation by evaluation [4] is to find a unique normal form with respect to the equational theory. In contrast, we shall be interested in normal forms with respect to a rewriting theory.

* Supported by EPSRC grant number EP/D046769/1.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 255–271, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In fact, the case ordering problem also manifests itself in the equational setting. In the example above, either the left hand side or the right hand side of the equivalence should be a normal form. But the terms are structurally identical so some non-structural property must be used to define normal forms. One possibility is to define an ordering on terms via an ordering on variable names. The ordering on terms can then be used to assign an ordering to nested cases. Such an ordering is undesirable as it requires us to dispense with α-conversion.

Altenkirch et al solve the problem by adding a new construct to the object language — a parallel case that simultaneously eliminates a set of sums. Using the extended language both sides of the equivalence are represented by the same parallel case. A big advantage of this approach is that it leads to a syntax which much more closely captures the semantics of the calculus. The main disadvantage is that it drastically increases the complexity of the machinery used by the syntax of the language. In Altenkirch et al's presentation functions appear in the syntax. These functions can be represented more concretely using sets or lists, but the resulting syntax is still significantly more complex than the standard one.

Balat et al [2] build on Altenkirch et al's work. Instead of adding parallel cases, they define a congruence over terms which contains the equivalence given in the example above as a special case. They identify normal forms up to this congruence, leading to a rather elegant presentation.

We adopt an extension $\sim$ of Balat et al's congruence, and perform rewriting modulo $\sim$. Sect. 2 introduces the equational theory $\lambda^{\to\times+}$ of simply-typed lambda-calculus extended with products and sums, and decomposes the general η axiom for sums into a number of simpler axioms. Sect. 3 describes a non-local rewriting theory that generates the equational theory. Sect. 4 gives a reducibility proof of strong normalisation for a fragment of the rewriting theory following the approach of Lindley and Stark [8, Chaper 3][9]. Sect. 5 uses strong normalisation results for fragments of the rewriting theory to prove weak normalisation and confluence modulo $\sim$ for the full rewriting theory, and hence decidability for the equational theory. Sect. 6 describes three variations of the rewriting theory. Sect. 7 concludes.

2 The Object Language

The simply typed lambda calculus extended with products and sums is standard [5]. We write $\lambda^{\to\times+}$ for the equational theory.

$$\text{(Types)} \qquad A, B ::= O \mid A \to B \mid A \times B \mid A + B$$

Types are constructed from a base type O, functions $A \to B$ from type A to type B, products $A \times B$ of types A and B, and sums of types A and B. We omit the unit and empty types, but restoring them does not radically change our proofs (though the empty type requires a little more care in the handling of typing contexts).

$$\text{(Terms)} \qquad m, n, p ::= x \mid \lambda x.m \mid m\,n \mid \langle m, n\rangle \mid \pi_1(m) \mid \pi_2(m) \mid \iota_1(m) \mid \iota_2(m) \mid \delta(m, x_1.n_1, x_2.n_2)$$

Terms are constructed from variables x, lambda abstractions $\lambda x.m$, applications $m\,n$, pairs $\langle m, n\rangle$, projections $\pi_i(m)$, injections $\iota_i(m)$ and cases $\delta(m, x_1.n_1, x_2.n_2)$. Terms are identified up to α-conversion.

The free $fv(m)$ and bound variables $bv(m)$ are defined in the usual way. We write $m[x := n]$ for the capture-avoiding substition of n for x in m, and $m[x_1 := n_1, \ldots, x_k := n_k]$ for the simultaneous capture-avoiding substition of n_i for x_i in m ($1 \leqslant i \leqslant k$). We write $size(m)$ for the size of the term m. The typing rules are standard. Each type constructor has an introduction rule and an elimination rule.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash m : B}{\Gamma \vdash \lambda x.m : A \to B} \qquad \frac{\Gamma \vdash m : A \to B \quad \Gamma \vdash n : A}{\Gamma \vdash m\,n : B}$$

$$\frac{\Gamma \vdash m : A \quad \Gamma \vdash n : B}{\Gamma \vdash \langle m, n\rangle : A \times B} \qquad \frac{\Gamma \vdash m : A_1 \times A_2}{\Gamma \vdash \pi_i(m) : A_i}\ i \in \{1, 2\}$$

$$\frac{\Gamma \vdash m : A_i}{\Gamma \vdash \iota_i(m) : A_1 + A_2}\ i \in \{1, 2\}$$

$$\frac{\Gamma \vdash m : A_1 + A_2 \quad \Gamma, x_i : A_i \vdash n_i : B \quad i \in \{1, 2\}}{\Gamma \vdash \delta(m, x_1.n_1, x_2.n_2) : B}$$

Axioms. The axioms for $\lambda^{\to\times+}$ consist of a β-axiom and an η-axiom for each type constructor.

$$\begin{array}{ll}
(\to.\beta) & (\lambda x.m)\,n = m[x := n] \\
(\times.\beta_i) & \pi_i(\langle m_1, m_2\rangle) = m_i, \quad i \in \{1, 2\} \\
(+.\beta_i) & \delta(\iota_i(m), x_1.n_1, x_2.n_2) = n_i[x_i := m], \quad i \in \{1, 2\} \\
(\to.\eta) & m = \lambda x.m\,x, \quad x \notin fv(m) \\
(\times.\eta) & m = \langle \pi_1(m), \pi_2(m)\rangle \\
(+.\eta^\dagger) & n[x := p] = \delta(p, x_1.n[x := \iota_1(x_1)], x_2.n[x := \iota_2(x_2)])
\end{array}$$

The equation $m = n$ is shorthand for the equality judgement $\Gamma \vdash m = n : A$ where $\Gamma \vdash m : A$ and $\Gamma \vdash n : A$. The equational theory is given by the least (typed) congruence satisfying the axioms.

Alternative Axioms. The generalised η-axiom for sums $+.\eta^\dagger$ is non-local and it is not at all obvious how it might give rise to a confluent rewriting system. In particular, note that substitutions appear both on the left and the right hand side of the axiom. We break $+.\eta^\dagger$ down into a number of simpler axioms.

$$(+.\eta) \quad p = \delta(p, x_1.\iota_1(x_1), x_2.\iota_2(x_2))$$

$$(\textit{move-case}) \quad F[\delta(p, x_1.n_1, x_2.n_2)] = \delta(p, x_1.F[n_1], x_2.F[n_2]),$$
$$x_1, x_2 \notin \mathit{fv}(F[\,]) \text{ and } \mathit{bv}(F[\,]) \cap \mathit{fv}(p) = \emptyset$$

$$(\textit{repeated-guard})$$
$$\delta(p, x_1.\delta(p, y_1.n_1, y_2.n_2), x_2.\delta(p, z_1.p_1, z_2.p_2))$$
$$= \delta(p, x_1.n_1[y_1 := x_1], x_2.p_2[z_2 := x_2]), \quad x_1, x_2 \notin \mathit{fv}(p)$$

$$(\textit{redundant-guard}) \quad \delta(p, x_1.n, x_2.n) = n, \quad x_1, x_2 \notin \mathit{fv}(n)$$

The local η axiom for sums $+.\eta$ is a special case of $+.\eta^\dagger$ in which n is just x. The *move-case* axiom is a generalisation of the usual commuting conversions for $\lambda^{\to\times+}$ [6,11]. As well as allowing cases to move across *elimination frames* ($F_1[\,]$), *move-case* also allows them to be moved across *neutral frames* ($F_2[\,]$), *lambda frames* ($F_3[\,]$) and *continuation frames* ($F_4[\,]$).

$$\text{(Frames)} \qquad F[\,] ::= F_1[\,] \mid F_2[\,] \mid F_3[\,] \mid F_4[\,]$$

$$F_1[\,] ::= [\,]\,n \mid \pi_1([\,]) \mid \pi_2([\,]) \mid \delta([\,], x_1.n_1, x_2.n_2)$$
$$F_2[\,] ::= m\,[\,] \mid \langle [\,], n\rangle \mid \langle m, [\,]\rangle \mid \iota_1([\,]) \mid \iota_2([\,])$$
$$F_3[\,] ::= \lambda x.[\,]$$
$$F_4[\,] ::= \delta(p, x_1.[\,], x_2.n_2) \mid \delta(p, x_1.n_1, x_2.[\,])$$

We write *move-case*$_i$ for the restriction of *move-case* to frames of the form F_i. Following Altenkirch et al, we use the word *guard* to refer to the first argument of a case. The axiom *repeated-guard* allows guards to be copied or deleted. The axiom *redundant-guard* is a special case of $+.\eta^\dagger$ in which x does not occur free in n.

Proposition 1. *Replacing the axiom $+.\eta^\dagger$ with the alternative axioms $+.\eta$, move-case, repeated-guard, redundant-guard yields the same equational theory.*

Proof. (sketch)
New axioms are sound:

- $+.\eta$ and *redundant-guard* are instances of $+.\eta^\dagger$
- *move-case*, *repeated-guard*: apply $+.\eta^\dagger$ from left to right using p as the substituted term, eliminate resulting $+.\beta_i$ redexes, then α-convert

New axioms are complete:

- η expand all instances of p in n
- use *move-case* to hoist all instances of p to the top
- use *repeated-guard* and *redundant-guard* to get rid of the multiple copies of p
- use *redundant-guard* for the case where $x \notin \mathit{fv}(n)$

The axioms *move-case*, *repeated-guard* and *redundant-guard*, are implicit in previous work on normalisation by evaluation with sums [1,2]. To the author's knowledge, they have not previously been used as the basis for a rewriting calculus.

3 A Rewriting Theory

As a first attempt at a rewriting theory consider defining a rewrite rule for each axiom by orienting from left to right (with the usual restrictions for η-expansion). Unfortunately the resulting theory has infinite reduction sequences arising from *move-case*$_4$. For instance, the following reductions can be applied indefinitely as m appears as a subterm of n.

$$\begin{aligned} m &= \delta(p, x.n, x.\delta(p, x.n, x.n)) \\ &\longrightarrow_{move\text{-}case_4} \delta(p, x.\delta(p, x.n, x.n), x.\delta(p, x.n, x.n)) \\ &\longrightarrow_{move\text{-}case_4} \delta(p, x.m, x.m) = n \end{aligned}$$

Ohta and Hasegawa [10] face a similar problem for a linear lambda calculus. Their solution is to separate the axioms of their equational theory into a reduction relation and an equivalence relation, and use Huet's technique for proving confluence of the reduction relation modulo the equivalence relation [7]. Balat et al use an equivalence for defining normal forms and implementing normalisation by evaluation with sums. Their equivalence is the least congruence satisfying the *move-case*$_4$ and *redundant-guard* axioms. We introduce a congruence that also includes the *repeated-guard* axiom.

Definition 2. *The relation $\sim$ is the least congruence satisfying the axioms move-case$_4$, repeated-guard and redundant-guard.*

Deciding equivalence modulo $\sim$ is straightforward. First we define some auxiliary functions.

Definition 3

$$Guards(m) = \begin{cases} p \cup Guards(x_1.n_1) \cup Guards(x_2.n_2), & \textit{if } m = \delta(p, x_1.n_1, x_2.n_2) \\ \emptyset, & \textit{otherwise} \end{cases}$$

$$Guards(x.n) = \{m \in Guards(n) \mid x \notin fv(m)\}$$

$$Paths(gs) = \{\rho \mid \rho \in \{1,2\}^{gs}\} \quad \nu(ps) = \{p^{x_1}_{x_2} \mid p \in ps_{/\sim} \textit{ and } x_1, x_2 \textit{ fresh}\}$$

$$Tail_{\rho[p^{x_1}_{x_2} \mapsto i]}(\delta(p', x'_1.n'_1, x'_2.n'_2)) = Tail_{\rho[p^{x_1}_{x_2} \mapsto i]}(n_i[x'_i := x_i]), \ \textit{if } p \sim p'$$

$$Tail_\rho(m) = m$$

The function $Guards(m)$ gives the set of independent *guards* at the top-level of m. The definition of *Guards* is the same as that used by Balat et al. If $ps = Guards(m)$, then $Paths(\nu(ps))$ represents the set of possible *paths* through m dictated by ps. Given a path ρ through a term m, the subterm of m at the end of that path, the *tail* of ρ, is given by $Tail_\rho(m)$. We write $ps_{/\sim}$ for the quotient set of ps by $\sim$.

Proposition 4

$$m_1 \sim m_2 \iff \forall \rho \in \mathit{Paths}(\nu(\mathit{Guards}(m_1) \cup \mathit{Guards}(m_2))).\mathit{Tail}_\rho(m_1) \sim \mathit{Tail}_\rho(m_2)$$

To decide whether $m_1 \sim m_2$: if one of m_1, m_2 is a case, then use Prop. 4; otherwise compare the top-level constructors and if equal recurse on the immediate subterms of m_1, m_2. Having defined the decidable equivalence $\sim$, we now present the rewrite rules. The β- and η-rules are standard.

β-*Rules*

$$
\begin{array}{ll}
(\to.\beta) & \lambda x.m\,n \longrightarrow m[x := n] \\
(\times.\beta_1) & \pi_1(\langle m_1, m_2\rangle) \longrightarrow m_1 \\
(\times.\beta_2) & \pi_2(\langle m_1, m_2\rangle) \longrightarrow m_2 \\
(+.\beta_1) & \delta(\iota_1(m), x_1.n_1, x_2.n_2) \longrightarrow n_1[x_1 := m] \\
(+.\beta_2) & \delta(\iota_2(m), x_1.n_1, x_2.n_2) \longrightarrow n_2[x_2 := m]
\end{array}
$$

η-*Rules.* The η-rules are type-directed expansions.

$$
\begin{array}{ll}
(\to.\eta) & m^{A\to B} \longrightarrow \lambda x.m\,x, \qquad \text{if } x \notin \mathit{fv}(m) \\
(\times.\eta) & m^{A\times B} \longrightarrow \langle \pi_1(m), \pi_2(m)\rangle \\
(+.\eta) & m^{A+B} \longrightarrow \delta(m, x_1.\iota_1(x_1), x_2.\iota_2(x_2))
\end{array}
$$

The annotation m^A means m has type A. The η-rules are applicable only if expansion does not create a new redex. More precisely, only variables, applications and projections (*pure neutral terms*) can be η-expanded, and only in a non-elimination frame.

In order to instantiate the *move-case* axiom as a rewrite rule we read it from left to right. This corresponds to *hoisting* a case over a frame. Of course, we do not generally need to allow hoisting over continuation frames, as this is captured by $\sim$. However, for confluence it is necessary to allow hoisting over several continuation frames followed by a non-continuation frame.

Frames and Contexts

$$
\begin{array}{lrl}
\text{(Hoisting frames)} & H[\,] ::= & F_1[\,] \mid F_2[\,] \mid F_3[\,] \\
\text{(Discriminator contexts)} & D[\,] ::= & [\,] \mid \delta(p, x_1.D[\,], x_2.n_2) \\
 & & \mid \delta(p, x_1.n_1, x_2.D[\,]) \\
\text{(Hoisting contexts)} & HD[\,] ::= & H[D[\,]]
\end{array}
$$

γ-*Rules.* We refer to reduction rules that arise from the *move-case*-, *repeated-guard*- and *redundant-guard*-axioms as γ-*rules.*

$(\mathit{hoist\text{-}case})$

$$HD[\delta(p, x_1.n_1, x_2.n_2)] \longrightarrow \delta(p, x_1.HD[n_1], x_2.HD[n_2]),$$
$$x_1, x_2 \notin \mathit{fv}(HD) \text{ and } \mathit{bv}(HD) \cap \mathit{fv}(p) = \emptyset$$

The *hoist-case*-rule is obtained from the *move-case* axiom. It is a generalisation of the usual commuting conversions. Note that continuation frames are not hoisting frames, as naïvely hoisting over continuation frames would lead to non-termination. However, the equivalence $\sim$ includes the possibility of moving cases over continuation frames, and the *hoist-case*-rule does allow a case to be hoisted over a hoisting frame from inside a discrimination context.

The discrimination context is necessary in the *hoist-case*-rule because it is only sound to hoist a case over a lambda abstraction if the lambda-bound variable does not occur free in the guard. If hoisting from within a discrimination context is disallowed, then some cases become blocked from being hoisted outside the lambda by outer cases that depend on the bound variable. For instance, suppose D is restricted to be the empty context $[\,]$, then the terms

$$\lambda x.\delta(x, x_1.\delta(z, y_1.y_1, y_2.y_2), x_2.x_2)$$

and

$$\delta(z, y_1.\lambda x.\delta(x, x_1.y_1, x_2.x_2), y_2.\lambda x.\delta(x, x_1.y_2, x_2.x_2))$$

become distinct normal forms, despite the fact that these terms are identified in the equational theory.

Remark. For confluence it is not necessary to have a discrimination context in the *hoist-case*$_1$- and *hoist-case*$_2$-rules, but here we gave the single general *hoist-case*-rule for the sake of uniformity.

Definition 5 (Reduction relations)

$\longrightarrow_\beta$ = *the compatible closure of the* β*-rules*
$\longrightarrow_\eta$ = *the restricted compatible closure of the* η*-rules*
$\longrightarrow_\gamma$ = *the compatible closure of the* γ*-rules*
$\longrightarrow_{\gamma_\mathsf{E}}$ = *the compatible closure of hoist-case with* HD *restricted to* F_1
(i.e. the standard commuting conversion reduction relation)
$\longrightarrow_{\gamma'} = \longrightarrow_\gamma \setminus \longrightarrow_{\gamma_\mathsf{E}}$
$\longrightarrow_\mathsf{c} = \longrightarrow_\beta \cup \longrightarrow_\eta \cup \longrightarrow_{\gamma_\mathsf{E}}$
$\longrightarrow = \longrightarrow_\beta \cup \longrightarrow_\eta \cup \longrightarrow_\gamma$

4 Strong Normalisation for $\beta\eta\gamma_\mathsf{E}$-reduction

Strong normalisation is standard for $\beta\eta\gamma_\mathsf{E}$-reduction [5,11]. In this section we present an adaptation of the strong normalisation proof given in the author's thesis [8, Chapter 3]. Our use of frame stacks alleviates difficulties with γ_E-reduction, and leads to a significantly simpler proof than Prawitz's original one [11].

Definition 6. *A term* m *is strongly normalising with respect to a reduction relation* R*, or* R-SN*, if all* R*-reduction sequences starting from* m *are finite. A reduction relation* R *is* strongly normalising*, or* SN*, if all terms* m *are* R*-SN. If* m *is* R*-SN, then we write* $max_\mathsf{R}(m)$ *for the maximum length of a reduction sequence starting from* m*.*

Definition 7 (frame stacks)

$$\begin{array}{ll} (\textit{elimination frames}) & \mathsf{E} ::= \mathsf{F}_1 \\ (\textit{frame stacks}) & \mathsf{S} ::= \mathit{Id} \mid \mathsf{S} \circ \mathsf{E} \end{array}$$

$$\begin{array}{ll} (\textit{stack length}) & |\mathit{Id}| = 0 \\ & |\mathsf{S} \circ \mathsf{E}| = |\mathsf{S}| + 1 \end{array}$$

$$\begin{array}{ll} (\textit{plugging}) & \mathit{Id}[\mathsf{m}] = \mathsf{m} \\ & (\mathsf{S} \circ \mathsf{E})[\mathsf{m}] = \mathsf{S}[(\mathsf{E}[\mathsf{m}])] \end{array}$$

Following Girard et al [6] we assume variables are annotated with types (it is straightforward, albeit somewhat tedious, to adapt the proof to use local typing contexts instead). We write $A \multimap B$ for the type of frame stack S, if $S[m] : B$ for all terms $m : A$.

Definition 8 (frame stack reduction)

$$S \longrightarrow_c S' \quad \stackrel{def}{\Longleftrightarrow} \quad \forall m.S[m] \longrightarrow_c S'[m]$$

A frame stack S is c-*strongly normalising* if all c-reduction sequences starting from S are finite.

Lemma 9

1. *$S \longrightarrow_c S'$ iff $S \neq Id$ and $S[x] \longrightarrow_c S'[x]$.*
2. *If $S \longrightarrow_c S'$, for frame stacks S, S', then $|S'| \leqslant |S|$.*
3. *If there exists m such that $S[m]$ is c-SN, then $S[x]$ is c-SN.*

Proof Induction on the structure of S.

Definition 10 (reducibility)

- *Id is reducible.*
- *$S \circ [\,]\, n : (A \to B) \multimap C$ is reducible if S and n are reducible.*
- *$S \circ \pi_i([\,]) : (A \times B) \multimap C$ is reducible if S is reducible.*
- *$S : (A + B) \multimap C$ is reducible if $S[\iota_1(m)]$ is c-SN for all reducible $m : A$, and $S[\iota_2(n)]$ is c-SN for all reducible $n : B$.*
- *$m{:}A$ is reducible if $S[m]$ is c-SN for all reducible $S : A \multimap C$.*

Lemma 11. *If $m : A$ is reducible then m is c-SN.*

Proof. Follows immediately from reducibility of *Id* and the definition of reducibility on terms.

Lemma 12. *$x : A$ is reducible.*

Proof. By induction on A using Lemma 9 and Lemma 11.

Corollary 13. *If $S : A \multimap C$ is reducible then S is c-SN.*

Each type constructor has an associated β-rule. Each β-rule gives rise to an SN-closure property.

Lemma 14 (SN-closure)

$\to$ *If* $S[m[x := n]]$ *and* n *are* c*-SN then* $S[(\lambda x.m)\, n]$ *is* c*-SN.*
$\times.1$ *If* $S[m]$ *and* n *are* c*-SN then* $S[\pi_1(\langle m, n\rangle)]$ *is* c*-SN.*
$\times.2$ *If* $S[n]$ *and* m *are* c*-SN then* $S[\pi_2(\langle m, n\rangle)]$ *is* c*-SN.*
$+.1$ *If* $S[n_1[x_1 := m]]$*,* $S[n_2]$ *and* m *are* c*-SN then* $S[\delta(\iota_1(m), x_1.n_1, x_2.n_2)]$ *is* c*-SN.*
$+.2$ *If* $S[n_2[x_2 := m]]$*,* $S[n_1]$ *and* m *are* c*-SN then* $S[\delta(\iota_2(m), x_1.n_1, x_2.n_2)]$ *is* c*-SN.*

Proof

$\to, \times.1, \times.2$**:** By induction on $max_c(S) + max_c(m) + max_c(n)$.
$+.1$**:** By induction on $|S| + max_c(S[n_1[x_1 := m]]) + max(S[n_2]) + max_c(m)$.
$+.2$**:** By induction on $|S| + max_c(S[n_2[x_2 := m]]) + max_c(S[n_1]) + max_c(m)$.

Now we obtain reducibility-closure properties for each type constructor.

Lemma 15 (reducibility-closure)

$\to$ *If* $m[x := n]$ *is reducible for all reducible* n*, then* $\lambda x.m$ *is reducible.*
$\times$ *If* m, n *are reducible, then* $\langle m, n\rangle$ *is reducible.*
$+$ *If* m *is reducible,* $n_1[x_1 := l]$ *is reducible for all reducible* l*, and* $n_2[x_2 := p]$ *is reducible for all reducible* p*, then* $\delta(m, x_1.n_1, x_2.n_2)$ *is reducible.*

Proof Each property follows from the corresponding part of Lemma 14 using Lemma 11 and Corollary 13.

Theorem 16. *Let* m *be any term. Suppose* $x_1 : A_1, \ldots, x_k : A_k$ *includes all the free variables of* m*. If* $p_1 : A_1, \ldots, p_k : A_n$ *are reducible then* $m[x_1{:=}p_1, \ldots, x_k{:=} p_k]$ *is reducible.*

Proof. By induction on the structure of terms using Lemma 15.

Theorem 17 (strong normalisation). *All terms are* c*-SN.*

Proof. Let m be a term with free variables $x_1, \ldots, x_k$. By Lemma 12, $x_1, \ldots, x_k$ are reducible. Hence, by Thm. 16, m is c-SN.

5 Weak Normalisation and Confluence

It is straightforward to prove that γ-reduction is strongly normalising.

Lemma 18. *If* $D[x], m$ *are* γ*-SN, then* $D[m]$ *is* γ*-SN.*

Proof. By induction on $max_\gamma(D[x]) + max_\gamma(m)$.

Lemma 19. *If* p, n_1, n_2 *are* γ*-SN, then* $\delta(p, x_1.n_1, x_2.n_2)$ *is* γ*-SN.*

Proof. By induction on $\langle max_\gamma(p), size(p), max_\gamma(n_1) + max_\gamma(n_2)\rangle$. The only interesting case is

$$\begin{array}{l}\delta(D[\delta(p, x_1.p_1, x_2.p_2)], y_1.n_1, y_2.n_2) \\ \quad \longrightarrow_\gamma \delta(p, x_1.D[\delta(p_1, y_1.n_1, y_2.n_2)], x_2.D[\delta(p_2, y_1.n_1, y_2.n_2)])\end{array}$$

By the induction hypothesis, $\delta(p_1, y_1.n_1, y_2.n_2)$ and $\delta(p_2, y_1.n_1, y_2.n_2)$ are both γ-SN. Then by Lemma 18 and the induction hypothesis

$$\delta(p, x_1.D[\delta(p_1, y_1.n_1, y_2.n_2)], x_2.D[\delta(p_2, y_1.n_1, y_2.n_2)])$$

is γ-SN.

Lemma 20

1. *If m, n are γ-SN then $m\,n$ is γ-SN.*
2. *If m, n are γ-SN then $\langle m, n\rangle$ is γ-SN.*
3. *If m is γ-SN then $\lambda x.m$ is γ-SN.*
4. *If m is γ-SN then $\pi_i(m)$ is γ-SN.*
5. *If m is γ-SN then $\iota_i(m)$ is γ-SN.*

Proof

1-2 By induction on $\langle max_\gamma(m) + max_\gamma(n), size(m) + size(n))\rangle$.
3-5 By induction on $\langle max_\gamma(m), size(m)\rangle$.

Theorem 21. *γ is strongly normalising.*

We now obtain weak normalisation for $\beta\eta\gamma$-reduction. The key observation is that γ-reduction following $\beta\eta\gamma_E$-normalisation cannot introduce new $\beta\eta$-redexes.

Lemma 22. *If m is in $\beta\eta\gamma_E$-normal form and $m \longrightarrow_\gamma^* m'$, then m' is in $\beta\eta$-normal form.*

Lemma 22 is easily proved by a straightforward syntactic analysis of the structure of the term m'. The details are omitted due to lack of space.

Theorem 23. *$\longrightarrow$ is weakly normalising.*

Proof. To normalise a term of m, first reduce to a $\beta\eta\gamma_E$-normal form m', then reduce m' to γ-normal form m''. By Lemma 22, m'' must be a $\beta\eta\gamma$-normal form.

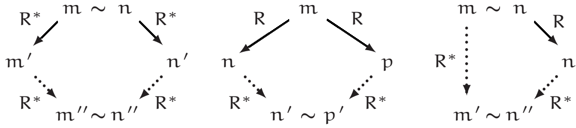
We could obtain confluence by appealing to correctness of normalisation by evaluation for sums [1]. Instead, we give a direct proof of confluence modulo $\sim$ using the strong normalisation results for $\longrightarrow_c$ and $\longrightarrow_\gamma$.

We write R^* for the transitive reflexive closure of the relation R.

Definition 24. *A reduction relation R is:*

- *confluent modulo $\sim$ iff for all m, n, m', n' with $m \sim n$, $m \longrightarrow_R^* m'$ and $n \longrightarrow_R^* n'$, there exist m'', n'' with $m' \longrightarrow_R^* m''$, $n' \longrightarrow_R^* n''$ and $m'' \sim n''$.*
- *weakly confluent modulo $\sim$ iff for all m, n, p with $m \longrightarrow_R n$ and $m \longrightarrow_R p$, there exist n', p' with $n \longrightarrow_R^* n'$, $p \longrightarrow_R^* p'$ and $n' \sim p'$.*
- *weakly coherent modulo $\sim$ iff for all m, n, n' with $m \sim n$ and $n \longrightarrow_R n'$, there exist m', n'' with $m \longrightarrow_R^* m'$, $n' \longrightarrow_R^* n''$ and $m' \sim n''$.*

Confluence, Weak Confluence, and Weak Coherence, All Modulo $\sim$



Theorem 25 (Huet's Theorem [7]). *If the reduction relation* R *is strongly normalising, weakly confluent modulo* $\sim$ *and weakly coherent modulo* $\sim$*, then* R *is also confluent modulo* $\sim$*.*

Proposition 26
$\longrightarrow_\beta, \longrightarrow_\eta, \longrightarrow_c, \longrightarrow_\gamma, \longrightarrow$ *are all weakly confluent modulo* $\sim$*.*

Proposition 27
$\longrightarrow_\beta, \longrightarrow_\eta, \longrightarrow_c, \longrightarrow_\gamma, \longrightarrow$ *are all weakly coherent modulo* $\sim$*.*

Proposition 28
$\longrightarrow_\beta, \longrightarrow_\eta, \longrightarrow_c, \longrightarrow_\gamma$ *are all confluent modulo* $\sim$*.*

Proof By Huet's Theorem using Prop. 26, Prop. 27, Thm. 17 and Thm. 21.

We now show confluence of $\longrightarrow$ modulo $\sim$ using some intermediate Lemmas. The only non-trivial interaction is between β- and γ'-reduction. Following Barendregt [3, Chapter 11] we allow β-redexes to be marked. A redex is marked by overlining it. Notice that γ'-reduction can *hide* β-redexes inside a γ_E-redex. In such cases, we allow the γ_E-redex to be marked. For instance

$$\overline{(\lambda x.\delta(p, x_1.n_1, x_2.n_2))\, m} \longrightarrow_{\gamma'} \overline{\delta(p, x_1.\lambda x.n_1, x_2.\lambda x.n_2)\, m}$$

Definition 29

$$\begin{aligned}
\varphi(\overline{(\lambda x.m)\, n}) &= \varphi(m)[x := \varphi(n)] \\
\varphi(\overline{\pi_i(\langle m, n\rangle)}) &= \varphi(m) \\
\varphi(\overline{\delta(\iota_i(m), x_1.n_1, x_2.n_2)}) &= \varphi(n_i)[x_i := \varphi(m)] \\
\varphi(\overline{E[\delta(p, x_1.n_1, x_2.n_2)]}) &= \delta(\varphi(p), x_1.\varphi(\overline{E[n_1]}), x_2.\varphi(\overline{E[n_2]}))
\end{aligned}$$

φ *commutes with all the other syntax constructors.*

The φ function contracts all of the marked β-redexes in a term.

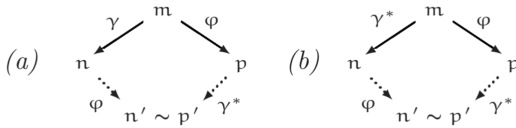
Lemma 30. $\varphi(m[x := n]) = \varphi(m)[x := n]$

Proof. By induction on the structure of m.

Lemma 31. *If* $m \sim m'$ *then* $\varphi(m) \sim \varphi(m')$*.*

Proof. By induction on the derivation of $m \sim m'$.

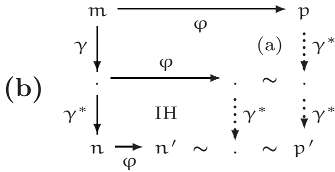
Lemma 32



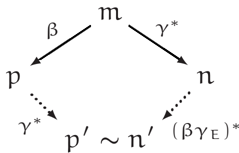
Proof

(a) By induction on the derivation of γ using Lemma 30 and Lemma 31.

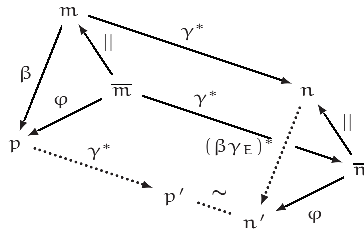
(b)



Lemma 33

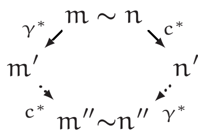


Proof. Let $\overline{\mathfrak{m}}$ be $\mathfrak{m}$ with the β-redex marked, and $||$ be an operator that erases marked redexes, but otherwise leaves a term unchanged.



The front triangle is proved by induction on the structure of $\overline{\mathfrak{n}}$. The bottom rectangle is proved by Lemma 32.

Proposition 34.



Proof. Using Lemma 33.

Theorem 35. $\longrightarrow$ *is confluent modulo* $\sim$*.*

Proof. By a diagram chase using Prop. 28 and Prop. 34.

Theorem 36. $\lambda^{\rightarrow\times+}$ *is decidable.*

Proof. By Thm. 35, and Thm. 23, every $\lambda^{\rightarrow\times+}$-term has a unique normal form obtained by reducing to $\beta\eta\gamma_E$-normal form and then to γ-normal form. To decide whether terms m, n are equal simply reduce them to normal forms m', n' and then compute whether $m' \sim n'$.

6 Variations

Unblocking Cases. It would be nice if it was possible to remove discrimination contexts from the *move-case*$_3$-rule, and so allow all the rewrite rules to be local. One way of doing so is to mark a case as blocked when it is adjacent to a lambda abstraction on whose bound variable the guard depends. Then unblocked cases can be lifted over blocked cases. The resulting calculus is somewhat fiddly, though, as blocked cases can subsequently become unblocked by β-reductions inside the guard. We omit the details, and instead consider some more well-behaved alternatives.

Parallel Cases. Altenkirch et al [1] use parallel cases in order to define normalisation by evaluation for sums. We write

$$\Delta([(x_0, p_0), \ldots, (x_{l-1}, p_{l-1})], [e_0, \ldots, e_{2^l-1}])$$

for the parallel case over the guards $p_0, \ldots, p_{l-1}$ with binders $x_0, \ldots, x_{l-1}$ and *tails* $e_0, \ldots, e_{2^l-1}$.

An easy way to comprehend the syntax is via the erasure *ser* from parallel cases to a tree of nested serial cases.

$$\begin{aligned} ser(\Delta((x,p) :: gs, es_1 +\!\!+ es_2)) &= \delta(p, x.ser(\Delta(gs, es_1)), x.ser(\Delta(gs, es_2))) \\ ser(\Delta([], [e])) &= e \end{aligned}$$

The *ser* function commutes with all other syntax constructors. The operator :: appends an element to the front of a list. The operator ++ concatenates two lists of equal length. The translation *par* from a $\lambda^{\rightarrow\times+}$-term to a term with parallel cases, simply converts each serial case into a parallel case with one guard.

$$par(\delta(p, x.n_1, x.n_2)) = \Delta((x, par(p)), [par(n_1), par(n_2)])$$

The *par* function commutes with all other syntax constructors.

Definition 37. *The relation* $\approx$ *is the least congruence such that for all permutations perm of the integers* $1, \ldots, n$*:*

$$\Delta([], [e]) \approx e \qquad \Delta(gs, [e_0, \ldots, e_{2^l-1}]) \approx \Delta(gs', [e'_0, \ldots, e'_{2^l-1}])$$

where

$$gs = (x_0, p_0) \ldots (x_{l-1}, p_{l-1}) \qquad e'_i = e_{perm*(i)}$$
$$gs' = (x'_0, p'_0) \ldots (x'_{l-1}, p'_{l-1}) \textit{ with } x'_i = x_{perm(i)} \textit{ and } p'_i = p_{perm(i)}$$

$$perm*(i) = \uparrow (perm_2(\downarrow i)) \quad perm_2(b_{l-1} \ldots b_0) = b_{perm(l-1)} \ldots b_{perm(0)}$$
$\downarrow, \uparrow$ *convert natural numbers to and from binary*

Given two terms m_1 and m_2, then $m_1 \approx m_2$ iff m_2 can be obtained from m_1 by permuting guards (and adjusting binders and tails accordingly).

β- and η-Rules. The β- and η-rules are as in the serial rewriting theory, except for sums, where they are translated in the obvious way.

$$(+.\beta_1) \qquad \Delta((x, \iota_1(m)) :: gs, es_1 +\!\!+ es_2) \longrightarrow \Delta(gs, es_1[x := m])$$
$$(+.\beta_2) \qquad \Delta((x, \iota_2(m)) :: gs, es_1 +\!\!+ es_2) \longrightarrow \Delta(gs, es_2[x := m])$$
$$(+.\eta) \qquad m^{A+B} \longrightarrow \Delta((x, m), [\iota_1(x), \iota_2(x)])$$

For sum types, β-rules are only needed for the first guard of a parallel elimination. Other guards can just be eliminated by first applying $\approx$.

γ-Rules

(*hoist-case*)

$$\mathsf{HP}[\Delta((x, p) :: gs, es_1 +\!\!+ es_2)]$$
$$\longrightarrow \Delta([x, p], [\mathsf{HP}[\Delta(gs, es_1)], \mathsf{HP}[\Delta(gs, es_2)]]]),$$
$$x \notin fv(\mathsf{HP}[\ ]) \text{ and } bv(\mathsf{HP}[\ ]) \cap fv(p) = \emptyset$$

(*redundant-guard*)

$$\Delta((x, p) :: gs, es_1 +\!\!+ es_2) \longrightarrow \Delta(gs, es_1),$$
$$es_1 \approx es_2 \text{ and } x \notin fv(es_1 +\!\!+ es_2)$$

(*repeated-guard*)

$$\Delta((x_1, p_1) :: (x_2, p_2) :: gs, (es_1 +\!\!+ es_2) +\!\!+ (es_3 +\!\!+ es_4))$$
$$\longrightarrow \Delta((x_1, p_1) :: gs, es_1 +\!\!+ es_4), \quad p_1 \approx p_2$$

(*join-cases*)

$$\Delta(gs, [e_1, \ldots, e_k, \ldots e_{2^l}]) \longrightarrow$$
$$\Delta((x, p) :: gs, [e'_{1,j} | 1 \leqslant j \leqslant 2^l] +\!\!+ [e'_{2,j} | 1 \leqslant j \leqslant 2^l])$$

where

$$e_k = \Delta((x, p) :: gs', es_1 +\!\!+ es_2)$$
$$\{x\} \notin \mathit{Binders}(gs) \text{ and } (\mathit{Binders}(gs) \cap fv(p)) = \emptyset$$
$$e'_{i,j} = \begin{cases} e_j, & \text{if } j \neq k \\ \Delta(gs', es_i), & \text{otherwise} \end{cases}$$
$$\mathit{Binders}(\Delta([(x_0, p_0), \ldots, (x_{l-1}, p_{l-1})], [e_0, \ldots, e_{2^l-1}])) = \{x_0, \ldots, x_{l-1}\}$$

$$\begin{aligned}\mathsf{HP}[\,] ::= {} & [\,]\,n \mid \pi_1([\,]) \mid \pi_2([\,]) \mid \Delta((x,[\,]) :: gs, es) \\ & \mid \lambda x.[\,] \mid m\,[\,] \mid \langle [\,], n\rangle \mid \langle m, [\,]\rangle \mid \iota_1([\,]) \mid \iota_2([\,])\end{aligned}$$

The *redundant-guard*- and *repeated-guard*-rules are both obtained by reading the corresponding axioms from left to right. The *move-case*$_4$ axiom is captured by the combination of: parallel cases, the relation $\approx$ and the *join-cases*-rule; which allows a guard of a tail to be merged with the guards of its parent parallel case, providing the guard is independent of the guards of the parent.

Definition 38. $\longrightarrow_{\mathsf{P}} =$ *the union of the compatible closure of the above* β- *and* γ- *rules, and the restricted compatible closure of the above* η*-rules.*

The proofs of Sections 4 and 5 are easily adapted to handle parallel cases.

Proposition 39. $\longrightarrow_{\mathsf{P}/\approx}$ *is weakly normalising and confluent.*

Simulating Parallel Cases. It is possible to simulate parallel cases using plain $\lambda^{\to\times+}$-syntax. The key to avoiding non-termination is to define a congruence such that guards can only be duplicated if in normal form.

Definition 40. *The relation* $\approx'$ *is the least congruence such that*

$$\begin{aligned}
&\delta(p_1, x_1.\delta(p_2, y_1.n_1, y_2.n_2), x_2.\delta(p_2, y_1.n_3, y_2.n_4)) \\
&\quad \approx' \delta(p_2, y_1.\delta(p_1, x_1.n_1, x_2.n_3), y_2.\delta(p_1, x_1.n_2, x_2.n_4)), \\
&\qquad x_1, x_2, y_1, y_2 \notin fv(p_1) \cup fv(p_2) \\
&\delta(p_1, x_1.\delta(p_2, y_1.n_1, y_2.n_2), x_2.n_3) \\
&\quad \approx' \delta(p_1, x_1.\delta(p_2, y_1.n_1, y_2.n_2), x_2.\delta(p_2, y_1.n_3, y_2.n_3)), \\
&\qquad x_2 \notin fv(p_2) \text{ and } y_1, y_2 \notin fv(n_3) \\
&\delta(p_1, x_1.n_1, x_2.\delta(p_2, y_1.n_2, y_2.n_3)) \\
&\quad \approx' \delta(p_1, x_1.\delta(p_2, y_1.n_1, y_2.n_1), x_2.\delta(p_2, y_1.n_2, y_2.n_3)) \\
&\qquad x_1 \notin fv(p_2) \text{ and } y_1, y_2 \notin fv(n_1)
\end{aligned}$$

where in each case p_1, p_2 *must be in normal form.*

γ*-Rules*

$$\begin{aligned}
&(\textit{hoist-case}) \\
&\quad H[\delta(p, x_1.n_1, x_2.n_2)] \longrightarrow \delta(p, x_1.H[n_1], x_2.H[n_2]), \\
&\qquad x_1, x_2 \notin fv(H) \text{ and } bv(H) \cap fv(p) = \emptyset \\
&(\textit{duplicate-guard}) \\
&\quad \delta(p, x_1.\delta(p, y_1.m_1, y_2.m_2), x_2.\delta(p, y_1.n_1, y_2.n_2)) \\
&\quad \longrightarrow \delta(p, x_1.m_1[y_1 := x_1], x_2.n_2[y_2 := x_2]), \quad x_1, x_2 \notin fv(p)
\end{aligned}$$

(*redundant-guard*)

$$C[\delta(p, x_1.n, x_2.n)] \longrightarrow C[n], \quad x_1, x_2 \notin fv(n) \text{ and}$$
$$C \equiv [\,]; \text{ or}$$
$$C \text{ is a hoisting frame; or}$$
$$C \text{ is a continuation frame with } (bv(C) \cap fv(p)) \neq \emptyset$$

The constraints on the context in which *redundant-guard* can be applied are necessary in order to prevent cycles with $\approx'$.

Definition 41. $\longrightarrow_p =$ *the union of* $\longrightarrow_\beta \cup \longrightarrow_\eta$ *and the compatible closure of the above* γ*-rules.*

Proposition 42. $\longrightarrow_{p/\sim'}$ *is weakly normalising and confluent.*

Conjecture 43. $\longrightarrow, \longrightarrow_{P/\approx}, \longrightarrow_{p/\sim'}$ are all strongly normalising.

Intuitively, it seems that $\longrightarrow$ should be strongly normalising. Both c-reduction and γ-reduction are strongly normalising, and γ'-reduction only interacts with c-reduction in such a way as to expose existing redexes, rather than actually creating new ones. If we could prove strong normalisation for $\longrightarrow$, then the confluence proof could be simplified.

7 Conclusion

We have proposed a confluent extensional rewriting theory for simply-typed lambda-calculus extended with sums. The key contribution is confluence and decidability for a conventional rewriting theory. This contrasts with the two previous approaches to decidability. Ghani [5] uses intricate rewriting techniques, whereas Altenkirch et al [1] use normalisation by evaluation and category theory.

Acknowledgements. Thanks to Philip Wadler and the anonymous reviewers for helpful feedback.

References

1. Altenkirch, T., Dybjer, P., Hofmann, M., Scott, P.: Normalization by evaluation for typed lambda calculus with coproducts. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, June 2001, pp. 303–310 (2001)
2. Balat, V., Cosmo, R.D., Fiore, M.: Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In: 31st Symposium on Principles of Programming Languages (POPL 2004), January 2004, pp. 64–76. ACM Press, New York (2004)
3. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logics and the Foundations of Mathmatics, vol. 103. North Holland, Amsterdam (1984)

4. Berger, U., Eberl, M., Schwichtenberg, H.: Normalization by evaluation. In: Möller, B., Tucker, J.V. (eds.) Prospects for Hardware Foundations. LNCS, vol. 1546, pp. 117–137. Springer, Heidelberg (1998)
5. Ghani, N.: Beta-eta equality for coproducts. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 171–185. Springer, Heidelberg (1995)
6. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press, Cambridge (1989)
7. Huet, G.P.: Confluent reductions: Abstract properties and applications to term rewriting systems. J. ACM 27(4), 797–821 (1980)
8. Lindley, S.: Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages. PhD thesis, University of Edinburgh (2005)
9. Lindley, S., Stark, I.: Reducibility and $\top\top$-lifting for computation types. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 262–277. Springer, Heidelberg (2005)
10. Ohta, Y., Hasegawa, M.: A terminating and confluent linear lambda calculus. RTA, pp. 166–180 (2006)
11. Prawitz, D.: Ideas and results in proof theory. In: Proceedings of the 2nd Scandinavian Logic Symposium. Studies in Logics and the Foundations of Mathmatics, vol. 63, pp. 235–307. North Holland, Amsterdam (1971)

Higher-Order Logic Programming Languages with Constraints: A Semantics

James Lipton[1] and Susana Nieva[2]

[1] Wesleyan University, USA, and visiting Professor
Univ. Politécnica de Madrid, Spain
jlipton@wesleyan.edu
[2] Dep. Sistemas Informáticos y Computación
Univ. Complutense de Madrid, Spain
nieva@sip.ucm.es

Abstract. A Kripke Semantics is defined for a higher-order logic programming language with constraints, based on Church's Theory of Types and a generic constraint formalism.

Our syntactic formal system, *hoHH($\mathcal{C}$)* (higher-order hereditary Harrop formulas with constraints), which extends λProlog's logic, is shown sound and complete.

A Kripke semantics for equational reasoning in the simply typed lambda-calculus (Kripke Lambda Models) was introduced by Mitchell and Moggi in 1990. Our model theory extends this semantics to include full impredicative higher-order intuitionistic logic, as well as the executable *hoHH* fragment with typed lambda-abstraction, implication and universal quantification in goals and constraints. This provides a Kripke semantics for the full higher-order hereditarily Harrop logic of λProlog as a special case (with the constraint system chosen to be β,η-conversion).

1 Introduction

Declarative programming languages have been developed with the aim of keeping code as close as possible to some notion of a specification, while at the same time having a reasonably efficient operational interpretation. This goal has usually been pursued by taking the syntax from some underlying formalism, which gives programs and inputs independent mathematical meaning, and then defining a mechanism that makes the code executable in a way consistent with that meaning. Semantics provides a framework in which the two readings can be understood and analyzed, and their compatibility verified.

The original Horn clause-with-resolution formulation of logic programming that underlies Prolog has been quite successful, but has required a significant control component outside the logic for efficiency, and includes controversial metaprogramming predicates.

Two basic lines of research have attempted to add to the expressive power of the original Horn clause core without compromising declarative transparency. One has been based on *expanding the logic* to include, for example, executable fragments of higher-order logic with lambda-conversion, higher-order unification, type theory, linear logic,

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 272–289, 2007.
© Springer-Verlag Berlin Heidelberg 2007

etc. [18], and the other on *adding constraints* [12,22,13] in a reasonably generic fashion. The latter addition can be thought of, declaratively, as transferring the logic from a term model to more complex data domains, giving enriched notions of computation, input, output and unification.

This paper provides semantic tools for modelling logic programming with intuitionistic higher-order logic, with implication and universal quantification in goals, types, λ-terms and constraints. We have chosen to work with one particular logic programming language, *hoHH($\mathcal{C}$)*, that combines the logic underlying λProlog, higher-order hereditarily Harrop formulas over an intuitionistic formulation of Church's Theory of Types [18], with Saraswat's constraint formalism [22], because it incorporates so many of logic programming extensions of interest and along with them, many of the problems that must be overcome in modelling similar languages.

The combination of these features creates a multiple challenge for the semantics: modelling higher-order intuitionistic formulas in an impredicative logic, giving meaning to λ-terms and types and relativizing interpretations to a foreign black-box constraint system.

Several problems must be solved here. To begin with, there is the *logical intensionality* problem: one must supply a denotation of λ-terms, including those of boolean type, while simultaneously giving them truth values. Logic programming in type theory requires a certain amount of noninterference between the two. In higher-order logic, and in particular in λProlog, predicates may appear as arguments to predicates, yet logically equivalent predicates *must not give rise inevitably to identical denotations*: if F_1 and F_2 are logically equivalent formulas, it need not follow that for any higher-order predicate p of the right type $p(F_1)$ and $p(F_2)$ are equivalent. Otherwise a goal $?-p(F_1)$ with a program $p(F_2)$ (which *fails* in λProlog) could not be handled just by unification over a constraint theory. It would call the entire proof search mechanism into play just to determine first of F_1 were equivalent to F_2.

Also *impredicativity* must be dealt with *a priori*. If X is of boolean type, an instance $G[t/X]$ of a higher-order formula such as $\exists XG$ may have *greater complexity* than the original formula (just consider $t = (\exists XG)$). Thus the usual inductive definability of truth must somehow be circumvented, either by inducting on something other than formulas (types in Henkin's completeness theorem [10], or other measures as in the second class of models defined in this paper) or by a mixed approach, making the definition of truth non-inductive as in our initial Kripke semantics, or as in [23,1,6].

In this paper we show *hoHH($\mathcal{C}$)* is sound and complete for our Kripke structures, extending earlier partial results [17,7,26], and including Kripke semantics for the full logic of λProlog as a special case, obtained by taking β,η-conversion for our constraint system.

2 Higher-Order Intuitionistic Logic with Constraints

In this section we briefly recapitulate for the reader's convenience the main syntactic features of the *hoHH($\mathcal{C}$)* programming language, and its sequent calculus, treated in much greater detail in [15,16].

2.1 Syntactic Preliminaries

The formalization of higher-order logic used here is based on an intuitionistic reformulation [18] of Church's theory of types [4], a theory that builds higher-order logic on top of the simply typed λ-calculus. The existence of types facilitates the incorporation of a foreign constraint system. Logical formulas are terms of type o, and constraints are terms of a new base type γ. We are thus able to define mixed theories using a single formal mechanism.

We begin by defining what terms, types, formulas and constraints are in Church's Theory of Types. In the next section (2.2), on the syntax of *hoHH($\mathcal{C}$)*, we will restrict this class of formulas to a subset of legal *clauses* and *goals*.

The main components of Church's Type Theory are *types* and *terms*. The set Ty of types, with elements α, includes at least atomic types, called sorts, and functional types, $\alpha \rightarrow \alpha$. The set of sorts must contain at least the two special sorts o and γ. The functional type $\rightarrow$ associates to the right. We will also freely make use of the more compact *Church's type notation* $\beta\alpha$ for $\alpha \rightarrow \beta$, which, since given in reverse order, associates to the left.

Typed terms, denoted by t (or t^{α} when displaying their type is of interest) are obtained from a set V of typed variables, x^{α}, and a signature Σ consisting of a set of typed constant symbols, c^{α}, by the abstraction and type-compatible application operations of the λ-calculus: $t := x^{\alpha} \mid c^{\alpha} \mid (\lambda x^{\alpha}.\, t) \mid (t^{\alpha\rightarrow\beta}\, t^{\alpha})$. We omit parentheses when they are not necessary, assuming that abstraction and application are right and left associative, respectively, and that application has smaller scope than abstraction.

Terms of type o are called *logic formulas*. Terms of type γ are called *constraint formulas*, and they are usually denoted by C.

We say that a term t is in λ-*normal* form $\Lambda(t)$ when it is in both β,η-normal form. By λ-equivalence we mean α,β,η-equivalence between terms, and we denote it $\equiv_{\lambda}$. Every term in our type theory (the simply typed λ-calculus) is equivalent to one in normal form [9,11].

We use the notation $\mathsf{fv}(t)$ or $\mathsf{fv}(S)$ to denote the set of free variables in a term t or in a set S, respectively. We denote simultaneous substitution of terms t_i for every free occurrence of x_i in a term t by $t[t_1/x_1,\ldots,t_n/x_n]$, or simply $t[\overline{t}/\overline{x}]$, and we will assume that $t[\overline{t}/\overline{x}]$ is in fact $\Lambda(t[\overline{t}/\overline{x}])$.

The signature Σ is partitioned into *logical* and *nonlogical constants*. The logical constants are the symbols: $\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\exists$, $\forall$, of certain specific types. Since constraints and pure logical formulas are terms of different types, Σ must contain logical constants of different types to build constraints or logical formulas. For instance, $\exists^{(\alpha\rightarrow\gamma)\rightarrow\gamma}$, $\exists^{(\alpha\rightarrow o)\rightarrow o}$, or $\wedge^{o\rightarrow o\rightarrow o}$, $\wedge^{\gamma\rightarrow o\rightarrow o}$ are always elements of Σ. The non-logical constants are those defined by the user, including a symbol for equality: $\approx^{\alpha\rightarrow\alpha\rightarrow\gamma}$, for every type α.

We use infix notation for $\approx, \wedge, \vee, \Rightarrow$, and, following Church, we abbreviate $\exists(\lambda x.F)$, $\forall(\lambda x.F)$ by $\exists x F$ and $\forall x F$, respectively. We call a logic formula in normal form whose leftmost non-parenthesis symbol is either a nonlogical constant or a variable an *atomic formula*, *rigid* in the former case, and *flexible* in the latter. This leading symbol is called the *predicate symbol* or *predicate variable*, respectively, of the atomic formula in

question. We denote atomic formulas by A. A_r represents rigid atomic formulas. For predicate variables (variables of type o) we use capital letters X, Y.

For the rest of this paper we will take the signature Σ and the initial set of variables V to be fixed, with one exception. The set of variables will be extended in the proof of the completeness theorem.

2.2 The Programming Language *hoHH($\mathcal{C}$)*

In [18] Miller *et. al.* identified the so-called *uniformity* property as a fundamental requirement for a logic programming language. This property guarantees completeness of goal-oriented search for proofs with respect to the underlying logic of intuitionistic type theory. Our language extends the class of *higher-order Hereditary Harrop formulas* of λProlog to include constraints in such a way as to preserve uniformity, as it is shown in detail in [15].

The Constraint System $\mathcal{C}$. The constraints we will consider here belong to a generic system $\mathcal{C}$ that is assumed to satisfy certain conditions. Following [22], we view a *constraint system* as a pair $\mathcal{C} = \langle \mathcal{L}_\mathcal{C}, \vdash_\mathcal{C} \rangle$, where $\mathcal{L}_\mathcal{C}$ is a set of λ-terms of type γ in normal form built up from Σ and V, and $\vdash_\mathcal{C}$ is a binary *entailment relation* between sets of constraints, Γ, and single constraints, C. $\mathcal{C}$ is required to satisfy:

- Every λ-term in normal form of type γ, built up using constraint predicate symbols (of type $\alpha \rightarrow \gamma$), the logical constants $\top^\gamma$, $\bot^\gamma$, $\exists^{(\alpha\rightarrow\gamma)\rightarrow\gamma}$ and optionally, other suitably typed logical constants (such as $\wedge^{\gamma\rightarrow\gamma\rightarrow\gamma}$, $\Rightarrow^{\gamma\rightarrow\gamma\rightarrow\gamma}$, $\forall^{(\alpha\rightarrow\gamma)\rightarrow\gamma}$) is in $\mathcal{L}_\mathcal{C}$.
- All equations $t_1 \approx t_2$ are in $\mathcal{L}_\mathcal{C}$.
- All the inference rules for equality and for those connectives included in $\mathcal{L}_\mathcal{C}$ that are valid in intuitionistic logic are valid inferences in $\vdash_\mathcal{C}$.
- *Compactness:* $\Gamma \vdash_\mathcal{C} C$ holds iff $\Gamma_0 \vdash_\mathcal{C} C$ for some finite $\Gamma_0 \subseteq \Gamma$.
- $\Gamma \vdash_\mathcal{C} C$ implies $\Gamma\sigma \vdash_\mathcal{C} C\sigma$ for every substitution σ.
- If $t_1 \equiv_\lambda t_2$, then $\vdash_\mathcal{C} t_1 \approx t_2$.
- The cut-rule is allowed in $\mathcal{C}$: if $\Gamma' \vdash_\mathcal{C} \Gamma$ and $\Gamma \vdash_\mathcal{C} C$, then $\Gamma' \vdash_\mathcal{C} C$.

An often used example is the constraint system $\mathcal{R}$ of Real-closed Fields: $\mathcal{L}_\mathcal{R}$ is a language with all classical logical connectives including negation, and $\Gamma \vdash_\mathcal{R} C$ holds iff $Ax_\mathcal{R} \cup \Gamma \vdash_\approx C$, where $Ax_\mathcal{R}$ is Tarski's axiomatization of the real numbers [24], and $\vdash_\approx$ is the entailment relation of classical logic with equality.

Now we spell out the syntax of our programming language. The atomic formulas of *hoHH($\mathcal{C}$)*, like those of λProlog's *hoHH* are limited to those formed by application of predicate symbols (symbols of type $\alpha \rightarrow o$) to *positive terms*, in accordance with the following definition.

Definition 1. *The set of* positive terms *consists of all the terms in λ-normal form built up from Σ and V, not containing constraint predicate symbols, nor the logical constants $\bot$ and $\Rightarrow$.*

A positive atomic formula *is an atomic logical formula containing only positive terms.*

We remind the reader that in higher-order logic, logical formulas can appear as subterms of atomic formulas, so this restriction is significant.

Given a generic constraint system $\mathcal{C}$ satisfying the requirements listed above, the syntax of the constraint-enriched formal system *hoHH*($\mathcal{C}$) consists of the following fragment of higher-order logic.

Definition 2. *The set of* definite clauses*, with elements denoted by* D*, and the set of* goals*, with elements denoted by* G*, are sets of formulas, in* λ*-normal form, defined by the following syntactic rules:*

$D := A_r \,|\, D_1 \wedge D_2 \,|\, G \Rightarrow A_r \,|\, \forall x D$

$G := A \,|\, C \,|\, G_1 \wedge G_2 \,|\, G_1 \vee G_2 \,|\, D \Rightarrow G \,|\, C \Rightarrow G \,|\, \exists x G \,|\, \forall x G$

where A *is a positive atomic formula,* A_r *a rigid positive atomic formula. Notice that* $\top$ *and* $\bot$ *are constraints, so they are goals.*

A program Δ *is a finite set of definite clauses (just called "clauses" for the rest of the paper).*

Clauses are always terms of type o. Goals that are pure constraint formulas C (which may themselves contain connectives of type e.g. $\gamma \rightarrow \gamma \rightarrow \gamma$) have type γ, but compound goals built up from them using the definition just given, must be of type o. Thus, for instance, depending on the nature of G_1, G_2, a goal of the form $G_1 \wedge G_2$ might be built with $\wedge^{\gamma\rightarrow o\rightarrow o}$, $\wedge^{o\rightarrow\gamma\rightarrow o}$ $\wedge^{\gamma\rightarrow\gamma\rightarrow o}$ or $\wedge^{o\rightarrow o\rightarrow o}$. When type of the logical constants can be deduced from the context, the typing is not shown.

Example 1. Consider the instance *hoHH*($\mathcal{R}$). The following program can be written.

$$\Delta = \{\forall x \forall y (x^2 + y^2 \approx 2 \Rightarrow \mathit{circle}(x, y)), \forall x \forall y (x^2 + 6y^2 \approx 2 \Rightarrow \mathit{ellipse}(x, y)),$$
$$\forall X (X \approx \mathit{circle} \vee X \approx \mathit{ellipse} \Rightarrow \mathit{figure}(X))\}.$$

And the goal

$$G \equiv \exists X_1 \exists X_2 (\mathit{figure}(X_1) \wedge \mathit{figure}(X_2) \wedge \neg(X_1 \approx X_2) \wedge X_1(x, y) \wedge X_2(x, y)).$$

However, the formula $\forall x (\exists y ((y^2 \approx x \vee \mathit{ellipse}(x, y)) \Rightarrow \mathit{circle}(x, y)))$ is not a logical formula of the language *hoHH*($\mathcal{C}$), because it is not a clause, due to the existential quantifier, nor a goal, since the disjunction $y^2 \approx x \vee \mathit{ellipse}(x, y)$ is not allowed in the antecedent of a goal.

