EPTCS 185

Proceedings of the

Tenth International Workshop on

Logical Frameworks and Meta Languages: Theory and Practice

Berlin, Germany, 1 August 2015

Edited by: Iliano Cervesato and Kaustuv Chaudhuri

Published: 27th July 2015
DOI: 10.4204/EPTCS.185
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume constitutes the proceedings of LFMTP 2015, the *Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, held on August 1st, 2015 in Berlin, Germany. The workshop was a one-day satellite event of CADE-25, the 25th International Conference on Automated Deduction.

The program committee selected seven papers for presentation at LFMTP 2015, and inclusion in this volume. In addition, the program included invited talks by Frank Pfenning (Carnegie Mellon University), Vivek Nigam (Federal University of Paraíba) and Marc Lasson (Inria).

Logical frameworks and meta-languages form a common substrate for representing, implementing, and reasoning about a wide variety of deductive systems of interest in logic and computer science. Their design and implementation and their use in reasoning tasks ranging from the correctness of software to the properties of formal computational systems have been the focus of considerable research over the last two decades. This workshop brought together designers, implementors, and practitioners to discuss various aspects impinging on the structure and utility of logical frameworks, including the treatment of variable binding, inductive and co-inductive reasoning techniques and the expressiveness and lucidity of the reasoning process.

Many people helped make LFMTP 2015 a success. We wish to thank the organizers of CADE-25 for their support. We are indebted to the program committee members and the external referees for their careful and efficient work in the reviewing process. Finally we are grateful to the authors, the invited speakers and the attendees who made this workshop an enjoyable and fruitful event.

July, 2015

Iliano Cervesato
Kaustuv Chaudhuri

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 1–2, doi:10.4204/EPTCS.185.0

Program Committee of LINEARITY 2014

- Andreas Abel (Chalmers and Gothenburg University)
- David Baelde (LSV, ENS Cachan)
- Iliano Cervesato (Carnegie Mellon University — co-chair)
- Kaustuv Chaudhuri (Inria & LIX/École polytechnique — co-chair)
- Assia Mahboubi (Inria)
- Stefan Monnier (University of Montreal)
- Gopalan Nadathur (University of Minnesota)
- Giselle Reis (Inria)
- Claudio Sacerdoti Coen (University of Bologna)
- Carsten Schürmann (IT University of Copenhagen & Demtech)

Additional Reviewers

Andrew Gacek and Mary Southern.

Gluing together Proof Environments: Canonical extensions of LF Type Theories featuring *Locks*[*]

Furio Honsell
Department of Mathematics and Computer Science
University of Udine, Italy
furio.honsell@uniud.it

Luigi Liquori
Inria Sophia Antipolis Méditerranée, France
luigi.liquori@inria.fr

Petar Maksimović
Inria Rennes Bretagne Atlantique, France
Mathematical Institute of the Serbian Academy of Sciences and Arts, Serbia
petar.maksimovic@inria.fr

Ivan Scagnetto
Department of Mathematics and Computer Science
University of Udine, Italy
ivan.scagnetto@uniud.it

We present two extensions of the LF Constructive Type Theory featuring monadic *locks*. A lock is a monadic type construct that captures the effect of an *external call to an oracle*. Such calls are the basic tool for *gluing together* diverse Type Theories and proof development environments. The oracle can be invoked either to check that a constraint holds or to provide a suitable witness. The systems are presented in the *canonical style* developed by the CMU School. The first system, $\mathsf{CLLF}_{\mathscr{P}}$, is the canonical version of the system $\mathsf{LLF}_{\mathscr{P}}$, presented earlier by the authors. The second system, $\mathsf{CLLF}_{\mathscr{P}?}$, features the possibility of invoking the oracle to obtain a witness satisfying a given constraint. We discuss encodings of Fitch-Prawitz Set theory, call-by-value λ-calculi, and systems of Light Linear Logic. Finally, we show how to use Fitch-Prawitz Set Theory to define a type system that types precisely the strongly normalizing terms.

1 Introduction

In recent years, the authors have introduced in a series of papers [18, 16, 21, 20] various extensions of the Constructive Type Theory LF, with the goal of defining a simple *Universal Meta-language* that can support the effect of *gluing together*, *i.e.* interconnecting, different type systems and proof development environments.

The basic idea underpinning these logical frameworks is to allow for the user to express explicitly, in an LF type-theoretic framework the *invocation*, and uniform *recording* of the *effect*, of external tools by means of a new *monadic* type-constructor $\mathscr{L}^{\mathscr{P}}_{M,\sigma}[\cdot]$, called a *lock*. More specifically, locks permit to express the fact that, in order to obtain a term of a given type, it is necessary to *verify*, first, a constraint $\mathscr{P}(\Gamma \vdash_{\Sigma} M : \sigma)$, *i.e.* to *produce* suitable *evidence*. No restrictions are enforced on producing such evidence. It can be supplied by calling an *external proof search tool* or an *external oracle*, or exploiting some other epistemic source, such as diagrams, physical analogies, or explicit computations according to the *Poincaré Principle* [3]. Thus, by using lock constructors, one can *factor-out* the goal, produce pieces of evidence using different proof environments and *glue* them back together, using the *unlock operator*, which *releases* the locked term in the calling framework. Clearly, the task of checking the validity of

[*]The work presented in this paper was partially supported by the Serbian Ministry of Education, Science, and Technological Development, projects ON174026 and III44006.

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 3–17, doi:10.4204/EPTCS.185.1

external evidence rests entirely on the external tool. In our framework we limit ourselves to recording in the proof term by means of an $\mathscr{U}$-destructor this recourse to an external tool.

One of the original contributions of this paper is that we show how locks can delegate to external tools not only the task of producing suitable evidence but also that of exhibiting suitable *witnesses*, to be further used in the calling environment. This feature is exhibited by $\mathsf{CLLF}_{\mathscr{P}?}$ (see Section 3).

Locks subsume different *proof attitudes*, such as proof-irrelevant approaches, where one is only interested in knowing that evidence does exist, or approaches relying on powerful terminating metalanguages. Indeed, locks allow for a straightforward accommodation of many different *proof cultures* within a single Logical Framework; which otherwise can be embedded only very deeply [6, 15] or axiomatically [22].

Differently from our earlier work, we focus in this paper only on systems presented in the *canonical format* introduced by the CMU school [35, 14]. This format is syntax-directed and produces a unique derivation for each derivable judgement. Terms are all in normal form and equality rules are replaced by *hereditary substitution*. We present the systems in canonical form, since this format streamlines the proof of adequacy theorems.

First, we present the very expressive system $\mathsf{CLLF}_{\mathscr{P}}$ and discuss the relationship to its non-canonical counterpart $\mathsf{LLF}_{\mathscr{P}}$ in [20], where we introduced *lock-types* following the paradigm of Constructive Type Theory (*à la* Martin-Löf), via *introduction*, *elimination*, and *equality rules*. This paradigm needs to be rephrased for the canonical format used here. Introduction rules correspond to *type checking* rules of *canonical objects*, whereas elimination rules correspond to *type synthesis* rules of *atomic objects*. Equality rules are rendered via the rules of *hereditary substitution*. In particular, we introduce a *lock constructor* for building canonical objects $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[M]$ of type $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$, via the *type checking rule* $(O{\cdot}Lock)$. Correspondingly, we introduce an *unlock destructor*, $\mathscr{U}^{\mathscr{P}}_{N,\sigma}[M]$, and an *atomic rule* $(O{\cdot}Unlock)$, allowing elimination, in the hereditary substitution rules, of the lock-type constructor, under the condition that a specific predicate $\mathscr{P}$ is verified, possibly *externally*, on a judgement:

$$\frac{\Gamma\vdash_\Sigma M\Leftarrow\rho \quad \Gamma\vdash_\Sigma N\Leftarrow\sigma}{\Gamma\vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{N,\sigma}[M]\Leftarrow\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]}\ (O{\cdot}Lock) \qquad \frac{\Gamma\vdash_\Sigma A\Rightarrow\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \quad \Gamma\vdash_\Sigma N\Leftarrow\sigma \quad \mathscr{P}(\Gamma\vdash_\Sigma N\Leftarrow\sigma)}{\Gamma\vdash_\Sigma \mathscr{U}^{\mathscr{P}}_{N,\sigma}[A]\Rightarrow\rho}\ (O{\cdot}Unlock)$$

Capitalizing on the monadic nature of the lock constructor, as we did for the systems in [21, 20], one can use locked terms without necessarily establishing the predicate, provided an *outermost* lock is present. This increases the expressivity of the system, and allows for reasoning under the assumption that the verification is successful, as well as for postponing and reducing the number of verifications. The rules which make all this work are:

$$\frac{\Gamma,x{:}\tau\vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho]\ \mathsf{type} \quad \Gamma\vdash_\Sigma A\Rightarrow\mathscr{L}^{\mathscr{P}}_{S,\sigma}[\tau] \quad \rho[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]^F_{(\tau)^-}=\rho'}{\Gamma\vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho']\ \mathsf{type}}\ (F{\cdot}Nested{\cdot}Unlock)$$

$$\frac{\begin{array}{ll}\Gamma,x{:}\tau\vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[M]\Leftarrow\mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho] & \Gamma\vdash_\Sigma A\Rightarrow\mathscr{L}^{\mathscr{P}}_{S,\sigma}[\tau]\\ \rho[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]^F_{(\tau)^-}=\rho' & M[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]^O_{(\tau)^-}=M'\end{array}}{\Gamma\vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[M']\Leftarrow\mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho']}\ (O{\cdot}Nested{\cdot}Unlock)$$

The $(O{\cdot}Nested{\cdot}Unlock)$-rule is the counterpart of the elimination rule for monads, once we realize that the standard destructor of monads (see, e.g., [25]) $let_{T_{\mathscr{P}(\Gamma\vdash S:\sigma)}}x=A\ in\ N$ can be replaced, in our context, by $N[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]$. And this holds since the $\mathscr{L}^{\mathscr{P}}_{S,\sigma}[\cdot]$-monad satisfies the property $let_{T_{\mathscr{P}}}\ x=M\ in\ N\to N$ if $x\notin \mathsf{Fv}(N)$, provided x occurs *guarded* in N, *i.e.* within subterms of the appropriate lock-type. The rule $(F{\cdot}Nested{\cdot}Unlock)$ takes care of elimination at the level of types.

K	$\in$	$\mathscr{K}$	K	$::=$	$\mathsf{type} \mid \Pi x{:}\sigma.K$	*Kinds*
α	$\in$	$\mathscr{F}_a$	α	$::=$	$a \mid \alpha N$	*Atomic Families*
σ,τ,ρ	$\in$	$\mathscr{F}$	σ	$::=$	$\alpha \mid \Pi x{:}\sigma.\tau \mid \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$	*Canonical Families*
A	$\in$	$\mathscr{O}_a$	A	$::=$	$c \mid x \mid AM \mid \mathscr{U}^{\mathscr{P}}_{N,\sigma}[A]$	*Atomic Objects*
M,N	$\in$	$\mathscr{O}$	M	$::=$	$A \mid \lambda x{:}\sigma.M \mid \mathscr{L}^{\mathscr{P}}_{N,\sigma}[M]$	*Canonical Objects*
Σ	$\in$	$\mathscr{S}$	Σ	$::=$	$\emptyset \mid \Sigma, a{:}K \mid \Sigma, c{:}\sigma$	*Signatures*
Γ	$\in$	$\mathscr{C}$	Γ	$::=$	$\emptyset \mid \Gamma, x{:}\sigma$	*Contexts*

Figure 1: Syntax of $\mathsf{CLLF}_{\mathscr{P}}$

We proceed then to introduce $\mathsf{CLLF}_{\mathscr{P}?}$. Syntactically, it might appear as a minor variation of $\mathsf{CLLF}_{\mathscr{P}}$, but the lock constructor is used here to express the *request* for a witness satisfying a given property, which is then *replaced* by the unlock operation. In $\mathsf{CLLF}_{\mathscr{P}?}$, the lock acts as a *binding operator* and the unlock as an *application*.

To illustrate the expressive power of $\mathsf{CLLF}_{\mathscr{P}}$ and $\mathsf{CLLF}_{\mathscr{P}?}$ we discuss various challenging encodings of subtle logical systems, as well as some novel applications. First, we encode in $\mathsf{CLLF}_{\mathscr{P}}$ Fitch-Prawitz consistent Set-Theory (FPST), as presented in [30], and to illustrate its expressive power, we show, by way of example, how it can type all strongly normalizing terms. Next, we give signatures in $\mathsf{CLLF}_{\mathscr{P}}$ of a strongly normalizing λ-calculus and a system of Light Linear Logic [2]. Finally, in Section 4.5, we show how to encode functions in $\mathsf{CLLF}_{\mathscr{P}?}$.

The paper is organized as follows: in Section 2 we present the syntax, the type system and the metatheory of $\mathsf{CLLF}_{\mathscr{P}}$, whereas $\mathsf{CLLF}_{\mathscr{P}?}$ is introduced in Section 3. Section 4 is devoted to the presentation and discussion of case studies. Finally, connections with related work in the literature appear in Section 5.

2 The Canonical System $\mathsf{CLLF}_{\mathscr{P}}$

In this section, we discuss the *canonical* counterpart of $\mathsf{LLF}_{\mathscr{P}}$[20], *i.e.* $\mathsf{CLLF}_{\mathscr{P}}$, in the style of [35, 14]. This approach amounts to restricting the language only to terms in long $\beta\eta$-normal form. These are the normal forms of the original system which are normal also w.r.t. typed η-like expansion rules, namely $M \to \lambda x{:}\sigma.Mx$ and $M \to \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\mathscr{U}^{\mathscr{P}}_{N,\sigma}[M]]$ if M is atomic. The added value of canonical systems such as $\mathsf{CLLF}_{\mathscr{P}}$ is that one can streamline results of adequacy for encoded systems. Indeed, reductions in the meta-language of non-canonical terms reflect only the history of how the proof was developed using lemmata.

2.1 Syntax and Type System for $\mathsf{CLLF}_{\mathscr{P}}$

The syntax of $\mathsf{CLLF}_{\mathscr{P}}$ is presented in Figure 1. The type system for $\mathsf{CLLF}_{\mathscr{P}}$ is shown in Figure 2. The judgements of $\mathsf{CLLF}_{\mathscr{P}}$ are the following:

	Σ	sig	Σ is a valid signature
	$\vdash_\Sigma$	Γ	Γ is a valid context in Σ
Γ	$\vdash_\Sigma$	K	K is a kind in Γ and Σ
Γ	$\vdash_\Sigma$	σ type	σ is a canonical family in Γ and Σ
Γ	$\vdash_\Sigma$	$\alpha \Rightarrow K$	K is the kind of the atomic family α in Γ and Σ
Γ	$\vdash_\Sigma$	$M \Leftarrow \sigma$	M is a canonical term of type σ in Γ and Σ
Γ	$\vdash_\Sigma$	$A \Rightarrow \sigma$	σ is the type of the atomic term A in Γ and Σ

Valid signatures

$$\frac{}{\emptyset \text{ sig}}\ (S{\cdot}Empty) \qquad \frac{\Sigma \text{ sig} \quad \vdash_\Sigma K \quad a \notin \mathrm{Dom}(\Sigma)}{\Sigma, a{:}K \text{ sig}}\ (S{\cdot}Kind) \qquad \frac{\Sigma \text{ sig} \quad \vdash_\Sigma \sigma \text{ type} \quad c \notin \mathrm{Dom}(\Sigma)}{\Sigma, c{:}\sigma \text{ sig}}\ (S{\cdot}Type)$$

Kind rules

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \text{type}}\ (K{\cdot}Type)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma K}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.K}\ (K{\cdot}Pi)$$

Atomic Family rules

$$\frac{\vdash_\Sigma \Gamma \quad a{:}K \in \Sigma}{\Gamma \vdash_\Sigma a \Rightarrow K}\ (A{\cdot}Const)$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma \alpha \Rightarrow \Pi x{:}\sigma.K_1 \\ \Gamma \vdash_\Sigma M \Leftarrow \sigma \\ K_1[M/x]^K_{(\sigma)^-} = K\end{array}}{\Gamma \vdash_\Sigma \alpha M \Rightarrow K}\ (A{\cdot}App)$$

Canonical Family rules

$$\frac{\Gamma \vdash_\Sigma \alpha \Rightarrow \text{type}}{\Gamma \vdash_\Sigma \alpha \text{ type}}\ (F{\cdot}Atom)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau \text{ type}}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau \text{ type}}\ (F{\cdot}Pi)$$

$$\frac{\Gamma \vdash_\Sigma \rho \text{ type} \quad \Gamma \vdash_\Sigma N \Leftarrow \sigma}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \text{ type}}\ (F{\cdot}Lock)$$

$$\frac{\begin{array}{l}\Gamma, x : \tau \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho] \text{ type} \\ \Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\tau] \\ \rho[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]^F_{(\tau)^-} = \rho'\end{array}}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho'] \text{ type}}\ (F{\cdot}Nested{\cdot}Unlock)$$

Context rules

$$\frac{\Sigma \text{ sig}}{\vdash_\Sigma \emptyset}\ (C{\cdot}Empty)$$

$$\frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma \sigma \text{ type} \quad x \notin \mathrm{Dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\sigma}\ (C{\cdot}Type)$$

Atomic Object rules

$$\frac{\vdash_\Sigma \Gamma \quad c{:}\sigma \in \Sigma}{\Gamma \vdash_\Sigma c \Rightarrow \sigma}\ (O{\cdot}Const)$$

$$\frac{\vdash_\Sigma \Gamma \quad x{:}\sigma \in \Gamma}{\Gamma \vdash_\Sigma x \Rightarrow \sigma}\ (O{\cdot}Var)$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma A \Rightarrow \Pi x{:}\sigma.\tau_1 \\ \Gamma \vdash_\Sigma M \Leftarrow \sigma \quad \tau_1[M/x]^F_{(\sigma)^-} = \tau\end{array}}{\Gamma \vdash_\Sigma AM \Rightarrow \tau}\ (O{\cdot}App)$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \\ \Gamma \vdash_\Sigma N \Leftarrow \sigma \quad \mathscr{P}(\Gamma \vdash_\Sigma N \Leftarrow \sigma)\end{array}}{\Gamma \vdash_\Sigma \mathscr{U}^{\mathscr{P}}_{N,\sigma}[A] \Rightarrow \rho}\ (O{\cdot}Unlock)$$

Canonical Object rules

$$\frac{\Gamma \vdash_\Sigma A \Rightarrow \alpha}{\Gamma \vdash_\Sigma A \Leftarrow \alpha}\ (O{\cdot}Atom)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma M \Leftarrow \tau}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.M \Leftarrow \Pi x{:}\sigma.\tau}\ (O{\cdot}Abs)$$

$$\frac{\Gamma \vdash_\Sigma M \Leftarrow \rho \quad \Gamma \vdash_\Sigma N \Leftarrow \sigma}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{N,\sigma}[M] \Leftarrow \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]}\ (O{\cdot}Lock)$$

$$\frac{\begin{array}{ll}\Gamma, x{:}\tau \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[M] \Leftarrow \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho] & \Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\tau] \\ \rho[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]^F_{(\tau)^-} = \rho' & M[\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]/x]^O_{(\tau)^-} = M'\end{array}}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{S,\sigma}[M'] \Leftarrow \mathscr{L}^{\mathscr{P}}_{S,\sigma}[\rho']}\ (O{\cdot}Nested{\cdot}Unlock)$$

Figure 2: The CLLF$_{\mathscr{P}}$ Type System

The judgements Σ sig, and $\vdash_\Sigma \Gamma$, and $\Gamma \vdash_\Sigma K$ are as in Section 2.1 of [19], whereas the remaining ones are peculiar to the canonical style. Informally, the judgment $\Gamma \vdash_\Sigma M \Leftarrow \sigma$ uses σ to check the type of the canonical term M, while the judgment $\Gamma \vdash_\Sigma A \Rightarrow \sigma$ uses the type information contained in the atomic term A and Γ to synthesize σ. Predicates $\mathscr{P}$ in CLLF$_{\mathscr{P}}$ are defined on judgements of the shape $\Gamma \vdash_\Sigma M \Leftarrow \sigma$.

There are two rules whose conclusion is the lock constructor $\mathscr{L}^{\mathscr{P}}_{S,\sigma}[\cdot]$. But nevertheless, this system is still *syntax directed*: when there are subterms of the form $\mathscr{U}^{\mathscr{P}}_{S,\sigma}[A]$ in either M' or ρ', the type checking algorithm always tries to apply the $(O \cdot Nested \cdot Unlock)$ rule. If this is not possible, it applies instead the $(O \cdot Lock)$ rule.

The type system makes use, in the rules $(A{\cdot}App)$ and $(F{\cdot}App)$, of the notion of *Hereditary Substitution*, which computes the normal form resulting from the substitution of one normal form into another.

$$\frac{}{(a)^- = a} \qquad \frac{(\alpha)^- = \rho}{(\alpha M)^- = \rho} \qquad \frac{(\sigma)^- = \rho_1 \quad (\tau)^- = \rho_2}{(\Pi x{:}\sigma.\tau)^- = \rho_1 \to \rho_2} \qquad \frac{(\tau)^- = \rho}{(\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\tau])^- = \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]}$$

Figure 3: Erasure to simple-types

Substitution in Kinds

$$\frac{}{\mathsf{type}[M_0/x_0]^K_{\rho_0} = \mathsf{type}}\ (\mathscr{S}{\cdot}K{\cdot}Type) \qquad \frac{\sigma[M_0/x_0]^F_{\rho_0} = \sigma' \quad K[M_0/x_0]^K_{\rho_0} = K'}{(\Pi x{:}\sigma.K)[M_0/x_0]^K_{\rho_0} = \Pi x{:}\sigma.'K'}\ (\mathscr{S}{\cdot}K{\cdot}Pi)$$

Substitution in Atomic Families

$$\frac{}{a[M_0/x_0]^f_{\rho_0} = a}\ (\mathscr{S}{\cdot}F{\cdot}Const) \qquad \frac{\alpha[M_0/x_0]^f_{\rho_0} = \alpha' \quad M[M_0/x_0]^O_{\rho_0} = M'}{(\alpha M)[M_0/x_0]^f_{\rho_0} = \alpha' M'}\ (\mathscr{S}{\cdot}F{\cdot}App)$$

Substitution in Canonical Families

$$\frac{\alpha[M_0/x_0]^f_{\rho_0} = \alpha'}{\alpha[M_0/x_0]^F_{\rho_0} = \alpha'}\ (\mathscr{S}{\cdot}F{\cdot}Atom) \qquad \frac{\sigma_1[M_0/x_0]^F_{\rho_0} = \sigma_1' \quad \sigma_2[M_0/x_0]^F_{\rho_0} = \sigma_2'}{(\Pi x{:}\sigma_1.\sigma_2)[M_0/x_0]^F_{\rho_0} = \Pi x{:}\sigma_1'.\sigma_2'}\ (\mathscr{S}{\cdot}F{\cdot}Pi)$$

$$\frac{\sigma_1[M_0/x_0]^F_{\rho_0} = \sigma_1' \quad M_1[M_0/x_0]^O_{\rho_0} = M_1' \quad \sigma_2[M_0/x_0]^F_{\rho_0} = \sigma_2'}{\mathscr{L}^{\mathscr{P}}_{M_1,\sigma_1}[\sigma_2][M_0/x_0]^F_{\rho_0} = \mathscr{L}^{\mathscr{P}}_{M_1',\sigma_1'}[\sigma_2']}\ (\mathscr{S}{\cdot}F{\cdot}Lock)$$

Figure 4: Hereditary substitution, kinds and families of $\mathsf{CLLF}_{\mathscr{P}}$

The general form of the hereditary substitution judgement is $T[M/x]^t_\rho = T'$, where M is the term being substituted, x is the variable being substituted for, T is the term being substituted into, T' is the result of the substitution, ρ is the *simple-type* of M, and t denotes the syntactic class (*e.g.* atomic families/object, canonical families/objects, etc.) under consideration. We give the rules of the Hereditary Substitution in the style of [14], where the erasure function to simple types is necessary to simplify the proof of termination, which we omit.

The simple-type ρ of M is obtained via the erasure function of [14] (Figure 3), mapping dependent into simple-types. The rules for Hereditary Substitution are presented in Figures 4 and 5, using Barendregt's hygiene condition.

Notice that, in the rule $(O{\cdot}Atom)$ of the type system (Figure 2), the syntactic restriction of the classifier to α atomic ensures that canonical forms are *long $\beta\eta$-normal forms* for the suitable notion of long $\beta\eta$-normal form, which extends the standard one for lock-types. For one, the judgement $x{:}\Pi z{:}a.a \vdash_\Sigma x \Leftarrow \Pi z{:}a.a$ is not derivable, as $\Pi z{:}a.a$ is not atomic, hence $\vdash_\Sigma \lambda x{:}(\Pi z{:}a.a).x \Leftarrow \Pi x{:}(\Pi z{:}a.a).\Pi z{:}a.a$ is not derivable. On the other hand, $\vdash_\Sigma \lambda x{:}(\Pi z{:}a.a).\lambda y{:}a.xy \Leftarrow \Pi x{:}(\Pi z{:}a.a).\Pi z{:}a.a$, where a is a family constant of kind *Type*, is derivable. Analogously, for lock-types, the judgement $x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \vdash_\Sigma x \Leftarrow \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$ is not derivable, since $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$ is not atomic. As a consequence, we have that $\vdash_\Sigma \lambda x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho].x \Leftarrow \Pi x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho].\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$ is not derivable. However, $x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\mathscr{U}^{\mathscr{P}}_{N,\sigma}[x]] \Leftarrow \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$ is derivable, if ρ is atomic. Hence, the judgment $\vdash_\Sigma \lambda x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho].\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\mathscr{U}^{\mathscr{P}}_{N,\sigma}[x]] \Leftarrow \Pi x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho].\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$ is derivable. Note that the unlock constructor takes an *atomic* term as its main argument, thus avoiding the creation of possible $\mathscr{L}$-redexes under substitution. Moreover, since unlocks can only receive locked terms in their body, no abstractions can ever arise. In Definition 2.3, we formalize the notion of η-expansion of a judgement, together with correspondence theorems between $\mathsf{LLF}_{\mathscr{P}}$ and $\mathsf{CLLF}_{\mathscr{P}}$.

We present $\mathsf{CLLF}_{\mathscr{P}}$ in a fully-typed style, *i.e. à la* Church, but we could also follow [14] and present a version *à la* Curry, where the canonical forms $\lambda x.M$ and $\mathscr{L}^{\mathscr{P}}_M[N]$ do not carry type information. The type rules would then be, *e.g.*:

Substitution in Atomic Objects

$$\frac{}{c[M_0/x_0]^o_{\rho_0} = c}\,(\mathscr{S}\cdot O\cdot Const) \quad \frac{}{x_0[M_0/x_0]^o_{\rho_0} = M_0 : \rho_0}\,(\mathscr{S}\cdot O\cdot Var\cdot H) \quad \frac{x \neq x_0}{x[M_0/x_0]^o_{\rho_0} = x}\,(\mathscr{S}\cdot O\cdot Var)$$

$$\frac{A_1[M_0/x_0]^o_{\rho_0} = \lambda x{:}\rho_2.M_1' : \rho_2 \to \rho \quad M_2[M_0/x_0]^O_{\rho_0} = M_2' \quad M_1'[M_2'/x]^O_{\rho_2} = M'}{(A_1M_2)[M_0/x_0]^o_{\rho_0} = M' : \rho}\,(\mathscr{S}\cdot O\cdot App\cdot H)$$

$$\frac{A_1[M_0/x_0]^o_{\rho_0} = A_1' \quad M_2[M_0/x_0]^O_{\rho_0} = M_2'}{(A_1M_2)[M_0/x_0]^o_{\rho_0} = A_1'M_2'}\,(\mathscr{S}\cdot O\cdot App)$$

$$\frac{\sigma[M_0/x_0]^F_{\rho_0} = \sigma' \quad M[M_0/x_0]^O_{\rho_0} = M' \quad A[M_0/x_0]^o_{\rho_0} = \mathscr{L}^{\mathscr{P}}_{M',\sigma'}[M_1] : \mathscr{L}^{\mathscr{P}}_{M',\sigma'}[\rho]}{\mathscr{U}^{\mathscr{P}}_{M,\sigma}[A][M_0/x_0]^o_{\rho_0} = M_1 : \rho}\,(\mathscr{S}\cdot O\cdot Unlock\cdot H)$$

$$\frac{\sigma[M_0/x_0]^F_{\rho_0} = \sigma' \quad M[M_0/x_0]^O_{\rho_0} = M' \quad A[M_0/x_0]^o_{\rho_0} = A'}{\mathscr{U}^{\mathscr{P}}_{M,\sigma}[A][M_0/x_0]^o_{\rho_0} = \mathscr{U}^{\mathscr{P}}_{M',\sigma'}[A']}\,(\mathscr{S}\cdot O\cdot Unlock)$$

Substitution in Canonical Objects

$$\frac{A[M_0/x_0]^o_{\rho_0} = A'}{A[M_0/x_0]^O_{\rho_0} = A'}\,(\mathscr{S}\cdot O\cdot R) \quad \frac{A[M_0/x_0]^o_{\rho_0} = M' : \rho}{A[M_0/x_0]^O_{\rho_0} = M'}\,(\mathscr{S}\cdot O\cdot R\cdot H) \quad \frac{M[M_0/x_0]^O_{\rho_0} = M'}{\lambda x{:}\sigma.M[M_0/x_0]^O_{\rho_0} = \lambda x{:}\sigma.M'}\,(\mathscr{S}\cdot O\cdot Abs)$$

$$\frac{\sigma_1[M_0/x_0]^F_{\rho_0} = \sigma_1' \quad M_1[M_0/x_0]^O_{\rho_0} = M_1' \quad M_2[M_0/x_0]^O_{\rho_0} = M_2'}{\mathscr{L}^{\mathscr{P}}_{M_1,\sigma_1}[M_2][M_0/x_0]^O_{\rho_0} = \mathscr{L}^{\mathscr{P}}_{M_1',\sigma_1'}[M_2']}\,(\mathscr{S}\cdot O\cdot Lock)$$

Substitution in Contexts

$$\frac{}{[M_0/x_0]^C_{\rho_0} = \emptyset}\,(\mathscr{S}\cdot Ctxt\cdot Empty) \quad \frac{x_0 \neq x \quad x \notin \mathsf{Fv}(M_0) \quad \Gamma[M_0/x_0]^C_{\rho_0} = \Gamma' \quad \sigma[M_0/x_0]^F_{\rho_0} = \sigma'}{(\Gamma, x{:}\sigma)[M_0/x_0]^C_{\rho_0} = \Gamma', x{:}\sigma'}\,(\mathscr{S}\cdot Ctxt\cdot Term)$$

Figure 5: Hereditary substitution, objects and contexts of $\mathsf{CLLF}_{\mathscr{P}}$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma M \Leftarrow \tau}{\Gamma \vdash_\Sigma \lambda x.M \Leftarrow \Pi x{:}\sigma.\tau}\,(O\cdot Abs) \qquad \frac{\Gamma \vdash_\Sigma M \Leftarrow \sigma \quad \Gamma \vdash_\Sigma N \Leftarrow \tau}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_M[N] \Leftarrow \mathscr{L}^{\mathscr{P}}_{M,\sigma}[\tau]}\,(O\cdot Lock)$$

This latter syntax is more suitable in implementations because it simplifies the notation. Following [18], we stick to the typeful syntax because it allows for a more direct comparison with non-canonical systems. This, however, is technically immaterial. Since judgements in canonical systems have unique derivations, one can show by induction on derivations that any provable judgement in the system where object terms are *à la* Curry has a *unique* type decoration of its object subterms, which turns it into a provable judgement in the version *à la* Church. Vice versa, any provable judgement in the version *à la* Church can forget the types in its object subterms, yielding a provable judgement in the version *à la* Curry.

2.2 The Metatheory of $\mathsf{CLLF}_{\mathscr{P}}$

For lack of space we omit proofs, but these follow the standard patterns in [14, 19]. We start by studying the basic properties of hereditary substitution and the type system. First of all, we need to assume that the predicates are *well-behaved* in the sense of [19]. In the context of canonical systems, this notion needs to be rephrased as follows:

Definition 2.1 (Well-behaved predicates for canonical systems)**.** A finite set of predicates $\{\mathscr{P}_i\}_{i\in I}$ is *well-behaved* if each $\mathscr{P}$ in the set satisfies the following conditions:

1. ***Closure under signature and context weakening and permutation:***
   (a) If Σ and Ω are valid signatures such that $\Sigma \subseteq \Omega$ and $\mathscr{P}(\Gamma \vdash_\Sigma N \Leftarrow \sigma)$, then $\mathscr{P}(\Gamma \vdash_\Omega N \Leftarrow \sigma)$.