The *elaboration* of a program Δ is a mapping from programs to sets of implicative clauses. It is the set $\mathit{elab}(\Delta) = \bigcup_{D \in \Delta} \mathit{elab}(D)$, where $\mathit{elab}(D)$ is defined by the following rules:

$\mathit{elab}(A_r) = \{\top \Rightarrow A_r\}$, $\quad \mathit{elab}(D_1 \wedge D_2) = \mathit{elab}(D_1) \cup \mathit{elab}(D_2)$,

$\mathit{elab}(G \Rightarrow A_r) = \{G \Rightarrow A_r\}$, $\quad \mathit{elab}(\forall x D) = \{\forall x D' \,|\, D' \in \mathit{elab}(D)\}$.

We will assume that any goal, constraint or element of any $\mathit{elab}(\Delta)$ is in normal form.

The Proof Rules. We now give the underlying sequent calculus, $\mathit{ho}\mathcal{UC}$, that makes the collection of $\mathit{program}; \mathit{constraint} \vdash \mathit{goal}$ triples in *hoHH*($\mathcal{C}$) a (nondeterministic) logic programming language. Possible interpreter design along the lines, say of [21] are not discussed here. This proof system combines traditional inference rules with the entailment relation of a generic constraint system $\mathcal{C}$.

Sequents have finite sets of programs and constraints on the left and single goals on the right. $\Delta; \Gamma \vdash_{\mathit{ho}\mathcal{UC}} G$ means the sequent $\Delta; \Gamma \vdash G$ is derivable in $\mathit{ho}\mathcal{UC}$. When

either or Δ or Γ is infinite, we mean the sequent $\Delta'; \Gamma' \vdash G$ is derivable in $ho\mathcal{UC}$ for some finite subsets $\Delta' \subseteq \Delta$ and $\Gamma' \subseteq \Gamma$.

C is called an *answer constraint for G from* Δ when $\Delta; C \vdash_{ho\mathcal{UC}} G$. For instance, in Example (1), $x^2 \approx 2 \wedge y \approx 0$ is an answer constraint. As with many constraint formalisms, constraints built up progressively on the left during a bottom-up construction of a proof of a given goal constitute the *output* of this programming language, considerably extending the expressive power of conventional logic programming, where outputs are restricted to equations of the form $variable = term$.

The set of rules of this proof system appears in Figure 1.

$$\frac{\Gamma \vdash_{\mathcal{C}} C}{\Delta; \Gamma \vdash C} \ (C_R) \qquad \frac{\Delta; \Gamma \vdash \exists \overline{x}((A'_r \approx A_r) \wedge G')}{\Delta; \Gamma \vdash A_r} \ (Clause)\ (*), \text{where}$$

$$\forall \overline{x}(G' \Rightarrow A'_r) \text{ is } \alpha\text{-equivalent to a formula of } elab(\Delta)$$

$$\frac{\Delta; \Gamma \vdash F \quad \Gamma \vdash_{\mathcal{C}} X \approx t}{\Delta; \Gamma \vdash X t_1 \dots t_n} (Flex),\ F \equiv \Lambda((X\, t_1 \dots t_n)[t/X]), \mathsf{fv}(t) \subseteq \mathsf{fv}(\overline{t}), t \text{ positive}$$

$$\frac{\Delta; \Gamma \vdash G_i}{\Delta; \Gamma \vdash G_1 \vee G_2} \ (\vee_R)\ (i = 1, 2) \qquad \frac{\Delta; \Gamma \vdash G_1 \quad \Delta; \Gamma \vdash G_2}{\Delta; \Gamma \vdash G_1 \wedge G_2} \ (\wedge_R)$$

$$\frac{\Delta, D; \Gamma \vdash G}{\Delta; \Gamma \vdash D \Rightarrow G} \ (\Rightarrow_R) \qquad \frac{\Delta; \Gamma, C \vdash G}{\Delta; \Gamma \vdash C \Rightarrow G} \ (\Rightarrow_{C_R})$$

$$\frac{\Delta; \Gamma, C \vdash G[y/x] \quad \Gamma \vdash_{\mathcal{C}} \exists y C}{\Delta; \Gamma \vdash \exists x G} \ (\exists_R)(*), \qquad \frac{\Delta; \Gamma \vdash G[y/x]}{\Delta; \Gamma \vdash \forall x G} \ (\forall_R)(*)$$

(*) $\overline{x}$, y do not appear free in the sequent of the conclusion.

Fig. 1. $ho\mathcal{UC}$ Sequent Rules

In all rules except (C_R), the principal formula is not a constraint. This means that any connective introduced by the rules must have target type o (and not γ).

This calculus is similar to those defined for higher-order formulas in the literature (see e.g. [18]), but the presence of constraints induces some modifications. The (λ) rule, that transforms formulas by λ-conversion, is not needed in $ho\mathcal{UC}$ because every formula in a sequent of a proof is in λ-normal form. Note that, save for the $(Flex)$ rule *no substitution of a compound term for a variable is made during the application of the rules*, illustrating what might be viewed as a fundamental slogan for this calculus: *constraints are generalized terms*. The burden is shifted from terms to potentially more expressive predicates in the constraint system. This is perhaps best exemplified in the $(\exists_R)$ rule, discussed more at length in [15,16], by use of which substitutions can be simulated by constraints C on the left which are inhabited, i.e. those for which a proof of $\exists y C$ can be found. This may just mean replacing a substitution $[t/x]$ in conventional logic programming by an equality constraint of the form $x \approx t$. However, it is considerably more powerful, because constraints allow for a more general description of a potential witness for an existentially quantified formula where a specific term might not

exist. For example, in $\mathcal{R}$ the constraint $(x * x \approx 2)$ may represent $\sqrt{2}$, which is not a legal term.

The use of constraints also broadens the scope of backchaining by means of the combination of the $(Clause)$ and $(\exists_R)$ rules. Inspection of the $(Clause)$ rule shows we are not required, as in conventional logic programming, to unify the head of the selected clause with the atomic goal to be solved, but rather to solve a new existentially quantified goal that, by the use just discussed of the $(\exists_R)$ rule, will result in a search for a *constraint* that implies equality of the atomic goal and the clause head. The $(Clause)$ rule is not applied to flexible atoms, instead flexible atoms are managed with the $(Flex)$ rule, which permits non-atomic instantiation of the predicate variable.

Proposition 1. *The following rules are admissible in $ho\mathcal{UC}$ (if the premises are derivable, the conclusion is derivable).*

$$\frac{\Delta; \Gamma, C[y/x] \vdash G}{\Delta; \Gamma, \exists x C \vdash G}\ (\exists_{\mathcal{C}_L})\,(*) \qquad \frac{\Delta; \Gamma, C \vdash G \quad \Gamma \vdash_{\mathcal{C}} C}{\Delta; \Gamma \vdash G}\ (cut_{\mathcal{C}}) \qquad \frac{\Delta; \Gamma \vdash G[t/x]}{\Delta; \Gamma, y \approx t \vdash G[y/x]}\ (Subst)\,(*)$$

where the condition $(*)$ means $y \notin \mathsf{fv}(\Delta, \Gamma, G, \exists x C, t)$, t positive.

Proof. By the induction on the length of the derivation, analyzing cases according to the last rule applied, and using the properties of the relation $\vdash_{\mathcal{C}}$. □

In fact, as shown in [15], the proof system $ho\mathcal{UC}$ is equivalent, with antecedents and consequents restricted to the executable *hoHH($\mathcal{C}$)* fragment, to the extended calculus $ho\mathcal{IC}$ (higher-order Intuitionistic Calculus over $\mathcal{C}$) which includes the full intuitionistic theory of types with constraints. $ho\mathcal{IC}$ therefore manipulates not necessarily positive terms, has rules introducing connectives in the left, and a simple axiom for dealing with atoms, instead of $(Clause)$. That equivalence means that, for any program Δ, for any set of constraints Γ, and for any goal G: $\Delta; \Gamma \vdash_{ho\mathcal{IC}} G \iff \Delta; \Gamma \vdash_{ho\mathcal{UC}} G$. Therefore *hoHH($\mathcal{C}$)* satisfies the so-called uniformity property and can be considered as an abstract logic programming language in the sense defined in [18]. In practical terms this means that a search for a proof restricted to an operational interpretation of the connectives does not sacrifice any theorems.

Positive-atomic Generated Formulas. These formulas constitute a subset of formulas in Church's theory of types that plays an special role in the definition of the Kripke semantics for our logic programming language. For these formulas (that include any *hoHH($\mathcal{C}$)* formula) a well-ordering can be defined which allows an induction argument in the proof of completeness of $ho\mathcal{UC}$.

Definition 3. *A formula in Church's theory of types is called* positive-atomic generated *or just* PA-generated*, if it is built up using logical constants from positive atomic formulas and constraints, and it is in normal form.*

Definition 4. *Let F be a PA-generated formula. We define the* non-positive depth *of F, $\delta(F)$, to be the length of the longest path from the root node of the parse tree of F to any occurrence of implication. Inductively:*

If F is positive or a constraint then $\delta(F) = 0$, otherwise
$\delta(F_1 \Diamond F_2) = 1 + max(\delta(F_1), \delta(F_2))$*, where* $\Diamond \in \{\wedge, \vee, \Rightarrow\}$,
$\delta(QxF) = 1 + \delta(F)$*, where Q is $\forall$ or $\exists$.*

This measure induces a well-founded order on the set of PA-generated logical formulas.

Lemma 1. *For any PA-generated formula F and any positive term t, $\delta(F[t/x]) = \delta(F)$. If $F_1 \Diamond F_2$, QxF are non-positive PA-generated logical formulas, then:*

i) $\delta(F_i) < \delta(F_1 \Diamond F_2)$, *for* $i = 1, 2$.
ii) $\delta(F[t/x]) < \delta(QxF)$ *for any positive term* t.

We finish this section with an example that illustrates some of the expressive power of *hoHH($\mathcal{C}$)*, when it is used to formalize inductive inference. Critical use is made of the availability of universal quantification in goals to specify induction conclusions and of nested implication in Hereditarily Harrop clauses to capture induction hypotheses. In addition, the presence of (arithmetic) constraints reduces the difficulty of many induction proofs, transferring some of the burden of proof to the constraint solver.

Example 2. Consider the instance *hoHH($\mathcal{N}$)*, i.e. using the equational theory of the natural numbers as our constraint system.

The predicate *even* can be defined by the program clauses below, where the operator $+$ is managed by the constraint system.

$$even(0), \forall\, x((even(x+2) \vee (x \geq 2 \wedge even(x-2))) \Rightarrow even(x)).$$

In a proof of the property $\forall x \forall y(\mathit{even}(x) \wedge \mathit{even}(y) \Rightarrow \mathit{even}(x+y))$, the induction step corresponds to the resolution of the goal:

$$\forall x(\forall y(\mathit{even}(x) \wedge \mathit{even}(y) \Rightarrow \mathit{even}(x+y)) \Rightarrow \forall y(\mathit{even}(x+2) \wedge \mathit{even}(y) \Rightarrow \mathit{even}((x+2)+y))).$$

Applying the ($\Rightarrow_R$) rule in reverse (i.e. the so-called *augment* rule of λProlog) the clause $D \equiv \forall y(\mathit{even}(x) \wedge \mathit{even}(y) \Rightarrow \mathit{even}(x+y))$, corresponding to the induction hypothesis, is added to the program as a local clause. Then it will be used during subsequent deduction steps, in particular in the proof of the subgoal $\mathit{even}((x+2)+y)$.

An interesting feature of such a deduction using a mix of constraints and logic is that since $+$ is a constraint operator, the search tree will be considerably pruned. For instance, during the proof, it is the constraint solver that checks the satisfiability of certain constraints such as $\forall x \forall y \exists x_1(x + x_1 \approx (x+2)+y)$. In addition, the usual search problem of the choice of the variable on which induction is done is irrelevant here. The proof is also successful if y is chosen instead of x, because in this case, the constraint solver will deal with $\forall y \forall x \exists x_1(x_1 + y \approx x + (y+2))$, in the same way as before.

3 Higher-Order Kripke Semantics

We first fix conventions and notation for the elementary model theory of the simply typed λ-calculus and the notion of an applicative structure indexed over a partially ordered set. Next we will define Kripke models for the full underlying logic (Church's Intuitionistic Theory of Types) without constraints that requires indexing models of the λ-calculus as well. Then our Kripke models are modified to deal with the the fragment *hoHH($\mathcal{C}$)*. Constraints and formulas including constraints must be interpreted and additional conditions must be imposed. The need for Kripke semantics arises from the existence of intuitionistic connectives in our logic.

3.1 Semantic Preliminaries

We start by recalling the definition of a model of the typed λ-calculus.

When considering indexed families $S = \{S_k\}, T = \{T_k\}$ of sets, we will say $f : S \rightarrow T$ is an *indexed function* if in fact f itself is a family of functions, indexed over the same set as S, T which *respects the indexed structure*, that is to say $f = \{f_k : S_k \rightarrow T_k\}$.

Definition 5. *A* Typed Applicative Structure (TAS), $\mathsf{D} = \langle D, \mathsf{App}, \mathsf{Const}\rangle$, *is given by:*

- *a type-indexed family of sets* $D = \{D^\alpha | \alpha \in \mathsf{Ty}\}$, *each member of which,* D^α, *is called the* carrier *for the type* α,
- *a family of functions* $\mathsf{App} = \{\mathsf{App}^{\alpha\beta} : D^{\beta\alpha} \times D^\alpha \rightarrow D^\beta | \alpha, \beta \in \mathsf{Ty}\}$, *and a type preserving indexed family of assignment functions* $\mathsf{Const} = \{\mathsf{Const}^\alpha : \Sigma^\alpha \rightarrow D^\alpha | \alpha \in \mathsf{Ty}\}$, *where* $\Sigma^\alpha \subseteq \Sigma$ *is the set of constants of type* α.

Definition 6. *Let* D *be a TAS. A* D-environment η *is a function from the set of variables into* D *which respects types.*

Definition 7. *Given a typed applicative structure* $\mathsf{D} = \langle D, \mathsf{App}, \mathsf{Const}\rangle$, *a* D-environmental model $[\![\ ]\!]$ *consists of an indexed family* $\{[\![\ ]\!]_\eta | \eta$ *a* D*-environment*$\}$ *of* total *functions from the terms into* D, *respecting types, for which the following hold, for any* D*-environment* η:

$$
\begin{array}{lcl}
[\![c]\!]_\eta & = & \mathsf{Const}(c), \quad \text{for constants } c,\\
[\![x]\!]_\eta & = & \eta(x), \quad \text{for variables } x,\\
[\![(t_1 t_2)]\!]_\eta & = & \mathsf{App}([\![t_1]\!]_\eta, [\![t_2]\!]_\eta),\\
[\![\lambda x^\alpha . t^\beta]\!]_\eta & = & d', \text{ where } d' \in D^{\beta\alpha}, d' \text{ is the unique element such that for any } d \in D^\alpha,\\
& & \mathsf{App}(d', d) = [\![t^\beta]\!]_{\eta[x:=d]}, \text{ where } \eta[x := d] \text{ is the } \mathsf{D}\text{-environment}\\
& & \text{coinciding with } \eta, \text{ save on } x, \text{ where its value is } d.
\end{array}
$$

A model *is a triple* $\langle \mathsf{D}, [\![\ ]\!], \eta\rangle$, *where* D *is a TAS,* $[\![\ ]\!]$ *is a* D*-environmental model and* η *is a* D*-environment.*

Note that existence and uniqueness of the d' denoting $\lambda x.t$ in environment η is imposed (following [19]) as part of the definition. The condition is quite strong: it ensures the substitution lemma (below and in [19]). It also guarantees uniqueness of an interpretation for a given environment, as is easily shown by induction on term structure. For this reason, when the existence of such a $[\![\ ]\!]_\eta$ is clear from context we will refer to the model together with its environment as the pair $\langle \mathsf{D}, \eta\rangle$.

3.2 Kripke Models for Church's Intuitionistic Higher Order Logic

One of the most widely used semantics for intuitionistic logic was introduced by Kripke (1963). In Tarski models for classical logic, one must supply a domain and interpretations for function, relation and constant symbols. Kripke models, however consist of a partially ordered collection of such domains, together with certain compatibility conditions between them. Perhaps the best way to visualize such a semantics is to think of a Kripke model as a function defined on a poset $(W, \leq)$, associating with each world $w \in W$ a domain D_w together with interpretations of the language.

Definition 8. *Let* $(W, \leq)$ *be a partially ordered set. A* $(W, \leq)$-indexed typed applicative structure *is a family* $\mathcal{D} = \{\mathsf{D}_w | w \in W\}$, *where for each* $w \in W$, $\mathsf{D}_w = \langle D_w, \mathsf{App}_w, \mathsf{Const}_w\rangle$ *is a typed applicative structure. For each* $w \leq w' \in W$ *the following conditions must be satisfied:*

- *Monotonicity:* $D_w \subseteq D_{w'}$.
- $\mathsf{App}_w(f, d) = \mathsf{App}_{w'}(f, d)$*, for any pair* (f, d) *(of the corresponding type) in* D_w.
- $\mathsf{Const}_w(c) = \mathsf{Const}_{w'}(c)$*, for any* $c \in \Sigma$.

Now we define the so-called forcing relation, $\Vdash$ between members w of W and certain members of D_w^o. We may think of $\Vdash$ as a partial function mapping such pairs, when defined, to $true$ or $false$. Note that, unlike conventional Kripke models, forcing is defined *entirely within the semantics*, that is to say as a relation between worlds and *denotations* of formulas, rather than syntactic formulas themselves. Since logical formulas are terms in higher-order logic, we must supply both a denotation and a truth value for formulas of type o. We are thus able to deal with a problem mentioned in the introduction, namely to allow formulas with the same truth values in all models to have different denotations.

Definition 9. *A* Kripke applicative structure *for Church's intuitionistic theory of types is a quadruple* $\mathcal{K} = \langle W, \leq, \mathcal{D}, \Vdash \rangle$*, where:*

$(W, \leq)$ *is a poset.*
$\mathcal{D} = \{\mathsf{D}_w | w \in W\}$ *is a* $(W, \leq)$*-indexed typed applicative structure.*
$\Vdash$ *is a binary* forcing *relation between worlds* $w \in W$ *and logical elements* d *in* D_w^o *(written* $w \Vdash d$*), satisfying:*

- *The* monotonicity requirement, *if* $d \in D_w^o$ *and* $w \Vdash d$ *then for any* $w' \in W$ *with* $w' \geq w$ *we have* $w' \Vdash d$.
- *The* logical conditions*, where* $d_1 \cdot d_2$ *abbreviates* $\mathsf{App}_w(d_1, d_2)$*, and the underlined symbols* $\underline{c}$ *denote the interpreted logical constants* $\mathsf{Const}_w(c)$*, for the appropriate world* w*:*

  1. $w \Vdash \underline{\top}$ *always,* *2.* $w \Vdash \underline{\bot}$ *never,*
  3. $w \Vdash \underline{\wedge} \cdot d_1 \cdot d_2$ *iff* $w \Vdash d_1$ *and* $w \Vdash d_2$,
  4. $w \Vdash \underline{\vee} \cdot d_1 \cdot d_2$ *iff* $w \Vdash d_1$ *or* $w \Vdash d_2$,
  5. $w \Vdash \underline{\Rightarrow} \cdot d_1 \cdot d_2$ *iff for any* $w' \geq w$ *if* $w' \Vdash d_1$*, then* $w' \Vdash d_2$,
  6. $w \Vdash \underline{\exists} \cdot f^{o\alpha}$ *iff for some* $d \in D_w^\alpha$*,* $w \Vdash f \cdot d$,
  7. $w \Vdash \underline{\forall} \cdot f^{o\alpha}$ *iff for every* $w' \geq w$ *and* $d \in D_{w'}^\alpha$*,* $w' \Vdash f \cdot d$.

Several features of this definition are different from its first order counterpart. Firstly, the forcing relation (viewed as a truth-valued function) is *partial*: $w \Vdash d$ need not be defined for all members of D_w^o. In particular, note that there is no atomic case, since the individuals on the right of the forcing relation are not syntactic formulas, but rather denotations in the carrier of type o. Because of the impredicativity of higher-order logic, a Kripke applicative structure is not necessarily uniquely determined by an atomic assignment of truth at each world (taking *atoms* to mean denotations in D_w^o of atomic formulas). Also monotonicity of forcing must be imposed by definition on all formulas at once.

Definition 10. *Let* $\mathcal{K}$ *be a Kripke applicative structure. A* $\mathcal{K}$-environment η *is a family* $\{\eta_w | w \in W\}$ *of* D_w*-environments satisfying the following coherence property, for each variable* x *and each pair* $w, w' \in W$ *with* $w \leq w'$*:* $\eta_w(x) = \eta_{w'}(x)$.

We can now extend the notion of environmental model to Kripke applicative structures along the lines of Definition 7.

Definition 11. *Given a Kripke applicative structure $\mathcal{K} = \langle W, \leq, \mathcal{D}, \Vdash \rangle$, a* Kripke environmental model *for $\mathcal{K}$, or a $\mathcal{K}$-*interpretation *is an indexed family $\{ [\![\ ]\!]_\eta |\ \eta$ a $\mathcal{K}$-environment$\}$ where for each world $w \in W$, $[\![\ ]\!]_{\eta_w}$ is a* total *function from the set of terms into D_w, respecting types, and which, for each η_w, satisfies the conditions of Definition 7. For instance, $[\![x]\!]_{\eta_w} = \eta_w(x)$, and $[\![(t_1 t_2)]\!]_{\eta_w} = \mathsf{App}_w([\![t_1]\!]_{\eta_w}, [\![t_2]\!]_{\eta_w})$.*

Given a Kripke environmental model for $\mathcal{K}$ we will define the $\mathcal{K}$*-interpretation of a term t* over a $\mathcal{K}$-environment η at the world w to be $[\![t]\!]_{\eta_w}$.

Putting the whole package together, we can define our Kripke semantics.

Definition 12. *A* Kripke model *$\langle \mathcal{K}, [\![\ ]\!], \eta \rangle$ for Church's intuitionistic theory of types is given by a Kripke applicative structure $\mathcal{K}$, a Kripke environmental model $[\![\ ]\!]$ for $\mathcal{K}$ and a $\mathcal{K}$-environment η. Furthermore, for each formula F and world w, $w \Vdash [\![F]\!]_{\eta_w}$ is defined (true or false).*

As before, the notation can be simplified to $\langle \mathcal{K}, \eta \rangle$, as $[\![\ ]\!]_\eta$ is uniquely induced.

Now we are able to interpret the intuitionistic formulation of Church's logic into our semantics in a straightforward manner.

Definition 13. *Let $\mathcal{K} = \langle W, \leq, \mathcal{D}, \Vdash \rangle$, $\langle \mathcal{K}, [\![\ ]\!], \eta \rangle$ be a Kripke model, and let F be a logical formula, i.e. a term of type o. Then we say F is* forced *at w (or true at w) with environment η, and write $w \Vdash_\eta F$, whenever $w \Vdash [\![F]\!]_{\eta_w}$. If F is forced at every $w \in W$ in this environment, we write $\mathcal{K} \models_\eta F$. If either property holds in the presence of* all *$\mathcal{K}$-environments, then we write $w \Vdash F$ or, respectively, $\mathcal{K} \models F$ and say that $\mathcal{K}$ models or* satisfies *F (or that F is true in $\mathcal{K}$).*

The previous definitions are extended to finite sets of formulas S, in the natural way. For instance, $w \Vdash_\eta S$, means $w \Vdash_\eta F$ for all $F \in S$. In addition we will say that $\langle \mathcal{K}, \eta \rangle$ models or *satisfies* a sequent $\Delta; \Gamma \vdash G$ when for any world w, if $w \Vdash_\eta \Delta$ and $w \Vdash_\eta \Gamma$ then $w \Vdash_\eta G$.

The whole of intuitionistic type theory can be proved to be sound and complete with respect to this Kripke semantics. Since our interest is to adapt our semantics to the logic programming formalism *hoHH($\mathcal{C}$)* we will restrict attention to soundness and completeness for that case. First we will need to extend these definitions to include constraint systems, and modify the logical conditions.

3.3 Kripke Models for *hoHH($\mathcal{C}$)*

In our language, since we are thinking of the constraint system as a generic black box, about which we want to say as little as possible, instead of additional structural properties we add a global requirement of soundness with respect to constraint deductions, and preservation of congruence properties of $\approx$. Since the formalization of constraints in Church's type theory only requires the added presence of a reserved type γ of constraints, and for each type α an equality relation symbol $\approx_{\gamma\alpha\alpha}$ in the language, there is nothing to add to the basic framework save interpretations for any new constant symbols and a new forcing relation between worlds w and the carriers D^γ_w of the constraints.

We represent the new forcing relation with the same symbol as logical forcing, since, as with the proof theory, we can always tell which one we are using by inspecting the types of the terms present.

However, in the fragment *hoHH($\mathcal{C}$)*, only positive terms are allowed in atomic formulas. Thus it is sufficient to define the forcing relation $w \Vdash_\eta F$, for PA-generated formulas F, which include both goals and clauses. The relevance that positive terms have in the syntax of *hoHH($\mathcal{C}$)* will be also reflected in the semantics by defining a *semantic counterpart* to the set of positive terms. This means defining, for each type α and world w, a subset $D_w^{\alpha+} \subseteq D_w^\alpha$, where positive terms of type α must be interpreted.

Definition 14. *A uniform* $\mathcal{C}$-Kripke model for *hoHH($\mathcal{C}$) is a triple* $\langle \mathcal{K}, [\![\ ]\!], \eta \rangle$, *where:*

- $\mathcal{K} = \langle W, \leq, \mathcal{D}, \Vdash \rangle$ *satisfies the requirements of Definition 9 with the following changes:*
  - *For each type α and world w, there is a distinguished subset $D_w^{\alpha+}$ of D_w^α (written D_w^+ when the type is not relevant).*
  - *The forcing relation $\Vdash$ is extended to a relation between W and $D^o \cup D^\gamma$, and it is defined for (at least) the members of $D^o \cup D^\gamma$ corresponding to $[\![F]\!]$, for all PA-generated formulas F.*
    As for the logical conditions of this definition, Conditions 6 and 7 of the definition of models for the full theory of types are restricted to members of $D_w^{\alpha+}$:
    6'. $w \Vdash \underline{\exists} \cdot f^{o\alpha}$ iff for some $d \in D_w^{\alpha+}$, $w \Vdash f \cdot d$,
    7'. $w \Vdash \underline{\forall} \cdot f^{o\alpha}$ iff for every $w' \geq w$ and $d \in D_{w'}^{\alpha+}$, $w' \Vdash f \cdot d$.
- $[\![\ ]\!]$ *is a $\mathcal{K}$-interpretation and η a $\mathcal{K}$-environment, such that for any $w \in W$, if t^α is a positive term, then $[\![t]\!]_{\eta_w} \in D_w^{\alpha+}$. As a consequence $\eta_w(x) \in D_w^{\alpha+}$, for every variable x.*
- *In addition $\Vdash$ must satisfy the following* $\mathcal{C}$-conditions *for every $w \in W$:*
  - $\mathcal{C}$-soundness*: For every Γ, C, if $\Gamma \vdash_\mathcal{C} C$ then, if $w \Vdash_\eta \Gamma$ then $w \Vdash_\eta C$.*
  - Congruence*:*
    (a) For every A_r, A'_r, such that $w \Vdash_\eta A_r \approx A'_r$, if $w \Vdash_\eta A_r$, then $w \Vdash_\eta A'_r$.
    (b) For every flexible atom $X t_1 \ldots t_n$ and positive term t, such that $w \Vdash_\eta X \approx t$, if $w \Vdash_\eta \Lambda((X t_1 \ldots t_n)[t/X])$, then $w \Vdash_\eta (X t_1 \ldots t_n)$.
  - $\mathcal{C}$-existential condition*: For every $w \in W$, $r \in D_w^{\gamma\alpha}$,*
    $w \Vdash \underline{\exists}^{\gamma(\gamma\alpha)} \cdot r \Longleftrightarrow$ for some $d \in D_w^{\alpha+}, w \Vdash r \cdot d$.

We will repeatedly make use of the following technical consequence of these definitions, whose proof is by a straightforward induction on λ-term structure.

Lemma 2 (Substitution). *Let $\langle \mathcal{K}, [\![\ ]\!], \eta \rangle$ be a uniform $\mathcal{C}$-Kripke model. For any positive term t, any PA-formula F, and any world w, $[\![F[t/x]]\!]_{\eta_w} = [\![F]\!]_{\eta_w[x := [\![t]\!]_{\eta_w}]}$.*

4 Soundness and Completeness of *hoHH($\mathcal{C}$)*

Since the language *hoHH($\mathcal{C}$)* is based on the calculus $ho\mathcal{UC}$, our aim is to prove the equivalence between provability in $ho\mathcal{UC}$ and validity in every uniform $\mathcal{C}$-Kripke model.

4.1 Soundness of $ho\mathcal{UC}$

We begin by showing that what is provable in $ho\mathcal{UC}$ is true.

Theorem 1 (Soundness). *For every Δ, Γ, G, if $\Delta; \Gamma \vdash_{ho\mathcal{UC}} G$ holds, then the sequent $\Delta; \Gamma \vdash G$ is satisfied in every uniform $\mathcal{C}$-Kripke model for hoHH($\mathcal{C}$).*

Proof. Let $\langle \mathcal{K}, \eta \rangle$ be a uniform $\mathcal{C}$-Kripke model for *hoHH($\mathcal{C}$)*. The proof proceeds by induction on the length of the proof of the sequent $\Delta; \Gamma \vdash G$. The inductive hypothesis is that all sequents with shorter proofs are satisfied at every world in every uniform $\mathcal{C}$-Kripke model. We consider the the most interesting case of the induction here, and leave the rest as an exercise for the reader.

Let w be any world such that $w \Vdash_\eta \Delta$ and $w \Vdash_\eta \Gamma$. If $\Delta; \Gamma \vdash_{ho\mathcal{UC}} A_r$ is derived using the $(Clause)$ rule as a final step, then there is a variant $\forall \overline{x}(G' \Rightarrow A'_r)$ of a clause of $elab(\Delta)$, such that the sequent $\Delta; \Gamma \vdash \exists \overline{x}((A'_r \approx A_r) \wedge G')$ has a proof shorter than the proof of $\Delta; \Gamma \vdash A_r$. By the induction hypothesis, $w \Vdash_\eta \exists \overline{x}((A'_r \approx A_r) \wedge G')$. Then there are $\overline{d} \in D^+_w$ such that $w \Vdash_{\eta[\overline{x}:=\overline{d}]} A'_r \approx A_r$ and $w \Vdash_{\eta[\overline{x}:=\overline{d}]} G'$. It is easy to prove that $w \Vdash_\eta \Delta$ implies $w \Vdash_\eta \forall \overline{x}(G' \Rightarrow A'_r)$, then $w \Vdash_{\eta[\overline{x}:=\overline{d}]} G' \Rightarrow A'_r$. We have $w \Vdash_{\eta[\overline{x}:=\overline{d}]} A'_r$, because $w \Vdash_{\eta[\overline{x}:=\overline{d}]} G'$. We conclude $w \Vdash_\eta A_r$, because $\overline{x}$ are not free in A_r, $w \Vdash_{\eta[\overline{x}:=\overline{d}]} A'_r \approx A_r$, and by the congruence of $\langle \mathcal{K}, \eta \rangle$. □

4.2 Completeness

The proof of the completeness is based on the construction of a particular uniform $\mathcal{C}$-Kripke model, $\mathcal{U}^{\mathcal{C}}$, in such a way that $\Vdash$ coincides with $\vdash_{ho\mathcal{UC}}$ *when the latter is defined*. We are not able to completely define $\Vdash$ this way because provability of a sequent $\Delta; \Gamma \vdash G$ in $ho\mathcal{UC}$ only makes sense when G is a goal, whereas the relation $\Vdash$ must be defined for more general (PA-generated) formulas.

To define the model, we will need to make use of a restricted version of the so-called Lindenbaum Lemma for the constraint system $\mathcal{C}$.

The Lindenbaum Construction for $\mathcal{C}$. In its original form, in classical logic, the Lindenbaum lemma (see e.g. [25]) states that a consistent set of sentences can be extended to a maximal consistent set. In our setting, to prove completeness, we only need to ensure that constraint theories satisfy a pure-variable form of the existential part of this claim, namely that if a formula A is not derivable from a theory Γ then the theory can be extended to one that still does not prove A and has the existence property: if it derives an existential formula, it proves a pure-variable instance over a language enriched only with new variables, but with no new constants.

In the following we will assume that X is a complete set of variables, by which we mean that it contains countably many variables $x_1^\alpha, x_2^\alpha \ldots$ for each type expression α.

Definition 15. *A set of constraints Γ is said to be* $\exists$-saturated *over a complete set of variables X if for any constraint C, whenever $\Gamma \vdash_{\mathcal{C}} \exists x C$ then for some y in X, we have $\Gamma \vdash_{\mathcal{C}} C[y/x]$.*

Lemma 3. *Let V be a complete set of variables (assumed to be the base set of variables used to build terms in this paper), and let X be a disjoint complete set of variables. For any Δ, Γ and G, with free variables in V, if $\Delta; \Gamma \vdash G$ is not derivable in $ho\mathcal{UC}$, then there is an extension $\hat{\Gamma}$ of Γ, with free variables in $V \cup X$, which:*

– *is* $\exists$*-saturated over* $V \cup X$*, and*
– *maintains* $\Delta; \hat{\Gamma} \not\vdash_{ho\mathcal{UC}} G$.

Some adaptation is required to make the proof of Lindenbaum lemma [25] suitable for *hoHH($\mathcal{C}$)*. In particular, instead of adding Henkin *constants* as witnesses for existential formulas, fresh *variables* are added. This is needed in the completeness theorem because of the special character of quantifier rules where variables and constraints, rather than terms, act as witnesses. Otherwise, the proof is straightforward.

The Uniform $\mathcal{C}$-Kripke Model $\mathcal{U}^{\mathcal{C}}$. We begin now the construction of the model we will use to establish completeness. We start with a countable sequence of countable complete sets of fresh variables $X_0 \subset X_1 \subset \cdots \subset X_n \subset \cdots$ where each $X_{i+1} \setminus X_i$ is countably infinite.

Definition 16. *Given a constraint system* $\mathcal{C}$ *we define* $\mathcal{U}^{\mathcal{C}} = \langle W, \leq, \mathcal{D}, \Vdash \rangle$*, as follows:*

– $(W, \leq)$*, the ordered set of worlds is defined as:*
  $W = \{\langle \Delta, \Gamma, n\rangle \,|\, \Delta$ *is a finite set of clauses over* Σ *and* X_n*;* Γ *is a set of constraints over* Σ *and* X_n*,* $\exists$*-saturated over* $X_n\}$.
  $\langle \Delta_1, \Gamma_1, n_1\rangle \leq \langle \Delta_2, \Gamma_2, n_2\rangle \overset{\text{def}}{\Longleftrightarrow} \Delta_1 \subseteq \Delta_2, \Gamma_1 \subseteq \Gamma_2$ *and* $n_1 \leq n_2$.
– $\mathcal{D} = \{\mathsf{D}_w | w \in W\}$.
  For each $w = \langle \Delta, \Gamma, n\rangle$, D_w *is defined as follows:*
  D^α_w *is the set of open* λ*-terms in normal form of type* α *over* Σ *and the set of variables* X_n.
  $D^{\alpha+}_w$ *is the subset consisting of the positive terms of* D^α_w.
  $\mathsf{Const}_w(c) = c, \quad c \in \Sigma. \qquad \mathsf{App}_w(t_1, t_2) = \Lambda(t_1 t_2)$.
– *The relation* $\Vdash$ *is defined for the elements of* $D^o_w \cup D^\gamma_w$ *that are PA-generated formulas. In order to define* $\langle \Delta, \Gamma, n\rangle \Vdash F$*, we use induction on the non-positive depth* $\delta(F)$*:*
  (1) If F is a constraint or a positive logical formula ($\delta(F) = 0$*), then*
  $\langle \Delta, \Gamma, n\rangle \Vdash F \overset{\text{def}}{\Longleftrightarrow} \Delta; \Gamma \vdash_{ho\mathcal{UC}} F$.
  This case includes pure constraints, and rigid and flexible atoms.
  (2) For PA-formulas F *that do not satisfy the preceding condition (*$\delta(F) > 0$*),* $\langle \Delta, \Gamma, n\rangle \Vdash F$ *is defined according to the definition of* $\Vdash$ *for uniform* $\mathcal{C}$*-Kripke models. For instance,*
  $\langle \Delta, \Gamma, n\rangle \Vdash \forall x^\alpha F \overset{\text{def}}{\Longleftrightarrow}$ *for every* $\langle \Delta', \Gamma', n'\rangle \in W$*,* $\langle \Delta, \Gamma, n\rangle \leq \langle \Delta', \Gamma', n'\rangle$*, and every* $t \in D^{\alpha+}_{\langle \Delta', \Gamma', n'\rangle}$*,* $\langle \Delta', \Gamma', n'\rangle \Vdash \mathsf{App}_{\langle \Delta', \Gamma', n'\rangle}(\lambda x.F, t)$*, i.e.,* $\langle \Delta', \Gamma', n'\rangle \Vdash F[t/x]$.

Note that the relation $\Vdash$ in this model is defined by induction on the non-positive depth of formulas, with positive and constraints formulas as the base case. In this way, we avoid problems with impredicativity, by working with a well-founded order. When a quantified non-positive formula is instantiated with positive terms, the instance is simpler with respect to that order, in accordance with Lemma 1.

We will show that when $\mathcal{U}^{\mathcal{C}}$ is supplied with a particular environment, it gives rise to a uniform $\mathcal{C}$-Kripke model for *hoHH($\mathcal{C}$)*.

For $\mathcal{U}^{\mathcal{C}}$, given any environment η there is a unique induced environmental model $[\![\ ]\!]$, satisfying, for all worlds w, the condition $[\![t]\!]_{\eta_w} = t\theta_{\eta_w}$, where θ_{η_w} is the substitution mapping each x free in t to $\eta_w(x)$. Let id be the *identity environment*: for every $w = \langle \Delta, \Gamma, n \rangle \in W$, id_w maps each variable $x \in X_n$ to itself, and let $[\![\ ]\!]$ be the induced environmental model. We will prove that $\langle \mathcal{U}^{\mathcal{C}}, id \rangle$ is a uniform $\mathcal{C}$-Kripke model for *hoHH($\mathcal{C}$)*.

We first establish some technical properties of the relation $\Vdash$, that defines $\mathcal{U}^{\mathcal{C}}$.

Lemma 4. *For every world $\langle \Delta, \Gamma, n \rangle$, and every clause D over Σ and X_n, if for all $D' \in elab(D)$, $\langle \Delta, \Gamma, n \rangle \Vdash D'$, then $\langle \Delta, \Gamma, n \rangle \Vdash D$.*

The proof is by induction on the sum of the non-positive depths of the formulas of $elab(D)$.

Lemma 5. *For every world $\langle \Delta, \Gamma, n \rangle$, and every $F \in \Delta \cup \Gamma$, we have $\langle \Delta, \Gamma, n \rangle \Vdash F$.*

Proof. Sketch. The proof of $\langle \Delta, \Gamma, n \rangle \Vdash C$, $C \in \Gamma$, is immediate, because, for all $C \in \Gamma$, $\Gamma \vdash_{\mathcal{C}} C$. In order to prove $\langle \Delta, \Gamma, n \rangle \Vdash D$, $D \in \Delta$, we proceed by induction on the non-positive depth of D. The base case implies that D is positive. It is then easy to show that $\Delta; \Gamma \vdash_{ho\mathcal{UC}} D$, so $\langle \Delta, \Gamma, n \rangle \Vdash D$, by definition. For the inductive case, in order to prove $\langle \Delta, \Gamma, n \rangle \Vdash D$, we first show that $\langle \Delta, \Gamma, n \rangle \Vdash D'$ for every $D' \in elab(D)$, then conclude $\langle \Delta, \Gamma, n \rangle \Vdash D$, using Lemma 4. If $D' \in elab(D)$, then $D' \equiv \forall \overline{x}(G \Rightarrow A_r)$. In order to establish $\langle \Delta, \Gamma, n \rangle \Vdash \forall \overline{x}(G \Rightarrow A_r)$ we need to make use of the following claim:

If $\forall \overline{x}(G \Rightarrow A_r) \in elab(\Delta)$, then for all $\langle \Delta', \Gamma', n' \rangle \in W$, $\langle \Delta, \Gamma, n \rangle \leq \langle \Delta', \Gamma', n' \rangle$ and $\overline{t} \in D^{+}_{\langle \Delta', \Gamma', n' \rangle}$, if $\langle \Delta', \Gamma', n' \rangle \Vdash G[\overline{t}/\overline{x}]$ then, $\Delta'; \Gamma' \vdash_{ho\mathcal{UC}} G[\overline{t}/\overline{x}]$.

From this fact, proving $\langle \Delta, \Gamma, n \rangle \Vdash \forall \overline{x}(G \Rightarrow A_r)$ can be reduced to proving that $\Delta; \Gamma \vdash_{ho\mathcal{UC}} A_r[\overline{t}/\overline{x}]$ (for any $\overline{t}$), if $\Delta; \Gamma \vdash_{ho\mathcal{UC}} G[\overline{t}/\overline{x}]$. But this is easy to prove using $(Subst)$ and $(Clause)$, since $\forall \overline{x}(G \Rightarrow A_r) \in elab(\Delta)$.

The proof of the claim is by induction on $\delta(G[\overline{t}/\overline{x}])$. The base case is trivial. For the inductive step, we must consider the possible structure of $G[\overline{t}/\overline{x}]$. We show here the most interesting case:

$G[\overline{t}/\overline{x}] \equiv D' \Rightarrow G'$: If $\Delta'' = \Delta' \cup \{D'\}$, $\langle \Delta'', \Gamma', n' \rangle \Vdash D'$, applying the outer induction on $\delta(D)$, since $D' \in \Delta''$, and observe that in fact $\delta(D') < \delta(D)$, because if $\forall \overline{x}(G \Rightarrow A_r) \in elab(D)$ and D non-positive, then $\delta(D) > \delta(G) = \delta(G[\overline{t}/\overline{x}])$, by Lemma 1, and $\delta(G[\overline{t}/\overline{x}]) > \delta(D')$. So $\langle \Delta'', \Gamma', n' \rangle \Vdash G'$, since $\langle \Delta', \Gamma', n' \rangle \Vdash D' \Rightarrow G'$ and $\langle \Delta'', \Gamma', n' \rangle \geq \langle \Delta', \Gamma', n' \rangle$. Then, $\Delta', D'; \Gamma' \vdash_{ho\mathcal{UC}} G'$, by the induction on $\delta(G[\overline{t}/\overline{x}])$. Therefore $\Delta'; \Gamma' \vdash_{ho\mathcal{UC}} D' \Rightarrow G'$, according to $(\Rightarrow_R)$. □

Proposition 2. *For all worlds $\langle \Delta, \Gamma, n \rangle$ and goal G, with free variables in X_n:*

If $\langle \Delta, \Gamma, n \rangle \Vdash G$, then $\Delta; \Gamma \vdash_{ho\mathcal{UC}} G$.

Proof. By induction on the non-positive depth of G. The argument is similar to that of the claim established in the proof of Lemma 5. □

Lemma 6. *$\langle \mathcal{U}^{\mathcal{C}}, [\![\ ]\!], id \rangle$ is a uniform $\mathcal{C}$-Kripke model for hoHH($\mathcal{C}$).*

Proof. The requirements of Definition 14 must be proved. It is easy to prove that $(W, \leq)$ is a poset, $\mathcal{D}$ a $(W, \leq)$-indexed *TAS*, $[\![\ ]\!]$ a $\mathcal{U}^{\mathcal{C}}$-interpretation and id a $\mathcal{U}^{\mathcal{C}}$-environment.

In addition $[\![t]\!]_{id_w} = t$, so if t is a positive term, $[\![t]\!]_{id_w} \in D^+_w$. Let us show the requirements for $\Vdash$ are satisfied.

Monotonicity requirement. By induction on the non-positive depth of PA-formulas. For the base case, the monotonicity of $\Vdash$ is derived from the monotonicity of $\vdash_{ho\mathcal{UC}}$ with respect to Δ and Γ. The inductive step is straightforward.

Logical conditions. For PA-formulas that are not constraints neither positive logical formulas, those conditions are satisfied by definition. For constraints and positive formulas, the arguments are straightfoward[1]. Here we establish one of the more delicate cases:

$\underline{\exists x G}$ Suppose $\langle \Delta, \Gamma, n\rangle \Vdash \exists x G$, then by definition of $\Vdash$ and $(\exists_R)$ rule, there is C such that $\Delta; \Gamma, C \vdash_{ho\mathcal{UC}} G[y/x]$ and $\Gamma \vdash_{\mathcal{C}} \exists y C$, where y is not free in $\Delta, \Gamma, \exists x G$. Since Γ is $\exists$-saturated over X_n, then $\Gamma \vdash_{\mathcal{C}} C[z/y]$ for some $z \in X_n$. By the properties of $ho\mathcal{UC}$, $\Delta; \Gamma, C[z/y] \vdash_{ho\mathcal{UC}} G[z/x]$, because y was fresh. But this implies that $\Delta; \Gamma \vdash_{ho\mathcal{UC}} G[z/x]$, from the fact $\Gamma \vdash_{\mathcal{C}} C[z/y]$ and $(cut_{\mathcal{C}})$. Therefore there is $z \in D^+_{\langle \Delta, \Gamma, n\rangle}$, such that $\langle \Delta, \Gamma, n\rangle \Vdash G[z/x]$.

Conversely if there is $t \in D^+_{\langle \Delta, \Gamma, n\rangle}$ such that $\langle \Delta, \Gamma, n\rangle \Vdash G[t/x]$, then $\Delta; \Gamma \vdash_{ho\mathcal{UC}} G[t/x]$ by definition, and $\Delta; \Gamma, y \approx t \vdash_{ho\mathcal{UC}} G[y/x]$, with y fresh, applying $(subst)$. So $\Delta; \Gamma \vdash_{ho\mathcal{UC}} \exists x G$ in accordance with $(\exists_R)$. Hence we can conclude $\langle \Delta, \Gamma, n\rangle \Vdash \exists x G$, because $\exists x G$ is positive and the definition of $\Vdash$.

The proof for the $\mathcal{C}$ *-conditions* are routine, and left to the reader. □

We will refer to our uniform $\mathcal{C}$ -Kripke model simply as $\mathcal{U}^{\mathcal{C}}$. By the definitions of $[\![\ ]\!]$ and id, the notation $\langle \Delta, \Gamma, n\rangle \Vdash_{id} G$ is equivalent to $\langle \Delta, \Gamma, n\rangle \Vdash G$.

Finally, we prove that the formal system $ho\mathcal{UC}$, is complete for $\mathcal{C}$ -Kripke semantics. That means that any $ho\mathcal{UC}$ sequent true in all of our models is derivable.

Theorem 2 (Completeness of *ho*$\mathcal{UC}$). *For every Δ, Γ, G over Σ, and V, if every uniform $\mathcal{C}$ -Kripke model for hoHH($\mathcal{C}$) satisfies the sequent $\Delta; \Gamma \vdash G$, then $\Delta; \Gamma \vdash_{ho\mathcal{UC}} G$.*

Proof. Suppose, for a contradiction, that there are Δ, Γ, G, such that any uniform $\mathcal{C}$ -Kripke model for *hoHH($\mathcal{C}$)* satisfies $\Delta; \Gamma \vdash G$, but there is no $ho\mathcal{UC}$ derivation of the sequent $\Delta; \Gamma \vdash G$. By the Lindenbaum Lemma ([3]), there is a set of constraints, Γ', that extends Γ, $\exists$-saturated over certain X_n, such that there is no $ho\mathcal{UC}$ derivation of the sequent $\Delta; \Gamma' \vdash G$.

So in the model $\mathcal{U}^{\mathcal{C}}$, by Lemma 5, $\langle \Delta, \Gamma', n\rangle \Vdash_{id} \Delta$, and $\langle \Delta, \Gamma', n\rangle \Vdash_{id} \Gamma$. Hence, $\langle \Delta, \Gamma', n\rangle \Vdash_{id} G$, because $\mathcal{U}^{\mathcal{C}}$ satisfies $\Delta; \Gamma \vdash G$, by the hypothesis of the theorem. Then by Proposition 2, $\Delta; \Gamma' \vdash_{ho\mathcal{UC}} G$, contradicting the hypothesis of Γ'. □

Logical Intensionality. As discussed in the first section, one of our aims was to produce a model theory in which logical equivalence of two logical formulas F_1 and F_2 would not necessarily imply validity of $p(F_1) \Rightarrow p(F_2)$ for every predicate symbol p. Take p be a constant of type $o \to o$, A_r a rigid atomic formula, and consider $\mathcal{C}$ for which $\approx$ coincides with $\equiv_\lambda$. In the model $\mathcal{U}^{\mathcal{C}}$, $p(A_r) \Rightarrow p(A_r \wedge A_r)$

[1] Notice that those formulas are always goals.

is *not* forced at the root node $\langle \emptyset, \emptyset, 0 \rangle$, since by Proposition 2, this would mean that $\emptyset; \emptyset \vdash_{ho\mathcal{UC}} p(A_r) \Rightarrow p(A_r \wedge A_r)$, and hence $\{p(A_r)\}; \emptyset \vdash_{ho\mathcal{UC}} p(A_r \wedge A_r)$. Since $\not\vdash_{\mathcal{C}} A_r \approx A_r \wedge A_r$, this is impossible.

5 Conclusion

We have introduced a semantic framework based on Kripke structures for Intuitionistic Higher-Order Type Theory with constraints to model the declarative content of a representative higher-order constraint logic programming language, with simply typed λ-terms, implication and universal quantification in goals. The underlying logic of λ-Prolog is covered as a special case. We have shown the program calculus sound and complete.

We build on Mitchell-Moggi Kripke λ-models [20], but go well beyond equational reasoning, to model predicates in an impredicative higher-order logic with constraints.

Our results extend earlier work on declarative semantics for some executable fragments of the logic: First-order Hereditarily Harrop formulas [17], classical Higher-order Horn formulas in [26,2] and semantics for *hoHH* in [5,14] and [8] for *HH*($\mathcal{C}$).

A key direction for future work is to understand how to adapt the framework defined here to deal with polymorphic types, linear logic or a linear constraint system, or to exploit the constraint framework for specific abstract syntax and metaprogramming applications. It would also be of interest to define Kripke models more sensitive operationally to a specific proof procedure, as well as to study observational equivalence and abstract interpretation in this context, a matter for further research.

Acknowledgements. This work was partially supported by the Spanish projects 'MERIT-FORMS': TIN2005-09207-C03-03, 'PROMESAS-CAM': S-0505/TIC/0407. The first author's research was also partially supported by the Pitney-Bowes Corporation.

The authors wish to thank Olivier Hermant and Frank Pfenning for helpful comments.

References

1. Andrews, P.: Resolution in type theory. Journal of Symbolic Logic, vol. 36(3) (1971)
2. Bai, M., Blair, H.: General model theoretic semantics for higher-order horn logic programming. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, Springer, Heidelberg (1992)
3. Benzmüller, C., Brown, C.E., Kohlhase, M.: Higher order semantics and extensionality. Journal of Symbolic Logic 69, 1027–1088 (2004)
4. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5, 56–68 (1940)
5. DeMarco, M.: Higher Order Logic Programming in the Theory of Types. PhD thesis, Wesleyan University (1999)
6. DeMarco, M., Lipton, J.: Completeness and cut elimination in the intuitionistic theory of types. The Journal of Logic and Computation 15(6), 821–854 (December 2005)
7. Gabbay, D.M., Reyle, U.: N-prolog: an extension of prolog with hypothetical implications i. Journal of Logic Programming 1(4), 319–355 (1984)

8. García-Díaz, M., Nieva, S.: Providing declarative semantics for HH extended constraint logic programs. In: PPDP '04, pp. 55–66. ACM Press, New York (2004)
9. Girard, J., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press, Cambridge (1998)
10. Henkin, L.: Completeness in the theory of types. Journal of Symbolic Logic 15, 81–91 (1950)
11. Hindley, J.R.: Basic simple type theory, New York, USA. Cambridge University Press, Cambridge (1997)
12. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19/20, 503–581 (1994)
13. Jagadeesan, R., Nadathur, G., Saraswat, V.A.: Testing concurrent systems: An interpretation of intuitionistic logic. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 517–528. Springer, Heidelberg (2005)
14. Lastres, E.: A Semantics for Logic Programs based on HH Formulas. PhD thesis, Università di Pisa (2002)
15. Leach, J., Nieva, S.: A higher-order logic programming language with constraints. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 108–122. Springer, Heidelberg (2001)
16. Leach, J., Nieva, S., Rodríguez-Artalejo, M.: Constraint logic programming with hereditary Harrop formulas. Theory and Practice of Logic Programming 1(4), 409–445 (2001)
17. Miller, D.: A logical analysis of modules in logic programming. Journal of Logic Programming, pp. 79–108 (1989)
18. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure. and Applied Logic 51(1-2), 125–157 (1991)
19. Mitchell, J.: Foundations for Programming Languages. MIT Press, Cambridge, Massachusetts (1996)
20. Mitchell, J., Moggi, E.: Kripke-style models for typed lambda calculus. Annals of Pure. and Applied Logic 51, 99–124 (1991)
21. Nadathur, G.: A proof procedure for the logic of hereditary harrop formulas. Journal of Automated Reasoning 11, 115–145 (1993)
22. Saraswat, V.: The category of constraints is cartesian closed. In: Proc. of the 7th Symposium on Logic in Computer Science (LICS '92), pp. 341–345. IEEE Computer Society Press, Los Alamitos (1992)
23. Takahashi, M.: A proof of cut-elimination in simple type theory. J. Math. Soc. Japan 19(4), 399–410 (1967)
24. Tarski, A.: A decision method for elementary algebra and geometry. University of California Press, Berkeley (1951)
25. van Dalen, D.: Logic and Structure. Springer, Heidelberg (2004)
26. Wolfram, D.A.: A semantics for λprolog. Theoretical Computer Science 136, 277–289 (1994)

Predicative Analysis of Feasibility and Diagonalization

Jean-Yves Marion

Nancy Université, Loria, ENSMN-INPL, B.P. 239, 54506 Vandœuvre-lès-Nancy
Cedex, France
Jean-Yves.Marion@loria.fr

Abstract. Predicative analysis of recursion schema is a method to characterize complexity classes like the class of polynomial time functions. This analysis comes from the works of Bellantoni and Cook, and Leivant. Here, we refine predicative analysis by using a ramified Ackermann's construction of a non-primitive recursive function. We obtain a hierarchy of functions which characterizes exactly functions, which are computed in $O(n^k)$ over register machine model of computation. Then, we are able to diagonalize using dependent types in order to obtain an exponential function.

1 Introduction

Predicative analysis of recursion comes from the works of Bellantoni and Cook [2] and Leivant [10]. This analysis is based on a ramification principle on data which is appealing because its concept is simple and purely syntactic. Each element of a computation has a tier, which determines its ability to run a recursion. The ramification principle states that a definition by recursion is ramified only if the tier of the recurrence parameter is strictly higher than the tier of the output. This analysis takes it roots in the paper of Simmons [16] and Leivant [9]. We revisit the ramification principle. The results mentioned above characterize the class of polynomial time computable functions using essentially two tiers of data ramification: one for recursion arguments and one for recursion outputs. In this work, we introduce a strict ramification principle which allows getting a characterization of a polynomial time hierarchy of functions. Functions which are defined with k tiers are exactly functions which are computable in $O(n^k)$ steps. The hierarchy is not robust in the sense that it depends on the model of computation which is a register machine model here. So, the result that we suggest is really about intrinsic complexity of functions in the tradition of the recursion Theory. We have tried to understand the mechanism that underpins the suggested classification. Our analysis shows how functions are defined and how we can jump from one class of functions to another one by strict ramified iteration. This leads us to introduce a double iteration operator, which captures each level of the polynomial time hierarchy $\mathrm{DTIME}(n^k)$ and escapes them. For this, we define an exponential function by a diagonaliszation method, which

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 290–304, 2007.
© Springer-Verlag Berlin Heidelberg 2007

reveals some analogies with Ackermann [1] construction as it is explained in Chapter 7 of Simmons book [17]. The construction that we propose is a kind of double recursion whose main ideas can be explained by considering the following example.
$f : \mathbb{N}(1), \mathbb{N}(0) \rightarrow \mathbb{N}(0)$

$$f(0, y) = y + 1$$
$$f(x + 1, y) = f(x, f(x, y))$$

The function f is defined by nested recursion and satisfies the ramification principle. Indeed, the first argument may be of tier 1 and the second of tier 0. So, the output of f is of tier 0 and f is well typed. However f computes the exponential function : $f(n, m) = 2^n + m$ for all n and m. In $f(x, f(x, y))$, the leftmost occurrence of f calls itself which violates the essence of the ramification principle. Now, we ramify f by assigning to each occurrence of f a tier, and so we obtain the following function sequence.

$$f_0(x, y) = y + 1$$
$$f_{k+1}(0, y) = y$$
$$f_{k+1}(x + 1, y) = f_k(x, f_{k+1}(x, y))$$

where f_1 computes the addition, and f_2 iterates the addition, and so on. We also see that the domain, or the type, of each f_k can be $\mathbb{N}(k), \mathbb{N}(0) \rightarrow \mathbb{N}(0)$. If we transform $(f_k)_{k \in \mathbb{N}}$ sequence of functions into a three place functions $\phi(k, x, y)$, we are able to produce by a diagonalization argument a function which eventually dominates each f_k. The type of ϕ depends of its first argument and so would be $\forall k : \mathbb{N}(k), \mathbb{N}(0) \rightarrow \mathbb{N}(0)$.

This example is just here to illustrate quickly the ideas that we develop in this paper, which is organized as follows. Section 2 presents the computational models and defines DTIME(n^k). Section 3 focuses on tiered recursion and Leivant's characterization of FPTIME. Section 4 gives the characterization of the polynomial time hierarchy, and the last section show how this characterization may be used to construct new classes of functions. We choose except for the last part to present this work in term of tiered function algebra. It is not too difficult to translate this formalism into an applied typed lambda-calculus, like in Simmons survey [15]. (and we partially do it in Section 5.) Lastly, we make this choice because it is more readable and so easier to understand.

2 Computations and a Polynomial Time Hierarchy

2.1 Register Machines

The set of binary words over the alphabet $\{a, b\}$ is $\mathbb{W}$. A register machine, abbreviated RM, works over words of $\mathbb{W}$. A RM consists in

1. an alphabet $\{a, b\}$.
2. a finite set $\mathcal{S} = \{s_0, s_1, \ldots, s_k\}$ of states, including a distinct state BEGIN.

3. a finite list $\mathcal{R} = \{R_1, \ldots, R_m\}$ of registers. Registers store words of $\mathbb{W}$.
4. a finite function LABEL mapping states to commands which are

$R = a(R)$	*add the letter a to R*
$R = b(R)$	*add the letter b to R*
$R = R'$	*assign the value of R' to R*
$R = \text{PRED}(R)$	*remove the first letter of R*
$\text{BRANCH}(R, s_\epsilon, s_a, s_b)$	*switch to the label s_i following the value of R*

A *configuration* of a RM M is given by a pair (s, σ) where s is a state and $\sigma : \mathcal{R} \to \mathbb{W}$ is an environment which stores register values. We guess that the above informal semantics should be enough to understand how register machines work. Throughout, we deal with functions which have a co-arity, that is function whose range is $\mathbb{W}^\beta$ for some β. A function $\phi : \mathbb{W}^\alpha \to \mathbb{W}^\beta$ is computed by a register machine M if for all $u_1, \ldots, u_\alpha$, we have $\phi(u_1, \ldots, u_\alpha) = (v_1, \ldots, v_\beta)$ then the execution of M starting from the initial configuration (BEGIN, σ_0) ends to a configuration (s, σ_f) such that: for $i = 1, \alpha$, $\sigma_0(R_i) = u_i$, otherwise $\sigma_0(R_i) = \epsilon$ and for $j = 1, \beta$, $\sigma_f(R_{m+1-j}) = v_j$.

2.2 A Polynomial Time Hierarchy

The size $|u|$ of a word u is the number of letters of the word. In particular the size of the empty word ϵ is 0. The size of pair of words is inductively defined as follows: $|\langle u, v\rangle_i| = |u| + |v|$ at any tier i.

The time measure corresponds to the number of steps to perform a computation on a register machine. We say that a function $\phi : \mathbb{W}^\alpha \to \mathbb{W}^\beta$ is computable in $O(n^k)$ if the runtime is bounded by $c.(n_1^k + \ldots + n_\alpha^k) + d$ for some c and d and where for each i, n_i is the size of the ith argument. The class DTIME(n^k) is the set of all functions which are computable in $O(n^k)$. The class FPTIME of polynomial time functions is $\cup_k$DTIME(n^k).

In this work, we study the classes DTIME(n^k) which delineates a polynomial hierarchy. It is well known that the class FPTIME is robust, which is not the case for polynomial hierarchies. Indeed, the definition of DTIME(n^k) is not invariant with respect to a large class of the computational models. The reason lies on the fact that the simulation of a computational model by another may have a quadratic cost. For example, the runtime of simulations of two-tape Turing machine by a one-tape Turing machine is quadratic. Such lower bound may be nicely obtained using Kolmogorov complexity. The reader may consult Jones' book [7] for further informations.

3 Ramified Primitive Iterations

3.1 Functions on Tiered Domains

We are interested in computational complexity, that is why we focus immediately on words. The domain of computation is the set $\mathbb{W}$ of words over the alphabet

$\{a, b\}$. It is generated from the empty word function 0 and two successors A and B. As usual $A(B(0))$ is the word ab.

This domain is tiered by duplicating $\mathbb{W}$ into $\mathbb{W}(0), \mathbb{W}(1), \ldots, \mathbb{W}(i), \ldots$ where each $\mathbb{W}(i)$ is an identical copy of $\mathbb{W}$ at tier i. Each domain $\mathbb{W}(i)$ is a set of words over the alphabet $\{a_i, b_i\}$. As previously, there are an empty word function 0_i and two successors A_i and B_i. In practice, we define functions by specifying their values with respect to tiered domain generators.

There are erasing bijections $\kappa_k : \mathbb{W}(k) \to \mathbb{W}$ for each k which just erase the tier of words. For example, we may represent a function $\phi : \mathbb{W} \to \mathbb{W}$ by $f : \mathbb{W}(k) \to \mathbb{W}(0)$ for some tier k if for each $u \in \mathbb{W}(k)$, $\kappa_0(f(u)) = \phi(\kappa_k(u))$. In this case, we shall just write $f(u) = \phi(u)$.

We also consider *downcasting bijections* $\mathit{coerce}_{k+1} : \mathbb{W}(k+1) \to \mathbb{W}(k)$ for each k such that $\mathit{coerce}_{k+1}(0_{k+1}) = 0_k$, $\mathit{coerce}_{k+1}(A_{k+1}(x)) = A_k(\mathit{coerce}_{k+1}(x))$ and $\mathit{coerce}_{k+1}(B_{k+1}(x)) = B_k(\mathit{coerce}_{k+1}(x))$. Similarly, we shall write that $f : \mathbb{W}(k+1) \to \mathbb{W}(0)$ is defined from $h : \mathbb{W}(k+1), \mathbb{W}(k) \to \mathbb{W}(0)$ by $f(x) = h(x, x)$ to mean that $f(x) = h(x, \mathit{coerce}_{k+1}(x))$. *Throughout, we shall reason with respect to erasing bijections and downcasting bijections without explicitly mentioning them.*

We consider functions with co-arity. For this, we construct Cartesian product of domains of same tier. We abbreviate $\mathbb{W}(i)^\alpha$ by $\mathbb{W}(i) \times \ldots \times \mathbb{W}(i)$. We have a pairing function $\langle\ ,\ \rangle_i$ and both projections π_i^1 and π_i^2, for each tier i.

We often leave out some brackets using familiar conventions and hence we abbreviate $\tau_1, \ldots, \tau_n \to \tau$ by $\tau_1 \to (\ldots(\tau_n \to \tau))$. It is also convenient to have a normal presentation of functions. We shall write $f : \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(r)^\beta$ in such a way that $i_1 \geq \ldots \geq i_n \geq r$. We say that the tier of the j*th* argument of f is i_j, and the output tier is r. We write $\boldsymbol{y}$ to mean $y_1, \ldots, y_n$ where y_j is an element of $\mathbb{W}(i_j)^{\alpha_j}$.

Conventions that we have described here will be extended to the typed lambda calculus that we suggest at the end in a natural manner.

3.2 Ramified Primitive Iteration

A function $f : \mathbb{W}(k+1), \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(r)^\beta$ is obtained by *ramified primitive iteration* from the functions $h_\epsilon : \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(r)^\beta$ and $h_a, h_b : \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n}, \mathbb{W}(r)^\beta \to \mathbb{W}(r)^\beta$ if

$$f(0_{k+1}, \boldsymbol{y}) = h_\epsilon(\boldsymbol{y}) \tag{1}$$

$$f(A_{k+1}(x), \boldsymbol{y}) = h_a(\boldsymbol{y}, f(x, \boldsymbol{y})) \tag{2}$$

$$f(B_{k+1}(x), \boldsymbol{y}) = h_b(\boldsymbol{y}, f(x, \boldsymbol{y})) \tag{3}$$

where conditions $k+1 \geq i_j$ for any j and $k \geq r$ hold. We call these last conditions the *ramification principle* based on [10]. The first argument is named the iteration argument and its tier is $k+1$.

3.3 Ramified Arithmetic

In order to compare function growth rate and to illustrate key notions, it is convenient to have an encoding of natural numbers. This encoding will be used in Sections 4.1 and 5.1.

We represent natural numbers by considering both successors A_i and B_i as the same. Hence, we have a single successor that we write S_i, for each tier i. It should be clear that this encoding is non-injective, which is sufficient because we are just interesting in the size of the handling values. So in this representation, a word represents a natural number, which corresponds to its size. Hence, 0_i will refer to zero at tier i, and $S_i(x)$ intuitively increases the size of x by one, which corresponds exactly to the successor operation in unary notation.

We represent in *ramified arithmetic* an arithmetical function $\phi : \mathbb{N}^\alpha \to \mathbb{N}$ by a function $f : \mathbb{W}(i_1), \ldots, \mathbb{W}(i_\alpha) \to \mathbb{W}(r)$ if

$$\phi(n_1, \ldots, n_\alpha) = |f(u_1, \ldots, u_\alpha)| \quad \text{for each } u_i \text{ such that } |u_i| = n_i \text{ and } i = 1, \alpha$$

Now, we can define below the addition add_k and the multiplication mul_k at tier k.

$add_k : \mathbb{W}(k+1), \mathbb{W}(k) \to \mathbb{W}(k)$ and $|add_k(u, v)| = |u| + |v|$, for all u and v.

$$\begin{aligned} add_k(0_{k+1}, y) &= y \\ add_k(S_{k+1}(x), y) &= S_k(add_k(x, y)) \qquad \text{where } S_i = A_i, B_i \end{aligned}$$

$mul_k : \mathbb{W}(k+1), \mathbb{W}(k+1) \to \mathbb{W}(k)$ and $|mul_k(u, v)| = |u|.|v|$, for all u and v

$$\begin{aligned} mul_k(0_{k+1}, y) &= 0_k \\ mul_k(S_{k+1}(x), y) &= add_k(y, mul_k(x, y)) \end{aligned}$$

We define polynomials by composition from tiered addition and multiplication, as it is illustrated below.

$cube_k : \mathbb{W}(k+2) \to \mathbb{W}(k)$

$$cube_k(x) = mul_k(x, mul_{k+1}(x, x))$$

We see that we compute the arithmetical function x^3 by composing two multiplications. However, two copies of the multiplication mul_k and mul_{k+1} at different tiers are necessary. Notice also that the tier of the first argument is lower, which is possible because of the use of a downcasting bijection. Actually, we may define $coerce_k$ by a simple ramified iteration.
$coerce_k : \mathbb{W}(k+1) \to \mathbb{W}(k)$

$$\begin{aligned} coerce_k(0_{k+1}) &= 0_k \\ coerce_k(S_{k+1}(x)) &= S_k(coerce_k(x)) \end{aligned}$$

We may then use it instead of the implicit downcasting.

3.4 Leivant's Characterization of FPTIME

Here, we use ramified iteration to facilitate the explanation, but we could also introduce ramified recursion. Nevertheless, it is quite elegant to follow [10] by introducing a particular kind of recursion.

A function $f : \mathbb{W}(r), \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(r)$ is obtained by *flat recursion* from the functions $h_\epsilon : \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(r)$ and $h_a, h_b : \mathbb{W}(r), \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_1} \to \mathbb{W}(r)$ if

$$f(0,\boldsymbol{y}) = h_\epsilon(\boldsymbol{y}) \tag{4}$$

$$f(A_r(x), \boldsymbol{y}) = h_a(x, \boldsymbol{y}) \tag{5}$$

$$f(B_r(x), \boldsymbol{y}) = h_b(x, \boldsymbol{y}) \tag{6}$$

This kind of recursion should be viewed as a mere action on the pattern of the recursive argument. Hence and unlike the ramified principle, the tier of a recurrence argument is not strictly higher that the output tier. The use of flat recursion is essential to define a predecessor over $\mathbb{W}$ and conditional functions.

Definition 1. *A function f is in $\mathcal{L}_\omega(\mathbb{W})$ if it is obtained by a finite number of applications of composition, flat recursion and ramified primitive iteration beginning with basic functions 0_k, A_k, B_k, $\langle _, _ \rangle_k$, π^1_k and π^2_k for each tier k.*

Leivant demonstrated in [10] the following result:

Theorem 1. *The class of functions $\mathcal{L}_\omega(\mathbb{W})$ is exactly the class FPTIME of the functions which are polynomial time computable.*

In this presentation we use functions with co-arity, unlike Leivant which introduces simultaneous ramified iteration.

Actually, Leivant also showed that only two tiers are sufficient. More generally,

Corollary 1. *Let $\mathcal{L}_k(\mathbb{W})$ be the class of functions restricted over $\mathbb{W}(0), \ldots, \mathbb{W}(k)$. For each k, the class of functions $\mathcal{L}_{k+1}(\mathbb{W})$ is exactly the class FPTIME of the functions which are polynomial time computable.*

In the same paper, Leivant shows how to capture DTIME(n^k) by counting the degree of nested iterations. So, he hasn't a calculus to characterize DTIME(n^k).

3.5 Other Approaches

The seminal work of Bellantoni and Cook [2] is similar to the one we have described. They characterize FPTIME by defining a function algebra in which functions have two kind of arguments: the normal ones which can be used as iteration parameters and the safe ones which can not be used as iteration parameters.

As we have seen, only two tiers are necessary to characterize FPTIME. Actually, this is also the essence of the characterization by simply typed lambda

calculus of [12]. The tier 1 arguments are represented by Church-numerals, and the tier 0 are represented by constant terms of atomic type on which no recursion can be made.

Lastly, we can not end this short paragraph by mentioning the pioneer work of Cobham [5].

4 Strict Ramified Primitive Iterations

We present the notion *strict ramified primitive iteration* which is central in this study. A function $f : \mathbb{W}(k), \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(0)^{\beta}$ is obtained by *k-ramified iteration* from the functions
$h_\epsilon : \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n} \to \mathbb{W}(0)^{\beta}$ and
$h_a, h_b : \mathbb{W}(i_1)^{\alpha_1}, \ldots, \mathbb{W}(i_n)^{\alpha_n}, \mathbb{W}(0)^{\beta} \to \mathbb{W}(0)^{\beta}$ if

$$f(0_k, \boldsymbol{y}) = h_\epsilon(\boldsymbol{y}) \tag{7}$$
$$f(A_k(x), \boldsymbol{y}) = h_a(\boldsymbol{y}, f(x, \boldsymbol{y})) \tag{8}$$
$$f(B_k(x), \boldsymbol{y}) = h_b(\boldsymbol{y}, f(x, \boldsymbol{y})) \tag{9}$$

where the inequalities between tiers $k > i_j$ for each j and $k > 0$ hold. We call this last condition *the strict ramification principle.*

Definition 2. *A function f is in $\mathcal{I}_k(\mathbb{W})$ if it is obtained by a finite number of applications of composition, flat recursion and i-ramified iteration, beginning with basic functions 0_i, A_i, B_i, $\langle\ ,\ \rangle_i$, π_i^1 and π_i^2 for each tier $i \leq k$.*

In particular, a function $\mathcal{I}_0(\mathbb{W})$ is not defined by iteration. The notion of 1-ramified iteration was underlying in [13].

4.1 Strict Ramified Arithmetic

We use the same encoding of natural numbers that the one we present in Section 3.3 on ramified arithmetic. However, we slightly modify the way that we represent arithmetical functions to take into account the fact that outputs are of tier 0.

An arithmetical function $\phi : \mathbb{N}^\alpha \to \mathbb{N}$ is represented in *strict ramified arithmetic* by a function $f : \mathbb{W}(i_1), \ldots, \mathbb{W}(i_\alpha) \to \mathbb{W}(0)$ if

$$\phi(n_1, \ldots, n_\alpha) = |f(u_1, \ldots, u_\alpha)| \quad \text{for each } u_i \text{ such that } |u_i| = n_i,\ i = 1, \alpha$$

The addition function defined in Section 3.3 is defined by 1-ramified iteration, setting $k = 0$. On the other hand, the definition of the multiplication proposed in 3.3 does not satisfy the strict ramification principle because both arguments are of the same tier.

Nevertheless, we can define any polynomial. For this, we present first a sequence $(F_k)_{k\in\mathbb{N}}$ of 3-placed monotonic functions, which shall play a *crucial role.* For this, let $g : \mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\alpha$.

$F_0 : \mathbb{W}(0), \mathbb{W}(0), \mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\alpha$

$$F_0(t, x, y) = g(y)$$

$F_{k+1} : \mathbb{W}(k+1), \mathbb{W}(k), \mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\alpha$

$$F_{k+1}(0_{k+1}, x, y) = y$$
$$F_{k+1}(S_{k+1}(t), x, y) = F_k(x, x, F_{k+1}(t, x, y))$$

It is worth noticing that $(F_k)_{k \in \mathbb{N}}$ is parameterized by the function g.

Proposition 1. *For any u,v and w, we have*

$$F_{k+1}(u, v, w) = g^{m.n^k}(w) \qquad \text{where } m = |u| \text{ and } n = |v|$$

The sequence of functions $(F_k)_k$ allows us to define polynomial length iterators over $\mathbb{W}(0)$.

Lemma 1. *Let $P[X]$ be a polynomial of degree k. There is a function $\tilde{P}$: $\mathbb{W}(k), \mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\alpha$ in $\mathcal{I}_k(\mathbb{W})$ such that for each x and y,*

$$\tilde{P}(x, y) = g^{P(|x|)}(y) \tag{10}$$

Proof. The proof is done by induction on the degree of the polynomial. The base case is trivial. Suppose that the degree of P is $k+1$. Hence, $P(x) = c.x^{k+1} + Q(x)$ where the degree of Q is less or equal to k. Suppose that $\tilde{Q}$ satisfies the induction hypothesis wrt Q. We define T^c_{k+1} by composition as follows

$$T^0_{k+1}(x, y) = \tilde{Q}(x, y)$$
$$T^{d+1}_{k+1}(x, y) = F_{k+1}(x, x, T^d_{k+1}(x, y)) \qquad d < c$$

We set $\tilde{P}(x, y) = T^c_{k+1}(x, y)$. We show by an induction on c that $\tilde{P}(x, y)$ satisfies 10.

$$\begin{aligned}\tilde{P}(x, y) &= T^{d+1}_{k+1}(x, y) = F_k(x, x, T^d_{k+1}(x, y)) \\ &= F_k(x, x, g^{d.n^{k+1}+Q(n)}(y)) \qquad \text{where } n = |x| \\ &= g^{n.n^k}(g^{d.n^{k+1}+Q(n)}(y)) = g^{(d+1).n^{k+1}+Q(n)}(y)\end{aligned}$$

□

Proposition 2. *Any polynomial P is represented in strict ramified arithmetics.*

Proof. We set $f(x) = \tilde{P}(x, 0_0)$ in which we replace g by the successor A_0 or B_0. □

We say that a multivariate polynomial $P[X_1, \ldots, X_n]$ with n distinct variables is simple if each monomial of P is of the form $c.X_i^d$ for some constants c and d. For example $2x^2 + 3.y^2 + 4y$ is simple, but $2yx^2 + y$ is not. The degree of a simple polynomial is the greatest exponent of P's variables.

Lemma 2. *Let $P[X_1,\ldots,X_n]$ be a simple polynomial of degree k. There is a function $\tilde{P}:\mathbb{W}(k)^n,\mathbb{W}(0)^\alpha\to\mathbb{W}(0)^\alpha$ in $\mathcal{I}_k(\mathbb{W})$ such that for each $x_1,\ldots,x_n$, and y,*

$$\tilde{P}(x_1,\ldots,x_n,y)=g^{P(|x_1|,\ldots,|x_n|)}(y) \tag{11}$$

Proof. The proof is done by induction on the number of variables. The base case is a consequence of Lemma 1. Suppose that the simple polynomial P has $n+1$ variables $X_1,\ldots,X_n,X_{n+1}$. Since P is simple, we write it as the sum $P(X_1,\ldots,X_n,X_{n+1})=P'(X_1,\ldots,X_n)+P''(X_{n+1})$. Suppose that $\tilde{P'}$ ($\tilde{P''}$) satisfies the induction hypothesis wrt P' (resp. P''). We define $\tilde{P}$ by

$$\tilde{P}(x_1,\ldots,x_{n+1})=\tilde{P'}(x_1,\ldots,x_n,\tilde{P''}(x_{n+1},y))$$

Indeed, we have

$$\begin{aligned}\tilde{P}(x_1,\ldots,x_{n+1})&=\tilde{P'}(x_1,\ldots,x_n,g^{P''(|x_{n+1}|)}(y))=g^{P'(|x_1|,\ldots,|x_n|)}(g^{P(|x_{n+1}|)}(y))\\&=g^{P'(|x_1|,\ldots,|x_n|)+P''(|x_{n+1}|)}(y)=g^{P(|x_1|,\ldots,|x_{n+1}|)}(y)\end{aligned}$$

□

4.2 A Polynomial Time Hierarchy

Theorem 2. *The set of functions $\mathcal{I}_k(\mathbb{W})$ is exactly DTIME(n^k).*

The demonstration of Theorem 2 is a consequence of Lemma 3 and 4 below.

Lemma 3. *Let $\phi:\mathbb{W}^\alpha\to\mathbb{W}^\beta$ be a function which is computable by a register machine M in time $(c.\sum_{i=1,\alpha}n_i^k)+d$ for some constants c,d and k, where n_i is the size of the ith argument. Then, there is a function $f:\mathbb{W}(k)^\alpha\to\mathbb{W}(0)^\beta$ of $\mathcal{I}_k(\mathbb{W})$ such that for each u, $f(u)=\phi(u)$.*

Proof. A configuration of M is given by a $m+1$-uplet of $\mathbb{W}(0)$ which encodes the state and the value of the m registers of M. Then, it is not difficult to design a function *next* : $\mathbb{W}(0)^{m+1}\to\mathbb{W}(0)^{m+1}$, which given a configuration, produces the next configuration wrt M. Definition details of *next* are tedious, so we skip them.

Now, we have to iterate *next* wrt the polynomial time bound. For this we use Lemma 2 since it is a simple polynomial. Therefore, there is a function *loop* : $\mathbb{W}(k)^\alpha$, $\mathbb{W}(0)^{m+1}\to\mathbb{W}(0)^{m+1}$ such that $loop(x_1,\ldots,x_\alpha,y)=g^{(c.\sum_{i=1,\alpha}|x_i|^k)+d}(y)$. We conclude by taking $g=next$. □

Lemma 4. *Assume that $f:\mathbb{W}(k_1)^{\alpha_1},\ldots,\mathbb{W}(k_n)^{\alpha_n}\to\mathbb{W}(r)^\beta$ is in $\mathcal{I}_k(\mathbb{W})$. Then there is a polynomial P of degree k, or less, such that for any $u_1,\ldots,u_n$, the computation of $f(u_1,\ldots,u_n)$, on register machines, is performed in time bounded by $P(\max\{|u_i|\mid$ where the tier of u_i is greater than 0, $k_i>0\}_{i=1,n})$*

Proof. The proof goes by induction on k. Suppose that $f\in\mathcal{I}_0(\mathbb{W})$. In this case, the definition of $\mathcal{I}_0(\mathbb{W})$ claims that f is not defined by strict ramified iteration.

Hence, it is not hard to compute f in constant time. In particular, this case includes all the cases where the output tier r is strictly greater than 0.

Now, suppose that $f \in \mathcal{I}_{k+1}(\mathbb{W})$. There are two main cases that we are considering below.

First, f is obtained by $k+1$-ramified iteration. We compute a loop whose length is bounded by the length of the first argument u_1. We begin by evaluating $v_0 = h_\epsilon(u_2, \ldots, u_n)$. Next we compute $h_\alpha(u_2, \ldots, u_n, v_0)$ where α is the last letter of u. And, we repeat this process till we have consumed all letters of the iteration argument u_1. As usual with tiering system, the key point is that the runtime of the auxiliary functions h_a and h_b does not depend on tier 0 values. Hence we associate three polynomials P_ϵ, P_a and P_b satisfying the induction hypothesis. The runtime of f is bounded by $P_\epsilon(\max\{|u_i| \mid \text{where } k_i > 0\}) + |u_1| \times \max_{\alpha=a,b}(P_\alpha(\max\{|u_i| \mid \text{where } k_i > 0\}))$. Since h_ϵ,h_a, and h_b have domains which have strictly lower tiers than $k+1$, it follows that the degree of the corresponding polynomials, P_ϵ, P_a and P_b is at most k by induction hypothesis. As a consequence, there is a polynomial which bounds $P_\epsilon(X) + X.\max_{\alpha=a,b}(P_\alpha(X))$ of degree at most $k+1$. This polynomial is an upper bound on f's runtime.

Second, f is defined by composition. Say that $f(\boldsymbol{x}) = h(\boldsymbol{x}, g(\boldsymbol{x}))$. There are two cases to consider. The first is when the output tier of g is 0. In this case, the runtime of f is bounded by the sum of the runtime of g and h. The second is when the output tier of g is strictly greater than 0. Then, the runtime of g is constant because g can not be defined by iterations. It follows that the runtime of f is bounded by the runtime of h plus an additive constant (due to g). $\square$

5 Diagonalization with Dependent Tiers

5.1 Jumping Outside

We shall now take another point of view and focus on growth rate of functions. At a given tier k, we can iterate the function F_k by composing with $\mathbb{W}(0)$ arguments. For this, we introduce an operator $\Delta[F_k] : \mathbb{W}(k+1), \mathbb{W}(k), \mathbb{W}(0) \to \mathbb{W}(0)$ which is defined by

$$\begin{aligned} \Delta[F_k](0_{k+1}, x, y) &= y \\ \Delta[F_k](S_{k+1}(r), x, y) &= F_k(x, x, \Delta[F_k](r, x, y)) \end{aligned}$$

We say that the rth iterate of F_k is $\Delta[F_k](r)$.

Next, we define a 4-placed operator Δ^ω based on a double iteration. It is a nested iteration based on lexicographic ordering.

$\Delta^\omega : \mathbb{N}, \mathbb{W}(k+1), \mathbb{W}(k), \mathbb{W}(0) \to \mathbb{W}(0)$

$$\begin{aligned} \Delta^\omega(0, 0_{k+1}, x, y) &= g(y) && g : \mathbb{W}(0) \to \mathbb{W}(0) \\ \Delta^\omega(k+1, 0_{k+1}, x, y) &= y \\ \Delta^\omega(k+1, S_{k+1}(r), x, y) &= \Delta^\omega(k, x, x, \Delta^\omega(k+1, r, x, y)) \end{aligned}$$

Here, Δ^ω is parameterized implicitly by g.

Fact 3. *For all* k,r,x *and* y, *we have*

$$\Delta^\omega(k, r, x, y) = F_k(r, x, y)$$
$$\Delta^\omega(k+1, r, x, y) = \Delta[F_k](r, x, y)$$

If we fix the first argument k, we iterate on tier 0 the function F_k. Now, if we fix the second argument r, we jump from tier to tier which allows to get outside each function set $\mathcal{I}_k(\mathbb{W})$, computing $\Delta[F_k](r)$.

Proposition 4. *For any tier* k, *and for each function* $f : \mathbb{W}(k), \mathbb{W}(0) \to \mathbb{W}(0)$ *in* $\mathcal{I}_k(\mathbb{W})$, f *is dominated by* $\Delta^\omega(k, r)$ *for some* r *and some function* g. *That is, for any* x *and* y, *we have* $|f(x, y)| \leq |\Delta^\omega(k, r, x, y)|$.

So, Δ^ω allows us to jump outside each $\mathcal{I}_k(\mathbb{W})$ for any k.

Proposition 5. *The* 4 *placed function* Δ^ω *is not in* $\cup_{k\in\mathbb{N}}\mathcal{I}_k(\mathbb{W})$.

Proof. We set $\phi(x) = \Delta^\omega(|x|, x, x, x)$ for all x. We see that $|\phi(u)| \geq |u|^{|u|+1} + |u|$ which is clearly not in $\cup_{k\in\mathbb{N}}\mathcal{I}_k(\mathbb{W})$ in which each function is polynomially bounded as it has been established in Theorem 2. □

The operator Δ^ω is not in $\cup_{k\in\mathbb{N}}\mathcal{I}_k(\mathbb{W})$. Therefore, Δ^ω is not a ramified function. However, we may see that intuitively the "domain" depends on the first argument, and so we should write $\Delta^\omega : \forall k \in \mathbb{N}.\mathbb{W}(k+1), \mathbb{W}(k), \mathbb{W}(0), \to \mathbb{W}(0)$. To formalize this idea, we now introduce a typed lambda-calculus with very restricted dependent types.

5.2 A Typed Lambda-Calculus

The aim of this last section is to present a typed lambda-calculus in which each function of $\mathcal{I}_k(\mathbb{W})$ for any k is representable from an iterator **diag** which mimics Δ^ω.

Types are built up from an atomic type ω and a unary predicate $\mathbf{W}$ using $\forall$, $\to$ and $\times$ formation. Terms are obtained by lambda-abstraction and application from the following constants:

- Terms of type ω are built up from variables of type ω, $\mathbf{0} : \omega$ and $\mathbf{S} : \omega \to \omega$. A natural number k is represented by $\underline{k}$:

$$\underline{0} = \mathbf{0} \qquad \underline{x+1} = \mathbf{S}(\underline{x})$$

- At each tier k, the predicate $\mathbf{W}$ is inhabited by terms which are generated by $\boldsymbol{\epsilon} : \forall k.\mathbf{W}(k)$ and $\mathbf{A}, \mathbf{B} : \forall k.\mathbf{W}(k) \to \mathbf{W}(k)$. A word u of $\mathbb{W}$ is represented by $\underline{u}_k$:

$$\underline{\epsilon}_k = \boldsymbol{\epsilon}(\underline{k}) \qquad \underline{a(x)}_k = \mathbf{A}(\underline{k}, \underline{x}_k) \qquad \underline{b(x)}_k = \mathbf{B}(\underline{k}, \underline{x}_k)$$

- The pairing construction is obtained by using $\langle _, _ \rangle_{_} : \forall k.\mathbf{W}(k), \mathbf{W}(k) \to \mathbf{W}(k) \times \mathbf{W}(k)$ and projections $\boldsymbol{\pi}^1, \boldsymbol{\pi}^2 : \forall k.\mathbf{W}(k) \times \mathbf{W}(k) \to \mathbf{W}(k)$.

- We have a flat recursion operator $\mathbf{flat} : \tau, (\mathbf{W}(k) \to \tau)^2, \mathbf{W}(k) \to \tau$
- Lastly, we have a double iteration operator

$$\mathbf{diag} : (\mathbf{W}(\mathbf{0})^\alpha \to \mathbf{W}(\mathbf{0})^\alpha) \to \forall k.\mathbf{W}(\mathbf{S}(k)), \mathbf{W}(k), \mathbf{W}(\mathbf{0})^\alpha \to \mathbf{W}(\mathbf{0})^\alpha$$

The one step (contextual) reduction $\triangleright$ is defined by
β-reduction

$$(\lambda x.M)N \triangleright M[x \leftarrow N]$$

projections

$$\boldsymbol{\pi}^1(\langle M, N\rangle_k) \triangleright M$$
$$\boldsymbol{\pi}^2(\langle M, N\rangle_k) \triangleright N$$

flat recursion

$$\mathbf{flat}(h_\epsilon, h_a, h_b, \boldsymbol{\epsilon}(k)) \triangleright h_\epsilon$$
$$\mathbf{flat}(h_\epsilon, h_a, h_b, \mathbf{A}(k, x)) \triangleright h_a x$$
$$\mathbf{flat}(h_\epsilon, h_a, h_b, \mathbf{B}(k, x)) \triangleright h_b x$$

double iteration where $\mathbf{J} = \mathbf{A}, \mathbf{B}$

$$\mathbf{diag}(g, \mathbf{0}, \boldsymbol{\epsilon}, x, y) \triangleright gy$$
$$\mathbf{diag}(g, \mathbf{S}(k), \boldsymbol{\epsilon}, x, y) \triangleright y$$
$$\mathbf{diag}(g, \mathbf{S}(k), \mathbf{J}(r), x, y) \triangleright \mathbf{diag}(g, k, x, x, \mathbf{diag}(g, \mathbf{S}(k), r, x, y))$$

The transitive closure of $\triangleright$ is $\triangleright^*$. Here $M[x \leftarrow N]$ means the usual substitution of all free occurrences of x in M by N.

A term M is of type τ, that we write $M : \tau$, if there is a derivation of $\vdash M : \tau$ following the typing rules of Figure 1. We also add weakening rules. In a judgment of the form $\Gamma \vdash M : \tau$, Γ is a set of variables type assignment. This system has the Church-Rosser and strong normalization properties, which can be established by translating it in the system λP of [18], chapter 10.

Let $\phi : \mathbb{W}^\alpha \to \mathbb{W}^\beta$. The function ϕ is represented at tier k if there is a term $M : \mathbf{W}(\underline{k+1})^\alpha \to \mathbf{W}(\mathbf{0})^\beta$ such that for all u of $\mathbb{W}^\alpha$.

$$M\underline{u}_{k+1} \triangleright^* \underline{\phi(u)}_0$$

Notice, there is a shift of one between function tier and the first argument of $\mathbf{W}$, because of the uniform type of $\mathbf{diag}$, for which the type of the base case is different from the domain of F_0. We define $\mathcal{CI}_k(\mathbb{W})$ as the set of functions which are represented at tier k.

Lemma 5. *Each function $g : \mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\beta$ in $\mathcal{I}_0(\mathbb{W})$ is represented at tier* 0*, and is in $\mathcal{CI}_0(\mathbb{W})$.*

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \text{Axiom}$$

$$\frac{}{\Gamma \vdash \mathbf{c}:\tau} \text{where } \mathbf{c} \text{ is a constant of type } \tau$$

$$\frac{\Gamma, x:\tau \vdash M:\sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma} \to \text{intro, where } x \text{ does not occur in } M$$

$$\frac{\Gamma \vdash M:\tau \to \sigma \qquad \Gamma \vdash N:\tau}{\Gamma \vdash MN:\sigma} \to \text{elim}$$

$$\frac{\Gamma, x:\omega \vdash M:\tau}{\Gamma \vdash M:\forall x.\tau} \forall \text{ intro, and } x \text{ not free in } \Gamma$$

$$\frac{\Gamma \vdash M:\forall x.\tau \qquad \Gamma \vdash k:\omega}{\Gamma \vdash Mk:\tau[x \leftarrow k]} \forall \text{ elim}$$

$$\frac{\Gamma \vdash M:\mathbf{W}(\mathbf{S}(x))}{\Gamma \vdash M:\mathbf{W}(x)} \text{Downcasting}$$

Fig. 1. Typing rules

Proof. The proof is done by induction on the definition of g. □

Proposition 6. *For each k, F_k is in a function in $\mathcal{CI}_k(\mathbb{W})$.*

Proof. Given a function $g : \mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\alpha$, it is represented by a term M at tier 0 of type $\mathbb{W}(0)^\alpha \to \mathbb{W}(0)^\alpha$. F_k is then represented by $\mathbf{diag}(M, \underline{k})$. □

As a direct consequence of the above Proposition, we have a result which is analogous to Lemma 2:

Corollary 2. *Let $P[X_1, \ldots, X_n]$ be a simple polynomial of degree k. There is a term $\mathbf{P} : \mathbf{W}(k+1)^n, \mathbf{W}(0)^\alpha \to \mathbf{W}(0)^\alpha$ such that for each $x_1, \ldots, x_n$, and y,*

$$\mathbf{P}(x_1, \ldots, x_n,) \rhd^* M^{P(|x_1|, \ldots, |x_n|)}(y)$$

where $M : \mathbf{W}(0)^\alpha \to \mathbf{W}(0)^\alpha$ is a term of $\mathcal{CI}_0(\mathbb{W})$.

Proof. The construction of the term $\mathbf{P}$ follows closely the lines of the demonstrations of Lemma 1 and 2. $\mathbf{P}$ is obtained by composition from F_k functions, which are representable at tier k following the above proposition. □

Theorem 3. *The set of functions $\mathcal{I}_k(\mathbb{W})$ is exactly the set $\mathcal{CI}_k(\mathbb{W})$, that is the class DTIME(n^k).*

Proof (Sketch of proof). First, we establish that DTIME$(n^k) \subseteq \mathcal{CI}_k(\mathbb{W})$. For this, observe that the transition function *next*, which is defined in the proof of Lemma 3, is represented at tier 0, by a term of type $\mathbf{W}(0)^{m+1} \rightarrow \mathbf{W}(0)^{m+1}$. We iterate *next* by defining a function *loop*, again as in the proof of Lemma 3, thanks to Corollary 2.

Conversely, we show that $\mathcal{CI}_k(\mathbb{W}) \subseteq \mathcal{I}_k(\mathbb{W})$. Consider a normal derivation ∇ of $\vdash M : \mathbf{W}(\underline{k})^\alpha \rightarrow \mathbf{W}(\underline{0})^\beta$. There is no application of the introduction rule for the universal quantifier $\forall$. Consider a judgment $\Gamma \vdash N : \tau$ of ∇. The typing context Γ contains only declaration of the form $x : \mathbf{W}(k)$. So, we interpret $\Gamma \vdash N : \tau$ as a function. Then, we prove by induction on the normal derivation that this function is in $\mathcal{I}_k(\mathbb{W})$.

The proof is complete by Theorem 2. $\square$

Let $\phi : \mathbb{W}^\alpha \rightarrow \mathbb{W}^\beta$. The function ϕ is represented at tier ω if there is a term $M : \forall k.\mathbf{W}(k)^\alpha \rightarrow \mathbf{W}(\mathbf{0})^\beta$ such that for all u,

$$M\ \underline{k}\ \underline{u}_k \rhd^* \underline{\phi(u)}_0 \qquad\qquad \text{where } k = |u|$$

We define $\mathcal{CI}_\omega(\mathbb{W})$ as the set of functions which are represented at tier ω.

Proposition 7. *There is a function represented at tier ω which is not representable at tier k. In other words, this functions is not $\mathcal{I}_k(\mathbb{W})$.*

Proof. The function Δ^ω is representable at tier ω. $\square$

5.3 Other Ways to Jump and Conclusions

We have presented a manner of constructing an exponential function by diagonalizing functions defined by strict ramified iteration. There are other approaches. In [11], Leivant ramifies the system T of Gödel [6] by introducing an atomic type constructor $\Omega(\tau)$ which allows to perform recursion over type τ terms. Thus, he obtains a characterization of FPTIME and of the elementary functions. Bellantoni and Niggl [3] characterized the Grzegorczyk hierarchy starting from the class FPTIME. For this, they define a rank function which, roughly speaking, is a bound on the number of nested recursion. The work of Caporaso, Covino and Pani seems also related to the research presented in this paper, see [4].

The construction that we suggest could be applied to characterize other complexity classes, like NC^k or classes in between Ptime and Pspace, and to define and study operators which allow to jump from a complexity class to another one. Lastly, it seems that the notion of strict ramified iteration is related to some logical systems derived from linear logic and that some arguments, in particular the diagonalization operator, resemble to arguments used in [14] and [8].

References

1. Ackermann, W.: Zum Hilbertschen Aufbau der reellen Zahlen. Math. annalen 99, 118–133 (1928)
2. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the poly-time functions. Computational Complexity 2, 97–110 (1992)
3. Bellantoni, S., Niggl, K.-H.: Ranking primitive recursions: The low Grzegorczyk classes revisited. SIAM Journal on Computing 29(2), 401–415 (1999)
4. Caporaso, S., Covino, E., Pani, G.: A predicative approach to the classification problem. J. Funct. Program. 11(1), 95–116 (2001)
5. Cobham, A.: The intrinsic computational difficulty of functions. In: Bar-Hillel, Y. (ed.) Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, pp. 24–30. North-Holland, Amsterdam (1962)
6. Gödel, K.: Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. Dialectica 12, 280–287 (1958) Republished with English translation and explanatory notes by A. S. Troelstra in Kurt Gödel: Collected Works, Vol. II. S. Feferman, ed. Oxford University Press, Oxford (1990)
7. Jones, N.: Computability and complexity, from a programming perspective. MIT Press, Cambridge (1997)
8. Dal Lago, U., Baillot, P.: Light affine logic, uniform encoding and polynomial time. Mathematical Structures in Computer Science 16(4), 713–733 (2006)
9. Leivant, D.: A foundational delineation of computational feasiblity. In: Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91) (1991)
10. Leivant, D.: Predicative recurrence and computational complexity I: Word recurrence and poly-time. In: Clote, P., Remmel, J. (eds.) Feasible Mathematics II, Birkhäuser, pp. 320–343 (1994)
11. Leivant, D.: Ramified recurrence and computational complexity III: Higher type recurrence and elementary complexity. Annals of Pure. and Applied Logic 96(1-3), 209–229 (1999)
12. Leivant, D., Marion, J.-Y.: Lambda calculus characterizations of poly-time. Fundamenta Informaticae 19(1,2), 167,184 (1993)
13. Leivant, D., Marion, J.-Y.: A characterization of alternating log time by ramified recurrence. Theoretical Computer Science 236(1-2), 192–208 (2000)
14. Neergaard, P.M., Mairson, H.: Lal is square: Representation and expressivness in light affine and logic. In: Implicit Computational complexity (2002)
15. Simmon, H.: Tiering as a recursion technique. Bulletin of Symbolic Logic 11(3), 321–350 (2005)
16. Simmons, H.: The realm of primitive recursion. Archive for Mathematical Logic 27, 177–188 (1988)
17. Simmons, H.: Derivation and Computation. Tracts in theoretical computer science, Cambridge, vol. 51 (2000)
18. Sorensen, M., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. DIKU-rapport 98/14 (1998)

Edifices and Full Abstraction for the Symmetric Interaction Combinators

Damiano Mazza

Laboratoire d'Informatique de Paris Nord
damiano.mazza@lipn.univ-paris13.fr
http://www-lipn.univ-paris13.fr/~mazza

Abstract. The symmetric interaction combinators are a variant of Lafont's interaction combinators. They are a graph-rewriting model of parallel deterministic computation. We define a notion analogous to that of head normal form in the λ-calculus, and make a semantical study of the corresponding observational equivalence. We associate with each net a compact metric space, called *edifice*, and prove that two nets are observationally equivalent iff they have the same edifice. Edifices may therefore be compared to Böhm trees in infinite η-normal form, or to Nakajima trees, and give a precise topological account of phenomena like infinite η-expansion.

1 Introduction

Lafont's interaction nets [1] are a powerful and versatile model of parallel deterministic computation, derived from the proof-nets of Girard's linear logic [2,3]. Interaction nets are characterized by the atomicity and locality of their rewriting rules. They can be seen as "parallel Turing machines": computational steps are elementary enough to be considered as executable in constant time, but several steps can be done at the same time.

Several interesting applications of interaction nets exist. The most notable ones are implementations of optimal evaluators for the λ-calculus [4,5], but efficient evaluation of other functional programming languages using richer data structures is also possible with interaction nets [6].

However, so far the practical aspects of this computational model have arguably received much more attention than the strictly theoretical ones. With the exception of Lafont's work on the interaction combinators [7] and Fernández and Mackie's work on operational equivalence [8], no foundational study of interaction nets can be found in the existing literature. For example, until very recently [9], no denotational semantics had been proposed for interaction nets.

This work aims precisely at studying and expanding the theory of interaction nets, in particular of the symmetric interaction combinators. These latter are particularly interesting because of their *universality*: any interaction net system can be translated in the symmetric interaction combinators [7]. Therefore, a semantical study of the symmetric combinators applies, modulo a translation, to all interaction net systems.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 305–320, 2007.
© Springer-Verlag Berlin Heidelberg 2007

We introduce *observable nets*, which are analogous to head normal forms in the λ-calculus, and we define an observational equivalence based on them. This equivalence is different from the one introduced by Fernández and Mackie: the latter is in fact based on *interface normal forms*, which appear to be related to λ-calculus *weak* head normal forms.

In the λ-calculus, head normal form equivalence (hnf-equivalence) was semantically characterized in the early '70s by the independent results of Wadsworth and Hyland [10,11]: two λ-terms are hnf-equivalent iff their Böhm tree has the same infinite η-normal form. Shortly after, Nakajima introduced a similar characterization in terms of what are now known as Nakajima trees [12].

In the present work we introduce *edifices*, which play the same rôle as Böhm or Nakajima trees, in that they provide a fully abstract model of the symmetric combinators. Edifices are compact (hence complete) metric spaces, related to Cantor spaces. When nets are interpreted as edifices, phenomena similar to infinite η-expansion, which are also present in the symmetric combinators, receive a precise topological explanation.

Apart from characterizing the interactive behavior of nets, edifices show other interesting aspects, not developed in this paper. They have many common features with the *strategies* of game semantics, and are related to the Geometry of Interaction interpretation of nets [13,7]. They may be of help in improving the theory of interaction nets, for example by serving as the base for a typed semantics, or by suggesting additive (or non-deterministic) extensions of interaction nets; they may also turn out to be useful in defining alternative models of other systems, like proof-nets, or the λ-calculus itself, as these can all be encoded in the symmetric combinators.

2 The Symmetric Interaction Combinators

2.1 Nets

The symmetric interaction combinators, or, more simply, the symmetric combinators, are an interaction net system [1,7]. An interaction net is the union of two structures: a labelled, directed hypergraph, and an undirected multigraph:

Definition 1 (Net). *A* net μ *is a triple* $(\mathsf{Ports}(\mu), \mathsf{Cells}(\mu), \mathsf{Wires}(\mu))$*, where:*

- $\mathsf{Ports}(\mu)$ *is a finite set, the elements of which are called the* ports *of* μ*;*
- $\mathsf{Cells}(\mu)$ *is a set of* cells*, which are tuples of the form* $(\alpha, i_0, i_1, \ldots, i_n)$*, where* $\alpha \in \{\delta, \varepsilon, \zeta\}$*, and* $i_0, i_1, \ldots, i_n$ *are pairwise distinct ports, such that* $n = 2$ *if* $\alpha = \delta$ *or* $\alpha = \zeta$*, and* $n = 0$ *if* $\alpha = \varepsilon$*;*
- $\mathsf{Wires}(\mu)$ *is a multiset of* wires*, which are unordered pairs of distinct ports.*

$\mathsf{Cells}(\mu)$ *and* $\mathsf{Wires}(\mu)$ *must satisfy the following constraints:*

- *each port appears in at least one wire;*
- *each port appears at most twice in* $\mathsf{Cells}(\mu) + \mathsf{Wires}(\mu)$ *(*$\mathsf{Cells}(\mu)$ *is considered as a multiset in this union).*

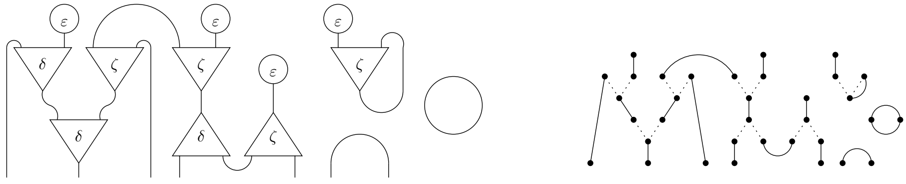


Fig. 1. A net (left) and its port graph (right, internal edges dotted)

The ports of μ appearing only once in $\mathsf{Cells}(\mu) + \mathsf{Wires}(\mu)$ *are called* free*; the set of the free ports of μ is referred to as its* interface*. In a cell $(\alpha, i_0, i_1, \ldots, i_n)$, the port i_0 is called* principal*, and the ports $i_1, \ldots, i_n$ are called* auxiliary*.*

Most of the time, it is convenient to present a net graphically, as in Fig. 1. In these representations, only cells and wires are drawn, and ports are left implicit. For a binary cell (i.e., of type δ or ζ), the principal port is represented by one of the "tips" of the triangle representing it. A wire is represented as... a wire, and the free ports appear as extremities of "pending" wires. For example, the net in Fig. 1 has 7 free ports. In the rest of the paper, we shall always assume that if a net has n free ports, then they are labelled by the integers in $\{1, \ldots, n\}$. Note also that graphical representations equate nets differing only modulo an injective renaming of ports and a collapse/extension of wires (a sort of α-equivalence).

Each net μ determines an undirected multigraph $\mathsf{PG}(\mu)$, which will be useful to speak of paths in μ (see Fig. 1):

Definition 2 (Port graph). *The* port graph *of a net μ, denoted* $\mathsf{PG}(\mu)$*, is the undirected multigraph whose vertices are the elements of* $\mathsf{Ports}(\mu)$ *and such that there is an edge between two ports i, j iff one of the following (non mutually exclusive) conditions hold:*

External edges: $\{i, j\} \in \mathsf{Wires}(\mu)$ *(multiplicities are counted here, i.e., if $\{i, j\}$ appears twice in* $\mathsf{Wires}(\mu)$*, there will be two edges relating i and j in* $\mathsf{PG}(\mu)$*);*

Internal edges: *i and j are principal and auxiliary ports of a cell of μ.*

2.2 Interaction Rules

An *active pair* is a net consisting of two cells whose principal ports are connected by a wire. Active pairs may be reduced according to the *interaction rules* (Fig. 2): the annihilations, concerning the interaction of two cells of the same type, and the commutations, concerning the interaction of two cells of different type.

Reducing an active pair inside a net means removing it and replacing it with the net given by the corresponding rule. If a net μ is transformed into μ' after such an operation, we write $\mu \to \mu'$. We define $\mu \simeq_\beta \nu$ iff there exists o such that $\mu \to^* o$ and $\nu \to^* o$. It is easy to show that the relation $\to^*$ is (strongly) confluent, so $\simeq_\beta$ is an equivalence relation (indeed a congruence).

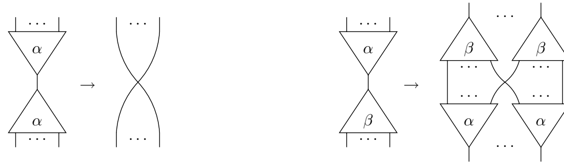


Fig. 2. The interaction rules: annihilation (left) and commutation (right). In the annihilation, the right member is empty in case $\alpha = \varepsilon$.

The interest of the symmetric combinators is given by the following result:

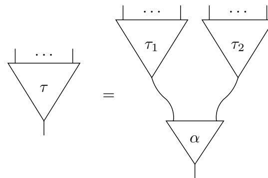
Theorem 1 (Lafont [7]). *Any interaction net system can be translated in the symmetric combinators.*

The definitions of interaction net system and of the notion of translation are out of the scope of this paper. We shall only say that, modulo an encoding, Turing machines, cellular automata, and the **SK** combinators are all examples of interaction net systems [7,9]. An example of encoding of linear logic and the λ-calculus in the symmetric combinators[1] is given by Mackie and Pinto [14]. We refer the reader to Lafont's paper [7] for a proper formulation and proof of Theorem 1.

2.3 Basic Nets

Wirings. A net containing no cell and no cyclic wire is called a *wiring*. Wirings are permutations of free ports; they are ranged over by ω.

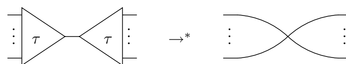
Trees. A single ε cell is a tree with no leaf, denoted by ε; a single wire is a tree with one leaf (it is arbitrary which of the two extremities is the root and which is the leaf), denoted by $\bullet$; if τ_1, τ_2 are two trees with resp. n_1, n_2 leaves, and if $\alpha \in \{\delta, \zeta\}$, the net



is a tree with $n_1 + n_2$ leaves, denoted by $\alpha(\tau_1, \tau_2)$.

Trees annihilate in a way which generalizes the annihilation of binary cells:

Lemma 1. *Let τ be a tree. Then, we have*



Proof. By induction on τ. □

[1] Actually these encodings use the interaction combinators, but they can be adapted with very minor changes to the symmetric combinators.

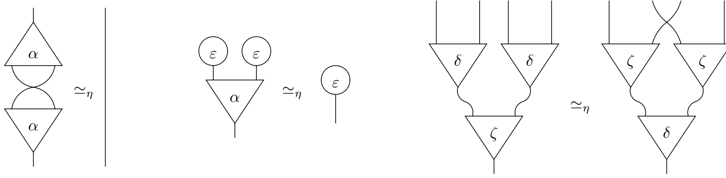


Fig. 3. The equations defining η-equivalence ($\alpha \in \{\delta, \zeta\}$)

3 Observational Equivalence

3.1 Eta Equivalence and Internal Separation

As in the λ-calculus, if reduction is extended by adding other suitable rewriting rules, a result similar to Böhm's theorem can be proved [15].

Definition 3 (Context, test). *Let μ be a net with n free ports. A* context *for μ is a net C with at least n free ports. We denote by $C[\mu]$ the* application *of C to μ, which is the net obtained by plugging the free port i of μ to the free port i of C, with $i \in \{1, \dots, n\}$. A* test *for μ is a forest of n trees $\tau_1, \dots, \tau_n$ such that the root of each τ_i is labelled by i. A test θ is therefore a particular context, and we denote by $\theta[\mu]$ its application to μ.*

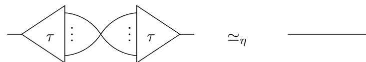
Definition 4 (η- and $\beta\eta$-equivalence). *η-equivalence is the reflexive, transitive, and contextual closure of the equations of Fig. 3. $\beta\eta$-equivalence is defined as $\simeq_{\beta\eta} = (\simeq_\beta \cup \simeq_\eta)^+$.*

In the following, W and E denote the nets with two free ports consisting resp. of a single wire and of two ε cells.

Theorem 2 (Separation [15]). *Let μ, ν be two total[2] nets with the same interface, such that $\mu \not\simeq_{\beta\eta} \nu$. Then, there exists a test θ such that $\theta[\mu] \to^* W$ and $\theta[\nu] \to^* E$, or vice versa.*

The following result is the analogous of Lemma 1 for η-equivalence, and will be used in Sect. 5 (like Lemma 1, the proof is a straight-forward induction):

Lemma 2. *Let τ be a tree without ε cells. Then, we have*



[2] *Total* means admitting a normal form without vicious circles. A vicious circle is either a cyclic wire, or a configuration consisting of n binary cells $c_1, \dots, c_n$ such that, for all $i \in \{1, \dots, n-1\}$, the principal port of c_i is connected to an auxiliary port of c_{i+1}, and the principal port of c_n is connected to an auxiliary port of c_1. Such configurations are stable under reduction, because cells can interact only through their principal port. Totality will not be relevant to the main definitions and results of this paper.

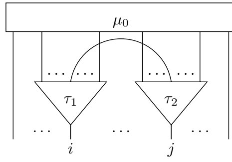
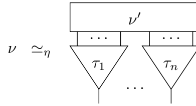


Fig. 4. An observable path

Corollary 1. *For any net ν and for any trees without ε cells $\tau_1, \ldots, \tau_n$, there exists a net ν' such that*



Proof. Simply "η-expand" the wires connected to the free ports of ν as in Lemma 2. □

3.2 Path-Based Observational Equivalence

The Separation Theorem distinguishes two nets by sending one to a net presenting a direct connection between its free ports, and the other to a net in which no such direct connection will ever form. This inspires the following definitions.

Definition 5 (Straight path, Danos and Regnier [13]). *Let μ be a net, and $i, j \in \mathsf{Ports}(\mu)$. We say that there is a* straight path *between i and j in μ iff there is a path (not necessarily simple) connecting i and j in $\mathsf{PG}(\mu)$ and alternating between internal and external edges (see Definition 2). We say that a straight path* crosses an active pair *iff it contains an edge connecting two principal ports. A* maximal path *is a non-empty straight path connecting two free ports of μ.*

Definition 6 (Observable path). *Let μ be a net. An observable path of μ is a maximal path crossing no active pair. We denote by $\mathsf{op}(\mu)$ the set of observable paths of μ, and we set*

$$\mathsf{op}^*(\mu) = \bigcup_{\mu \to^* \mu'} \mathsf{op}(\mu').$$

It is perhaps useful to visualize observable paths. A net μ contains an observable path between its free ports i and j iff it is of the shape given in Fig. 4. If $i = j$, then $\tau_1 = \tau_2$, and the wire shown connects two leaves of the same tree. The actual observable path, if seen from i to j, takes the branch of τ_1 leading to the leaf connected by the wire shown, follows this wire, and descends to the root of τ_2 through the only possible branch.

Proposition 1. *Let $\mu \to^* \mu'$. Then, $\mathsf{op}(\mu) \subseteq \mathsf{op}(\mu')$, and $\mathsf{op}^*(\mu) = \mathsf{op}^*(\mu')$.*

Proof. An immediate consequence of the locality of interaction rules. □

Note that, for any net μ, $\mathsf{op}(\mu)$ is always finite; then, by Proposition 1, $\mathsf{op}^*(\mu)$ is finite whenever μ has a normal form. The stability of observable paths under reduction is the main reason for considering them as the base of observational equivalence.

Definition 7 (Observability predicates). *We say that μ is* immediately observable*, and we write $\mu\downarrow$, iff $\mathsf{op}(\mu) \neq \emptyset$. We say that μ is* observable*, and we write $\mu\Downarrow$, iff $\mathsf{op}^*(\mu) \neq \emptyset$, or, equivalently, $\mu \to^* \mu'\downarrow$. If $\mathsf{op}^*(\mu) = \emptyset$, we say that μ is* blind*, and we write $\mu\Uparrow$.*

Definition 8 (Observational equivalence). *Two nets μ, ν with the same interface are* observationally equivalent*, and we write $\mu \simeq \nu$, iff for all contexts C, $C[\mu]\Downarrow$ iff $C[\nu]\Downarrow$.*

It helps thinking of an immediately observable net as a head normal form in the λ-calculus. As a matter of fact, it is possible to extend our definition of observable path to any interaction net system, in particular to sharing graphs [4]. In these latter, observable paths can be seen to be related to *persistent* paths [13]. Then, one can adapt the definition of observable net so as to obtain that a λ-term is in head normal form iff its corresponding net is immediately observable. This adaptation, which we do not detail here, takes into account only the observable paths starting from the free port representing the “root” of the term, and iteratively using the “root” of each subterm.

The existence of a “root” (i.e., a distinguished free port in sharing graphs) is what allows one to define the notion of *principal* head normal form, of which no meaningful equivalent exists for nets. This is because nets, like proof-nets, are “classical”, as opposed to λ-terms, which are “intuitionistic”. This is also the reason why the symmetric combinators equivalent of Böhm trees will not be trees (cf. Sect. 4).

Following the analogy with the λ-calculus, blind nets correspond to unsolvable terms. If we deem *semi-sensible* a congruence on nets including $\simeq_\beta$ and not equating a blind and an observable net, then it is not hard to show that $\simeq$ is the greatest semi-sensible congruence, just like the corresponding theory $\mathcal{H}^*$ in the λ-calculus.

We also have that, if μ is a blind net with n free ports, then $\mu \simeq E_n$, where E_n is the net consisting of n cells of type ε. Thus, each equivalence class of blind nets (for any interface) has a representative which is normal, in sharp contrast with the λ-calculus. In this respect, one may consider ε cells as the “reification” of unsolvability. Additionally, it can be shown that $\simeq_{\beta\eta}$ is a semi-sensible congruence, so that $\simeq_{\beta\eta} \subseteq \simeq$. Therefore, by Theorem 2, $\simeq_{\beta\eta}$ and $\simeq$ coincide on total nets; in particular, two normal nets without vicious circles are observationally equivalent iff they are $\beta\eta$-equivalent.[3] These results are all consequences of Theorem 3 (Sect. 5), but can also be proved independently.

We conclude by stating an important Context Lemma, saying that tests suffice to discriminate between nets (the proof is technical, and is omitted here):

Lemma 3 (Context). *$\mu \simeq \nu$ iff, for every test θ, $\theta[\mu]\Downarrow$ iff $\theta[\nu]\Downarrow$.*

[3] See footnote 2 for the definition of vicious circle and total net.

4 Edifices

We shall now introduce the main mathematical objects of our paper, namely *edifices*. These will be used to develop a denotational semantics for nets, borrowing ideas from the path semantics of linear logic, i.e., Girard's Geometry of Interaction as formulated by Danos and Regnier [13]. Although edifices and Böhm trees are technically quite different, there are strong analogies between the two. Also, the topology used to define edifices is the same used by Kennaway et al. to define the infinitary λ-calculus [16].

In what follows, $\mathcal{C} = \{\mathbf{p}, \mathbf{q}\}^{\mathbb{N}}$ is the set of infinite binary words, ranged over by x, y, equipped with the Cantor topology. We remind that $\mathcal{C}$ is metrizable, with the distance defined for example by $d_{\mathcal{C}}(x, y) = 2^{-k}$, where k is the length of the longest common prefix of x, y. We denote by $\mathcal{B}^{\circ}_{x,r}$ the open ball of center x and radius r. The elements of $\mathcal{C} \times \mathcal{C}$, which is also a Cantor space, will be denoted by $x \otimes y$, and ranged over by u, v, w. Below, the set $\mathbb{N}$ of non-negative integers, ranged over by i, j, will be considered equipped with the discrete topology.

Definition 9 (Pillar). *Given $I \subseteq \mathbb{N}$, set $\mathcal{P}_I = \mathcal{C} \times \mathcal{C} \times I$, equipped with the product topology. A* pillar *is an element of $\mathcal{P} = \mathcal{P}_{\mathbb{N}}$. Pillars are denoted by $u \,@\, i$, and are ranged over by ξ, υ. The pillar $u \,@\, i$ is said to be* based *at i.*

Observe that $\mathcal{P}$ is also metrizable; if $\xi = x \otimes y \,@\, i$ and $\upsilon = x' \otimes y' \,@\, i'$, we shall consider the distance $d(\xi, \upsilon) = \max\{d_{\mathcal{C}}(x, y), d_{\mathcal{C}}(x', y'), d_{\mathrm{disc}}(i, i')\}$, where d_{disc} is the discrete metric, defined as $d_{\mathrm{disc}}(i, j) = 0$ if $i = j$, and $d_{\mathrm{disc}}(i, j) = 2$ if $i \neq j$. Therefore, to be "close", two pillars must be based at the same integer.

Definition 10 (Arch). *Given $I \subseteq \mathbb{N}$, pose $\overrightarrow{\mathcal{A}}_I = \mathcal{P}_I \times \mathcal{P}_I$, equipped with the product topology, and set $(\xi, \upsilon) \sim (\xi', \upsilon')$ iff $\xi' = \upsilon$ and $\upsilon' = \xi$, or $\xi' = \xi$ and $\upsilon' = \upsilon$. We then define $\mathcal{A}_I = \overrightarrow{\mathcal{A}}_I / \sim$, equipped with the quotient topology. An* arch *is an element of $\mathcal{A} = \mathcal{A}_{\mathbb{N}}$. Arches are denoted by $\xi \frown \upsilon$ (which is the same as $\upsilon \frown \xi$), and ranged over by $\mathfrak{a}$; sets of arches are ranged over by $\mathfrak{E}$. An arch is said to be* based *at the unordered pair where its two pillars are based.*

The following helps understanding the topology given to $\mathcal{A}$:

Proposition 2. *The space $\mathcal{A}$ is metrizable; if $\mathfrak{a} = \xi \frown \upsilon$ and $\mathfrak{a}' = \xi' \frown \upsilon'$, the function $D(\mathfrak{a}, \mathfrak{a}') = \min\{\max\{d(\xi, \xi'), d(\upsilon, \upsilon')\}, \max\{d(\xi, \upsilon'), d(\upsilon, \xi')\}\}$ is a distance inducing its topology.*

In other words, to compare two arches, we overlap them in both possible ways, and we take the way that "fits best". The distance D is in fact the standard quotient metric; in this case, it collapses to this simple form.