(b) If Γ and Δ are valid contexts such that $\Gamma \subseteq \Delta$ and $\mathscr{P}(\Gamma \vdash_\Sigma N \Leftarrow \sigma)$, then $\mathscr{P}(\Delta \vdash_\Sigma N \Leftarrow \sigma)$.

2. ***Closure under hereditary substitution:*** If $\mathscr{P}(\Gamma, x{:}\sigma', \Gamma' \vdash_\Sigma N \Leftarrow \sigma)$ and $\Gamma \vdash_\Sigma N' : \sigma'$, then $\mathscr{P}(\Gamma, \Gamma'[N'/x]^C_{(\sigma')^-} \vdash_\Sigma N[N'/x]^O_{(\sigma')^-} \Leftarrow \sigma[N'/x]^F_{(\sigma')^-})$.

As canonical systems do not feature reduction, the "classical" third constraint for well-behaved predicates (closure under reduction) is not needed here. Moreover, the second condition (*closure under substitution*) becomes "closure under hereditary substitution".

Lemma 2.1 (Decidability of hereditary substitution)**.**

1. *For any T in $\{\mathscr{K}, \mathscr{A}, \mathscr{F}, \mathscr{O}, \mathscr{C}\}$, and any M, x, and ρ, it is decidable whether there exists a T' such that $T[M/x]^m_\rho = T'$ or there is no such T'.*
2. *For any M, x, ρ, and A, it is decidable whether there exists an A', such that $A[M/x]^o_\rho = A'$, or there exist M' and ρ', such that $A[M/x]^o_\rho = M' : \rho'$, or there are no such A' and M'.*

Lemma 2.2 (Head substitution size)**.** *If $A[M_0/x_0]^o_{\rho_0} = M{:}\rho$, then ρ is a subexpression of ρ_0.*

Lemma 2.3 (Uniqueness of substitution and synthesis)**.**

1. *It is not possible that $A[M_0/x_0]^o_{\rho_0} = A'$ and $A[M_0/x_0]^o_{\rho_0} = M{:}\rho$.*
2. *For any T, if $T[M_0/x_0]^m_{\rho_0} = T'$, and $T[M_0/x_0]^m_{\rho_0} = T''$, then $T' = T''$.*
3. *If $\Gamma \vdash_\Sigma \alpha \Rightarrow K$, and $\Gamma \vdash_\Sigma \alpha \Rightarrow K'$, then $K = K'$.*
4. *If $\Gamma \vdash_\Sigma A \Rightarrow \sigma$, and $\Gamma \vdash_\Sigma A \Rightarrow \sigma'$, then $\sigma = \sigma'$.*

Lemma 2.4 (Composition of hereditary substitution)**.** *Let $x \neq x_0$ and $x \notin \mathsf{Fv}(M_0)$. Then:*

1. *For all T_1' in $\{\mathscr{K}, \mathscr{F}_a, \mathscr{F}, \mathscr{O}_a, \mathscr{O}\}$, if $M_2[M_0/x_0]^O_{\rho_0} = M_2'$, $T_1[M_2/x]^m_{\rho_2} = T_1'$, and $T_1[M_0/x_0]^m_{\rho_0} = T_1''$, then there exists a T: $T_1'[M_0/x_0]^m_{\rho_0} = T$, and $T_1''[M_2'/x]^m_{\rho_2} = T$.*
2. *If $M_2[M_0/x_0]^O_{\rho_0} = M_2'$, $A_1[M_2/x]^o_{\rho_2} = M : \rho$, and $A_1[M_0/x_0]^o_{\rho_0} = A$, then there exists an M': $M[M_0/x_0]^O_{\rho_0} = M'$, and $A[M_2'/x]^o_{\rho_2} = M' : \rho$.*
3. *If $M_2[M_0/x_0]^O_{\rho_0} = M_2'$, $A_1[M_2/x]^o_{\rho_2} = A$, and $A_1[M_0/x_0]^o_{\rho_0} = M : \rho$, then there exists an M': $A[M_0/x_0]^o_{\rho_0} = M' : \rho$, and $M[M_2'/x]^O_{\rho_2} = M'$.*

Theorem 2.5 (Transitivity)**.** *Let Σ sig, $\vdash_\Sigma \Gamma, x_0{:}\rho_0, \Gamma'$ and $\Gamma \vdash_\Sigma M_0 \Leftarrow \rho_0$, and assume that all predicates are well-behaved. Then,*

1. *There exists a Γ'': $[M_0/x_0]^C_{\rho_0} = \Gamma''$ and $\vdash_\Sigma \Gamma, \Gamma''$.*
2. *If $\Gamma, x_0{:}\rho_0, \Gamma' \vdash_\Sigma K$ then there exists a K': $[M_0/x_0]^K_{\rho_0} K = K'$ and $\Gamma, \Gamma'' \vdash_\Sigma K'$.*
3. *If $\Gamma, x_0{:}\rho_0, \Gamma' \vdash_\Sigma \sigma$ type, then there exists a σ': $[M_0/x_0]^F_{\rho_0}\sigma = \sigma'$ and $\Gamma, \Gamma'' \vdash_\Sigma \sigma'$ type.*
4. *If $\Gamma, x_0{:}\rho_0, \Gamma' \vdash_\Sigma \sigma$ type and $\Gamma, x_0{:}\rho_0, \Gamma' \vdash_\Sigma M \Leftarrow \sigma$, then there exist σ' and M': $[M_0/x_0]^F_{\rho_0}\sigma = \sigma'$ and $[M_0/x_0]^O_{\rho_0} M = M'$ and $\Gamma, \Gamma'' \vdash_\Sigma M' \Leftarrow \sigma'$.*

Theorem 2.6 (Decidability of typing)**.** *If predicates in $\mathsf{CLLF}_{\mathscr{P}}$ are decidable, then all of the judgements of the system are decidable.*

We can now precisely state the relationship between $\mathsf{CLLF}_{\mathscr{P}}$ and the $\mathsf{LLF}_{\mathscr{P}}$ system of [20]:

Theorem 2.7 (Soundness)**.** *For any predicate $\mathscr{P}$ of $\mathsf{CLLF}_{\mathscr{P}}$, we define a corresponding predicate in $\mathsf{LLF}_{\mathscr{P}}$ as follows: $\mathscr{P}(\Gamma \vdash_\Sigma M : \sigma)$ holds if and only if $\Gamma \vdash_\Sigma M : \sigma$ is derivable in $\mathsf{LLF}_{\mathscr{P}}$ and $\mathscr{P}(\Gamma \vdash_\Sigma M \Leftarrow \sigma)$ holds in $\mathsf{CLLF}_{\mathscr{P}}$. Then, we have:*

1. *If Σ sig is derivable in $\mathsf{CLLF}_{\mathscr{P}}$, then Σ sig is derivable in $\mathsf{LLF}_{\mathscr{P}}$.*
2. *If $\vdash_\Sigma \Gamma$ is derivable in $\mathsf{CLLF}_{\mathscr{P}}$, then $\vdash_\Sigma \Gamma$ is derivable in $\mathsf{LLF}_{\mathscr{P}}$.*
3. *If $\Gamma \vdash_\Sigma K$ is derivable in $\mathsf{CLLF}_{\mathscr{P}}$, then $\Gamma \vdash_\Sigma K$ is derivable in $\mathsf{LLF}_{\mathscr{P}}$.*

4. *If* $\Gamma \vdash_\Sigma \alpha \Rightarrow K$ *is derivable in* $\mathsf{CLLF}_{\mathscr{P}}$*, then* $\Gamma \vdash_\Sigma \alpha : K$ *is derivable in* $\mathsf{LLF}_{\mathscr{P}}$.
5. *If* $\Gamma \vdash_\Sigma \sigma$ type *is derivable in* $\mathsf{CLLF}_{\mathscr{P}}$*, then* $\Gamma \vdash_\Sigma \sigma :$ type *is derivable in* $\mathsf{LLF}_{\mathscr{P}}$.
6. *If* $\Gamma \vdash_\Sigma A \Rightarrow \sigma$ *is derivable in* $\mathsf{CLLF}_{\mathscr{P}}$*, then* $\Gamma \vdash_\Sigma A : \sigma$ *is derivable in* $\mathsf{LLF}_{\mathscr{P}}$.
7. *If* $\Gamma \vdash_\Sigma M \Leftarrow \sigma$ *is derivable in* $\mathsf{CLLF}_{\mathscr{P}}$*, then* $\Gamma \vdash_\Sigma M : \sigma$ *is derivable in* $\mathsf{LLF}_{\mathscr{P}}$.

Vice versa, all $\mathsf{LLF}_{\mathscr{P}}$ judgements in *long* $\beta\eta$*-normal form* ($\beta\eta$-lnf) are derivable in $\mathsf{CLLF}_{\mathscr{P}}$. The definition of a judgement in $\beta\eta$-lnf is based on the following extension of the standard η-rule to the lock constructor $\lambda x{:}\sigma.Mx \to_\eta M$ and $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\mathscr{U}^{\mathscr{P}}_{N,\sigma}[M]] \to_\eta M$.

Definition 2.2. An occurrence ξ of a constant or a variable in a term of an $\mathsf{LLF}_{\mathscr{P}}$ judgement is *fully applied and unlocked* w.r.t. its type or kind $\Pi\vec{x}_1{:}\vec{\sigma}_1.\vec{\mathscr{L}}_1[\ldots\Pi\vec{x}_n{:}\vec{\sigma}_n.\vec{\mathscr{L}}_n[\alpha]\ldots]$, where $\vec{\mathscr{L}}_1,\ldots,\vec{\mathscr{L}}_n$ are vectors of locks, if ξ appears only in contexts that are of the form $\vec{\mathscr{U}}_n[(\ldots(\vec{\mathscr{U}}_1[\xi\vec{M}_1])\ldots)\vec{M}_n]$, where $\vec{M}_1,\ldots,\vec{M}_n$, $\vec{\mathscr{U}}_1,\ldots,\vec{\mathscr{U}}_n$ have the same arities of the corresponding vectors of Π's and locks.

Definition 2.3 (Judgements in long $\beta\eta$-normal form)**.**

1. A term T in a judgement is in $\beta\eta$*-lnf* if T is in normal form and every constant and variable occurrence in T is fully applied and unlocked w.r.t. its classifier in the judgement.
2. A judgement is in $\beta\eta$*-lnf* if all terms appearing in it are in $\beta\eta$-lnf.

Theorem 2.8 (Correspondence)**.** *Assume that all predicates in* $\mathsf{LLF}_{\mathscr{P}}$ *are well-behaved, according to Definition 2.1 [19]. For any predicate* $\mathscr{P}$ *in* $\mathsf{LLF}_{\mathscr{P}}$*, we define a corresponding predicate in* $\mathsf{CLLF}_{\mathscr{P}}$ *with:* $\mathscr{P}(\Gamma \vdash_\Sigma M \Leftarrow \sigma)$ *holds if* $\Gamma \vdash_\Sigma M \Leftarrow \sigma$ *is derivable in* $\mathsf{CLLF}_{\mathscr{P}}$ *and* $\mathscr{P}(\Gamma \vdash_\Sigma M : \sigma)$ *holds in* $\mathsf{LLF}_{\mathscr{P}}$*. Then, we have:*

1. *If* Σ sig *is in* $\beta\eta$*-lnf and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* Σ sig *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*
2. *If* $\vdash_\Sigma \Gamma$ *is in* $\beta\eta$*-lnf and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* $\vdash_\Sigma \Gamma$ *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*
3. *If* $\Gamma \vdash_\Sigma K$ *is in* $\beta\eta$*-lnf, and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* $\Gamma \vdash_\Sigma K$ *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*
4. *If* $\Gamma \vdash_\Sigma \alpha : K$ *is in* $\beta\eta$*-lnf and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* $\Gamma \vdash_\Sigma \alpha \Rightarrow K$ *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*
5. *If* $\Gamma \vdash_\Sigma \sigma{:}$type *is in* $\beta\eta$*-lnf and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* $\Gamma \vdash_\Sigma \sigma$ type *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*
6. *If* $\Gamma \vdash_\Sigma A : \alpha$ *is in* $\beta\eta$*-lnf and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* $\Gamma \vdash_\Sigma A \Rightarrow \alpha$ *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*
7. *If* $\Gamma \vdash_\Sigma M : \sigma$ *is in* $\beta\eta$*-lnf and is* $\mathsf{LLF}_{\mathscr{P}}$*-derivable, then* $\Gamma \vdash_\Sigma M \Leftarrow \sigma$ *is* $\mathsf{CLLF}_{\mathscr{P}}$*-derivable.*

Notice that, by the Correspondence Theorem above, any well-behaved predicate $\mathscr{P}$ in $\mathsf{LLF}_{\mathscr{P}}$ in the sense of Definition 2.1 [19] induces a well-behaved predicate in $\mathsf{CLLF}_{\mathscr{P}}$. Finally, notice that *not* all $\mathsf{LLF}_{\mathscr{P}}$ judgements have a corresponding $\beta\eta$-lnf. Namely, the judgement $x{:}\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \vdash_\Sigma x : \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$ does not admit an η-expanded normal form when the predicate $\mathscr{P}$ does *not* hold on N, as the rule $(O{\cdot}Unlock)$ can be applied only when the predicate holds.

3 The Type System $\mathsf{CLLF}_{\mathscr{P}?}$

The main idea behind $\mathsf{CLLF}_{\mathscr{P}?}$ (see Figures 6, 7, and 8)[1] is to "empower" the framework of $\mathsf{CLLF}_{\mathscr{P}}$ by *adding* to the lock/unlock mechanism the possibility to receive from the external oracle a *witness* satisfying suitable constraints. Thus, we can pave the way for gluing together different proof development environments beyond proof irrelevance scenarios. In this context, the lock constructor behaves as a *binder*. The new $(O{\cdot}Lock)$ rule is the following:

[1] For lack of space, we present in these figures only the categories and rules of $\mathsf{CLLF}_{\mathscr{P}?}$ that differ from their $\mathsf{CLLF}_{\mathscr{P}}$ counterparts.

$$\begin{array}{rcccll} \sigma,\tau,\rho & \in & \mathscr{F} & \sigma & ::= & \alpha \mid \Pi x{:}\sigma.\tau \mid \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] \qquad \textit{Canonical Families} \\ M,N & \in & \mathscr{O} & M & ::= & A \mid \lambda x{:}\sigma.M \mid \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M] \qquad \textit{Canonical Objects} \end{array}$$

Figure 6: $\mathsf{CLLF}^{?}_{\mathscr{P}}$ Syntax — changes w.r.t. $\mathsf{CLLF}_{\mathscr{P}}$

Canonical Family rules

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \rho \text{ type}}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] \text{ type}} \ (F{\cdot}Lock)$$

$$\frac{\begin{array}{l}\Gamma, y:\tau \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] \text{ type} \\ \Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\tau] \\ \rho[\mathscr{U}^{\mathscr{P}}_{x,\sigma}[A]/y]^{F}_{(\tau)^-} = \rho'\end{array}}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho'] \text{type}} \ (F{\cdot}Nested{\cdot}Unlock)$$

Atomic Object rules

$$\frac{\begin{array}{ll}\Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] & \Gamma \vdash_\Sigma N \Leftarrow \sigma \\ \mathscr{P}(\Gamma \vdash_\Sigma N \Leftarrow \sigma) & \rho[N/x]^{F}_{(\sigma)^-} = \rho'\end{array}}{\Gamma \vdash_\Sigma \mathscr{U}^{\mathscr{P}}_{N,\sigma}[A] \Rightarrow \rho'} \ (O{\cdot}Unlock)$$

Canonical Object rules

$$\frac{\Gamma x{:}\sigma \vdash_\Sigma M \Leftarrow \rho}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M] \Leftarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho]} \ (O{\cdot}Lock)$$

$$\frac{\begin{array}{ll}\Gamma, y{:}\tau \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M] \Leftarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] & \Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\tau] \\ \rho[\mathscr{U}^{\mathscr{P}}_{x,\sigma}[A]/y]^{F}_{(\tau)^-} = \rho' & M[\mathscr{U}^{\mathscr{P}}_{x,\sigma}[A]/y]^{O}_{(\tau)^-} = M'\end{array}}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M'] \Leftarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho']} \ (O{\cdot}Nested{\cdot}Unlock)$$

Figure 7: The $\mathsf{CLLF}^{?}_{\mathscr{P}}$ Type System — changes w.r.t. $\mathsf{CLLF}_{\mathscr{P}}$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma M \Leftarrow \rho}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M] \Leftarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho]}$$

where the variable x is a placeholder bound in M and ρ, which will be replaced by the concrete term that will be returned by the external oracle call. The intuitive meaning behind the $(O{\cdot}Lock)$ rule is, therefore, that of recording the need to delegate to the external oracle the inference of a suitable witness of a given type. Indeed, M can be thought of as an "incomplete" term which needs to be completed by an inhabitant of a given type σ satisfying the constraint $\mathscr{P}$. The actual term, possibly synthesized by the external tool, will be "released" in $\mathsf{CLLF}^{?}_{\mathscr{P}}$, by the unlock constructor in the $(O{\cdot}Unlock)$ rule as follows:

$$\frac{\Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] \quad \rho[N/x]^{F}_{(\sigma)^-} = \rho' \quad \Gamma \vdash_\Sigma N \Leftarrow \sigma \quad \mathscr{P}(\Gamma \vdash_\Sigma N \Leftarrow \sigma)}{\Gamma \vdash_\Sigma \mathscr{U}^{\mathscr{P}}_{N,\sigma}[A] \Rightarrow \rho'}$$

The term $\mathscr{U}^{\mathscr{P}}_{N,\sigma}[M]$ intuitively means that N is precisely the synthesized term satisfying the constraint $\mathscr{P}(\Gamma \vdash_\Sigma N \Leftarrow \sigma)$ that will replace in $\mathsf{CLLF}^{?}_{\mathscr{P}}$ all the free occurrences of x in ρ. This replacement is executed in the $(\mathscr{S}{\cdot}O{\cdot}Unlock{\cdot}H)$ hereditary substitution rule (Figure 8).

Similarly to $\mathsf{CLLF}_{\mathscr{P}}$, also in $\mathsf{CLLF}^{?}_{\mathscr{P}}$ it is possible to "postpone" or delay the verification of an external predicate in a lock, provided an *outermost* lock is present. Whence, the synthesis of the actual inhabitant N can be delayed, thanks to the $(O{\cdot}Nested{\cdot}Unlock)$ rule:

$$\frac{\Gamma, y{:}\tau \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M] \Leftarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho] \quad \Gamma \vdash_\Sigma A \Rightarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\tau] \quad \rho[\mathscr{U}^{\mathscr{P}}_{x,\sigma}[A]/y]^{F}_{(\tau)^-} = \rho' \quad M[\mathscr{U}^{\mathscr{P}}_{x,\sigma}[A]/y]^{O}_{(\tau)^-} = M'}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{x,\sigma}[M'] \Leftarrow \mathscr{L}^{\mathscr{P}}_{x,\sigma}[\rho']}$$

The Metatheory of $\mathsf{CLLF}^{?}_{\mathscr{P}}$ follows closely that of $\mathsf{CLLF}_{\mathscr{P}}$ as far as decidability. We have no correspondence theorem since we did not introduce a non-canonical variant $\mathsf{CLLF}^{?}_{\mathscr{P}}$. This could have been done similarly to $\mathsf{LLF}_{\mathscr{P}}$.

Substitution in Canonical Families

$$\frac{\sigma_1[M_0/x_0]^F_{\rho_0} = \sigma_1' \quad \sigma_2[M_0/x_0]^F_{\rho_0} = \sigma_2'}{\mathscr{L}^{\mathscr{P}}_{x,\sigma_1}[\sigma_2][M_0/x_0]^F_{\rho_0} = \mathscr{L}^{\mathscr{P}}_{x,\sigma_1'}[\sigma_2']} \ (\mathscr{S}\cdot F\cdot Lock)$$

Substitution in Atomic Objects

$$\frac{\sigma[M_0/x_0]^F_{\rho_0} = \sigma' \quad M[M_0/x_0]^o_{\rho_0} = M' \quad M_1[M'/x]^o_{(\sigma')^-} = M_2 \quad A[M_0/x_0]^o_{\rho_0} = \mathscr{L}^{\mathscr{P}}_{x,\sigma'}[M_1] : \mathscr{L}^{\mathscr{P}}_{x,\sigma'}[\rho]}{\mathscr{U}^{\mathscr{P}}_{M,\sigma}[A][M_0/x_0]^o_{\rho_0} = M_2 : \rho} \ (\mathscr{S}\cdot O\cdot Unlock\cdot H)$$

Substitution in Canonical Objects

$$\frac{\sigma_1[M_0/x_0]^F_{\rho_0} = \sigma_1' \quad M_1[M_0/x_0]^O_{\rho_0} = M_1'}{\mathscr{L}^{\mathscr{P}}_{x,\sigma_1}[M_1][M_0/x_0]^O_{\rho_0} = \mathscr{L}^{\mathscr{P}}_{x,\sigma_1'}[M_1']} \ (\mathscr{S}\cdot O\cdot Lock)$$

Figure 8: $\mathsf{CLLF}_{\mathscr{P}?}$ Hereditary Substitution — changes w.r.t. $\mathsf{CLLF}_{\mathscr{P}}$

4 Case studies

In this section, we discuss the encodings of a collection of logical systems which illustrate the expressive power and the flexibility of $\mathsf{CLLF}_{\mathscr{P}}$ and $\mathsf{CLLF}_{\mathscr{P}?}$. We discuss Fitch-Prawitz Consistent Set theory, FPST [30], some applications of FPST to normalizing λ-calculus, a system of Light Linear Logic in $\mathsf{CLLF}_{\mathscr{P}}$, and an the encoding of a *partial* function in $\mathsf{CLLF}_{\mathscr{P}?}$.

The crucial step in encoding a logical system in $\mathsf{CLLF}_{\mathscr{P}}$ or $\mathsf{CLLF}_{\mathscr{P}?}$ is to define the predicates involved in locks. Predicates defined on closed terms are usually unproblematic. Difficulties arise in enforcing the properties of closure under hereditary substitution and closure under signature and context extension, when predicates are defined on open terms. To be able to streamline the definition of well-behaved predicates we introduce the following:

Definition 4.1. Given a signature Σ let Λ_Σ (respectively Λ^o_Σ) be the set of $\mathsf{LLF}_{\mathscr{P}}$ terms (respectively *closed* $\mathsf{LLF}_{\mathscr{P}}$ terms) definable using constants from Σ. A term M has a *skeleton* in Λ_Σ if there exists a term $N[x_1,\ldots,x_n] \in \Lambda_\Sigma$, whose free variables (called *holes* of the skeleton) are in $\{x_1,\ldots,x_n\}$, and there exist terms $M_1,\ldots,M_n$ such that $M \equiv N[M_1/x_1,\ldots,M_n/x_n]$.

4.1 Fitch Set Theory *à la* Prawitz - FPST

In this section, we present the encoding of a formal system of remarkable logical as well as historical significance, namely the system of consistent *Naïve* Set Theory, FPST, introduced by Fitch [11]. This system was first presented in Natural Deduction style by Prawitz [30]. As Naïve Set Theory is inconsistent, to prevent the derivation of inconsistencies from the unrestricted *abstraction* rule, only normalizable *deductions* are allowed in FPST. Of course, this side-condition is extremely difficult to capture using traditional tools.

In the present context, instead, we can put to use the machinery of $\mathsf{CLLF}_{\mathscr{P}}$ to provide an appropriate encoding of FPST where the *global* normalization constraint is enforced *locally* by checking the proof-object. This encoding beautifully illustrates the *bag of tricks* that $\mathsf{CLLF}_{\mathscr{P}}$ supports. Checking that a proof term is normalizable would be the obvious predicate to use in the corresponding lock-type, but this would not be a well-behaved predicate if free variables, *i.e.* assumptions, are not sterilized. To this end, We introduce a distinction between *generic* judgements, which cannot be directly utilized in arguments, but which can be assumed, and *apodictic* judgements, which are directly involved in proof rules. In order to make use of generic judgements, one has to downgrade them to an apodictic one. This is achieved by

a suitable coercion function.

Definition 4.2 (Fitch Prawitz Set Theory, FPST)**.** For the lack of space, here we only give the crucial rules for implication and for *set-abstraction* and the corresponding elimination rules of the full system of Fitch (see [30]), as presented by Prawitz:

$$\frac{\Gamma, A \vdash_{\mathsf{FPST}} B}{\Gamma \vdash_{\mathsf{FPST}} A \supset B}(\supset I) \qquad \frac{\Gamma \vdash_{\mathsf{FPST}} A \quad \Gamma \vdash_{\mathsf{FPST}} A \supset B}{\Gamma \vdash_{\mathsf{FPST}} B}(\supset E)$$

$$\frac{\Gamma \vdash_{\mathsf{FPST}} A[T/x]}{\Gamma \vdash_{\mathsf{FPST}} T \in \lambda x.A}(\lambda I) \qquad \frac{\Gamma \vdash_{\mathsf{FPST}} T \in \lambda x.A}{\Gamma \vdash_{\mathsf{FPST}} A[T/x]}(\lambda E)$$

The intended meaning of the term $\lambda x.A$ is the set $\{x \mid A\}$. In Fitch's system, FPST, conjunction and universal quantification are defined as usual, while negation is defined constructively, but it still allows for the usual definitions of disjunction and existential quantification. What makes FPST *consistent* is that not all standard deductions in FPST are legal. Standard deductions are called *quasi-deductions* in FPST. A *legal deduction* in FPST is defined instead, as a quasi-deduction which is *normalizable* in the standard sense of Natural Deduction, namely it can be transformed in a derivation where all elimination rules occur before introductions.

Definition 4.3 ($\mathsf{LLF}_{\mathscr{P}}$ signature Σ_{FPST} for Fitch Prawitz Set Theory)**.** The following constants are introduced:

```
o   : Type              ι         : Type
T   : o -> Type         δ         : ΠA:o.  (V(A) -> T(A))
V   : o -> Type          λ_intro  : ΠA:ι ->o.Πx:ι.T(A x) -> T(ε x (lam A))
lam : (ι -> o)-> ι     λ_elim    : ΠA:ι ->o.Πx:ι.T(ε x (lam A))->T(A x)
ε   : ι -> ι -> o      ⊃_intro: ΠA,B:o.(V(A) -> T(B)) -> (T(A ⊃B))
⊃   : o -> o -> o      ⊃_elim : ΠA,B:o.Πx:T(A).Πy:T(A⊃B) -> 𝒟^Fitch_⟨x,y⟩,T(A)×T(A⊃B)[T(B)]
```

where o is the type of propositions, $\supset$ and the "membership" predicate ε are the syntactic constructors for propositions, lam is the "abstraction" operator for building "sets", T is the apodictic judgement, V is the generic judgement, δ is the coercion function, and $\langle \mathtt{x}, \mathtt{y} \rangle$ denotes the encoding of pairs, whose type is denoted by $\sigma \times \tau$, *e.g.* $\lambda \mathtt{u}{:}\sigma \to \tau \to \rho.\ \mathtt{u}\ \mathtt{x}\ \mathtt{y} : (\sigma \to \tau \to \rho) \to \rho$. The predicate in the lock is defined as follows:

$$\mathtt{Fitch}(\Gamma \vdash_{\Sigma_{\mathsf{FPST}}} \langle \mathtt{x}, \mathtt{y} \rangle \ \Leftarrow\ \mathtt{T(A)} \times \mathtt{T(A} \supset \mathtt{B)})$$

it holds iff x and y have skeletons in $\Lambda_{\Sigma_{\mathsf{FPST}}}$, all the holes of which have either type o or are guarded by a δ, and hence have type V(A), and, moreover, the proof derived by combining the skeletons of x and y is normalizable in the natural sense. Clearly, this predicate is only semi-decidable.

For lack of space, we do not spell out the rules concerning the other logical operators, because they are all straightforward provided we use only the apodictic judgement $\mathtt{T}(\cdot)$, but a few remarks are mandatory. The notion of *normalizable proof* is the standard notion used in natural deduction. The predicate Fitch is well-behaved because it considers terms only up-to holes in the skeleton, which can have type o or are generic judgements. Adequacy for this signature can be achieved in the format of [19]:

Theorem 4.1 (Adequacy for Fitch-Prawitz Naive Set Theory)**.** *If $A_1, \ldots, A_n$ are the atomic formulas occurring in $B_1, \ldots, B_m, A$, then $B_1 \ldots B_m \vdash_{\mathsf{FPST}} A$ iff there exists a normalizable* M *such that* $\mathtt{A_1}{:}\mathtt{o}, \ldots, \mathtt{A_n}{:}\mathtt{o}$, $\mathtt{x_1}{:}\mathtt{V(B_1)}, \ldots, \mathtt{x_m}{:}\mathtt{V(B_m)} \vdash_{\Sigma_{\mathsf{FPST}}} \mathtt{M} \Leftarrow \mathtt{T(A)}$ *(where* A*, and* $\mathtt{B_i}$ *represent the encodings of, respectively, A and B_i in $\mathsf{CLLF}_{\mathscr{P}}$, for $1 \leq i \leq m$).*

4.2 A Type System for strongly normalizing λ-terms

Fitch-Prawitz Set Theory, FPST, is a rather intriguing, albeit unexplored, set theoretic system. The normalizability criterion for accepting a quasi-deduction prevents the derivation of contradictions and hence makes the system consistent. Of course, some intuitive rules are not derivable. For instance *modus ponens* does not hold and if $t \in \lambda x.A$ then we do not have necessarily that $A[t/x]$ holds. Similarly, the *transitivity* property does not hold. However FPST is a very expressive type system which "encompasses" many kinds of quantification, provided normalization is preserved and Fitch has shown, see *e.g.* [11], that a large portion of ordinary Mathematics can be carried out in FPST.

In this subsection, we sketch how to use FPST to define a type system which can type *precisely all* the strongly normalizing λ-terms. Namely, we show that in FPST there exists a set Λ to which belong only the strongly normalizing λ-terms. We speak of a *type system* because the proof in FPST that a term belongs to Λ is *syntax directed*. First we need to be able to define recursive objects in FPST. We adapt, to FPST, Prop. 4, Appendix A.1 of [13], originally given by J-Y. Girard for Light Linear Logic, as:

Theorem 4.2 (Fixpoint)**.** *Let $A[P,x_1\ldots,x_n]$ be a formula of* FPST *with an n-ary predicate variable P. Then, there exists a formula B of* FPST*, such that there exists a normalizable deduction in* FPST *between $A[\lambda x_1\ldots,x_n.B[x_1,\ldots,x_n],x_1\ldots,x_n]$ and B, and viceversa.*

Proof. Let equality be Leibniz equality, then, assuming $n=1$, define $\Lambda \equiv \lambda z.\exists x.\exists y.z = \langle x,y\rangle \& A[(\lambda w.\langle w, y\rangle \in y),x]$. Then $\langle x,\Lambda\rangle \in \Lambda$ is equivalent, in the sense of FPST, to $A[(\lambda w.\langle w,\Lambda\rangle \in \Lambda),x]$. □

Using the Fixpoint Theorem we define first natural numbers, then a concrete representation of the terms of λ-calculus, say Λ_0. Using again the Fixed Point Theorem, we define a (representation of) the substitution function over terms in Λ_0 and finally the set Λ, such that $x \in \Lambda$ is equivalent in FPST to $x \in \Lambda_0 \& \forall y.y \in \Lambda_0 \subset app(x,y) \in \Lambda$. Here, $app(x,y)$ denotes the concrete representation of "applying" x to y. One can derive in FPST that (a representation of) a λ-term, say M, belongs to Λ, only if there is a normalizable derivation of $M \in \Lambda$. But then it is straightforward to check that only normalizing terms can be typed in FPST with Λ, *i.e.* belong to Λ. There is indeed a natural reflection of the normalizability of the FPST derivation of the typing judgement $M \in \Lambda$, and the fact that the term represented by M is indeed normalizable!

4.3 A Normalizing call-by-value λ-calculus

In this section we sketch how to express in $\mathsf{CLLF}_{\mathscr{P}}$ a call-by-value λ-calculus where β-reductions fire only if the result is *normalizing*.