The space $\mathcal{A}$ is not compact. In fact, we can give a characterization of its compact subsets:

Proposition 3. *$\mathfrak{E}$ is compact iff it is a closed subset of $\mathcal{A}_I$ for some finite I.*

Proof. If $\mathfrak{E}$ is compact, then it must be closed; suppose however that $\mathfrak{E} \not\subseteq \mathcal{A}_I$ for any finite I. Then, let $\mathfrak{a}_{i,j}$ be a sequence of arches spanning all of the i, j where

the arches of $\mathfrak{E}$ are based, and set $U_{i,j} = \mathfrak{E} \cap \mathcal{B}^\circ_{\mathfrak{a}_{i,j},2}$. These are all open sets in the relative topology, and since, for all i, j, $D(\mathfrak{a}_{i,j}, \mathfrak{a}) < 2$ iff $\mathfrak{a}$ is based at i, j, they form an open cover of $\mathfrak{E}$. Now observe that, by the same remark on the distance, if we remove any $U_{m,n}$ we loose all arches of $\mathfrak{E}$ based at m, n. But we have supposed the sequence $\mathfrak{a}_{i,j}$ to be infinite, so $U_{i,j}$ is an infinite open cover of $\mathfrak{E}$ admitting no finite subcover, in contradiction with the compactness of $\mathfrak{E}$.

For the converse, I being finite, it is not hard to show that $\mathcal{P}_I$ is homeomorphic to $\mathcal{C}$. Therefore, $\mathcal{P}_I$ is a Cantor space, hence compact. So $\mathcal{A}_I$ is compact, because it is the quotient of a product of compact spaces. But a closed subset of a compact space is compact, hence the result. □

It can be shown that each $\mathcal{A}_I$ is also perfect and totally disconnected, which means that actually these are all Cantor spaces whenever I is finite. What really matters to us though is compactness, which implies completeness (with respect to the metric D of Proposition 2): when I is finite, there is identity between closed, compact, and complete subsets of $\mathcal{A}_I$.

Definition 11 (Edifice). *An* edifice *is a compact set of arches.*

5 Nets as Edifices

The basic idea to assign an edifice to a net is that arches model observable paths.[4] These latter in fact can be seen as unordered pairs of addresses in trees; now, in a pillar $x \otimes y \,@\, i$, any pair of finite prefixes of x, y may be seen as an address, and the base i identifies the tree (a net may have several free ports, and each may be the root of a tree). The need for infinite words arises from η-expansion (the $\alpha\alpha$ equation at left in Fig. 3), which can be applied indefinitely, as in the pure λ-calculus.

In the following, we let a, b range over the set $\{\mathbf{p}, \mathbf{q}\}^*$ of finite binary words, and we denote by $\mathbf{1}$ the empty word. Pairs of finite words are denoted by $a \otimes b$, and ranged over by s, t. The concatenation of two finite words a, b or of a finite word a and an infinite word x are denoted by simple juxtaposition, i.e., as ab and ax respectively. The concatenation of two pairs of finite words $a \otimes b, a' \otimes b'$ or of a pair of finite words $a \otimes b$ and a pair of infinite words $x \otimes y$ are defined resp. as $aa' \otimes bb'$ and $ax \otimes by$, and are also denoted by juxtaposition. If u is a pair of infinite words, when we say that s is a prefix of u we mean that $u = su'$ for some u', and we always implicitly assume that $s = a \otimes b$ with a, b of equal length, which is also said to be the length of s.

Definition 12 (Address of a leaf). *Let τ be a tree, and l a leaf of τ. The* address *of l in τ, denoted by $\mathsf{addr}_\tau(l)$, is a pair of finite binary words defined by induction on τ.*[5]

– *$\tau = \bullet$: $\mathsf{addr}_\tau(l) = \mathbf{1} \otimes \mathbf{1}$;*

---

[4] Graphically (Fig. 4), observable paths look like arches, hence the terminology.

[5] For the acquainted reader, $\mathsf{addr}_\tau(l)$ is nothing but the GoI weight of the path going down from l to the root of τ [7]. This justifies our notations for binary words.

– $\tau = \delta(\tau_1, \tau_2)$: $\mathsf{addr}_\tau(l) = (\mathbf{p} \otimes \mathbf{1})\mathsf{addr}_{\tau_1}(l)$ *if* l *is a leaf of* τ_1, $\mathsf{addr}_\tau(l) = (\mathbf{q} \otimes \mathbf{1})\mathsf{addr}_{\tau_2}(l)$ *if* l *is a leaf of* τ_2;
– $\tau = \zeta(\tau_1, \tau_2)$: $\mathsf{addr}_\tau(l) = (\mathbf{1} \otimes \mathbf{p})\mathsf{addr}_{\tau_1}(l)$ *if* l *is a leaf of* τ_1, $\mathsf{addr}_\tau(l) = (\mathbf{1} \otimes \mathbf{q})\mathsf{addr}_{\tau_2}(l)$ *if* l *is a leaf of* τ_2.

Definition 13 (Edifice of an observable path). *Let* μ *be a net, and let* ϕ *be an observable path of* μ *connecting the free ports* i *and* j. *By definition,* ϕ *is completely described by the free ports* i, j *and the leaves* l_i, l_j *of the two trees* τ_i, τ_j *rooted at* i, j *which are connected in* ϕ *(cf. Fig. 4). Therefore, if we put* $s = \mathsf{addr}_{\tau_i}(l_i)$ *and* $t = \mathsf{addr}_{\tau_j}(l_j)$, *we define*

$$\phi^\bullet = \{sw \,@\, i \frown tw \,@\, j \;;\; \forall w \in \mathcal{C} \times \mathcal{C}\}.$$

It is not hard to check that the set defined above is indeed an edifice:

Proposition 4. *If* μ *is a net with* n *free ports and* ϕ *an observable path of* μ, $\phi^\bullet$ *is a closed subset of* $\mathcal{A}_{\{1,\ldots,n\}}$.

Definition 14 (Edifice of a net). *Let* μ *be a net. The* pre-edifice *of* μ *is the set*

$$\mathfrak{E}_0(\mu) = \bigcup_{\phi \in \mathsf{op}^*(\mu)} \phi^\bullet.$$

The edifice of μ *is the closure of its pre-edifice:* $\mathfrak{E}(\mu) = \overline{\mathfrak{E}_0(\mu)}$.

The soundness of the above definition can be checked as follows: by Proposition 4, all of the $\phi^\bullet$ are subsets of $\mathcal{A}_I$ for some finite I; arches based outside I are "too far" to be adherent to $\mathfrak{E}_0(\mu)$, therefore its closure is still in $\mathcal{A}_I$. By Proposition 3, this is enough to ensure the compactness of $\mathfrak{E}(\mu)$.

Observe that if μ is normalizable, then $\mathsf{op}^*(\mu)$ is finite, hence by Proposition 4 $\mathfrak{E}_0(\mu)$ is already closed. It is however possible to find non-normalizable nets whose pre-edifice is not an edifice (e.g. the net of Fig. 5 discussed below).

The closure is in fact essential for yielding a fully-abstract denotational semantics of nets. It is crucial in the proof of the following result:

Lemma 4. *Let* μ, ν *be two nets with* n *free ports. Then,* $\mathfrak{E}(\mu) \neq \mathfrak{E}(\nu)$ *implies that there exist* $i, j \in \{1, \ldots, n\}$, *two pairs of finite words* s, t, *and two observable paths* $\phi \in \mathsf{op}^*(\mu)$ *and* $\psi \in \mathsf{op}^*(\nu)$ *such that, if we put* $\mathfrak{a}_w = sw \,@\, i \frown tw \,@\, j$, *either for all* w, *we have* $\mathfrak{a}_w \in \phi^\bullet \setminus \mathfrak{E}(\nu)$, *or for all* w, *we have* $\mathfrak{a}_w \in \psi^\bullet \setminus \mathfrak{E}(\mu)$.

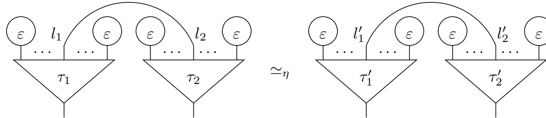
Proof. Suppose, without loss of generality, that there exists $\mathfrak{a} \in \mathfrak{E}(\mu) \setminus \mathfrak{E}(\nu)$, based at $i, j \in \{1, \ldots, n\}$. Remember that $\mathfrak{E}(\mu)$ and $\mathfrak{E}(\nu)$ are defined as the closures of resp. $\mathfrak{E}_0(\mu)$ and $\mathfrak{E}_0(\nu)$, and that by Proposition 3 they are both compact, hence complete. Then, if $\mathfrak{a} \in \mathfrak{E}(\mu) \setminus \mathfrak{E}_0(\mu)$, $\mathfrak{a}$ must be a "missing limit" of a Cauchy sequence $\mathfrak{a}_n \in \mathfrak{E}_0(\mu)$. Since a subsequence of a Cauchy sequence is still a Cauchy sequence, there must exists an integer m such that, for all $n \geq m$, $\mathfrak{a}_n \in \mathfrak{E}_0(\mu) \setminus \mathfrak{E}(\nu)$, otherwise $\mathfrak{a}$ would belong to $\mathfrak{E}(\nu)$ because of its completeness. Therefore, modulo replacing it by one of these $\mathfrak{a}_n$, we can always assume that $\mathfrak{a} \in \mathfrak{E}_0(\mu)$. If it is so, then by definition there exists an observable

path $\phi \in \mathsf{op}^*(\mu)$ such that $\mathfrak{a} \in \phi^\bullet$, which means that $\mathfrak{a} = sw_0 \mathbin{@} i \frown tw_0 \mathbin{@} j$ and, for every $w \in \mathcal{C} \times \mathcal{C}$, $sw \mathbin{@} i \frown tw \mathbin{@} j \in \phi^\bullet$, where s and t are the addresses of two leaves in the reduct(s) of μ in which ϕ appears. Now let $s'_1, \ldots, s'_n, \ldots$ be a sequence of prefixes of increasing length of w_0, and set, for all n, $s_n = ss'_n$ and $t_n = ts'_n$. Suppose that, for all n, there exist two pairs of infinite words u_n, v_n such that $\mathfrak{a}_n = s_n u_n \mathbin{@} i \frown t_n v_n \mathbin{@} j \in \mathfrak{E}(\nu)$; it is not hard to verify that the arches $\mathfrak{a}_n$ would form a Cauchy sequence of limit $\mathfrak{a}$, and thus, by the completeness of $\mathfrak{E}(\nu)$, we would obtain $\mathfrak{a} \in \mathfrak{E}(\nu)$, a contradiction. Therefore, there must exist an integer n such that, for all w, $s_n w \mathbin{@} i \frown t_n w \mathbin{@} j \in \phi^\bullet \setminus \mathfrak{E}(\nu)$. □

Lemma 5. *$\mu \simeq_\eta \nu$ and $\mu \to^* \mu'$ implies that there exist $\mu'' \simeq_\eta \nu''$ such that $\mu' \to^* \mu''$ and $\nu \to^* \nu''$.*

Proof. Omitted (see [15]). □

Definition 15 (η-equivalent observable paths). *Let $\tau_1, \tau_2, \tau'_1, \tau'_2$ be trees, with $\tau_1 = \tau_2$ iff $\tau'_1 = \tau'_2$, and let ϕ, ϕ' be two observable paths, such that in ϕ there is a connection between two leaves l_1, l_2 of τ_1 and τ_2, and in ϕ' there is a connection between two leaves l'_1, l'_2 of τ'_1 and τ'_2. We say that ϕ is η-equivalent to ϕ' iff*



Lemma 6. *Let $\mu \simeq_\eta \nu$, and let $\phi \in \mathsf{op}^*(\mu)$. Then, there exists $\psi \in \mathsf{op}^*(\nu)$ such that ϕ and ψ are η-equivalent.*

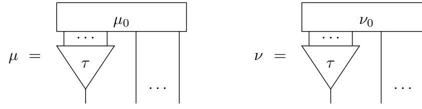
Proof (sketch). By definition, $\phi \in \mathsf{op}^*(\mu)$ means that ϕ is an observable path of a reduct μ' of μ. By Lemma 5, $\mu' \to^* \mu''$ and $\nu \to^* \nu''$ such that $\mu'' \simeq_\eta \nu''$. But observable paths are preserved under reduction, so ϕ is also present in μ''. Now if, in rewriting μ'' to ν'', no active pair is introduced to alter the observability of ϕ, then clearly ν'' contains an observable path η-equivalent to ϕ. Otherwise, it is easy to check that an $\alpha\alpha$ equation must have been used. In this case, one can prove that the active pairs introduced can be reduced to obtain $\nu'' \to^* o$ such that o contains an observable path ψ η-equivalent to ϕ. But the reducts of ν'' are also reducts of ν, so $\psi \in \mathsf{op}^*(\nu)$. □

Lemma 7. *If $\mu \simeq_\eta \nu$, then $\mathfrak{E}_0(\mu) = \mathfrak{E}_0(\nu)$ (hence $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$).*

Proof (sketch). By Lemma 6, it is enough to check that, whenever ϕ and ψ are η-equivalent observable paths, $\phi^\bullet = \psi^\bullet$. The η-equations concerning ε cells need not be considered; in the case of the $\delta\zeta$ equation, the fact that in pillars δ and ζ cells are treated by separate words makes their relative order irrelevant, and thus accounts for the their commutation; the $\alpha\alpha$ equations, which in this case may only be applied to the wire connecting the two leaves of an observable path, are modelled by the fact that all possible "uniform completions" of the addresses of the leaves are taken in the edifice of an observable path. □

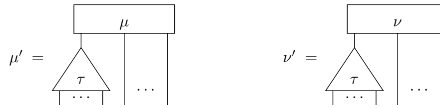
We now prove that $\mathfrak{E}(\cdot)$ induces a congruence with respect to tests:

Lemma 8. *1. Let τ be a tree, and let*



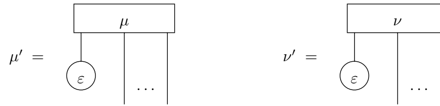
Then, $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$ iff $\mathfrak{E}(\mu_0) = \mathfrak{E}(\nu_0)$.

2. Let μ, ν be two nets with the same interface such that $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$, and let τ be a tree without ε cells. Then, if we pose



we have $\mathfrak{E}(\mu') = \mathfrak{E}(\nu')$.

3. Let μ, ν be two nets with the same interface such that $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$, and let



Then, $\mathfrak{E}(\mu') = \mathfrak{E}(\nu')$.

Proof

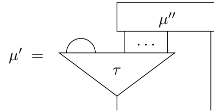
1. Easy.
2. Simply consider the nets μ'', ν'' obtained from μ', ν' by adding a copy of τ to the one already existing in the two nets, so that each leaf l in one copy is connected to the same leaf l in the other copy. By Lemma 2, we have that $\mu'' \simeq_\eta \mu$ and $\nu'' \simeq_\eta \nu$; by point 1, we have $\mathfrak{E}(\mu') = \mathfrak{E}(\nu')$ iff $\mathfrak{E}(\mu'') = \mathfrak{E}(\nu'')$; but by Lemma 7, and by hypothesis, $\mathfrak{E}(\mu'') = \mathfrak{E}(\mu) = \mathfrak{E}(\nu) = \mathfrak{E}(\nu'')$.
3. Call k the free port of μ to which the ε cell is connected in μ'. Observe that such ε cell can either disappear, or be duplicated, and that, in any case, ε cells cannot be used by observable paths. Hence, $\phi \in \mathsf{op}^*(\mu')$ iff $\phi \in \mathsf{op}^*(\mu)$ and ϕ connects two free ports of μ both different than k. Therefore, $\mathfrak{E}(\mu') = \{u\,@\,i \frown u\,@\,j \in \mathfrak{E}(\mu)\ ;\ j, k \neq i\}$. The same holds for ν, so from $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$ it easily follows that $\mathfrak{E}(\mu') = \mathfrak{E}(\nu')$. □

Corollary 2. *Let μ, ν be two nets with the same interface, and let θ be a test. Then, $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$ implies $\mathfrak{E}(\theta[\mu]) = \mathfrak{E}(\theta[\nu])$.*

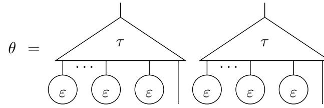
To prove full abstraction, we first need the following separation result:

Lemma 9. *Let W be a net with two free ports connected by a wire, and let μ be a net with two free ports, such that $\phi \in \mathsf{op}^*(\mu)$ implies that ϕ does not connect the port 1 to the port 2. Then, there exists a test θ such that $\theta[W]\Downarrow$ and $\theta[\mu]\Uparrow$.*

Proof. If $\mu\Uparrow$, the identity test suffices, so suppose $\mu\Downarrow$. By hypothesis, all observable paths appearing in the reducts of μ connect one of the free ports to itself. Therefore, there exists μ' such that $\mu \to^* \mu'$, and



In the above picture, we have supposed that the observable path connects the free port 1 to itself, and that the leaves connected in the path are the two "left-most" leaves of τ. These are just graphically convenient assumptions, causing no loss of generality: the observable path may as well connect port 2 to itself, and the leaves connected may be any two leaves of τ. Now, if we define



we have that, thanks to Lemma 1, $\theta[W] \to^* W$, while $\theta[\mu]$ reduces to a net whose free port 1 is connected to an ε cell. If this net is blind, we are done; otherwise, there is a reduct of $\theta[\mu]$ containing an observable path between the free port 2 and itself. This observable path can be "eliminated" with the same technique, while the ε cell on port 1 will "eat" any tree fed to it, so in the end we obtain a test θ' such that $\theta'[W] \to^* W\downarrow$, while $\theta'[\mu]\Uparrow$, as desired. □

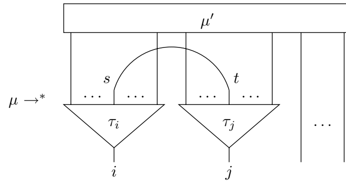
We are now ready to prove our main result:

Theorem 3 (Full abstraction). *$\mu \simeq \nu$ iff $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$.*

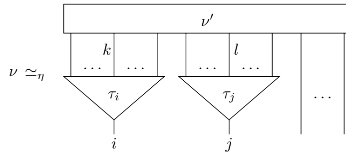
Proof. Consider first the backward implication (also known as the adequacy property). We start by observing that, for any net o, $o\Downarrow$ iff $\mathsf{op}^*(o) \neq \emptyset$ iff $\mathfrak{E}(o) \neq \emptyset$. Now, suppose $\mathfrak{E}(\mu) = \mathfrak{E}(\nu)$, and let θ be a test. By Corollary 2, we have $\mathfrak{E}(\theta[\mu]) = \mathfrak{E}(\theta[\nu])$, so following the above remark $\theta[\mu]\Downarrow$ iff $\mathfrak{E}(\theta[\mu]) \neq \emptyset$ iff $\mathfrak{E}(\theta[\nu]) \neq \emptyset$ iff $\theta[\nu]\Downarrow$. Then $\mu \simeq \nu$ follows from the Context Lemma 3.

Now we turn to the actual full abstraction property. For this, we consider the contrapositive statement, and assume $\mathfrak{E}(\mu) \neq \mathfrak{E}(\nu)$. Let I be the interface of μ and ν. By Lemma 4, we know that there exist $i, j \in I$, $\phi \in \mathsf{op}^*(\mu)$, and two leaves in a reduct of μ of addresses s, t such that, for all w, $sw \,@\, i \frown tw \,@\, j \in \phi^\bullet \setminus \mathfrak{E}(\nu)$ (it could actually be that these arches belong to $\psi^\bullet \setminus \mathfrak{E}(\mu)$, where $\psi \in \mathsf{op}^*(\nu)$, but obviously our assumption causes no loss of generality). We shall suppose

$i \neq j$; the reader is invited to check that the argument can be adapted to the case $i = j$. By Definition 13, and by the fact that $\phi \in \mathsf{op}^*(\mu)$, we have

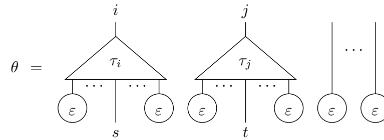


where we have explicitly drawn the connection between the two leaves of resp. addresses s and t. On the other hand, by Corollary 1, we have

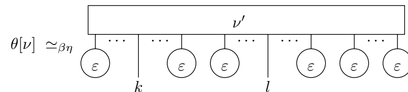


where we have called k and l the two free ports of ν' corresponding resp. to the addresses t and s in τ_i and τ_j. Observe that, by Lemma 5, the edifice of the net on the right is still $\mathfrak{E}(\nu)$. Now if, in any reduct of ν', there appeared an observable path between k and l, then we would contradict the fact that, for all w, $sw \,@\, i \frown tw \,@\, j \notin \mathfrak{E}(\nu)$. Therefore, no observable path ever appears between k and l in any reduct of ν'.

Consider then the test



where we have left free only the leaves corresponding to the addresses s and t of τ_i and τ_j. Now, by Lemma 1, $\theta[\mu] \to^* W$, where W is a wire plus a net with no interface; on the other hand, we have



But ν' never develops observable paths between k and l, so Lemma 9 applies, and we obtain $\mu \not\simeq \nu$. □

As an immediate application of Theorem 3, we give an example of a net which is not normalizable, and yet is observationally equivalent to a wire; this is analogous

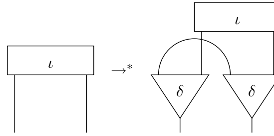


Fig. 5. A non-normalizable net observationally equivalent to a wire

to Wadsworth's "infinitely η-expanding" term $J = RR$, where $R = \lambda xzy.z(xxy)$, which is well known to be hnf-equivalent to $\lambda z.z$.

Consider a net ι containing no observable paths, and reducing as in Fig. 5. Such a net exists, although its description is not as concise as that of J. We see that $\phi \in \mathsf{op}^*(\iota)$ iff $\phi^\bullet = \{\mathbf{q}^n\mathbf{p}x \otimes y\,@\,1 \frown \mathbf{q}^n\mathbf{p}x \otimes y\,@\,2\ ;\ \forall x, y \in \mathcal{C}\}$ for some non-negative integer n. On the other hand, if W denotes a wire, $\mathfrak{E}(W) = \mathfrak{E}_0(W) = \{u\,@\,1 \frown u\,@\,2\ ;\ \forall u \in \mathcal{C} \times \mathcal{C}\}$. Now, if $\mathbf{q}^\infty$ denotes an infinite sequence of $\mathbf{q}$'s, all arches of the form $\mathfrak{a}_y = \mathbf{q}^\infty \otimes y\,@\,1 \frown \mathbf{q}^\infty \otimes y\,@\,2$ are missing from $\mathfrak{E}_0(\iota)$, hence $\mathfrak{E}_0(\iota) \subsetneq \mathfrak{E}_0(W)$. But these arches are all adherent to $\mathfrak{E}_0(\iota)$: in fact, it is very easy to construct a Cauchy sequence in $\mathfrak{E}_0(\iota)$ of limit $\mathfrak{a}_y$, for any y. Therefore, $\mathfrak{E}(\iota) = \mathfrak{E}(W)$, and $\iota \simeq W$.

Notice that the reducts of ι are "almost" η-equivalent to a wire: there is just one missing connection. We can say that this connection forms "in the limit", when the reduction is carried on forever. When one interprets nets as edifices, this informal remark becomes a precise topological fact, i.e., we have a true limit.

Compactness is crucial for obtaining full abstraction. Notice in fact that $\mathfrak{E}_0(\cdot)$ already gives an adequate semantics of nets, which however fails to be fully abstract, as the above example itself shows.

References

1. Lafont, Y.: Interaction nets. In: Conference Record of POPL'90, ACM Press, pp. 95–108. ACM Press, New York (1990)
2. Girard, J.Y.: Proof-nets: The parallel syntax for proof-theory. In: Ursini, Agliano (eds.) Logic and Algebra, Marcel Dekker, Inc. (1996)
3. Lafont, Y.: From proof nets to interaction nets. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic, pp. 225–247. Cambridge University Press, Cambridge (1995)
4. Gonthier, G., Abadi, M., Lévy, J.J.: The geometry of optimal lambda reduction. In: Conference Record of POPL 92, pp. 15–26. ACM Press, New York (1992)
5. Mackie, I.: Efficient lambda evaluation with interaction nets. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 155–169. Springer, Heidelberg (2004)
6. Mackie, I.: An interaction net implementation of additive and multiplicative structures. Journal of Logic and Computation 15(2), 219–237 (2005)
7. Lafont, Y.: Interaction combinators. Information and Computation 137(1), 69–101 (1997)
8. Fernández, M., Mackie, I.: Operational equivalence for interaction nets. Theoretical Computer Science 297(1–3), 157–181 (2003)

9. Mazza, D.: A denotational semantics for the symmetric interaction combinators. To appear in Mathematical Structures in Computer Science, vol. 17(3) (2007)
10. Wadsworth, C.: The relation between computational and denotational properties for Scott's D_∞ models. Siam J. Comput. 5(3), 488–521 (1976)
11. Hyland, M.: A syntactic characterization of the equality in some models of the lambda calculus. J. London Math. Society 2(12), 361–370 (1976)
12. Nakajima, R.: Infinite normal forms for the λ-calculus. In: Böhm, C. (ed.) Lambda-Calculus and Computer Science Theory. LNCS, pp. 62–82. Springer, Heidelberg (1975)
13. Danos, V., Regnier, L.: Proof nets and the Hilbert space. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic, pp. 307–328. Cambridge University Press, Cambridge (1995)
14. Mackie, I., Pinto, J.S.: Encoding linear logic with interaction combinators. Information and Computation 176(2), 153–186 (2002)
15. Mazza, D.: Observational equivalence for the interaction combinators and internal separation. In: Mackie, I. (ed.) Proceedings of TERMGRAPH 2006. ENTCS, pp. 7–16. Elsevier, Amsterdam (2006)
16. Kennaway, R., Klop, J.W., Sleep, R., de Vries, F.J.: Infinitary lambda calculus. Theoretical Computer Science 175(1), 93–125 (1997)

Two Session Typing Systems for Higher-Order Mobile Processes

Dimitris Mostrous and Nobuko Yoshida

Department of Computing, Imperial College London

Abstract. This paper proposes two typing systems for session interactions in higher-order mobile processes. Session types for the HOπ-calculus capture high-level structures of communication protocols and code mobility as type abstraction, and can be used to statically check the safe and consistent process composition in communication-centric distributed software. Integration of arbitrary higher-order code mobility and sessions leads to technical difficulties in type soundness, because linear usage of session channels and completion of sessions are required in executed code. By using techniques from the linear λ-calculus, we develop a coherent and tractable session typing system for the HOπ-calculus. We also present an alternative system based on fine-grained process types. The formal comparison between the two systems offers insight on the interplay between higher-order code mobility and session types.

1 Introduction

In global computing environments, applications are executed across multiple distributed sites or devices. The use of mobile code is prominent in such environments, where several participants are synthesised by communication of not only passive values but also of runnable code: for example a service can be delegated to different participants, by sending either a channel via which it is accessible, or code that accesses it; and incoming code may transit through several devices that alter their computational behaviour or their data through interaction with it.

The Higher-Order π-calculus (HOπ-calculus) [19] is a general formalism of interaction in which two kinds of mobility, name passing and process passing, are integrated in a simple and universal form: in this model, processes can be instantiated by names and other processes, just like a piece of mobile code is instantiated with local capability after migration. This additional expressiveness inherited from the λ-calculus provides a powerful basis for describing and analysing dynamicity in global computing scenarios.

As a type-theoretic foundation for highly structured communication protocols often found in distributed applications, this paper focuses on the notion of *sessions* and their types. A session is a series of communications between two parties which form a meaningful logical unit, just like a web session between a browser and a server when a human user interacts with an e-commerce site. Session types model such interactions as an abstract structure of typed inputs and outputs. The study of session typing systems is now wide-spread due to the need for structured communications in various scenarios in distributed computing. Starting from 1994, it has been studied for the π-calculus [13,20,12,11,4,28,16], ambients [10], CORBA [21], Concurrent

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 321–335, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Haskell [18], multi-threaded functional languages [23] and distributed [8] and multi-threaded Java [7]. At the industry level, languages with variants of session types are implemented in an operating system [9] and W3C Choreography Web Description Language [5,25].

While many advanced session types for the π-calculus and programming languages have been studied, there existed no session typing systems for the HOπ-calculus. Incorporation of sessions into the HOπ-calculus offers a general theoretical basis for examining the interplay between two non-trivial features in communication-based programming, higher-order mobility and session-based structured interaction. This paper establishes the first session type theory for the HOπ-calculus which can statically validate the type safety of complex distributed scenarios with code mobility. In spite of their simple type syntax, the previous literature have shown that obtaining type soundness for session types is an intricate task because of delegation of sessions [28]. In addition, in the presence of higher-order process passing, with the instantiation of names within executable code, preservation of typability becomes even more non-trivial. We provide two different solutions: one by controlling the linear use of variables for higher-order processes, which enjoys simplicity and tractability; and another by exporting channel capabilities as types of processes, which needs more annotations but has wider, more flexible typability. Both methods provide a potential type-theoretic basis of future programming idioms for dynamic code mobility and structured communications [2,17]. Due to the space limitation, the detailed definitions and proofs are left to [1].

2 The Higher-Order π-Calculus with Sessions

2.1 Syntax and Reduction

The calculus is given in Fig. 1, based on the π-calculus augmented with session primitives and the call-by-value λ-calculus. A session is a series of reciprocal interactions between two parties, possibly with branching, serving as a unit of type abstraction. A session is initiated over a *shared channel* and communications belonging to a session are performed via two fresh end-point channels specific to that session, called *session channels*. The indices 0 and 1 of session channels are used to distinguish the two end points, taking a similar approach to [13,28]. We write $\tilde{V}$ for a potentially empty vector $V_1 ... V_n$. Types, given later, are denoted by t and σ, but annotations are often omitted.

For terms, we have prefixes for declaring session connections, $!u(x).P$ for servers and $\overline{u}(x).P$ for clients. Here the identifier u represents the public interaction point over which a session may commence. The bound variable x represents the actual channel over which the session's communications will take place. Session communications are performed using the next four primitives: input $k(x).P$, output $\overline{k}\langle V\rangle.P$, branching $k \triangleright \{l_1 : P_1; \dots ; l_n : P_n\}$ (often written as $k \triangleright \{l_i : P_i\}_{i\in I}$ with index set I) which offers alternative interaction patterns, and selection $k \triangleleft l.P$ which chooses an available branch. $(\nu a : \sigma)P$ restricts (and binds) a channel a to the scope of P. Similarly, $(\nu\kappa)P$ binds κ_0 and κ_1, making them private to P. Other primitives are standard. We often omit $\mathbf{0}$. The *bindings* are induced by $(\nu a : \sigma)P$, $(\nu\kappa)P$, $!u(x).P$, $\overline{u}(x).P$ and $\lambda x.P$. The derived notions of bound and free identifiers, alpha equivalence and substitution are standard. We write $\mathsf{fv}(P)$/$\mathsf{fn}(P)$ for the set of free variables/channels, respectively. The single-step

(Identifiers) $u, v, w ::= x, y, z$ variables $\quad k ::= x, y, z$ variables
$\mid a, b, c$ shared channels $\quad \mid \kappa_i \quad i \in \{0,1\}$ session channels

(Terms)

$P, Q, R ::=$		
	V	value
$\mid$	$!u(x).P$	server
$\mid$	$\overline{u}(x).P$	client
$\mid$	$k(x).P$	input
$\mid$	$\overline{k}\langle V \rangle.P$	output
$\mid$	$k \triangleright \{l_1 : P_1; \ldots; l_n : P_n\}$	branching
$\mid$	$k \triangleleft l.P$	selection
$\mid$	$P \mid Q$	parallel
$\mid$	$(\nu a : \sigma) P$	restriction
$\mid$	$(\nu \kappa) P$	restriction
$\mid$	PQ	application
$\mid$	$\mathbf{0}$	nil process

(Values)

$V, V', W ::=$		
	u, v, w	shared identifier
$\mid$	k, k', k''	linear identifier
$\mid$	$()$	unit
$\mid$	$\lambda(x : t).P$	abstraction

(Abbreviations)

$\ulcorner P \urcorner \stackrel{\text{def}}{=} \lambda(x : \mathsf{unit}).P \quad (x \notin \mathsf{fv}(P))$ thunk

$run \stackrel{\text{def}}{=} \lambda x.(x\,())$ run

Fig. 1. Syntax

(beta) $\quad (\lambda(x : t).P)V \longrightarrow P\{V/x\}$

(conn) $\quad !a(x).P \mid \overline{a}(z).Q \longrightarrow !a(x).P \mid (\nu\kappa)(P\{\kappa_0/x\} \mid Q\{\kappa_1/z\})$ κ_0, κ_1 fresh

(comm) $\quad \kappa_i(x).P \mid \overline{\kappa}_j\langle V \rangle.Q \longrightarrow P\{V/x\} \mid Q \quad i \neq j$

(label) $\quad \kappa_j \triangleright \{l_1 : P_1; \ldots; l_n : P_n\} \mid \kappa_i \triangleleft l_m.P \longrightarrow P_m \mid P \quad i \neq j,\ 1 \leq m \leq n$

$$\text{(app-l)}\ \frac{P \longrightarrow P'}{PQ \longrightarrow P'Q} \qquad \text{(app-r)}\ \frac{Q \longrightarrow Q'}{VQ \longrightarrow VQ'}$$

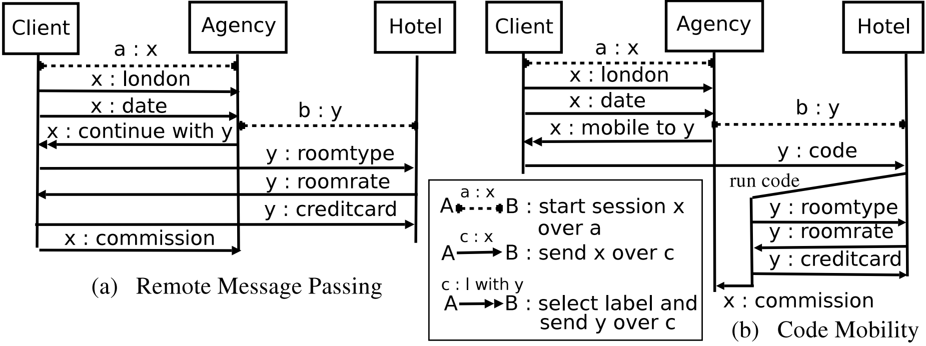
$$\text{(par)}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \text{(res)}\ \frac{P \longrightarrow P'}{(\nu\tilde{a} : \tilde{\sigma})(\nu\tilde{\kappa})P \longrightarrow (\nu\tilde{a} : \tilde{\sigma})(\nu\tilde{\kappa})P'} \qquad \text{(str)}\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Fig. 2. Reduction

call-by-value reduction relation, denoted $\longrightarrow$, is a binary relation from closed terms to closed terms, defined by the rules in Fig. 2. Rule (conn) establishes a new session between server and client via shared name u; fresh κ_0 and κ_1 are instantiated, and the server stays as it is, waiting another interaction. Rule (comm) transmits values between the private session channels. Note that a session channel can be sent and received (when $V = k$), with which various protocols are expressed, allowing complex nested and private structured communications. This interaction is called *higher-order session passing* (delegation). Rule (label) selects P_m (a communication version of the case reduction in the λ-calculus). We use the standard structure rules $\equiv$ such as $(\nu\kappa)P \mid Q \equiv (\nu\kappa)(P \mid Q)$ if $\kappa_{i\in\{0,1\}} \notin \mathsf{fn}(Q)$ (see [1]).

2.2 Example: Business Protocol with Code Mobility

We show a simple protocol which contains essential features by which we can demonstrate the expressivity of the code mobility and session primitives for the HOπ-calculus;



(a) Remote Message Passing

(b) Code Mobility

Fig. 3. Sequence Diagram for Hotel Booking

it consists of a combination of session establishing, code mobility, session delegation and branching. This extends a typical collaboration pattern that appears in many web service business protocols [25,5] to code mobility. In Fig. 3, we show the sequence diagram for a protocol which models a hotel booking: first, `Booking Agency` and `Client` initiate interaction at session x over channel a; then `Client` starts exchanging a series of information with `Agency`; during this initial communication, `Agency` calculates its Round Trip Time (RTT) between `Client` and `Agency`; `Agency` selects an appropriate `Hotel` and creates a new session y over channel b with that `Hotel`. If the RTT is short (Fig. 3 (a)), then `Agency` delegates to `Client` its part of the remaining activity with `Hotel`, by sending session channel y; then `Client` and `Hotel` continue negotiations by message passing. If the RTT is long (Fig. 3 (b)), since many remote interactions increase the communication time as well as danger of communication failures, `Agency` asks back `Client` to *send mobile code* which contains the communication of the `Client`'s room plan and negotiation behaviour. `Agency` sends the code to `Hotel`, then `Hotel` runs it locally, finishing a series of interactions in its location. Finally `Agency` receives a commission fee (10 percent of the room rate) via session x, concluding the transaction.

The given scenario is straightforwardly encoded in our calculus, where session primitives make the structure of interactions clearer; we omit the subject of the intermediate communications within the same session e.g. $x \triangleleft l.x\langle v\rangle.x(y).P$ is written as $x \triangleleft l;\langle v\rangle;(y).P$. `Agency` first initiates at a and starts the interactions with `Client`; then it initiates at b and establishes session y; next it invokes either label cont or label move in `Client` depending on the RTT and sends y (higher-order session passing) to it, and waits for completion of the transaction between `Client` and `Hotel` at x ("if-then-else" can be encoded using branching, and we use other base types and their operators).

$$!a(x).x(area);\ldots\overline{b}(y).\texttt{if}\ \mathrm{rtt} < 100\ \texttt{then}\ x \triangleleft \mathsf{cont};\langle y\rangle;(z).P \quad (1)$$

$$\texttt{else}\ x \triangleleft \mathsf{move};\langle y\rangle;(z).P \quad (2)$$

`Client` requests a service at a and starts a series of interactions with `Agency`, and either continues the remaining activity with `Hotel` or sends the code (a thunk in Line 4). Note that `Client` can safely send back the commission fee to `Agency` because the return message $\overline{x}\langle z \times 0.1\rangle$ which uses session channel x is embedded in the thunk.

$$\overline{a}(x).\overline{x}\langle london\rangle;..x \rhd \{\ \mathsf{cont}: (y).y \lhd \mathsf{cont}; \langle roomtype\rangle; (z);..\overline{x}\langle z \times 0.1\rangle\ ; \quad (3)$$
$$\mathsf{move}: (y).y \lhd \mathsf{move}; \langle \ulcorner \overline{y}\langle roomtype\rangle; (z);..\overline{x}\langle z \times 0.1\rangle \urcorner\rangle\} \quad (4)$$

`Hotel` performs the interactions with `Agency` and `Client` via a single session at y (by the facility of higher-order session). In Line 6, the code sent by `Client` is run locally.

$$!b(y).y \rhd \{\ \mathsf{cont}: (z); \langle \mathrm{roomrate}(z)\rangle; ...Q\ ; \quad (5)$$
$$\mathsf{move}: (code).(run\ code \mid y(z); \langle \mathrm{roomrate}(z)\rangle; ...Q)\} \quad (6)$$

This encoding shows a couple of subtle points whose slight modification breaks the session structures. First, in Line 4, if we send code which does not complete the session, then the protocol is broken: e.g. if we have interactions at y (say $\overline{y}\langle w\rangle$) after sending a thunk in Line 4 in `Client`, the session at y will appear in the three threads (two in `Hotel`, one in `Client`), so the session at y is interfered with and values may get mixed up. Secondly, in Line 6, if we have two or more applications (say $run\ code \mid run\ code$) instead of one $run\ code$, it again breaks the session structure (both at y and x). Finally, if the code is not ran in Line 6 (like $(\lambda x.\mathbf{0})code$ instead of $run\ code$), the receiver $y(z); \langle \mathrm{roomrate}(z)\rangle; ...Q$ cannot find a matching output. Hence the variable $code$ must appear exactly once and become instantiated into a process exactly once.

3 The First System: Higher-Order Linear Typing

3.1 Types

This section presents the first session system based on linear typing for higher-order functions. The syntax of types is given below.

$$\text{Term}\ \ \tau ::= t \mid \diamond \qquad \text{Chan}\ \ \sigma ::= \mathsf{begin}.\alpha \qquad \text{Val}\ \ t ::= \mathsf{unit} \mid t \to \tau \mid t \xrightarrow{1} \tau \mid \sigma \mid \alpha$$
$$\text{Session}\ \ \alpha ::= ![t].\alpha \mid ?[t].\alpha \mid \oplus[l_1:\alpha_1;\ldots;l_n:\alpha_n] \mid \&[l_1:\alpha_1;\ldots;l_n:\alpha_n] \mid \mathsf{end}$$

It is an integration of the types from the simply typed λ-calculus with unit and the session types from the π-calculus, with the exception of linear functional types, $t \xrightarrow{1} \tau$, which represent *functions to be used exactly once*. *Term types*, ranging over τ, include all value types and the process type $\diamond$. *Channel types*, ranging over σ, take the shape $\mathsf{begin}.\alpha$. *Session types* range over $\alpha, \beta, \gamma...$ In $\mathsf{begin}.\alpha$, begin represents the start of the session, while end represents its termination. *Value types* consist of the unit type, the function types, the linear function types and the channel and session types. Note that linear annotations are attached only to function types. In the session types, $![t].\alpha$ represents the output of a value typed by t followed by a session typed by α; $?[t]$ is its dual. $\oplus[l_1:\alpha_1;\ldots;l_n:\alpha_n]$ is the selection type on which one of the labels l_i can be sent, with the subsequent session typed by α_i; $\&[l_1:\alpha_1;\ldots;l_n:\alpha_n]$ is its dual called the branching type. We often write $\&[l_i:\alpha_i]_{i\in I}$ and $\oplus[l_i:\alpha_i]_{i\in I}$ for branching and selection types, $\ulcorner\tau\urcorner$ for `unit` $\to \tau$ and $\ulcorner\tau\urcorner^1$ for `unit` $\xrightarrow{1} \tau$. end is often omitted. Each session type α has a *dual* type, denoted by $\overline{\alpha}$, which describes complementary behaviour. This is inductively defined as: $\overline{![t].\alpha} = ?[t].\overline{\alpha}$, $\overline{\oplus[l_1:\alpha_1;\ldots l_n:\alpha_n]} = \&[l_1:\overline{\alpha_1};...;l_n:\overline{\alpha_n}]$, $\overline{?[t].\alpha} = ![t].\overline{\alpha}$, $\overline{\&[l_1:\alpha_1;\ldots l_n:\alpha_n]} = \oplus[l_1:\overline{\alpha_1};...;l_n:\overline{\alpha_n}]$, and $\overline{\mathsf{end}} = \mathsf{end}$.

(Common)

(Shared) $\dfrac{t \neq t' \xrightarrow{1} \tau}{\Gamma, u\colon t;\emptyset;\emptyset \vdash u : t}$ (Session) $\dfrac{}{\Gamma; k:\alpha;\emptyset \vdash k:\alpha}$ (LVar) $\dfrac{}{\Gamma, x: t \xrightarrow{1} \tau;\emptyset;\{x\} \vdash x : t \xrightarrow{1} \tau}$

(Function)

(Base) $\dfrac{}{\Gamma;\emptyset;\emptyset \vdash () : \mathtt{unit}}$ (Abs) $\dfrac{\Gamma, x\colon t;\Sigma;\mathcal{S} \vdash P:\tau \quad (\star)}{\Gamma;\Sigma;\mathcal{S}\setminus x \vdash \lambda(x{:}t).P : t \to \tau}$ (Abs$_\mathcal{S}$) $\dfrac{\Gamma;\Sigma, x\colon \alpha;\mathcal{S} \vdash P:\tau}{\Gamma;\Sigma;\mathcal{S} \vdash \lambda(x{:}\alpha).P : \alpha \to \tau}$

(App) $\dfrac{\Gamma;\Sigma_1;\mathcal{S}_1 \vdash P : t \xrightarrow{1} \tau \quad \Gamma;\Sigma_2;\mathcal{S}_2 \vdash Q : t \quad (\dagger)}{\Gamma;\Sigma_1,\Sigma_2;\mathcal{S}_1,\mathcal{S}_2 \vdash PQ : \tau}$ (Sub) $\dfrac{\Gamma;\Sigma;\mathcal{S} \vdash P : t \to \tau}{\Gamma;\Sigma;\mathcal{S} \vdash P : t \xrightarrow{1} \tau}$

(**Process**)

(Nil) $\dfrac{\Sigma = \{\tilde{k} : \widetilde{\mathsf{end}}\}}{\Gamma;\Sigma;\emptyset \vdash \mathbf{0} : \diamond}$ (New) $\dfrac{\Gamma, a\colon \sigma;\Sigma;\mathcal{S} \vdash P:\diamond}{\Gamma;\Sigma;\mathcal{S} \vdash (\nu a\colon \sigma)P : \diamond}$ (Newκ) $\dfrac{\Gamma;\Sigma,\kappa_i\colon \alpha,\kappa_j\colon \overline{\alpha};\mathcal{S} \vdash P:\diamond}{\Gamma;\Sigma;\mathcal{S} \vdash (\nu\kappa)P : \diamond}$

(Acc) $\dfrac{\Gamma;\emptyset;\emptyset \vdash u : \mathsf{begin}.\overline{\alpha} \quad \Gamma; x\colon \alpha;\emptyset \vdash P : \diamond}{\Gamma;\emptyset;\emptyset \vdash !u(x).P : \diamond}$ (Req) $\dfrac{\Gamma;\emptyset;\emptyset \vdash u : \mathsf{begin}.\alpha \quad \Gamma;\Sigma, x\colon \alpha;\mathcal{S} \vdash P : \diamond}{\Gamma;\Sigma;\mathcal{S} \vdash \overline{u}(x).P : \diamond}$

(Rec) $\dfrac{\Gamma, x\colon t;\Sigma, k\colon \alpha;\mathcal{S} \vdash P:\diamond \quad (\star)}{\Gamma;\Sigma, k\colon ?[t].\alpha;\mathcal{S}\setminus x \vdash k(x).P : \diamond}$ (Rec$_\mathcal{S}$) $\dfrac{\Gamma;\Sigma, k\colon \alpha', x\colon \alpha;\mathcal{S} \vdash P:\diamond}{\Gamma;\Sigma, k\colon ?[\alpha].\alpha';\mathcal{S} \vdash k(x).P : \diamond}$

(Send) $\dfrac{\Gamma;\Sigma_1;\mathcal{S}_1 \vdash P:\diamond \quad \Gamma;\Sigma_2;\mathcal{S}_2 \vdash V : t \quad k\colon \alpha \in \Sigma_{i\in\{1,2\}} \quad (\dagger)}{\Gamma;(\Sigma_1,\Sigma_2)\setminus\{k:\alpha\}, k: ![t].\alpha;\mathcal{S}_1,\mathcal{S}_2 \vdash \overline{k}\langle V\rangle.P : \diamond}$ (Par) $\dfrac{\Gamma;\Sigma_{1,2};\mathcal{S}_{1,2} \vdash P_{1,2} : \diamond}{\Gamma;\Sigma_1,\Sigma_2;\mathcal{S}_1,\mathcal{S}_2 \vdash P_1 \mid P_2 : \diamond}$

(Bra) $\dfrac{\Gamma;\Sigma, k\colon \alpha_i;\mathcal{S} \vdash P_i : \diamond \quad (\forall i \in I)}{\Gamma;\Sigma, k : \&[l_i : \alpha_i]_{i\in I};\mathcal{S} \vdash k \rhd \{l_i : P_i\}_{i\in I} : \diamond}$ (Sel) $\dfrac{\Gamma;\Sigma, k : \alpha_j;\mathcal{S} \vdash P : \diamond \quad j \in I}{\Gamma;\Sigma, k : \oplus[l_i : \alpha_i]_{i\in I};\mathcal{S} \vdash k \lhd l_j.P : \diamond}$

$(\star)$ if $t = t' \xrightarrow{1} \tau'$ then $x \in \mathcal{S}$. $(\dagger)$ if $t = t' \to \tau'$ then $\Sigma_2 = \mathcal{S}_2 = \emptyset$.

Fig. 4. Session Typing based on Linear Types

3.2 Linear Higher-Order Typing System

We first define the two kinds of finite mappings for environments:

$$\Gamma ::= \emptyset \mid \Gamma, u:\sigma \mid \Gamma, x:\mathsf{unit} \mid \Gamma, x:t \to \tau \mid \Gamma, x:t \xrightarrow{1} \tau \qquad \Sigma ::= \emptyset \mid \Sigma, k:\alpha$$

Γ is a mapping, associating value types (*except session types*) to identifiers. Σ is a mapping from session channels to session types that records precise usage information for all free session channels in a term, so that the cumulative result can be compared with the expected session type. In addition, we use a set of linear variables ranged over $\mathcal{S}$, $\mathcal{S}'$, ... to ensure linear usage of function terms that may contain session channels. Σ, Σ'

and $\mathcal{S}, \mathcal{S}'$ denote disjoint-domain unions. $\Gamma, u : \sigma$ means $u \notin \mathrm{dom}(\Gamma)$. Then the typing judgement takes the shape:

$$\Gamma; \Sigma; \mathcal{S} \vdash P : \tau$$

which is read: under a global environment Γ, a term P has a type τ with session usages described by Σ and linear variables specified by $\mathcal{S}$. We say the judgement is *well-formed* if $\mathrm{dom}(\Gamma) \supseteq \mathcal{S}$ and $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Sigma) = \emptyset$. The typing system is given in Fig. 4. In each rule, we assume the environments of the consequence are defined.

In the first group, **(Common)**, (Shared) is an introduction rule for identifiers with shared types, i.e. neither $t' \xrightarrow{1} \tau$ or α. (Session) is for session channels and (LVar) is for linear variables, recording k in Σ and x in $\mathcal{S}$, respectively.

The second group, **(Function)**, comes from the simply typed linear λ-calculus. In (Abs), the side condition ($\star$) ensures that the formal parameter x, to be substituted with the received function, appears in the linear variables' premise. In the conclusion, we remove x from the function environment. (Abs_S) is an abstraction rule for session channels. (App) is the rule for application; the side condition ($\dagger$) ensures that when the right term is of shared function type, it is required not to have free session channels or linear variables. The conclusion says that P and Q's session environments and linear variable sets are disjoint. (Sub) is a subsumption rule to lift from the shared $t \rightarrow \tau$ to linear function $t \xrightarrow{1} \tau$. The converse is unsafe.

The final group, **(Process)**, are for processes integrated with linear functional typing. In (Nil), we start from the session environment only with end-usages and the empty linear variable set. (New) and (Newκ) hide a shared name and a pair of session channels, respectively. The latter erases, in the session environment, complementary communication patterns for the two endpoints of κ, in order to ensure compatible dyadic interactions. (Acc) and (Req) are for initiating sessions. (Acc) forbids the use of any *free* linear identifier because of replication. The type expected for the session channel is dual ($\overline{\alpha}$) to that portion of the declared session type for the shared identifier. In (Req), it is used as it is (α). (Rec) handles the reception (input) of values. Just as (Abs), if received values have a linear function type, x should be recorded to ensure its linear usage in P. The relevant consumption is composed in the conclusion's session environment, in a way that agrees with the protocol. (Rec_S) is for the input of session channels.

(Send) is the most complex rule, integrating session typing and linear typing. Firstly, ($\dagger$), as in (App), enforces safety when sending linear functions. Secondly $k\colon \alpha \in \Sigma_{i \in \{1,2\}}$ means either Σ_1 or Σ_2 contains the complete session $k\colon \alpha$ (since Σ_1, Σ_2 is defined in the conclusion). When $k\colon \alpha \in \Sigma_1$ and V has a functional type, it ensures that all occurring session channels within V being sent are complete (i.e. suffixed with end). Hence they cannot occur in the continuation P, because, if they did, we would have a race condition between the receiver of V and P, w.r.t. communications over these common channels, as noted in the example in § 2.2. This condition forces V to be k itself when it has the session type α, uniformly generalising the corresponding rule in the session types [13,20,28,16]. This is important since, in the presence of higher-order mobility, the sent code containing k can be executed locally and privately in the receiver side: `Client` in the example in § 2.2 becomes typable with this general rule. In the conclusion, we delete k in either Σ_1 or Σ_2, and the relevant consumption is recorded in the conclusion's session environment. Note the function environments are disjoint. In (Par), we parallel-compose

two processes, assuming disjointness of session environments and linear variable sets as in (App). (Bra) and (Sel) are the standard rules for branching and selection from [16].

3.3 Type Soundness and Type Safety

The typed processes enjoy type soundness and type safety. We have the standard weakening and strengthening for Γ (but not for Σ and $\mathcal{S}$). Then the substitution lemmas follow.

Lemma 3.1 (Substitution Lemma)

1. *Suppose* $\Gamma, x\colon t; \Sigma_1; \mathcal{S}_1 \vdash P : \tau$ *and* $\Gamma; \Sigma_2; \mathcal{S}_2 \vdash V : t$ *with* $t \neq t' \to \tau'$, $x \in \mathsf{fv}(P)$, *and* Σ_1, Σ_2 *and* $\mathcal{S}_1, \mathcal{S}_2$ *are defined. Then* $\Gamma; \Sigma_1, \Sigma_2; \mathcal{S}_1 \setminus x, \mathcal{S}_2 \vdash P\{^V/_x\} : \tau$.
2. *Assume* $\Gamma, x\colon t' \to \tau'; \Sigma; \mathcal{S} \vdash P : \tau$ *and* $\Gamma; \emptyset; \emptyset \vdash V : t' \to \tau'$. *Then* $\Gamma; \Sigma; \mathcal{S} \vdash P\{^V/_x\} : \tau$.
3. *Suppose* $\Gamma; \Sigma, x : \alpha; \mathcal{S} \vdash P : \tau$ *and* $k \notin \mathrm{dom}(\Gamma, \Sigma)$. *Then* $\Gamma; \Sigma, k : \alpha; \mathcal{S} \vdash P\{^k/_x\} : \tau$.

Before stating the main theorems, we introduce the important notion of *balanced* session environments [13]. Clearly, typability over arbitrary session environments is not closed under reduction. For example, the process $\overline{\kappa_0}\langle \mathtt{true}\rangle \mid \kappa_1(x).\overline{\kappa_i'}\langle x+1\rangle$ is typable, but it reduces to $\overline{\kappa_i'}\langle \mathtt{true}+1\rangle$, leading to a run-time error. Hence we allow only typings where the two ends of a channel are of dual types. Formally, we say that a session environment Σ is *balanced* if whenever $\kappa_i\colon \alpha, \kappa_j\colon \beta \in \Sigma$, then $\alpha = \overline{\beta}$.

Theorem 3.2 (Type Soundness)

1. *Suppose* $\Gamma; \Sigma; \mathcal{S} \vdash P : \diamond$ *with* Σ *balanced. Then* $P \equiv P'$ *implies* $\Gamma; \Sigma; \mathcal{S} \vdash P' : \diamond$.
2. *Suppose* $\Gamma; \Sigma; \emptyset \vdash P : \tau$ *with* Σ *balanced. Then* $P \longrightarrow P'$ *implies* $\Gamma; \Sigma'; \emptyset \vdash P' : \tau$ *with* Σ' *balanced.*

We now formalise type safety. First, a *k-process* is a prefixed process with subject k (such as $k(x)$ and $\overline{k}\langle V\rangle$). Next, a κ*-redex* is a parallel composition of two dual processes, of the form $(\overline{\kappa_i}\langle V\rangle.P \mid \kappa_j(x).Q)$ or $(\kappa_i \lhd l_m.P \mid \kappa_j \rhd \{l_1 : Q_1; \cdots l_n : Q_n\})$ with $1 \leq m \leq n$. Then we say P is an *error* if $P \equiv (\nu\tilde{a})(\nu\tilde{\kappa})(Q \mid R)$ where Q is, for some κ, the $|$-composition of *either* two κ-processes that do not form a κ-redex, *or* three or more κ-processes. We then have:

Theorem 3.3 (Type Safety). *A typable process* $\Gamma; \Sigma; \mathcal{S} \vdash P : \diamond$ *with balanced* Σ *never reduces into an error.*

Typing Hotel Booking Example. Using the typing system, we can now type the hotel booking example in § 2.2, guaranteeing its type safety. `Agent` has the following types at a and b.

$$a : \mathsf{begin}.![\mathtt{string}]\ldots \oplus [\mathsf{rtt} < 100 :\ \alpha\ ;\ \mathsf{rtt} \geq 100 : \alpha\ ],\ b : \mathsf{begin}.![\beta].\mathsf{end}$$
$$\text{with } \alpha = \&[\mathsf{cont} : ?[\beta].![\mathtt{int}].\mathsf{end}\ ;\ \mathsf{move} : ?[\beta].![\mathtt{int}].\mathsf{end}]$$
$$\text{and } \beta = \&[\mathsf{cont} : ![\mathtt{string}].?[\mathtt{int}]\ldots\mathsf{end}\ ;\ \mathsf{move} : ![\ulcorner\diamond\urcorner^1].\mathsf{end}]$$

Note that the type of a is dualised because a is used as the input in `Agent` (see (Acc)). α consists of higher-order session passing, and the thunk has a linear arrow type. `Client` and `Hotel` just have the dual of `Agent`'s type at a and the dual of `Agent`'s type at b, respectively. Note that in `Client`, subject y is shared in the sent code V, which is typed by (Send) with a general side condition $k : \alpha \in \Sigma_2$ explained in § 3.2.

4 The Second System: Fine-Grained Process Typing

Linear variables in the previous system "might be instantiated by a function which contains free session channels, hence it should occur exactly once": if we have *prior* knowledge as to channel capabilities with which each functional variable (hence any code instantiated into it) is associated, then we might have more flexible control over migrating code that holds session capabilities. This motivates the use of the fine grained process typing introduced in [27,26,15]. Consider the following server which receives thunked processes via shared channel a.

$$\mathsf{Serv}(a) \;=\; !a(x).x(y\colon \tau).run\ y \tag{7}$$

Since accepting arbitrary processes for execution obviously breaks access control of local resources, one might wish to restrict the behaviour of incoming code so that it can only access some specified channels. In [27,26,15], we introduced a type discipline which can control the effect of migrating code, by assigning a different type to each process depending on its intended use, so that a process can use a typed inputting channel (τ at a in (7) above) to detect, for example, malicious behaviour of received code via static type checking. A type for representing capability is given as a finite channel environment Δ, prescribing channel usage of each process.

$$\Gamma \vdash P \;:\; \Delta$$

This judgement means "P accesses channels at most as specified by Δ under global environment Γ". For example, under appropriate $\Gamma \supset \{b\colon \sigma, c\colon \sigma\}$ with $\sigma = \mathsf{begin}.![\mathtt{nat}].\mathsf{end}$, a client may be assigned a different type depending on its destination.

$$\Gamma \vdash \overline{b}(x).\overline{x}\langle 1\rangle : \{b\colon \sigma\} \quad \text{and} \quad \Gamma \vdash \overline{c}(x).\overline{x}\langle 2\rangle : \{c\colon \sigma\}$$

Then the following indicates a server which only accepts a process which accesses at most the specified resource, b.

$$\mathsf{Serv}(a) \;=\; !a(x).x(y\colon \ulcorner b\colon \sigma\urcorner).run\ y \tag{8}$$

Using the type system in [27,26,15], one can check $\mathsf{Serv}(a) \mid \overline{a}(x).\overline{x}\langle\ulcorner\overline{b}(x).\overline{x}\langle 1\rangle\urcorner\rangle$ is typable while $\mathsf{Serv}(a) \mid \overline{a}(x).\overline{x}\langle\ulcorner\overline{c}(x).\overline{x}\langle 1\rangle\urcorner\rangle$ is not. Using process types with session capabilities, we can type-check that the following process is illegal:

$$k(y\colon \ulcorner k'\colon ![\mathtt{nat}]\urcorner).(run\ y \mid run\ y) \tag{9}$$

since $run\ y$ has a process type $\Delta = \{k'\colon ![\mathtt{nat}]\}$, and Δ and Δ are not disjoint, so two $run\ y$ must not be composed. Now we no longer require linear annotation on functional types. Moreover the additional type information leads to a larger typability than the previous system. For example, $k(y\colon \ulcorner k'\colon ![\mathtt{nat}]\urcorner).(run\ y \mid (\lambda z.\mathbf{0})y)$, $(\lambda x.\mathbf{0})\kappa_0 \mid \kappa_0(z).\mathbf{0} \mid \overline{\kappa_1}\langle 1\rangle$, and more interestingly $(\lambda x.\overline{k}\langle 1\rangle.run\ x)(\ulcorner\overline{k}\langle()\rangle.\mathbf{0}\urcorner)$ which do not destroy session communication but are untypable in the previous one become typable since the resulting process types are balanced.

4.1 Types

The second typing system introduced below is built on the fine-grained types of [27,26]. The syntax of environments and types is given below.

$$
\begin{array}{rll}
\text{Env} & \Gamma ::= \emptyset \mid \Gamma, x\colon t \mid \Gamma, u\colon \sigma \quad \Delta ::= \emptyset \mid \Delta, u\colon \sigma & \text{Term} \quad \tau ::= t \mid \Delta \\
\text{Chan} & \sigma ::= \mathsf{begin}.\alpha \mid \alpha \qquad \text{Func} \quad t ::= \mathsf{unit} \mid t \to \tau \mid \Pi x\colon \sigma.\tau & \\
\text{Session} & \alpha ::= ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha \mid ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha \mid \oplus[l_i\colon \alpha_i]_{i\in I} \mid \&[l_i\colon \alpha_i]_{i\in I} \mid \mathsf{end} &
\end{array}
$$

These types are from the first system except for the introduction of fines-grained process types Δ, functional dependent types $\Pi x\colon \sigma.\tau$ and channel dependent $\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}$. Note from this system, $u, v, w, \ldots$ (resp. σ) include session names and variables (resp. session types), but τ do not include channels.[1] In $\Pi x\colon \sigma.\tau$, we allow the type τ to contain occurrences of the channel variable x; then x in τ is bound. Note $\sigma \to \tau$ is a special case of $\Pi x\colon \sigma.\tau$ with $x \notin \mathsf{fv}(\sigma)$. A process type Δ, assigned to a process, is a mapping from a finite subset of identifiers to channel types.

A channel type incorporates dependent quantification, and has the form $\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}$ indicating a vector of channels typed by $\sigma_1, .., \sigma_n$ and a vector of higher-order values typed by $t_1, .., t_m$; free occurrences of x_i in $\sigma_{i+1}, \ldots, \sigma_n$ as well as $t_1, \ldots, t_m$ are bound occurrences. We write $\sigma_1, \ldots, \sigma_n, t_1, \ldots, t_m$ for $\Pi(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n)t_1, \ldots, t_m$ if $x_1, \ldots, x_n \notin \mathsf{fv}(\sigma_1, \ldots, \sigma_n, t_1, \ldots, t_m)$. Under this abbreviation, $?[t].\beta$ is subsumed to the case $n = 0$, and $?[\alpha].\beta$ to the case $\sigma_1 = \alpha$ and $m = 0$. The set of free names and variables are defined in the standard way [26,1]. The sets of free variables/channels incorporate those occurring in annotating types. For example, we have $\mathsf{fv}(\lambda(x\colon t).P) = (\mathsf{fv}(t) \cup \mathsf{fv}(P)) \setminus x$. Substitution by channels $P\{u/x\}$ affects not only terms but also types which annotate bound variables: when the channel u is substituted for x in a process type Δ, then the types σ of x and σ' of u are joined as: $\{u_1\colon \sigma_1, \ldots, u_n\colon \sigma_n\}\{V/x\} = \cup_i\{u_i\{V/x\}\colon \sigma_i\{V/x\}\}$. Others are defined homomorphically. Duality is defined by adding $\overline{?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha} = ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\overline{\alpha}$ and $\overline{![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha} = ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\overline{\alpha}$; others remain unchanged.

4.2 Fine-Grained Process Typing System

The key typing rules are given in Fig. 5, and use two kinds of judgements: the main is $\Gamma \vdash P \triangleright \Delta$, which reads "under the environment Γ, process P has an interface type Δ". Also we have $\Gamma \vdash u\colon \sigma$, which reads as "a channel u has a type σ under Γ" and the standard well-formedness $\Gamma \vdash \mathtt{Env}$ and $\Gamma \vdash \tau : \mathtt{tp}$ for environments and types following [27,26] (which are left to [1]). For channel inference, we define the ordering $\succ$ on channel types as the smallest partial order such that: $![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha \succ \alpha$ and $\oplus[l_1 : \alpha_1; \ldots l_n\colon \alpha_n] \succ \alpha_i$; dually for input and branching types.

The inference rules are a combination of [26] and the session typing system of the π-calculus. We use the notation $\Delta \cdot u\colon \sigma$ for $\Delta \cup \{u\colon \sigma\}$ if $\sigma = \mathsf{begin}.\alpha$; $\Delta, u\colon \sigma$ otherwise. We extend this to $\Delta \cdot \Delta'$; and $\tilde{u}\colon \tilde{\sigma}$ which means $u_1 : \sigma_1 \cdots u_n\colon \sigma_n$.

[1] For simplicity of presentation, the tail type τ does not include the channel type σ. This inclusion can be straightforwardly formalised by using the standard type equality approach [3].

(Chan)
$$\frac{\Gamma, u\colon \sigma, \Gamma' \vdash \mathtt{Env} \quad \sigma \succ \sigma'}{\Gamma, u\colon \sigma, \Gamma' \vdash u : \sigma'}$$

(Abs$_N$)
$$\frac{\Gamma, x\colon \sigma \vdash P : \tau}{\Gamma \vdash \lambda(x{:}\sigma).P : \Pi x\colon \sigma.\tau}$$

(App$_N$)
$$\frac{\Gamma \vdash P : \Pi x\colon \sigma.\tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash P\,u : \tau\{u/x\}}$$

(Nil)
$$\frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \mathbf{0} : \emptyset}$$

(Par)
$$\frac{\Gamma \vdash P_{1,2} : \Delta_{1,2}}{\Gamma \vdash P_1 \mid P_2 : \Delta_1 \cdot \Delta_2}$$

(Weak)
$$\frac{\Gamma \vdash P : \Delta \quad \Gamma \vdash u\colon \sigma \quad \sigma \in \{\mathsf{begin}.\alpha, \mathsf{end}\} \quad u \notin \mathrm{dom}(\Delta)}{\Gamma \vdash P : \Delta, u\colon \sigma}$$

(Rec)
$$\frac{\Gamma \vdash k : ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha \qquad \Gamma, \tilde{x}\colon \tilde{\sigma}, \tilde{y}\colon \tilde{t} \vdash P : \Delta, \tilde{x}\colon \tilde{\sigma}, k\colon \alpha}{\Gamma \vdash k(\tilde{x}\colon \tilde{\sigma}, \tilde{y}\colon \tilde{t}).P : \Delta, k\colon ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha}$$

(Send)
$$\frac{\Gamma \vdash k : ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha \qquad \Gamma \vdash P : \Delta \quad \{k\colon \alpha\} \subseteq \Delta \cdot \tilde{v}\colon \tilde{\sigma} \qquad \Gamma \vdash V_j : t_j\{\tilde{v}/\tilde{x}\} \quad \Gamma \vdash v_i : \sigma_i\{\tilde{v}/\tilde{x}\}}{\Gamma \vdash \overline{k}\langle \tilde{v}, \tilde{V}\rangle.P : \Delta \cdot \tilde{v}\colon \tilde{\sigma} \setminus k, k\colon ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha}$$

Fig. 5. Session Typing based on Fine-Grained Process Types

(Chan) uses $\succ$ to infer *shorter* types for sessions than the type of u declared in the environment Γ. The rules for channel abstractions, (Abs$_N$) and (App$_N$), are defined following [26]. In (Nil), we start from the empty interface, and in (Par), we merge two interfaces together. The rule (Weak) corresponds to the process subsumption rule; since $\Gamma \vdash P : \Delta$ means "P would access channels specified at most by Δ", we can increment its interface. Note that we can*not* weaken session channels except end.

(Rec) is a combination of the input rule for session types and that in [26]. This single rule subsumes both the value input rule and the session channel input rule (recall the abbreviation in the previous paragraph). The first assumption ensures u can input channels typed by σ_i and higher-order values typed by t_j, and in the conclusion, the free occurrences of $\tilde{x}$ in both P and t_j are bound (hence t_j is dependent on $\tilde{x}$), resulting in the process type Δ with a new session type $?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha$ at k (note $k \notin \mathrm{dom}(\Delta)$). (Send) is again a combination with the output rule in [26] (see also (Send) for the first system): the first assumption ensures u outputs a pair of names typed by σ_i and higher-order values typed by t_j. The third assumption says that k is either sent name v_i or a free name in P. The first part of the arguments is v_i, then the second part of the arguments should have type $t_j\{\tilde{v}/\tilde{x}\}$ since x_i binds free occurrences of x_i in t_j. Then the effect of channel k and v_i should be recorded as a type of $\overline{k}\langle \tilde{v}, \tilde{V}\rangle$ because they will be used by the opponent input after interaction (note that we do *not* have to record the effect of V). Other rules (variable, unit, higher-order abstraction, application, hiding, accept, request, branching and selection) are standard and left to [1].

By essentially the same routine as in the proofs in [26], we obtain the following theorem. Note that Γ does not have to be balanced.

Theorem 4.1 (Type Soundness and Type Safety)

1. *Suppose* $\Gamma \vdash P : t$ *and* $P \longrightarrow P'$. *Then* $\Gamma \vdash P' : t$.
2. *Suppose* $\Gamma \vdash P : \Delta$ *with* Δ *balanced. Then* $P \equiv P'$ *implies* $\Gamma \vdash P' : \Delta'$ *with* Δ' *balanced.*
3. *Suppose* $\Gamma \vdash P : \Delta$ *with* Δ *balanced. Then* $P \longrightarrow P'$ *implies* $\Gamma \vdash P' : \Delta'$ *with* Δ' *balanced. In addition, P never reduces into an error.*

Typing Hotel Booking Example. We revisit the example in § 2.2. The only change from the previous types in § 3.2 is $![\ulcorner\diamond\urcorner^1]$ in β. This is changed to $![\Pi(x:\gamma_x, y:\gamma_y)\ulcorner\Delta\urcorner]$ with $\gamma_x = ![\mathtt{string}].?[\mathtt{int}]...\mathsf{end}$ and $\gamma_y = ![\mathtt{int}].\mathsf{end}$, and $\Delta = \{x:\gamma_x, y:\gamma_y\}$. Note that we also have to change the syntax in Line 4 from $y \lhd \mathsf{move}; \langle\ulcorner R\urcorner\rangle$ to $y \lhd \mathsf{move}; \langle x, y, \ulcorner R\urcorner\rangle$ since the type of the thunk is dependent on x and y. This suggests a trade-off between the two approaches. In the channel-dependent typing, we gain more flexibility by having more type information, but this in turn demands additional type annotation in programs. The approach based on linear typing does not need heavy annotations, though it allows the typability of a smaller, but probably pragmatically sufficient, class of programs. We may also refine the dependently typed approach with the existential types of [26,15] (this integration is straightforward, but requires more rules), in which case we do not have to declare session names explicitly. The syntax of the example is unchanged, and the type becomes $![\exists(x:\gamma_x, y:\gamma_y)\ulcorner\Delta\urcorner]$. The reader can also check the processes in the beginning of the section are typable: in the first process, $(\lambda x.\mathbf{0})y$ has the empty process type $\emptyset$ so that we can compose with *run* y by (Par). Similarly for the second. In the third, $(\lambda(x\colon t).\overline{k}\langle 1\rangle.\mathit{run}\ x)(\ulcorner\overline{k}\langle()\rangle.\mathbf{0}\urcorner)$ with $t = \ulcorner k\colon ![\mathtt{unit}].\mathsf{end}\urcorner$ has a process type $\Delta = \{k\colon ![\mathtt{nat}].![\mathtt{unit}].\mathsf{end}\}$ under environment Δ. These are untypable in the first system.

4.3 Comparison of the Two Systems

We conclude this section with a comparison of the two typing systems. The examples in the beginning of this section show the existence of terms typable in the second system but not in the first system introduced in § 3. A natural question is which subsystem of the second system can precisely characterise the first, i.e. a sound and complete embedding of the first system into a subset of the second system. Observing that it is linear functions that can inhabit those types with free session capabilities (e.g. $\lambda(x\colon\alpha).\overline{x}\langle 1\rangle$ of type $\Pi x\colon\alpha.x\colon\mathtt{nat}$ is not a linear function, while $\ulcorner\overline{x}\langle 1\rangle\urcorner$ of type $\ulcorner x\colon\mathtt{nat}\urcorner$ is linear), we introduce the following three functions.

- $\mathit{Erase}(P)$ erases the dependent binding from the input and output, and $\mathit{Erase}(\tau)$ erases the dependent binding from the functional and channel dependent types; and translates process types into $\diamond$; and puts the linear annotation to a functional type which has free session typings in its tail.
- $\mathit{Proc}(\tau)$ extracts the session environment Σ from τ.
- $\mathit{Lin}(\Gamma)$ extracts the linear variable set $\mathcal{S}$ from Γ.

For example, $\mathit{Erase}(k(\tilde{x}\colon\tilde{\sigma}, y\colon\tau).P) = k(y:\mathit{Erase}(\tau)).\mathit{Erase}(P)$, $\mathit{Erase}(\overline{k}\langle\tilde{v}, V\rangle.P) = \overline{k}\langle\mathit{Erase}(V)\rangle.\mathit{Erase}(P)$, $\mathit{Erase}(\Delta) = \diamond$, and if $\mathit{Proc}(t\to\tau)\neq\emptyset$, $\mathit{Erase}(t\to\tau) = \mathit{Erase}(t)\overset{1}{\to}\mathit{Erase}(\tau)$ else $\mathit{Erase}(t\to\tau) = \mathit{Erase}(t)\to\mathit{Erase}(\tau)$. $\mathit{Proc}(\Delta) = \{k:\mathit{Erase}(\alpha)\mid k:\alpha\in\Delta\}$, $\mathit{Proc}(\mathtt{unit}) = \mathtt{unit}$, $\mathit{Proc}(t\to\tau) = \mathit{Proc}(\tau)\setminus\mathit{Proc}(t)$, $\mathit{Proc}(\Pi x\colon\sigma.\tau) = \mathit{Proc}(\tau)\setminus x$, and $\mathit{Lin}(\Gamma) = \{x\mid\mathit{Erase}(\Gamma(x)) = t\overset{1}{\to}\tau\}$.

Next we re-formulate the rules for the arrow types to ensure that all session capabilities are preserved during β-reductions (which is a property of the first system): $t\to\tau$ is well-formed if $\mathit{Proc}(t)\subseteq\mathit{Proc}(\tau)$; and $\Pi x\colon\alpha.\tau$ is so if $x\colon\alpha\in\mathit{Proc}(\tau)$. We also replace $\Delta\cdot k\colon\alpha$ to mean $k\notin\mathsf{fn}(\Delta)\cup\mathsf{fv}(\Delta)$ in the rules for processes. We can now describe the corresponding side conditions directly using $\mathit{Proc}(\tau)$ and $\mathit{Lin}(\Gamma)$ instead of recording

Σ and $\mathcal{S}$. Below we say P is *initial* if a sent function appearing in P does not contain identifiers of shared channel types and variables of function types.

Theorem 4.2 (Embedding). *Below* $\Gamma \restriction \mathcal{S}$ *means* $\{u : \Gamma(u) \mid u \in \mathcal{S}\}$.

- *Suppose* $\Gamma \vdash P : \tau$ *is derived by the restricted system defined in this subsection. Then we have:* $Erase(\Gamma); Proc(\tau) \setminus \Sigma; \mathcal{S} \vdash Erase(P) : Erase(\tau)$ *where* $\mathcal{S} = Lin(\Gamma \restriction \mathsf{fv}(P))$ *and* $\Sigma = \{Proc(t) \mid x : t \in \Gamma \restriction \mathcal{S}\}$.
- *Suppose* $\Gamma; \emptyset; \emptyset \vdash P : \diamond$ *and* P *is initial. Then there exist* Γ', P' *and* Δ *such that* $\Gamma' \vdash P' : \Delta$ *with* $Erase(\Gamma') = \Gamma$, $Erase(P') = P$ *and* $\Delta \subset \Gamma'$ *in the restricted system.*

The first statement means that the session capabilities of P except those that appear in types of the linear variables in P are placed as Σ, and the linear variables in P are placed as $\mathcal{S}$ in the first system. The second statement says that the second system can always infer initial processes derived by the first one.[2]

5 Related and Future Work

This paper studies session types for higher-order processes using two different approaches and compares their typability. The robust formulations hinted by the linear and dependent λ-type theories [24,3] lead to new process typing systems for protocol validation. Straightforward extensions are recursive types [16,28], subtyping [13,26] and polymorphism [23,11]. In particular, recursive session types are useful to type various common "repetitive" protocols appearing in many practices [25,9]. For this extension, an explicit recursion construct in the form of the recursive agent $\mathtt{def}\ X(\tilde{x}\tilde{k}) = P\ \mathtt{in}\ M$ is introduced in [16,28]. In our calculus, this agent can be replaced by a more familiar syntax such as $\mathtt{letrec}\ x = P\ \mathtt{in}\ M$. The important constraint is that P cannot hold linear variables nor free session channels (i.e. $\Sigma = \mathcal{S} = \emptyset$), which does not reduce the expressivity by using parameterised processes as in [24]. By taking the approach in [28], we can construct the typing rule for the recursive agent, and can type scenarios with repetition, fully integrated with code mobility, see [1].

There is a large literature on linear and session types for both the λ-calculus and the π-calculus. Below we give the most closely related work, focusing on the linear typing system of the λ-calculus and on the session types for distributed and functional programming languages. See also [7,1,5,25,6] for discussions on other type disciplines of the π-calculus as well as on applications of session types.

Our first typing system is substructural [24] in the sense that for session environments Σ we do not allow weakening and contraction, ensuring that a session channel is recorded as having been used only when it actually occurs in session communication expressions. Similarly no structural transformations can apply to linear variable environments, ensuring that the occurrence of a variable manifests that it has indeed been used exactly once. The ways in which our typing system enforces linearity can be seen as an amalgamation of the two approaches in [24], retaining the simplicity of declarative systems, and the decidability of algorithmic ones. Contrary to the systems of [24],

[2] We can delete the initial condition if the shared channel type σ includes a recursive type [16].

there is no need of linear usage for other than functional types. Applying the techniques in [7,22], constructing its type inference system would be a straightforward task.

Relating to distribution, [10] studies session types for boxed ambients, preventing session interruption when an ambient crosses its boundary. One of the technical challenges of our work is to formalise sound typing systems for arbitrarily parameterised processes (i.e. λ-abstractions in processes with the full type hierarchy), which is not treated in ambient primitives. In [23] the authors define a concurrent multi-threaded functional language with sessions. It has an explicit multi-threading primitive (`fork`) and explicit stores. Their recent draft paper [14] further extends the language to a variant of session types where message sending is non-blocking. This is handled by explicitly storing an entry for the two endpoint channels in a buffer. Its functionality is the same as our use of two session channels indexed by 0 and 1 for distinguishing two endpoints (based on [13]). They simplify their previous type judgement which requires input and output environments in [23] by using the linear typing with `split` operator, which is more directly related to the original non-deterministic typing [24]. While a precise typability comparison is difficult due to our additional primitives and their operational semantics with buffers (which is essential for type soundness in their language), their work also shows a use of linear types for functional languages with sessions. Our comparison between the first and second systems via Theorem 4.2 makes the relationship between controlling usage of functional variables and effects of channel accessibility clear: the idea of "balanced" seems more suited to effect-like systems since our concern is well-formedness of process types, not intermediate functional types, while the linear typing approach is simpler and more tractable. This line of study is not explored in the previous literature.

As on-going work, we have been investigating the incorporation of session types and code mobility with Sockets in Java [17] and Web Service Description Languages [25,5]. From these experiences, we find that not only type checking by session types after writing a protocol, but also declaring its session types before compilation, greatly helps programmers implement error-free interactions. For developing programming language designs, the presented type theory needs further explorations, including its incorporation with advanced concurrent programming primitives such as exceptions, timeout and priority checking.

References

1. On-line Appendix of this paper. `http://www.doc.ic.ac.uk/~yoshida/hopis`
2. Ahern, A., Yoshida, N.: Formalising Java RMI with Explicit Code Mobility (A full version will appear in TCS). In: OOPSLA '05, pp. 403–422. ACM Press, New York (2005)
3. Aspinall, D., Hofmann, M.: Advanced Topics in Types and Programming Languages. In: Pierce, B.C. (ed.) chapter Dependent Types, MIT Press, Cambridge (2005)
4. Bonelli, E., Compagnoni, A., Gunter, E.: Correspondence Assertions for Process Synchronization in Concurrent Communications. J. of Funct. Progr. 15(2), 219–248 (2005)
5. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: ESOP'07. LNCS, vol. 4421, pp. 2–17. Springer, Berlin Heidelberg (2007)
6. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A theoretical basis of communication-centred concurrent programming. To be published by W3C (2006), Available at http://www.dcs.qmul.ac.uk/c̃arbonem/cdlpaper/workingnote.pdf

7. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session types for object-oriented languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
8. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A distributed object oriented language with session types. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 299–318. Springer, Heidelberg (2005)
9. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language Support for Fast and Reliable Message-based Communication in Singularity OS. In: Zwaenepoel, W. (ed.) EuroSys2006, ACM SIGOPS, pp. 177–190. ACM Press, New York (2006)
10. Garralda, P., Compagnoni, A., Dezani-Ciancaglini, M.: BASS: Boxed Ambients with Safe Sessions. In: Maher, M. (ed.) PPDP'06, pp. 61–72. ACM Press, New York (2006)
11. Gay, S.: Bounded polymorphism in session types. MSCS (To appear)
12. Gay, S., Hole, M.: Types and Subtypes for Client-Server Interactions. In: Swierstra, S.D. (ed.) ESOP 1999 and ETAPS 1999. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999)
13. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Info. 42(2/3), 191–225 (2005)
14. Gay, S., Vasconcelos, V.T.: A new approach to functional session types (October 2006)
15. Hennessy, M., Rathke, J., Yoshida, N.: SafeDpi: A language for controlling mobile code. Acta. Informatica 42(4-5), 227–290 (2005)
16. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: Hankin, C. (ed.) ESOP 1998 and ETAPS 1998. LNCS, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)
17. Hu, R.: Implementation of a distributed mobile Java. Master's thesis, Imperial College London (2006)
18. Neubauer, M., Thiemann, P.: An implementation of session types. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 56–70. Springer, Heidelberg (2004)
19. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms. PhD thesis, University of Edinburgh (1992)
20. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
21. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the Behavior of Objects and Components using Session Types. In: FOCLASA'02. ENTCS, vol. 68(3), Elsevier, Amsterdam (2002)
22. Vasconcelos, V.T.: A note on a typing system for the higher-order π-calculus. Keio University (September 1993)
23. Vasconcelos, V.T., Gay, S., Ravara, A.: Typechecking a multithreaded functional language with session types. TCS 368(1–2), 64–87 (2006)
24. Walker, D.: Advanced Topics in Types and Programming Languages. Pierce, B.C. (ed.) chapter Substructural Type Systems, MIT Press, Cambridge (2005)
25. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/
26. Yoshida, N.: Channel dependency types for higher-order mobile processes. In: POPL '04, pp. 147–160. ACM Press, New York (2004) Full version available at www.doc.ic.ac.uk/~yoshida
27. Yoshida, N., Hennessy, M.: Assigning types to processes. I & C 172, 82–120 (2002)
28. Yoshida, N., Vasconcelos, V.T.: Language Primitives and Type Disciplines for Structured Communication-based Programming Revisit. In: SecRet'06. ENTCS, Elsevier, Amsterdam (To appear)

An Isomorphism Between Cut-Elimination Procedure and Proof Reduction

Koji Nakazawa

Graduate School of Informatics, Kyoto University
knak@kuis.kyoto-u.ac.jp

Abstract. This paper introduces a cut-elimination procedure of the intuitionistic sequent calculus and shows that it is isomorphic to the proof reduction of the intuitionistic natural deduction with general elimination and explicit substitution. It also proves strong normalization and Church-Rosser property of the cut-elimination procedure by projecting the sequent calculus to the natural deduction with general elimination without explicit substitution.

1 Introduction

The Curry-Howard isomorphism between proof reduction and program computation is a useful tool to study logical systems and calculus systems. The correspondence has been investigated for the intuitionistic natural deduction and the λ-calculus, but that for the sequent calculus has not been studied enough and this research area is still developing.

To clarify the computational meaning of the sequent calculus, one of the most favorable approach is to study the relationship between the sequent calculus and the natural deduction, because the computational aspect of the natural deduction is relatively clear by the Curry-Howard isomorphism. Gentzen, who introduced the sequent calculus and the natural deduction, gave translations from proofs of each system to those of the other [5]. Prawitz gave a many-one mapping from proofs of the sequent calculus to those of the natural deduction [11]. Zucker studied on a correspondence between the cut-elimination in the sequent calculus and the proof reduction in the natural deduction [16]. Herbelin introduced a term reduction system for a variant of sequent calculus, called LJT [6], and gave a one-to-one correspondence between cut-free proofs in his sequent calculus and normal proofs in the natural deduction. He also showed his cut-elimination steps includes propagation steps of explicit substitution [1]. Herbelin style formulation of the sequent calculus has been widely studied [4,9]. For the original style sequent calculus, Urban and Bierman proposed a cut-elimination procedure for the classical sequent calculus [13,14], and proved its strong normalization. In particular, Urban [13] investigated a local-step cut-elimination procedure of Gentzen style sequent calculus, where "local-step" means that each cut-elimination step is a local transformation of proofs. He gave translations between sequent calculus and natural deduction in intuitionistic and classical cases, but neither of them

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 336–350, 2007.
© Springer-Verlag Berlin Heidelberg 2007

is an isomorphism. Kikuchi introduced a term assignment for the intuitionistic sequent calculus and its local step cut-elimination procedure [10]. He defined a subclass of proofs of the sequent calculus, called *pure terms*, which corresponds to proofs of the natural deduction. He also showed the cut-elimination can simulates the proof reduction of pure terms. However, strong normalization of the cut-elimination was not shown in [10]. Von Plato gave a correspondence between the sequent calculus and the natural deduction with general elimination rules [15]. However, the relationship between cut-elimination steps and proof reduction steps was not studied.

This paper gives a cut-elimination procedure for a Gentzen style intuitionistic sequent calculus LJ and an isomorphism between it and a proof normalization for the natural deduction with general elimination and explicit substitution. The computational meaning of the natural deduction with general elimination rules is relatively easy to understand. Indeed, we can find a reduction preserving continuation passing style (CPS) translation which gives an interpretation of our system in the well-known simply typed λ-calculus. This paper also proves strong normalization (SN) and Church-Rosser property (CR) of the cut-elimination of LJ. Though our system can be seen as a confluent subsystem of Urban's classical sequent calculus [13], SN of our system is proved by another method with a modified CPS-translation. In the chapter 7 of [12], we can find Dragalin's simple proof of SN of a sequent calculus. We cannot apply his proof to our sequent calculus due to the rule (π) corresponding to the permutative conversion.

First, we introduce a cut-elimination procedure of LJ and systems $\mathsf{LJ_p}$ and Λ_g. $\mathsf{LJ_p}$ is a subsystem of LJ which includes only a particular type of cuts, called *principal cuts* (*p-cuts*). $\mathsf{LJ_p}$ is not closed under the cut-elimination procedure of LJ, so we define *cut-elimination strategies* for cut-elimination of p-cuts. Λ_g is a simply typed λ-calculus with general elimination and permutative conversion. By the Curry-Howard isomorphism, Λ_g corresponds to the natural deduction with the general elimination rules. We show that $\mathsf{LJ_p}$ and Λ_g are isomorphic as reduction systems, where p-cuts and left-rules correspond to general eliminations. In particular, this isomorphism also gives a one-to-one correspondence between cut-free LJ-proofs and normal Λ_g-proofs. Secondly, we show SN and CR of LJ. These are proved by reducing to those of Λ_g. Joachimski and Matthes [8] proved SN of Λ_g by an inductive characterization of the set of SN terms. In this paper, we give another proof by Ikeda and Nakazawa's CGPS-translation method [7]. Then, SN of LJ is proved by Bloo's method [2]. Finally, we define Λ_{gx}, which is Λ_g with explicit substitution, and show that LJ is isomorphic to Λ_{gx} modulo a term quotient and that Λ_{gx} enjoys SN and CR. The figure 1 summarizes the relationship of systems in this paper, where Λ is the simply typed λ-calculus, Λ^p_{gx} is the modified Λ_{gx}, and horizontal arrows represent projection maps.

2 Definitions of Systems

In this section, we define the intuitionistic sequent calculus with a cut-elimination procedure and the simply typed λ-calculus with the general elimination rules.

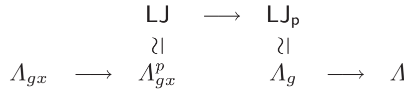


Fig. 1. Relationship of systems

2.1 LJ: Intuitionistic Sequent Calculus

Definition 1 (LJ). LJ *consists of the following.*

1. *We suppose that there are countable* atomic formulas. Formulas *(denoted by* A*,* $B\ldots$*) are defined as*

$$A ::= p \mid A \to A,$$

where p *denotes an atomic formula.*

2. Term variables *are denoted by* x*,* y*,* z *....* Pseudo-terms *(denoted by* M*,* N*,* $\mathsf{P}\ldots$*) are defined as*

$$\mathsf{M} ::= x \mid \underline{\lambda} x.\mathsf{M} \mid x[\![\mathsf{M}, x.\mathsf{M}]\!] \mid \mathsf{M} \curlyvee^{x} \mathsf{M}.$$

In terms $\underline{\lambda} x.\mathsf{M}$*,* $y[\![\mathsf{N}, x.\mathsf{M}]\!]$ *and* $\mathsf{N} \curlyvee^{x} \mathsf{M}$*, variable occurrences of* x *in* M *are bound. Renaming bound variables is admitted as usual. We write* $\underline{x}[\![\mathsf{M}, y.\mathsf{P}]\!]$ *for* $x[\![\mathsf{M}, y.\mathsf{P}]\!]$ *if the variable* x *does not freely occur in* $[\![\mathsf{M}, y.\mathsf{P}]\!]$ *following [9].*

3. Contexts *(denoted by* Γ*,* $\Delta\ldots$*) are sets of formulas labeled by term variables, such as* $\Gamma = \{A_1^{x_1}, A_2^{x_2} \cdots\}$*. We also write* A^x *for the singleton* $\{A^x\}$*.* Judgments *have the form of* $\Gamma \vdash \mathsf{M} : A$*.* Proofs *are defined by the inference rules in the figure 2, where* $\cup$ *and* $\setminus$ *are usual set-theoretical operators, union and difference. Each context in derivation trees has to meet the following condition: for any two distinct elements* A^x*,* B^y *in the context,* x *and* y *are distinct variables. A pseudo-term* M *is an* LJ-term *iff* $\Gamma \vdash \mathsf{M} : A$ *is derivable for some* Γ *and* A*.*

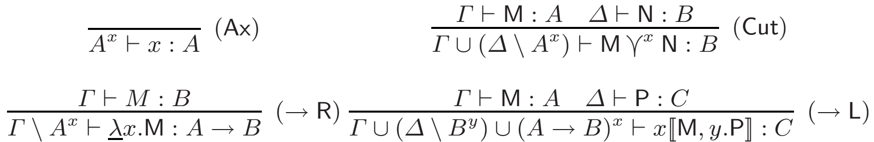


Fig. 2. Inference rules of LJ

Note 1. In LJ, structural rules are admitted implicitly. In fact, neither Γ in $(\to \mathsf{R})$ nor Δ in (Cut) have to contain A^x, so the weakening is admissible. In $(\to \mathsf{L})$, Γ and Δ may contain common elements, and Γ or Δ may contain $(A \to B)^x$, so the contraction is also admissible.

Definition 2 (Principal cuts). *A cut* $\mathsf{M} \curlyvee^x \mathsf{N}$ *is a* principal cut *(or* p-cut*) iff* N *has the form of* $\underline{x}\llbracket \mathsf{N}_1, y.\mathsf{N}_2 \rrbracket$ *and* M *is not a variable. We write* $\mathsf{M}\llbracket \mathsf{N}_1, y.\mathsf{N}_2 \rrbracket$ *for the p-cut* $\mathsf{M} \curlyvee^x \underline{x}\llbracket \mathsf{N}_1, y.\mathsf{N}_2 \rrbracket$*. If a cut is not principal, it is a* non-principal cut *(or* n-cut*). We write* $\mathsf{M} \curlyvee^x_n \mathsf{N}$ *if the cut is an n-cut.* $\mathsf{LJ_p}$ *denotes the set of terms of* LJ *whose subterms of the form* $\mathsf{M} \curlyvee^x \mathsf{N}$ *are all p-cuts.*

Note 2. An expression $\mathsf{M}\llbracket \mathsf{N}, x.\mathsf{P} \rrbracket$ denotes either a left-rule application or a p-cut depending on M is a variable or not. This notation is not ambiguous because $\mathsf{M} \curlyvee^x \underline{x}\llbracket \mathsf{N}, y.\mathsf{P} \rrbracket$ is a p-cut iff M is not a variable.

Definition 3 (Cut-elimination procedure). *Rules of the* cut-elimination procedure *of* LJ *are in the figure 3. We suppose that* y *is not free in* P *of* (β)*, and* z *is not free in* $\underline{x}\llbracket \mathsf{N}', z'.\mathsf{P}' \rrbracket$ *of* (π) *by renaming bound variables. Note that the cut* $\mathsf{P} \curlyvee^x \underline{x}\llbracket \mathsf{N}', z'.\mathsf{P}' \rrbracket$ *in the right-hand side of* (π) *is not a p-cut if* P *is a variable.* $\Rightarrow_\beta$ *denotes the one-step cut-elimination defined as the congruence relation including the rule* (β)*.* $\Rightarrow^+_\beta$ *and* $\Rightarrow^*_\beta$ *denotes the transitive closure and the*

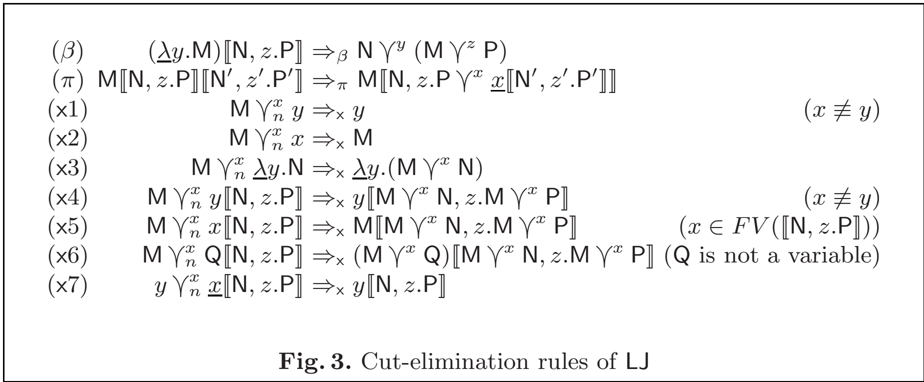


Fig. 3. Cut-elimination rules of LJ

reflexive transitive closure of $\Rightarrow_\beta$ *respectively. For* π*- and* x*-rules, relations are similarly defined.* $\Rightarrow$ *denotes the union of* $\Rightarrow_\beta$*,* $\Rightarrow_\pi$ *and* $\Rightarrow_{\mathsf{x}}$*.* $\Rightarrow^+$ *and* $\Rightarrow^*$ *are similarly defined. An* LJ*-term* M *is* $\Rightarrow$-normal *iff there is no term* N *such that* $\mathsf{M} \Rightarrow \mathsf{N}$.

Each p-cut is a redex of either (β) or (π), which corresponds to a redex of β-reduction or permutative conversion in the natural deduction. On the other hand, each n-cut is a redex of some x-rule, which corresponds to the propagation of explicit substitutions, if we understand an n-cut $\mathsf{M} \curlyvee^x \mathsf{P}$ as an explicit substitution $\langle \mathsf{M}/x \rangle \mathsf{P}$. Note that $\mathsf{LJ_p}$ is not closed under (β) and (π) because a result of them may contain n-cuts.

Note 3. This cut-elimination can be seen as a refinement of LJ^t [3,4]. The cut-elimination corresponding to x-reductions of LJ is treated as a meta-operation of substitution in LJ^t. On the other hand, the cut-elimination in this paper consists of local transformations of proofs. For example, (β) and (π) are the following:

$$\dfrac{\dfrac{\overset{\vdots\ \mathsf{M}}{A^y \vdash B}}{\vdash A \to B}\,(\to \mathsf{R}) \qquad \dfrac{\overset{\vdots\ \mathsf{N}}{\vdash A} \qquad \overset{\vdots\ \mathsf{P}}{B^z \vdash C}}{(A \to B)^x \vdash C}\,(\to \mathsf{L})}{\vdash C}\,(\mathsf{Cut}) \quad \Rightarrow_\beta \quad \dfrac{\overset{\vdots\ \mathsf{N}}{\vdash A} \qquad \dfrac{\overset{\vdots\ \mathsf{M}}{A^y \vdash B} \qquad \overset{\vdots\ \mathsf{P}}{B^z \vdash C}}{A^y \vdash C}\,(\mathsf{Cut})}{\vdash C}\,(\mathsf{Cut}) \quad ,$$

$$\dfrac{\dfrac{\overset{\vdots\ \mathsf{M}}{\vdash A_1 \to A_2} \qquad \dfrac{\overset{\vdots\ \mathsf{N}}{\vdash A_1} \qquad \overset{\vdots\ \mathsf{P}}{A_2^z \vdash B_1 \to B_2}}{(A_1 \to A_2)^y \vdash B_1 \to B_2}\,(\to \mathsf{L})}{\vdash B_1 \to B_2}\,(\mathsf{Cut}) \qquad \overset{\vdots\ \underline{x}[\![\mathsf{N}', z'.\mathsf{P}']\!]}{(B_1 \to B_2)^x \vdash C}}{\vdash C}\,(\mathsf{Cut})$$

$$\Rightarrow_\pi \quad \dfrac{\overset{\vdots\ \mathsf{M}}{\vdash A_1 \to A_2} \qquad \dfrac{\overset{\vdots\ \mathsf{N}}{\vdash A_1} \qquad \dfrac{\overset{\vdots\ \mathsf{P}}{A_2^z \vdash B_1 \to B_2} \qquad \overset{\vdots\ \underline{x}[\![\mathsf{N}', z'.\mathsf{P}']\!]}{(B_1 \to B_2)^x \vdash C}}{A_2^z \vdash C}\,(\mathsf{Cut})}{(A_1 \to A_2)^y \vdash C}\,(\to \mathsf{L})}{\vdash C}\,(\mathsf{Cut}) \quad .$$

We can divide the π-step into two steps such as

$$\begin{aligned} \mathsf{M}[\![\mathsf{N}, z.\mathsf{P}]\!][\![\mathsf{N}', z'.\mathsf{P}']\!] &\overset{(\star)}{\Rightarrow} \mathsf{M} \curlyvee^y (\underline{y}[\![\mathsf{N}, z.\mathsf{P}]\!][\![\mathsf{N}', z.\mathsf{P}']\!]) \\ &\Rightarrow \mathsf{M}[\![\mathsf{N}, z.\mathsf{P} \curlyvee^x \underline{x}[\![\mathsf{N}', z'.\mathsf{P}']\!]]\!], \end{aligned}$$

by a provisional cut-elimination step $(\star)$. But the middle term is reduced to the left-hand side by the x-rules, so if we admit $(\star)$, the cut-elimination is not SN.

Note 4. Our set of rules is similar to rules introduced by Kikuchi [10]. The rules (x1) through (x5), (x7) and (β) are the same as (1) through (6) and (*Beta*) of [10] respectively. ($Perm_1$) of [10] corresponds to $(\star)$. To avoid the loop noted above, in the rule ($Perm_2$) of [10], which is corresponding to (x6), Q is restricted to $\underline{\lambda}$-abstractions. SN is, however, not proved in [10]. If we choose the rules (π) and (x6), SN can be proved as shown in the following. Moreover our system does not contain the rule (7) of [10], since there is not its counterpart in the natural deduction.

Proposition 1 (Subject reduction). *If* $\Gamma \vdash \mathsf{M} : A$ *and* $\mathsf{M} \Rightarrow \mathsf{N}$ *hold, there exists* Γ' *such that* $\Gamma' \subseteq \Gamma$ *and* $\Gamma' \vdash \mathsf{M} : A$ *hold.*

Proof. By induction on $\mathsf{M} \Rightarrow \mathsf{N}$.

2.2 Λ_g: λ-Calculus with General Elimination Rules

In this subsection, we define the simply typed λ-calculus Λ and its variant Λ_g with *general elimination rules*, and show that Λ_g is a generalization of Λ.

Definition 4 (Λ). Formulas, term variables *and* contexts *of Λ are the same as* LJ. Pseudo-terms *(denoted by M, N, P...) are defined as*

$$M ::= x \mid \lambda x.M \mid MM.$$

Bound variables and capture-avoiding substitution $[M/x]N$ are defined as usual. Inference rules and reduction rules are in the figure 4. Λ-terms *and relations $\rightarrow$ and $\rightarrow^+$ are defined similarly to* LJ.

$$\frac{}{A^x \vdash x : A}\ (\mathsf{Ax})$$

$$\frac{\Gamma \vdash M : B}{\Gamma \setminus A^x \vdash \lambda x.M : A \rightarrow B}\ (\rightarrow \mathsf{I}) \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Delta \vdash N : A}{\Gamma \cup \Delta \vdash MN : B}\ (\rightarrow \mathsf{E})$$

$$(\beta)\ (\lambda x.M)N \rightarrow_\beta [N/x]M$$

Fig. 4. Inference rules and reduction rules of Λ

Definition 5 (Λ_g). Formulas, term variables *and* contexts *of Λ_g are the same as those of* LJ. Pseudo-terms *(denoted by M, N, P...) are defined as*

$$M ::= x \mid \lambda x.M \mid M[M, x.M].$$

In $\lambda x.P$ and $M[N, x.P]$, variable occurrences of x in P are bound. Capture-avoiding substitution $[M/x]N$ is defined as usual. Inference rules and reduction rules are in the figure 5. We suppose that x is not free in P of (β) and that x is not free in the subexpression $[N', x'.P']$ of (π) by renaming bound variables. The π-reduction is called permutative conversion. Λ_g-terms *and relations $\rightarrow$, $\rightarrow_\beta$, $\rightarrow^+$ and so on are defined similarly to* LJ.

$$\frac{}{A^x \vdash x : A}\ (\mathsf{Ax})$$

$$\frac{\Gamma \vdash M : B}{\Gamma \setminus A^x \vdash \lambda x.M : A \rightarrow B}\ (\rightarrow \mathsf{I}) \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Delta_1 \vdash N : A \quad \Delta_2 \vdash P : C}{\Gamma \cup \Delta_1 \cup (\Delta_2 \setminus B^x) \vdash M[N, x.P] : C}\ (\rightarrow \mathsf{E})$$

$$(\beta)\qquad (\lambda x.M)[N, y.P] \rightarrow_\beta [N/x][M/y]P$$
$$(\pi)\ M[N, x.P][N', x'.P'] \rightarrow_\pi M[N, x.P[N', x'.P']]$$

Fig. 5. Inference rules and reduction rules of Λ_g

Definition 6. *We define two maps φ from Λ to Λ_g and ψ from Λ_g to Λ as*

$$\begin{aligned} \varphi(x) &\equiv x, & \psi(x) &\equiv x, \\ \varphi(\lambda x.M) &\equiv \lambda x.\varphi(M), & \psi(\lambda x.M) &\equiv \lambda x.\psi(M), \\ \varphi(MN) &\equiv \varphi(M)[\varphi(N), x.x], & \psi(M[N, x.P]) &\equiv [\psi(M)\psi(N)/x]\psi(P). \end{aligned}$$

Proposition 2. *1. For any Λ-term M, $\psi(\varphi(M)) \equiv M$.*
2. For any Λ-terms M and N, if $M \to N$, then we have $\varphi(M) \to_\beta \varphi(N)$.
3. For any Λ_g-terms M and N, if $M \to N$, then we have $\psi(M) \to^* \psi(N)$. In particular, if $M \to_\pi N$, then $\psi(M) \equiv \psi(N)$.

Proof. Straightforward.

Note 5. The image of φ is characterized as

$$M ::= \ x \mid \lambda x.M \mid M[M, x.x].$$

Their π-normal forms are

$$M ::= \ V \mid V\varepsilon, \quad \text{where } V ::= x \mid \lambda x.M \quad \text{and } \varepsilon ::= \ [M, x.x] \mid [M, x.\underline{x}\varepsilon],$$

which correspond to pure terms in [10]. It is shown in [10] that normal pure terms correspond to normal Λ-terms. The class of pure terms is not closed under substitution, so the definition of β-reduction on pure terms needs some technical meta operation in [10]. In our paper, instead, the image of φ is closed under β-reduction of Λ_g, so the correspondence between it and Λ is much simpler.

2.3 Isomorphism Between $\mathsf{LJ_p}$ and Λ_g as Sets of Terms

Definition 7. *We define two maps M^* from $\mathsf{LJ_p}$ to Λ_g and M_* from Λ_g to $\mathsf{LJ_p}$ as follows.*

$$\begin{aligned} x^* &\equiv x & x_* &\equiv x \\ (\underline{\lambda} x.\mathsf{M})^* &\equiv \lambda x.\mathsf{M}^* & (\lambda x.M)_* &\equiv \underline{\lambda} x.M_* \\ (\mathsf{M}[\![\mathsf{N}, x.\mathsf{P}]\!])^* &\equiv \mathsf{M}^*[\mathsf{N}^*, x.\mathsf{P}^*] & (M[N, x.P])_* &\equiv M_*[\![N_*, x.P_*]\!] \end{aligned}$$

These maps are almost the same as identity maps except the $\mathsf{LJ_p}$-term $\mathsf{M}[\![\mathsf{N}, x.\mathsf{P}]\!]$ is an abbreviation of the p-cut $\mathsf{M} \curlyvee^x \underline{x}[\![\mathsf{N}, x.\mathsf{P}]\!]$ if M is not a variable. In particular, between cut-free LJ-terms and $\beta\pi$-normal Λ_g-terms.

Proposition 3. *1. For any $\mathsf{LJ_p}$-term M, we have $(\mathsf{M}^*)_* \equiv \mathsf{M}$. For any Λ_g-term M, we have $(M_*)^* \equiv M$.*
2. If $\Gamma \vdash \mathsf{M} : A$ is derivable in $\mathsf{LJ_p}$, then $\Gamma \vdash \mathsf{M}^* : A$ is derivable in Λ_g. If $\Gamma \vdash M : A$ is derivable in Λ_g, then $\Gamma \vdash M_* : A$ is derivable in $\mathsf{LJ_p}$.
3. For any cut-free $\mathsf{LJ_p}$-term M, M^* is normal. For any normal Λ_g-term M, M_* is cut-free.

Proof. By induction on M and M. For 3, note that the $\beta\pi$-normal terms in Λ_g have the form of either x or $x[M, y.P]$.

3 Correspondence Between Cut-Elimination and Proof Reduction

As shown in the previous section, $\mathsf{LJ_p}$ and Λ_g are almost identical as sets of terms. In fact, this correspondence can be extended to the cut-elimination and the reduction of λ-calculus. $\mathsf{LJ_p}$ is not closed under the cut-elimination procedure, so we introduce *cut-elimination strategies* for $\mathsf{LJ_p}$.

3.1 Projection from LJ to $\mathsf{LJ_p}$

First we define a projection from LJ to $\mathsf{LJ_p}$. It is nothing but the normalization function with respect to the x-reduction.

Definition 8 (Projection)

1. Pseudo-substitution $\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{N}$ *for* $\mathsf{LJ_p}$*-terms* M *and* N *is defined as follows.*

$$\llparenthesis \mathsf{M}/x \rrparenthesis y \equiv y \quad (x \not\equiv y) \qquad \llparenthesis \mathsf{M}/x \rrparenthesis (\underline{\lambda} y.\mathsf{N}) \equiv \underline{\lambda} y.\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{N}$$
$$\llparenthesis \mathsf{M}/x \rrparenthesis x \equiv \mathsf{M} \qquad \llparenthesis \mathsf{M}/x \rrparenthesis (\mathsf{Q}[\![\mathsf{N}, y.\mathsf{P}]\!]) \equiv (\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{Q})[\![\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{N}, y.\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{P}]\!]$$

Note that the right hand side of the last equation is either a left-rule application or a p-cut depending on whether $\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{Q}$ *is a variable or not.*

2. Projection M^{x} *from* LJ *to* $\mathsf{LJ_p}$ *is defined as follows.*

$$x^{\mathsf{x}} \equiv x \qquad (\mathsf{M}[\![\mathsf{N}, x.\mathsf{P}]\!])^{\mathsf{x}} \equiv \mathsf{M}^{\mathsf{x}}[\![\mathsf{N}^{\mathsf{x}}, x.\mathsf{P}^{\mathsf{x}}]\!]$$
$$(\underline{\lambda} x.\mathsf{M})^{\mathsf{x}} \equiv \underline{\lambda} x.\mathsf{M}^{\mathsf{x}} \qquad (\mathsf{M} \curlyvee^x_n \mathsf{N})^{\mathsf{x}} \equiv \llparenthesis \mathsf{M}^{\mathsf{x}}/x \rrparenthesis \mathsf{N}^{\mathsf{x}}$$

Lemma 1. *1. For any* $\mathsf{LJ_p}$*-terms* M *and* N*, we have* $(\llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{N})^* \equiv [\mathsf{M}^*/x]\mathsf{N}^*$*.*
2. For any Λ_g*-terms* M *and* N*, we have* $([M/x]N)_* \equiv \llparenthesis M_*/x \rrparenthesis N_*$*.*

Proof. By induction on N and N respectively.

Lemma 2. *For any* LJ*-term* M*,* M^{x} *is a* $\mathsf{LJ_p}$*-term and we have* $\mathsf{M} \Rightarrow^*_{\mathsf{x}} \mathsf{M}^{\mathsf{x}}$*.*

Proof. For any $\mathsf{LJ_p}$-terms M and N, $\mathsf{M} \curlyvee^x \mathsf{N} \Rightarrow^*_{\mathsf{x}} \llparenthesis \mathsf{M}/x \rrparenthesis \mathsf{N}$ is proved by induction on N. And then, $\mathsf{M} \Rightarrow^*_{\mathsf{x}} \mathsf{M}^{\mathsf{x}}$ is proved by induction on M.

In the following, we show that each step of the cut-elimination is projected to the reduction steps of Λ_g. To describe the claim more precisely, we prepare an auxiliary notion, which was introduced in [2].

Definition 9. *A subterm occurrence* N *in* M *is* void *iff there is a subterm* $\mathsf{P} \curlyvee^x_n \mathsf{Q}$ *of* M *such that* N *occurs in* P *and* $x \notin FV(\mathsf{Q}^{\mathsf{x}})$*. A cut-elimination step* $\mathsf{M} \Rightarrow \mathsf{N}$ *is* void *iff the redex of* M *is void. We write* $\mathsf{M} \stackrel{\mathsf{v}}{\Rightarrow} \mathsf{N}$ *when the step is void.*

Proposition 4. *Let the symbol* $\bullet$ *be either* β *or* π*. For any* LJ*-terms* M *and* N*, if* $\mathsf{M} \Rightarrow_{\bullet} \mathsf{N}$ *holds and it is not void, then we have* $(\mathsf{M}^{\mathsf{x}})^* \rightarrow^+_{\bullet} (\mathsf{N}^{\mathsf{x}})^*$ *in* Λ_g*. In particular, when* M *is an* $\mathsf{LJ_p}$*-term, we have* $(\mathsf{M}^{\mathsf{x}})^* \rightarrow_{\bullet} (\mathsf{N}^{\mathsf{x}})^*$*. If either* $\mathsf{M} \Rightarrow_{\mathsf{x}} \mathsf{N}$ *or* $\mathsf{M} \stackrel{\mathsf{v}}{\Rightarrow} \mathsf{N}$ *holds, then we have* $\mathsf{M}^{\mathsf{x}} \equiv \mathsf{N}^{\mathsf{x}}$*.*

Proof. By induction on $\mathsf{M} \Rightarrow \mathsf{N}$. We prove only the case where $\mathsf{M} \equiv \mathsf{M}_0 \curlyvee^x \mathsf{M}_1$, $\mathsf{N} \equiv \mathsf{N}_0 \curlyvee^x \mathsf{M}_1$, $\mathsf{M}_0 \Rightarrow \mathsf{N}_0$ and $x \notin FV(\mathsf{M}_1{}^{\mathsf{x}})$, that is, $\mathsf{M} \overset{\mathsf{v}}{\Rightarrow} \mathsf{N}$. We have $\mathsf{M}^{\mathsf{x}} \equiv (\!|\mathsf{M}_0^{\mathsf{x}}/x|\!)\mathsf{M}_1^{\mathsf{x}}$ and $\mathsf{N}^{\mathsf{x}} \equiv (\!|\mathsf{N}_0^{\mathsf{x}}/x|\!)\mathsf{M}_1^{\mathsf{x}}$, which are identical since $x \notin FV(\mathsf{M}_1^{\mathsf{x}})$. Other cases are easily proved by the Lemma 1.

Proposition 5. *$\Rightarrow_{\mathsf{x}}$ is* SN *and* CR*, so any* LJ*-term* M *has the unique $\Rightarrow_{\mathsf{x}}$-normal form, which is* M^{x}*.*

Proof. For SN, we define $|\mathsf{M}|$ and $\#\mathsf{M}$ as

$$
\begin{aligned}
|x| &= 1, & \#x &= 1,\\
|\underline{\lambda} x.\mathsf{M}| &= |\mathsf{M}|, & \#(\underline{\lambda} x.\mathsf{M}) &= \#\mathsf{M} + 1,\\
|x[\![\mathsf{M}, y.\mathsf{N}]\!]| &= |\mathsf{M}| + |\mathsf{N}|, & \#(x[\![\mathsf{M}, y.\mathsf{N}]\!]) &= \#\mathsf{M} + \#\mathsf{N} + 2,\\
|\mathsf{P}[\![\mathsf{M}, y.\mathsf{N}]\!]| &= |\mathsf{P}| + |\mathsf{M}| + |\mathsf{N}|, & \#(\mathsf{P}[\![\mathsf{M}, y.\mathsf{N}]\!]) &= \#\mathsf{P} + \#\mathsf{M} + \#\mathsf{N} + 1,\\
|z \curlyvee^x_n \underline{x}[\![\mathsf{M}, y.\mathsf{N}]\!]| &= |\mathsf{M}| + |\mathsf{N}| + 1, & \#(\mathsf{M} \curlyvee^x_n \mathsf{N}) &= \#\mathsf{M} \cdot \#\mathsf{N},\\
|\mathsf{M} \curlyvee^x_n \mathsf{N}| &= |\mathsf{M}| \cdot \#\mathsf{N} + |\mathsf{N}| \text{ (o.w.)}, & &
\end{aligned}
$$

where P is not a variable. We can prove that $\mathsf{M} \Rightarrow_{\mathsf{x}} \mathsf{N}$ implies $|\mathsf{M}| > |\mathsf{N}|$ and $\#\mathsf{M} \geq \#\mathsf{N}$. For CR, suppose that $\mathsf{M} \Rightarrow_{\mathsf{x}}^* \mathsf{M}_1$ and $\mathsf{M} \Rightarrow_{\mathsf{x}}^* \mathsf{M}_2$ holds. By the Proposition 4, $\mathsf{M}^{\mathsf{x}} \equiv \mathsf{M}_1^{\mathsf{x}} \equiv \mathsf{M}_2^{\mathsf{x}}$ holds, so we have $\mathsf{M}_1 \Rightarrow_{\mathsf{x}}^* \mathsf{M}_1^{\mathsf{x}} \equiv \mathsf{M}^{\mathsf{x}}$ and $\mathsf{M}_2 \Rightarrow_{\mathsf{x}}^* \mathsf{M}_2^{\mathsf{x}} \equiv \mathsf{M}^{\mathsf{x}}$ by the Lemma 2.

3.2 Isomorphism Between $\mathsf{LJ_p}$ and Λ_g

We define *cut-elimination strategies* on $\mathsf{LJ_p}$corresponding the reductions in Λ_g.

Definition 10 ($\beta\pi$-strategy). *Relation* $\mathsf{M} \to_\beta \mathsf{N}$ *on* $\mathsf{LJ_p}$ *is defined as* $\mathsf{M} \Rightarrow_\beta \mathsf{M}'$ *and* $\mathsf{M}'^{\mathsf{x}} \equiv \mathsf{N}$ *for some* M'*. We call* $\to_\beta$ β-strategy*. Similarly* π-strategy $\mathsf{M} \to_\pi \mathsf{N}$ *is defined as* $\mathsf{M} \Rightarrow_\pi \mathsf{M}'$ *and* $\mathsf{M}'^{\mathsf{x}} \equiv \mathsf{N}$ *for some* M'*.*

Lemma 3. *Let the symbol $\bullet$ be either β or π. For any* $\mathsf{LJ_p}$*-terms* M *and* N*,* $\mathsf{M} \to_\bullet \mathsf{N}$ *implies* $\mathsf{M} \Rightarrow_{\bullet n}^+ \mathsf{N}$ *in* LJ*.*

Proof. By the definition of the $\beta\pi$-strategy and the Lemma 2.

Theorem 1. *Let the symbol $\bullet$ be either β or π.*

1. *For any* $\mathsf{LJ_p}$*-terms* M *and* N*, if* $\mathsf{M} \to_\bullet \mathsf{N}$ *holds, then* $\mathsf{M}^* \to_\bullet \mathsf{N}^*$ *holds in* Λ_g*.*
2. *For any Λ_g-terms M and N, if $M \to_\bullet N$ holds, then $M_* \to_\bullet N_*$ holds in* $\mathsf{LJ_p}$*.*

Proof. 1. Suppose $\mathsf{M} \to_\bullet \mathsf{N}$. By the definition of the strategy, there exists an LJ-term M' such that $\mathsf{M} \Rightarrow_\bullet \mathsf{M}' \Rightarrow_{\mathsf{x}}^* \mathsf{N}$. By the Proposition 4, we have $(\mathsf{M}^{\mathsf{x}})^* \to_\bullet (\mathsf{N}^{\mathsf{x}})^*$, that is, $\mathsf{M}^* \to_\bullet \mathsf{N}^*$ since $\mathsf{M}^{\mathsf{x}} \equiv \mathsf{M}$ for any $\mathsf{LJ_p}$-term M.
2. By induction on $M \to N$.

Corollary 1. *Let the symbol $\bullet$ be either β or π. For any* LJ*-terms* M *and* N*, if* $\mathsf{M} \Rightarrow_\bullet \mathsf{N}$ *holds and it is not void, then we have* $\mathsf{M}^{\mathsf{x}} \to_\bullet^+ \mathsf{N}^{\mathsf{x}}$ *in* $\mathsf{LJ_p}$*.*

Corollary 2.
1. *For any* LJ*-terms* M *and* N*, if* $\mathsf{M} \Rightarrow_\beta \mathsf{N}$ *holds, then* $\psi((\mathsf{M}^{\mathsf{x}})^*) \to^* \psi((\mathsf{N}^{\mathsf{x}})^*)$ *holds in* Λ*. If* $\mathsf{M} \Rightarrow_{\pi\mathsf{x}} \mathsf{N}$ *holds, then* $\psi((\mathsf{M}^{\mathsf{x}})^*) \equiv \psi((\mathsf{N}^{\mathsf{x}})^*)$ *holds.*
2. *For any Λ-terms M and N, if $M \to N$ holds, then $(\varphi(M))_* \Rightarrow_{\beta\mathsf{x}}^+ (\varphi(M))_*$ holds in* LJ*.*

4 Strong Normalization and Church-Rosser Property

In this section, we prove SN and CR of the cut-elimination procedure for LJ by those of the $\beta\pi$-reduction in Λ_g.