Definition 4.4 (Normalizing call-by-value λ-calculus, $\Sigma_{\lambda N}$)**.**

```
o      : Type       Eq  : o -> o -> Type       app : o -> o -> o
v      : Type       var : v -> o               lam : (v -> o) -> o
c_beta : ΠM:o->o,N:o.𝓛^{𝒫^N}_{⟨M,N⟩,(o->o)×o}[Eq (app (lam λx:v.M(var x)) N) (M N)]
```

where the predicate $\mathscr{P}^{\mathtt{N}}$ holds on $\Gamma \vdash_{\Sigma_{\lambda N}} \langle \mathtt{M},\mathtt{N}\rangle \Leftarrow \mathtt{(o->o)\times o}$ if both `M` and `N` have skeletons in $\Lambda_{\Sigma_{\lambda N}}$ whose holes are guarded by a `var` and, moreover, `M N` "normalizes", in the intuitive sense, outside terms guarded by a `var`.

4.4 Elementary Affine Logic

In this section we give a *shallow* encoding of *Elementary Affine Logic* as presented in [2]. This example will exemplify how locks can be used to deal with global syntactic constraints as in the *promotion rule* of Elementary Affine Logic.

Definition 4.5 (Elementary Affine Logic [2])**.** Elementary Affine Logic can be specified by the following rules:

$$\frac{}{A \vdash_{EAL} A}\ (Var) \qquad \frac{\Gamma \vdash_{EAL} B}{\Gamma, A \vdash_{EAL} B}\ (Weak) \qquad \frac{\Gamma, A \vdash_{EAL} B}{\Gamma \vdash_{EAL} A \multimap B}\ (Abst) \qquad \frac{\Gamma \vdash_{EAL} A \quad \Delta \vdash_{EAL} A \multimap B}{\Gamma, \Delta \vdash_{EAL} B}\ (Appl)$$

$$\frac{\Gamma \vdash_{EAL} !A \quad \Delta, !A, \ldots, !A \vdash_{EAL} B}{\Gamma, \Delta \vdash_{EAL} B}\ (Contr) \qquad \frac{A_1, \ldots, A_n \vdash_{EAL} A \quad \Gamma_1 \vdash_{EAL} !A_1 \quad \ldots \quad \Gamma_n \vdash_{EAL} !A_n}{\Gamma_1 \ldots \Gamma_n \vdash_{EAL} !A}\ (Prom)$$

Definition 4.6 ($\mathsf{LLF}_{\mathscr{P}}$ signature Σ_{EAL} for Elementary Affine Logic)**.**

```
o : Type     T : o -> Type     V : o -> Type     —o : o -> o -> o     ! : o -> o
c_appl    : ΠA,B :o. T(A) -> T(A —o B)-> T(B)        c_val : ΠA:o. V(A) -> T(!A)
c_abstr   : ΠA,B :o. Πx:(T(A) -> T(B)) -> 𝒮^Light_{x,T(A)->T(B)}[T(A —o B)]
c_promV_1 : ΠA,B :o. Πx:(T(A —o B)) -> 𝒮^Closed_{x,T(A—o B)}[T(!A) -> V(B)]
c_promV_2 : ΠA,B :o. Πx:(V(A —o B)) -> 𝒮^Closed_{x,V(A—o B)}[T(!A) -> V(B)]
```

where `o` is the type of propositions, —o and ! are the obvious syntactic constructors, `T` is the basic judgement, and $\mathtt{V}(\cdot)$ is an auxiliary judgement. The predicates involved in the locks are defined as follows:

- $\mathit{Light}(\Gamma \vdash_{\Sigma_{EAL}} \mathtt{x} \Leftarrow \mathtt{T(A)} \to \mathtt{T(B)})$ holds iff if `A` is not of the shape `!A` then the bound variable of `x` occurs at most once in the normal form of `x`.
- $\mathit{Closed}(\Gamma \vdash_{\Sigma_{EAL}} \mathtt{x} \Leftarrow \mathtt{T(A)})$ holds iff the skeleton of x contains only free variables of type `o`, *i.e.*no variables of type `T(B)`, for any `B : o`.

A few remarks are mandatory. The promotion rule in [2] is in effect a *family* of natural deduction rules with a growing number of assumptions. Our encoding achieves this via the auxiliary judgement $\mathtt{V}(\cdot)$, the effect of which is self-explanatory. Adequacy for this signature can be achieved only in the format of [19], namely:

Theorem 4.3 (Adequacy for Elementary Affine Logic)**.** *if $A_1, \ldots, A_n$ are the atomic formulas occurring in $B_1, \ldots, B_m, A$, then $B_1 \ldots B_m \vdash_{EAL} A$ iff there exists* `M` *and* $\mathtt{A_1}:\mathtt{o}, \ldots, \mathtt{A_n}:\mathtt{o}, \mathtt{x_1}: \mathtt{T(B_1)}, \ldots, \mathtt{x_m}:\mathtt{T(B_m)} \vdash_{\Sigma_{EAL}} \mathtt{M} \Leftarrow \mathtt{T(A)}$ *(where* `A`*, and* $\mathtt{B_i}$ *represent the encodings of, respectively, A and B_i in* $\mathsf{CLLF}_{\mathscr{P}}$*, for $1 \leq i \leq m$) and all variables* $\mathtt{x_1} \ldots \mathtt{x_m}$ *occurring more than once in* `M` *have type of the shape* $\mathtt{T(B_i)} \equiv \mathtt{T(!C_i)}$ *for some suitable formula* $\mathtt{C_i}$*.*

The check on the context of the Adequacy Theorem is *external* to the system $\mathsf{LLF}_{\mathscr{P}}$, but this is in the nature of results which relate *internal* and *external* concepts. For example, the very concept of $\mathsf{LLF}_{\mathscr{P}}$ context, which appears in any adequacy result, is external to $\mathsf{LLF}_{\mathscr{P}}$. Of course, this check is internalized if the term is closed.

4.5 Square roots of natural numbers in $\mathsf{CLLF}_{\mathscr{P}?}$

It is well-known that logical frameworks based on Constructive Type Theory do not permit definitions of non-terminating functions (*i.e.*, all the functions one can encode in such frameworks are total). One interesting example of $\mathsf{CLLF}_{\mathscr{P}?}$ system is the possibility of reasoning about partial functions by delegating their computation to external oracles, and getting back their possible outputs, via the lock-unlock mechanism of $\mathsf{CLLF}_{\mathscr{P}?}$.

For instance, we can encode natural numbers and compute their square roots by means of the following signature ($\langle \mathtt{x}, \mathtt{y} \rangle$ denotes the encoding of pairs, whose type is denoted by $\sigma \times \tau$, and `fst` and `snd` are the first and second projections, respectively):

```
nat: type   O: nat   S: nat->nat   plus : nat->nat->nat   minus : nat->nat->nat
mult  : nat->nat->nat        sqroot: nat->nat             eval : nat->nat->type
sqrt  : Πx:nat.𝒮^SQRT_{y,nat×σ}[(eval (sqroot x) (fst y))]
```

where `eval` represents the usual evaluation predicate, the variable `y` is a pair and

$$\sigma \equiv (\texttt{eval}\ (\texttt{plus}\ (\texttt{minus}\ \texttt{x}\ (\texttt{mult}\ \texttt{z}\ \texttt{z}))\ (\texttt{minus}\ (\texttt{mult}\ \texttt{z}\ \texttt{z})\ \texttt{x})\ \texttt{0}))$$

and $\mathit{SQRT}(\Gamma \vdash_\Sigma \texttt{y} \Leftarrow \texttt{nat} \times \sigma)$ holds if and only if the first projection of `y` is the minimum number `N` such that $(\texttt{x} \dot{-} \texttt{N} * \texttt{N}) + (\texttt{N} * \texttt{N} \dot{-} \texttt{x}) = \texttt{0}$, where $+$ and * are represented by `plus` and `mult`, while $\dot{-}$ (represented by `minus` in our signature) is defined as follows:

$$\texttt{x} \dot{-} \texttt{y} \stackrel{\Delta}{=} \begin{cases} \texttt{x} - \texttt{y} & \texttt{if}\ \texttt{x} \geq \texttt{y} \\ \texttt{0} & \texttt{otherwise} \end{cases}$$

Thus, the specification of `sqroot` is not explicit in $\mathsf{CLLF}_{\mathscr{P}?}$, since it is implicit in the definition of *SQRT*.

5 Related work

Building a universal framework with the aim of "gluing" different tools and formalisms together is a long standing goal that has been extensively explored in the inspiring work on Logical Frameworks by [4, 27, 35, 31, 7, 5, 26, 28, 29, 17]. Moreover, the appealing monadic structure and properties of the lock/unlock mechanism go back to Moggi's notion of computational monads [25]. Indeed, our system can be seen as a generalization to a family of dependent *lax* operators of Moggi's *partial* λ-calculus [24] and of the work carried out in [8, 23] (which is also the original source of the term "lax"). A correspondence between lax modalities and monads in functional programming was pointed out in [1, 12]. On the other hand, although the connection between constraints and monads in logic programming was considered in the past, *e.g.*, in [26, 10, 9], to our knowledge, our systems are the first attempt to establish a clear correspondence between side conditions and monads in a *higher-order dependent-type theory* and in logical frameworks. Of course, there are a lot of interesting points of contact with other systems in the literature which should be explored. For instance, in [26], the authors introduce a contextual modal logic, where the notion of context is rendered by means of monadic constructs. We only point out that, as we did in our system, they could have also simplified their system by doing away with the `let` construct in favor of a deeper substitution. Schröder-Heister has discussed in a number of papers, see *e.g.* [33, 32], various restrictions and side conditions on rules and on the nature of assumptions that one can add to logical systems to prevent the arising of paradoxes. There are some potential connections between his work and ours. It would be interesting to compare his requirements on side conditions being "closed under substitution" to our notion of *well-behaved* predicate. Similarly, there are commonalities between his distinction between *specific* and *unspecific* variables, and our treatment of free variables in well-behaved predicates. LFSC, presented in [34], is more reminiscent of our approach as "it extends LF to allow side conditions to be expressed using a simple first-order functional programming language". Indeed, the author factors the verifications of side-conditions out of the main proof. The task is delegated to the type checker, which runs the code associated with the side-condition, verifying that it yields the expected output. The proposed machinery is focused on providing improvements for SMT solvers.

References

[1] N. Alechina, M. Mendler, V. De Paiva, E. Ritter. Categorical and Kripke semantics for constructive s4 modal logic. In *Computer Science Logic*, pp. 292–307. Springer, 2001, doi:10.1007/3-540-44802-0_21.

[2] P. Baillot, P. Coppola, U. Dal Lago. Light logics and optimal reduction: Completeness and complexity. In *LICS*, pp. 421–430. IEEE Computer Society, 2007, doi:10.1016/j.ic.2010.10.002.

[3] H.P. Barendregt, E. Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28:321–336, 2002, doi:10.1.1.39.3551.

[4] G. Barthe, H. Cirstea, C. Kirchner, L. Liquori. Pure Pattern Type Systems. In *POPL'03*, pp. 250–261, ACM, doi:10.1.1.298.4555.

[5] M. Boespflug, Q. Carbonneaux, O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. In *PxTP 2012*, v. 878, pp.28–43, 2012, doi:10.1.1.416.1602.

[6] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, J. Van Tassel. Experience with embedding hardware description languages in HOL. In *TPCD*, pp. 129–156. North-Holland, 1992, doi:10.1.1.111.260.

[7] D. Cousineau, G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *TLCA*, v. 4583 of *LNCS*, pp. 102–117. Springer-Verlag, 2007, doi:10.1.1.102.4096.

[8] M. Fairtlough, M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997, doi:10.1.1.22.5812.

[9] M. Fairtlough, M. Mendler, X. Cheng. Abstraction and refinement in higher order logic. In *Theorem Proving in Higher Order Logics*, pp. 201–216. Springer, 2001, doi:10.1.1.29.3515.

[10] M Fairtlough, M. Mendler, M. Walton. First-order lax logic as a framework for constraint logic programming. Technical report, 1997, doi:10.1.1.36.1549.

[11] F. B. Fitch. *Symbolic logic - An Introduction*. New York, 1952, ASIN: B0007DLS2O.

[12] D. Garg, M. C. Tschantz. From indexed lax logic to intuitionistic logic. Tech. rep. CMU, 2008, doi:10.1.1.295.8643.

[13] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998, doi:10.1.1.134.4420.

[14] R. Harper, D. Licata. Mechanizing metatheory in a logical framework. *JFP*, 17:613–673, 2007, doi:10.1017/S0956796807006430.

[15] D. Hirschkoff. Bisimulation proofs for the π-calculus in the Calculus of Constructions. In *TPHOL'97*, n. 1275 in LNCS. Springer, 1997, doi:10.1007/BFb0028392.

[16] F. Honsell. 25 years of formal proof cultures: Some problems, some philosophy, bright future. In *LFMTP'13*, pp. 37–42, ACM, 2013, doi:10.1145/2503887.2503896.

[17] F. Honsell, M. Lenisa, L. Liquori. A Framework for Defining Logical Frameworks. *Volume in Honor of G. Plotkin, ENTCS*, 172:399–436, 2007, doi:10.1016/j.entcs.2007.02.014.

[18] F. Honsell, M. Lenisa, L. Liquori, P. Maksimovic, I. Scagnetto. $\mathsf{LF}_{\mathscr{P}}$: a logical framework with external predicates. In *LFMTP*, pp. 13–22. ACM, 2012, doi:10.1145/2364406.2364409.

[19] F. Honsell, M. Lenisa, L. Liquori, P. Maksimovic, I. Scagnetto. An open logical framework. Accepted for publication in *Journal of Logic and Computation*, doi:10.1093/logcom/ext028.

[20] F. Honsell, L. Liquori, P. Maksimovic, I. Scagnetto. $\mathsf{LLF}_{\mathscr{P}}$: A Logical Framework for modeling External Evidence, Side Conditions, and Proof Irrelevance using Monads. Available at http://www.dimi.uniud.it/scagnett/LLFP_LMCS.pdf.

[21] F. Honsell, L. Liquori, I. Scagnetto. $\mathsf{L}^{\mathsf{ax}}\mathsf{F}$: Side Conditions and External Evidence as Monads. In *MFCS 2014, Part I*, v. 8634 of *LNCS*, pp. 327–339, Budapest, Hungary, August 2014. Springer, doi:10.1007/978-3-662-44522-8_28.

[22] F. Honsell, M. Miculan, I. Scagnetto. π-calculus in (Co)Inductive Type Theories. *Theoretical Computer Science*, 253(2):239–285, 2001, doi:10.1016/S0304-3975(00)00095-5.

[23] M. Mendler. Constrained proofs: A logic for dealing with behavioral constraints in formal hardware verification. In *Designing Correct Circuits*, pp. 1–28. Springer-Verlag, 1991, doi:10.1007/978-1-4471-3544-9_1.

[24] E. Moggi. *The partial lambda calculus*. PhD thesis, University of Edinburgh, 1988, doi:10.1.1.53.8462.

[25] E. Moggi. Computational lambda-calculus and monads. In *LICS 1989*, pp. 14–23. IEEE Press, doi:10.1.1.26.2787.

[26] A. Nanevski, F. Pfenning, B. Pientka. Contextual Modal Type Theory. *ACM TOCL*, 9(3), 2008, doi:10.1145/1352582.1352591.

[27] F. Pfenning, C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *CADE*, v. 1632 of *LNCS*, pp. 202–206. Springer-Verlag, 1999, doi:10.1007/3-540-48660-7_14.

[28] B. Pientka, J. Dunfield. Programming with proofs and explicit contexts. In *PPDP'08*, pp. 163–173, ACM, doi:10.1145/1389449.1389469.

[29] B. Pientka, J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR 2010*, v. 6173 of *LNCS*, pp. 15–21. Springer-Verlag, doi:10.1007/978-3-642-14203-1_2.

[30] D. Prawitz. *Natural Deduction. A Proof Theoretical Study*. Almqvist Wiksell, Stockholm, 1965, ISBN: 978-0486446554.

[31] A. Schack-Nielsen, C. Schürmann. Celf–A logical framework for deductive and concurrent systems (System description). in *Automated Reasoning*, pp. 320–326, Springer, 2008, doi:10.1007/978-3-540-71070-7_28.

[32] P. Schroeder-Heister. Paradoxes and Structural Rules. *Insolubles and consequences : essays in honor of Stephen Read*, pp. 203–211. College Publications, London, 2012, ISBN 978-1-84890-086-8.

[33] P. Schroeder-Heister. Proof-theoretic semantics, self-contradiction, and the format of deductive reasoning. *Topoi*, 31(1):77–85, 2012, doi:10.1007/s11245-012-9119-x.

[34] A. Stump. Proof checking technology for satisfiability modulo theories. In *LFMTP 2008*, v. 228, pp. 121–133, 2009, doi:10.1.1.219.1459.

[35] K. Watkins, I. Cervesato, F. Pfenning, D. Walker. A Concurrent Logical Framework I: Judgments and Properties. Tech. Rep. CMU-CS-02-101, CMU, 2002, doi:10.1.1.14.5484.

An Open Challenge Problem Repository for Systems Supporting Binders

Amy Felty
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
afelty@eecs.uottawa.ca

Alberto Momigliano
Dipartimento di Informatica
Università degli Studi di Milano
Milano, Italy
momigliano@di.unimi.it

Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

A variety of logical frameworks support the use of higher-order abstract syntax in representing formal systems; however, each system has its own set of benchmarks. Even worse, general proof assistants that provide special libraries for dealing with binders offer a very limited evaluation of such libraries, and the examples given often do not exercise and stress-test key aspects that arise in the presence of binders. In this paper we design an open repository *ORBI* (Open challenge problem Repository for systems supporting reasoning with BInders). We believe the field of reasoning about languages with binders has matured, and a common set of benchmarks provides an important basis for evaluation and qualitative comparison of different systems and libraries that support binders, and it will help to advance the field.

1 Introduction

A variety of logical frameworks support the use of higher-order abstract syntax (HOAS) in representing formal systems; however, each system has its own set of benchmarks, often encoding the same object logics with minor differences. Even worse, general proof assistants that provide special libraries for dealing with binders often offer only a very limited evaluation of such libraries, and the examples given often do not exercise and stress-test key aspects that arise in the presence of binders.

The POPLMARK challenge [2] was an important milestone in surveying the state of the art in mechanizing the meta-theory of programming languages. We ourselves proposed several specific benchmarks [8] that are *crafted* to highlight the differences between the designs of various meta-languages with respect to reasoning with and within a context of assumptions, and we compared their implementation in four systems: the logical framework Twelf [23], the dependently-typed functional language Beluga [17, 18], the two-level Hybrid system [6, 15] as implemented on top of Coq and Isabelle/HOL, and the Abella system [10]. Finally, several systems that support reasoning with binders, in particular systems concentrating on modeling binders using HOAS, also provide a large collection of examples and case studies. For example, Twelf's wiki (`http://twelf.org/wiki/Case_studies`), Abella's library (`http://abella-prover.org/examples`), Beluga's distribution, and the Coq implementation of Hybrid (`http://www.site.uottawa.ca/~afelty/HybridCoq/`) contain sets of examples that highlight the many issues surrounding binders.

As the field matures, we believe it is important to be able to systematically and qualitatively evaluate approaches that support reasoning with binders. Having benchmarks is a first step in this direction. In this paper, we propose a common infrastructure for representing challenge problems and a central, Open challenge problem Repository for systems supporting reasoning with BInders (ORBI) for sharing benchmark problems based on the notation we have developed.

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 18–32, doi:10.4204/EPTCS.185.2

ORBI is designed to be a human-readable, easily machine-parsable, uniform, yet flexible and extensible language for writing specifications of formal systems including grammar, inference rules, contexts and theorems. The language directly upholds HOAS representations and is oriented to support the mechanization of benchmark problems in Twelf, Beluga, Abella, and Hybrid, without hopefully precluding other existing or future HOAS systems. At the same time, we hope it also is amenable to translations to systems using other representation techniques such as nominal ones.

We structure the language in two parts:

1. the problem description, which includes the grammar of the object language syntax, inference rules, context schemas, and context relations
2. the logic language, which includes syntax for expressing theorems and directives to ORBI2X[1] tools.

We begin in Sect. 2 with a running example. We consider the untyped lambda-calculus as an object logic (OL), and present the syntax, some judgments, and sample theorems. In Sect. 3, we present ORBI by giving its grammar and explaining how it is used to encode our running example; Sect. 3.1 and Sect. 3.2 present the two parts of this specification as discussed above. We discuss related work in Sect. 4.

We consider the notation that we present here as a first attempt at defining ORBI (Version 0.1), where the goal is to cover the benchmarks considered in [8]. As new benchmarks are added, we are well aware that we will need to improve the syntax and increase the expressive power—we discuss limitations and some possible extensions in Sect. 5.

2 A Running Example

The first question that we face when defining an OL is how to describe well-formed objects. Consider the untyped lambda-calculus, defined by the following grammar:

$$M ::= x \mid \mathsf{lam}\, x.M \mid \mathsf{app}\, M_1\, M_2.$$

To capture additional information that is often useful in proofs, such as when a given term is *closed*, it is customary to give inference rules in natural deduction style for well-formed terms using hypothetical and parametric judgments. However, it is often convenient to present hypothetical judgments in a *localized* form, reducing some of the ambiguity of the two-dimensional natural deduction notation, and providing more structure. We therefore introduce an *explicit* context for bookkeeping, since when establishing properties about a given system, it allows us to consider the variable case(s) separately and to state clearly when considering closed objects, i.e., an object in the empty context. More importantly, while structural properties of contexts are implicitly present in the natural deduction presentation of inference rules (where assumptions are managed informally), the explicit context presentation makes them more apparent and highlights their use in reasoning about contexts.

$$\frac{\mathsf{is\_tm}\ x \in \Gamma}{\Gamma \vdash \mathsf{is\_tm}\ x}\ tm_v \qquad \frac{\Gamma, \mathsf{is\_tm}\ x \vdash \mathsf{is\_tm}\ M}{\Gamma \vdash \mathsf{is\_tm}\ (\mathsf{lam}\, x.M)}\ tm_l \qquad \frac{\Gamma \vdash \mathsf{is\_tm}\ M_1 \qquad \Gamma \vdash \mathsf{is\_tm}\ M_2}{\Gamma \vdash \mathsf{is\_tm}\ (\mathsf{app}\, M_1\, M_2)}\ tm_a$$

[1] Following TPTP's nomenclature [25], we call "ORBI2X" any tool taking an ORBI specification as input; for example, the translator for Hybrid described in [13] turns syntax, inference rules, and context definitions of ORBI into input to the Coq version of Hybrid, and it is designed so that it can be adapted fairly directly to output Abella scripts.

Traditionally, a context of assumptions is characterized as a sequence of formulas $A_1, A_2, \ldots, A_n$ listing its elements separated by commas [12, 19]. In [7], we argue that this is not expressive enough to capture the structure present in contexts, especially when mechanizing object logics, and we define context *schemas* to introduce the required extra structure:

$$\begin{array}{rrcl} \text{Atom} & A & & \\ \text{Block of declarations} & D & ::= & A \mid D;A \\ \text{Context} & \Gamma & ::= & \cdot \mid \Gamma, D \\ \text{Schema} & S & ::= & D_s \mid D_s + S \end{array}$$

A context is a sequence of declarations D where a declaration is a block of individual atomic assumptions separated by ';'.[2] The ';' binds tighter than ','. We treat contexts as ordered, i.e., later assumptions in the context may depend on earlier ones, but not vice versa—this in contrast to viewing contexts as multi-sets. Just as types classify terms, a *schema* will classify meaningful structured sequences. A schema consists of declarations D_s, where we use the subscript s to indicate that the declaration occurring in a concrete context having schema S may be an *instance* of D_s. We use $+$ to denote the alternatives in a context schema. For well-formed terms, contexts have a simple structure where each block contains a single atom, expressed as the following schema declaration:

$$S_x := \mathsf{is\_tm}\ x.$$

We write Φ_x to represent a context *satisfying* schema S_x (and similarly for other context schemas appearing in this paper). Informally, this means that Φ_x has the form $\mathsf{is\_tm}\ x_1, \ldots, \mathsf{is\_tm}\ x_n$ where $n \geq 0$ and $x_1, \ldots, x_n$ are distinct variables. (See [7] for a more formal account.)

For our running example, we consider two more simple judgments. The first is *algorithmic equality* for the untyped lambda-calculus, written $(\mathsf{aeq}\ M\ N)$. We say that two terms are algorithmically equal provided they have the same structure with respect to the constructors.

$$\frac{\mathsf{aeq}\ x\ x \in \Gamma}{\Gamma \vdash \mathsf{aeq}\ x\ x}\ ae_v \qquad \frac{\Gamma, \mathsf{is\_tm}\ x; \mathsf{aeq}\ x\ x \vdash \mathsf{aeq}\ M\ N}{\Gamma \vdash \mathsf{aeq}\ (\mathsf{lam}\, x.M)\ (\mathsf{lam}\, x.N)}\ ae_l \qquad \frac{\Gamma \vdash \mathsf{aeq}\ M_1\ N_1 \quad \Gamma \vdash \mathsf{aeq}\ M_2\ N_2}{\Gamma \vdash \mathsf{aeq}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)}\ ae_a$$

The second is *declarative equality* written $(\mathsf{deq}\ M\ N)$, which includes versions of the above three rules called de_v, de_l, and de_a, where aeq is replaced by deq everywhere, plus reflexivity, symmetry and transitivity shown below.[3]

$$\frac{}{\Gamma \vdash \mathsf{deq}\ M\ M}\ de_r \qquad \frac{\Gamma \vdash \mathsf{deq}\ N\ M}{\Gamma \vdash \mathsf{deq}\ M\ N}\ de_s \qquad \frac{\Gamma \vdash \mathsf{deq}\ M\ L \quad \Gamma \vdash \mathsf{deq}\ L\ N}{\Gamma \vdash \mathsf{deq}\ M\ N}\ de_t$$

These judgments give rise to the following schema declarations:

$$\begin{array}{rcl} S_{xa} & := & \mathsf{is\_tm}\ x; \mathsf{aeq}\ x\ x \\ S_{xd} & := & \mathsf{is\_tm}\ x; \mathsf{deq}\ x\ x \\ S_{da} & := & \mathsf{is\_tm}\ x; \mathsf{deq}\ x\ x; \mathsf{aeq}\ x\ x \end{array}$$

[2] This is an oversimplification, since there are well-known specifications where contexts have more structure, see the solution to the POPLMARK challenge in [16] and the examples in [26]. In fact, those are already legal ORBI specs.

[3] We acknowledge that this definition of declarative equality has a degree of redundancy: the assumption $\mathsf{deq}\ x\ x$ in rule de_l is not needed, since rule de_r plays the variable role. However, this formulation exhibits issues, such as *context subsumption*, that would otherwise require more complex benchmarks.

The first two come directly from the ae_l and de_l rules where declaration blocks come in pairs. The third combines the two, and is used below in stating one of the example theorems.

When stating properties, we often need to relate two judgments to each other, where each one has its own context. For example, we may want to prove statements such as "if $\Phi_x \vdash J_1$ then $\Phi_{xa} \vdash J_2$." The proofs in [8] use two approaches.[4] In the first, the statement is reinterpreted in the *smallest* context that collects all relevant assumptions; we call this the *generalized context* approach (G). The above statement becomes "if $\Phi_{xa} \vdash J_1$ then $\Phi_{xa} \vdash J_2$." As an example theorem, we consider the completeness of declarative equality with respect to algorithmic equality, of which we only show the interesting left-to-right direction.

Theorem 2.1 (Completeness, G Version)

Admissibility of Reflexivity *If* $\Phi_{xa} \vdash \mathsf{is\_tm}\ M$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ M$.

Admissibility of Symmetry *If* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ N$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ N\ M$.

Admissibility of Transitivity *If* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ N$ *and* $\Phi_{xa} \vdash \mathsf{aeq}\ N\ L$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ L$.

Main Theorem *If* $\Phi_{da} \vdash \mathsf{deq}\ M\ N$ *then* $\Phi_{da} \vdash \mathsf{aeq}\ M\ N$.

In the second approach, we state how two (or more) contexts are related via context relations. For example, the following relation captures the fact that $\mathsf{is\_tm}\ x$ will occur in Φ_x in sync with an assumption block containing $\mathsf{is\_tm}\ x; \mathsf{aeq}\ x\ x$ in Φ_{xa}.

$$\frac{}{\cdot \sim \cdot} \qquad \frac{\Phi_x \sim \Phi_{xa}}{\Phi_x, \mathsf{is\_tm}\ x \sim \Phi_{xa}, \mathsf{is\_tm}\ x; \mathsf{aeq}\ x\ x}$$

Similarly, we can define $\Phi_{xa} \sim \Phi_{xd}$.

$$\frac{}{\cdot \sim \cdot} \qquad \frac{\Phi_{xa} \sim \Phi_{xd}}{\Phi_{xa}, \mathsf{is\_tm}\ x; \mathsf{aeq}\ x\ x \sim \Phi_{xd}, \mathsf{is\_tm}\ x; \mathsf{deq}\ x\ x}$$

We call this the *context relations* approach (R). The theorems are then typically stated as: "if $\Phi_x \vdash J_1$ and $\Phi_x \sim \Phi_{xa}$ then $\Phi_{xa} \vdash J_2$." We can then revisit the completeness theorem for algorithmic equality together with the necessary lemmas as follows.

Theorem 2.2 (Completeness, R Version)

Admissibility of Reflexivity *Assume* $\Phi_x \sim \Phi_{xa}$. *If* $\Phi_x \vdash \mathsf{is\_tm}\ M$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ M$.

Admissibility of Symmetry *If* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ N$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ N\ M$.

Admissibility of Transitivity *If* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ N$ *and* $\Phi_{xa} \vdash \mathsf{aeq}\ N\ L$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ L$.

Main Theorem *Assume* $\Phi_{xa} \sim \Phi_{xd}$. *If* $\Phi_{xd} \vdash \mathsf{deq}\ M\ N$ *then* $\Phi_{xa} \vdash \mathsf{aeq}\ M\ N$.

3 ORBI

ORBI aims to provide a common framework for systems that support reasoning with binders. Currently, our design is geared towards systems supporting HOAS, where there are (currently) two main approaches. On one side of the spectrum we have systems that implement various dependently-typed

[4] In proofs on paper, the differences between the two approaches usually do not appear; they are present in the details that are left implicit, but must be made explicit when mechanizing proofs. For example, on-paper versions of the admissibility of reflexivity that make these distinctions explicit appear in [7] as proofs of Theorems 7 and 8.

calculi. Such systems include Twelf, Beluga, and Delphin [20]. All these systems also provide, to various degrees, built-in support for reasoning modulo structural properties of a context of assumptions. These systems support inductive reasoning over terms as well as rules. Often it is more elegant in these systems to state theorems using the G-version [8].

On the other side there are systems based on a proof-theoretic foundation, which typically follow a two-level approach: they implement a specification logic (SL) inside a higher-order logic or type theory. Hypothetical judgments of object languages are modeled using implication in the SL and parametric judgments are handled via (generic) universal quantification. Contexts are commonly represented explicitly as lists or sets in the SL, and structural properties are established separately as lemmas. For example substituting for an assumption is justified by appealing to the cut-admissibility lemma of the SL. These lemmas are not directly and intrinsically supported through the SL, but may be integrated into a system's automated proving procedures, usually via tactics. Induction is usually only supported on derivations, but not on terms. Systems following this philosophy include Hybrid and Abella. Often these systems are better suited to proving R-versions of theorems.

The desire for ORBI to cater to both type and proof theoretic frameworks requires an almost impossible balancing act between the two views. For example, contexts are first-class and part of the specification language in Beluga; in Twelf, schemas for contexts are part of the specification language, which is an extension of LF, but users cannot explicitly quantify over contexts and manipulate them as first-class objects; in Abella and Hybrid, contexts are (pre)defined using inductive definitions on the reasoning level. We will describe next our common infrastructure design, directives, and guidelines that allow us to cater to existing systems supporting HOAS.

3.1 Problem Description in ORBI

ORBI's language for defining the grammar of an object language together with inference rules is based on the logical framework LF; pragmatically, we have adopted the concrete syntax of LF specifications in Beluga, which is almost identical to Twelf's. The advantage is that specifications can be directly parsed and more importantly type checked by Beluga, thereby eliminating many syntactically correct but meaningless expressions.