4.1 SN and CR of Λ_g

First, we prove SN and CR of the $\beta\pi$-reduction on Λ_g by a *continuation and garbage passing style (CGPS) translation*, which is a variant of continuation passing style translations and was introduced by Ikeda and Nakazawa [7].

The CGPS-translation maps Λ_g-terms to Λ-terms, preserving typability and one-or-more step reduction relation. In the following definition, metavariables K and G for Λ-terms denote continuation terms and garbage terms respectively. We introduce garbage parts to map each π-reduction step in Λ_g into dummy β-reduction steps in Λ. Note that, if we ignore the garbage parts denoted by g and G, we can get a CPS-translation of Λ_g.

Definition 11 (CGPS-translation). *Let $\bot$ be a fixed atomic formula, $\neg A \equiv A \to \bot$ for a formula A, and $\langle\!\langle M; N \rangle\!\rangle \equiv (\lambda x.M)N$ for Λ-terms M and N, where x is a fresh variable.* Negative translation $\overline{A}$ *of a formula A is defined as $\neg\bot \to \neg\neg A^\dagger$, where $A^\dagger$ is defined as $p^\dagger \equiv p$ and $(A \to B)^\dagger \equiv \overline{A} \to \overline{B}$.* CGPS-translation $\overline{M}$ *of a Λ_g-term M is defined as a Λ-term $\lambda gk.(M :_g k)$, where g and k are fresh and $M :_G K$ for a Λ_g-term M and Λ-terms G, K is defined as*

$$
\begin{aligned}
x :_G K &\equiv xGK, \\
\lambda x.M :_G K &\equiv \langle\!\langle K(\lambda x.\overline{M}); G \rangle\!\rangle, \\
M[N, x.P] :_G K &\equiv M :_{\langle\!\langle G; K' \rangle\!\rangle} K' \qquad (K' \equiv \lambda y.(\lambda x.(P :_G K))(y\overline{N})).
\end{aligned}
$$

Lemma 4. *1. If $\Gamma \vdash M : A$ is derivable in Λ_g, we have $\overline{\Gamma} \vdash \overline{M} : \overline{A}$ in Λ, where $\overline{\Gamma}$ is defined as $\{\overline{A}^x \mid A^x \in \Gamma\}$.*
2. If $M \to N$ holds in Λ_g, we have $\overline{M} \to^+ \overline{N}$ in Λ.

Proof. 1. By induction on M. We need to prove it simultaneously with another claim: $\Gamma \vdash M : A$ in Λ_g implies $\overline{\Gamma}, (\neg A^\dagger)^k, (\neg\bot)^g \vdash (M :_g k) : \bot$ in Λ.
2. By induction on $M \to N$. We use the fact that we have $M :_G N \to^+ M :_G N'$ and $M :_N K \to^+ M :_{N'} K$ for any $N \to N'$, and we have $[\overline{M}/x](P :_G K) \to^* ([M/x]P) :_{[\overline{M}/x]G} [\overline{M}/x]K$.

Proposition 6 (SN of Λ_g). *For any Λ_g-term M, there is no infinite $\beta\pi$-reduction sequence from M.*

Proof. Suppose that $M_0 \to M_1 \to \cdots$ is an infinite reduction sequence from an Λ_g-term M_0. By the Lemma 4, $\overline{M}_0 \to^+ \overline{M}_1 \to^+ \cdots$ is an infinite sequence from the Λ-term M_0, which contradicts the SN of Λ.

Proposition 7 (CR of Λ_g). *For any Λ_g-terms M, M_1 and M_2, if $M \to^* M_1$ and $M \to^* M_2$ hold, then there exists a Λ_g-term M_3 such that $M_1 \to^* M_3$ and $M_2 \to^* M_1$.*

Proof. It is sufficient to prove WCR: if we have $M \to M_1$ and $M \to M_2$, then there exists M_3 such that $M_1 \to^* M_3$ and $M_2 \to^* M_3$ hold. Then, CR is derived from SN and WCR by Newman's lemma. We prove WCR by induction on M. For the case where M has the form of $Q[N, x.P]$, there are three types of forms of $Q[N, x.P] \to M_i$: (1) β-redex $(\lambda y.Q_0)[N, x.P] \to [N/y][Q_0/x]P$, (2) π-redex $Q'[N', x'.P'][N, x.P] \to Q'[N', x'.P'[N, x.P]]$, (3) $Q[N, x.P] \to Q'[N', x.P']$, where either $Q \to Q'$, $N \to N'$ or $P \to P'$. The case where both $M \to M_i$ are the same type of either (1) or (2), we have $M_1 \equiv M_2$. Other cases are easily proved by the induction hypothesis. Note that there is no case where one of $M \to M_i$ is the type (1) and the other is the type (2).

Corollary 3. $\mathsf{LJ_p}$ *enjoys* SN *and* CR *with respect to the* $\beta\pi$*-strategy.*

4.2 SN and CR of LJ

Theorem 2. *The cut-elimination procedure of* LJ *enjoys* CR*.*

Proof. Suppose $\mathsf{M} \Rightarrow^* \mathsf{M}_1$ and $\mathsf{M} \Rightarrow^* \mathsf{M}_2$ hold in LJ. By the Corollary 1, we have $\mathsf{M}^{\mathsf{x}} \to^* \mathsf{M}_i^{\mathsf{x}}$ for $i = 1$ and 2. By CR of $\mathsf{LJ_p}$, there is an $\mathsf{LJ_p}$-term M_3 such that $\mathsf{M}_i^{\mathsf{x}} \to^* \mathsf{M}_3$. And then we have $\mathsf{M}_i \Rightarrow_{\mathsf{x}}^* \mathsf{M}_i^{\mathsf{x}} \Rightarrow^* \mathsf{M}_3$ by the Lemma 2 and 3.

SN of LJ is proved by the method which has been applied to SN proofs for calculi with explicit substitution in [2,9] and so on.

Definition 12. *An* LJ*-term* M *is* decent *iff for any subterm* $\mathsf{N} \curlyvee_n^x \mathsf{P}$ *of* M*,* N *is* SN. Rank $\rho(\mathsf{M})$ *of an* LJ*-term* M *is defined as the maximum length of* $\beta\pi$*-strategy sequence from* M^{x}*.*

Lemma 5. *1. In any infinite sequence of* $\Rightarrow_{\mathsf{x}}$ *and* $\overset{\mathsf{v}}{\Rightarrow}$*, all reduction steps except for finitely many steps are void.*
2. In any infinite sequence of $\Rightarrow$*, all reduction steps except for finitely many steps are void.*
3. Any decent term M *is* SN *with respect to* $\overset{\mathsf{v}}{\Rightarrow}$*.*

Proof. 1. We define the norm $\|\mathsf{M}\|$ and $\flat\mathsf{M}$ for a LJ-term M as follows, where P is a variable.

$$\begin{aligned}
\|x\| &= 1 & \flat x &= 1\\
\|\underline{\lambda} x.\mathsf{M}\| &= \|\mathsf{M}\| & \flat(\underline{\lambda} x.\mathsf{M}) &= \flat\mathsf{M} + 1\\
\|x[\![\mathsf{M}, y.\mathsf{N}]\!]\| &= \|\mathsf{M}\| + \|\mathsf{N}\| & \flat(x[\![\mathsf{M}, y.\mathsf{N}]\!]) &= \flat\mathsf{M} + \flat\mathsf{N} + 2\\
\|\mathsf{P}[\![\mathsf{M}, y.\mathsf{N}]\!]\| &= \|\mathsf{P}\| + \|\mathsf{M}\| + \|\mathsf{N}\| & \flat(\mathsf{P}[\![\mathsf{M}, y.\mathsf{N}]\!]) &= \flat\mathsf{P} + \flat\mathsf{M} + \flat\mathsf{N} + 1\\
\|z \curlyvee_n^x \underline{x}[\![\mathsf{M}, y.\mathsf{N}]\!]\| &= \|\mathsf{M}\| + \|\mathsf{N}\| + 1 & \flat(\mathsf{M} \curlyvee_n^x \mathsf{N}) &= \flat\mathsf{M} \cdot \flat\mathsf{N} \quad (x \in FV(\mathsf{N}^{\mathsf{x}}))\\
\|\mathsf{M} \curlyvee_n^x \mathsf{N}\| &= \|\mathsf{M}\| \cdot \flat\mathsf{N} + \|\mathsf{N}\| \quad (x \in FV(\mathsf{N}^{\mathsf{x}})) & \flat(\mathsf{M} \curlyvee_n^x \mathsf{N}) &= \flat\mathsf{N} \quad (x \notin FV(\mathsf{N}^{\mathsf{x}}))\\
\|\mathsf{M} \curlyvee_n^x \mathsf{N}\| &= \flat\mathsf{N} + \|\mathsf{N}\| \quad (x \notin FV(\mathsf{N}^{\mathsf{x}}))
\end{aligned}$$

We can prove that if $\mathsf{M} \Rightarrow_{\mathsf{x}} \mathsf{N}$ holds and it is not void, then we have $\|\mathsf{M}\| > \|\mathsf{N}\|$ and $\flat\mathsf{M} \geq \flat\mathsf{N}$, and if $\mathsf{M} \overset{\mathsf{v}}{\Rightarrow} \mathsf{N}$ holds, then we have $\|\mathsf{M}\| = \|\mathsf{N}\|$.

2. Suppose that $\mathsf{M}_0 \Rightarrow \mathsf{M}_1 \Rightarrow \mathsf{M}_2 \Rightarrow \cdots$ is an infinite sequence. By the Corollary 1, we have $\mathsf{M}_0^{\mathsf{x}} \to^* \mathsf{M}_1^{\mathsf{x}} \to^* \mathsf{M}_2^{\mathsf{x}} \to^* \cdots$, where there is an index m such that $\mathsf{M}_i^{\mathsf{x}} \equiv \mathsf{M}_{i+1}^{\mathsf{x}}$ for any $i > m$ by SN of $\mathsf{LJ_p}$. So $\mathsf{M}_i \Rightarrow \mathsf{M}_{i+1}$ is $\Rightarrow_{\mathsf{x}}$ or $\overset{\mathsf{v}}{\Rightarrow}$ for any $i > m$. By 1, the sequence contains only finitely many non-void $\Rightarrow_{\mathsf{x}}$-steps.
3. By induction on M.

Lemma 6. *If* M *is a decent* LJ*-term,* M *is* SN *with respect to* $\Rightarrow$*.*

Proof. By induction on $\rho(\mathsf{M})$. Suppose that, for any N such that $\rho(\mathsf{M}) > \rho(\mathsf{N})$, if N is decent, then N is SN. First, we show that any N such that $\mathsf{M} \Rightarrow \mathsf{N}$ is decent by induction on $\mathsf{M} \Rightarrow \mathsf{N}$. For the case where $\mathsf{M} \equiv (\underline{\lambda} x.\mathsf{M}_1)[\![\mathsf{M}_2, y.\mathsf{M}_3]\!]$ and $\mathsf{N} \equiv \mathsf{M}_2 \curlyvee^x (\mathsf{M}_1 \curlyvee^y \mathsf{M}_3)$, since $\mathsf{M}_1^{\mathsf{x}}$ and $\mathsf{M}_2^{\mathsf{x}}$ are proper subterms of a β-redex $\mathsf{M}^{\mathsf{x}} \equiv (\underline{\lambda} x.\mathsf{M}_1^{\mathsf{x}})[\![\mathsf{M}_2^{\mathsf{x}}, y.\mathsf{M}_3^{\mathsf{x}}]\!]$, we have $\rho(\mathsf{M}) > \rho(\mathsf{M}_1)$ and $\rho(\mathsf{M}) > \rho(\mathsf{M}_2)$. Moreover, M_1 and M_2 are decent since M is decent, so they are SN by the hypothesis of the outer induction. Therefore, N is decent. The cases of π and x-redexes are similarly proved. Other cases are proved by the hypothesis for the inner induction. Secondly, suppose that there is an infinite sequence $\mathsf{M} \Rightarrow \mathsf{M}_1 \Rightarrow \mathsf{M}_2 \Rightarrow \cdots$. By the fact proved above, any M_i is decent. Furthermore, by 2 of the Lemma 5, there is an index m such that $\mathsf{M}_i \overset{\mathsf{v}}{\Rightarrow} \mathsf{M}_{i+1}$ holds for any $i > m$, which contradicts 3 of the Lemma 5.

Theorem 3. *The cut-elimination procedure of* LJ *enjoys* SN*.*

Proof. By induction on LJ-terms. For any M, any proper subterm of M is SN by the induction hypothesis, so M is decent. Therefore, M is SN by the Lemma 6.

5 Sequent Calculus and Explicit Substitutions

In this section, we define another system Λ_{gx}, which is a λ-calculus with general elimination rules and explicit substitutions. We show that LJ is isomorphic to Λ_{gx} modulo a term quotient.

Definition 13 (Λ_{gx}). Λ_{gx} *is defined as an extension of* Λ_g. Pseudo-terms *are extended by* explicit substitutions $\langle M/x\rangle N$. *The only additional inference rule* (X) *and reduction rules of* Λ_{gx} *are in the figure 6.*

Λ_{gx} is almost the same as LJ under an identification of $\mathsf{M} \curlyvee_n^x \mathsf{N}$ and $\langle M/x\rangle N$. The only difference is that, if M is not a variable, $\langle M/x\rangle(\underline{x}[N, y.P])$ and $M[N, y.P]$ are distinct Λ_{gx}-terms, while the corresponding LJ-terms are identical, since the cut $\mathsf{M} \curlyvee^x \underline{x}[\![\mathsf{N}, y.\mathsf{P}]\!]$ is a p-cut.

Lemma 7. $\Rrightarrow_p$ *is* CR *and* SN*.*

Proof. Each $\Rrightarrow_p$-step decreases size of terms, so SN holds. WCR is easily proved.

Definition 14. *For* Λ_{gx}*-term* M*,* M^p *denotes its* $\Rrightarrow_p$*-normal form.* Λ_{gx}^p *consists of the following. Terms are* $\Rrightarrow_p$*-normal* Λ_{gx}*-terms. For* Λ_{gx}^p*-terms* M *and* N*,* β-strategy $M \Rightarrow_\beta N$ *holds iff there exists an* Λ_{gx}*-term* M' *such that* $M \Rrightarrow_\beta M'$

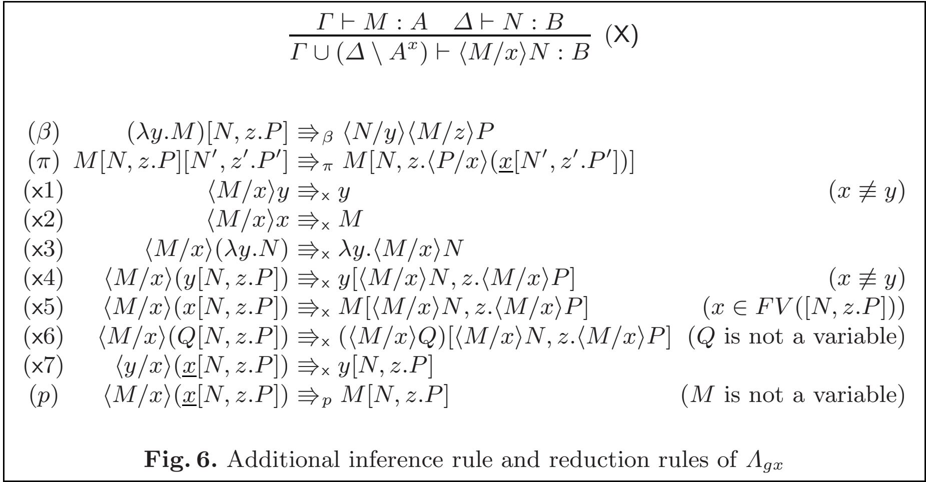


Fig. 6. Additional inference rule and reduction rules of Λ_{gx}

and $M'^{p} \equiv N$ *hold.* π- *and* x-strategies *are similarly defined. Furthermore, We extend* M^* *and* M_* *to maps between* LJ *and* Λ^p_{gx} *as follows:*

$$(\mathsf{M} \curlyvee^x_n \mathsf{N})^* \equiv \langle \mathsf{M}^*/x\rangle \mathsf{N}^*, \qquad (\langle M/x\rangle N)_* \equiv M_* \curlyvee^x N_*.$$

The following properties are proved in a straightforward way.

Proposition 8. *1. For any* LJ*-term* M*, we have* $(\mathsf{M}^*)_* \equiv \mathsf{M}$*. For any* Λ^p_{gx}*-term* M*, we have* $(M_*)^* \equiv M$*.*
2. If $\Gamma \vdash \mathsf{M} : A$ *is derivable in* LJ*, then* $\Gamma \vdash \mathsf{M}^* : A$ *is derivable in* Λ^p_{gx}*. If* $\Gamma \vdash M : A$ *is derivable in* Λ^p_{gx}*, then* $\Gamma \vdash M_* : A$ *is derivable in* LJ*.*

Lemma 8. *1. For any* LJ*-terms* M *and* N*, we have* $(\mathsf{M} \curlyvee^x \mathsf{N})^* \equiv (\langle \mathsf{M}^*/x\rangle \mathsf{N}^*)^p$*.*
2. For any Λ_{gx}*-terms* M *and* N*, we have* $((\langle M/x\rangle N)^p)_* \equiv (M^p)_* \curlyvee^x (N^p)_*$*.*

Lemma 9. *Let the symbol* $\bullet$ *be either* β*,* π *or* x*.*

1. *For any* Λ^p_{gx}*-terms* M *and* N*,* $M \Rrightarrow_\bullet N$ *implies* $M \Rrightarrow^*_{\bullet p} N$*.*
2. *For any* Λ_{gx}*-terms* M *and* N*,* $M \Rrightarrow_\bullet N$ *implies* $M^p \Rrightarrow_\bullet N^p$*, and* $M \Rrightarrow_p N$ *implies* $M^p \equiv N^p$*.*

Theorem 4. *Let* $\bullet$ *be either* β*,* π *or* x*.*

1. *For any* LJ*-terms* M *and* N*,* $\mathsf{M} \Rightarrow_\bullet \mathsf{N}$ *in* LJ *implies* $\mathsf{M}^* \Rrightarrow_\bullet \mathsf{N}^*$ *in* Λ^p_{gx}*.*
2. *For any* Λ^p_{gx}*-terms* M *and* N*,* $M \Rrightarrow_\bullet N$ *in* Λ^p_{gx} *implies* $\mathsf{M}_* \Rightarrow_\bullet \mathsf{N}_*$ *in* LJ*.*

Proof. 1. By induction on $\mathsf{M} \Rightarrow \mathsf{N}$. We prove only the case where $\mathsf{M} \equiv (\underline{\lambda} x.\mathsf{Q})[\![\mathsf{N}, z.\mathsf{P}]\!]$ and $\mathsf{N} \equiv \mathsf{N} \curlyvee^x (\mathsf{Q} \curlyvee^z \mathsf{P})$. We have $\mathsf{M}^* \equiv (\lambda x.\mathsf{Q}^*)[\mathsf{N}^*, z.\mathsf{P}^*] \Rrightarrow_\beta (\langle \mathsf{N}^*/x\rangle\langle \mathsf{Q}^*/z\rangle \mathsf{P}^*)^p$, which is identical to $(\mathsf{N} \curlyvee^x (\mathsf{Q} \curlyvee^z \mathsf{P}))^*$ by 1 of the Lemma 8.
2. It is similarly proved by induction on $M \Rightarrow N$, using 2 of the Lemma 8.

CR and SN of Λ^p_{gx} immediately follows those of LJ. Furthermore, CR and SN of Λ_{gx} can be easily proved by means of those of Λ^p_{gx}.

Theorem 5. *Λ_{gx} enjoys* CR *and* SN*.*

Proof. CR is proved in a similar way to the Theorem 2 by the Lemma 9. Furthermore, by 2 of the Lemma 9, the map $(\cdot)^p$ translates an infinite $\Rrightarrow$-sequence in Λ_{gx} to an infinite $\Rightarrow$-sequence in Λ^p_{gx}, since $\Rrightarrow_p$ is SN.

6 Concluding Remarks

This paper proposes an SN and CR cut-elimination procedure of the intuitionistic sequent calculus which is isomorphic to the proof reduction of the natural deduction with general elimination and explicit substitutions. The discussion in this paper can be extended to other logical connectives. For example, general elimination rules for conjunction and disjunction are

$$\frac{\Gamma \vdash M : A \wedge B \quad \Delta \vdash P : C}{\Gamma \cup (\Delta \setminus \{A^x, B^y\}) \vdash M[(x,y).P] : C}\ (\wedge\mathsf{E}), \quad \frac{\Gamma \vdash M : A \vee B \quad \Delta_1 \vdash P : C \quad \Delta_2 \vdash Q : C}{\Gamma \cup (\Delta_1 \setminus A^x) \cup (\Delta_2 \setminus B^y) \vdash M[x.P, y.Q] : C}\ (\vee\mathsf{E})$$

and other definitions and proofs can be extended in a straightforward way.

The sequent calculus is well-suited to classical logic because of its beautiful symmetry, so we hope that our result will be extended to classical logic. However, even for the intuitionistic case, the cut-elimination contains much richer computation than our proposal. In fact, our cut-elimination includes only *t-protocol* cut-elimination of LK^{tq} [3]. For example, we can consider the cut-elimination steps following the *q-protocol* such as

$$\frac{\dfrac{\vdash B_1 \quad B_2 \vdash A}{B_1 \to B_2 \vdash A}\ (\to\mathsf{L}) \qquad \dfrac{\vdots}{A \vdash C_1 \to C_2}\ (\to\mathsf{R})}{B_1 \to B_2 \vdash C_1 \to C_2}\ (\mathsf{Cut}) \quad \Rightarrow \quad \frac{\vdash B_1 \qquad \dfrac{B_2 \vdash A \qquad \dfrac{\vdots}{A \vdash C_1 \to C_2}\ (\to\mathsf{R})}{B_2 \vdash C_1 \to C_2}\ (\mathsf{Cut})}{B_1 \to B_2 \vdash C_1 \to C_2}\ (\to\mathsf{L}),$$

whose term representation is $x[\![\mathsf{M}, y.\mathsf{P}]\!] \curlyvee^z_n (\underline{\lambda} w.\mathsf{N}) \Rightarrow x[\![\mathsf{M}, y.\mathsf{P} \curlyvee^z_n (\underline{\lambda} w.\mathsf{N})]\!]$, which is not contained in our system. Moreover, we can consider another *orientation* (in [3]) of logical cut-elimination such as $(\underline{\lambda} x.\mathsf{M})[\![\mathsf{N}, y.\mathsf{P}]\!] \Rightarrow (\mathsf{N} \curlyvee^x \mathsf{M}) \curlyvee^y \mathsf{P}$. Adding this rule makes no trouble with the isomorphism between the cut-elimination and the proof reduction. However the SN proof of the cut-elimination in this paper does not work for the new β-rule, since SN of $\mathsf{N} \curlyvee^x \mathsf{M}$ in the contractum is not guaranteed by decency of the redex. Urban gave a strongly normalizable cut-elimination for the classical sequent calculus [13], which admits to permute cuts in both direction by the notion of *labelled cuts*. He also gave a correspondence between the sequent calculus and a classical natural deduction with multiple conclusion, but it is not an isomorphism. It is a future work to extend the isomorphism established in this paper to classical logic and clarify computational meaning of cut-elimination in the classical sequent calculus.

Acknowledgment

The author is grateful to Makoto Tatsuta and Kentaro Kikuchi for fruitful discussion and their valuable comments.

References

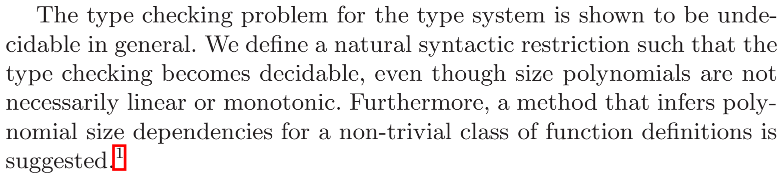
1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. Journal of Functional Programming 1, 375–416 (1991)
2. Bloo, R., Rose, K.H.: Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. In: Computer Science in the Netherlands (CSN'95), pp. 62–72 (1995)
3. Danos, V., Joinet, J.-B., Schellinx, H.: A new deconstructive logic: linear logic. The. Journal of Symbolic Logic 62(2), 755–807 (1997)
4. Espírito Santo, J.: Revisiting the correspondence between cut elimination and normalisation. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 600–611. Springer, Heidelberg (2000)
5. Gentzen, G.: Investigations into logical deduction. In: Szabo, M.E. (ed.) the collected papers of Gerhard Gentzen, pp. 68–131. North-Holland, Amsterdam (1969)
6. Herbelin, H.: A λ-calculus structure isomorphic to Gentzen-style sequent calculus. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 61–75. Springer, Heidelberg (1994)
7. Ikeda, S., Nakazawa, K.: Strong normalization proofs by CPS-translations. Information Processing Letters 99, 163–170 (2006)
8. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödels's T. Archive for Mathematical Logic 42, 59–87 (2003)
9. Kikuchi, K.: A direct proof of strong normalization for an extended Herbelin's calculus. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 244–259. Springer, Heidelberg (2004)
10. Kikuchi, K.: On a local-step cut-elimination procedure for the intuitionistic sequent calculus. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 120–134. Springer, Heidelberg (2006)
11. Prawitz, D.: Natural deduction, a proof-theoretical study. Almqvist & Wiksell (1965)
12. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism. Elsevier, Amsterdam (2006)
13. Urban, C.: Classical Logic and Computation. PhD thesis, University of Cambridge (2000)
14. Urban, C., Bierman, G.M.: Strong normalisation of cut-elimination in classical logic. Fundamenta Informaticae 45, 123–155 (2001)
15. von Plato, J.: Natural deduction with general elimination rules. Archive for Mathematical Logic 40, 541–567 (2001)
16. Zucker, J.: The correspondence between cut-elimination and normalization. Annals of Mathematical Logic 7, 1–112 (1974)

Polynomial Size Analysis of First-Order Functions

Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen

Institute for Computing and Information Sciences
Radboud University Nijmegen
{O.Shkaravska,R.vanKesteren,M.vanEekelen}@cs.ru.nl

Abstract. We present a size-aware type system for first-order shapely function definitions. Here, a function definition is called *shapely* when the size of the result is determined exactly by a polynomial in the sizes of the arguments. Examples of shapely function definitions may be matrix multiplication and the Cartesian product of two lists.



The type checking problem for the type system is shown to be undecidable in general. We define a natural syntactic restriction such that the type checking becomes decidable, even though size polynomials are not necessarily linear or monotonic. Furthermore, a method that infers polynomial size dependencies for a non-trivial class of function definitions is suggested.[1]

Keywords: Shapely Functions, Size Analysis, Type Checking, Diophantine equations.

1 Introduction

We explore typing support for checking size dependencies for *shapely* first-order function definitions (functions for short). The shapeliness of these functions lies in the fact that the size of the result is a polynomial in terms of the arguments' sizes.

Without loss of generality, we restrict our attention to a language with polymorphic lists as the only data-type. For such a language, this paper develops a size-aware type system for which we define a fully automatic type checking procedure.

A typical example of a shapely function in this language is the Cartesian product, which is given below. It uses the auxiliary function pairs that creates pairs of a single value and the elements of a list. To get a Cartesian product, cprod does this for all elements from the first list separately and appends the resulting intermediate lists. Furthermore, the function definition of append is assumed:

$$\begin{array}{lll}\mathsf{cprod}(x, y) = \mathsf{match}\ x\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ & \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{append}(\mathsf{pairs}(hd, y),\ \mathsf{cprod}(tl, y))\end{array}$$

[1] This research is sponsored by the Netherlands Organisation for Scientific Research (NWO), project Amortized Heap Space Usage Analysis (AHA), grantnr. 612.063.511.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 351–365, 2007.
© Springer-Verlag Berlin Heidelberg 2007

where

$$\mathsf{pairs}(x,y) = \mathsf{match}\ y\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow \mathsf{nil}$$
$$\mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{cons}(\mathsf{cons}(x, hd, \mathsf{nil}), \mathsf{pairs}(x, tl))$$

Given two lists, for instance [1, 2, 3] and [4, 5], it returns the list with all pairs created by taking one element from the first list and one element from the second list: [[1, 4], [1, 5], [2, 4], [2, 5], [3, 4], [3, 5]]. Hence, given two lists of length n and m, it always returns a list of length nm containing pairs. This can be expressed in a type by $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$.

Shapeliness is restrictive, but it is an important foundational step. It makes type checking decidable in the non-linear case and it allows to infer types "out-of-the-box", since experimental points are positioned exactly on the graph of the polynomial. Exact sizes will be used in future work to derive lower/upper bounds on the output sizes because many non-shapely functions may be transformed into shapely in such a way that the new functions output-size polynomial will be an lower/upper bound for output sizes of the original function. We need such bounds for our AHA project.

1.1 Related Work

Information about input-output size dependencies is applied in time and space analysis and optimization, because run time and heap-space consumption obviously depend on the sizes of the data structures involved in the computations. Knowledge of the exact size of data structures can be used to improve heap space analysis for expressions with destructive pattern matching. Amortized heap space analysis has been developed for linear bounds by Hofmann and Jost [5]. Precise knowledge of sizes is required to extend this approach to non-linear bounds. Another application of exact size information is *load distribution* for parallel computation. For instance, size information helps to distribute a storage effectively and to safely store vector fragments [3].

The analysis of (exact) input-output size dependencies of functions itself has been explored in a series of work. Some interesting work on shape analysis has been done by Jay and Sekanina [7]. In this work, a shapely program expression is translated into a corresponding abstract program expression over sizes. Thus, the dependency of the result size on the argument sizes has the form of a program expression. However, deriving an arithmetic function from it is beyond the scope of their work.

Functional dependencies of sizes in a *recurrent form* may be derived via program analysis and transformation, as in the work of Herrmann and Lengauer [6], or through a type inference procedure, as presented by Vasconcelos and Hammond [12]. Both results can be applied to non-shapely functions, higher-order functions and non-linear size expressions. However, solving the recurrence equations to obtain a closed-form solution is left as an open problem for external solvers. In the second paper monotonic bounds are studied.

To our knowledge, the only work yielding closed-form solutions for size dependencies is limited to monotonic dependencies. For instance, in the well-known

work of Pareto [8], where *non-strict* sized types are used to prove termination, monotonic linear upper bounds are inferred. There linearity is a sufficient condition for the type checking procedure to be decidable.In the series of works on polynomial quasi-interpretations [1] one studies polynomial upper bounds. The checking and inference rely on real arithmetic. Our approach differs twofold. Firstly, quasi-interpretations give monotonic bounds. With non-monotonic size dependencies polynomial quasi-interpretations may lead to significant overestimations. Secondly, to get exact bounds we use integer arithmetic instead of the real one.

The approaches summarized in the previous paragraphs either leave the (possibly undecidable) solving of recurrences as a problem external to their approach, or are limited to monotonic dependencies.

1.2 Contents of the Paper

In this work, we go beyond monotonicity and linearity and consider a type checking procedure for a first-order functional programming language (section 2) with polynomial size dependencies (section 3). We show that type checking is reduced to the entailment checking over Diophantine equations. Type checking is shown to be undecidable in general, but decidable under a natural syntactic condition ("no-let-before-match", section 4). We suggest a method for type inference in section 5. It terminates on a nontrivial class of shapely functions. It non-terminates when either the function under consideration non-terminates, or it is not shapely, or its correct size dependency is rejected by the type-checker due type-checker's incompleteness. (Note that there is no complete shapeliness-checker.)

In section 6 we define a heap-aware semantics of types and expressions and sketch the proof of the soundness statement with respect to this semantics. Finally, in section 7 we overview the results and discuss further work.

2 Language

The typing system is designed for a first-order functional language over integers and (polymorphic) lists.

The syntax of language expressions is defined by the following grammar, where c ranges over integer constants, x and y denote zero-order program variables, and f denotes a function name (the example in the introduction used a sugared version of this syntax):

$$\begin{array}{lll} \mathit{Basic}\ b & ::= & c \mid \mathsf{nil} \mid \mathsf{cons}(x, y) \mid f(x_1, \ldots, x_n) \\ \mathit{Expr}\ \ e & ::= & \mathsf{letfun}\ f(x_1, \ldots, x_n) = e_1\ \mathsf{in}\ e_2 \\ & & \mid b \mid \mathsf{let}\ x = b\ \mathsf{in}\ e \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & & \mid \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_1 \\ & & \qquad\qquad\qquad \mid \mathsf{cons}(x_{\mathtt{hd}}, x_{\mathtt{tl}}) \Rightarrow e_2 \end{array}$$

The syntax distinguishes between zero-order let-binding of variables and first-order letfun-binding of functions. In a function body, the only free program

variables that may occur are its parameters: $FV(e_1) \subseteq \{x_1, \ldots, x_n\}$. The operational semantics is standard, therefore the definition is postponed until it is used to prove soundness (section 6.1).

We prohibit head-nested let-expressions and restrict sub-expressions in function calls to variables to make type-checking straightforward. Program expressions of a general form may be equivalently transformed to expressions of this form. It is useful to think of the presented language as an intermediate language.

3 Type System

Sized types are derived using a type and effect system in which types are annotated with size expressions. Size expressions are polynomials representing lengths of finite lists and arithmetic operations over these lengths:

$$\mathit{SizeExpr}\ p ::= \mathbb{N} \mid n \mid p + p \mid p - p \mid p * p,$$

where n, possibly decorated with sub- and superscripts, denotes a size variable, which stands for any concrete size (natural number). For any natural number k, n^k denotes the k-fold product $n * \ldots * n$.

Zero-order types are assigned to program values, which are integers and finite lists. The list type is annotated by a size expression that represents the length of the list:

$$\mathit{Types}\ \tau ::= \mathtt{Int} \mid \alpha \mid \mathsf{L}_p(\tau),$$

where α is a type variable. Note that this structure entails that if the elements of a list are lists themselves, all these element-lists have to be of the same size. Thus, instead of lists it would be more precise to talk about matrix-like structures. For instance, the type $\mathsf{L}_6(\mathsf{L}_2(\mathtt{Int}))$ is given to a list which elements are all lists of exactly two integers, such as $[[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]$.

It is easy to see that sets $\mathsf{L}_0(\mathsf{L}_m(\mathtt{Int}))$ are equal and contain the single element $[\,]$ for all m. The similar holds for $\mathsf{L}_0(\mathsf{L}_m(\alpha))$. This induces the natural equivalence relation on types. For instance $\mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_p(\alpha))) \equiv \mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_{p'}(\alpha)))$. The canonical representative of this class is $\mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_0(\alpha)))$. Everywhere below τ (decorated with sub- or superscripts) denote in fact the canonical representative of $\tau_{\equiv}$. The equivalence expresses the fact that sizes of lists that do not exist, because a containing list is empty, are not important.

The sets $FV(\tau)$ and $FVS(\tau)$ of the free type and size variables of a type τ are defined inductively in the obvious way. Note, that $FVS(\mathsf{L}_0(\mathsf{L}_m(\alpha))) = \emptyset$, since the type is equivalent to $\mathsf{L}_0(\mathsf{L}_0(\alpha))$.

Zero-order types without size or type variables are ground types:

$$\mathit{GTypes}\ \tau^{\bullet} ::= \tau \text{ such that } FVS(\tau) = \emptyset \wedge FV(\tau) = \emptyset\,,$$

First-order types are assigned to shapely functions over values of a zero-order type. Let τ° denote a zero order type of which the annotations are all size variables. First-order types are then defined by:

$$\textit{FTypes}\ \tau^f ::= \tau_1^\circ \times \ldots \times \tau_n^\circ \to \tau_{n+1}$$
$$\text{such that } FVS(*\tau_{n+1}) \subseteq FVS(*\tau_1^\circ) \cup \cdots \cup FVS(*\tau_n^\circ)$$
$$\text{for all instantiations * of type variables with size expressions.}$$

Recalling the Cartesian product from the introduction, one expects append to be of type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$, pairs of type $\alpha \times \mathsf{L}_m(\alpha) \to \mathsf{L}_m(\mathsf{L}_2(\alpha))$, and cprod of type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$.

A context Γ is a mapping from zero-order variables to zero-order types. A signature Σ is a mapping from function names to first-order types. The definition of $FVS(-)$ is straightforwardly extended to contexts.

3.1 Typing Rules

A typing judgment is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e : \tau$, where D is a set of Diophantine equations which is used to keep track of the size information. In the current language, the only place where size information is available is in the nil-branch of the match-rule. The signature Σ contains the type assumptions for the functions that are going to be checked.

In the typing rules, $D \vdash p = p'$ means that $p = p'$ is derivable from D from equational reasoning in the ring of integers. $D \vdash \tau = \tau'$ is a shorthand that means that τ and τ' have the same underlying type and equality of their size annotations is derivable. The typing judgment relation is defined by the following rules:

$$\frac{}{D;\ \Gamma\ \vdash_\Sigma c : \mathtt{Int}}\ \textsc{IConst} \qquad \frac{D \vdash \tau = \tau'}{D;\ \Gamma,\ x:\tau\ \vdash_\Sigma x:\tau'}\ \textsc{Var}$$

$$\frac{FVS(\mathsf{L}_p(\tau)) \subseteq FVS(\Gamma) \qquad D \vdash p = 0}{D;\ \Gamma\ \vdash_\Sigma \mathsf{nil} : \mathsf{L}_p(\tau)}\ \textsc{Nil}$$

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd:\tau,\ tl:\mathsf{L}_{p'}(\tau)\ \vdash_\Sigma \mathsf{cons}(hd, tl) : \mathsf{L}_p(\tau)}\ \textsc{Cons}$$

$$\frac{\Gamma(x) = \mathtt{Int} \qquad D;\ \Gamma\ \vdash_\Sigma e_t : \tau \qquad D;\ \Gamma\ \vdash_\Sigma e_f : \tau}{D;\ \Gamma\ \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}\ \textsc{If}$$

$$\frac{x \notin dom(\Gamma) \qquad D;\ \Gamma\ \vdash_\Sigma e_1 : \tau_x \qquad D;\ \Gamma,\ x:\tau_x\ \vdash_\Sigma e_2 : \tau}{D;\ \Gamma\ \vdash_\Sigma \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x:\mathsf{L}_p(\tau')\ \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \qquad D;\ \Gamma, hd:\tau',\ x:\mathsf{L}_p(\tau'),\ tl:\mathsf{L}_{p-1}(\tau')\ \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{D;\ \Gamma,\ x:\mathsf{L}_p(\tau')\ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with} \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau}\ \textsc{Match}$$

The rule LetFun demands that all defined functions, including recursive ones, must be in the domain of the signature, and the corresponding first-order type must pass type-checking. We do not prohibit calls of not-defined functions in the code. In practice, a type checker may allow calls of undefined within the given code functions. This happens when a specification comes from a trusty source.

Such relaxed treatment of LetFun does make sense for functions written in another language. However, for the soundness proof one needs all called functions to be defined within an expression under consideration.

$$\frac{\Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{n+1} \qquad \mathsf{True};\ x_1 : \tau_1^\circ, \ldots, x_n : \tau_n^\circ\ \vdash_\Sigma e_1 : \tau_{n+1} \qquad D;\ \Gamma\ \vdash_\Sigma e_2 : \tau'}{D;\ \Gamma\ \vdash_\Sigma \mathsf{letfun}\ f(x_1, \ldots, x_n) = e_1\ \mathsf{in}\ e_2 : \tau'}\ \text{LetFun}$$

In the FunApp-rule, Θ computes the substitution $*$ from the first argument (whose size expressions, by definition of first order types, are always variables) to the second argument, and the set C of equations over size expressions from ${\tau_1}' \times \cdots \times \tau_k'$. The set C contains $p = p'$ if and only the expressions are substituted to the same size variable, like, for instance, to m in $\mathsf{L}_m(\mathtt{Int}) \times \mathsf{L}_m(\mathtt{Int}) \to \mathtt{Int}$.

$$\frac{\langle *, C\rangle = \Theta(\tau_1^\circ \times \cdots \times \tau_n^\circ, {\tau_1}' \times \cdots \times {\tau_n}') \qquad \Sigma(f) = \tau_1^\circ \times \ldots \times \tau_n^\circ \to \tau_{n+1} \qquad D \vdash *(\tau_{n+1}) = \tau_{n+1}' \qquad D \vdash C}{D;\ \Gamma, x_1 : {\tau_1}', \ldots, x_n : {\tau_n}'\ \vdash_\Sigma f(x_1, \ldots, x_k) : {\tau_{n+1}}'}\ \text{FunApp}$$

The type system needs no conditions on non-negativity of size expressions. Size expressions in types of meaningful data structures are always non-negative. The soundness of the type system (section 6.2) ensures that this property is preserved throughout (the evaluation of) a well-typed expression.

3.2 Example of Type Checking

Because for every syntactic construction there is only one typing rule that is applicable, type checking is straightforward. In the introduction, the Cartesian product was presented using a "sugared" syntax. Here, we present the cprod function in the language defined in section 2.

$$\begin{array}{l} \mathsf{letfun}\ \mathsf{cprod}(x, y) = \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ z_1 \quad = \mathsf{pairs}(hd, y) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{let}\ z_2 = \mathsf{cprod}(tl, y) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{append}(z_1, z_2) \end{array}$$

Functions pairs and append are assumed to be defined in the core syntax of the language as well. Hence, Σ contains the following types:

$$\begin{array}{ll} \Sigma(\mathsf{append}) & = \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha) \\ \Sigma(\mathsf{pairs}) & = \alpha \times \mathsf{L}_m(\alpha) \to \mathsf{L}_m(\mathsf{L}_2(\alpha)) \\ \Sigma(\mathsf{cprod}) & = \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha)) \end{array}$$

To type-check $\mathsf{cprod} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$ means to check:

Prove: $x : \mathsf{L}_n(\alpha), y : \mathsf{L}_m(\alpha)\ \vdash_\Sigma e_{\mathsf{cprod}} : \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$,

where e_{cprod} is the function body. This is demanded by the first branch of the LetFun-rule. Applying the Match-rule branches the proof:

NIL: $n = 0;\ y\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{nil}\!:\!\mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$
CONS: $hd\!:\!\alpha,\ x\!:\!\mathsf{L}_n(\alpha),\ tl\!:\!\mathsf{L}_{n-1}(\alpha),\ y\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma$
$\mathsf{let}\ z_1 \ = \mathsf{pairs}(hd, y)\!:\!\mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$
$\mathsf{in\ let}\ z_2 = \mathsf{cprod}(tl, y)$
$\mathsf{in\ append}(z_1, z_2)$

Applying the NIL-rule to the NIL-branch gives $n = 0 \vdash n * m = 0$, which is trivially true. The CONS-branch is proved by applying the LET-rule twice. This results in three proof obligations:

BIND-Z1: $hd\!:\!\alpha,\ y\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{pairs}(hd, y)\!:\!\tau_1$
BIND-Z2: $tl\!:\!\mathsf{L}_{n-1}(\alpha),\ y\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{cprod}(tl, y)\!:\!\tau_2$
BODY: $z_1\!:\!\tau_1, z_2\!:\!\tau_2\ \vdash_\Sigma \mathsf{append}(z_1, z_2)\!:\!\mathsf{L}_{n*m}(\alpha)$

From the applications of the FUNAPP-rule to BIND-Z1 and BIND-Z2 it follows that τ_1 should be $\mathsf{L}_m(\mathsf{L}_2(\alpha))$ and τ_2 should be $\mathsf{L}_{(n-1)*m}(\mathsf{L}_2(\alpha))$. Lastly, applying the FUNAPP-rule to BODY yields the proof obligation $\vdash (n-1)*m+m = n*m$, which is true in the axiomatics.

3.3 Example with Negative Coefficients

In contrast to the system presented by Vasconcelos and Hammond [12], where only subtraction of constants are allowed, our system allows negative coefficients in size expressions. Of course, this is only a valid size expression if the polynomial is non-negative for all values of its variables. Here, we show an example where this is the case. Given two lists, the function "subtracts" elements from lists simultaneously, till one of the lists is empty. Then, the Cartesian product of the rest list with itself is returned:

```
sqdiff (l1, l2) =
  match l1 with | nil ⇒ cprod l2 l2
                | cons(h, t) ⇒ match l2 with | nil ⇒ cprod l1 l1
                                             | cons(h', t') ⇒ sqdiff (t, t')
```

It can be checked that sqdiff has type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))$.

4 Decidability Issues for Type Checking

In the examples above, type checking ends up with a set of entailments like $n = 0 \vdash 0 = n * m$ or $\vdash m + m * (n-1) = m * n$ that have to hold. However, we show that there is no procedure that can check all entailments that possibly arise. To make type checking decidable, we formulate a syntactical condition on the structure of a program expression that ensures the entailments have a trivial form. The idea is to *prohibit pattern-matchings in a let-body.*

4.1 Type Checking in General Is Undecidable

We show that the existence of a procedure that may check all possible entailments at the end of type checking is reduced to Hilbert's tenth problem: whether

there exists a general procedure that given a polynomial with integer coefficients decides if this polynomial has natural roots or not.[2] Matiyasevich [10] has shown that such a procedure does not exist. This means that type checking, in the general case, is undecidable as well.

We show that type checking is reducible to a procedure of checking if arbitrary size polynomials of shapely functions have natural roots. It turns out that the latter is the same as finding natural roots of integer polynomials.

Consider the following expression e_H with free variables $x_1, \ldots, x_k$:

$$\begin{array}{l} \mathsf{let}\ x = f_0(x_1, \ldots, x_k)\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow f_1(x_1, \ldots, x_k) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow f_2(x_1, \ldots, x_k) \end{array}$$

We check if it has the type $\mathsf{L}_{n_1}(\alpha_1) \times \ldots \times \mathsf{L}_{n_k}(\alpha_k) \longrightarrow \mathsf{L}_{p(n_1,\ldots,n_k)}(\alpha)$, given that $f_i : \mathsf{L}_{n_1}(\alpha_1) \times \ldots \times \mathsf{L}_{n_k}(\alpha_k) \longrightarrow \mathsf{L}_{p_i(n_1,\ldots,n_k)}(\alpha)$, with $i = 0, 1, 2$. Then at the end of the type checking procedure we obtain the entailment:

$$p_0(n_1, \ldots, n_k) = 0 \vdash p_1(n_1, \ldots, n_k) = p(n_1, \ldots, n_k).$$

Even if p and p_1 are not equal, say $p_1 = 0$ and $p = 1$, it does not mean that type checking fails; it might not be possible to enter the "bad" nil-branch. To check if the nil-branch is entered means to check if $p_0 = 0$ has a solution in natural numbers. Thus, a type-checker for any size polynomial p_0 must be able to define if it has natural roots or not.

Checking if any size polynomial has roots in natural numbers, is the same as checking whether an arbitrary polynomial has roots or not. For polynomials $q(n_1, \ldots, n_k) = 0$ if and only if $q^2(n_1, \ldots, n_k) = 0$ so it is sufficient to prove that the square of any polynomial is a size polynomial for some shapely function. First, note that any polynomial q may be presented as the difference $q_1 - q_2$ of two polynomials with non-negative coefficients[3]. So, $q^2 = (q_1 - q_2)^2$ is a size polynomial, obtained by superposition of sqdiff with q_1 and q_2. Here q_1 and q_2 are size polynomials with positive coefficients for corresponding compositions of cprod and append functions.

So, existence of a general type-checker reduces to solving Hilbert's tenth problem. Hence, type checking is undecidable.

We can show this in a more constructive way using the stronger form of the undecidability of Hilbert's tenth problem: for any type-checking procedure $\mathcal{I}$ one can construct an expression, for which $\mathcal{I}$ fails to give the correct answer. We will use the result of Matiyasevich who has proved the following: there is a one-parameter Diophantine equation $W(a, n_1, \ldots, n_k) = 0$ and an algorithm which for given algorithm $\mathcal{A}$ produces a number $a_{\mathcal{A}}$ such that $\mathcal{A}$ fails to give the correct answer for the question whether equation $W(a_{\mathcal{A}}, n_1, \ldots, n_k) = 0$ has a solution in $(n_1, \ldots, n_k)$. So, if in the example above one takes the function

[2] The original formulation is about integer roots. However, both versions are equivalent and logicians consider natural roots.

[3] If $q = \Sigma a_{i_1,\ldots,i_k} x_1^{i_1} \ldots x_k^{i_k}$, then $q_1 = \Sigma_{a_{i_1,\ldots,i_k} \geq 0} a_{i_1,\ldots,i_k} x_1^{i_1} \ldots x_k^{i_k}$, and $q_2 = \Sigma_{a_{i_1,\ldots,i_k} < 0} |a_{i_1,\ldots,i_k}| x_1^{i_1} \ldots x_k^{i_k}$.

f_0 such that its size polynomial p_0 is the square of the $W(a_{\mathcal{I}}, n_1, \ldots, n_k)$ and $p = 1$, $p_1 = 0$, then the type checker $\mathcal{I}$ fails to give the correct answer for e_H.

For checking a particular expression it is sufficient to solve the corresponding sets of Diophantine equations. Type checking depends on decidability of Diophantine equations from D in any entailment $D \vdash p = p'$, where p is not equal to p' in general (but might be if the equations from D hold). If we have a solution for D we can substitute this solution in p and p'. A solution over variables $n_1, \ldots, n_m, n_{m+1}, \ldots, n_k$ is a set of equations $n_i = q_i(n_{m+1}, \ldots, n_k)$ where $1 \leq i \leq m$. The expressions for n_i are substituted into $p = p'$ and one trivially checks the equality of the two polynomials over $n_{m+1}, \ldots, n_k$ in the axiomatics of the integers' ring. Recall that two polynomials are equal if and only if the coefficient at monomials with the same degrees of variables are equal.

4.2 Syntactical Condition for Decidability

The most simple way to ensure decidability is to require that all equations in D have the form $n = c$, where c is a constant. This would in particular exclude the example e_H from above. As we will see below, this requirement can be fulfilled by imposing a syntactical condition on program expressions: "no let before match".

The refined grammar of the language is defined as the main grammar where the let-construct in e is replaced by $\mathsf{let}\ x = b\ \mathsf{in}\ e_{nomatch}$ with

$$\begin{array}{rll} e_{nomatch} := b & & \mid \mathsf{letfun}\ f(x_1, \ldots, x_n) = e\ \mathsf{in}\ e' \\ & \mid \mathsf{let}\ x = b\ \mathsf{in}\ e_{nomatch} & \mid \mathsf{if}\ x\ \mathsf{then}\ e'_{nomatch}\ \mathsf{else}\ e'_{nomatch} \end{array}$$

Theorem 1. *Let a program expression e satisfy the refined grammar, and let us check the judgment* $\mathsf{True};\ x_1 : \tau_1^o, \ldots, x_k : \tau_k^o \vdash_\Sigma e : \tau$. *Then, at the end of the type-checking procedure one has to check entailments of the form*

$$D \vdash p' = p,$$

where D is a set of equations of the form $n - c = 0$ for some $n \in FVS(\tau_1^o \times \ldots \times \tau_k^o)$ and constant c and p, p' are polynomials in $FVS(\tau_1^o \times \ldots \times \tau_k^o)$.

Sketch of the proof. Consider a path in the type checking tree which ends up with some $D \vdash p' = p$ and let an equation $q = 0$ belong to D. It means that in the path there is the nil-branch of the pattern matching for some $x : \mathsf{L}_q(\tau)$.

By induction on the length of the path, one can show that $q = n - c$ for some size variable $n \in FVS(\tau_1 \times \ldots \times \tau_k)$ and some constant c. This uses the fact, that variables which are not free in an expression and pattern-matched may be introduced only by another pattern-matching, but not a let-binding. The technical report contains the full proof [11].

Note, that prohibiting pattern matching in `let`-bodies is very natural, since it prohibits "risky" definitions of the form $f(x) = g(f(f_0(x)))$. Here x is a non-nil list, and f_0 is a function over lists, possibly with the property $|f_0(x)| \geq |x|$, with $|\cdot|$ denoting length, so termination of f becomes questionable. In a "shapely world" the condition $|f_0(x)| < |x|$ for all x starting from a certain x_0, which ensures termination, implies $|f_0(x)| = |x| - c$ or $|f(x)| = c$ for some constant c.

In principle, any program expression that does not do pattern matching on a variable bound by a let-expression may be recoded so that it satisfies the refined grammar and defines the same map. For instance, an expression

$$\mathsf{let}\ x' = f_0(y)\ \mathsf{in}\ \ \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow f_1(x,\, x') \\ \mid \mathsf{cons}(hd, tl) \Rightarrow f_2(x,\ x') \end{array}$$

and the expression

$$\mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow \mathsf{let}\ x' = f_0(y)\ \mathsf{in}\ f_1(x,\, x') \\ \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ x' = f_0(y)\ \mathsf{in}\ f_2(x,\, x') \end{array}$$

define the same map of lists.

Of course, the syntactical condition of the theorem may be relaxed. One may allow expressions with pattern-matching in a let-body, assuming that functions that appear in let-bindings, like f_0, give rise to solvable Diophantine equations. For instance, when p_0 is a linear function, one of the variables is expressed via the others and the constant and substituted into $p_1 = p$. Or, p_0 is a 1-variable quadratic, cubic or degree 4 equation. We leave relaxations of the condition for future work.

5 Method for Type Inference

Here we discuss type inference under the syntactical condition defined in the previous section. Since we consider shapely functions, there is a way to reduce type inference to type-checking using the well-known fact that a finite polynomial is defined by a finite number of points.

For each size polynomial in the output type of a given function, one assumes a hypothesis about the degree and the variables. Then, to obtain the coefficients, the function is run (preferably in a sand-box) as many times as the number of coefficients the polynomial has. This finite number of input-output size pairs defines a system of linear equations, where the unknowns are the coefficients of the polynomial. When (the sizes of the data-structures in) the set of input data satisfies some criteria known from the polynomial interpolation theory [4,9], the system has a unique solution. Input sizes that satisfy these criteria, which are nontrivial for multivariate polynomials, can be determined algorithmically.

The interpolation theory used in the previous paragraph finds the Lagrange interpolation of a size function. If the hypothesis about the degree and the variables of the size expression was correct, the Lagrange interpolation coincides with that desired size function. To check if this is the case, the interpolation is given to the type checking procedure. If it passes, it is correct. Otherwise, one may repeat the procedure for a higher degree of the polynomial.

The method, that is the sequence of such loops, non-terminates when

- the function under consideration does not terminate on test data,
- the function is non-shapely,
- the function is shapely but the type-checker rejects it due to the type system's incompleteness (see 6.3). Note that no complete algorithm for shapeliness-checking exists, even for functions subject to the syntactical restriction.

The method infers polynomial size dependencies for a nontrivial class of shapely functions.

For instance, standard type inference for the underlying type system yields that the function cprod has the following underlying type cprod : $\mathsf{L}(\alpha) \times \mathsf{L}(\alpha) \longrightarrow \mathsf{L}(\mathsf{L}(\alpha))$. Adding size annotations with unknown output polynomials gives cprod : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \longrightarrow \mathsf{L}_{p_1}(\mathsf{L}_{p_2}(\alpha))$. We assume p_1 is quadratic so we have to compute the coefficients in its presentation:

$$p_1(x, y) = a_{0,0} + a_{0,1}x + a_{1,0}y + a_{1,1}xy + a_{0,2}x^2 + a_{2,0}y^2$$

Running the function cprod on six pairs of lists of length 0, 1, 2 yields:

n	m	x	y	cprod(x, y)	$p_1(n, m)$	$p_2(n, m)$
0	0	[]	[]	[]	0	?
1	0	[0]	[]	[]	0	?
0	1	[]	[0]	[]	0	?
1	1	[0]	[1]	[[0, 1]]	1	2
2	1	[0, 1]	[2]	[[0, 2], [1, 2]]	2	2
1	2	[0]	[1, 2]	[[0, 1], [0, 2]]	2	2

This defines the following linear system for the external output list:

$$\begin{aligned} a_{0,0} &= 0 \\ a_{0,0} + a_{0,1} + a_{0,2} &= 0 \\ a_{0,0} + a_{1,0} + a_{2,0} &= 0 \\ a_{0,0} + a_{0,1} + a_{1,0} + a_{0,2} + a_{1,1} + a_{2,0} &= 1 \\ a_{0,0} + 2a_{0,1} + a_{1,0} + 4a_{0,2} + 2a_{1,1} + a_{2,0} &= 2 \\ a_{0,0} + a_{0,1} + 2a_{1,0} + a_{0,2} + 2a_{1,1} + 4a_{2,0} &= 2 \end{aligned}$$

The unique solution is $a_{1,1} = 1$ and the rest of coefficients are zero. To verify whether the interpolation is indeed the size polynomial, one checks if cprod : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \longrightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$. This is the case, as was shown in section 3.2.

As an alternative way of finding the coefficients, one could try to directly solve the systems defined by entailments $D \vdash p = p'$. When the degree is assumed, the unknowns in these systems are the polynomial coefficients. However, the systems are nonlinear in general [11]. By combining testing with type checking we do not have to solve these nonlinear Diophantine equations anymore.

6 Semantics of the Type System

Informally, soundness of the type system ensures that "well-typed programs will not go wrong". This is achieved by demanding that, when a function is given meaningful values of the types required as arguments, the result will be a meaningful value of the output type.

In section 6.1, we formalize the notion of a meaningful value using a heap-aware semantics of types and give an operational semantics of the language. Section 6.2 formulates the soundness statement with respect to this semantics and sketches the proof. The system is shown not to be complete in section 6.3.

6.1 Semantics of Program Values and Expressions

In our semantic model, the purpose of the heap is to store lists. Therefore, it essentially is a collection of locations l that can store list elements. A location is the address of a cons-cell each consisting of a hd-field, which stores the value of the list element, and a tl-field, which contains the location of the next cons-cell of the list (or the NULL address). Formally, a program value is either an integer constant, a location, or the null-address and a heap is a finite partial mapping from locations and fields to such program values:

$$\begin{array}{llll} \mathit{Val}\ v & ::= & c \mid \ell \mid \mathtt{NULL} \qquad \ell \in \mathit{Loc} \qquad c \in \mathtt{Int} \\ \mathit{Hp}\ h & : & \mathit{Loc} \rightharpoonup \{\mathtt{hd}, \mathtt{tl}\} \rightharpoonup \mathit{Val} \end{array}$$

We will write $h[\ell.\mathtt{hd} := v_h,\ \ell.\mathtt{tl} := v_t]$ for the heap equal to h everywhere but in ℓ, which at the hd-field of ℓ gets value v_h and at the tl-field of ℓ gets value v_t.

The semantics w of a program value v is a set-theoretic interpretations with respect to a specific heap h and a ground type τ, via the four-place relation $v \models^h_\tau w$. Integer constants interprets themselves, and locations are interpreted as non-cyclic lists:

$$\begin{array}{lll} i & \models^h_{\mathtt{Int}} & i \\ \mathtt{NULL} & \models^h_{\mathsf{L}_0(\tau)} & [] \\ \ell & \models^h_{\mathsf{L}_n(\tau)} & w_{\mathtt{hd}} :: w_{\mathtt{tl}} \text{ iff } n \geq 1, \ell \in \mathit{dom}(h), \\ & & \qquad h.\ell.\mathtt{hd} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\tau} w_{\mathtt{hd}}, \\ & & \qquad h.\ell.\mathtt{tl} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\mathsf{L}_{n-1}(\tau)} w_{\mathtt{tl}} \end{array}$$

where $h|_{\mathit{dom}(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except for ℓ, where it is undefined.

When a function body is evaluated, a frame store maintains the mapping from program variables to values. It only contains the actual function parameters, thus preventing access beyond the caller's frame. Formally, a frame store is a finite partial map from variables to values:

$$\mathit{Store}\ s\ :\ \mathit{ExpVar} \rightharpoonup \mathit{Val}$$

Using heaps and frame stores, and maintaining a mapping $\mathcal{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is defined by the following rules:

$$\frac{c \in \mathtt{Int}}{s;\ h;\ \mathcal{C}\ \vdash\ c\ \rightsquigarrow\ c;\ h}\ \text{OSICons} \qquad \frac{}{s;\ h;\ \mathcal{C}\ \vdash\ x\ \rightsquigarrow\ s(x);\ h}\ \text{OSVar}$$

$$\frac{}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{nil}\ \rightsquigarrow\ \mathtt{NULL};\ h}\ \text{OSNil}$$

$$\frac{s(hd) = v_{\mathtt{hd}} \qquad s(tl) = v_{\mathtt{tl}} \qquad \ell \notin \mathit{dom}(h)}{s;\ h\ \vdash\ \mathsf{cons}(hd, tl)\ \rightsquigarrow\ \ell;\ h[\ell.\mathtt{hd} := v_{\mathtt{hd}},\ \ell.\mathtt{tl} := v_{\mathtt{tl}}]}\ \text{OSCons}$$

$$\frac{s(x) \neq 0 \qquad s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \rightsquigarrow\ v;\ h'}\ \text{OSIfTrue}$$

$$\frac{s(x) = 0 \qquad s;\, h;\, \mathcal{C} \;\vdash\; e_2 \;\rightsquigarrow\; v;\, h'}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \;\rightsquigarrow\; v;\, h'}\ \text{OSIfFalse}$$

$$\frac{s(x) = \mathtt{NULL} \qquad s;\, h;\, \mathcal{C} \;\vdash\; e_1 \;\rightsquigarrow\; v;\, h'}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array} \;\rightsquigarrow\; v;\, h'}\ \text{OSMatch-Nil}$$

$$\frac{\begin{array}{c} h.s(x).\mathtt{hd} = v_{\mathtt{hd}} \qquad h.s(x).\mathtt{tl} = v_{\mathtt{tl}} \\ s[hd := v_{\mathtt{hd}}, tl := v_{\mathtt{tl}}];\, h \;\vdash\; e_2 \;\rightsquigarrow\; v;\, h' \end{array}}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array} \;\rightsquigarrow\; v;\, h'}\ \text{OSMatch-Cons}$$

$$\frac{s;\, h;\, \mathcal{C}[f := ((x_1, \ldots, x_n) \times e_1)] \;\vdash\; e_2 \;\rightsquigarrow\; v;\, h'}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{letfun}\ f(x_1, \ldots, x_n) = e_1\ \mathsf{in}\ e_2 \;\rightsquigarrow\; v;\, h'}\ \text{OSLetFun}$$

$$\frac{\begin{array}{c} s(x_1) = v_1 \ \ldots\ s(x_m) = v_n \qquad \mathcal{C}(f) = (y_1, \ldots, y_n) \times e_f \\ [y_1 := v_1, \ldots, y_n := v_n];\, h;\, \mathcal{C} \;\vdash\; e_f \;\rightsquigarrow\; v;\, h' \end{array}}{s;\, h;\, \mathcal{C} \;\vdash\; f(x_1, \ldots, x_n) \;\rightsquigarrow\; v;\, h'}\ \text{OSFunApp}$$

$$\frac{s;\, h;\, \mathcal{C} \;\vdash\; e_1 \;\rightsquigarrow\; v_1;\, h_1 \qquad s[x := v_1];\, h_1;\, \mathcal{C} \;\vdash\; e_2 \;\rightsquigarrow\; v;\, h'}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \;\rightsquigarrow\; v;\, h'}\ \text{OSLet}$$

6.2 Soundness

In this subsection the soundness theorem is formulated and a proof is sketched. The technical report [11] contains the full proof.

Let a valuation ϵ map size variables to concrete (natural) sizes and an instantiation μ map type variables to ground types:

$$\begin{array}{lll} \mathit{Valuation} & \epsilon : & \mathit{SizeVar} \to \mathbb{N} \\ \mathit{Instantiation} & \eta : & \mathit{TypeVar} \to \tau^{\bullet} \end{array}$$

Applied to a type, context, or size equation, valuations (and instantiations) map all variables occurring in it to their valuation (or instantiation) images.

The soundness statement is defined by means of the following two predicates. One indicates if a program value is meaningful with respect to a certain heap and ground type. The other does the same for sets of values and types, taken from a frame store and context respectively:

$$\begin{array}{ll} \mathit{Valid}_{\mathsf{val}}(v, \tau^{\bullet}, h) & = \exists_{w} [\ v\ \models^{h}_{\tau}\ w\ ] \\ \mathit{Valid}_{\mathsf{store}}(\mathit{vars}, \Gamma, s, h) & = \forall_{x \in \mathit{vars}} [\ \mathit{Valid}_{\mathsf{val}}(s(x), \Gamma(x), h)\ ] \end{array}$$

Now the soundness statement is straightforward:

Theorem 2. *Let* $s;\, h;\, [\,] \vdash e \rightsquigarrow v;\, h'$ *and all called in* e *functions are defined in it via the let-fun construct. Then for any context* Γ*, signature* Σ *and type* τ*, such that* $\mathsf{True};\ \Gamma \vdash_{\Sigma} e : \tau$ *is derivable in the type system, and any size valuation* ϵ *and type instantiation* η*, it holds that if the store is meaningful w.r.t. the context* $\eta(\epsilon(\Gamma))$ *then the output value is meaningful w.r.t the type* $\eta(\epsilon(\tau))$*:*

$$\forall_{\eta,\epsilon}[\ \mathit{Valid}_{\mathsf{store}}(FV(e), \eta(\epsilon(\Gamma)), s, h) \implies \mathit{Valid}_{\mathsf{val}}(v, \eta(\epsilon(\tau)), h')\ ]$$

Sketch of the proof. The proof is done by induction on the length of the operational semantics derivation tree and is presented in the technical report [11]. The proof for the LET-rule relies on the *benign sharing* [5] of data structures. It means that the heap data to be used further are not changed by the head expression in let. There are type systems approximating this semantic condition, e.g. linear typing and uniqueness typing [2] We consider sharing aware type systems separately and combine with the resource aware one afterwords.

6.3 Completeness

The system is not complete – there are shapely functions that are not well-typed. For instance, the type checking fails for the function $\mathsf{faildueif} : \mathsf{L}_n(\mathtt{Int}) \rightarrow \mathsf{L}_n(\mathtt{Int})$ defined by:

$\mathsf{letfun\ faildueif}(x) = \mathsf{let}\ y = \mathsf{length}(x)\ \mathsf{in\ if}\ y\ \mathsf{then}\ x\ \mathsf{else\ nil}$

where $\mathsf{length}(x)$ returns the length of list x. We believe that in some cases program transformations might help to make such functions typable.

7 Conclusion and Further Work

We have presented a natural syntactic restriction such that type checking of a size-aware type system for first-order shapely functions is decidable for polynomial size expressions without any limitations on the degree of the polynomials.

A non-standard, practical method to infer types is introduced. It uses run-time results to generate a set of equations. These equations are linear and hence automatically solvable. The method terminates on a non-trivial class of shapely functions.

7.1 Further Work

The system is defined for polymorphic lists. In principle, the system may be extended so that more general data structures will be allowed. This extension should not influence the approach itself, however it brings additional technical overhead.

An obvious limitation of our approach is that we consider only shapely functions. In practice, one is often interested to obtain upper bounds on space complexity for non-shapely functions. A simple example where for a non-shapely function an upper bound would be useful, is the function to `insert` an element in a list, provided the list does not contain the element. In the future we plan to consider code transformations which, given a non-shapely function `f` with upper bound (worst-case) complexity `c`, translate it into a shapely function `f'` with complexity `c`. Effectively, this will make the analysis applicable to non-shapely functions obtaining upper bounds on the space consumption complexity.

We plan to add non-trivial sizes to integers. At the same time leaving out non-sized integers will result in lists with elements of different sizes. Hence, the

decision how to add sizes to integers is connected to the problem of using sized and non-sized types within the same system. We leave it for future work based on [12] and [7].

Addition of other data structures and extension to non-shapely functions will open the possibility to use the system for an actual programming language.

References

1. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations, a way to control resources. Theoretical Computer Science (to appear)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science 6, 579–612 (1996)
3. Chatterjee, S., Blelloch, G.E., Fischer, A.L.: Size and access inference for data-parallel programs. PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, pp. 130–144 (1991)
4. Chui, C., Lai, H.C.: Vandermonde determinant and Lagrange interpolation in R^s. Nonlinear and convex analysis, pp. 23–35 (1987)
5. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. SIGPLAN Not. 38(1), 185–197 (2003)
6. Herrmann, C.A., Lengauer, C.: A transformational approach which combines size inference and program optimization. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 199–218. Springer, Heidelberg (2001)
7. Jay, C.B., Sekanina, M.: Shape checking of array programs. In: Computing: the Australasian Theory Seminar, Proceedings. Australian Computer Science Communications, vol. 19, pp. 113–121 (1997)
8. Pareto, L.: Sized Types. Dissertation for the Licentiate Degree in Computing Science. Chalmers University of Technology (1998)
9. Lorenz, R.A.: Multivariate Birkhoff Interpolation. LNCS, vol. 1516. Springer, Heidelberg (1992)
10. Matiyasevich, Y., Jones, J.-P.: Proof or recursive unsolvability of Hilbert's tenth problem. American Mathematical Monthly 98(10), 689–709 (1991)
11. Shkaravska, O., van Kesteren, R., van Eekelen, M.: polynomial size analysis of first-order functions. Technical Report ICIS-R07004, Radboud University Nijmegen (2007)
12. Vasconcelos, P.-B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs (Revised Papers). In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer, Heidelberg (2004)

Simple Saturated Sets for Disjunction and Second-Order Existential Quantification

Makoto Tatsuta

National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
tatsuta@nii.ac.jp

Abstract. This paper gives simple saturated sets for disjunction and second-order existential quantification by using the idea of segments in Prawitz's strong validity. Saturated sets for disjunction are defined by Pi-0-1 comprehension and those for second-order existential quantification are defined by Sigma-1-1 comprehension. Saturated-set semantics and a simple strong normalization proof are given to the system with disjunction, second-order existential quantification, and their permutative conversions. This paper also introduces the contraction property to saturated sets, which gives us saturated sets closed under union. This enables us to have saturated-set semantics for the system with union types, second-order existential types, and their permutative conversions, and prove its strong normalization.

1 Introduction

Recently permutative conversions have been studied actively [2,3,4,5,7,10,11]. Permutative conversions transform a proof with a disjunction or existential quantification elimination rule followed by an elimination rule into a proof with the second rule in the minor deduction of the first rule. Permutative conversions are indispensable for normalizing a proof in a natural deduction system with disjunction or existential quantification, since without permutative conversions, a normal proof fails to have the subformula property. Permutative conversions also give program transformation for if-then-else statements and abstract data types [6].

Strong normalization of systems with permutative conversions has been investigated for a long time since Prawitz developed proof theory for natural deduction [8,9]. Moreover, since [4] proposed a new idea which gave us perspective of strong normalization with permutative conversions, many researchers have been interested in this subject and trying to give a simple proof of strong normalization.

So far two methods are known to prove strong normalization of a second-order system with permutative conversions. One method is saturated sets or reducibility [5,7,10] and another method is CPS-translation [3].

The technique of saturated sets gives us a proof of strong normalization as well as set-theoretic interpretation of types. This enables us to investigate

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 366–380, 2007.
© Springer-Verlag Berlin Heidelberg 2007

other semantical properties of types, for example, characterization of persistently strongly normalizing terms [12].

Saturated sets had two problems. One is that the definition of saturated sets for disjunction and second-order existential quantification was complicated. In both [5] and [7], the definition of the saturated sets for disjunction used Π^1_1-comprehension, and similar ideas would require Π^1_2-comprehension to define their saturated sets for second-order existential quantification. Another problem is that some ways of defining saturated sets were not closed under union, that is, $S_1 \cup S_2$ may not be a saturated set even if S_1 and S_2 are saturated sets. Such saturated sets could not interpret union types by set-theoretic union.

This paper solves these two problems. First, we will give simple saturated sets for disjunction and second-order existential quantification. Π^0_1-comprehension and Σ^1_1-comprehension are sufficient to define our saturated sets for disjunction and second-order existential quantification, respectively. Moreover, the definition of saturated sets for disjunction is predicative. So we can expect to apply this saturated-set semantics to various systems. In order to get this solution, we use the idea of segments, which were used by Prawitz in [9,11] to define strong validity.

Secondly, we will provide saturated sets closed under set-theoretic union. This enables us to interpret systems with union types where union types are interpreted by set-theoretic union. This technique works for a system with implication, second-order existential quantification, conjunction, intersection, union, and second-order universal quantification. This is achieved by requiring the contraction property stating that if a term is in a saturated set, then its contraction is also in the set.

Section 2 defines the second-order natural deduction NJ_2 in Curry-style. Section 3 gives new saturated sets and proves strong normalization of NJ_2. Section 4 provides the system $\lambda^{\to\exists\wedge\cap\cup\forall}$ with second-order existential types, intersection types, and union types. Section 5 shows that the same saturated sets in Section 3 works also for this system and proves its strong normalization. Section 6 gives concluding remarks.

2 The System NJ_2 with Disjunction and Second-Order Existential Quantification

We will give the definition of the system NJ_2, the second order intuitionistic natural deduction with permutative conversions in Curry style. It has disjunction, existential quantification, and their permutative conversions.

Definition 2.1 (Language)

Type variables $X, Y, Z, \ldots$.

Types $A, B, \ldots ::= X | A \to B | A \vee B | \exists X A | A \wedge B | \forall X A$.

Variables $x, y, z, \ldots$.

Terms $M, N, L, P, Q ::= x | \lambda x.M | MN | \langle 0, M\rangle | \langle 1, M\rangle | M[x.N, y.L] |$
$\langle \exists, M\rangle | M[x, N] | \langle M, N\rangle | Mp_0 | Mp_1$

Λ is the set of terms.

Substitutions $M[x := N]$ and $A[X := B]$ are defined in a familiar way.

Definition 2.2 (Typing rules)
Assumptions

$$x : A$$

Inference rules

$$\dfrac{\begin{array}{c}[x:A]\\ \vdots\\ M : B\end{array}}{\lambda x.M : A \to B}\ (\to I) \qquad \dfrac{M : A \to B \quad N : A}{MN : B}\ (\to E)$$

$$\dfrac{M : A}{\langle 0, M\rangle : A \vee B}\ (\vee I1) \qquad \dfrac{M : B}{\langle 1, M\rangle : A \vee B}\ (\vee I2)$$

$$\dfrac{M : A \vee B \quad \begin{array}{c}[x:A]\\ \vdots\\ N : C\end{array} \quad \begin{array}{c}[y:B]\\ \vdots\\ L : C\end{array}}{M[x.N, y.L] : C}\ (\vee E)$$

$$\dfrac{M : A[X := B]}{\langle \exists, M\rangle : \exists X A}\ (\exists I) \qquad \dfrac{M : \exists X A \quad \begin{array}{c}[x:A]\\ \vdots\\ N : C\end{array}}{M[x.N] : C}\ (\exists E)$$

$$\dfrac{M : A \quad N : B}{\langle M, N\rangle : A \wedge B}\ (\wedge I) \quad \dfrac{M : A \wedge B}{M p_0 : A}\ (\wedge E1) \quad \dfrac{M : A \wedge B}{M p_1 : B}\ (\wedge E2)$$

$$\dfrac{M : A}{M : \forall X A}\ (\forall I) \qquad \dfrac{M : \forall X A}{M : A[X := B]}\ (\forall E)$$

The rules $(\forall I)$ and $(\exists E)$ have standard variable conditions.

Definition 2.3 (Reduction rules)
β-reductions:

$$\begin{array}{ll}
(\beta\to) & (\lambda x.M)N \to M[x := N] \\
(\beta \vee 1) & \langle 0, M\rangle[x.N, y.L] \to N[x := M] \\
(\beta \vee 2) & \langle 1, M\rangle[x.N, y.L] \to L[y := M] \\
(\beta\exists) & \langle \exists, M\rangle[x.N] \to N[x := M] \\
(\beta \wedge 1) & \langle M, N\rangle p_0 \to M \\
(\beta \wedge 2) & \langle M, N\rangle p_1 \to N
\end{array}$$

Eliminators $E ::= M|[x.M, y.N]|[x.M]|p_0|p_1$
Permutative conversions:

$$\begin{array}{ll}
(\pi\vee) & M[x.N, y.L]E \to M[x.NE, y.LE] \\
(\pi\exists) & M[x.N]E \to M[x.NE]
\end{array}$$

A context $\Xi[\ ]$ is defined in a standard way.
Congruency:

$$(congr) \qquad \Xi[M] \to \Xi[N] \quad \text{if} \quad M \to N.$$

The relation $\to^*$ is the reflexive transitive closure of the relation $\to$. We say that M reduces to N if $M \to^* N$.

Remark. Subject reduction property and Church Rosser property hold.

A term M is strongly normalizing if there is no infinite reduction sequence $M \to M_1 \to M_2 \to \ldots$ beginning with M. SN is the set of strongly normalizing terms.

The next section will prove that every term typed in the system NJ_2 is strongly normalizing.

Remark. If we choose the following rules for second-order existential quantification instead, the subject reduction property fails to hold for that system.

$$\frac{M : A[X := B]}{M : \exists X A}\ (\exists I) \qquad \frac{M : \exists X A \quad \begin{array}{c}[x : A]\\ \vdots \\ N : C\end{array}}{N[x := M] : C}\ (\exists E)$$

A counterexample is: $M_1 = z((\lambda x.x)xy)((\lambda x.x)xy)$, $M_2 = z(xy)((\lambda x.x)xy)$. Then we have $x : B \to \exists X(X \to X), y : B, z : \forall X((X \to X) \to (X \to X) \to C) \vdash M_1 : C$, but we do not have $x : B \to \exists X(X \to X), y : B, z : \forall X((X \to X) \to (X \to X) \to C) \vdash M_2 : C$.

3 Strong Normalization for $\mathbf{NJ_2}$

We will provide saturated-set semantics for NJ_2 with simple saturated sets and give a simple proof of strong normalization for NJ_2. To do this, we will introduce new saturated sets for disjunction and second-order existential quantification by using the idea of segments used in Prawitz's strong validity. We will also introduce the contraction property to saturated sets, which will play an important role for union types in Section 5.

Notation. The symbol $=$ is used for the syntactical identity modulo bound variable renaming. We will use vector notation to denote a sequence. For example, $\vec{M}$ denotes the sequence $M_1, M_2, \ldots, M_n$ and $M\vec{N}$ denotes $MN_1N_2 \ldots N_n$. Length$(\vec{M})$ is the length of the sequence $\vec{M}$.

When M is in SN, $|M|$ is the maximum length of its reduction sequence. $|\vec{M}|$ is $\Sigma_{i=1}^{n}|M_i|$ where $\vec{M} = M_1, M_2, \ldots, M_n$. $|E|$ is defined as follows: $|[x.M, y.N]| = |M| + |N|$, $|[x.M]| = |M|$, and $|p_0| = |p_1| = 0$.

When we say induction on (n, m), this means induction on the lexicographical order (n, m) such that $(n, m) < (n', m')$ if $n < n'$ and $(n, m) < (n, m')$ if $m < m'$.

Our saturated sets are the same as those in [5] except the contraction property (4) and the clauses (8) to (10) for second-order quantification, and the reduction closure property (12). The contraction property is new and will really work for union types in Section 5. Without (4), the clause (8) would require an additional condition $N\vec{E} \in S$, which will make saturated sets not closed under intersection.

Definition 3.1 (Saturated sets). The relation $N \succ M$ is defined to hold if N is obtained from M by replacing some (maybe zero) free occurrences of some variable by some term.

A set S of terms is saturated if the following hold, where $R ::= M|p_0|p_1$.

(1) $S \subseteq \mathrm{SN}$
(2) $x\vec{R} \in S$ if $x\vec{R} \in \mathrm{SN}$
(3) $(\lambda x.M)N\vec{E} \in S$ if $M[x := N]\vec{E} \in S$ and $N \in \mathrm{SN}$
(4) $M \in S$ if $N \succ M$ and $N \in S$
(5.1) $\langle 0, M\rangle[x_0.N_0, x_1.N_1]\vec{E} \in S$
if $N_0[x_0 := M]\vec{E} \in S$ and $N_1\vec{E} \in S$ and $M \in \mathrm{SN}$
(5.2) $\langle 1, M\rangle[x_0.N_0, x_1.N_1]\vec{E} \in S$
if $N_0\vec{E} \in S$ and $N_1[x_1 := M]\vec{E} \in S$ and $M \in \mathrm{SN}$
(6) $x\vec{R}[x_0.N_0, x_1, N_1] \in S$ if $x\vec{R} \in \mathrm{SN}$ and $N_0, N_1 \in S$
(7) $x\vec{R}[x_0.N_0, x_1.N_1]E\vec{E} \in S$ if $x\vec{R}[x_0.N_0E, x_1.N_1E]\vec{E} \in S$
(8) $\langle \exists, M\rangle[x.N]\vec{E} \in S$ if $N[x := M]\vec{E} \in S$ and $M \in \mathrm{SN}$
(9) $x\vec{R}[y.N] \in S$ if $x\vec{R} \in \mathrm{SN}$ and $N \in S$
(10) $x\vec{R}[y.N]E\vec{E} \in S$ if $x\vec{R}[y.NE]\vec{E} \in S$
(11.1) $\langle M, N\rangle p_0\vec{E} \in S$ if $M\vec{E} \in S$ and $N \in \mathrm{SN}$
(11.2) $\langle M, N\rangle p_1\vec{E} \in S$ if $N\vec{E} \in S$ and $M \in \mathrm{SN}$
(12) $M \in S$ if $N \to M$ and $N \in S$

SAT is the set of saturated sets. We will use notation like sat (1) to denote the clause in the above definition, for example, sat (1) denotes the clause (1). We will use S to denote a saturated set.

We define several operations for saturated sets to interpret types by saturated sets. We will show saturated sets are closed under these operations.

Definition 3.2. (1) $S_1 \to S_2 = \{M \in \Lambda | MN \in S_2 \text{ for all } N \in S_1\}$.
(2) $S_1 \wedge S_2 = \{M \in \Lambda | Mp_0 \in S_1 \text{ and } Mp_1 \in S_2\}$.

To define simple saturated sets for disjunction and second-order existential quantification, we have to use the following auxiliary notions. A segment is a context that is nested minor deductions of the rules $(\vee E)$ and $(\exists E)$. Note that $\mathcal{C}[M]E$ reduces to $\mathcal{C}'[ME]$ by permutative conversions. A wrong-application term is an application term that fails to have any type because of its wrong head construction. A wrong-disjunctive term is a term that fails to have any disjunctive type because of its wrong head construction. A wrong-existential term is a term that fails to have any existential type because of its wrong head construction.

$W^\vee$ and $W^\exists$ are introduced to keep the same clauses (6),(7),(9), and (10) as those in other papers such as [5]. Otherwise we should use more complicated clauses because of soundness of $(\vee E)$ and $(\exists E)$.

Definition 3.3. (1) Segments $\mathcal{C}[\cdot] ::= \cdot | M[x.\mathcal{C}[\cdot], y.L] | M[x.N, y.\mathcal{C}[\cdot]] | M[x.\mathcal{C}[\cdot]]$
(2) Wrong-application terms ::=
$(\lambda x.M)[y.N, z.L]\vec{E} | (\lambda x.M)[y.N]\vec{E} | (\lambda x.M)p_0\vec{E} | (\lambda x.M)p_1\vec{E} |$
$\langle 0, M\rangle N\vec{E} | \langle 0, M\rangle[x.N]\vec{E} | \langle 0, M\rangle p_0\vec{E} | \langle 0, M\rangle p_1\vec{E} |$
$\langle 1, M\rangle N\vec{E} | \langle 1, M\rangle[x.N]\vec{E} | \langle 1, M\rangle p_0\vec{E} | \langle 1, M\rangle p_1\vec{E} |$

$\langle\exists, M\rangle N\vec{E}|\langle\exists, M\rangle[x.N, y.L]\vec{E}|\langle\exists, M\rangle p_0\vec{E}|\langle\exists, M\rangle p_1\vec{E}|$
$\langle M, N\rangle L\vec{E}|\langle M, N\rangle[x_0.L_0, x_1.L_1]\vec{E}|\langle M, N\rangle[x.L]\vec{E}$

W is used to denote a wrong-application term.

(2) Wrong-disjunctive terms $::= \lambda x.M|\langle\exists, M\rangle|\langle M, N\rangle|W$

$W^\vee$ is the set of wrong-disjunctive terms.

(3) Wrong-existential terms $::= \lambda x.M|\langle 0, M\rangle|\langle 1, M\rangle|\langle M, N\rangle|W$

$W^\exists$ is the set of wrong-existential terms.

(4) $S_1 \vee S_2 = \{M \in \text{SN}|M \to^* \mathcal{C}[\langle 0, P\rangle]$ implies $P \in S_1$ and $M \to^* \mathcal{C}[\langle 1, P\rangle]$ implies $P \in S_2$ and $M \to^* \mathcal{C}[Q]$ implies $Q \notin W^\vee\}$.

(5) For a function $F : \text{SAT} \to \text{SAT}$, we define
$\exists(F) = \{M \in \text{SN}|M \to^* \mathcal{C}[\langle\exists, P\rangle]$ implies $P \in F(S)$ for some $S \in \text{SAT}$ and $M \to^* \mathcal{C}[Q]$ implies $Q \notin W^\exists\}$.

Lemma 3.4. *(1)* $SN \in \text{SAT}$.
(2) $S_1 \to S_2 \in \text{SAT}$ *if* $S_1, S_2 \in \text{SAT}$.
(3) $S_1 \wedge S_2 \in \text{SAT}$ *if* $S_1, S_2 \in \text{SAT}$.
(4) $S_1 \vee S_2 \in \text{SAT}$ *if* $S_1, S_2 \in \text{SAT}$.
(5) $\exists(F) \in \text{SAT}$ *if* $F : \text{SAT} \to \text{SAT}$.
(6) $\bigcap_{i\in I} S_i \in \text{SAT}$ *if* $S_i \in \text{SAT}$ *for each* $i \in I$.

Proof. (1) We will show SN satisfies sat (1) to (12). We will discuss only non-trivial cases.

sat (3) is proved by induction on $(\text{Length}(\vec{E}), |M| + |N| + |\vec{E}|)$. Assume $(\lambda x.M)N\vec{E} \to L$ and we will show $L \in \text{SN}$. We will consider cases according to L. If $L = (\lambda x.M')N'\vec{E}$, then induction hypothesis can apply, since $|M'| + |N'| < |M| + |N|$. If $L = (\lambda x.M)N\vec{E}'$, then induction hypothesis can apply, since $(\text{Length}(\vec{E}), |\vec{E}|) > (\text{Length}(\vec{E}'), |\vec{E}'|)$. If $L = M[x := N]\vec{E}$, then the assumption implies $L \in \text{SN}$.

sat (5.1) and (5.2) are proved by induction on $(\text{Length}(\vec{E}), |M|+|N_0|+|N_1|+|\vec{E}|)$ similarly to sat (3).

sat (7) is proved by induction on $(\text{Length}(\vec{E}), |\vec{R}| + |N_0| + |N_1| + |E| + |\vec{E}|)$. Assume $x\vec{R}[x_0.N_0, x_1.N_1]E\vec{E} \to L$ and we will show $L \in \text{SN}$. We will consider cases according to L.

If $L = x\vec{R}'[x_0.N_0', x_1.N_1']E'\vec{E}'$ and $\text{Length}(\vec{E}') = \text{Length}(\vec{E})$, then induction hypothesis can apply, since $|\vec{R}'| + |N_0'| + |N_1'| + |E'| + |\vec{E}'| < |\vec{R}| + |N_0| + |N_1| + |E| + |\vec{E}|$.

If $L = x\vec{R}[x_0.N_0, x_1.N_1]E'\vec{E}'$ and $\text{Length}(\vec{E}') < \text{Length}(\vec{E})$, then induction hypothesis can apply, since we have $x\vec{R}[x_0.N_0E', x_1.N_1E']\vec{E}' \in \text{SN}$ from the assumption $x\vec{R}[x_0.N_0E, x_1.N_1E]\vec{E} \in \text{SN}$ and $x\vec{R}[x_0.N_0E, x_1.N_1E]\vec{E} \to^* x\vec{R}[x_0.N_0E', x_1.N_1E']\vec{E}'$.

If $L = x\vec{R}[x_0.N_0E, x_1.N_1E]\vec{E}$, then the assumption gives $L \in \text{SN}$.

sat (8) is proved by induction on $(\text{Length}(\vec{E}), |M| + |N| + |\vec{E}|)$ similarly to sat (3).

sat (10) is proved by induction on $(\text{Length}(\vec{E}), |\vec{R}| + |N| + |E| + |\vec{E}|)$ similarly to sat (7).

sat (11.1) and (11.2) are proved by induction on $(\text{Length}(\vec{E}), |M|+|N|+|\vec{E}|)$ similarly to sat (3).

(2) We will show $S_1 \to S_2$ satisfies sat (1) to (12). We will discuss only non-trivial cases.

sat (1) is proved from S_1 sat (2) and S_2 sat (1).

sat (2) is proved from S_1 sat (1) and S_2 sat (2).

sat (6). Assume $L \in S_1$ and we will show $x\vec{R}[x_0.N_0, x_1.N_1]L \in S_2$. From the assumption $N_0 \in S_1 \to S_2$, we have $N_0 L \in S_2$. Similarly we have $N_1 L \in S_2$. By S_2 sat (6), $x\vec{R}[x_0.N_0 L, x_1.N_1 L] \in S_2$ holds. By S_2 sat (7), we get $x\vec{R}[x_0.N_0, x_1.N_1]L \in S_2$.

sat (9) is proved in a similar manner to sat (6).

(3) The claim is proved similarly to (2).

(4) We will show $S_1 \vee S_2$ satisfies sat (1) to (12). sat (1) trivially holds. To show sat (2) to (12), we have to show the left-hand side M_0 in each clause is in $S_1 \vee S_2$. $M_0 \in \text{SN}$ follows from (1) for each case. So we discuss only (a) $M_0 \to^* \mathcal{C}[\langle 0, P\rangle]$ implies $P \in S_1$ and (b) $M_0 \to^* \mathcal{C}[Q]$ implies $Q \notin W^\vee$ for non-trivial cases.

sat (2). (a) We do not have such P. (b) The only such Q is M_0, which is not in $W^\vee$.

sat (3). (a) Case 1. $\mathcal{C}[\langle 0, P\rangle] = (\lambda x.M')N'\vec{E}'$. We have $M[x := N]\vec{E} \to^* M[x := N]\vec{E}' = \mathcal{C}'[\langle 0, P\rangle]$, so $P \in S_1$.

Case 2. $M_0 \to^* (\lambda x.M')N'\vec{E}' \to M'[x := N']\vec{E}' \to^* \mathcal{C}[\langle 0, P\rangle]$. We have $M[x := N]\vec{E} \to^* \mathcal{C}[\langle 0, P\rangle]$ and hence P is in S_1.

(b) is proved similarly to (a).

sat (4). (a) We have $M' \to^* \mathcal{C}'[\langle 0, P'\rangle]$ where $M_0 \prec M'$ and $P \prec P'$. Hence $P' \in S_1$. By S_1 sat (4), we have $P \in S_1$.

(b) We have $M' \to^* \mathcal{C}'[Q']$ where $M_0 \prec M'$ and $Q \prec Q'$. Hence $Q' \notin W^\vee$. So we have $Q \notin W^\vee$.

sat (5.1). (a) Case 1. $\mathcal{C}[\langle 0, P\rangle] = \langle 0, M'\rangle[x_0.N_0', x_1.N_1']\vec{E}'$ and $\text{Length}(\vec{E}') > 0$. We have $N_1\vec{E} \to^* N_1'\vec{E}' = \mathcal{C}'[\langle 0, P\rangle]$, so P is in S_1.

Case 2. $\mathcal{C}[\langle 0, P\rangle] = \langle 0, M'\rangle[x_0.N_0', x_1.N_1']$.

Case 2.1. $N_0' = \mathcal{C}'[\langle 0, P\rangle]$. By the assumption $N_0[x := M]\vec{E} \in S_1 \vee S_2$ and $S_1 \vee S_2$ sat (4), we have $N_0\vec{E} \in S_1 \vee S_2$. Since $N_0\vec{E} \to^* N_0'$, we get $P \in S_1$.

Case 2.2. $N_1' = \mathcal{C}'[\langle 0, P\rangle]$. By the assumption $N_1\vec{E} \in S_1 \vee S_2$ and $N_1\vec{E} \to^* N_1'$, we get $P \in S_1$.

Case 3. $M_0 \to^* \langle 0, M'\rangle[x_0.N_0', x_1.N_1']\vec{E}' \to N_0'[x_0 := M']\vec{E}' \to^* \mathcal{C}[\langle 0, P\rangle]$. We have $N_0[x_0 := M]\vec{E} \to^* N_0'[x_0 := M']\vec{E}' \to^* \mathcal{C}[\langle 0, P\rangle]$, so P is in S_1.

(b) is proved similarly to (a).

sat (6) is proved similarly to sat (5.1) Case 2.

sat (7). (a) $\mathcal{C}[\langle 0, P\rangle] = x\vec{R}[x_0.N_0', x_1.N_1']\vec{E}'$. In both cases $\text{Length}(\vec{E}) > 0$ and $\text{Length}(\vec{E}) = 0$ we have some $\mathcal{C}'$ such that $x\vec{R}[x_0.N_0 E, x_1.N_1 E]\vec{E} \to^* x\vec{R}[x_0.N_0'', x_1.N_1''] = \mathcal{C}'[\langle 0, P\rangle]$ holds, so P is in S_1.

(b) is proved similarly to (a).

sat (8). (a) Case 1. $\mathcal{C}[\langle 0, P\rangle] = \langle \exists, M'\rangle[x.N']\vec{E}'$ and $\text{Length}(\vec{E}') > 0$. We have $N[x := M]\vec{E} \to^* N'[x := M]\vec{E}' = \mathcal{C}'[\langle 0, P\rangle]$, so P is in S_1.

Case 2. $\mathcal{C}[\langle 0, P\rangle] = \langle \exists, M'\rangle[x.N']$.

By the assumption $N[x := M]\vec{E} \in S_1 \vee S_2$ and $S_1 \vee S_2$ sat (4), we have $N\vec{E} \in S_1 \vee S_2$. Since $N\vec{E} \to^* N' = \mathcal{C}'[\langle 0, P\rangle]$, we get $P \in S_1$.

Case 3. $M_0 \to^* \langle \exists, M'\rangle[x.N']\vec{E}' \to N'[x := M']\vec{E}' \to^* \mathcal{C}[\langle 0, P\rangle]$. We have $N[x := M]\vec{E} \to^* \mathcal{C}[\langle 0, P\rangle]$ and hence P is in S_1.

(b) is proved similarly to (a).

sat (9) and (10) are proved similarly to sat (6) and (7) respectively.

sat (11.1) and (11.2) are proved in a similar way to sat (3).

(5) is proved similarly to (4).

(6) follows immediately from the definition of saturated sets. □

Definition 3.5 (Interpretation of types). A set valuation is a function from type variables to SAT. A set valuation is denoted by σ. For a set valuation σ, a type variable X, and a saturated set S, the set valuation $\sigma[X := S]$ is defined by $(\sigma[X := S])(X) = S$ and $(\sigma[X := S])(Y) = \sigma(Y)$ for $X \neq Y$.

For a type A and a set valuation σ, the set $[\![A]\!]\sigma$ of terms is defined by induction on A as follows.

$[\![X]\!]\sigma = \sigma(X)$.
$[\![A \to B]\!]\sigma = [\![A]\!]\sigma \to [\![B]\!]\sigma$.
$[\![A \vee B]\!]\sigma = [\![A]\!]\sigma \vee [\![B]\!]\sigma$.
$[\![\exists X A]\!]\sigma = \exists(\lambda S.[\![A]\!]\sigma[X := S])$.
$[\![A \wedge B]\!]\sigma = [\![A]\!]\sigma \wedge [\![B]\!]\sigma$.
$[\![\forall X A]\!]\sigma = \bigcap_{S \in \mathrm{SAT}}[\![A]\!]\sigma[X := S]$.

Proposition 3.6. *The set $[\![A]\!]\sigma$ is saturated for every type A and every set valuation σ.*

Proof. This claim is proved by induction on A from Lemma 3.4. □

Lemma 3.7. *Suppose that $S = S_1 \vee S_2$ for some saturated sets S_1 and S_2, or $S = \exists(F)$ for some function $F : \mathrm{SAT} \to \mathrm{SAT}$.*

(1) If $M[x.N, y.L] \in S$, then N and L are in S.

(2) If $M[x.N] \in S$, then N is in S.

Proof. These claims immediately follow from the definition of $S_1 \vee S_2$ and $\exists(F)$. □

Lemma 3.8. *Suppose that S_1, S_2, and S are saturated, $N_0[x_0 := L] \in S$ for all $L \in S_1$, and $N_1[x_1 := L] \in S$ for all $L \in S_2$. If M is in $S_1 \vee S_2$, then $M[x_0.N_0, x_1.N_1]$ is in S.*

Proof. This claim is proved by induction on $(|M|, M)$. We consider cases according to M. As M is not in $W^{\vee}$, M is in the set defined by

$$x\vec{E}|\langle 0, M\rangle|\langle 1, M\rangle|(\lambda x.M)N\vec{E}|\langle 0, M\rangle[x.N, y.L]\vec{E}|\langle 1, M\rangle[x.N, y.L]\vec{E}|$$
$$\langle \exists, M\rangle[x.N]\vec{E}|\langle M, N\rangle p_0\vec{E}|\langle M, N\rangle p_1\vec{E}$$

Case 1. $x\vec{R}$. We get the claim by the assumption $N_0, N_1 \in S$ and S sat (6).

Case 2. $x\vec{R}[y_0.L_0, y_1.L_1]$. By Lemma 3.7 (1), we have $L_0 \in S_1 \vee S_2$. By induction hypothesis for L_0, we have $L_0[x_0.N_0, x_1.N_1] \in S$. Similarly we have

$L_1[x_0.N_0, x_1.N_1] \in S$. By S sat (6), we get $x\vec{R}[y_0.L_0[x_0.N_0, x_1.N_1], y_1.L_1[x_0.N_0, x_1.N_1]] \in S$. By S sat (7), we have the claim.

Case 3. $x\vec{R}[y_0.L_0, y_1.L_1]E\vec{E}$. By $S_1 \vee S_2$ sat (12), we have $M' \in S_1 \vee S_2$ where $M' = x\vec{R}[y_0.L_0E, y_1.L_1E]\vec{E}$. By induction hypothesis for M' with $|M'| < |M|$, we have $M'[x_0.N_0, x_1.N_1] \in S$. By S sat (7), we get the claim.

Case 4. $x\vec{R}[y.L]$. This case is proved in a similar way to Case 2 by using Lemma 3.7 (2).

Case 5. $x\vec{R}[y.L]E\vec{E}$. This case is proved similarly to Case 3.

Case 6. $\langle 0, M_1\rangle$. By the definition of $S_1 \vee S_2$, we have $M_1 \in S_1$. By the assumption, $N_0[x_0 := M_1]$ is in S. From S_2 sat (2), we have $x_1 \in S_2$ and so the assumption implies $N_1 \in S$. By S sat (5.1), we have the claim.

The case $\langle 1, M_1\rangle$ is proved similarly.

Case 7. $(\lambda x.M_1)N\vec{E}$. By $S_1 \vee S_2$ sat (12), M' is in $S_1 \vee S_2$ where $M' = M_1[x := N]\vec{E}$. By induction hypothesis for M' with $|M'| < |M|$, we have $M'[x_0.N_0, x_1.N_1]$. By S sat (3), we have the claim.

Case 8. $\langle 0, M_1\rangle[y_0.L_0, y_1.L_1]\vec{E}$. By $S_1 \vee S_2$ sat (12), M' is in $S_1 \vee S_2$ where $M' = L_0[y_0 := M_1]\vec{E}$. By induction hypothesis for M' with $|M'| < |M|$, we have $M'[x_0.N_0, x_1.N_1]$ is in S.

By $S_1 \vee S_2$ sat (12), $\langle 0, M_1\rangle[y_0.L_0\vec{E}, y_1.L_1\vec{E}]$ is in $S_1 \vee S_2$. By Lemma 3.7 (1), M_3 is in $S_1 \vee S_2$ where $M_3 = L_1\vec{E}$. By induction hypothesis for M_3 with $|M_3| < |M|$, $M_3[x_0.N_0, x_1.N_1]$ is in S. By S sat (5.1), we have the claim.

The case $\langle 1, M_1\rangle[y_0.L_0, y_1.L_1]\vec{E}$ is proved similarly.

Case 9. $\langle \exists, M_1\rangle[x.N]\vec{E}$. By $S_1 \vee S_2$ sat (12), M' is in $S_1 \vee S_2$ where $M' = N[x := M_1]\vec{E}$. By induction hypothesis for M' with $|M'| < |M|$, we have $M'[x_0.N_0, x_1.N_1] \in S$. By S sat (8), we have the claim.

Case 10. $\langle M_1, M_2\rangle p_0\vec{E}$ or $\langle M_1, M_2\rangle p_1\vec{E}$. This case is proved similarly to Case 7. □

Lemma 3.9. *Suppose that $F : \mathrm{SAT} \to \mathrm{SAT}$ is saturated, and $N[x := L] \in S$ for all $S_1 \in \mathrm{SAT}$ and all $L \in F(S_1)$. If M is in $\exists(F)$, then $M[x.N]$ is in S.*

Proof. This claim is proved by induction on $(|M|, M)$ in a similar way to Lemma 3.8. □

A valuation is a function from variables to Λ. We will use ρ to denote a valuation. For a valuation ρ and a term M, the valuation $\rho[x := M]$ is defined in the same way as $\sigma[X := S]$.

For a term M, the substitution $M\rho$ is defined as $M[x_1 := \rho(x_1), \ldots, x_n := \rho(x_n)]$ where free variables of M are among $\{x_1, \ldots, x_n\}$.

Theorem 3.10 (Soundness). *Suppose ρ is a valuation and σ is a set valuation. If $\overrightarrow{x : B} \vdash M : A$ and $\rho(x_i) \in [\![B_i]\!]\sigma$ for $1 \le i \le Length(\overrightarrow{x : B})$, then $M\rho \in [\![A]\!]\sigma$ holds.*

Proof. We will use induction on the proof of $\overrightarrow{x : B} \vdash M : A$. We will consider cases according to the last rule.

Case Assumptions. The claim is proved by the assumption.

Case ($\to I$). Suppose $\Gamma \vdash \lambda x.M : A \to B$ is inferred from $\Gamma, x : A \vdash M : B$. We will show $(\lambda x.M)\rho \in [\![A \to B]\!]\sigma$. Assume $N \in [\![A]\!]\sigma$. By induction hypothesis with $\rho[x := N]$, we have $(M\rho)[x := N] \in [\![B]\!]\sigma$. By $[\![B]\!]\sigma$ sat (3), $(\lambda x.M\rho)N \in [\![B]\!]\sigma$ holds. Hence we get the claim.

Case ($\to E$). The claim is proved by induction hypothesis and the definition of $S_1 \to S_2$.

Case ($\vee I1$). Suppose $\langle 0, M\rangle \to^* \mathcal{C}[\langle 0, P\rangle]$. Then M reduces to P. By induction hypothesis $M \in S_1$ and S_1 sat (12), we have $P \in S_1$. If $\langle 0, M\rangle$ reduces to $\mathcal{C}[Q]$, then Q is $\langle 0, M'\rangle$, so Q is not in $W^{\vee}$.

Case ($\vee I2$) is similarly proved.

Case ($\vee E$). By letting $S_1 := [\![A]\!]\sigma$,$S_2 := [\![B]\!]\sigma$, and $S := [\![C]\!]\sigma$ in Lemma 3.8, we have the claim.

Case ($\exists I$). The claim is proved by induction hypothesis and the definition of $[\![\exists X A]\!]\sigma$ in a similar way to Case ($\vee I1$).

Case ($\exists E$). By letting $F(S) = [\![A]\!]\sigma[X := S]$ and $S := [\![C]\!]\sigma$ in Lemma 3.9, we have the claim.

Case ($\wedge I$). Suppose $\langle M, N\rangle : A \wedge B$ is inferred from $M : A$ and $N : B$. By induction hypothesis, we have $M\rho \in [\![A]\!]\sigma$ and $N\rho \in [\![B]\!]\sigma$. By $[\![A]\!]\sigma$ sat (11.1), $\langle M, N\rangle\rho p_0$ is in $[\![A]\!]\sigma$. By (11.2) we similarly have $\langle M, N\rangle\rho p_1 \in [\![B]\!]\sigma$. By the definition of $S_1 \wedge S_2$, we have the claim.

Cases ($\wedge E1$) and ($\wedge E2$). The claim is proved by induction hypothesis and the definition of $S_1 \wedge S_2$.

Case ($\forall I$). Suppose $M : \forall X A$ is inferred from $M : A$. We will prove $M\rho \in [\![\forall X A]\!]\sigma$. Assume $S \in \mathrm{SAT}$. By induction hypothesis with $\sigma[X := S]$, we have $M\rho \in [\![A]\!]\sigma[X := S]$. Hence we have $M\rho \in \bigcap_{S \in \mathrm{SAT}} [\![A]\!]\sigma[X := S] = [\![\forall X A]\!]\sigma$.

Case ($\forall E$). The claim is proved by induction hypothesis and the definition of $[\![\forall X A]\!]\sigma$. □

Theorem 3.11 (Strong Normalization). *If $\overrightarrow{x : B} \vdash M : A$ is proved in the system NJ_2, then M is strongly normalizing.*

Proof. Let $\rho = \lambda x.x$ and $\sigma(X) = \mathrm{SN}$. By $[\![B_i]\!]\sigma$ sat (2), x_i is in $[\![B_i]\!]\sigma$. By Theorem 3.10, we get $M\rho \in [\![A]\!]\sigma$. By $[\![A]\!]\sigma$ sat (1), we have the claim. □

4 The System $\lambda^{\to\exists\wedge\cap\cup\forall}$ with Second-Order Existential Types, Intersection Types, and Union Types

We will define the system $\lambda^{\to\exists\wedge\cap\cup\forall}$, which has second-order existential types, intersection types, and union types together with permutative conversions for existential types.

The language is the same as that of NJ_2 except it does not have disjunction and instead it has intersection types and union types.

Definition 4.1 (Language)

Type variables $X, Y, Z, \ldots$.

Types $A, B, \ldots ::= X | A \to B | \exists X A | A \wedge B | A \cap B | A \cup B | \forall X A$.

Variables $x, y, z, \ldots$.
Terms $M, N, L, P, Q ::= x|\lambda x.M|MN|\langle \exists, M\rangle|M[x, N]|\langle M, N\rangle|Mp_0|Mp_1$
Λ is the set of terms.

Definition 4.2 (Typing rules). The typing rules are those of NJ_2 except $(\vee I1)$,$(\vee I2)$, and $(\vee E)$ with the following rules.

$$\frac{M : A \quad M : B}{M : A \cap B}\ (\cap I) \qquad \frac{M : A \cap B}{M : A}\ (\cap E1) \qquad \frac{M : A \cap B}{M : B}\ (\cap E2)$$

$$\frac{M : A}{M : A \cup B}\ (\cup I1) \qquad \frac{M : B}{M : A \cup B}\ (\cup I2) \qquad \frac{M : A \cup B \quad \begin{array}{c}[x : A]\\ \vdots\\ N : C\end{array} \quad \begin{array}{c}[x : B]\\ \vdots\\ N : C\end{array}}{N[x := M] \in C}\ (\cup E)$$

Remark. The discussion in the next section such as Theorems 5.6 and 5.7 will hold, even if the system has any type preorder $\leq$ that is set-theoretically valid and the subsumption rule $\dfrac{M : A \quad A \leq B}{M : B}\ (\leq)$

The reduction system is the same as that of NJ_2 except disjunction.

Definition 4.3 (Reduction rules)

β-reductions:

$$\begin{array}{ll}
(\beta\rightarrow) & (\lambda x.M)N \rightarrow M[x := N] \\
(\beta\exists) & \langle \exists, M\rangle[x.N] \rightarrow N[x := M] \\
(\beta \wedge 1) & \langle M, N\rangle p_0 \rightarrow M \\
(\beta \wedge 2) & \langle M, N\rangle p_1 \rightarrow N
\end{array}$$

Eliminators $E ::= M|[x.M]|p_0|p_1$
Permutative conversions:

$$(\pi\exists) \qquad M[x.N]E \rightarrow M[x.NE]$$

The relation $\rightarrow$ is defined as the compatible closure of the relation $\rightarrow$ defined above. The relation $\rightarrow^*$ is the reflexive transitive closure of the relation $\rightarrow$. We say that M reduces to N if $M \rightarrow^* N$.

Remark. Church Rosser property holds. Subject reduction property does not hold because of union types [1].

SN is the set of strongly normalizing terms.

The next section will prove that every term typed in the system $\lambda^{\rightarrow\exists\wedge\cap\cup\forall}$ is strongly normalizing.

5 Strong Normalization for $\lambda^{\rightarrow\exists\wedge\cap\cup\forall}$

We will give saturated-set semantics to $\lambda^{\rightarrow\exists\wedge\cap\cup\forall}$, and prove its strong normalization. To do this, we will use the saturated sets introduced in Section 3. We will

interpret union types by set-theoretic union of these saturated sets, and show its soundness.

Saturated sets are defined by removing the clauses (5.1) to (7) from Definition 3.1. We give their definition here to avoid ambiguity.

Definition 5.1 (Saturated sets)
A set S of terms is saturated if the following hold, where $R ::= M|p_0|p_1$

(1) $S \subseteq \mathrm{SN}$
(2) $x\vec{R} \in S$ if $x\vec{R} \in \mathrm{SN}$
(3) $(\lambda x.M)N\vec{E} \in S$ if $M[x := N]\vec{E} \in S$ and $N \in \mathrm{SN}$
(4) $M \in S$ if $N \succ M$ and $N \in S$
(5) $\langle \exists, M\rangle[x.N]\vec{E} \in S$ if $N[x := M]\vec{E} \in S$ and $M \in \mathrm{SN}$
(6) $x\vec{R}[y.N] \in S$ if $x\vec{R} \in \mathrm{SN}$ and $N \in S$
(7) $x\vec{R}[y.N]E\vec{E} \in S$ if $x\vec{R}[y.NE]\vec{E} \in S$
(8.1) $\langle M, N\rangle p_0\vec{E} \in S$ if $M\vec{E} \in S$ and $N \in \mathrm{SN}$
(8.2) $\langle M, N\rangle p_1\vec{E} \in S$ if $N\vec{E} \in S$ and $M \in \mathrm{SN}$
(9) $M \in S$ if $N \to M$ and $N \in S$

$S_1 \to S_2$, $S_1 \wedge S_2$, and $\exists(F)$ are defined in the same way as in Section 3.

Definition 5.2. (1) Segments $\mathcal{C}[\cdot] ::= \cdot|M[x.\mathcal{C}[\cdot]]$
(2) Wrong-application terms $::= (\lambda x.M)[y.N]\vec{E}|(\lambda x.M)p_0\vec{E}|(\lambda x.M)p_1\vec{E}|$
$\langle \exists, M\rangle N\vec{E}|\langle \exists, M\rangle p_0\vec{E}|\langle \exists, M\rangle p_1\vec{E}|$
$\langle M, N\rangle L\vec{E}|\langle M, N\rangle[x.L]\vec{E}$
W is used to denote a wrong-application term.
(3) Wrong-existential terms $::= \lambda x.M|\langle M, N\rangle|W$
$W^\exists$ is the set of wrong-existential terms.
(4) For a function $F : \mathrm{SAT} \to \mathrm{SAT}$, we define
$\exists(F) = \{M \in \mathrm{SN}|M \to^* \mathcal{C}[\langle \exists, P\rangle]$ implies $P \in F(S)$ for some $S \in \mathrm{SAT}$ and $M \to^* \mathcal{C}[Q]$ implies $Q \notin W^\exists\}$.

The saturated sets satisfy similar properties to those in Section 3.

Lemma 5.3. *(1)* $SN \in \mathrm{SAT}$.
(2) $S_1 \to S_2 \in \mathrm{SAT}$ *if* $S_1, S_2 \in \mathrm{SAT}$.
(3) $S_1 \wedge S_2 \in \mathrm{SAT}$ *if* $S_1, S_2 \in \mathrm{SAT}$.
(4) $\exists(F) \in \mathrm{SAT}$ *if* $F : \mathrm{SAT} \to \mathrm{SAT}$.
(5) $\bigcap_{i\in I} S_i \in \mathrm{SAT}$ *if* $S_i \in \mathrm{SAT}$ *for each* $i \in I$.
(6) $\bigcup_{i\in I} S_i \in \mathrm{SAT}$ *if* $S_i \in \mathrm{SAT}$ *for each* $i \in I$.

Proof. The claims (1) to (5) are proved similarly to Lemma 3.4.

(6) We will show only non-trivial cases.

sat (4). $N \in \bigcup_{i\in I} S_i$ implies $N \in S_i$ for some i. By S_i sat (4), M is in S_i. Hence we get the claim.

sat (5). $N[x := M]\vec{E} \in \bigcup_{i\in I} S_i$ implies $N[x := M]\vec{E} \in S_i$ for some i. By S_i sat (5), $\langle \exists, M\rangle[x.N]\vec{E}$ is in S_i. Hence we get the claim. □

Definition 5.4 (Interpretation of types)
A set valuation is defined in the same way as Section 3.

For a type A and a set valuation σ, the set $[\![A]\!]\sigma$ of terms is defined by induction on A as follows.

$[\![X]\!]\sigma = \sigma(X)$.
$[\![A \to B]\!]\sigma = [\![A]\!]\sigma \to [\![B]\!]\sigma$.
$[\![\exists X A]\!]\sigma = \exists(\lambda S.[\![A]\!]\sigma[X := S])$.
$[\![A \wedge B]\!]\sigma = [\![A]\!]\sigma \wedge [\![B]\!]\sigma$.
$[\![A \cap B]\!]\sigma = [\![A]\!]\sigma \cap [\![B]\!]\sigma$.
$[\![A \cup B]\!]\sigma = [\![A]\!]\sigma \cup [\![B]\!]\sigma$.
$[\![\forall X A]\!]\sigma = \bigcap_{S \in \mathrm{SAT}} [\![A]\!]\sigma[X := S]$.

Proposition 5.5. *The set $[\![A]\!]\sigma$ is saturated for every type A and every set valuation σ.*

Proof. This claim is proved by induction on A from Lemma 5.3. □

The same claims as Lemmas 3.7 and 3.9 hold.

A valuation ρ and the substitution $M\rho$ are defined in the same way as in Section 3.

Theorem 5.6 (Soundness). *Suppose ρ is a valuation and σ is a set valuation. If $\overrightarrow{x:B} \vdash M : A$ and $\rho(x_i) \in [\![B_i]\!]\sigma$ for $1 \le i \le Length(\overrightarrow{x:B})$, then $M\rho \in [\![A]\!]\sigma$ holds.*

Proof. We will use induction on the proof of $\overrightarrow{x:B} \vdash M : A$. We will consider cases according to the last rule. We will discuss only new non-trivial cases we did not have in the proof of Theorem 3.10.

Case $(\cap I)$. Suppose $M : A \cap B$ is inferred from $M : A$ and $M : B$. We have to show $M\rho \in [\![A]\!]\sigma \cap [\![B]\!]\sigma$. By induction hypothesis, we have $M\rho \in [\![A]\!]\sigma$ and $M\rho \in [\![B]\!]\sigma$. Hence we have the claim.

Cases $(\cap E1)$, $(\cap E2)$, $(\cup I1)$, and $(\cup I2)$ are similarly proved to Case $(\cap I)$.

Case $(\cup E)$. Suppose $\Gamma \vdash N[x := M] : C$ is inferred from $\Gamma \vdash M : A \cup B$ and $\Gamma, x : A \vdash N : C$ and $\Gamma, x : B \vdash N : C$. We have to show $N[x := M]\rho \in [\![C]\!]\sigma$. By induction hypothesis, we have $M\rho \in [\![A \cup B]\!]\sigma$.

Case 1. $M\rho \in [\![A]\!]\sigma$. By induction hypothesis for the second assumption of $(\cup E)$ with $\rho[x := M\rho]$, we have $N\rho[x := M\rho] \in [\![C]\!]\sigma$. Therefore we have the claim.

Case 2. $M\rho \in [\![B]\!]\sigma$. The claim is proved in a similar way to Case 1. □

The next theorem is derived from the previous theorem in the same way as in Section 3.

Theorem 5.7 (Strong Normalization). *If $\overrightarrow{x:B} \vdash M : A$ is proved in the system $\lambda^{\to\exists\wedge\cap\cup\forall}$, then M is strongly normalizing.*

Remark. The concluding remarks in [10] states that if their system does not have disjunction, their saturated sets are closed under union. Therefore, if we add second-order existential quantification, intersection types, and union types to their system without disjunction, a similar strong normalization theorem immediately follows.

6 Concluding Remarks

[5] and [7] proposed different saturated sets for disjunction from our saturated sets. In our notation, they are as follows:

$$\begin{aligned}
S_1 \vee^{intro} S_2 = \bigcap\{&S \in \mathrm{SAT} | \langle 0, M\rangle \in S \text{ for all } M \in S_1 \text{ and} \\
&\langle 1, M\rangle \in S \text{ for all } M \in S_2\} \\
S_1 \vee^{elim} S_2 = \{&M \in \mathrm{SN} | \forall S \in \mathrm{SAT}.\forall N_0 N_1 ((\forall L \in S_1.N_0[x_0 := L] \in S) \rightarrow \\
&(\forall L \in S_2.N_1[x_1 := L] \in S) \rightarrow M[x_0.N_0, x_1.N_1] \in S)\}
\end{aligned}$$

Roughly speaking, [5] and [7] gave $S_1 \vee^{intro} S_2$ and $S_1 \vee^{elim} S_2$, respectively. Those saturated sets work also in our setting. In general, in order to prove soundness theorem of saturated-set semantics and strong normalization theorem, we could choose any saturated set S that satisfies $S_1 \vee^{intro} S_2 \subseteq S \subseteq S_1 \vee^{elim} S_2$. For our saturated sets, we have $S_1 \vee^{intro} S_2 = S_1 \vee S_2 = S_1 \vee^{elim} S_2$.

However, our definition of $S_1 \vee S_2$ is simpler, because it is defined by Π^0_1-comprehension and on the other hand the definition of $S_1 \vee^{intro} S_2$ and $S_1 \vee^{elim} S_2$ uses Π^1_1-comprehension. Moreover, the definition of $S_1 \vee S_2$ is predicative since it does not mention the set of saturated sets, and on the other hand, the definitions of $S_1 \vee^{intro} S_2$ and $S_1 \vee^{elim} S_2$ are impredicative because they refer to the set SAT of saturated sets. This simplification comes from the idea of segments by Prawitz, which catches the essence of permutative conversions.

For saturated sets for second-order existential quantification, our definition of saturated sets is simpler, too. According to ideas in [5] and [7], we can define the following, respectively:

$$\begin{aligned}
&[\![\exists X A]\!]^{intro}\sigma = \bigcap\{S \in \mathrm{SAT} | (\exists S_1 \in \mathrm{SAT}.M \in [\![A]\!]\sigma[X := S_1]) \rightarrow \langle \exists, M\rangle \in S\} \\
&[\![\exists X A]\!]^{elim}\sigma = \{M \in \mathrm{SN} | \forall S \in \mathrm{SAT}.\forall N(\\
&\quad (\forall S_1 \in SAT.\forall L \in [\![A]\!]\sigma[X := S_1].N[x := L] \in S) \rightarrow M[x.N] \in S)\}.
\end{aligned}$$

In general, in order to prove soundness theorem of saturated-set semantics and strong normalization theorem, we could choose any saturated set S that satisfies $[\![\exists X A]\!]^{intro}\sigma \subseteq S \subseteq [\![\exists X A]\!]^{elim}\sigma$. For our saturated sets, we have $[\![\exists X A]\!]^{intro}\sigma = [\![\exists X A]\!]\sigma = [\![\exists X A]\!]^{elim}\sigma$. Our definition of $[\![\exists X A]\!]\sigma$ is simpler, since it can be defined by Σ^1_1-comprehension. On the other hand the definitions of $[\![\exists X A]\!]^{intro}\sigma$ and $[\![\exists X A]\!]^{elim}\sigma$ use Π^1_2-comprehension. This simplification also comes from the idea of segments by Prawitz.