Object languages are written according to the EBNF grammar in Fig. 1, which uses certain standard conventions: `{a}` means repeat a production zero or more times, and comments in the grammar are enclosed between `(*` and `*)`. The token `id` refers to identifiers starting with a lower or upper case letter. These grammar rules are basically the standard ones used both in Twelf and Beluga and we do not discuss them in detail here. We only note that while the presented grammar permits general dependent types up to level *n*, ORBI specifications will only use level 0 and level 1. Intuitively, specifications at level 0 define the syntax of a given object language, while specifications at level 1 (i.e., type families that are indexed by terms of level 0) describe the judgments and rules for a given OL. We exemplify the grammar relative to the example of algorithmic vs. declarative equality. For more example specifications, we refer the reader to our survey paper [8] or to `https://github.com/pientka/ORBI`.[5]

Syntax An ORBI file starts in the `Syntax` section with the declaration of the constants used to encode the syntax of the OL in question, here untyped lambda-terms, which are introduced with the declarations:

[5] The observant reader will have noticed that ORBI's concrete syntax for schemas differs from the one that we have presented in Sect. 2, in so much that blocks are separated by commas and not by semi-colons. This is forced on us by our choice to re-use Beluga's parsing and checking tools.

```
sig      ::= {decl                        (* declaration *)
           | s_decl}                      (* schema declaration *)

decl     ::= id ":" tp "."                (* constant declaration *)
           | id ":" kind "."              (* type declaration *)

op_arrow ::= "->" | "<-"                  (* A <- B same as B -> A *)

kind     ::= type
           | tp op_arrow kind             (* A -> K *)
           | "{" id ":" tp "}" kind       (* Pi x:A.K *)

tp       ::= id {term}                    (* a M1 ... M2 *)
           | tp op_arrow tp
           | "{" id ":" tp "}" tp         (* Pi x:A.B *)

term     ::= id                           (* constants, variables *)
           | "\" id "." term              (* lambda x. M *)
           | term term                    (* M N *)

s_decl    ::= schema s_id "=" alt_blk ";"

s_id      ::= id

alt_blk   ::= blk {"+" blk}

blk       ::= block id ":" tp {"," id ":" tp}
```

Figure 1: ORBI grammar for syntax, judgments, inference rules, and context schemas

```
%% Syntax
tm: type.
app: tm -> tm -> tm.
lam: (tm -> tm) -> tm.
```

The declaration introducing type `tm` along with those of the constructors `app` and `lam` fully specify the syntax of OL terms. We represent binders in the OL using binders in the HOAS meta-language. Hence the constructor `lam` takes in a function of type `tm -> tm`. For example, the OL term $(\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\, x\, y)$ is represented as `lam (\x. lam (\y. app x y))`, where "\" is the binder of the metalanguage. Bound variables found in the object language are not explicitly represented in the meta-language.

Judgments and Rules These are introduced as LF type families (predicates) in the `Judgments` section followed by object-level inference rules for these judgments in the `Rules` section. In our running example, we have two judgments:

```
%% Judgments
aeq: tm -> tm -> type.
```

```
deq: tm -> tm -> type.
```

Consider first the inference rule for algorithmic equality for application, where the ORBI text is a straightforward encoding of the rule:

```
ae_a: aeq M1 N1 -> aeq M2 N2
   -> aeq (app M1 M2) (app N1 N2).
```

$$\frac{\Gamma \vdash \mathsf{aeq}\ M_1\ N_1 \quad \Gamma \vdash \mathsf{aeq}\ M_2\ N_2}{\Gamma \vdash \mathsf{aeq}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)}\ ae_a$$

Uppercase letters such as `M1` denote schematic variables, which are implicitly quantified at the outermost level, namely `{M1:tm}`, as is commonly done for readability purposes in Twelf and Beluga.
The binder case is more interesting:

```
ae_l: ({x:tm} aeq x x -> aeq (M x) (N x))
   -> aeq (lam (\x. M x)) (lam (\x. N x)).
```

$$\frac{\Gamma, \mathsf{is\_tm}\ x; \mathsf{aeq}\ x\ x \vdash \mathsf{aeq}\ M\ N}{\Gamma \vdash \mathsf{aeq}\ (\mathsf{lam} x.M)\ (\mathsf{lam} x.N)}\ ae_l$$

We view the is_tm x assumption as the parametric assumption `x:tm`, while the hypothesis aeq x x (and its scoping) is encoded within the embedded implication `aeq x x -> aeq (M x) (N x)` in the current (informal) signature augmented with the dynamic declaration for `x`. As is well known, parametric assumptions and embedded implication are unified in the type-theoretic view. Note that the "variable" case, namely rule ae_v, is folded inside the binder case. We list here the rest of the `Rules` section:

```
%% Rules
de_a: deq M1 N1 -> deq M2 N2 -> deq (app M1 M2) (app N1 N2).
de_l: ({x:tm} deq x x -> deq (M x) (N x))
        -> deq (lam (\x. M x)) (lam (\x. N x)).
de_r: deq M M.
de_s: deq N M -> deq M N.
de_t: deq M L -> deq L N -> deq M N.
```

Schemas A schema declaration `s_decl` is introduced using the keyword `schema`. A `blk` consists of one or more declarations and `alt_blk` describes *alternating* schemas. For example, schemas mentioned in Sect. 2 appear in the `Schemas` section as:

```
%% Schemas
schema xG =  block (x:tm);
schema xaG = block (x:tm, u:aeq x x);
schema xdG = block (x:tm, u:deq x x);
schema daG = block (x:tm, u:deq x x, v:aeq x x);
```

To illustrate alternatives in contexts, consider extending our OL to the polymorphically typed lambda-calculus, which includes a new type `tp` in the `Syntax` section, and a new judgment:

```
atp: tp -> tp -> type.
```

representing equality of types in the `Judgments` section (as well as type constructors and rules for well-formed types and type equality, omitted here). With this extension, the following two examples replace the first two schemas in the `Schemas` section.

```
schema xG  = block (x:tm) + block (a:tp);
schema xaG = block (x:tm, u:aeq x x) + block (a:tp, v:atp a a);
```

While we type-check the schema definitions using an extension of the LF type checker (as implemented in Beluga), we do not verify that the given schema definition is meaningful with respect to the specification of the syntax and inference rules; in other words, we do not perform "world checking" in Twelf lingo.

Definitions So far we have considered the specification language for encoding formal systems. ORBI also supports declaring inductive definitions for specifying context relations. We start with the grammar for inductive definitions (Fig. 2). Although we plan to provide syntax for specifying more general inductive definitions, in this version of ORBI we *only* define *context relations* inductively, that is *n*-ary predicates between contexts of some given schemas. Hence the base predicate is of the form `id {ctx}` relating different contexts. For example, the `Definitions` section defines the relations $\Phi_x \sim \Phi_{xa}$ and

```
def_dec  ::= "inductive" id ":" r_kind "=" def_body ";"

r_kind   ::= "prop"
           | "{" id ":" s_id "}" r_kind

def_body ::= "|" id ":" def_prp {def_body}

def_prp  ::= id {ctx}
           | def_prp "->" def_prp

ctx      ::= [] | [id] | ctx "," id ":" blk
```

Figure 2: ORBI grammar for inductive definitions describing context relations

$\Phi_{xa} \sim \Phi_{xd}$. To illustrate, only the former is shown below.

```
%% Definitions
inductive Rxa : {g:xG} {h:xaG} prop =
| Rxa_nl: Rxa [] []
| Rxa_cs: Rxa [g] [h]
          -> Rxa [g, b:block (x:tm)] [h, b: block (x:tm, u:aeq x x)];
```

This kind of relation can be translated fairly directly to inductive n-ary predicates in systems supporting the proof-theoretic view. In the type-theoretic framework underlying Beluga, inductive predicates relating contexts correspond to recursive data types indexed by contexts; in fact ORBI adopts Beluga's concrete syntax, so as to directly type-check those definitions as well. Twelf's type theoretic framework, however, is not rich enough to support inductive definitions.

3.2 Theorems and Directives in ORBI

While the elements of an ORBI specification detailed in the previous subsection were relatively easy to define in a manner that is well understood by all the different systems we are targeting, we illustrate in this subsection those elements that are harder to describe uniformly due to the different treatment and meaning of contexts in the different systems.

Theorems We list the grammar for theorems in Fig. 3. Our reasoning language includes a category `prp` that specifies the logical formulas we support. The base predicates include `false,true`, term equality, atomic predicates of the form `id {ctx}`, which are used to express context relations, and predicates of the form `[ctx |- J]`, which represent judgments of an object language within a given context. Connectives and quantifiers include implication, conjunction, disjunction, universal and existential quantification over terms, and universal quantification over context variables.

```
thm      ::= "theorem" id ":" prp ";"

prp      ::= id {ctx}                       (* Context relation *)
           | "[" ctx  "|-" id {term} "]"    (* Judgment in a context *)
           | term "=" term                  (* Term equality *)
           | false                          (* Falsehood *)
           | true                           (* Truth *)
           | prp "&" prp                    (* Conjunction *)
           | prp "||" prp                   (* Disjunction *)
           | prp "->" prp                   (* Implication *)
           | quantif prp                    (* Quantification *)

quantif  ::= "{" id ":" s_id "}"            (* universal over contexts *)
           | "{" id ":" tp "}"              (* universal over terms *)
           | "<" id ":" tp ">"              (* existential over terms *)
```

Figure 3: ORBI grammar for theorems

To illustrate, the reflexivity lemmas and completeness theorems for both the G and R versions as they appear in the `Theorems` section are shown below. These theorems are a straightforward encoding of those stated in Sect. 2.

```
%% Theorems
theorem reflG: {h:xaG}{M:tm} [h |- aeq M M];
theorem ceqG:  {g:daG}{M:tm}{N:tm} [g |- deq M N] -> [g |- aeq M N];

theorem reflR: {g:xG}{h:xaG}{M:tm}        Rxa [g] [h] -> [h |- aeq M M];
theorem ceqR:  {g:xdG}{h:xaG}{M:tm}{N:tm} Rda [g] [h] ->
                                               [g |- deq M N] -> [h |- aeq M N];
```

As mentioned, we do not type-check theorems; in particular, we do not define the meaning of `[ctx |- J]`, since several interpretations are possible. In Beluga, every judgment `J` must be meaningful within the given context `ctx`; in particular, *terms* occurring in the judgment `J` must be meaningful in `ctx`. As a consequence, both parametric and hypothetical assumptions relevant for establishing the proof of `J` must be contained in `ctx`. Instead of the local context view adopted in Beluga, Twelf has one global ambient context containing all relevant parametric and hypothetical assumptions. Systems based on proof-theory such as Hybrid and Abella distinguish between assumptions denoting eigenvariables (i.e., parametric assumptions), which live in a global ambient context and proof assumptions (i.e., hypothetical assumptions), which live in the context `ctx`. While users of different systems understand how to interpret `[ctx |- J]`, reconciling these different perspectives in ORBI is beyond the scope of

this paper. Thus for the time being, we view theorem statements in ORBI as a kind of *comment*, where it is up to the user of a particular system to determine how to translate them.

Directives In ORBI, *directives* are comments that help the ORBI2X tools to generate target representations of the ORBI specifications. The idea is reminiscent of what *Ott* [24] does to customize certain declarations, e.g., the representation of variables, to the different programming languages/proof assistants it supports. The grammar for directives is listed in Fig. 4.

```
dir       ::=  '%' sy_set what decl {"," decl} {dest} '.'
             | '%%' sepr '.'

sy_id     ::=  hy | ab | bel | tw

sy_set    ::= '[' sy_id {',' sy_id} ']'

what      ::=  wf | explicit | implicit

dest      ::= 'in' ctx | 'in' s_id | 'in' id

sepr      ::=  Syntax | Judgments | Rules | Schemas | Definitions
             | Directives | Theorems
```

Figure 4: ORBI grammar for directives

The `sepr` directives, such as `Syntax`, are simply means to structure ORBI specifications. Most of the other directives that we consider in this version of ORBI are dedicated to help the translations into proof-theoretical systems, although we also include some to facilitate the translation of theorems to Beluga. The set of directives is not intended to be complete and the meaning of directives is system-specific. The directives `wf` and `explicit` are concerned with the asymmetry in the proof-theoretic view between declarations that give typing information, e.g., `tm:type`, and those expressing judgments, e.g., `aeq:tm -> tm -> type`. In Abella and Hybrid, the former may need to be reified in a judgment, in order to show that judgments preserve the well-formedness of their constituents, as well as to provide induction on the structure of terms; yet, in order to keep proofs compact and modular, we want to minimize this reification and only include them where necessary. The `Directives` section of our sample specification includes, for example,

```
% [hy,ab] wf tm.
```

which refers to the first line of the `Syntax` section where `tm` is introduced, and indicates that we need a predicate (e.g., `is_tm`) to express well-formedness of terms of type `tm`. Formulas expressing the definition of this predicate are automatically generated from the declarations of the constructors `app` and `lam` with their types.

The keyword `explicit` indicates when such well-formedness predicates should be included in the translation of the declarations in the `Rules` section. For example, the following formulas both represent possible translations of the `ae_l` rule to proof-theoretic systems. We use Abella's concrete syntax to exemplify:

```
aeq (lam M) (lam N) :- pi x\ is_tm x => aeq x x => aeq (M x) (N x).
aeq (lam M) (lam N) :- pi x\ aeq x x => aeq (M x) (N x).
```

where the typing information is explicit in the first and implicit in the second. By default, we choose the latter, that is well-formed judgments are assumed to be *implicit*, and require a directive if the former is desired. Consider, for example, that we want to conclude that whenever a judgment is provable, the terms in it are well-formed, e.g., if `aeq M N` is provable, then so are `is_tm M` and `is_tm N`. Such a lemma is indeed provable in Abella and Hybrid from the *implicit* translation of the rules for `aeq`. Proving a similar lemma for the `deq` judgment, on the other hand, requires some strategically placed explicit well-formedness information. In particular, the two directives:

```
% [hy,ab] explicit (x : tm) in de_l.
% [hy,ab] explicit (M : tm) in de_r.
```

require the clauses `de_l` and `de_r` to be translated to the following formulas:

```
deq (lam M) (lam N) :- pi x\ is_tm x => deq x x => deq (M x) (N x).
deq M M :- is_tm M.
```

The case for schemas is analogous. In the systems based on proof-theoretic approaches, contexts are typically represented using lists and schemas are translated to unary inductive predicates that verify that these lists have a particular regular structure. We again leave typing information implicit in the translation unless a directive is included. For example, the `xaG` schema with no associated directive will be translated to the following inductive definition in Abella:

```
Define aG : olist -> prop by
  xaG nil;
  nabla x, xaG (aeq x x :: As) := xaG As.
```

The directive `% [hy,ab] explicit (x : tm) in daG` will yield this Hybrid definition:

```
Inductive daG : list atm -> Prop :=
| nil_da : daG nil
| cns_da : forall (Gamma:list atm) (x:uexp),
    proper x -> daG Gamma -> daG (is_tm x :: deq x x :: aeq x x :: Gamma).
```

Similarly, directives in context relations, such as:

```
% [hy,ab] explicit (x : tm) in g in Rxa.
```

also state which well-formedness annotations to make explicit in the translated version. In this case, when translating the definition of `Rxa` in the `Definitions` section, they are to be kept in `g`, but skipped in `h`.

Keeping in mind that we consider the notion of directive *open* to cover other benchmarks and different systems, we offer some speculation about directives that we may need to translate theorems for the examples and systems that we are considering. For example, theorem `reflG` is proven by induction over `M`. As a consequence, `M` must be explicit.

```
% [hy,ab,bel] explicit (M : tm) in h in reflG.
```

The ORBI2Hybrid and ORBI2Abella tools will interpret the directive by adding an explicit assumption, as illustrated by the result of the ORBI2Abella translation:

```
forall H M, xaG H -> {H |- is_tm M} -> {H |- aeq M M}.
```

In Beluga, the directive is interpreted as:

```
{h:xaG} {M:[h |- tm]} [h |- aeq M M].
```

where M will have type tm in the context h. Moreover, since the term M is used in the judgment aeq within the context h, we associate M with an identity substitution, which is not displayed. In short, the directive allows us to lift the type specified in ORBI to a contextual type that is meaningful in Beluga. In fact, Beluga always needs additional information on how to interpret terms—are they closed or can they depend on a given context? For translating symG for example, we use the following directive to indicate the dependence on the context:

```
% [bel] implicit (M : tm), (N : tm) in h in symG.
```

3.3 Guidelines

In addition, we introduce a set of *guidelines* for ORBI specification writers, with the goal of helping translators generate output that is more likely to be accepted by a specific system. ORBI 0.1 includes four such guidelines, which are motivated by the desire to avoid putting too many constraints in the grammar rules. First, as we have seen in our examples, we use as a convention that free variables which denote schematic variables in rules are written using upper case identifiers; we use lower case identifiers for eigenvariables in rules and for context variables. Second, while the grammar does not restrict what types we can quantify over, the intention is that we quantify over types of level-0, i.e., objects of the syntax level, only. Third, in order to more easily accommodate systems without dependent types, Pi should not be used when writing non-dependent types; an arrow should be used instead. (In LF, for example, A -> B is an abbreviation for Pi x:A.B for the case when x does not occur in B. Following this guideline means favoring this abbreviation whenever it applies.) Fourth, when writing a context (grammar ctx), distinct variable names should be used in different blocks.

4 Related Work

Our approach to structuring contexts of assumptions takes its inspiration from Martin-Löf's theory of judgments, especially in the way it has been realized in Edinburgh LF. However, our formulation owes more to Beluga's type theory, where contexts are first-class citizens, than to the notion of *regular world* in Twelf.

The creation and sharing of a library of benchmarks has proven to be very beneficial to the field it represents. The brightest example is *TPTP* [25], whose influence on the development, testing and evaluation of automated theorem provers cannot be underestimated. Clearly our ambitions are much more limited. We have also taken some inspiration from its higher-order extension *THF0* [3], in particular in its construction in stages.

The success of TPTP has spurred other benchmark suites in related subjects, see for example *SATLIB* [14]; however, the only one concerned with induction is the *Induction Challenge Problems* (http://www.cs.nott.ac.uk/~lad/research/challenges), a collection of examples geared to the *automation* of inductive proof. The benchmarks are taken from arithmetic, puzzles, functional programming specifications, etc. and as such have little connection with our endeavor. On the other hand, the examples mentioned earlier coming from Twelf's wiki, Abella's library, Beluga's distribution, and Hybrid's web page contain a set of examples that highlight the issues around binders. As such they are prime candidates to be included in ORBI.

Other projects have put forward LF as a common ground: the goal of *Logosphere*'s (http://www.logosphere.org) was the design of a representation language for logical formalisms, individual theories, and proofs, with an interface to other theorem proving systems that were somewhat connected, but

the project never materialized. *SASyLF* [1] originated as a tool to teach programming language theory: the user specifies the syntax, judgments, theorems *and* proofs thereof (albeit limited to *closed* objects) in a paper-and-pencil HOAS-friendly way and the system converts them to totality-checked Twelf code. The capability to express and share proofs is of obvious interest to us, although such proofs, being a literal proof verbalization of the corresponding Twelf type family, are irremediably verbose. Finally, work on modularity in LF specifications [21] is of critical interest to give more structure to ORBI files.

Why3 (`http://why3.lri.fr`) is a software verification platform that intends to provide a front-end to third-party theorem provers, from proof assistants such as Coq to SMT-solvers. To this end Why3 provides a first-order logic with rank-1 polymorphism, recursive definitions, algebraic data types and inductive predicates [9], whose specifications are then translated to the several systems that Why3 supports. Typically, those translations are forgetful, but sometimes, e.g., with respect to Coq, they add some annotations, for example to ensure non-emptiness of types. Although we are really not in the same business as Why3, there are several ideas that are relevant; to name one, the notion of a *driver*, that is, a configuration file to drive transformations specific to a system. Moreover, Why3 provides an API for users to write and implement their own drivers and transformations.

Ott [24] is a highly engineered tool for "working semanticists," allowing them to write programming language definitions in a style very close to paper-and-pen specifications; then those are compiled into LaTeX and, more interestingly, into proof assistant code, currently supporting Coq, Isabelle/HOL, and HOL. Ott's metalanguage is endowed with a rich theory of binders, but at the moment it favors the "concrete" (non α-quotiented) representation, while providing support for the nameless representation for a single binder. Conceptually, it would be natural to extend Ott to generate ORBI code, as a bridge for Ott to support HOAS-based systems. Conversely, an ORBI user would benefit from having Ott as a front-end, since the latter view of grammar and judgment seems at first sight general enough to support the notion of schema and context relation.

In the category of environments for programming language descriptions, we mention *PLT-Redex* [5] and also the *K* framework [22]. In both, several large-scale language descriptions have been specified and tested. However, none of those systems has any support for binders, let alone context specifications, nor can any meta-theory be formally verified.

Finally, there is a whole research area dedicated to the handling and sharing of mathematical content (*MMK* `http://www.mkm-ig.org`) and its representation (*OMDoc* `https://trac.omdoc.org/OMDoc`), which is only very loosely connected to our project.

5 Conclusion

We have presented the preliminary design of a language, and more generally, of a common infrastructure for representing challenge problems for HOAS-based logical frameworks. The common notation allows us to express the syntax of object languages that we wish to reason about, as well as the context schemas, the judgments and inference rules, and the statements of benchmark theorems.

We strongly believe that the field has matured enough to benefit from the availability of a set of benchmarks on which qualitative and hopefully quantitative comparison can be carried out. We hope that ORBI will foster sharing of examples in the community and provide a common set of examples. We also see our benchmark repository as a place to collect and propose "open" challenge problems to push the development of meta-reasoning systems.

The challenge problems also play a role in allowing us, as designers and developers of logical frameworks, to highlight and explain how the design decisions for each individual system lead to differences

in using them in practice. Additionally, our benchmarks aim to provide a better understanding of what practitioners should be looking for, as well as help them foresee what kind of problems can be solved elegantly and easily in a given system, and more importantly, why this is the case. Therefore the challenge problems provide guidance for users and developers in better comprehending differences and limitations. Finally, they serve as an excellent regression suite.

The description of ORBI presented here is best thought of as a stepping stone towards a more comprehensive specification language, much as *THF0* [3] has been extended to the more expressive formalism THF_i, adding for instance, rank-1 polymorphism. Many are the features that we plan to provide in the near future, starting from general (monotone) *(co)inductive* definitions; currently we only relate contexts, while it is clearly desirable to relate arbitrary well-typed terms, as shown for example in [4] and [11] with respect to normalization proofs. Further, it is only natural to support infinite objects and behavior. However, full support for (co)induction is a complex matter, as it essentially entails fully understanding the relationship between the proof-theory behind Abella and Hybrid and the type theory of Beluga. Once this is in place, we can "rescue" ORBI theorems from their current status as comments and even include proof sketches in ORBI.

Clearly, there is a significant amount of implementation work ahead, mainly on the ORBI2X tools side, but also on the practicalities of the benchmark suite. Finally, we would like to open up the repository to other styles of formalization such as nominal, locally nameless, etc.

References

[1] Jonathan Aldrich, Robert J. Simmons & Key Shin (2008): *SASyLF: An Educational Proof Assistant for Language Theory*. In: *International Workshop on Functional and Declarative Programming in Education*, ACM Press, pp. 31–40, doi:10.1145/1411260.1411266.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The* POPLMARK *Challenge*. In: *Eighteenth International Conference on Theorem Proving in Higher Order Logics, LNCS* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[3] Christoph Benzmüller, Florian Rabe & Geoff Sutcliffe (2008): *THF0—The Core of the TPTP Language for Higher-Order Logic*. In: *Fourth International Joint Conference on Automated Reasoning, LNCS* 5195, Springer, pp. 491–506, doi:10.1007/978-3-540-71070-7_41.

[4] Andrew Cave & Brigitte Pientka (2012): *Programming with Binders and Indexed Data-Types*. In: *Thirty-Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, pp. 413–424, doi:10.1145/2103656.2103705.

[5] Matthias Felleisen, Robert Bruce Findler & Matthew Flatt (2009): *Semantics Engineering with PLT Redex*. The MIT Press.

[6] Amy P. Felty & Alberto Momigliano (2012): *Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. *Journal of Automated Reasoning* 48(1), pp. 43–105, doi:10.1007/s10817-010-9194-x.

[7] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1—A Common Infrastructure for Benchmarks*. *CoRR* abs/1503.06095. Available at http://arxiv.org/abs/1503.06095.

[8] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2—A Survey*. *Journal of Automated Reasoning*. (to appear).

[9] Jean-Christophe Filliâtre (2013): *One Logic To Use Them All*. In: *Twenty-Fourth International Conference on Automated Deduction, LNCS* 7898, Springer, pp. 1–20, doi:10.1007/978-3-642-38574-2_1.

[10] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description)*. In: *Fourth International Joint Conference on Automated Reasoning, LNCS* 5195, Springer, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[11] Andrew Gacek, Dale Miller & Gopalan Nadathur (2012): *A Two-Level Logic Approach to Reasoning About Computations*. *Journal of Automated Reasoning* 49(2), pp. 241–273, doi:10.1007/s10817-011-9218-1.

[12] J.-Y. Girard, Y. Lafont & P. Tayor (1990): *Proofs and Types*. Cambridge University Press.

[13] Nada Habli & Amy P. Felty (2013): *Translating Higher-Order Specifications to Coq Libraries Supporting Hybrid Proofs*. In: *Third International Workshop on Proof Exchange for Theorem Proving, EasyChair Proceedings in Computing* 14, pp. 67–76.

[14] Holger H. Hoos & Thomas Stützle (2000): *SATLIB: An Online Resource for Research on SAT*. In: *SAT 2000: Highlights of Satisfiability Research in the Year 2000, Frontiers in Artificial Intelligence and Applications* 63, IOS Press, pp. 283–292.

[15] Alberto Momigliano, Alan J. Martin & Amy P. Felty (2008): *Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax*. In: *Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2007, ENTCS* 196, Elsevier, pp. 85–93, doi:10.1016/j.entcs.2007.09.019.

[16] Brigitte Pientka (2007): *Proof Pearl: The Power of Higher-Order Encodings in the Logical Framework LF*. In: *Twentieth International Conference on Theorem Proving in Higher-Order Logics*, LNCS, Springer, pp. 246–261, doi:10.1007/978-3-540-74591-4_19.

[17] Brigitte Pientka & Andrew Cave (2015): *Inductive Beluga:Programming Proofs (System Description)*. In: *Twenty-Fifth International Conference on Automated Deduction*, Springer.

[18] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In: *Fifth International Joint Conference on Automated Reasoning, LNCS* 6173, Springer, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[19] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[20] Adam Poswolsky & Carsten Schürmann (2009): *System Description: Delphin—A Functional Programming Language for Deductive Systems*. In: *Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008), ENTCS* 228, Elsevier, pp. 113–120, doi:10.1016/j.entcs.2008.12.120.

[21] Florian Rabe & Carsten Schürmann (2009): *A Practical Module System for LF*. In: *Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, ACM Press, pp. 40–48, doi:10.1145/1577824.1577831.

[22] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[23] Carsten Schürmann (2009): *The Twelf Proof Assistant*. In: *Twenty-Second International Conference on Theorem Proving in Higher Order Logics, LNCS* 5674, Springer, pp. 79–83, doi:10.1007/978-3-642-03359-9_7.

[24] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2010): *Ott: Effective Tool Support for the Working Semanticist*. *Journal of Functional Programming* 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[25] Geoff Sutcliffe (2009): *The TPTP Problem Library and Associated Infrastructure*. *Journal of Automated Reasoning* 43(4), pp. 337–362, doi:10.1007/s10817-009-9143-8.

[26] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek & Gopalan Nadathur (2013): *Reasoning About Higher-Order Relational Specifications*. In: *Fifteenth International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ACM Press, pp. 157–168, doi:10.1145/2505879.2505889.

A Case Study on Logical Relations using Contextual Types

Andrew Cave
McGill University
Montreal QC, Canada
acave1@cs.mcgill.ca

Brigitte Pientka
McGill University
Montreal QC, Canada
bpientka@cs.mcgill.ca

Proofs by logical relations play a key role to establish rich properties such as normalization or contextual equivalence. They are also challenging to mechanize. In this paper, we describe the completeness proof of algorithmic equality for simply typed lambda-terms by Crary where we reason about logically equivalent terms in the proof environment Beluga. There are three key aspects we rely upon: 1) we encode lambda-terms together with their operational semantics and algorithmic equality using higher-order abstract syntax 2) we directly encode the corresponding logical equivalence of well-typed lambda-terms using recursive types and higher-order functions 3) we exploit Beluga's support for contexts and the equational theory of simultaneous substitutions. This leads to a direct and compact mechanization, demonstrating Beluga's strength at formalizing logical relations proofs.

1 Introduction

Proofs by logical relations play a fundamental role to establish rich properties such as contextual equivalence or normalization. This proof technique goes back to Tait (26) and was later refined by Girard (12). The central idea of logical relations is to specify relations on well-typed terms via structural induction on the syntax of types instead of directly on the syntax of terms themselves. Thus, for instance, logically related functions take logically related arguments to related results, while logically related pairs consist of components that are related pairwise.

Mechanizing logical relations proofs is challenging: first, specifying logical relations themselves typically requires a logic which allows arbitrary nesting of quantification and implications; second, to establish soundness of a logical relation, one must prove the Fundamental Property which says that any well-typed term under a closing simultaneous substitution is in the relation. This latter part requires some notion of simultaneous substitution together with the appropriate equational theory of composing substitutions. As Altenkirch (1) remarked,

> "I discovered that the core part of the proof (here proving lemmas about CR) is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening."

While logical normalization proofs often are not large, they are conceptually intricate and mechanizing them has become a challenging benchmark for proof environments. There are several key questions, when we attempt to formalize such proofs: How should we represent the abstract syntax tree for lambda-terms and enforce the scope of bound variables? How should we represent well-typed terms or typing derivations? How should we deal with substitution? How can we define the logical relation on closed terms?

Early work (1; 2; 5) represented lambda-terms using (well-scoped) de Bruijn indices which leads to a substantial amount of overhead to prove properties about substitutions such as substitution lemmas

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop
on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 33–45, doi:10.4204/EPTCS.185.3

and composition of substitution. To improve readability and generally better support such meta-theoretic reasoning, nominal approaches support α-renaming but substitution and properties about them are specified separately; the Isabelle Nominal package has been used in a variety of logical relations proofs from proving strong normalization for Moggi's modal lambda-calculus (7) to mechanically verifying the meta-theory of LF itself including the completeness of equivalence checking (16; 27).

Approaches representing lambda-terms using higher-order abstract syntax (HOAS) trees (also called λ-tree syntax) model binders in the object language (i.e. in our case the simply typed lambda-calculus) as binders in the meta language (i.e. in our case the logical framework LF (13)). Such encodings inherit not only α-renaming and substitution from the meta-language, but also weakening and substitution lemmas. However, direct encodings of logical relations proofs is beyond the logical strength supported in systems such as Twelf (17). In this paper, we demonstrate the power and elegance of logical relations proofs within the proof environment Beluga (22) which is built on top of the logical framework LF. Beluga allows programmers to pair LF objects together with their surrounding context and this notion is internalized as a contextual type $[\Psi \vdash A]$ which is inhabited by term M of type A in the context Ψ (15). Proofs about contexts and contextual LF objects are then implemented as dependently-typed recursive functions via pattern matching (18; 21). Beluga's functional language supports higher-order functions and indexed recursive data-types (3) which we use to encode the logical relation. As such it does not impose any restrictions as for example found in Twelf (17) which does not support arbitrary quantifier alternation or Delphin (23) which lacks recursive data-types. Recently, Beluga has been extended to first-class simultaneous substitutions allowing abstraction over substitutions and supporting a rich equational theory about them (4; 20).

In this paper, we describe the completeness proof of algorithmic equality for simply typed lambda-terms by Crary (6) where we reason about logically equivalent terms in the proof environment Beluga. There are three key aspects we rely upon: 1) we encode lambda-terms together with their operational semantics together with algorithmic equality using higher-order abstract syntax 2) we directly encode the corresponding logical equivalence of well-typed lambda-terms using recursive types and higher-order functions 3) we exploit Beluga's support for contexts and the equational theory of simultaneous substitutions. This leads to a direct and compact mechanization and allows us to demonstrate Beluga's strength at formalizing logical relations proofs. Based on this case study we also draw some general lessons.