Standard saturated sets with these new ones for existential types were not closed under union. Instead of the condition (5) in Definition 5.1, they would have the following condition:

$$(5') \qquad \langle \exists, M\rangle[x.N]\vec{E} \in S \quad \text{if} \quad N[x := M]\vec{E} \in S \text{ and } N\vec{E} \in S \text{ and } M \in \mathrm{SN}$$

In order to show $S_1 \cup S_2$ is saturated, we would have to show

$$\langle \exists, M\rangle[x.N]\vec{E} \in S_1 \cup S_2 \quad \text{if} \quad N[x := M]\vec{E} \in S_1 \text{ and } N\vec{E} \in S_2 \text{ and } M \in \mathrm{SN},$$

but it is not the case. This paper can simply the standard condition (5') to (5) by introducing the contraction property (4) in Definition 5.1.

Remark that the sets of the shape $\mathcal{X} \to \mathrm{SN}$ used in [7] are not closed under union.

Finding saturated sets with both disjunction and union types would be an interesting question. Our saturated sets will not work with both disjunction and union types because there are some saturated sets S_1 and S_2 such that

$$x\vec{R}[x_0.N_0, x_1, N_1] \notin S_1 \cup S_2 \text{ and } x\vec{R} \in \mathrm{SN} \text{ and } N_0 \in S_1 \text{ and } N_1 \in S_2$$

and $S_1 \cup S_2$ does not satisfy sat (6) in Definition 3.1.

Future work will be applying this technique of saturated sets developed in this paper to other systems to give them saturated-set semantics and prove their strong normalization.

Acknowledgments

We would like to thank Prof. Takeshi Yamazaki for discussion on set comprehension. We would also like to thank the anonymous referees for valuable comments. We would also like to thank participants of NII logic seminar for discussions.

References

1. Barbanera, F., Dezani-Ciancaglini, M.: Intersection and Union Types: Syntax and Semantics. Information and Computation 119, 202–230 (1995)
2. David, R., Nour, K.: A short proof of the strong normalization of classical natural deduction with disjunction. Journal of Symbolic Logic 68(4), 1277–1288 (2003)
3. de Groote, P.: On the strong normalisation of intuitionistic natural deduction with permutation-conversions. Information and Computation 178, 441–464 (2002)
4. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Archive for Mathematical Logic 42, 59–87 (2003)
5. Matthes, R.: Non-strictly positive fixed-points for classical natural deduction. Annals of Pure and Applied Logic 133(1–3), 205–230 (2005)
6. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems 10(3), 470–502 (1988)
7. Nour, K., Saber, K.: A semantical proof of the strong normalization theorem for full propositional classical natural deduction. Archive for Mathematical Logic 45(3), 357–364 (2006)
8. Prawitz, D.: Natural Deduction. Almquist and Wiksell (1965)
9. Prawitz, D.: Ideas and results of proof theory. In: Fenstad, J.E. (ed.) Proceedings of the Second Scandinavian Logic Logic Symposium, pp. 235–307. North-Holland, Amsterdam (1971)
10. Tatsuta, M., Mints, G.: A simple proof of second-order strong normalization with permutative conversions. Annals of Pure and Applied Logic 136(1–2), 134–155 (2005)
11. Tatsuta, M.: Second order permutative conversions with Prawitz's strong validity. Progress in Informatics 2, 41–56 (2005)
12. Tatsuta, M., Dezani-Ciancaglini, M.: Normalisation is Insensible to lambda-term Identity or Difference. In: Proceedings of Twenty First Annual IEEE Symposium on Logic in Computer Science, pp. 327–336 (2006)

Convolution $\bar{\lambda}\mu$-Calculus

Lionel Vaux

Institut de Mathématiques de Luminy, Marseille, France
vaux@iml.univ-mrs.fr
http://iml.univ-mrs.fr/~vaux

Abstract. We define an extension of Herbelin's $\bar{\lambda}\mu$-calculus, introducing a product operation on contexts (in the sense of lists of arguments, or stacks in environment machines), similar to the convolution product of distributions. This is the computational couterpart of some new semantical constructions, extending models of Ehrhard-Regnier's differential interaction nets, along the lines of Laurent's polarization of linear logic. We demonstrate this correspondence by providing this calculus with a denotational semantics inside a lambda-model in the category of sets and relations.

1 Introduction

Herbelin's $\bar{\lambda}\mu$-calculus [Her95] transposes the Curry-Howard correspondence between classical natural deduction and $\lambda\mu$-calculus to the setting of classical sequent calculus LK (in fact one of its deterministic versions: LKT [DJS95]). In particular, the notion of application, corresponding to the *modus ponens* of natural deduction, is replaced with the notion of cut between a term and a context. More precisely, $\bar{\lambda}\mu$-calculus involves three syntactic categories — terms, contexts and commands — given by the following grammar:

$$\begin{aligned} s, t &::= x \mid \lambda x\, s \mid \mu\alpha\, c && \text{(terms)} \\ e, f &::= \alpha \mid s \cdot e && \text{(contexts)} \\ c &::= \langle s\,, e\rangle && \text{(commands)}\ . \end{aligned}$$

Reduction is defined by the following two basic rules:

$$\langle \lambda x\, s\,, t\cdot e\rangle \to \langle s\,[t/x]\,, e\rangle \quad \text{and} \quad \langle \mu\alpha\, c\,, e\rangle \to c\,[e/\alpha]\ .$$

In the present paper, we introduce an extension of $\bar{\lambda}\mu$-calculus, featuring a binary operation $*$ on contexts (and the corresponding context unit $\mathbf{1}$), which bears similarities with the convolution product of distributions. For that purpose, we further endow the set of terms with a structure of commutative monoid, with addition $+$ and neutral $\mathbf{0}$, and give the following three reduction rules:

$$\begin{aligned} \langle \lambda x\, s\,, (t\cdot e) * f\rangle &\to \langle \lambda x\, \mu\alpha\, \langle s\,[t+x/x]\,, e*\alpha\rangle\,, f\rangle \\ \langle \lambda x\, s\,, \mathbf{1}\rangle &\to \langle s\,[\mathbf{0}/x]\,, \mathbf{1}\rangle \\ \langle \mu\alpha\, c\,, e\rangle &\to c\,[e/\alpha]\ . \end{aligned}$$

The reader may check that the reduction rules of usual $\bar{\lambda}\mu$-calculus can be simulated in this new setting.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 381–395, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Outline of the Paper. In the remaining of this introduction, we briefly review notions and ideas that led to the definition of convolution $\bar{\lambda}\mu$-calculus: it is the pure calculus associated with an extension of Ehrhard-Regnier's differential nets [ER05], along the lines of Laurent's polarization of linear logic proof nets [Lau02]. In section 2 we introduce the objects of convolution $\bar{\lambda}\mu$-calculus, and the associated notion of reduction, for which we prove the Church-Rosser property. In section 3 we define a reflexive object $\mathcal{D}$ in the category of sets and relations, following [BE04]. Then we introduce a type-system, in which types are elements of $\mathcal{D}$, along the lines of Carvalho's system R [dC06]. We conclude by proving that terms identified by reduction have exactly the same types.

1.1 Classical Logic and Co-structural Rules

Denotational semantics of linear logic gives rise to models of simply typed λ-calculus, through the Curry-Howard isomorphism and well known translations from intuitionistic logic into linear logic. This relationship may be explicited in the syntax by encodings of typed λ-calculus into linear logic proof nets, in which β-reduction of λ-terms is accurately simulated by cut-elimination in proof nets. In fact, one may also establish such a correspondence in an untyped setting: pure λ-terms are succesfully encoded into the weakly typed nets of [Reg92].

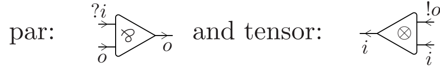
An analoguous relationship may be observed in a setting related with classical logic rather than intuitionistic logic. In [Lau02], Olivier Laurent introduced polarized linear logic and polarized proof nets. Polarized linear logic is linear logic where all formulas are polarized, and weakening and contraction are allowed on every negative formula; also, promotion is allowed on any sequent formed only of negative formulas. In [Lau03], Laurent showed how to encode Parigot's $\lambda\mu$-calculus [Par92] into polarized proof nets. Since $\lambda\mu$-calculus lifts the Curry-Howard correspondence from intuitionistic logic to classical logic, this encoding is the counterpart of a translation from classical natural deduction into polarized linear logic. A thorough semantical investigation on the nature of this translation may be found in [LR03].

It also happens that original models of linear logic lead to novel extensions of λ-calculus, using the Curry-Howard correspondence as a tool to draw semantical properties back into the syntax. In [Ehr01] and [Ehr05], Ehrhard introduced models of linear logic in which formulas are interpreted by particular vector spaces, and proofs by linear maps between these spaces. Moreover, morphisms with type $!A \multimap B$ correspond to analytic functions between A and B. This not only provided a semantics of typed λ-calculus in which λ-terms are interpreted by smooth functions between vector spaces in a very natural way, but also led to the introduction of differential λ-calculus by Ehrhard and Regnier in [ER03].

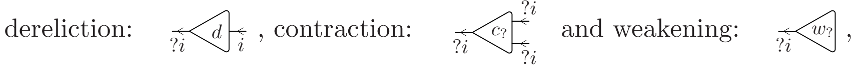
Differential Nets. Here we briefly outline some features of the differential interaction nets from [ER05], which may be considered as a syntactic presentation of the models from [Ehr01] and [Ehr05]. For the sake of simplicity, we use a weak typing scheme: using usual linear logic connectors and modalities, introduce type o such that $o = !o \multimap o$; we will use types o, $i = o^{\perp}$, $!o$ and $?i = (!o)^{\perp}$, to type

the wires of interaction nets. We do not consider criteria for well-formedness of nets; we only focus on local, weakly typed reduction rules.

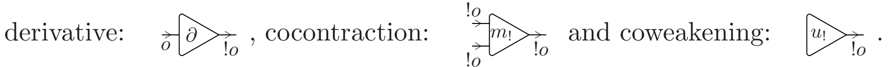
Differential interaction nets extend the interaction nets for multiplicative exponential linear logic from [Laf95] as follows: besides multiplicative cells



and structural cells

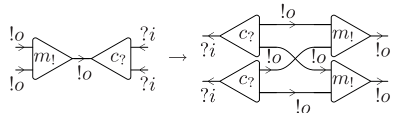


come costructural cells

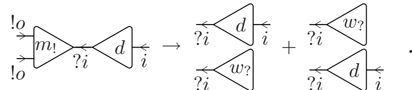


Cells $m_!$ and $u_!$ have the same geometry as tensor and tensor unit respectively: $m_!$ connects two nets together, and $u_!$ is a net by itself.

Recall that in the formalism of interaction nets, typing depends on the orientation of wires: if a wire has type A in one orientation, it has type $A^\perp$ in the reverse orientation. Also, recall that each cell has exactly one principal port (this we put on the point of our triangular cells) together with any number of auxiliary ports. A redex consists of two cells connected by their respective principal ports, in accordance with typing. Among new reduction rules introduced in differential nets, interaction between cells $c_?$ and $w_?$ on the one hand, and $m_!$ and $u_!$ on the other hand, endow exponential types with a structure of bialgebra, mainly characterized by the following interaction rule:



Also, dereliction d interacts with $m_!$ as follows:

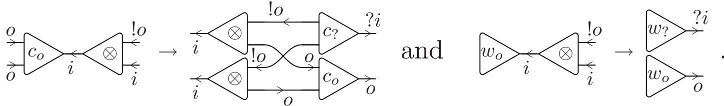


The idea is that d requires one copy of an argument from its principal port, which it feeds to its auxiliary port; this argument is taken nondeterministically from either auxiliary port of $m_!$, hence the sum. The redex between d and $w_!$ reduces to the 0 net (which is the neutral element of sum of nets) following the same intuition. There are of course symmetric rules for interaction between derivative ∂, and $c_?$ and $w_?$, that we do not explicit here.

An extensive discussion of the intuitions behind differential reduction rules may be found in [ER05] and the relationship between sum and nondeterminism is developped in the introduction of [ER03]. Although this is not done in [ER05], one may also introduce promotion boxes in differential interaction nets, along the lines of [Laf95], and provide appropriate reduction rules: this allows to encode differential λ-calculus.

Polarized Nets. Polarized nets are another extension of linear logic nets. Again, we only outline a weakly typed version of the polarized proof nets of [Lau03], which we present in an interaction net flavor. The main feature of polarized nets is that contraction and weakening are generalized to type o:

This accounts for structural rules on output types, which is a characteristic of classical logic. Generalized structural rules give rise to a new redex between tensor and contraction c_o (or weakening w_o) cells:



Polarized proofs nets are well suited to encode classical extensions of λ-calculus: see [Lau03] for an encoding of $\lambda\mu$-calculus and [Lau02, Section 12.2] for an encoding of both deterministic variants of $\bar{\lambda}\mu\tilde{\mu}$-calculus.

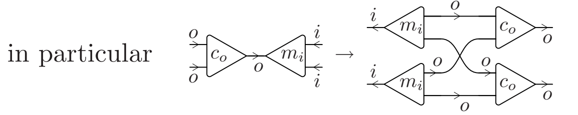
Differential Structures and Classical Logic. In [Vau07a], the author introduced the differential $\lambda\mu$-calculus, as an attempt to uncover possible interactions between differential structures and classical logic, in a purely computational setting. The result is a pure calculus which consistently extends both differential λ-calculus and $\lambda\mu$-calculus.

Another possible path for studying how differential and classical constructs interact with each other lies at the level of interaction nets. One may come up with a notion of differential polarized nets, with cells those of differential nets and polarized nets altogether. Then it is easily checked that the union of the reduction rules for differential nets and for polarized nets address all possible redexes.

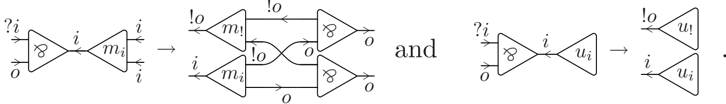
We do not detail further this construction, but one may verify that differential $\lambda\mu$-calculus enjoys a natural encoding into these differential polarized nets. Hence, although differential $\lambda\mu$-calculus introduces a new reduction rule, which was not present in $\lambda\mu$-calculus nor in differential λ-calculus (namely that associated with the derivative of a μ-abstraction), one may claim that it is only a side-effect of the sequentiality of λ-calculus. Indeed, in the more parallel syntax of interaction nets, differential cells on one hand, and generalized structural rules on the other hand, do not interact with each other.

A Convolution Product on Contexts. The convolution $\bar{\lambda}\mu$-calculus defined in this paper, is the pure calculus associated with the following other variant of polarized nets: together with the cells of polarized nets, introduce the abovementioned costructural cells $m_!$ and $u_!$, and also generalized versions of these,

i.e. cocontraction and coweakening on input type i. New redexes arise and we give the following reduction rules: m_i and u_i interact with c_o and w_o the same way as $m_!$ and $u_!$ interact with $c_?$ and $w_?$,

in particular  ;

also, m_i duplicates $⅋$ and u_i erases $⅋$, the same way as c_o and w_o act on $\otimes$:



Again, we do not detail this system further: this is still the subject of ongoing work in [Vau07b]. Rather, we explicit how convolution $\bar{\lambda}\mu$-calculus stems from it. In the translation of $\lambda\mu$-calculus into polarized nets, the type of dangling wires (oriented outwards) is only o or $?i$. Type i only occurs in the cuts involved in the translation of application. This suggests that the computational counterpart of costructural cells on type i may be more fruitfully studied in the setting of Herbelin's $\bar{\lambda}\mu$-calculus, where cuts appear explicitly.

One may derive a translation of $\bar{\lambda}\mu$-calculus into polarized nets from that of $\bar{\lambda}\mu\tilde{\mu}_T$-calculus given in [Lau02], in which i types the active wire of the translation of *contexts*. The counterpart of m_i and u_i is then an associative and commutative operation on contexts that we denote by $*$, and its unit $\mathbf{1}$. It is argued in [Ehr01, Section 5.4], that $m_!$ acts as a convolution product on $!o$, with properties similar to those of convolution of distributions; in that analogy, $u_!$ corresponds to the Dirac mass at 0 (see also [Ehr05, Section 3], in paragraph *Algebraic structure*). Since the behaviour of m_i and u_i on type i mimics that of $m_!$ and $u_!$ on type $!o$, we may call $*$ *convolution product on contexts*. We will show later that, although they live in different formalisms, $*$ actually shares a distinctive feature with the convolution product of distributions, which further enforces this designation.

From the reduction rules of nets we have outlined, one derives that β-reduction may be generalized so that:

$$\langle \lambda x\, s\,, (t \cdot e) * (t' \cdot e') \rangle \to \langle s\,[t + t'/x]\,, e * e' \rangle \quad .$$

Recall that nondeterministic choice provides a possible computational interpretation of sum, as described in the introduction of [ER03]. Convolution product of contexts may then be interpreted as a nondeterministic intertwining of lists of arguments.

The reduction step given above, however, only amounts to usual reduction in $\bar{\lambda}\mu$-calculus, together with the identity $(t \cdot e) * (t' \cdot e') = (t + t') \cdot (e * e')$ which is semantically valid (see Remark 3). Moreover, it involves synchronisation between contexts: both must have a term at top-level, for a reduction to occur. In this paper, we use a finer grained notion of reduction, the basic rules of which were given in the beginning of this introduction: this matches cut elimination between $⅋$ and costructural cells more closely. Also, that notion will enable us to demonstrate the link with convolution of distributions in Remark 2.

Relational Semantics. In order to underline the correspondence between convolution $\bar{\lambda}\mu$-calculus and generalized costructural rules on i, we provide it with a denotational semantics in the category of sets and relations. In the standard relational model of linear logic, formulas are interpreted as sets and proofs as relations between these sets: see, e.g., [Ehr05, Appendix A] for precise definitions. In particular, structural rules on type A correspond to relations $d_A \subseteq\, !A \multimap A = \mathcal{M}_{\text{fin}}(A) \times A$, $c_A \subseteq\, !A \multimap (!A \otimes !A) = \mathcal{M}_{\text{fin}}(A) \times (\mathcal{M}_{\text{fin}}(A) \times \mathcal{M}_{\text{fin}}(A))$ and $w_A \subseteq\, !A \multimap 1 = \mathcal{M}_{\text{fin}}(A) \times 1$, where $\mathcal{M}_{\text{fin}}(A)$ denotes the set of all finite multisets of elements of A and 1 is a singleton set. One interesting feature of the relational model is that it identifies dual connectors in linear logic. In particular, if $\varphi \subseteq A \multimap B$ in the relational model, then $\varphi^\perp \subseteq B \multimap A$ where $\varphi^\perp = \{(b,a);\ (a,b) \in \varphi\}$. By setting $\partial_A = d_A^\perp$, $m_A = c_A^\perp$, and $u_A = w_A^\perp$, it turns out that we obtain a model of differential interaction nets. This suggests that we may derive a denotational semantics of convolution $\bar{\lambda}\mu$-calculus from a model of $\lambda\mu$-calculus in the category of sets and relations.

The model we use was introduced by Bucciarelli and Ehrhard in [BE04], as an extensional lambda-model in the category of sets and relations. It is naturally endowed with a monoid structure, which is suitable to provide a denotational semantics of $\lambda\mu$-calculus along the lines of [LR03]: monoid operation and unit model structural rules on o. We also use those monoid laws to handle the denotational semantics of convolution product: the same as for costructural rules, this amounts to reverse the direction of the corresponding relation.

1.2 Related Work

A system of intersection and union types for the $\bar{\lambda}\mu$-calculus is presented [DGL05]. This system bears some similarity with the type system we present in section 3: this is underlined by the fact that all weakly normalizing terms are typable. It comes as no surprise, since our system is derived from Carvalho's system R, which is related to a system of intersection types for λ-calculus.

One important outcome of [dC06] and our paper is that they provide the aforementioned type systems with a strong grounding into well known denotational semantics of linear logic and its variants.

2 Syntax

In this section, we introduce the syntax of convolution $\bar{\lambda}\mu$-calculus. Like ordinary $\bar{\lambda}\mu$-calculus, it involves three distinct syntactic categories: terms, contexts and commands. We introduce convolution product $*$ as a commutative and associative binary operation on contexts, with unit $\mathbf{1}$.

Similarly to what is done in [ER03] and [Vau07a], each category of objects is endowed with a structure of commutative monoid, and we denote by $+$ and $\mathbf{0}$ the corresponding operation and neutral element. Moreover, all but one syntactic construct are linear, *i.e.* they commute to sums: in particular, $*$ distributes over $+$. In order to implement these high-level, metatheoretical requirements, we first define a basic syntax with a simple equality, then provide extended notations.

Remark 1. Since we only form sums (without coefficients) it is quite clear that our constructions are well defined, and that nothing tricky is hiding behind equality of terms. Recall from [Vau06, Section 4] that the introduction of linear combinations of terms (rather than just sums) may break normalization properties, or even trivialize β-equality [Vau06, Section 2.6], depending on the structure of the set of coefficients.

2.1 Morphology

Basic Syntax. Fix two denumerably infinite sets $\mathfrak{V}$ (set of variables, denoted by x, y, z) and $\mathfrak{N}$ (set of names, denoted by α, β, γ).

Definition 1. *Define sets $\mathcal{T}$ of simple terms and $\mathcal{T}^+$ of terms, set $\mathcal{S}$ of stacks, sets $\mathcal{E}$ of simple contexts and $\mathcal{E}^+$ of contexts, and sets $\mathcal{C}$ of simple commands and $\mathcal{C}^+$ of commands, by the following grammar:*

$$
\begin{array}{rll}
s & ::= x \mid \lambda x\, s \mid \mu\alpha\, c & \textit{(simple terms)} \\
\sigma & ::= \alpha \mid S \cdot e & \textit{(stacks)} \\
e & ::= \mathbf{1} \mid \sigma * e & \textit{(simple contexts)} \\
c & ::= \langle s\,, e \rangle & \textit{(simple commands)} \\
S & ::= \mathbf{0} \mid s + S & \textit{(terms)} \\
E & ::= \mathbf{0} \mid e + E & \textit{(contexts)} \\
C & ::= \mathbf{0} \mid c + C & \textit{(commands)}\ .
\end{array}
$$

We consider terms, commands and contexts up to permutativity of sum in the sense that, e.g., $s+(s'+S) = s'+(s+S)$. Also, we consider simple contexts up to permutativity of convolution product: e.g., $\alpha * ((S \cdot e) * e') = (S \cdot e) * (\alpha * e')$. Notice that these identities preserve free and bound variables and names: hence they are compatible with α-conversion. Equality of terms (resp. commands, contexts) is then given by permutativity of sum and product, together with α-equivalence.

Notations. We call simple object any simple term, simple context or simple command, and object any term, context or command. We may use greek letter θ to denote a simple object and capital Θ to denote an object. In general, if simple object θ and object Θ appear in the same sentence, it should be clear they are of the same kind: Θ is a term, context or command, if θ is a simple term, a simple context or a simple command respectively.

If $\theta_1, \ldots, \theta_n$ are simple objects and Θ an object, all of the same kind, we write $\theta_1 + \cdots + \theta_n + \Theta$ for $\theta_1 + (\cdots + (\theta_n + \Theta) \cdots)$. If θ is a simple object, we may also denote by θ the corresponding object $\theta + \mathbf{0}$. Hence, all object Θ may be written $\Theta = \theta_1 + \cdots + \theta_n$ or even $\Theta = \sum_{i=1}^{n} \theta_i$. Assume $\Theta = \theta_1 + \cdots + \theta_n$ and $\Theta' = \theta'_1 + \cdots + \theta'_p$: we write $\Theta + \Theta'$ for $\theta_1 + \cdots + \theta_n + \theta'_1 + \cdots + \theta'_p$. Up to these conventions, sum $+$ becomes an associative and commutative binary operation on terms, contexts and commands respectively, and object $\mathbf{0}$ is neutral.

Similarly, we identify any stack σ with the simple context $\sigma * \mathbf{1} \in \mathcal{E}$: then we may write any simple context e as $e = \sigma_1 * \cdots * \sigma_n$ where the stacks σ_i are names or of shape $S \cdot e'$. With notations similar to those we used for sum, we consider

$*$ as an associative and commutative binary operation on simple contexts, with unit $\mathbf{1}$.

Now we extend our syntactic constructs by linearity in order to be able to write $\lambda x\, S$, $\mu\alpha\, C$, $S \cdot E$, $E * F$ and $\langle S\,, E\rangle$ for all $S \in \mathcal{T}^+$, $E, F \in \mathcal{E}^+$ and $C \in \mathcal{C}^+$.

Definition 2. *Assume $s_1, \ldots, s_n \in \mathcal{T}$, $e_1, \ldots, e_p, f_1, \ldots, f_q \in \mathcal{E}$, $c_1, \ldots, c_r \in \mathcal{C}$ and $S \in \mathcal{T}^+$. Then we write:*

$$\begin{array}{ll} \lambda x\, (\sum_{i=1}^n s_i) = \sum_{i=1}^n \lambda x\, s_i & (\sum_{j=1}^p e_j) * (\sum_{k=1}^q f_k) = \sum_{j=1}^p \sum_{k=1}^q e_j * f_k \\ \mu\alpha\, (\sum_{l=1}^r c_l) = \sum_{l=1}^r \mu\alpha\, c_l & \left\langle \sum_{i=1}^n s_i\,, \sum_{j=1}^p e_j \right\rangle = \sum_{i=1}^n \sum_{j=1}^p \langle s_i\,, e_j\rangle \\ S \cdot (\sum_{j=1}^p e_j) = \sum_{j=1}^p S \cdot e_j\ . & \end{array}$$

Notice that the cons of a term and a context is *not* linear in the term: this is the analogue of application not being linear in the argument, in usual λ-calculus. This definition introduces some overlap of notations: e.g., $\lambda x\, s$ denotes both a simple term in our basic syntax, and the value of $\lambda x\, (s + \mathbf{0})$ in the above definition. This is however harmless since both writings denote the same term.

Up to the notations we have just introduced, the set of terms (resp. contexts, commands) is endowed with a structure of commutative monoid. The set of contexts is moreover endowed with a structure of commutative rig (*i.e.* a commutative ring, without the condition that every element admits an opposite), with addition $+$ and multiplication $*$. Also, λ- and μ-abstractions are linear, cons is linear in the context, and cut is bilinear. Then substitution of a term for a variable (resp. of a context for a name) in an object is defined as usual, by induction on objects:

$$\begin{array}{rlrl} y\,[T/x] &= \begin{cases} T \text{ if } x = y \\ y \text{ otherwise} \end{cases} & y\,[E/\alpha] &= y \\ (\lambda y\, s)\,[T/x] &= \lambda y\, (s\,[T/x]) & (\lambda y\, s)\,[E/\alpha] &= \lambda y\, (s\,[E/\alpha]) \\ (\mu\beta\, c)\,[T/x] &= \mu\beta\, (c\,[T/x]) & (\mu\beta\, c)\,[E/\alpha] &= \mu\beta\, (c\,[E/\alpha]) \\ \beta\,[T/x] &= \beta & \beta\,[E/\alpha] &= \begin{cases} E \text{ if } \alpha = \beta \\ \beta \text{ otherwise} \end{cases} \\ (S \cdot e)\,[T/x] &= (S\,[T/x]) \cdot (e\,[T/x]) & (S \cdot e)\,[E/\alpha] &= (S\,[E/\alpha]) \cdot (e\,[E/\alpha]) \\ 1\,[T/x] &= 1 & 1\,[E/\alpha] &= 1 \\ (\sigma * e)\,[T/x] &= (\sigma\,[T/x]) * (e\,[T/x]) & (\sigma * e)\,[E/\alpha] &= (\sigma\,[E/\alpha]) * (e\,[E/\alpha]) \\ \langle s\,, e\rangle\,[T/x] &= \langle s\,[T/x]\,, e\,[T/x]\rangle & \langle s\,, e\rangle\,[E/\alpha] &= \langle s\,[E/\alpha]\,, e\,[E/\alpha]\rangle \\ \mathbf{0}\,[T/x] &= \mathbf{0} & \mathbf{0}\,[E/\alpha] &= \mathbf{0} \\ \theta + \Theta\,[T/x] &= \theta\,[T/x] + \Theta\,[T/x] & \theta + \Theta\,[E/\alpha] &= \theta\,[E/\alpha] + \Theta\,[E/\alpha] \end{array}$$

assuming usual conditions to avoid variable and name capture.

2.2 Reduction

Convolution Reduction. We call *simply contextual relation* any triplet r of binary relations respectively on terms, contexts and commands, all denoted r, and such that:

- if $s \mathrel{\mathrm{r}} S'$ then $\lambda x\, s \mathrel{\mathrm{r}} \lambda x\, S'$ and $\langle s\,, e\rangle \mathrel{\mathrm{r}} \langle S'\,, e\rangle$;
- if $e \mathrel{\mathrm{r}} E'$ then $s \cdot e \mathrel{\mathrm{r}} s \cdot E'$, $\langle s\,, e\rangle \mathrel{\mathrm{r}} \langle s\,, E'\rangle$ and $e * f \mathrel{\mathrm{r}} E' * f$;
- if $c \mathrel{\mathrm{r}} C'$ then $\mu\alpha\, c \mathrel{\mathrm{r}} \mu\alpha\, C'$;
- if $S \mathrel{\mathrm{r}} S'$ then $S \cdot e \mathrel{\mathrm{r}} S' \cdot e$;
- and if $\theta_0 \mathrel{\mathrm{r}} \Theta'_0$ then $\theta_0 + \Theta_1 \mathrel{\mathrm{r}} \Theta'_0 + \Theta_1$.

Definition 3. *Reduction $\to_\beta$ is the least simply contextual relation such that:*

$$\langle \mu\alpha\, c\,, e\rangle \to_\beta c\,[e/\alpha] \tag{1}$$

$$\langle \lambda x\, s\,, (S \cdot e) * f\rangle \to_\beta \langle \lambda y\, \mu\alpha\, \langle s\,[y + S/x]\,, \alpha * e\rangle\,, f\rangle \tag{2}$$

$$\langle \lambda x\, s\,, \mathbf{1}\rangle \to_\beta \langle s\,[\mathbf{0}/x]\,, \mathbf{1}\rangle \tag{3}$$

with y a fresh variable and α a fresh name in (2).

Notice that $\langle \lambda x\, s\,, S \cdot e\rangle \to^*_\beta \langle s\,[S/x]\,, e\rangle$ and, more interestingly,

$$\langle \lambda x\, s\,, (S \cdot e) * (S' \cdot e')\rangle \to^*_\beta \langle s\,[S + S'/x]\,, e * e'\rangle$$

where $\to^*_\beta$ denotes the reflexive and transitive closure of $\to_\beta$. This enlightens the fact that $\to_\beta$ is a refined version of both usual reduction of $\bar{\lambda}\mu$-calculus and the coarser notion of convolution reduction we first derived from cut elimination in the introduction. Conversely, $\to_\beta$ may be simulated by that coarse reduction, up-to the following generalization of η-expansion on commands: recalling that the analogue of η-expansion in $\bar{\lambda}\mu$-calculus is $s \leftarrow_\eta \lambda x\, \mu\alpha\, \langle s\,, x \cdot \alpha\rangle$ we set

$$\langle s\,, e * e'\rangle \leftarrow_{\eta'} \langle \lambda x\, \mu\alpha\, \langle s\,, e * (x \cdot \alpha)\rangle\,, e'\rangle \quad .$$

This can be thought of as η-expansion w.r.t. only one component of a product. If e' actually holds an argument at top-level, *i.e.* $e' = S \cdot f$, we can get back:

$$\langle \lambda x\, \mu\alpha\, \langle s\,, e * (x \cdot \alpha)\rangle\,, S \cdot f\rangle \to^*_\beta \langle s\,, e * (S \cdot f)\rangle = \langle s\,, e * e'\rangle$$

which validates $\leftarrow_{\eta'}$ as a notion of η-expansion.

Remark 2. Recall (e.g., from [Sch66]) that the definition of the convolution product of distributions is as follows: if e and f are distributions with compact domains and φ is a test function, then $e * f$ is such that

$$\langle \lambda z\, \varphi(z)\,, e * f\rangle = \langle \lambda y\, \langle \lambda x\, \varphi(x + y)\,, e\rangle\,, f\rangle \quad .$$

Analoguously, one can check that the following two commands

$$\langle \lambda z\, s\,, (S \cdot e) * (T \cdot f)\rangle \;\text{ and }\; \langle \lambda y\, \mu\beta \langle \lambda x\, \mu\alpha \langle s\,[x + y/z]\,, \alpha * \beta\rangle\,, S \cdot e\rangle\,, T \cdot f\rangle$$

are identified by reduction: both reduce to $\langle s\,[S + T/z]\,, e * f\rangle$. The apparent complexity of that last identity has two main causes.

First, in the formalism of distributions and test functions, φ is supposed to be a function with scalar values. The type corresponding to scalars is that of

commands, but in $\bar{\lambda}\mu$-calculus, as in λ-calculus, functions *and* values are represented by terms. Hence the μ-abstractions and the innermost cut: these handle the possible remaining arguments. Second, functions are in general considered extensionally. Expansion $\leftarrow_{\eta'}$ may be used to introduce sufficient extensionality:

$$\begin{aligned}\langle \lambda z\, s\,, e * f\rangle &\leftarrow_{\eta'} \langle \lambda y\, \mu\beta \,\langle \lambda z\, s\,, e * (y\cdot\beta)\rangle\,, f\rangle \\ &\rightarrow_\beta \langle \lambda y\, \mu\beta\langle \lambda x\, \mu\alpha\langle s\,[x+y/z]\,, \alpha * \beta\rangle\,, e\rangle\,, f\rangle \quad .\end{aligned}$$

Confluence. We prove confluence of reduction using usual Tait-Martin-Löf technique: introduce a parallel extension of one-step reduction, and prove this has the diamond property.

A binary relation r on commutative monoid $\mathcal{A}$ is said to be *linear* if: for all $a_1,\ldots, a_n, b_1,\ldots,b_n \in \mathcal{A}$, if $a_i \mathrel{\mathrm{r}} b_i$ holds for all i, then $\sum_{i=1}^n a_i \mathrel{\mathrm{r}} \sum_{i=1}^n b_i$ also holds (in particular $0 \mathrel{\mathrm{r}} 0$). Notice that $\rightarrow_\beta$ is not linear: $\mathbf{0} \not\rightarrow_\beta \mathbf{0}$. We call *contextual relation* any triplet r of binary relations respectively on terms, contexts and commands, all denoted r, such that each of them is reflexive and linear, and if $S \mathrel{\mathrm{r}} S'$, $E \mathrel{\mathrm{r}} E'$, $F \mathrel{\mathrm{r}} F'$ and $C \mathrel{\mathrm{r}} C'$, then $\lambda x\, S \mathrel{\mathrm{r}} \lambda x\, S'$, $\mu\alpha\, C \mathrel{\mathrm{r}} \mu\alpha\, C'$, $S\cdot E \mathrel{\mathrm{r}} S'\cdot E'$, $E*F \mathrel{\mathrm{r}} E'*F'$ and $\langle S\,, E\rangle \mathrel{\mathrm{r}} \langle S'\,, E'\rangle$.

Definition 4. *Parallel reduction $\rightarrow_{/\!/}$ is the least contextual relation $\rightarrow_{/\!/}$ such that, if $s \rightarrow_{/\!/} S'$, $c \rightarrow_{/\!/} C'$, $e \rightarrow_{/\!/} E'$, and for all $i = 1,\ldots,n$, $S_i \rightarrow_{/\!/} S'_i$ and $e_i \rightarrow_{/\!/} E'_i$, then:*

$$\langle \mu\alpha\, c\,, e\rangle \rightarrow_{/\!/} C'\,[E'/\alpha] \tag{4}$$

$$\langle \lambda x\, s\,, e * \textstyle\prod_{i=0}^n (S_i\cdot e_i)\rangle \rightarrow_{/\!/} \langle \lambda y\, \mu\alpha\,\langle S'\,[y+\sum_{i=0}^n S'_i/x]\,, \alpha * \prod_{i=0}^n E'_i\rangle\,, E'\rangle \tag{5}$$

$$\langle \lambda x\, s\,, \textstyle\prod_{i=1}^n (S_i\cdot e_i)\rangle \rightarrow_{/\!/} \langle s\,[\sum_{i=1}^n S'_i/x]\,, \prod_{i=1}^n E'_i\rangle\,. \tag{6}$$

with y a fresh variable α a fresh name in (5).

It should be clear that $\rightarrow_\beta \subset \rightarrow_{/\!/}$, in the sense that if $\Theta \rightarrow_\beta \Theta'$ then $\Theta \rightarrow_{/\!/} \Theta'$. In particular, (6) is reminiscent of the coarse version of reduction. Moreover, $\rightarrow_{/\!/} \subset \rightarrow_\beta^*$ by simple contextuality of $\rightarrow_\beta$. The following lemma states the essential property of parallel reduction.

Lemma 1. *If Θ and Θ' are objects, S and $S' \in \mathcal{T}^+$, and E and $E' \in \mathcal{E}^+$, such that $\Theta \rightarrow_{/\!/} \Theta'$, $S \rightarrow_{/\!/} S'$ and $E \rightarrow_{/\!/} E'$, then for every variable x and every name α, the following reductions hold:*

$$\Theta\,[S/x] \rightarrow_{/\!/} \Theta'\,[S'/x] \quad \textit{and} \quad \Theta\,[E/\alpha] \rightarrow_{/\!/} \Theta'\,[E'/\alpha] \quad .$$

Proof. This is a simple induction on Θ, using contextuality of $\rightarrow_{/\!/}$.

We now prove that $\rightarrow_{/\!/}$ enjoys the diamond property. Assume S is a term and E is a stack, and write $E = (\prod_{i=1}^n S_i\cdot e_i) * (\prod_{j=1}^k \alpha_j)$; we then define

$$\langle \lambda x\, S\,, E\rangle_0 = \begin{cases} \langle s\,[\sum_{i=1}^n S_i/x]\,, \prod_{i=1}^n e_i\rangle & \text{if } k=0; \\ \left\langle \lambda y\, \mu\alpha\,\langle S\,[y+\sum_{i=1}^n S_i/x]\,, \alpha * \prod_{i=1}^n e_i\rangle\,, \prod_{j=1}^k \alpha_j\right\rangle & \text{otherwise.}\end{cases}$$

Clearly, $\langle \lambda x\, S\,, E\rangle \rightarrow_{/\!/} \langle \lambda x\, S\,, E\rangle_0$, as a particular case of (5) or (6).

Definition 5. *We define full reduction as follows:*

$$\begin{array}{rclrclrcl} x\!\downarrow &=& x & \alpha\!\downarrow &=& \alpha & \langle x\,,e\rangle\downarrow &=& \langle x\,,e\!\downarrow\rangle \\ (\lambda x\, s)\!\downarrow &=& \lambda x\, s\!\downarrow & (S\cdot e)\!\downarrow &=& S\!\downarrow\cdot e\!\downarrow & \langle \lambda x\, s\,,e\rangle\downarrow &=& \langle \lambda x\, s\!\downarrow,e\!\downarrow\rangle_0 \\ (\mu\alpha\, c)\!\downarrow &=& \mu\alpha\, c\!\downarrow & \mathbf{1}\!\downarrow &=& \mathbf{1} & \langle \mu\alpha\, c\,,e\rangle\downarrow &=& c\!\downarrow[e\!\downarrow/\alpha] \\ & & & (\sigma * e)\!\downarrow &=& \sigma\!\downarrow * e\!\downarrow & & & \end{array}$$

and $(\sum_{i=1}^{n}\theta_i)\!\downarrow = \sum_{i=1}^{n}\theta_i\!\downarrow$.

Full reduction fires all possible redexes in an object. Then one obtains the diamond property for parallel reduction:

Lemma 2. *If Θ and Θ' are objects such that $\Theta \rightarrow_{/\!/} \Theta'$, then $\Theta' \rightarrow_{/\!/} \Theta\!\downarrow$.*

Proof. This result is proved by inspecting all possible cases of reduction $\Theta \rightarrow_{/\!/} \Theta'$, using Lemma 1 in redex cases.

Theorem 1. *Reduction is confluent.*

Proof. This is a corollary of Lemma 2 and the inclusions $\rightarrow_\beta \subset \rightarrow_{/\!/} \subset \rightarrow_\beta^*$.

3 Relational Semantics

In this section, we adapt the system R of [dC06] to the setting of convolution $\overline{\lambda}\mu$-calculus: we introduce a type system, the types of which are elements of the extensional λ-model described in [BE04].

A Reflexive Object in the Category of Sets and Relations. If X is a set, we denote by $\mathcal{M}_{\mathrm{fin}}(X)$ the set of all finite multisets $[x_1,\ldots,x_n]$ of elements $x_1,\ldots,x_n \in X$ (possibly with repetitions). Also, we write $(\mathcal{M}_{\mathrm{fin}}(X))^{(\omega)}$ for the set of all infinite sequences $a = (a(i))_{i\in\omega}$ of finite multisets of elements of X such that $a(i) = []$ holds for almost all $i \in \omega$.

We define an increasing family $(\mathcal{D}_n)_{n\in\mathbf{N}}$ of sets by: $\mathcal{D}_0 = \emptyset$ and $\mathcal{D}_{n+1} = (\mathcal{M}_{\mathrm{fin}}(\mathcal{D}_n))^{(\omega)}$. Then we write $\mathcal{D} = \bigcup_{n\in\mathbf{N}}\mathcal{D}_n$. If $A \in \mathcal{M}_{\mathrm{fin}}(\mathcal{D})$ and $a \in \mathcal{D}$, we write $A :: a$ for the sequence b such that $b(0) = A$ and $b(i+1) = a(i)$ for all $i \in \omega$. This mapping is clearly a bijection between $\mathcal{D}$ and $\mathcal{M}_{\mathrm{fin}}(\mathcal{D}) \times \mathcal{D}$. We write ι for the sequence in which only the empty multiset occurs: $\iota(i) = []$ for all $i \in \omega$, so that $\iota = [] :: \iota$. Observe that $\mathcal{D}_1 = \{\iota\}$.

Type System. Call types the elements of $\mathcal{D}$. We impose a commutative monoid structure on types as follows. For all $a, b \in \mathcal{D}$, we define $a \star b$ as the sequence such that, for all $i \in \omega$, $(a \star b)(i) = a(i) + b(i)$ where $+$ denotes the union of multisets. Clearly ι is neutral for that associative and commutative operation.

A variable environment is a function $\Gamma : \mathfrak{V} \longrightarrow \mathcal{M}_{\mathrm{fin}}(\mathcal{D})$ such that $\Gamma(x) = []$ for almost all $x \in \mathfrak{V}$. If $x \in \mathfrak{V}$ and $A \in \mathcal{M}_{\mathrm{fin}}(\mathcal{D})$, we write $x : A$ for the variable environment Γ such that: $\Gamma(x) = A$ and, for all $y \neq x$, $\Gamma(y) = []$. If Γ and Γ' are variable environments, we write $\Gamma + \Gamma'$ for the variable environment defined

$$\frac{}{x:[a]\vdash x:a\mid}\ \textsf{Var} \qquad \frac{\Gamma+x:A\vdash s:a\mid\Delta \quad \Gamma(x)=[]}{\Gamma\vdash\lambda x\, s:A::a\mid\Delta}\ \textsf{Abs}$$

$$\frac{c:(\Gamma\vdash\Delta\star\alpha:a)\quad \Delta(\alpha)=\iota}{\Gamma\vdash\mu\alpha\, c:a\mid\Delta}\ \textsf{Mu} \qquad \frac{}{\mid\alpha:a\vdash\alpha:a}\ \textsf{Name}$$

$$\frac{\Gamma_0\mid e:a_0\vdash\Delta_0 \quad \Gamma_1\vdash S:a_1\mid\Delta_1 \quad \cdots \quad \Gamma_n\vdash S:a_n\mid\Delta_n}{\Gamma_0+\cdots+\Gamma_n\mid S\cdot e:[a_1,\ldots,a_n]::a_0\vdash\Delta_0\star\cdots\star\Delta_n}\ \textsf{Cons}$$

$$\frac{}{\mid\mathbf{1}:\iota\vdash}\ \textsf{Unit} \qquad \frac{\Gamma\mid\sigma:a\vdash\Delta \quad \Gamma'\mid e:a'\vdash\Delta'}{\Gamma+\Gamma'\mid\sigma*e:a\star a'\vdash\Delta\star\Delta'}\ \textsf{Conv}$$

$$\frac{\Gamma\vdash s:a\mid\Delta \quad \Gamma'\mid e:a\vdash\Delta'}{\langle s\,,e\rangle:(\Gamma+\Gamma'\vdash\Delta\star\Delta')}\ \textsf{Cut} \qquad \frac{\Gamma\vdash\theta_i:a\mid\Delta}{\Gamma\vdash\sum_{i=0}^{n}\theta_i:a\mid\Delta}\ \textsf{Sum}_i$$

Fig. 1. Typing rules for system $R_{\bar{\lambda}\mu*}$

by $(\Gamma+\Gamma')(x)=\Gamma(x)+\Gamma'(x)$. If Γ is a variable environment, we define its support $\mathrm{Supp}\,(\Gamma)=\{x\in\mathfrak{V};\Gamma(x)\neq[]\}$.

Similarly, a name environment is a function $\Delta:\mathfrak{N}\longrightarrow\mathcal{D}$ such that $\Delta(\alpha)=\iota$ for almost all $\alpha\in\mathfrak{N}$. If $\alpha\in\mathfrak{N}$ and $a\in\mathcal{D}$, we write $\alpha:a$ for the name environment Δ such that: $\Delta(\alpha)=a$ and, for all $\beta\neq\alpha$, $\Delta(\beta)=\iota$. If Δ and Δ' are name environments, we write $\Delta\star\Delta'$ for the name environment defined by $(\Delta\star\Delta')(\alpha)=\Delta(\alpha)\star\Delta'(\alpha)$ and we set $\mathrm{Supp}\,(\Delta)=\{\alpha\in\mathfrak{N};\Delta(\alpha)\neq\iota\}$.

Now we introduce type system $R_{\bar{\lambda}\mu*}$ for the objects of convolution $\bar{\lambda}\mu$-calculus. Typing judgements are of form $\Gamma\vdash S:a\mid\Delta$, $\Gamma\mid E:a\vdash\Delta$ or $C:(\Gamma\vdash\Delta)$, where Γ is a variable environment and Δ is a name environment. We may omit Γ (resp. Δ) if it is the constant function with value $[]$ (resp. ι). The rules of system $R_{\bar{\lambda}\mu*}$ are given in Fig. 1.

The reader may refer to [DGL05] and check that the rules of system $R_{\bar{\lambda}\mu*}$, restricted to the objects of ordinary $\bar{\lambda}\mu$-calculus, are quite similar to those of system $\mathcal{M}^{\cap\cup}$. This similarity actually extends to the fact that all weakly normalizing objects are typable in system $R_{\bar{\lambda}\mu*}$, as we will show later. This feature is a characteristic of intersection type systems: this was already prominent in system R.

Example 1. The term $\lambda x\,\mu\alpha\,\langle x\,,x\cdot\alpha\rangle$ (the $\bar{\lambda}\mu$-calculus variant of $\delta=\lambda x\,(x)\,x$) has the following typing derivation, recalling that $\iota=[]::\iota$:

$$\frac{\dfrac{\dfrac{\dfrac{}{x:[\iota]\vdash x:\iota\mid} \quad \dfrac{\dfrac{}{\mid\alpha:\iota\vdash}}{\mid x\cdot\alpha:\iota\vdash}}{\langle x\,,x\cdot\alpha\rangle:(x:[\iota]\vdash)}}{x:[\iota]\vdash\mu\alpha\,\langle x\,,x\cdot\alpha\rangle:\iota\mid}}{\vdash\lambda x\,\mu\alpha\,\langle x\,,x\cdot\alpha\rangle:[\iota]::\iota\mid}\ .$$

Lemma 3. *Every term, context or command which is in normal form is typable.*

Proof. The result is proved by mutual induction on normal terms, contexts and commands. Among the typing rules in figure 1, only $\textsf{Cut}$ involves some compatibility condition on the types of subobjects. Hence the only interesting induction

case is that of simple commands. Simple commands in normal form are those $c = \langle s\,, e\rangle$ such that:

(i) either s is a variable and e is a simple context in normal form;
(ii) or $s = \lambda x\, t$, where t is a simple term in normal form, and $e = \alpha_0 * \cdots * \alpha_n$ is a product of names, with $n > 0$.

In both cases, it is easy to build a typing derivation using the inductive hypothesis and axiom rules (Var or Name).

Denote by $\mathrm{FV}\,(\Theta)$ (resp. $\mathrm{FN}\,(\Theta)$) the set of all variables (resp. names) free in Θ. To simplify some of our next statements, we write $R_{\bar{\lambda}\mu *}\,(\Gamma, \Delta, \Theta, a)$ for:

$$\begin{array}{rl} \Gamma \vdash S : a \mid \Delta & \text{if} \;\; \Theta = S \in \mathcal{T}^+ \;\; ; \\ \Gamma \mid E : a \vdash \Delta & \text{if} \;\; \Theta = E \in \mathcal{E}^+ \;\; ; \\ C : (\Gamma \vdash \Delta) & \text{if} \;\; \Theta = C \in \mathcal{C}^+ \;\; . \end{array}$$

Lemma 4. *If* $R_{\bar{\lambda}\mu *}\,(\Gamma, \Delta, \Theta, a)$*, then* $\mathrm{Supp}\,(\Gamma) \subseteq \mathrm{FV}\,(\Theta)$ *and* $\mathrm{Supp}\,(\Delta) \subseteq \mathrm{FN}\,(\Theta)$*.*

Proof. This is easily proved by induction on Θ.

Denotational Semantics. We define the relational semantics of an object, as the set of all its typings in system $R_{\bar{\lambda}\mu *}$. More precisely:

Definition 6. *Assume* $S \in \mathcal{T}^+$*,* $E \in \mathcal{E}^+$ *and* $C \in \mathcal{C}^+$ *are such that* $\mathrm{FV}\,(\Theta) \subseteq \{x_1, \ldots, x_n\}$ *and* $\mathrm{FN}\,(\Theta) \subseteq \{\alpha_1, \ldots, \alpha_p\}$*, for* $\Theta = S, E, C$*. We define*

$$\begin{aligned} [\![S]\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} &= \{(\Gamma(x_1), \ldots, \Gamma(x_n), a, \Delta(\alpha_1), \ldots, \Delta(\alpha_p));\; \Gamma \vdash S : a \mid \Delta\} \;\; , \\ [\![E]\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} &= \{(\Gamma(x_1), \ldots, \Gamma(x_n), a, \Delta(\alpha_1), \ldots, \Delta(\alpha_p));\; \Gamma \mid E : a \vdash \Delta\} \;\; \textit{and} \\ [\![C]\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} &= \{(\Gamma(x_1), \ldots, \Gamma(x_n), \Delta(\alpha_1), \ldots, \Delta(\alpha_p));\; C : (\Gamma \vdash \Delta)\} \;\; . \end{aligned}$$

Remark 3. The reader can easily check that

$$[\![(S \cdot e) * (S' \cdot e')]\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} = [\![(S + T) \cdot (e * e')]\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} \;\; .$$

The following three lemmas are proved by induction on objects.

Lemma 5. *We have* $R_{\bar{\lambda}\mu *}\,(\Gamma, \Delta, \Theta\,[\mathbf{0}/x]\,, a)$ *if and only if* $x \notin \mathrm{Supp}\,(\Gamma)$ *and* $R_{\bar{\lambda}\mu *}\,(\Gamma, \Delta, \Theta, a)$*.*

Lemma 6. *Assume* $x \notin \mathrm{FV}\,(T)$*. Then the following are equivalent:*

- $R_{\bar{\lambda}\mu *}\,(\Gamma, \Delta, \Theta\,[x + T/x]\,, a)$*;*
- *there exist variable environments* $\Gamma', \Gamma_1, \ldots, \Gamma_n$*, and name environments* Δ'*,* $\Delta_1, \ldots, \Delta_n$ *and types* $a_1, \ldots, a_n \in \mathcal{D}$ *such that*
  - $\Gamma = \Gamma' + \Gamma_1 + \cdots + \Gamma_n$ *and* $\Delta = \Delta' \star \Delta_1 \star \cdots \star \Delta_n$*;*
  - *for all* $i = 1, \ldots, n$*,* $\Gamma_i \vdash S : a_i \mid \Delta_i$*;*
  - *and* $R_{\bar{\lambda}\mu *}\,(\Gamma' + x : [a_1, \ldots, a_n]\,, \Delta', \Theta, a)$*.*

Lemma 7. *The following statements are equivalent:*

- $R_{\bar{\lambda}\mu*}(\Gamma, \Delta, \Theta[E/\alpha], a)$;
- *there exist variable environments* Γ' *and* Γ'', *name environments* Δ' *and* Δ'', *and type* $b \in \mathcal{D}$, *such that*
  - $\alpha \notin \mathrm{Supp}(\Delta')$;
  - $\Gamma = \Gamma' + \Gamma''$ *and* $\Delta = \Delta' \star \Delta''$;
  - $\Gamma'' \mid E : b \vdash \Delta''$;
  - *and* $R_{\bar{\lambda}\mu*}(\Gamma', \Delta' \star \alpha : b, \Theta, a)$.

Theorem 2. *If* $\Theta \to_\beta \Theta'$, *then we have:* $R_{\bar{\lambda}\mu*}(\Gamma, \Delta, \Theta, a)$ *iff* $R_{\bar{\lambda}\mu*}(\Gamma, \Delta, \Theta', a)$. *If moreover* $\mathrm{FV}(\Theta) \subseteq \{x_1, \ldots, x_n\}$ *and* $\mathrm{FN}(\Theta) \subseteq \{\alpha_1, \ldots, \alpha_p\}$, *then*

$$[\![\Theta]\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} = [\![\Theta']\!]^{\alpha_1,\ldots,\alpha_p}_{x_1,\ldots,x_n} \quad .$$

Proof. The proof is by induction on Θ, inspecting all possible cases for reduction $\Theta \to_\beta \Theta'$, and using the previous three lemmas in redex cases.

Hence the relational semantics is preserved by reduction. As a corollary, Lemma 3 implies that every object that has a normal form is typable.

4 Future Work

On Pure Calculi. Although grounded in ideas coming from models of differential λ-calculus, convolution $\bar{\lambda}\mu$-calculus provides no differentiation primitive. Indeed, recall from our introduction that the nets associated with convolution $\bar{\lambda}\mu$-calculus are polarized nets, extended with cocontraction and coweakening on types $!o$ and i. In particular, they do not involve derivative ∂.

One may augment these nets by including ∂ and the associated cut elimination rules, but this needs caution. Uncontrolled use of ∂ breaks one essential property of polarized nets: namely, the occurrence of at most one positive type ($!o$ or i in our setting) among all output wires. That matter is discussed in [Vau07b] in more details.

This remark, however, does not hamper the fact that one may propose differential extensions of convolution $\bar{\lambda}\mu$-calculus. Some first attempts even suggest that the introduction of convolution product of contexts actually simplifies the presentation of a would-be differential $\bar{\lambda}\mu$-calculus.

On Denotational Semantics. In [dC06], Carvalho provides precise results relating the relational semantics of λ-terms with their normalization properties (which are very close to those we expect from an intersection type system); he also provides bounds for the execution time of terms in variants of Krivine's abstract machine, according to the size of their typing derivations in system R. We do not know yet, to which extent these results may accomodate themselves to the setting of Bucciarelli-Ehrhard's model and convolution $\bar{\lambda}\mu$-calculus (or even usual $\lambda\mu$- or $\bar{\lambda}\mu$-calculus for that matter).

Another promising direction for further research is to study more precisely how the categorical constructions of [LR03] may be extended to a setting with costructural rules.

References

[BE04] Bucciarelli, A., Ehrhard, T.: An extensional model of the lambda-calculus in the category of sets and relations. Manuscript (2004)

[dC06] de Carvalho, D.: Execution time of lambda-terms via non uniform semantics and intersection types (2006) Revised version available at http://iml.univ-mrs.fr/~carvalho/Pub/computation.pdf

[DGL05] Dougherty, D.J., Ghilezan, S., Lescanne, P.: Intersection and union types in the lambda-bar-mu-mu-tilde-calculus. Electr. Notes Theor. Comput. Sci. 136, 153–172 (2005)

[DJS95] Danos, V., Joinet, J.-B., Schellinx, H.: Sequent calculi for second order logic. In: Girard, J.-Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic, pp. 211–224. Cambridge University Press, Cambridge (1995)

[Ehr01] Ehrhard, T.: On Köthe sequence spaces and linear logic. Mathematical Structures in Computer Science 12, 579–623 (2001)

[Ehr05] Ehrhard, T.: Finiteness spaces. Mathematical. Structures in Comp. Sci. 15(4), 615–646 (2005)

[ER03] Ehrhard, T., Regnier, L.: The differential lambda-calculus. Theoretical Computer Science 309, 1–41 (2003)

[ER05] Ehrhard, T., Regnier, L.: Differential interaction nets. Electr. Notes Theor. Comput. Sci. 123, 35–74 (2005)

[Her95] Herbelin, H.: Séquents qu'on calcule. Phd thesis, Université Paris 7 (1995)

[Laf95] Lafont, Y.: From proof nets to interaction nets. In: Girard, J.-Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic, pp. 225–247. Cambridge University Press, Cambridge (1995)

[Lau02] Laurent, O.: Etude de la polarisation en logique. Thèse de doctorat, Université Aix-Marseille II (March 2002)

[Lau03] Laurent, O.: Polarized proof-nets and $\lambda\mu$-calculus. Theoretical Computer Science 290(1), 161–188 (2003)

[LR03] Laurent, O., Regnier, L.: About translations of classical logic into polarized linear logic. In: Proceedings of the 18th annual IEEE symposium on Logic In Comp. Sci., pp. 11–20. IEEE Computer Society Press, Los Alamitos (2003)

[Par92] Parigot, M.: $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)

[Reg92] Regnier, L.: Lambda-calcul et réseaux. PhD thesis, Université Paris 7 (1992)

[Sch66] Schwartz, L.: Théorie des distributions. Hermann (1966)

[Vau06] Vaux, L.: λ-calculus in an algebraic setting (2006) Research report, available at http://iml.univ-mrs.fr/~vaux/articles/alglam.ps.gz

[Vau07a] Vaux, L.: The differential $\lambda\mu$-calculus. Theoretical Computer Science (To appear, 2007) doi:10.1016/j.tcs.2007.02.028

[Vau07b] Vaux, L.: Polarized proof nets and differential structures. Unpublished manuscript (2007)

Author Index

Abel, Andreas 8

Baillot, Patrick 2
Berardi, Stefano 23
Blum, William 39
Boulmé, Sylvain 54
Bove, Ana 70

Capretta, Venanzio 70
Cousineau, Denis 102

David, René 84
Dowek, Gilles 102

Espírito Santo, José 118, 133

Faggian, Claudia 148
Fiore, Marcelo P. 163

Ghani, Neil 207

Intrigila, Benedetto 178

Jiang, Ying 194
Johann, Patricia 207

Kiselyov, Oleg 223
Kuśmierek, Dariusz 240

Lindley, Sam 255
Lipton, James 272

Marion, Jean-Yves 290
Matthes, Ralph 133
Mazza, Damiano 305
Mostrous, Dimitris 321

Nakazawa, Koji 336
Nieva, Susana 272
Nour, Karim 84

Ong, C.-H. Luke 39

Pfenning, Frank 1
Piccolo, Mauro 148
Pinto, Luís 133

Shan, Chung-chieh 223
Shkaravska, Olha 351
Statman, Richard 178

Tatsuta, Makoto 366

van Eekelen, Marko 351
van Kesteren, Ron 351
Vaux, Lionel 381

Yoshida, Nobuko 321

Zhang, Guo-Qiang 194