2 Proof Overview: Completeness of Algorithmic Equality

In this section we give a brief overview of the motivation and high level structure of the completeness proof of algorithmic equality. For more detail, we refer the reader to (6) and (14). Extensions of this proof are important for the metatheory of dependently typed systems such as LF and varieties of Martin-Löf Type Theory, where they are used to establish decidability of typechecking. The proof concerns three judgements, the first of which is declarative equivalence:

$$\Gamma \vdash M \equiv N : A \quad \text{terms M and N are declaratively equivalent at type } A$$

Declarative equivalence includes convenient but non-syntax directed rules such as transitivity and symmetry, among rules for congruence, extensionality and β-contraction. We will see the full definition in Sec. 3. In particular, it may include apparently type-directed rules such as extensionality at unit type:

$$\frac{\Gamma \vdash M : \text{Unit} \quad \Gamma \vdash N : \text{Unit}}{\Gamma \vdash M \equiv N : \text{Unit}}$$

This rule relies crucially on type information, so the common untyped rewriting strategy for deciding equivalence no longer applies. Instead, one can define an algorithmic notion of equivalence which is directed by the syntax of types. This is the path we follow here. We define algorithmic term equivalence mutually with path equivalence, which is the syntactic equivalence of terms headed by variables, i.e. terms of the form $x\,M_1 \ldots M_n$.

$\Gamma \vdash M \Leftrightarrow N : A$ terms M and N are algorithmically equivalent at type A
$\Gamma \vdash M \leftrightarrow N : A$ paths M and N are algorithmically equivalent at type A

In what follows, we sketch the proof of completeness of algorithmic equivalence for declarative equivalence. A direct proof by induction over derivations fails unfortunately in the application case where we need to show that applying equivalent terms to equivalent arguments yields equivalent results, which is not so easy. Instead, one can proceed by proving a more general statement that declaratively equivalent terms are *logically equivalent*, and so in turn algorithmically equivalent. Logical equivalence is a relation defined directly on the structure of the types. We write it as follows:

$\Gamma \vdash M \approx N : A$ Terms M and N are logically equivalent at type A

The key case is at function type, which directly defines logically equivalent terms at function type as taking logically equivalent arguments to logically equivalent results. Crary defines:

$$\begin{array}{lll} \Gamma \vdash M_1 \approx M_2 : A \Rightarrow B & \text{iff} & \text{for all } \Delta \geq \Gamma \text{ and } N_1, N_2, \\ & & \text{if } \Delta \vdash N_1 \approx N_2 : A \\ & & \text{then } \Delta \vdash M_1\ N_1 \approx M_2\ N_2 : B \end{array}$$

A key complication is the quantification over all extensions Δ of the context Γ. This is essential to show completeness of the algorithmic rule for function types, which states that that to compare two terms $\Gamma \vdash M \Leftrightarrow N : A \Rightarrow B$ it suffices to compare their applications to *fresh* variables: $\Gamma, x : A \vdash Mx \Leftrightarrow Nx : B$. The generalization to *all* extensions Δ of Γ then arises naturally. This Kripke-style monotonicity condition is one of the reasons that this proof is more challenging than normalization proofs for simply typed lambda-terms, where this quantification can often be avoided using other technical tricks.

For our formalization, we take a slightly different approach which better exploits the features of Beluga available to us. We instead quantify over an arbitrary context Δ together with a simultaneous substitution π which provides for each $x{:}T$ in Γ, a path M satisfying $\Delta \vdash M \leftrightarrow M : T$. We will call such a substitution a *path substitution* and write this condition as $\Delta \vdash \pi : \Gamma$. In the course of the completeness proof, π will actually only ever be instantiated by substitutions which simply perform weakening. That is, Δ will be of the form Γ, Γ' where $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ and π will be of the form $\Gamma, \Gamma' \vdash x_1/x_1, \ldots, x_n/x_n : \Gamma$. However, the extra generality of path substitutions surprisingly does no harm to the proof, and fits well within Beluga.

$$\begin{array}{lll} \Gamma \vdash M_1 \approx M_2 : A \Rightarrow B & \text{iff} & \text{for all } \Delta, \text{ path substitutions } \Delta \vdash \pi : \Gamma, \text{ and } N_1, N_2 \\ & & \text{if } \Delta \vdash N_1 \approx N_2 : A \\ & & \text{then } \Delta \vdash M_1[\pi]\ N_1 \approx M_2[\pi]\ N_2 : B \end{array}$$

The high level goal is to establish that declaratively equivalent terms are logically equivalent, and that logically equivalent terms are algorithmically equivalent. The proof requires establishing a few key properties of logical equivalence. The first is monotonicity, which is crucially used for weakening logical equivalence. This is used when applying terms to fresh variables.

Lemma 2.1 (Monotonicity)
If $\Gamma \vdash M \approx N : A$ *and* $\Delta \vdash \pi : \Gamma$ *is a path substitution, then* $\Delta \vdash M[\pi] \approx N[\pi] : A$

The second key property is (backward) closure of logical equivalence under weak head reduction. This is proved by induction on the type A.

Lemma 2.2 (Logical weak head closure)
If $\Gamma \vdash N_1 \approx N_2 : A$ *and* $M_1 \longrightarrow^*_{wh} N_1$ *and* $M_2 \longrightarrow^*_{wh} N_2$ *then* $\Gamma \vdash M_1 \approx M_2 : A$

In order to escape logical equivalence to obtain algorithmic equivalence in the end, we need the main lemma, which is a mutually inductive proof showing that path equivalence is included in logical equivalence, and logical equivalence is included in algorithmic equivalence:

Lemma 2.3 (Main lemma)

1. *If* $\Gamma \vdash M \leftrightarrow N : A$ *then* $\Gamma \vdash M \approx N : A$
2. *If* $\Gamma \vdash M \approx N : A$ *then* $\Gamma \vdash M \Leftrightarrow N : A$

Also required are symmetry and transitivity of logical equivalence, which in turn require symmetry and transitivity of algorithmic equivalence, determinacy of weak head reduction, and uniqueness of types for path equivalence. We will not go into detail about these lemmas, as they are relatively mundane, but refer the reader to the discussion in (6).

What remains is to show that declarative equivalence implies logical equivalence. This requires a standard technique to generalize the statement to all instantiations of open terms by related substitutions. If σ_1 is of the form $M_1/x_1,...,M_n/x_n$ and σ_2 is of the form $N_1/x_1,...,N_n/x_n$ and Γ is of the form $x_1{:}A_1,...,x_n{:}A_n$, we write $\Delta \vdash \sigma_1 \approx \sigma_2 : \Gamma$ to mean that $\Delta \vdash M_i \approx N_i : A_i$ for all i.

Theorem 2.4 (Fundamental theorem)
If $\Gamma \vdash M \equiv N : A$ *and* $\Delta \vdash \sigma_1 \approx \sigma_2 : \Gamma$ *then* $\Delta \vdash M[\sigma_1] \approx N[\sigma_2] : A$

The proof goes by induction on the derivation of $\Gamma \vdash M \equiv N : A$. We show one interesting case in order to demonstrate some sources of complexity.

Proof Case: $\dfrac{\Gamma, x : A \vdash M_1 \equiv M_2 : B}{\Gamma \vdash \lambda x.M_1 \equiv \lambda x.M_2 : A \Rightarrow B}$

1. Suppose we are given Δ', a path substitution $\Delta' \vdash \pi : \Delta$ and N_1, N_2 with $\Delta' \vdash N_1 \approx N_2 : A$.
2. We have $\Delta' \vdash \sigma_1[\pi] \approx \sigma_2[\pi] : \Gamma$ (by monotonicity)
3. Hence $\Delta' \vdash (\sigma_1[\pi], N_1/x) \approx (\sigma_2[\pi], N_2/x) : \Gamma, x : A$ (by definition)
4. Hence $\Delta' \vdash M_1[\sigma_1[\pi], N_1/x] \approx M_2[\sigma_2[\pi], N_2/x] : B$ (by induction hypothesis)
5. Hence $\Delta' \vdash M_1[\sigma_1[\pi], x/x][N_1/x] \approx M_2[\sigma_2[\pi], x/x][N_2/x] : B$ (by substitution properties)
6. Hence $\Delta' \vdash (\lambda x.M_1[\sigma_1[\pi], x/x])\ N_1 \approx (\lambda x.M_2[\sigma_2[\pi], x/x])\ N_2 : B$ (by weak head closure)
7. Hence $\Delta' \vdash ((\lambda x.M_1)[\sigma_1])[\pi]\ N_1 \approx ((\lambda x.M_2)[\sigma_2])[\pi]\ N_2 : B$ (by substitution properties)
8. Hence $\Delta \vdash (\lambda x.M_1)[\sigma_1] \approx (\lambda x.M_2)[\sigma_2] : A \Rightarrow B$ (by definition of logical equivalence)

We observe that this proof relies heavily on equational properties of substitutions. Some of this complexity appears to be due to our choice of quantifying over substitutions $\Delta \vdash \pi : \Gamma$ instead of extensions $\Delta \geq \Gamma$. However, we would argue that reasoning instead about extensions $\Delta \geq \Gamma$ does not remove this complexity, but only rephrases it.

Finally, by establishing the relatedness of the identity substitution to itself, i.e. $\Gamma \vdash \mathrm{id} \approx \mathrm{id} : \Gamma$ we can combine the fundamental theorem with the main lemma to obtain completeness.

Corollary 2.5 (Completeness) *If* $\Gamma \vdash M \equiv N : A$ *then* $\Gamma \vdash M \Leftrightarrow N : A$

3 Mechanization

We mechanize the development of the declarative and algorithmic equivalence together with its completeness proof in Beluga, a dependently typed proof language built on top of the logical framework LF. The central idea is to specify lambda-terms, small-step semantics, and type-directed algorithmic equivalence in the logical framework LF. This allows us to model bindings uniformly using the LF function space and obviates the need to model and manage names explicitly. Beluga's proof language allows programmers to encapsulate LF objects together with their surrounding context as contextual objects and provides support for higher-order functions, indexed recursive types, and pattern matching on contexts and contextual objects. We define logical equivalence and (for technical reasons) declarative equivalence using indexed recursive types. All our proofs will then be implemented as recursive functions using pattern matching and pass the totality checker. The complete source code for our development can be found in the directory `examples/logrel` of the Beluga distribution which is available at `https://github.com/Beluga-lang/Beluga`.

3.1 Encoding lambda-terms, typing and reduction in the logical framework LF

Our proof is about a simply-typed lambda calculus with one base type `i`. Extending the proof to support a unit type and products is straightforward. We describe the types and terms in LF as follows, employing HOAS for the representation of lambda abstraction. That is, we express the body of the lambda expression as an LF function `tm → tm`. There is no explicit case for variables; they are implicitly handled by LF. We show side by side the corresponding grammar.

```
LF tp : type =
| i :  tp
| ⇒ : tp → tp → tp % infix
;
LF tm : type =
| app : tm → tm → tm
| lam : (tm → tm) → tm;
```

$$\text{Types } T,S ::= i \mid T \Rightarrow S$$

$$\text{Terms } M,N ::= x \mid \text{lam}\, x.M \mid \text{app}\, M\, N$$

Finally, we describe also weak head reduction for our terms. Notice here that the substitution of `N` into `M` in the β-reduction case is accomplished using LF application. We then describe multi-step reductions as a sequence of single step reductions. All free variables occurring in the LF signature are reconstructed and bound implicitly at the outside.

```
LF step : tm → tm → type =
| beta    : step (app (lam M) N) (M N)
| stepapp : step M M' → step (app M N) (app M' N);

LF mstep : tm → tm → type =
| refl   : mstep M M
| trans1 : step M M' → mstep M' M'' → mstep M M'';
```

3.2 Encoding algorithmic equivalence

We now describe the algorithmic equality of terms. This is defined as two mutually recursive LF specifications. We write `algeqTm M N T` for algorithmic equivalence of terms `M` and `N` at type `T` and `algeqP P Q T` for algorithmic path equivalence at type `T` – these are terms whose head is a variable, not a lambda abstraction. Term equality is directed by the type, while path equality is directed by the syntax. Two terms `M` and `N` at base type `i` are equivalent if they weak head reduce to weak head normal terms `P` and `Q` which are path equivalent. Two terms `M` and `N` are equivalent at type `T ⇒ S` if applying them to a fresh variable `x`

of type `T` yields equivalent terms. Variables are only path equivalent to themselves, and applications are path equivalent if the terms at function position are path equivalent, and the terms at argument positions are term equivalent.

```
LF algeqTm: tm → tm → tp → type =
| algbase: mstep M P → mstep N Q → algeqP P Q i → algeqTm M N i.
| algarr : ({x:tm} algeqP x x T → algeqTm (app M x) (app N x) S) → algeqTm M N (arr T S)
and algeqP : tm → tm → tp → type
| algapp : algeqP M1 M2 (arr T S) → algeqTm N1 N2 T → algeqP (app M1 N1) (app M2 N2) S;
```

By describing algorithmic equality in LF, we gain structural properties and substitution for free. For this particular proof, only weakening is important.

A handful of different forms of contexts are relevant for this proof. We describe these with **schema** definitions in Beluga. Schemas classify contexts in a similar way as LF types classify LF objects. Although schemas are similar to Twelf's world declarations, schema checking does not involve verifying that a given LF type family only introduces the assumptions specified in the schema; instead schemas will be used by the computation language to guarantee that we are manipulating contexts of a certain shape. Below, we define the schema `actx`, which enforces that term variables come paired with an algorithmic equality assumption `algeqP x x t` for some type `t`.

```
schema actx = some [t:tp] block x:tm, ax:algeqP x x t;
```

3.3 Encoding logical equivalence

To define logical equivalence, we need the notion of path substitution mentioned in Sec. 2. For this purpose, we use Beluga's built-in notion of simultaneous substitutions. We write `[`$\delta \vdash \gamma$`]` for the built-in type of simultaneous substitutions which provide for each variable in the context γ a corresponding term in the context δ. When γ is of schema `actx`, such a substitution consists of blocks of the form `M/x,P/ax` where `M` is a term and `P` is a derivation of `algeqP M M T`, just as we need.

To achieve nice notation, we define an LF type of pairs of terms, where the infix operator $\approx$ simply constructs a pair of terms:

```
LF tmpair : type =
| ≈ : tm → tm → tmpair % infix;
```

Logical equivalence, written `Log [`$\gamma \vdash$ `M ≈ N] [A]`, expresses that `M` and `N` are logically related in context γ at type A. We embed contextual objects into computations and recursive types wrapping them inside `[ ]`. Since `M` and `N` are used in the context γ, by default, they can depend on γ. Formally, each of these meta-variables is associated with an identity substitution which can be omitted.

We define `Log [`$\gamma \vdash$ `M ≈ N] [A]` in Beluga as a *stratified* type, which is a form of recursive type which is defined by structural recursion on one of its indices, as an alternative to an inductive (strictly positive) definition. Beluga verifies that this stratification condition is satisfied. In this case, the definition is structurally recursive on the type `A`.

```
stratified Log : (γ:actx) [γ ⊢ tmpair] → [tp] → ctype =
| LogBase :  [γ ⊢ algeqTm M N i] → Log [γ ⊢ M ≈ N] [i]
| LogArr  : ({δ:actx}{π:[δ ⊢ γ]}{N1:[δ ⊢ tm]}{N2:[δ ⊢ tm]}
               Log [δ ⊢ N1 ≈ N2] [T] → Log [δ ⊢ app M1[π] N1 ≈ app M2[π] N2] [S])
             → Log [γ ⊢ M1 ≈ M2] [T ⇒ S];
```

At base type, two terms are logically equivalent if they are algorithmically equivalent. At arrow type we employ the monotonicity condition mentioned in Sec. 2: `M1` is related to `M2` in Γ if, for any context Δ, path substitution $\Delta \vdash \pi : \Gamma$, and `N1` , `N2` related in Δ, we have that `app M1[`π`] N1` is related to `app M2[`π`] N2`

in Δ. We quantify over `(γ:actx)` in round parentheses, which indicates that it is implicit and recovered during reconstruction. Variables quantified in curly braces such as `{δ:actx}` are passed explicitly. As in LF specifications, all free variables occurring in constructor definitions are reconstructed and bound implicitly at the outside. They are passed implicitly and recovered during reconstruction.

Crucially, logical equality is monotonic under path substitutions.

```
rec log_monotone : {δ:actx}{π:[δ ⊢ γ]} Log [γ ⊢ M1 ≈ M2] [A] → Log [δ ⊢ M1[π] ≈ M2[π]] [A]
```

We show below the mechanized proof of this lemma only to illustrate the general structure of Beluga proofs. The proof is simply by case analysis on the logical equivalence. In the base case, we obtain a proof P of $\gamma \vdash$ `algeqTm M N i`, which we can weaken for free by simply applying π to P. Here we benefit significantly from Beluga's built-in support for simultaneous substitutions; we gain not just weakening by a single variable for free as we would in Twelf, but arbitrary simultaneous weakening. The proof proceeds in the arrow case by simply composing the two substitutions. We use λ as the introduction form for universal quantifications over metavariables (contextual objects), for which we use uppercase and Greek letters, and **fn** with lowercase letters for computation-level function types (implications).

```
rec log_monotone : {δ:actx}{π:[δ ⊢ γ]} Log [γ ⊢ M1 ≈ M2] [A] → Log [δ ⊢ M1[π] ≈ M2[π]] [A] =
λ δ,π ↦ fn e ↦ case e of
| LogBase [γ ⊢ P] ↦ LogBase [δ ⊢ P[π]]
| LogArr f ↦ LogArr (λ δ',π' ↦ f [δ'] [δ' ⊢ π[π']])
```

The main lemma is mutually recursive, expressing that path equivalence is included in logical equivalence, and logical equivalence is included in algorithmic term equivalence. This enables "escaping" from the logical relation to obtain an algorithmic equality in the end. They are structurally recursive on the type. Crucially, in the arrow case, `reify` instantiates the path substitution π with a weakening substitution in order to create a fresh variable.

```
rec reflect : {A:[tp]} [γ ⊢ algeqP M1 M2 A]  → Log [γ ⊢ M1 ≈ M2] [A]
and reify   : {A:[tp]} Log [γ ⊢ M1 ≈ M2] [A] → [γ ⊢ algeqTm M1 M2 A]
```

We can state weak head closure directly as follows. The proof is structurally recursive on the type, which is implicit.

```
rec closed : [γ ⊢ mstep N1 M1] → [γ ⊢ mstep N2 M2] → Log [γ ⊢ M1 ≈ M2] [T]
           → Log [γ ⊢ N1 ≈ N2] [T]
```

3.4 Encoding declarative equivalence

We now define declarative equality of terms, which includes non-algorithmic rules such as transitivity and symmetry. Declarative equality makes use of a schema which lists only term variables, which we write `ctx`.

```
schema ctx = tm;
```

For technical reasons which we will go into more detail on later, we resort to a different treatment of typing contexts. We explicitly represent typing contexts `dctx` as a list of types, and declarative equality as a computation-level inductive datatype, instead of an LF specification.

```
LF dctx : type =
| nil : dctx
| &   : dctx → tp → dctx % infix ;
```

We describe next the result of looking up the type of a variable `x` in γ in typing context Γ by its position. If `x` is the top variable of γ, then its type in Γ is the type of the top variable of Γ. Otherwise, if looking up the type of `x` in γ yields `T`, then looking it up in an extended context also yields `T`. Here we

write [γ ⊢ tm] for the contextual type of terms of type tm in context γ, and [tp] for (closed) types. We use #p for a meta-variable standing for an object-level variable from γ (as opposed to a general term).

```
inductive  Lookup : {Γ:[dctx]}(γ:ctx)[γ ⊢ tm] → [tp] → ctype =
| Top : Lookup [Γ & T] [γ,x:tm ⊢ x] [T]
| Pop : Lookup [Γ] [γ ⊢ #p] [T] → Lookup [Γ & S] [γ,x:tm ⊢ #p] [T];
```

We write Decl [Γ] [γ ⊢ M ≈ N] [T] for declarative equivalence of M and N at type T. We employ the convention that Γ and Δ stand for typing contexts (of type [dctx]), while γ and δ stand for corresponding term contexts.

```
inductive  Decl : {Γ:[dctx]}(γ:ctx) [γ ⊢ tmpair] → [tp] → ctype =

| DecBeta : Decl [Γ & T] [γ,x:tm ⊢ M2 ≈ N2] [S] → Decl [Γ] [γ ⊢ M1 ≈ N1] [T]
       → Decl [Γ] [γ ⊢ app (lam (\x. M2)) M1  ≈  N2[.., N1]] [S]

| DecLam : Decl [Γ & T] [γ,x:tm ⊢ M ≈ N] [S]
       → Decl [Γ] [γ ⊢ lam (\x. M)  ≈  lam (\x. N)] [T ⇒ S]

| DecExt : Decl [Γ & T] [γ,x:tm ⊢ app M x  ≈  app N x] [S]
       → Decl [Γ] [γ ⊢ M ≈ N] [T ⇒ S]

| DecVar : Lookup [Γ] [γ ⊢ #p] [T] → Decl [Γ] [γ ⊢ #p ≈ #p] [T]

| DecApp : Decl [Γ] [γ ⊢ M1 ≈ M2] [T ⇒ S] → Decl [Γ] [γ ⊢ N1 ≈ N2] [T]
       → Decl [Γ] [γ ⊢ app M1 N1  ≈  app M2 N2] [S]

| DecSym : Decl [Γ] [γ ⊢ M ≈ N] [T] → Decl [Γ] [γ ⊢ N ≈ M] [T]

| DecTrans : Decl [Γ] [γ ⊢ M ≈ N] [T] → Decl [Γ] [γ ⊢ N ≈ O] [T]
         → Decl [Γ] [γ ⊢ M ≈ O] [T];
```

Declarative equality includes a β rule, as well as an extensionality rule, which states that for two terms M and N to be equal at type T ⇒ S, it suffices for them to be equal when applied to a fresh variable of type T. We again remind the reader that all meta-variables are silently associated with the identity substitution; in particular in [γ ⊢ lam (\x.M) ≈ lam (\x.N)], the meta-variables M and N are associated with the identity substitution on the context γ, x:tm. Note that every meta-variable is associated with a simultaneous substitutions in Beluga. If this substitution is the identity, then it can be omitted. Hence, [γ ⊢ lam (\x.M) ≈ lam (\x.N)] is equivalent to writing [γ ⊢ lam (\x.M[.., x]) ≈ lam (\x.N[..,x])]. Written in η-contracted form this is equivalent to: [γ ⊢ lam M ≈ lam N] or making the identity substitution explicit [γ ⊢ lam M[..] ≈ lam N[..]].

Note that meta-variables associated with simultaneous substitutions do not exist other systems. For example in LF and its implementation in Twelf (17) the context of assumptions is ambient and we cannot express dependencies of LF-variables on them. In Twelf, writing lam M is equivalent to its η-expanded form lam \x. M x.

3.5 Fundamental theorem

The fundamental theorem requires us to speak of all instantiations of open terms by related substitutions. We express here the notion of related substitutions using inductive types. Trivially, empty substitutions, written as ·, are related at empty domain. If σ_1 and σ_2 are related at Γ and M1 and M2 are related at T, then σ_1 ,M1 and σ_2 ,M2 are related at Γ & T. The technical reason we use the schema ctx of term assumptions only is that we would like the substitutions σ_1 and σ_2 to carry only terms M, but *not* derivations algeqP M M T (or declarative equality assumptions). If we had used the schema actx or a schema with declarative equality assumptions, the proof of the fundamental theorem would be obligated to construct these derivations, which would be more cumbersome.

```
inductive  LogSub : (γ:ctx)(δ:actx){σ1:[δ ⊢ γ]}{σ2:[δ ⊢ γ]}{Γ:[dctx]} ctype =
| Nil : LogSub [δ ⊢ ·] [δ ⊢ ·] [nil]
```

```
| Dot : LogSub [h ⊢ σ1] [h ⊢ σ2] [Γ] → Log [δ ⊢ M1 ≈ M2] [T]
      → LogSub [δ ⊢ σ1, M1] [δ ⊢ σ2, M2] [Γ & T]
```

We have a monotonicity lemma for logically equivalent substitutions which is similar to the monotonicity lemma for logically equivalent terms:

```
rec wknLogSub : {π:[δ' ⊢ δ]} LogSub [δ ⊢ σ1] [δ ⊢ σ2] [Γ]
                           → LogSub [δ' ⊢ σ1[π]] [δ' ⊢ σ2[π]] [Γ]
```

The fundamental theorem requires a proof that `M1` and `M2` are declaratively equal, together with logically related substitutions σ_1 and σ_2, and produces a proof that `M1[`σ_1`]` and `M2[`σ_2`]` are logically related. In the transitivity and symmetry cases, we appeal to transitivity and symmetry of logical equivalence, the proofs of which can be found in the accompanying Beluga code.

```
rec thm : Decl [Γ] [γ ⊢ M1 ≈ M2] [T]
        → LogSub [δ ⊢ σ1] [δ ⊢ σ2] [Γ]
        → Log [δ ⊢ M1[σ1] ≈ M2[σ2]] [T] =
```

We show the `lam` case of the proof term only to make a high-level comparison to the hand-written proof in Sec. 2. Below, one can see that we appeal to monotonicity (`wknLogSub`), weak head closure (`closed`), and the induction hypothesis on the subderivation `d1`. However, remarkably, there is no explicit equational reasoning about substitutions, since applications of substitutions are automatically simplified. We refer the reader to (4) for the technical details of this simplification.

```
fn d, s ↦ case d of
| DecLam d1 ↦
   LogArr (λ δ', π, N1, N2 ↦ fn rn ↦
     let ih = thm d1 (Dot (wknLogSub [δ'] [δ] [δ' ⊢ π] s) rn) in
     closed [δ' ⊢ trans1 beta refl] [δ' ⊢ trans1 beta refl] ih
   )
...
```

Completeness is a corollary of the fundamental theorem. Our statement of the completeness theorem is slightly complicated by the fact that declarative equality and algorithmic equality live in different context schemas. To overcome this, we describe a predicate `EmbedSub [Γ] [γ] [γ' ⊢ ι]` which states that ι is a simple weakening substitution which performs the work of moving from term context γ`:ctx` to the corresponding (larger) algorithmic equality context γ`':actx` with added algorithmic equality assumptions at the types listed in Γ`:[dctx]`. Morally, this ι substitution plays the role of the identity substitution mentioned in Sec. 2.

```
inductive   EmbedSub : {Γ:[dctx]}{γ:ctx}(γ':actx){ι:[γ' ⊢ γ]} ctype =
| INil : EmbedSub [nil] [] [·]
| ISnoc : EmbedSub [Γ] [γ] [γ' ⊢ ι]
       → EmbedSub [Γ & T] [γ,x:tm] [γ',b:block x:tm,ax:algeqP x x T ⊢ ι, b.1]
```

It is then straightforward to show that embedding substitutions ι are logically related to themselves using the main lemma.

```
rec embed_log : EmbedSub [Γ] [γ] [γ' ⊢ ι] → LogSub [Γ] [γ] [γ' ⊢ ι] [γ' ⊢ ι]
```

The completeness theorem is stated below, and follows trivially by composing the fundamental theorem with `embed_log` and the main lemma to escape the logical relation.

```
rec completeness : EmbedSub [Γ] [γ] [γ' ⊢ ι] → Decl [Γ] [γ ⊢ M1 ≈ M2] [T]
                → [γ' ⊢ algeqTm M1[ι] M2[ι] T]
```

It is unfortunate that this transportation from γ to γ' is required by the current framework of contextual types, since intuitively the algorithmic equality assumptions in γ' are completely irrelevant for the terms `M1` and `M2`. It's an open problem how to improve on this.

3.6 Remarks

The proof passes Beluga's typechecking and totality checking. As part of the totality checker, Beluga performs a strict positivity check for inductive types (19; 20), and a stratification check for logical relation-style definitions.

Beluga's built-in support for simultaneous substitutions is a big win for this proof. The proof of the monotonicity lemma is very simple, since the (simultaneous) weakening of algorithmic equality comes for free, and there is no need for explicit reasoning about substitution equations in the fundamental theorem or elsewhere. We also found that the technique of quantifying over path substitutions as opposed to quantifying over all extensions of a context to work surprisingly well. However, it seems to be non-obvious when this technique will work. In an earlier version of this proof, we had resorted to explicitly enforcing that the substitution π contained only *variables*, limiting its capabilities to weakening, exchange, and contraction. This was done with an inductive datatype like the following, where the contextual type `#[δ ⊢ tm]` contains only *variables* of type `tm`:

```
datatype  IsRenaming : {γ:ctx}(δ:ctx) {π:[δ ⊢ γ]} ctype =
| Nil : IsRenaming [] [δ ⊢ ·]
| Cons :  {#p:#[δ ⊢ tm]} IsRenaming [γ] [δ ⊢ π] → IsRenaming [γ,x:tm] [δ ⊢ π, #p]
```

We were surprised to learn that in fact this restriction was unnecessary, and we could instead simply directly quantify over path substitutions, as the schema `actx` we rely on in our proof already effectively restricts the substitutions we can build. However, we suspect that the technique of explicitly restricting to renaming substitutions may still be necessary in some cases, and that it might be convenient to have a built-in type of these renaming-only substitutions.

We remark that the completeness theorem can in fact be executed, viewing it as an algorithm for normalizing derivations in the declarative system to derivations in the algorithmic system. The extension to a proof of decidability would be a correct-by-construction functional algorithm for the decision problem. This is a unique feature of carrying out the proof in a type-theoretic setting like Beluga, where the proof language also serves as a computation language.

Some aspects of this proof could still be improved. In particular, our treatment of the different context schemas and the relationship between them seems unsatisfactory. We had to do a bit of work in order to move terms from `γ:ctx` to `γ':actx`, and this polluted the final statement of the completeness theorem. It can also be difficult to know when to resort to using an explicit context and a computation-level datatype, like we did for declarative equality. This suggests there is room for improvement in Beluga's treatment of contexts, and we are exploring possible approaches.

Furthermore, one might argue that having to explicitly apply the path substitutions π to terms like $M[\pi]$ is somewhat unsatisfactory, so one might wish for the ability to directly perform the bounded quantification $\forall \Delta \geq \Gamma$ and a notion of subtyping which permits for example `[Γ ⊢ tm]` $\leq$ `[Δ ⊢ tm]`. This is also a possibility we are exploring.

Overall, we found that that the tools provided by Beluga, especially its support for simultaneous substitutions, worked remarkably well to express this proof and to obviate the need for bureaucratic lemmas about substitutions and contexts, and we are optimistic that these techniques can scale to many other varieties of logical relations proofs.

4 Related Work

Mechanizing proofs by logical relations is an excellent benchmark to evaluate the power and elegance of a given proof development. Because it requires nested quantification and recursive definitions, encoding

logical relations has been particularly challening for systems supporting HOAS encodings.

There are two main approaches to support reasoning about HOAS encodings: 1) In the proof-theoretic approaches, we adopt a two-level system where we implement a specification logic (similar to LF) inside a higher-order logic supporting (co)inductive definitions, the approach taken in Abella (9), or type theory, the aproach taken in Hybrid (8). To distinguish in the proof theory between quantification over variables and quantification over terms, (10) introduce a new quantifier, ∇, to describe nominal abstraction logically. To encode logical relations one uses recursive definitions which are part of the reasoning logic (11). Induction in these systems is typically supported by reasoning about the height of a proof tree; this reduces reasoning to induction over natural numbers, although much of this complexity can be hidden in Abella. Compared to our development in Beluga, Abella lacks support for modelling a context of assumptions and simultanous substitutions. As a consequence, some of the tedious basic infrastructure to reason about open and closed terms and substitutions still needs to be built and maintained. Moreover, Abella's inductive proofs cannot be executed and do not yield a program for normalizing derivations. It is also not clear what is the most effective way to perform the quantification over all *extensions* of a context in Abella.

2) The type-theoretic approaches fall into two categories: we either remain within the logical framework and encode proofs as relations as advocated in Twelf (17) or we build a dependently typed functional language on top of LF to support reasoning about LF specifications as done in Beluga. The former approach lacks logical strength; the function space in LF is "weak" and only represents binding structures instead of computations. To circumvent these limitations, (25) proposes to implement a reasoning logic within LF and then use it to encode logical relation arguments. This approach scales to richer calculi (24) and avoids reasoning about contexts, open terms and simultanous substitutions explicitly. However, one might argue that it not only requires additional work to build up a reasoning logic within LF and prove its consistency, but is also conceptually different from what one is used to from on-paper proofs. It is also less clear whether the approach scales easily to proving completeness of algorithmic equality due to the need to talk about context extensions in the definition of logical equivalence of terms of function type.

Outside the world of HOAS, (16) have carried out essentially the same proof in Nominal Isabelle, and later (27) tackle the extension from the simply-typed lambda calculus to LF. Relative to their approach, Beluga gains substitution for free, but more importantly, equations on substitutions are silently discharged by Beluga's built-in support for their equational theory, so they do not even appear in proofs. In contrast, proving these equations manually requires roughly a dozen intricate lemmas.

5 Conclusion

Our implementation of completeness of algorithmic equality takes advantage of key infrastructure provided by Beluga: it utilizes first-class simultaneous substitutions, contexts, contextual objects and the power of recursive types. This yields a direct and compact implementation of all the necessary proofs which directly correspond to their on-paper developments. Moreover, our proof yields an executable program. While more work on Beluga's frontend will improve and make simpler such developments, we have demonstrated that the core language is not only suitable for standard structural induction proofs such as type safety, but also proofs by logical relations.

References

[1] Thorsten Altenkirch (1993): *A Formalization of the Strong Normalization Proof for System F in LEGO*. In Marc Bezem & Jan Friso Groote, editors: *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, *Lecture Notes in Computer Science* 664, Springer, pp. 13–28, doi:10.1007/BFb0037095.

[2] Stefano Berardi (1990): *Girard Normalization Proof in LEGO*. In: *Proceedings of the First Workshop on Logical Frameworks*, pp. 67–78.

[3] Andrew Cave & Brigitte Pientka (2012): *Programming with binders and indexed data-types*. In: *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, ACM Press, pp. 413–424, doi:10.1145/2103656.2103705.

[4] Andrew Cave & Brigitte Pientka (2013): *First-class substitutions in contextual type theory*. In: *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, ACM Press, pp. 15–24, doi:10.1145/2503887.2503889.

[5] Catarina Coquand (1992): *A proof of normalization for simply typed lambda calculus writting in ALF*. In: *Informal Proceedings of Workshop on Types for Proofs and Programs*, Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., pp. 80–87.

[6] Karl Crary (2005): *Logical Relations and a Case Study in Equivalence Checking*. In Bejamin C. Pierce, editor: *Advanced Topics in Types and Programming Languages*, The MIT Press.

[7] Christian Doczkal & Jan Schwinghammer (2009): *Formalizing a Strong Normalization Proof for Moggi's Computational Metalanguage: A Case Study in Isabelle/HOL-nominal*. In: *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09)*, ACM, pp. 57–63, doi:10.1145/1577824.1577834.

[8] Amy Felty & Alberto Momigliano (2012): *Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. *J. Autom. Reasoning* 48(1), pp. 43–105, doi:10.1007/s10817-010-9194-x.

[9] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description)*. In: *4th International Joint Conference on Automated Reasoning, Lecture Notes in Artificial Intelligence* 5195, Springer, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[10] Andrew Gacek, Dale Miller & Gopalan Nadathur (2008): *Combining generic judgments with recursive definitions*. In F. Pfenning, editor: *23rd Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 33–44, doi:10.1109/LICS.2008.33.

[11] Andrew Gacek, Dale Miller & Gopalan Nadathur (2009): *Reasoning in Abella about Structural Operational Semantics Specifications*. In: *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, *Electronic Notes in Theoretical Computer Science (ENTCS)* 228, Elsevier, pp. 85 – 100, doi:10.1016/j.entcs.2008.12.118.

[12] J.-Y. Girard, Y. Lafont & P. Tayor (1990): *Proofs and types*. Cambridge University Press.

[13] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[14] Robert Harper & Frank Pfenning (2005): *On Equivalence and Canonical Forms in the LF Type Theory*. *ACM Transactions on Computational Logic* 6(1), pp. 61–101, doi:10.1145/1042038.1042041.

[15] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory*. *ACM Transactions on Computational Logic* 9(3), pp. 1–49, doi:10.1145/1352582.1352591.

[16] Julien Narboux & Christian Urban (2008): *Formalising in Nominal Isabelle Crary's Completeness Proof for Equivalence Checking*. *Electr. Notes Theor. Comput. Sci.* 196, pp. 3–18, doi:10.1016/j.entcs.2007.09.014.

[17] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In H. Ganzinger, editor: *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), Springer, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[18] Brigitte Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In: *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, ACM Press, pp. 371–382, doi:10.1145/1328438.1328483.

[19] Brigitte Pientka & Andreas Abel (2015): *Structural Recursion over Contextual Objects*. In Thorsten Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, pp. 273–287, doi:10.4230/LIPIcs.TLCA.2015.273.

[20] Brigitte Pientka & Andrew Cave (2015): *Inductive Beluga:Programming Proofs (System description)*. In: *25th International Conference on Automated Deduction (CADE-25)*, Springer.

[21] Brigitte Pientka & Joshua Dunfield (2008): *Programming with proofs and explicit contexts*. In: *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, ACM Press, pp. 163–173, doi:10.1145/1389449.1389469.

[22] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: a Framework for Programming and Reasoning with Deductive Systems (System Description)*. In Jürgen Giesl & Reiner Haehnle, editors: *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), Springer-Verlag, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[23] Adam B. Poswolsky & Carsten Schürmann (2008): *Practical programming with higher-order encodings and dependent types*. In: *17th European Symposium on Programming (ESOP '08)*, 4960, Springer, pp. 93–107, doi:10.1007/978-3-540-78739-6_7.

[24] Ulrik Rasmussen & Andrzej Filinski (2013): *Structural Logical Relations with Case Analysis and Equality Reasoning*. In: *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'13)*, ACM Press, pp. 43–54, doi:10.1145/2503887.2503891.

[25] Carsten Schürmann & Jeffrey Sarnat (2008): *Structural Logical Relations*. In: *23rd Annual Symposium on Logic in Computer Science (LICS), Pittsburgh, PA, USA*, IEEE Computer Society, pp. 69–80, doi:10.1109/LICS.2008.44.

[26] William Tait (1967): *Intensional Interpretations of Functionals of Finite Type I*. *J. Symb. Log.* 32(2), pp. 198–212, doi:10.2307/2271658.

[27] Christian Urban, James Cheney & Stefan Berghofer (2011): *Mechanizing the metatheory of LF*. *ACM Trans. Comput. Log.* 12(2), p. 15, doi:10.1145/1877714.1877721.

Proof-relevant π-calculus

Roly Perera
University of Glasgow
Glasgow, UK
roly.perera@glasgow.ac.uk

James Cheney
University of Edinburgh
Edinburgh, UK
jcheney@inf.ed.ac.uk

Formalising the π-calculus is an illuminating test of the expressiveness of logical frameworks and mechanised metatheory systems, because of the presence of name binding, labelled transitions with name extrusion, bisimulation, and structural congruence. Formalisations have been undertaken in a variety of systems, primarily focusing on well-studied (and challenging) properties such as the theory of process bisimulation. We present a formalisation in Agda that instead explores the theory of concurrent transitions, residuation, and causal equivalence of traces, which has not previously been formalised for the π-calculus. Our formalisation employs de Bruijn indices and dependently-typed syntax, and aligns the "proved transitions" proposed by Boudol and Castellani in the context of CCS with the proof terms naturally present in Agda's representation of the labelled transition relation. Our main contributions are proofs of the "diamond lemma" for residuation of concurrent transitions and a formal definition of equivalence of traces up to permutation of transitions.

1 Introduction

The π-calculus [18, 19] is an expressive model of concurrent and mobile processes. It has been investigated extensively and many variations, extensions and refinements have been proposed, including the asynchronous, polyadic, and applied π-calculus (among many others). The π-calculus has also attracted considerable attention from the logical frameworks and meta-languages community, and formalisations of its syntax and semantics have been performed using most of the extant mechanised metatheory techniques, including (among others) Coq [13, 12, 15], Nominal Isabelle [2], Abella [1] (building on Miller and Tiu [26]), CLF [6], and Agda [21]. These formalisations have overcome challenges that tested the limits of these systems (at least at the time), particularly relating to the encoding of name binding, scope extrusion and structural congruence. Indeed, some early formalisations motivated or led to important contributions to the understanding of these issues in different systems, such as the Theory of Contexts, or CLF's support for monadic encapsulation of concurrent executions.

Prior formalisations have typically considered the syntax, semantics (usually via a variation on labelled transitions), and bisimulation theory of the π-calculus. However, as indicated above, while these aspects of the π-calculus are essential, they only scratch the surface of the properties that could be investigated. Most of these developments have been carried out using informal paper proofs, and formalising them may reveal challenges or motivate further research on logical frameworks.

One interesting aspect of the π-calculus that has not been formally investigated, and remains to some extent ill-understood informally, is its theory of *causal equivalence*. Two transitions t_1, t_2 that can be taken from a process term p are said to be *concurrent* ($t_1 \frown t_2$) if they could be performed "in either order" — that is, if after performing t_1, there is a natural way to transform the other transition t_2 so that its effect is performed on the result of t_1, and vice versa. The translation of the second transition is said to be the *residual* of t_2 after t_1, written t_2/t_1. The key property of this operation, called the "diamond lemma", is that the two residuals t_1/t_2 and t_2/t_1 of transitions $t_1 \frown t_2$ result in the same process. Finally,

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS ??, 2015, pp. 46–70, doi:10.4204/EPTCS.??.4

permutation of concurrent transitions induces a *causal equivalence* relation on pairs of traces. This is the standard notion of permutation-equivalence from the theory of traces over concurrent alphabets [17].

Our interest in this area stems from previous work on provenance, slicing and explanation (e.g. [22]), which we wish to adapt to concurrent settings. Ultimately, we would like to formalise the relationship between informal "provenance graphs" often used informally to represent causal relationships [7] and the semantics of concurrent languages and traces. The π-calculus is a natural starting point for this study. We wish to understand how to represent, manipulate, and reason about π-calculus execution traces safely: that is, respecting well-formedness and causality.

In classical treatments, starting with Lévy [16], a transition is usually considered to be a triple (e, t, e') where e and e' are the terms and t is some information about the step performed. Boudol and Castellani [4] introduced the *proved transitions* approach for CCS in which the labels of transitions are enriched with more information about the transition performed. Boreale and Sangiorgi [3] and Degano and Priami [11] developed theories of causal equivalence for the π-calculus, building indirectly on the proved transition approach; Danos and Krivine [10] and Cristescu, Krivine and Varacca [8] developed notions of causality in the context of reversible CCS and π-calculus respectively. However, there does not appear to be a consensus about the correct definition of causal equivalence for the π-calculus. For example, Cristescu et al. [8] write "[in] the absence of an indisputable definition of permutation equivalence for [labelled transition system] semantics of the π-calculus it is hard to assert the correctness of one definition over another." In their work on reversible π-calculus, they noted that some previous treatments of causality in the π-calculus did not allow permuting transitions within the scope of a ν-binder, and showed how their approach would allow this. Moreover, none of the above approaches has been formalised.

In this paper, we report on a new formalisation of the π-calculus carried out in the dependently-typed programming language Agda [20]. Our main contributions include formalisations of concurrency, residuation, the diamond lemma, and causal equivalence. We do not attempt to formalise the above approaches directly, any one of which seems to be a formidable challenge. Instead, we have chosen to adapt the ideas of Boudol and Castellani to the π-calculus as directly as we can, guided by the hypothesis that their notion of *proved transitions* can be aligned with the *proof terms* for transition steps that arise naturally in a constructive setting. For example, we define the concurrency relation on (compatibly-typed) transition proof terms, and we define residuation as a total function taking two transitions along with a proof that the transitions are concurrent, rather than having to deal with a partial operation.

Our formalisation employs de Bruijn indices [5], an approach with well-known strengths and weaknesses compared, for example, to higher-order or nominal abstract syntax techniques employed in existing formalisations. For convenience, we employ a restricted form of structural congruence called *braiding congruence*, and we have not formalised as many of the classical results on the π-calculus as others have, but we do not believe there are major obstacles to filling these gaps. To the best of our knowledge, ours is the first mechanised proof of the diamond lemma for any process calculus.

The rest of the paper is organised as follows. §2 presents our variant of the (synchronous) π-calculus, including syntax, renamings, transitions and braiding congruence. §3 presents our definitions of concurrency and residuation for transitions, and discusses the diamond lemma. §4 presents our definition of causal equivalence. §5 discusses related work in greater detail and §6 concludes and discusses prospects for future work. Appendix A summarises the Agda module structure; the source code can be found at `https://github.com/rolyp/proof-relevant-pi`, release 0.1. Appendix B contains graphical proof-sketches for some lemmas, and Appendix C some further examples of residuation.

2 Synchronous π-calculus

We present our formalisation in the setting of a first-order, synchronous, monadic π-calculus with recursion and internal choice, using a labelled transition semantics. The syntax of the calculus is conventional (using de Bruijn indices) and is given below.

Name	x, y, z	$::=$	$0 \mid 1 \mid \cdots$		Process	P, Q, R, S	$::=$	$\mathbf{0}$	inactive
Action	a	$::=$	$\underline{x}$	input				$\underline{x}.P$	input
			$\overline{x}\langle y\rangle$	output				$\overline{x}\langle y\rangle.P$	output
			$\overline{x}$	bound output				$P + Q$	choice
			τ	silent				$P \mid Q$	parallel
								νP	restriction
								$!P$	replication

Names are ranged over by x, y and z. An input action is written $\underline{x}$. Output actions are written $\overline{x}\langle y\rangle$ if y is in scope and $\overline{x}$ if the action represents the output of a name whose scope is extruding, in which case we say the action is a *bound* output. Bound outputs do not appear in user code but arise during execution.

To illustrate, the conventional π-calculus term $(\nu x)\, x(z).\overline{y}\langle z\rangle.0 \mid \overline{x}\langle c\rangle.0$ would be represented using de Bruijn indices as $\nu(\underline{0}.\overline{n+1}\langle 0\rangle.\mathbf{0} \mid \overline{0}\langle m+1\rangle.\mathbf{0})$, provided that y and c are associated with indices n and m. Here, the first 0 represents the bound variable x, the second 0 the bound variable z, and the third refers to x again. Note that the symbol $\mathbf{0}$ denotes the inactive process term, not a de Bruijn index.

Let Γ and Δ range over *contexts*, which are finite initial segments of the natural numbers. The function which extends a context with a new element is written as a postfix $\cdot + 1$. A context Γ *closes* P if Γ contains the free variables of P. We denote by $\mathsf{Proc}\,\Gamma$ the set of processes closed by Γ, as defined below. We write $\Gamma \vdash P$ to mean $P \in \mathsf{Proc}\,\Gamma$. Similarly, actions are well-formed only in closing contexts; we write $a : \mathsf{Action}\,\Gamma$ to mean that Γ is closing for a, as defined below.

$\Gamma \vdash P$

$$\frac{}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma + 1 \vdash P}{\Gamma \vdash \underline{x}.P}\ x \in \Gamma \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \overline{x}\langle y\rangle.P}\ x, y \in \Gamma \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P + Q} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}$$

$$\frac{\Gamma + 1 \vdash P}{\Gamma \vdash \nu P} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}$$

$a : \mathsf{Action}\,\Gamma$

$$\frac{}{\underline{x} : \mathsf{Action}\,\Gamma}\ x \in \Gamma \qquad \frac{}{\overline{x} : \mathsf{Action}\,\Gamma}\ x \in \Gamma \qquad \frac{}{\overline{x}\langle y\rangle : \mathsf{Action}\,\Gamma}\ x, y \in \Gamma \qquad \frac{}{\tau : \mathsf{Action}\,\Gamma}$$

To specify the labelled transition semantics, it is convenient to distinguish *bound* actions b from non-bound actions c. A bound action $b : \mathsf{Action}\,\Gamma$ is of the form $\underline{x}$ or $\overline{x}$, and shifts a process from Γ to a target context $\Gamma + 1$, freeing the index 0. A non-bound action $c : \mathsf{Action}\,\Gamma$ is of the form $\overline{x}\langle y\rangle$ or τ, and has a target context which is also Γ. Meta-variable a ranges over all actions, bound and non-bound.

2.1 Renamings

A de Bruijn indices formulation of π-calculus makes extensive use of renamings. A *renaming* $\rho : \Gamma \longrightarrow \Delta$ is any function (injective or otherwise) from Γ to Δ. The labelled transition semantics makes use of the lifting of the successor function $\cdot + 1$ on natural numbers to renamings, which we call push to avoid confusion with the $\cdot + 1$ operation on contexts; $\mathsf{pop}\ y$ which undoes the effect of push, replacing 0 by y; and swap, which transposes the roles of 0 and 1. This de Bruijn treatment of π-calculus is similar to that of Hirschkoff's asynchronous μs calculus [14], except that we give a late rather than early semantics; other differences are discussed in §5 below.

$\mathsf{push}_\Gamma : \Gamma \longrightarrow \Gamma + 1$

$$\mathsf{push}\ x = x + 1$$

$\mathsf{pop}_\Gamma : \Gamma \longrightarrow \Gamma + 1 \longrightarrow \Gamma$

$$\mathsf{pop}\ y\ 0 = y$$
$$\mathsf{pop}\ y\ (x+1) = x$$

$\mathsf{swap}_\Gamma : \Gamma + 2 \longrightarrow \Gamma + 2$

$$\mathsf{swap}\ 0 = 1$$
$$\mathsf{swap}\ 1 = 0$$
$$\mathsf{swap}\ (x+2) = x + 2$$

The Γ subscripts that appear on push_Γ, $\mathsf{pop}_\Gamma\ x$ and swap_Γ are shown in grey to indicate that they may be omitted when their value is obvious or irrelevant; this is a convention we use throughout the paper.

2.1.1 Lifting renamings to processes and actions

The functorial extension $\rho^* : \mathsf{Proc}\ \Gamma \longrightarrow \mathsf{Proc}\ \Delta$ of a renaming $\rho : \Gamma \longrightarrow \Delta$ to processes is defined in the usual way. Renaming under a binder utilises the action of $\cdot + 1$ on renamings, which is also functorial. Syntactically, ρ^* binds tighter than any process constructor.

$\cdot^* : (\Gamma \longrightarrow \Delta) \longrightarrow \mathsf{Proc}\ \Gamma \longrightarrow \mathsf{Proc}\ \Delta$

$$\rho^* \mathbf{0} = \mathbf{0}$$
$$\rho^* (\underline{x}.P) = \underline{\rho x}.(\rho+1)^* P$$
$$\rho^* (\overline{x}\langle y \rangle.P) = \overline{\rho x}\langle \rho y \rangle.\rho^* P$$
$$\rho^* (P + Q) = \rho^* P + \rho^* Q$$
$$\rho^* (P \mid Q) = \rho^* P \mid \rho^* Q$$
$$\rho^* (\nu P) = \nu(\rho+1)^* P$$
$$\rho^* (!P) = !\rho^* P$$

$\cdot^* : (\Gamma \longrightarrow \Delta) \longrightarrow \mathsf{Action}\ \Gamma \longrightarrow \mathsf{Action}\ \Delta$

$$\rho^*\ \underline{x} = \underline{\rho x}$$
$$\rho^*\ \overline{x} = \overline{\rho x}$$
$$\rho^*\ \tau = \tau$$
$$\rho^*\ \overline{x}\langle y \rangle = \overline{\rho x}\langle \rho y \rangle$$

$\cdot + 1 : (\Gamma \longrightarrow \Delta) \longrightarrow \Gamma + 1 \longrightarrow \Delta + 1$

$$(\rho + 1)\ 0 = 0$$
$$(\rho + 1)\ (x+1) = \rho x + 1$$

2.1.2 Properties of renamings

Several equational properties of renamings are used throughout the development; here we present the ones mentioned elsewhere in the paper. Diagrammatic versions of the lemmas, along with string diagrams that offer a graphical intuition for why the lemmas hold, are given in Appendix B.

Lemma 1. $pop\ x \circ push = id$

Freeing the index 0 and then immediately substituting x for it is a no-op.

Lemma 2. $pop\ 0 \circ push + 1 = id$

Lemma 3. $swap + 1 \circ swap \circ swap + 1 = swap \circ swap + 1 \circ swap$

The above are two equivalent ways of swapping indices 0 and 2.

Lemma 4. $pop\ 0 \circ swap = pop\ 0$

Lemma 5. $swap \circ push + 1 = push$, $swap \circ push = push + 1$

Lemma 6. $push \circ \rho = \rho + 1 \circ push$

Lemma 7. $\rho \circ pop\ x = pop\ \rho x \circ \rho + 1$

Lemma 8. $swap \circ \rho + 2 = \rho + 2 \circ swap$

These last two lemmas assert various naturality properties of push, pop x and swap.

2.2 Labelled transition semantics

An important feature of our semantics is that each transition rule has an explicit constructor name. This allow derivations to be written in a compact, expression-like form, similar to the *proven transitions* used by Boudol and Castellani to define notions of concurrency and residuation for CCS [4]. However, rather than giving an additional inductive definition describing the structure of a "proof" that $P \xrightarrow{a} R$, we simply treat the inductive definition of $\longrightarrow$ as a data type. This is a natural approach in a dependently-typed setting.

The rule names are summarised below, and have been chosen to reflect, where possible, the structure of the process triggering the rule. The corresponding relation $P \xrightarrow{a} R$ is defined in Figure 1, for any process $\Gamma \vdash P$, any $a : \mathsf{Action}\ \Gamma$ with target $\Delta \in \{\Gamma, \Gamma + 1\}$, and any $\Delta \vdash R$.

Transition	E, F	$::=$	$\underline{x}.P$	input on x
			$\overline{x}\langle y \rangle.P$	output y on x
			$E + Q \quad P + F$	choose left or right branch
			$E \mid^{a} Q \quad P \mid^{a} F$	propagate a through parallel composition on the left or right
			$E \mid^{\tau}_{y} F \quad E \mid^{\tau}_{y} F$	rendezvous (receiving y on the left or right)
			$\overline{\nu} E$	initiate name extrusion
			$E \mid^{\tau}_{\nu} F \quad E \mid^{\tau}_{\nu} F$	extrusion rendezvous (receiving 0 on the left or right)
			$\nu^{a} E$	propagate a through binder
			$!E$	replicate

The constructor name for each rule is shown to the left of the rule. There is an argument position, indicated by $\cdot$, for each premise of the rule. Note that there are two forms of the transition constructors $\cdot \mid^{a} \cdot$ and $\nu^{a} \cdot$ distinguished by whether they are indexed by a bound action b or by a non-bound action c. Moreover there are additional (but symmetric) rules of the form $P + \cdot$, $P \mid^{b} \cdot$ and $P \mid^{b} \cdot$ where the sub-transition occurs on the opposite side of the operator, and similarly $\cdot \mid^{\tau}_{\nu} \cdot$ and $\cdot \mid^{\tau}_{y} \cdot$ rules in which the positions of sender and receiver are transposed. These are all straightforward variants of the rules shown, and are omitted from Figure 1 for brevity. Meta-variables E and F range over transition derivations; if $E : P \xrightarrow{a} R$ then $\mathrm{src}(E)$ denotes P and $\mathrm{tgt}(E)$ denotes R.

Although a de Bruijn formulation of pi calculus requires a certain amount of housekeeping, one pleasing consequence is that the usual side-conditions associated with the π-calculus transition rules are either subsumed by syntactic constraints on actions, or "operationalised" using the renamings above. In particular:

$$P \xrightarrow{a} R$$

$$\underline{x}.P \frac{}{\underline{x}.P \xrightarrow{\underline{x}} P} \qquad \overline{x}\langle y\rangle.P \frac{}{\overline{x}\langle y\rangle.P \xrightarrow{\overline{x}\langle y\rangle} P} \qquad \cdot + Q \frac{P \xrightarrow{a} R}{P+Q \xrightarrow{a} R}$$

$$\cdot^{c}\mid Q \frac{P \xrightarrow{c} R}{P \mid Q \xrightarrow{c} R \mid Q} \qquad \cdot^{b}\mid Q \frac{P \xrightarrow{b} R}{P \mid Q \xrightarrow{b} R \mid \mathsf{push}^{*}Q} \qquad \cdot\mid^{\tau}_{y}\cdot \frac{P \xrightarrow{\underline{x}} R \qquad Q \xrightarrow{\overline{x}\langle y\rangle} S}{P \mid Q \xrightarrow{\tau} (\mathsf{pop}\ y)^{*}R \mid S}$$

$$\overline{\nu}\cdot \frac{P \xrightarrow{\overline{x+1}\langle 0\rangle} R}{\nu P \xrightarrow{\overline{x}} R} \qquad \cdot\mid^{\tau}_{\nu}\cdot \frac{P \xrightarrow{\underline{x}} R \qquad Q \xrightarrow{\overline{x}} S}{P \mid Q \xrightarrow{\tau} \nu(R \mid S)} \qquad \nu^{c}\cdot \frac{P \xrightarrow{\mathsf{push}^{*}c} R}{\nu P \xrightarrow{c} \nu R}$$

$$\nu^{b}\cdot \frac{P \xrightarrow{\mathsf{push}^{*}b} R}{\nu P \xrightarrow{b} \nu(\mathsf{swap}^{*}R)} \qquad !\cdot \frac{P \mid !P \xrightarrow{a} R}{!P \xrightarrow{a} R}$$

Figure 1: Labelled transition rules ($P+\cdot$, $P\mid^{b}\cdot$, $P\mid^{c}\cdot$, $\cdot\mid^{\tau}_{\nu}\cdot$ and $\cdot\mid^{\tau}_{y}\cdot$ variants omitted)

1. The use of push in the $\cdot^{b}\mid Q$ rule corresponds to the usual side-condition asserting that the binder being propagated by P is not free in Q. In the de Bruijn setting every binder "locally" has the name 0, and so this requirement can be operationalised by rewiring Q so that the name 0 is reserved. The push will be matched by a later pop which substitutes for 0, in the event that the action has a successful rendezvous.
2. The $\overline{\nu}\cdot$ rule requires an extrusion to be initiated by an output of the form $\overline{x+1}\langle 0\rangle$, capturing the usual side-condition that the name being extruded *on* is distinct from the name being extruded.
3. The rules of the form ν^{a} require that the action being propagated has the form $\mathsf{push}^{*}a$, ensuring that it contains no uses of index 0. This corresponds to the usual requirement that an action can only propagate through a binder that it does not mention.

The use of swap in the ν^{b} case follows Hirschkoff [14] and has no counterpart outside of the de Bruijn setting. As a propagating binder passes through another binder, their local names are 0 and 1. Propagation transposes the binders, and so to preserve naming we rewire R with a "braid" that swaps 0 and 1. Since binders are also reordered by *permutations* that relate causally equivalent executions, the swap renaming will also play an important role when we consider concurrent transitions (§3).

The following schematic derivation shows how the compact notation works. Suppose $E : P \xrightarrow{\overline{z+2}\langle 0\rangle} R$ takes place immediately under a ν-binder, causing the scope of the binder to be extruded. Then suppose the resulting bound output propagates through another binder, giving the partial derivation on the left:

$$\nu^{\overline{z}}\cdot \frac{\overline{\nu}\cdot \dfrac{E \dfrac{\vdots}{P \xrightarrow{\overline{z+2}\langle 0\rangle} R}}{\nu P \xrightarrow{\overline{z+1}} R}}{\nu\nu P \xrightarrow{\overline{z}} \nu R} \qquad \nu^{\overline{z}}\cdot \frac{\overline{\nu}E \dfrac{\vdots}{\nu P \xrightarrow{\overline{z+1}} R}}{\nu\nu P \xrightarrow{\overline{z}} \nu R} \qquad \nu^{\overline{z}}\overline{\nu}E \frac{\vdots}{\nu\nu P \xrightarrow{\overline{z}} \nu R}$$

with E standing in for the rest of the derivation. The blue constructors annotating the left-hand side of the derivation tree can be thought of as a partially unrolled "transition term" representing the proof.

The · placeholders associated with each constructor are conceptually filled by the transition terms annotating the premises of that step. We can "roll up" the derivation by a single step, by moving the premises into their corresponding placeholders, as shown in the middle figure.

By repeating this process, we can write the whole derivation compactly as $\nu^{\overline{Z}}\overline{\nu}E$, as shown on the right. Thus the compact form is simply a flattened transition derivation: similar to a simply-typed lambda calculus term written as a conventional expression, in a (Church-style) setting where a term is, strictly speaking, a typing derivation.

2.2.1 Residuals of transitions and renamings

A transition survives any suitably-typed renaming. As alluded to already, this will be essential to formalising causal equivalence. First we define the (rather trivial) residual of a renaming $\rho : \Gamma \longrightarrow \Delta$ after an action $a : \mathsf{Action}\,\Gamma$.

Definition 1 (Residual of ρ after a).

$$\rho/b \stackrel{\text{def}}{=} \rho + 1$$
$$\rho/c \stackrel{\text{def}}{=} \rho$$

The complementary residual a/ρ is also defined and is simply the renamed action $\rho^* a$ defined earlier in §2.1.1. We use the latter notation.

Lemma 9. *Suppose $E : P \xrightarrow{a} Q$ and $\rho : \Gamma \longrightarrow \Delta$, where $\Gamma \vdash P$. Then there exists a transition $E/\rho : \rho^* P \xrightarrow{\rho^* a} (\rho/a)^* Q$ such that $tgt(E/\rho) = \rho/a^* Q$.*

$$\begin{array}{ccc} P & \xrightarrow{\quad E \quad} & Q \\ {\scriptstyle \rho^*}\downarrow & & \downarrow{\scriptstyle (\rho/a)^*} \\ \rho^* P & \xrightarrow[E/\rho]{} & (\rho/a)^* Q \end{array}$$

The proof is the obvious lifting of a renaming to a transition, and is given in Appendix C.

We would not expect E/ρ to be derivable for arbitrary ρ in all extensions of the π-calculus. In particular, the mismatch operator $[x \neq y]P$ that steps to P if x and y are distinct names is only stable under injective renamings.

2.2.2 Structural congruences

We believe our semantics to be closed under the usual π-calculus congruences, but have not attempted to formalise this. The "braiding" congruence $\cong$ introduced in §3.2.1 is in fact a standard π-calculus congruence, which we use to track changes in the relative position of binders under permutations of traces. This could be generalised to include more congruences, but at a corresponding cost in formalisation complexity.

3 Concurrency and residuals

We now use the compact notation for derivations to define a notion of *concurrency* for transitions with the same source state, following the work of Boudol and Castellani for CCS [4]. Concurrent transitions are independent, or causally unordered: they can execute in either order without significant interference. Permutation of concurrent transitions induces a congruence on traces, which is the topic of §4.

3.1 Concurrent transitions

Transitions $P \xrightarrow{a} R$ and $Q \xrightarrow{a'} S$ are *coinitial* iff $P = Q$. We now define a symmetric and irreflexive relation $\smile$ over coinitial transitions. If $E \smile E'$ we say E and E' are *concurrent*. The relation is defined as the symmetric closure of the rules given in Figure 2, again with trivial variants of the rules omitted. For the transition constructors of the form $\cdot \,|^a\, Q$ and $\nu^a \cdot$ which come in bound and non-bound variants, we abuse notation a little and write a single $\smile$ rule quantified over a to mean that there are two separate (but otherwise identical) cases.

$E \smile E'$

$$\frac{}{P \,|^a\, F \smile E \,^{a'}|\, Q} \qquad \frac{E \smile E'}{E \,^{a}|\, Q \smile E' \,|^{\tau}_{y}\, F} \qquad \frac{F \smile F'}{P \,|^a\, F \smile E \,|^{\tau}_{y}\, F'} \qquad \frac{E \smile E'}{E \,^{a}|\, Q \smile E' \,|^{\tau}_{\nu}\, F} \qquad \frac{F \smile F'}{P \,|^a\, F \smile E \,|^{\tau}_{\nu}\, F'}$$

$$\frac{E \smile E'}{E + Q \smile E' + Q} \qquad \frac{F \smile F'}{P \,|^a\, E \smile P \,|^{a'}\, E'} \qquad \frac{E \smile E'}{E \,^{a}|\, Q \smile E' \,^{a'}|\, Q} \qquad \frac{E \smile E' \qquad F \smile F'}{E \,|^{\tau}_{y}\, F \smile E' \,|^{\tau}_{z}\, F'}$$

$$\frac{E \smile E' \qquad F \smile F'}{E \,|^{\tau}_{y}\, F \smile E' \,^{\tau}_{z}|\, F'} \qquad \frac{E \smile E' \qquad F \smile F'}{E \,|^{\tau}_{y}\, F \smile E' \,|^{\tau}_{\nu}\, F'} \qquad \frac{E \smile E' \qquad F \smile F'}{E \,|^{\tau}_{y}\, F \smile E' \,^{\tau}_{\nu}|\, F'} \qquad \frac{E \smile E' \qquad F \smile F'}{E \,|^{\tau}_{\nu}\, F \smile E' \,|^{\tau}_{\nu}\, F'}$$

$$\frac{E \smile E' \qquad F \smile F'}{E \,|^{\tau}_{\nu}\, F \smile E' \,^{\tau}_{\nu}|\, F'} \qquad \frac{E \smile E'}{\overline{\nu} E \smile \overline{\nu} E'} \qquad \frac{E \smile E'}{\overline{\nu} E \smile \nu^a E'} \qquad \frac{E \smile E'}{\nu^a E \smile \nu^{a'} E'} \qquad \frac{E \smile E'}{!E \smile !E'}$$

Figure 2: Concurrent coinitial transitions ($P + \cdot$, and some $\cdot \,^{\tau}_{y}|\, \cdot$ and $\cdot \,^{\tau}_{\nu}|\, \cdot$ variants omitted)

The first rule, $P \,|^a\, F \smile E \,^{a'}|\, Q$, says that two transitions E and F are concurrent if they take place on opposite sides of the same parallel composition. The remaining rules propagate concurrent sub-transitions up through ν, choice, parallel composition, and replication. Note that there are no rules allowing us to conclude that a left-choice step is concurrent with a right-choice step: choices are mutually exclusive. Likewise, there are no rules allowing us to conclude that an input or output transition is concurrent with any other transition; since both E and E' are required to be coinitial, if one of them is an input or output step then they are equal and hence not concurrent.

The $E \,|^{\tau}_{y}\, F \smile E' \,|^{\tau}_{z}\, F'$ rule says that a rendezvous is concurrent with another rendezvous under the same parallel composition, as long as the two inputs are concurrent on the left, and the two outputs are concurrent on the right. The $E \,|^{\tau}_{y}\, F \smile E' \,^{\tau}_{z}|\, F'$ variant is similar, but permits concurrent input and output on the left, with their rendezvous partners concurrent on the right. The $E \,|^{\tau}_{y}\, F \smile E' \,^{\tau}_{\nu}|\, F'$ rule and variants permit a regular rendezvous and an extrusion-rendezvous to be concurrent.

3.2 Residuals of concurrent transitions

Intuitively, if $E \smile E'$ then E and E' are "parallel moves" in the sense of Curry and Feys [9]: if either execution step is taken, the other remains valid, and if both are taken, one ends up in (essentially) the same state, regardless of which step is taken first.

However, concurrent transitions are not completely independent: the location and nature of the redex identified by one transition may change as a consequence of the earlier transition. This intuition is captured by the notion of the residual E/E', explored notably by Lévy in the lambda calculus [16],

and later considered by Stark for concurrent transition systems [25] and in the specific setting of CCS by Boudol and Castellani [4]. The residual specifies how E must be adjusted to take into account the fact that E' has taken place.

Definition 2 (Residual). Suppose $E \smile E'$. Then the *residual* of E after E', written E/E', is given by the least function satisfying the equations in Figure 3.

The operator $\cdot/\cdot$ has higher precedence than any transition constructor. The definition makes use of the renaming lemmas in §2.1.2, and is rather tricky; Appendix C.1 gives several examples which illustrate some of the subtleties that arise in the π-calculus setting, in particular relating to name extrusion.

E/E'

$$
\begin{aligned}
(P \mid^a F)/(E\,{}^c\!\mid Q) &= \mathsf{tgt}(E) \mid^a F \\
(P \mid^a F)/(E\,{}^b\!\mid Q) &= \mathsf{tgt}(E) \mid^a \mathsf{push}^* F \\
(E\,{}^a\!\mid Q)/(P \mid^c F) &= E\,{}^a\!\mid \mathsf{tgt}(F) \\
(E\,{}^a\!\mid Q)/(P \mid^b F) &= \mathsf{push}^* E\,{}^a\!\mid \mathsf{tgt}(F) \\
(E\,{}^a\!\mid Q)/(E' \mid^\tau_y F) &= (\mathsf{pop}\ y)^*(E/E')\,{}^a\!\mid \mathsf{tgt}(F) \\
(P \mid^a F)/(E \mid^\tau_y F') &= (\mathsf{pop}\ y)^* \mathsf{tgt}(E) \mid^a F/F' \\
(E \mid^\tau_y F)/(E'\,{}^b\!\mid Q) &= E/E' \mid^\tau_y \mathsf{push}^* F \\
(E \mid^\tau_y F)/(E'\,{}^c\!\mid Q) &= E/E' \mid^\tau_y F \\
(E \mid^\tau_y F)/(P \mid^b F') &= \mathsf{push}^* E \mid^\tau_y F/F' \\
(E \mid^\tau_y F)/(P \mid^c F') &= E \mid^\tau_y F/F' \\
(E \mid^\tau_\nu F)/(E' \mid^\tau_\nu F') &= \nu^x(E/E'\,{}^{\underline{x+1}}\!\mid \mathsf{tgt}(F)) \\
(E \mid^\tau_\nu F)/(E' \mid^\tau_\nu F') &= \overline{\nu}(E/E'\,{}^{\overline{x+1}\langle 0\rangle}\!\mid \mathsf{tgt}(F)) \\
(E \mid^\tau_\nu F)/(E' \mid^\tau_\nu F') &= \nu^c(E/E'\,{}^{\mathsf{push}^* c}\!\mid \mathsf{tgt}(F)) \\
(P \mid^{\underline{x}} F)/(E \mid^\tau_\nu F') &= \nu^x(\mathsf{tgt}(E) \mid^{\underline{x+1}} F/F') \\
(P \mid^{\overline{x}} F)/(E \mid^\tau_\nu F') &= \overline{\nu}(\mathsf{tgt}(E) \mid^{\overline{x+1}\langle 0\rangle} F/F') \\
(P \mid^c F)/(E \mid^\tau_\nu F') &= \nu^c(\mathsf{tgt}(E) \mid^{\mathsf{push}^* c} F/F') \\
(E \mid^\tau_\nu F)/(E'\,{}^b\!\mid Q) &= E/E' \mid^\tau_\nu \mathsf{push}^* F \\
(E \mid^\tau_\nu F)/(E'\,{}^c\!\mid Q) &= E/E' \mid^\tau_\nu F \\
(E \mid^\tau_\nu F)/(P \mid^{\underline{x}} F') &= \mathsf{push}^* E \mid^\tau_\nu F/F' \\
(E \mid^\tau_\nu F)/(P \mid^{\overline{x}} F') &= \mathsf{push}^* E \mid^\tau_0 F/F' \\
(E \mid^\tau_\nu F)/(P \mid^c F') &= E \mid^\tau_\nu F/F' \\
(E + Q)/(E' + Q) &= E/E' \\
(P \mid^{\underline{x}} F)/(P \mid^b F') &= \mathsf{push}^* P \mid^{\underline{x}} F/F' \\
(P \mid^b F)/(P \mid^{\underline{x}} F') &= \mathsf{push}^* P \mid^b F/F' \\
(P \mid^{\overline{x}} F)/(P \mid^{\overline{u}} F') &= \mathsf{push}^* P \mid^{\overline{x+1}\langle 0\rangle} F/F' \\
(P \mid^c F)/(P \mid^b F') &= \mathsf{push}^* P \mid^c F/F' \\
(P \mid^a F)/(P \mid^c F') &= P \mid^a F/F' \\
(E\,{}^{\underline{x}}\!\mid Q)/(E'\,{}^b\!\mid Q) &= E/E'\,{}^{\underline{x}}\!\mid \mathsf{push}^* Q \\
(E\,{}^b\!\mid Q)/(E'\,{}^{\underline{x}}\!\mid Q) &= E/E'\,{}^b\!\mid \mathsf{push}^* Q \\
(E\,{}^{\overline{x}}\!\mid Q)/(E'\,{}^{\overline{u}}\!\mid Q) &= E/E'\,{}^{\overline{x+1}\langle 0\rangle}\!\mid \mathsf{push}^* Q \\
(E\,{}^c\!\mid Q)/(E'\,{}^b\!\mid Q) &= E/E'\,{}^c\!\mid \mathsf{push}^* Q \\
(E\,{}^a\!\mid Q)/(E'\,{}^c\!\mid Q) &= E/E'\,{}^a\!\mid Q \\
(E \mid^\tau_y F)/(E' \mid^\tau_z F') &= (\mathsf{pop}\ z)^*(E/E') \mid^\tau_y F/F' \\
(E \mid^\tau_y F)/(E' \mid^\tau_\nu F') &= \nu^\tau(E/E' \mid^\tau_y F/F') \\
(E \mid^\tau_\nu F)/(E' \mid^\tau_z F') &= (\mathsf{pop}\ z)^*(E/E') \mid^\tau_\nu F/F' \\
(E \mid^\tau_\nu F)/(E' \mid^\tau_\nu F') &= \nu^\tau(E/E' \mid^\tau_\nu F/F') \\
(\overline{\nu}E)/(\overline{\nu}E') &= E/E' \\
(\overline{\nu}E)/(\nu^b E') &= \overline{\nu}\ \mathsf{swap}^*(E/E') \\
(\overline{\nu}E)/(\nu^c E') &= \overline{\nu}\ E/E' \\
(\nu^b E)/(\overline{\nu}E') &= E/E' \\
(\nu^c E)/(\overline{\nu}E') &= E/E' \\
(\nu^b E)/(\nu^b E') &= \nu E/E' \\
(\nu^c E)/(\nu^b E') &= \nu^c\ \mathsf{swap}^*(E/E') \\
(\nu^b E)/(\nu^c E') &= \nu^b\ E/E' \\
(\nu^c E)/(\nu^c E') &= \nu^c\ E/E' \\
(!E)/(!E') &= E/E'
\end{aligned}
$$

Figure 3: Residual of E after E', omitting $\cdot \mid^\tau_y \cdot$ and $\cdot \mid^\tau_\nu \cdot$ cases

3.2.1 Cofinality of residuals

The idea that one ends up in the same state regardless of whether E or E' is taken first is called *cofinality*. In CCS, where actions never involve binders, and in the lambda calculus, where binders do not move around, cofinality simply means the target states are equivalent. Things are not quite so simple in late-style π-calculus, because binders propagate during execution, as bound actions. Consider

the process $\underline{x}.P \mid \underline{z}.Q$ with two concurrent input actions. Initiating one of the inputs (say $\underline{x}$) starts propagating a binder. As this binder passes through the parallel composition, the transition rules use push to "reserve" the free variable 0 in the right half of the process for potential use by a subsequent pop:

$$\cdot \overset{\underline{x}}{\mid} \underline{z}.Q \;\frac{\Gamma \vdash \underline{x}.P \xrightarrow{\underline{x}} \Gamma + 1 \vdash P}{\Gamma \vdash \underline{x}.P \mid \underline{z}.Q \xrightarrow{\underline{x}} \Gamma + 1 \vdash P \mid \underline{z+1}.(\mathsf{push}+1)^* Q}$$

When the action $(\underline{z+1})$ is performed, a push on the left leaves the final state with both 0 and 1 reserved:

$$P \overset{\underline{z+1}}{\mid} \cdot \;\frac{\Gamma + 1 \vdash \underline{z+1}.(\mathsf{push}+1)^* Q \xrightarrow{\underline{z+1}} \Gamma + 2 \vdash (\mathsf{push}+1)^* Q}{\Gamma + 1 \vdash P \mid \underline{z+1}.(\mathsf{push}+1)^* Q \xrightarrow{\underline{z+1}} \Gamma + 2 \vdash \mathsf{push}^* P \mid (\mathsf{push}+1)^* Q}$$

Had these concurrent actions happened in the opposite order, the push on the left would have been applied first. The final state would be $(\mathsf{push}+1)^* P \mid \mathsf{push}^* Q$, which is the image of $\mathsf{push}^* P \mid (\mathsf{push}+1)^* Q$ in the permutation swap which renames 0 to 1 and 1 to 0. Instead of the usual cofinality square, the final states are related by a "braid" (in the form of a swap) which permutes the free names:

$$\begin{array}{ccc}
\Gamma + 1 \vdash P \mid \underline{z+1}.(\mathsf{push}+1)^* Q & \xrightarrow{\underline{z+1}} & \Gamma + 2 \vdash \mathsf{push}^* P \mid (\mathsf{push}+1)^* Q \\
\overset{\underline{x}}{\nearrow} & & \downarrow \mathsf{swap}^* \\
\Gamma \vdash \underline{x}.P \mid \underline{z}.Q & & \Gamma + 2 \vdash \mathsf{swap}^* \mathsf{push}^* P \mid \mathsf{swap}^* (\mathsf{push}+1)^* Q \\
\underset{\underline{z}}{\searrow} & & \Vert\, \alpha \mid \beta \\
\Gamma + 1 \vdash \underline{x+1}.(\mathsf{push}+1)^* P \mid Q & \xrightarrow[\underline{x+1}]{} & \Gamma + 2 \vdash (\mathsf{push}+1)^* P \mid \mathsf{push}^* Q
\end{array}$$

Here α and β are equalities obtained from Lemma 5.

It is not just the reordering of bound actions which nuances π-calculus cofinality. When two τ actions are reordered, which happen to be extrusion rendezvous of distinct binders, the resulting binders exchange positions in the final process. In the standard π-calculus this would be subsumed by the congruence $(\nu x y)\, P \cong (\nu y x)\, P$. In the de Bruijn setting, where adjacent binders cannot be distinguished, the analogous rule is $\nu\nu P \cong \nu\nu(\mathsf{swap}^* P)$, which applies a swap braid under the two binders.

These two possibilities are subsumed by the following generalised notion of cofinality. First we define a braiding congruence $\cong$ just large enough to permit swap under a pair of binders. "Cofinality" is then defined using a more general braiding relation which additionally permits swaps of free variables. Examples showing reordered extrusions are given in Appendix C.1, including concurrent extrusions of the *same* binder, an interesting case identified by Cristescu et al. [8].

Definition 3 (Braiding congruence). Inductively define the binary relation $\cong$ over processes using the rules given in Figure 4.

In Figure 4, rule names are shown to the left in blue, permitting a compact term-like notation for $\cong$ proofs similar to the convention we introduced earlier for transitions. The process constructors are overloaded to witness compatibility; transitivity is denoted by $\circ$. It is easy to see that $\cong$ is also reflexive and symmetric, and therefore a congruence. $P_{\cong}$ denotes the canonical proof that $P \cong P$.

In what follows ϕ and ψ range over braiding congruences; $\mathrm{src}(\phi)$ and $\mathrm{tgt}(\phi)$ denote P and R, for any $\phi : P \cong R$. As with transitions, braiding congruences are stable under renamings, giving rise to the usual notion of residuation; however ρ/ϕ is always ρ. The proof is a straightforward induction.

$P \cong R$

$$\nu\nu\text{-swap}_P \ \frac{}{\nu\nu(\mathrm{swap}^* P) \cong \nu\nu P} \qquad \nu\nu\text{-swap}_P^{-1} \ \frac{}{\nu\nu P \cong \nu\nu(\mathrm{swap}^* P)} \qquad \cdot \circ \cdot \ \frac{R \cong S \qquad P \cong R}{P \cong S} \qquad \mathbf{0} \ \frac{}{\mathbf{0} \cong \mathbf{0}}$$

$$\underline{x}.\cdot \ \frac{P \cong R}{\underline{x}.P \cong \underline{x}.R} \qquad \overline{x}\langle y\rangle.\cdot \ \frac{P \cong R}{\overline{x}\langle y\rangle.P \cong \overline{x}\langle y\rangle.R} \qquad \cdot + \cdot \ \frac{P \cong R \qquad Q \cong S}{P + Q \cong R + S} \qquad \cdot \mid \cdot \ \frac{P \cong R \qquad Q \cong S}{P \mid Q \cong R \mid S}$$

$$\nu\cdot \ \frac{P \cong R}{\nu P \cong \nu R} \qquad !\cdot \ \frac{P \cong R}{!P \cong !R}$$

Figure 4: Braiding congruence $\cong$

Lemma 10. *For any $\Gamma \vdash P$, suppose $\phi : P \longrightarrow Q$ and $\rho : \Gamma \longrightarrow \Delta$. Then there exists a braiding congruence $\phi/\rho : \rho^* P \longrightarrow \rho^* Q$.*

$$\begin{array}{ccc} P & \xrightarrow{\ \phi\ } & Q \\ {\scriptstyle\rho}\downarrow & & \downarrow{\scriptstyle\rho} \\ \rho^* P & \xrightarrow[\ \phi/\rho\ ]{} & \rho^* Q \end{array}$$

Definition 4 (Braiding). For any $\Delta \in \{0,1,2\}$ define the following family of bijective renamings $\mathrm{braid}_{\Gamma,\Delta} : \Gamma + \Delta \longrightarrow \Gamma + \Delta$ and symmetric *braiding* relations $\bowtie_{\Gamma,\Delta}$ over processes in $\Gamma + \Delta$.

$$\begin{array}{l} \mathrm{braid}_{\Gamma,0} = \mathrm{id}_\Gamma : \Gamma \longrightarrow \Gamma \\ \mathrm{braid}_{\Gamma,1} = \mathrm{id}_{\Gamma+1} : \Gamma + 1 \longrightarrow \Gamma + 1 \\ \mathrm{braid}_{\Gamma,2} = \mathrm{swap}_\Gamma : \Gamma + 2 \longrightarrow \Gamma + 2 \end{array} \qquad P \bowtie_{\Gamma,\Delta} P' \iff \mathrm{braid}_{\Gamma,\Delta}{}^* P \cong P'$$

Our key soundness result is that residuals of concurrent transitions E and E' are always cofinal up to a braiding of type $\bowtie_{\Gamma,\Delta}$ where $\Delta \in \{0,1,2\}$ is the number of free variables introduced by E and E'/E. Rather than the usual parallel-moves square on the left, the residuals satisfy pentagons of the form shown in the centre of Figure 5, where $\gamma : Q \bowtie_{\Gamma,\Delta} Q'$ is a braiding.

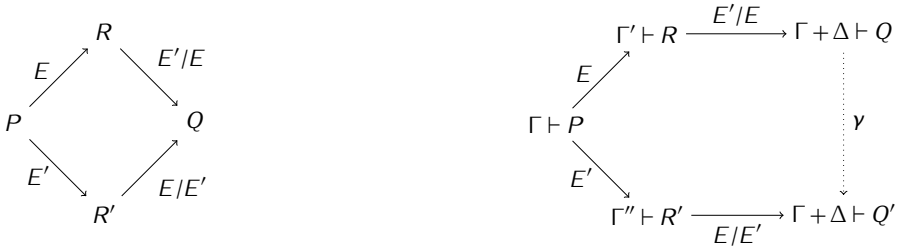


Figure 5: Cofinality in the style of CCS (left); with explicit braiding (right)

Arranging for this to hold by construction introduces a certain amount of complexity, so we prove cofinality as a separate theorem.

Theorem 1 (Cofinality of residuals). *Suppose E and E' are the transitions on the right of Figure 5, with $E \smile E'$. Then there exists $cofin_{E,E'} : Q \bowtie_\Delta Q'$.*

The notion of concurrency extends into dimensions greater than two. Following Pratt's higher-dimensional automata [23], we can consider a proof $\chi : E \smile E'$ as a surface that represents the concurrency of E and E' without committing to an order of occurrence. Every such $\chi : E \smile E'$ has a two-dimensional residual χ/E'' with respect to a third concurrent transition E''. First we note that concurrent transitions are closed under renamings.

Lemma 11. *Suppose $\rho : \Gamma \longrightarrow \Delta$ and E, E' are both transitions from $\Gamma \vdash P$, with $\chi : E \smile E'$. Then there exists $\chi/\rho : E/\rho \smile E'/\rho$.*

Proof. By induction on χ, using Lemma 9. □

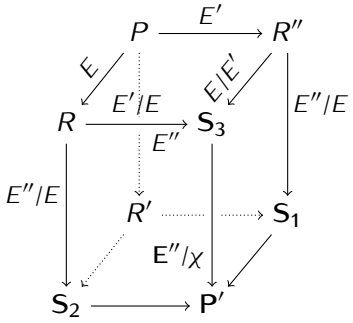
Theorem 2 (Residuation preserves concurrency).
Suppose $\chi : E \smile E'$ with $E \smile E''$ and $E' \smile E''$. Then there exists $\chi/E'' : E/E'' \smile E'/E''$.

Proof. By induction on χ and inversion on the other two derivations, using Lemma 11. □

Theorem 3. *Suppose $\chi : E \smile E'$, with $E' \smile E''$ and $E'' \smile E$. Then:*

$$((E'/E'')/(E/E''))/cofin_{E'',E} = (E'/E)/(E''/E)$$

The diagram below illustrates Theorems 2 and 3 informally. The three faces χ, χ' and χ'' with P as a vertex witness the pairwise concurrency of E, E' and E''. Theorem 2 ensures that these have opposite faces χ/E'', χ'/E and χ''/E'. Theorem 3 states that, up to a suitable braiding, there is a unique residual of a one-dimensional transition after a concurrent two-dimensional one, connecting the faces χ'/E and χ''/E' via the shared edge E''/χ. Analogous reasoning for E/χ' and E'/χ'' yields a cubical transition with target $\mathbf{P'}$.



The bold font for $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_3}$ and $\mathbf{P'}$ indicates that they represent not a unique process but a permutation group of processes related by braidings. At $\mathbf{P'}$ there are potentially $3! = 6$ variants of the target process, one for each possible interleaving of E, E' and E''. The notation $\mathbf{E''}/\chi$ is again informal, referring not to a unique transition but to a permutation group related by braidings.

4 Causal equivalence

4.1 Traces

Define Action* Γ to be the set of finite sequences of composable actions starting at Γ. The empty sequence at Γ is written $[]$; extension to the left is written $a :: \widetilde{a}$. A *trace* $t : P \xrightarrow{\widetilde{a}} R$ is a finite sequence of composable transitions with initial state $\mathrm{src}(t) = P$ and final state $\mathrm{tgt}(t) = R$. The empty trace at P is written $[]_P$; extension to the left of $t : R \xrightarrow{\widetilde{a}} S$ by $E : P \xrightarrow{a} R$ is written $E :: t$.

4.2 Residuals of traces and braidings

To define the residual of a trace t with respect to a braiding γ, we first observe that a braiding congruence $\phi : P \cong P'$ commutes (on the nose) with a transition $E : P \xrightarrow{a} Q$, inducing the corresponding notions of residual ϕ/E (the image of the braiding congruence in the transition) and E/ϕ (the image of the transition in the braiding congruence).

Theorem 4. *Suppose $E : P \xrightarrow{a} R$ and $\phi : P \cong P'$. Then there exists a process R', transition $E/\phi : P' \xrightarrow{a} R'$ and structural congruence $\phi/E : R \cong R'$.*

$$\begin{array}{ccc} P & \xrightarrow{\quad E \quad} & R \\ {\scriptstyle\phi}\downarrow & & \vdots\,{\scriptstyle\phi/E} \\ P' & \xrightarrow[\quad E/\phi \quad]{} & R' \end{array}$$

Proof. By the defining equations in Figure 6.

Unlike residuals of the form E/E', the cofinality of E/ϕ and ϕ/E is by construction. Appendix C.2 illustrates cofinality for the cases where ϕ is of the form $\nu\nu\text{-}\mathsf{swap}_P$.

To extend this notion of residuation from braiding congruences to braidings requires a more general notion of braiding which permits the renaming component of the braiding to be shifted under a binder. First recall (from Definition 4) that any braiding $\gamma : P \bowtie_{\Gamma,\Delta} P'$ is of the form $\phi \circ \mathsf{braid}_{\Gamma,\Delta}$, where $\mathsf{braid}_{\Gamma,\Delta} : \Gamma + \Delta \longrightarrow \Gamma + \Delta$ is the renaming id or swap, as determined by $\Delta \in \{0,1,2\}$, and ϕ is a braiding congruence. We omit the Γ, Δ subscripts whenever possible. The more general form of braiding allows the braid and ϕ components to be translated by an arbitrary context Δ'.

Definition 5 (Δ-shifted braiding). For any context Δ define

$$P \bowtie^{\Delta}_{\Gamma,\Delta'} P' \iff (\mathsf{braid}_{\Gamma,\Delta'} + \Delta)^* P \cong P'$$

Now we define the residual of a transition $E : \Gamma \vdash P \xrightarrow{a} \Gamma + \Delta \vdash R$, where $\Delta \in \{0,1\}$, and coinitial braiding γ and show that the residual γ/E is γ shifted by Δ.

Definition 6 (Residuals of transitions and braidings). For any transition $E : P \xrightarrow{a} R$ and braiding $\gamma : P \bowtie^{\Delta} P'$ with $\gamma = \phi \circ \sigma$, define E/γ and γ/E by the following equations.

$$E/(\phi \circ \sigma) = (E/\sigma)/\phi \qquad\qquad (\phi \circ \sigma)/E = (\phi/(E/\sigma)) \circ \sigma/a$$

Cofinality is immediate by composing the square obtained by applying Lemma 9 to E and σ with the square obtained from Theorem 4 above to ϕ and E/σ. Closure of (Δ-shifted) braidings under residuation follows from the fact that $\sigma/a = \sigma + \Delta'$ for some $\Delta' \in \{0,1\}$.

E/ϕ

$$
\begin{aligned}
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\overline{\nu}\nu^{\overline{x+1}\langle 0\rangle}E) &= \nu^{\overline{x}}\overline{\nu}(\mathsf{swap}^*E)\\
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\nu^{\overline{x}}\overline{\nu}E) &= \overline{\nu}\nu^{\overline{x+1}\langle 0\rangle}(\mathsf{swap}^*E)\\
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\nu^c\nu^{c'}E) &= \nu^c\nu^{c'}(\mathsf{swap}^*E)\\
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\nu^b\nu^{b'}E) &= \nu^b\nu^{b'}(\mathsf{swap}^*E)\\
(\underline{x}.P)/(\underline{x}.\phi) &= \underline{x}.\mathrm{tgt}(\phi)\\
(\overline{x}\langle y\rangle.P)/(\overline{x}\langle y\rangle.\phi) &= \overline{x}\langle y\rangle.\mathrm{tgt}(\phi)\\
(E+Q)/(\phi+\psi) &= E/\phi+\mathrm{tgt}(\psi)\\
(E\,{}^b|\,Q)/(\phi+\psi) &= E/\phi\,{}^b|\,\mathrm{tgt}(\psi)\\
(E\,{}^c|\,Q)/(\phi+\psi) &= E/\phi\,{}^c|\,\mathrm{tgt}(\psi)\\
(P\,|^b\,F)/(\phi+\psi) &= \mathrm{tgt}(\phi)\,|^b\,F/\psi\\
(P\,|^c\,F)/(\phi+\psi) &= \mathrm{tgt}(\phi)\,|^c\,F/\psi\\
(E\,|^\tau_y\,F)/(\phi\mid\psi) &= E/\phi\,|^\tau_y\,F/\psi\\
(E\,|^\tau_\nu\,F)/(\phi\mid\psi) &= E/\phi\,|^\tau_\nu\,F/\psi\\
(\overline{\nu}E)/(\nu\phi) &= \overline{\nu}\;E/\phi\\
(\nu^bE)/(\nu\phi) &= \nu^bE/\phi\\
(\nu^cE)/(\nu\phi) &= \nu^cE/\phi\\
(!E)/(!\phi) &= !E/(\phi\mid !\phi)\\
E/(\phi'\circ\phi) &= (E/\phi)/\phi'
\end{aligned}
$$

ϕ/E

$$
\begin{aligned}
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\overline{\nu}\nu^{\overline{x+1}\langle 0\rangle}E) &= \nu\;\mathrm{tgt}(E)_{\cong}\\
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\nu^{\overline{x}}\overline{\nu}E) &= \nu\;\mathsf{swap}^*\mathrm{tgt}(E)_{\cong}\\
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\nu^c\nu^{c'}E) &= \nu\nu\text{-swap}^{-1}_{\mathrm{tgt}(E)}\\
\nu\nu\text{-swap}_{\mathrm{src}(E)}/(\nu^b\nu^{b'}E) &= \nu\nu\text{-swap}^{-1}_{\mathsf{swap}^*(\mathsf{swap}+1)^*\mathsf{swap}^*\mathrm{tgt}(E)}\\
(\underline{x}.\phi)/(\underline{x}.P) &= \phi\\
(\overline{x}\langle y\rangle.\phi)/(\overline{x}\langle y\rangle.P) &= \phi\\
(\phi+\psi)/(E+Q) &= \phi/E\\
(\phi+\psi)/(E\,{}^b|\,Q) &= \phi/E\mid\mathsf{push}^*\psi\\
(\phi+\psi)/(E\,{}^c|\,Q) &= \phi/E\mid\psi\\
(\phi+\psi)/(P\,|^b\,F) &= \mathsf{push}^*\phi\mid\psi/F\\
(\phi+\psi)/(P\,|^c\,F) &= \phi\mid\psi/F\\
(\phi\mid\psi)/(E\,|^\tau_y\,F) &= (\mathsf{pop}\;y)^*\phi/E\\
(\phi\mid\psi)/(E\,|^\tau_\nu\,F) &= \nu(\phi/E\mid\psi/F)\\
(\nu\phi)/(\overline{\nu}E) &= \phi/E\\
(\nu\phi)/(\nu^bE) &= \nu\;\mathsf{swap}^*\phi/E\\
(\nu\phi)/(\nu^cE) &= \nu\;\phi/E\\
(!\phi)/(!E) &= (\phi\mid !\phi)/E\\
(\phi'\circ\phi)/E &= (\phi'/(E/\phi))\circ\phi/E
\end{aligned}
$$

Figure 6: Residual of transition E and coinitial braiding congruence ϕ

$$
\begin{array}{ccc}
P & \xrightarrow{\quad E\quad} & R\\
{\scriptstyle\sigma+\Delta}\downarrow & & \downarrow{\scriptstyle\sigma+\Delta'}\\
(\sigma+\Delta)^*P & \xrightarrow{E/(\sigma+\Delta)} & (\sigma+\Delta')^*R\\
{\scriptstyle\phi}\downarrow & & \downarrow{\scriptstyle\phi/(E/(\sigma+\Delta))}\\
P' & \xrightarrow[(E/(\sigma+\Delta))/\phi]{} & R'
\end{array}
$$

where both $E/(\sigma+\Delta)$ and $(E/(\sigma+\Delta))/\phi$ have the action $(\sigma+\Delta)^*a$.

Finally, we extend the definition to traces.

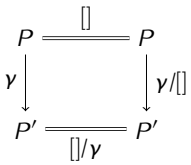
Definition 7 (Residuals of action sequences and renamings).
Suppose $\rho:\Gamma\longrightarrow\Delta$ and $\widetilde{a}:\mathrm{Action}^*\,\Gamma$. Define the residuals $\rho/\widetilde{a}$ and $\widetilde{a}/\rho$, writing the latter as $\rho^*\widetilde{a}$.

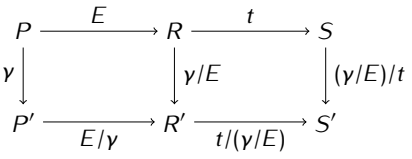
$$
\begin{aligned}
\rho/[]_\Gamma &= \rho & \qquad \rho/(a::\widetilde{a}) &= (\rho/a)/\widetilde{a}\\
\rho^*[]_\Gamma &= []_\Delta & \qquad \rho^*(a::\widetilde{a}) &= (\rho^*a)::(\rho/a)^*\widetilde{a}
\end{aligned}
$$

Lemma 12 (Residuals of traces and braidings).
Suppose $t:P\xrightarrow{\widetilde{a}}R$ and $\gamma=\phi\circ\sigma:P\bowtie^\Delta P'$. Then there exists a process R', trace $P'\xrightarrow{\sigma^*\widetilde{a}}R'$ and braiding $\gamma/t:R\bowtie R'$.

$$
\begin{array}{ccc}
P & \xrightarrow{\quad t\quad} & R\\
{\scriptstyle\gamma}\downarrow & & \vdots{\scriptstyle\gamma/t}\\
P' & \xrightarrow[t/\gamma]{} & R'
\end{array}
$$

Proof. By the following defining equations.





$$[]_P/\gamma = []_{P'} \qquad (E :: t)/\gamma = (E/\gamma) :: t/(\gamma/E)$$
$$\gamma/[]_P = \gamma \qquad \gamma/(E :: t) = (\gamma/E)/t$$

4.3 Causal equivalence

We now define *causal equivalence*, the congruence over traces induced by the notion of transition residual from §3.2. A causal equivalence $\alpha : t \simeq u$ witnesses the reordering of one trace t into a coinitial trace u by the permutation of concurrent transitions. Meta-variables α, β range over causal equivalences.

Definition 8. Inductively define the relation $\simeq$ given by the rules in Figure 7, where syntactically $\simeq$ has lower priority than $\cdot :: \cdot$. If $\alpha : t \simeq u$ then $\mathrm{src}(\alpha)$ and $\mathrm{tgt}(\alpha)$ denote t and u respectively.

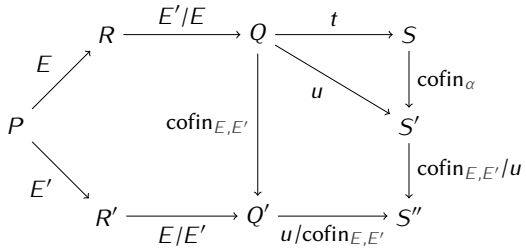
$t \simeq u$

$$[]_P \ \frac{}{[]_P \simeq []_P} \qquad \cdot :: \cdot \ \frac{E : P \xrightarrow{a} R \qquad t \simeq u}{E :: t \simeq E :: u} \ \mathrm{src}(t) = R \qquad \cdot \circ \cdot \ \frac{t' \simeq u \qquad t \simeq t'}{t \simeq u}$$

$$(\cdot :\rightleftharpoons: \cdot) :: \cdot \ \frac{E : P \xrightarrow{a} R \qquad E' : P \xrightarrow{a'} R' \qquad t \simeq u}{E :: E'/E :: t \simeq E' :: E/E' :: u/\mathrm{cofin}_{E,E'}} \ E \smile E'$$

Figure 7: Causal equivalence

The $[]_P$ and $E :: \alpha$ rules are the congruence cases. The $\alpha \circ \beta$ rule closes under transitivity, which is a form of vertical composition. The transposition rule $(E :\rightleftharpoons: E') :: \alpha$ extends an existing causal equivalence $\alpha : t \simeq u$ with the two possible interleavings of concurrent steps $E \smile E'$. What is interesting about this rule is that the trace u must be transported through the braiding $\mathrm{cofin}_{E,E'}$ witnessing the cofinality of E and E', in order to obtain a trace $u/\mathrm{cofin}_{E,E'}$ composable with E'/E. The following diagram illustrates.



As the diagram suggests, the transposition rule causes braidings to compose vertically. Here, cofin_α is a composite braiding relating S to S', which is extended by the braiding $\mathsf{cofin}_{E,E'}/u$ to relate S to S''. We leave formalising this aspect of causal equivalence to future work.

Theorem 5. *$\simeq$ is an equivalence relation.*

Proof. Reflexivity is a trivial induction, using the $[]_P$ and $E :: \alpha$ rules. Transitivity is immediate from the $\alpha \circ \beta$ rule. Symmetry is trivial in the $[]_P$, $E :: \alpha$ and $\alpha \circ \beta$ cases. The $(E :\rightleftharpoons: E') :: \alpha$ case requires the symmetry of $\smile$ and that $(u/\mathsf{cofin}_\alpha)/\mathsf{cofin}_\alpha^{-1} = u$, where $u = \mathsf{tgt}(\alpha)$.

5 Related work

Hirschkoff's μs calculus [14] has a similar treatment of de Bruijn indices. Its renaming operators $\langle x \rangle$, ϕ and ψ are effectively our $\mathsf{pop}\ x$, push and swap renamings, but fused with the $\cdot^*$ operator which applies a renaming to a process. Hirschkoff's operators are also syntactic forms in the μs calculus, rather than meta-operations, and therefore the operational semantics also includes rules for reducing occurrences of the renaming operators that arise during a process reduction step.

Formalisations of the π-calculus have been undertaken in several theorem provers used for mechanised metatheory. Due to space limits, we limit attention to closely-related formalisation techniques based on constructive logics.

Coq. Hirschkoff [13] formalised the π-calculus in Coq using de Bruijn indices, and verified properties such as congruence and structural equivalence laws of bisimulation. Despeyroux [12] formalised the π-calculus in Coq using weak higher-order abstract syntax, assuming a decidable type of names, and using two separate transitions, for ordinary, input and output transitions respectively; for input and output transitions the right-hand side is a function of type $\mathsf{name} \longrightarrow \mathsf{proc}$. This formalisation included a simple type system and proof of type soundness. Honsell, Miculan and Scagnetto [15] formalised the π-calculus in Coq, also using weak higher-order abstract syntax. The type of names name is a type parameter assumed to admit decidable equality and freshness (notin) relations. Transitions are encoded using two inductive definitions, for free and bound actions, which differ in the type of the third argument (proc vs. $\mathsf{name} \longrightarrow \mathsf{proc}$). Numerous results from Milner, Parrow and Walker [19] are verified, using the *theory of contexts* (whose axioms are assumed in their formalisation, but have been validated semantically).

CLF. Cervesato, Pfenning, Walker and Watkins [6] formalise synchronous and asynchronous versions of π-calculus in the Concurrent Logical Framework (CLF). CLF employs higher-order abstract syntax, linearity and a monadic encapsulation of certain linear constructs that can identify objects such as traces up to causal equivalence. Thus, CLF's π-calculus encodings naturally induce equivalences on traces. However, a nontrivial effort appears necessary to compare CLF's notion of trace equivalence with others (including ours) due to the distinctive approach taken in CLF.

Agda. Orchard and Yoshida [21] present a translation from a functional language with effects to a π-calculus with session types and verify some type-preservation properties of the translation in Agda.

6 Conclusions and future work

To the best of our knowledge, we are the first to report on a formalisation of the operational behavior of the π-calculus in Agda. Compared to prior formalisations, ours is distinctive in two ways.

First, our formalisation employs an indexed family of types for process terms and uses the indices instead of binding to deal with scope extrusion. Formalisations of lambda-calculi often employ this

technique, but to our knowledge only Orchard and Yoshida report a similar approach for a π-calculus formalisation. This choice helps tame the complexity of de Bruijn indices, because many invariants are automatically checked by the type system rather than requiring additional explicit reasoning.

Second, our work appears to be the first to align the notion of "proved transitions" from Boudol and Castellani's work on CCS with "transition proofs" in the π-calculus. This hinges on the capability to manipulate and perform induction or recursion over derivations, and means we can leverage dependent typing so that residuation is defined only for concurrent transitions, rather than on all pairs of transitions. It is worth noting that while CLF's approach to encoding π-calculus automatically yields an equivalence on traces, it is unclear (at least to us) whether this equivalence is the same as the one we propose, or whether such traces can be manipulated explicitly as proof objects if desired.

In future work we may explore trace structures explicitly quotiented by causal equivalence, such as dependence graphs [17] or event structures [4]. We are also interested in extending braiding congruence to the full π-calculus structural congruence, and in understanding whether and how ideas from homotopy type theory [24], such as quotients or higher inductive types, could be applied to ease reasoning about or correct programming with π-calculus terms (modulo structural congruence) or traces (modulo causal equivalence).

Acknowledgements The authors were supported by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The first author was also supported by UK EPSRC project *From Data Types to Session Types: A Basis for Concurrency and Distribution* (EP/K034413/1).

References

[1] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications. J. Formalized Reasoning* 7(2), doi:10.6092/issn.1972-5787/4650.

[2] Jesper Bengtson & Joachim Parrow (2009): *Formalising the pi-calculus using nominal logic. Logical Methods in Computer Science* 5(2:16), doi:10.2168/LMCS-5(2:16)2009.

[3] Michele Boreale & Davide Sangiorgi (1998): *A Fully Abstract Semantics for Causality in the π-Calculus. Acta Inf.* 35(5), pp. 353–400, doi:10.1007/s002360050124.

[4] Gérard Boudol & Ilaria Castellani (1989): *Permutation of transitions: An event structure semantics for CCS and SCCS.* In J.W. Bakker, W.-P. Roever & G. Rozenberg, editors: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, *LNCS* 354, Springer, pp. 411–427, doi:10.1007/BFb0013028.

[5] N.G. de Bruijn (1972): *Lambda-Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem. Indagationes Mathematicae* 34(5), pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[6] Iliano Cervesato, Frank Pfenning, David Walker & Kevin Watkins (2002): *A Concurrent Logical Framework II: Examples and Applications.* Technical Report CMU-CS-02-102, Carnegie Mellon University.

[7] James Cheney & Roly Perera (2014): *An Analytical Survey of Provenance Sanitization.* In: *IPAW*, pp. 113–126, doi:10.1007/978-3-319-16462-5_9.

[8] Ioana Cristescu, Jean Krivine & Daniele Varacca (2013): *A compositional semantics for the reversible pi-calculus.* In: *LICS*, pp. 388–397, doi:10.1109/LICS.2013.45.

[9] Haskell B. Curry & Robert Feys (1958): *Combinatory Logic. Studies in Logic and the Foundations of Mathematics* 1, North-Holland, Amsterdam, Holland.

[10] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[11] Pierpaolo Degano & Corrado Priami (1999): *Non-Interleaving Semantics for Mobile Processes*. *Theor. Comput. Sci.* 216(1-2), pp. 237–270, doi:10.1016/S0304-3975(99)80003-6.

[12] Joëlle Despeyroux (2000): *A Higher-Order Specification of the pi-Calculus*. In: *IFIP TCS*, *LNCS* 1872, Springer-Verlag, pp. 425–439, doi:10.1007/3-540-44929-9_30.

[13] Daniel Hirschkoff (1997): *A Full Formalisation of pi-Calculus Theory in the Calculus of Constructions*. In: *TPHOLs*, pp. 153–169, doi:10.1007/BFb0028392.

[14] Daniel Hirschkoff (1999): *Handling Substitutions Explicitly in the pi-Calculus*. In: *Proceedings of the Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*.

[15] Furio Honsell, Marino Miculan & Ivan Scagnetto (2001): *π-calculus in (Co)Inductive-type Theory*. *Theor. Comput. Sci.* 253(2), pp. 239–285, doi:10.1016/S0304-3975(00)00095-5.

[16] Jean-Jacques Lévy (1980): *Optimal reductions in the lambda-calculus*. In J. P. Seldin & J. R. Hindley, editors: *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 159–191.

[17] A. Mazurkiewicz (1987): *Trace Theory*. In: *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, *LNCS* 255, Springer-Verlag, pp. 279–324, doi:10.1007/3-540-17906-2_30.

[18] Robin Milner (1999): *Communicating and mobile systems: the π calculus*. Cambridge University Press.

[19] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I and II*. *Inf. Comput.* 100(1), pp. 1–77, doi:10.1016/0890-5401(92)90009-5.

[20] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: *Advanced Functional Programming*, *LNCS* 5832, Springer, pp. 230–266, doi:10.1007/978-3-642-04652-0_5.

[21] Dominic A. Orchard & Nobuko Yoshida (2015): *Using session types as an effect system*. In: *PLACES*.

[22] Roly Perera, Umut A. Acar, James Cheney & Paul Blain Levy (2012): *Functional Programs That Explain Their Work*. In: *ICFP*, ACM, pp. 365–376, doi:10.1145/2364527.2364579.

[23] Vaughan Pratt (2000): *Higher Dimensional Automata Revisited*. *Mathematical Structures in Computer Science* 10(4), pp. 525–548, doi:10.1017/S0960129500003169.

[24] The Univalent Foundations Program (2013): *Homotopy type theory: Univalent foundations of mathematics*. Technical Report, Institute for Advanced Study.

[25] Eugene W. Stark (1989): *Concurrent Transition Systems*. *Theoretical Computer Science* 64(3), pp. 221–269, doi:10.1016/0304-3975(89)90050-9.

[26] Alwen Tiu & Dale Miller (2010): *Proof Search Specifications of Bisimulation and Modal Logics for the π-calculus*. *ACM Trans. Comput. Logic* 11(2), pp. 13:1–13:35, doi:10.1145/1656242.1656248.

A Agda module structure

Figure 8 summarises the module structure of the Agda formalisation.

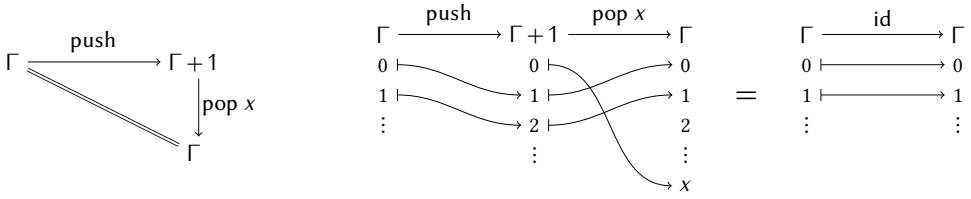
Utilities	
`Common`	Useful definitions not found in the Agda standard library
`SharedModules`	Common imports from standard library
Core modules	
`Action`	Actions a
`Action.Concur`	Concurrent actions $a \smile a'$; residuals a/a'
`Action.Concur.Action`	Residual of $a \smile a'$ after a''
`Action.Seq`	Action sequences $\widetilde{a}$
`Name`	Contexts Γ; names x
`Proc`	Processes P
`Ren`	Renamings $\rho : \Gamma \longrightarrow \Gamma'$
`StructuralCong.Proc`	Braiding congruence relation $\phi : P \cong P'$
`StructuralCong.Transition`	Residuals E/ϕ and ϕ/E
`Transition`	Transitions $E : P \xrightarrow{a} R$
`Transition.Concur`	Concurrent transitions $\chi : E \smile E'$; residuals E/E'
`Transition.Concur.Cofinal`	Cofinality braidings γ
`Transition.Concur.Cofinal.Transition`	Residuals E/γ and γ/E
`Transition.Concur.Transition`	Residual χ/E
`Transition.Seq`	Transition sequences
`Transition.Seq.Cofinal`	Residuals t/γ and γ/t; permutation equivalence $\alpha : t \simeq u$
Typical sub-modules	
`.Properties`	Additional properties relating to X
`.Ren`	Renaming lifted to X

Figure 8: Module overview

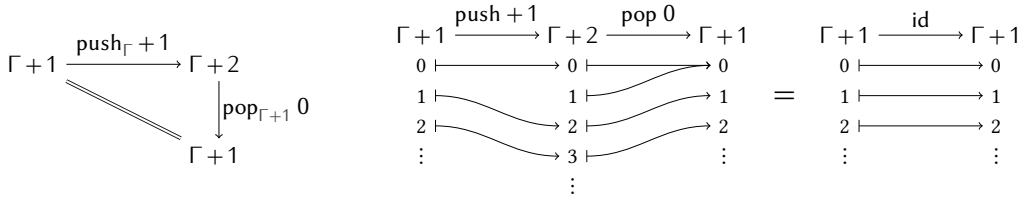
B Renaming lemmas

Each lemma asserts the commutativity of the diagram on the left; when a string diagram is also provided, it should be interpreted as an informal proof sketch.

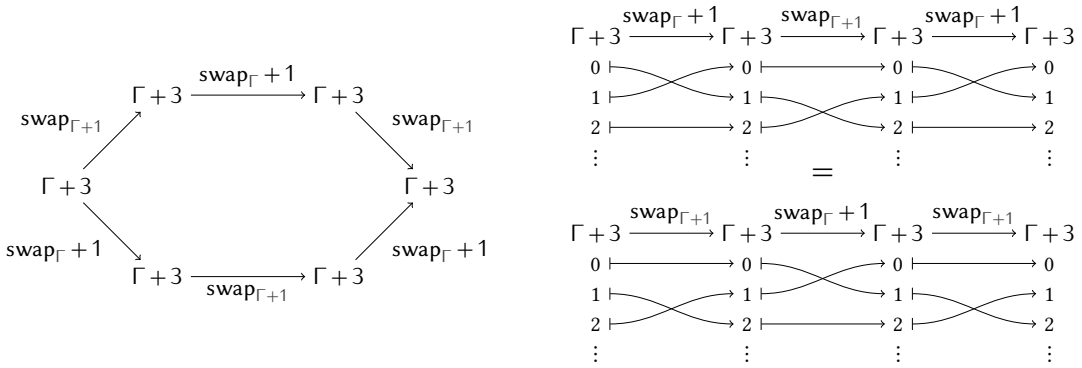
Lemma 1.



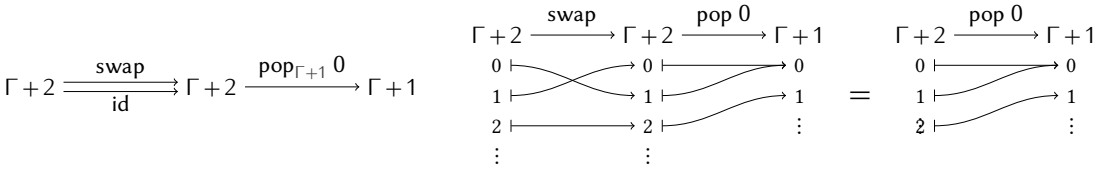
Lemma 2.



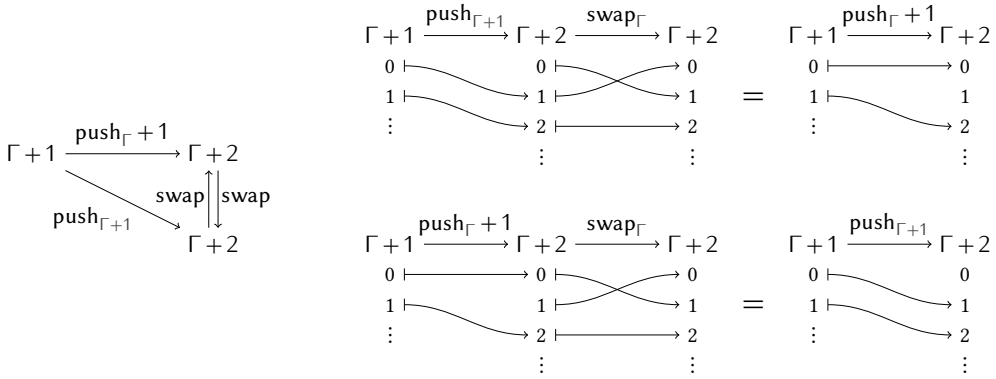
Lemma 3.



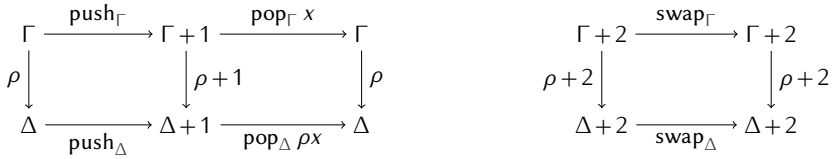
Lemma 4.



Lemma 5.



Lemmas 6, 7 and 8.



C Additional proofs

Proof of Lemma 9. By the following mutually recursive proofs-by-induction on the derivations. The various renaming lemmas needed to enable the induction hypothesis in each case are omitted.

$\rho^* E^c$

$$\begin{aligned}
\rho^*(\overline{x}\langle y\rangle.P) &= \overline{\rho x}\langle \rho y\rangle.\rho^* P \\
\rho^*(E+F) &= \rho^* E + \rho^* F \\
\rho^*(P \mid^c F) &= \rho^* P \mid^{\rho^* c} \rho^* F \\
\rho^*(E \,{}^c\!\mid Q) &= \rho^* E \,{}^{\rho^* c}\!\mid \rho^* Q \\
\rho^*(E \mid^\tau_y F) &= \rho^* E \mid^\tau_{\rho^* y} \rho^* F \\
\rho^*(E \mid^\tau_\nu F) &= \rho^* E \mid^\tau_\nu \rho^* F \\
\rho^*(\nu^c E) &= \nu^{\rho^* c}(\rho+1)^* E \\
\rho^*(!E) &= !\rho^* E
\end{aligned}$$

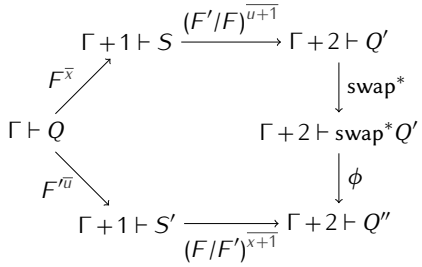
$\rho^* E^b$

$$\begin{aligned}
\rho^*(\underline{x}.P) &= \underline{\rho x}.(\rho+1)^* P \\
\rho^*(E+F) &= \rho^* E + \rho^* F \\
\rho^*(P \mid^b F) &= \rho^* P \mid^{\rho^* b} \rho^* F \\
\rho^*(E \,{}^b\!\mid Q) &= \rho^* E \,{}^{\rho^* b}\!\mid \rho^* Q \\
\rho^*(\overline{\nu} E) &= \overline{\nu}(\rho+1)^* E \\
\rho^*(\nu^b E) &= \nu^{\rho^* b}(\rho+1)^* E \\
\rho^*(!E) &= !\rho^* E
\end{aligned}$$

C.1 Additional illustrative cases of Theorem 1

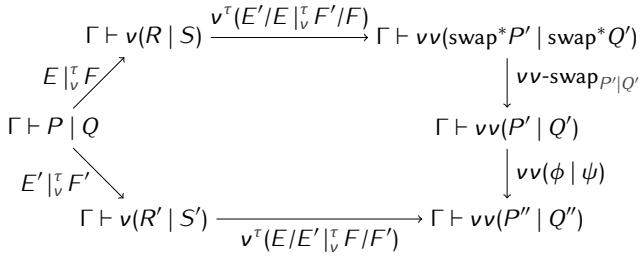
Example: permuting concurrent extrusions (different binders). First, note that the residuals of bound output transitions are not themselves necessarily bound. More specifically, the residuals of the output transition on $\overline{x}$ with the output on $\overline{z}$ is bound only if the outputs represent extrusions of different ν-binders. In this section we consider only the case when the concurrent extrusions are of different ν-binders.

In this case, each binder is unaffected by the extrusion of the other, and the residuals remain bound outputs, shifted into $\Gamma+1$ as usual. The general form of such residuals is:



where ϕ ranges over braiding congruence. Then the residual is able to handle the inner extrusion, with the resulting τ action again propagated through the outer binder:

$$\cdot \mid^\tau_\nu \cdot \quad \frac{E \;\dfrac{\vdots}{\Gamma \vdash P \xrightarrow{\;x\;} R} \qquad F \;\dfrac{\vdots}{\Gamma \vdash Q \xrightarrow{\;\overline{x}\;} S}}{\Gamma \vdash P \mid Q \xrightarrow{\;\tau\;} \nu(R \mid S)}$$



Example: permuting concurrent extrusions (same binder). Consider the process $\nu(\overline{x+1}\langle 0\rangle.P \mid \overline{z+1}\langle 0\rangle.Q)$, as described in Cristescu et al. [8]. There are two concurrent outputs, both of which try to extrude the top-level binder. Suppose we take the $\overline{x+1}\langle 0\rangle$ action first:

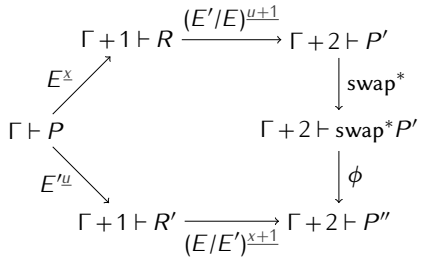
$$\overline{\nu}\cdot \frac{\cdot \mid \cdot \dfrac{\overline{x+1}\langle 0\rangle.P \xrightarrow{\overline{x+1}\langle 0\rangle} P}{\Gamma+1 \vdash \overline{x+1}\langle 0\rangle.P \mid \overline{z+1}\langle 0\rangle.Q \xrightarrow{\overline{x+1}\langle 0\rangle} \Gamma+1 \vdash P \mid \overline{z+1}\langle 0\rangle.Q}}{\Gamma \vdash \nu(\overline{x+1}\langle 0\rangle.P \mid \overline{z+1}\langle 0\rangle.Q) \xrightarrow{\overline{x}} \Gamma+1 \vdash P \mid \overline{z+1}\langle 0\rangle.Q}$$

If we then take the $\overline{z+1}\langle 0\rangle$ action, the enclosing ν-binder no longer exists, and so $\overline{z+1}\langle 0\rangle$ simply propagates as a non-bound action.

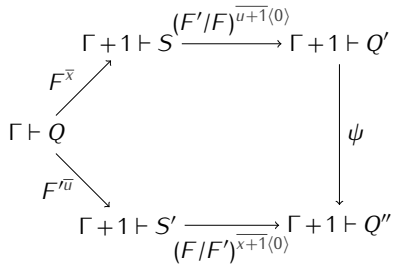
$$P \mid^{\overline{z+1}\langle 0\rangle} \cdot \frac{\Gamma+1 \vdash \overline{z+1}\langle 0\rangle.Q \xrightarrow{\overline{z+1}\langle 0\rangle} \Gamma+1 \vdash Q}{\Gamma+1 \vdash P \mid \overline{z+1}\langle 0\rangle.Q \xrightarrow{\overline{z+1}\langle 0\rangle} \Gamma+1 \vdash P \mid Q}$$

Example: permuting one extrusion-rendezvous with another. Now consider what happens when the extrusions from the previous example eventually rendezvous with a compatible input.

$$E \mid^\tau_\nu F : \Gamma \vdash P \mid Q \xrightarrow{\tau} \Gamma \vdash \nu(R \mid S)$$
$$E' \mid^\tau_\nu F' : \Gamma \vdash P \mid Q \xrightarrow{\tau} \Gamma \vdash \nu(R' \mid S')$$



When the extrusions are of the same ν-binder, and the residual outputs are not bound, then we have:



and the residual of one extrusion-handling after another is a plain communication, with the resulting τ action simply propagated through the second ν binder:

$$\begin{array}{ccc}
\Gamma \vdash \nu(R \mid S) & \xrightarrow{\nu^\tau(E'/E\,|_0^\tau\, F'/F)} & \Gamma \vdash \nu((\mathsf{pop}\ 0)^*\mathsf{swap}^*P' \mid Q') \\
{\scriptstyle E\,|_\nu^\tau\, F}\nearrow & & \| \\
\Gamma \vdash P \mid Q & & \nu(\alpha \mid Q') \\
{\scriptstyle E'\,|_\nu^\tau\, F'}\searrow & & \Gamma \vdash \nu((\mathsf{pop}\ 0)^*P' \mid Q') \\
 & & \downarrow{\scriptstyle \nu((\mathsf{pop}\ 0)^*\phi \mid \psi)} \\
\Gamma \vdash \nu(R' \mid S') & \xrightarrow[\nu^\tau(E/E'\,|_0^\tau\, F/F')]{} & \Gamma \vdash \nu((\mathsf{pop}\ 0)^*P'' \mid Q'')
\end{array}$$

Here α is the equality $(\mathsf{pop}\ 0) \circ \mathsf{swap} = \mathsf{pop}\ 0$ (Lemma 4) applied to P'.

Example: permuting bound actions propagating through a binder. Now suppose we have a process of the form νP which has two concurrent transitions propagating an input action through the ν binder:

$$\nu^{\underline{x}}\cdot\ \dfrac{E\ \dfrac{\vdots}{\Gamma+1 \vdash P \xrightarrow{\underline{x+1}} \Gamma+2 \vdash R}}{\Gamma \vdash \nu P \xrightarrow{\underline{x}} \Gamma+1 \vdash \nu(\mathsf{swap}^*R)} \qquad \nu^{\underline{u}}\cdot\ \dfrac{E'\ \dfrac{\vdots}{\Gamma+1 \vdash P \xrightarrow{\underline{u+1}} \Gamma+2 \vdash R'}}{\Gamma \vdash \nu P \xrightarrow{\underline{u}} \Gamma+1 \vdash \nu(\mathsf{swap}^*R')}$$

(The derivations are valid because both $\underline{x+1}$ and $\underline{z+1}$ are of the form push^*b.) The residuals of E and E' with respect to each other have the form:

$$\begin{array}{ccc}
\Gamma+2 \vdash R & \xrightarrow{(E'/E)^{\underline{u+2}}} & \Gamma+3 \vdash P' \\
{\scriptstyle E^{\underline{x+1}}}\nearrow & & \downarrow{\scriptstyle \mathsf{swap}^*} \\
\Gamma+1 \vdash P & & \Gamma+3 \vdash \mathsf{swap}^*P' \\
{\scriptstyle E'^{\underline{u+1}}}\searrow & & \downarrow{\scriptstyle \phi} \\
\Gamma+2 \vdash R' & \xrightarrow[(E/E')^{\underline{x+2}}]{} & \Gamma+3 \vdash P''
\end{array}$$

We can use these residuals to define the following composite residual $(\nu^{\underline{u}}E')/(\nu^{\underline{x}}E)$:

$$\nu^{\cdot}\cdot\ \dfrac{\mathsf{swap}^*\cdot\ \dfrac{E'/E\ \dfrac{\vdots}{\Gamma+2 \vdash R \xrightarrow{\underline{u+2}} \Gamma+3 \vdash P'}}{\Gamma+2 \vdash \mathsf{swap}^*R \xrightarrow{\underline{u+2}} \Gamma+3 \vdash (\mathsf{swap}+1)^*P'}}{\Gamma+1 \vdash \nu(\mathsf{swap}^*R) \xrightarrow{\underline{u+1}} \Gamma+2 \vdash \nu(\mathsf{swap}^*(\mathsf{swap}+1)^*P')}$$

noting that $\mathsf{swap}^*(\underline{u+2}) = \underline{u+2}$ by Lemma 8. The complementary residual $(\nu^{\underline{x}}E)/(\nu^{\underline{u}}E')$ is similar, with x instead of u and R' instead of R. It remains to show that the terminal states are swap-congruent:

$$\begin{array}{ll}
\quad \mathsf{swap}^*\nu(\mathsf{swap}^*(\mathsf{swap}+1)^*P') & \\
= \nu((\mathsf{swap}+1)^*\mathsf{swap}^*(\mathsf{swap}+1)^*P') & \text{(definition of } \cdot^*) \\
= \nu(\mathsf{swap}^*(\mathsf{swap}+1)^*\mathsf{swap}^*P') & \text{(Lemma 3)} \\
\cong \nu(\mathsf{swap}^*(\mathsf{swap}+1)^*P'') & (\nu(\mathsf{swap}^*(\mathsf{swap}+1)^*\phi))
\end{array}$$

Example: permuting extruding rendezvous and unhandled extrusion. Of course concurrent transitions are not always as symmetric as the ones we have seen. Here a name extrusion which has a successful rendezvous, resulting in a τ action, is concurrent with another which does not and which therefore propagates as a bound output:

$$P \mid^{\overline{u}} F : \Gamma \vdash P \mid Q \xrightarrow{\overline{u}} \Gamma + 1 \vdash \mathsf{push}^* P \mid S$$
$$E \mid^{\tau}_{\nu} F' : \Gamma \vdash P \mid Q \xrightarrow{\tau} \Gamma \vdash \nu(R \mid S')$$

As before, it matters whether the extrusions $F^{\overline{x}} \smile F'^{\overline{u}}$ are of the same or different binders.

Sub-case: extrusions of same binders. In this case, the residuals F'/F and F/F' become sends of index 0, the binder being extruded.

$$\cdot \mid^{\tau}_{0} \cdot \dfrac{\mathsf{push}^* \cdot \dfrac{E \dfrac{\vdots}{\Gamma \vdash P \xrightarrow{\overline{x}} \Gamma + 1 \vdash R}}{\Gamma + 1 \vdash \mathsf{push}^* P \xrightarrow{\overline{x+1}} \Gamma + 2 \vdash (\mathsf{push} + 1)^* R} \qquad F'/F \dfrac{\vdots}{\Gamma + 1 \vdash S \xrightarrow{\overline{x+1}\langle 0 \rangle} \Gamma + 1 \vdash Q'}}{\Gamma + 1 \vdash \mathsf{push}^* P \mid S \xrightarrow{\tau} \Gamma + 1 \vdash (\mathsf{pop}\ 0)^* (\mathsf{push} + 1)^* R \mid Q'}$$

For the other residual, we can derive:

$$\overline{\nu} \cdot \dfrac{R \mid^{\overline{u+1}\langle 0 \rangle} \cdot \dfrac{F/F' \dfrac{\vdots}{\Gamma + 1 \vdash S' \xrightarrow{\overline{u+1}\langle 0 \rangle} \Gamma + 1 \vdash Q''}}{\Gamma + 1 \vdash R \mid S' \xrightarrow{\overline{u+1}\langle 0 \rangle} \Gamma + 1 \vdash R \mid Q''}}{\Gamma \vdash \nu(R \mid S') \xrightarrow{\overline{u}} \Gamma + 1 \vdash R \mid Q''}$$

with $Q' \cong Q''$, and noting that $\mathsf{pop}\ 0$ retracts $\mathsf{push} + 1$ (Lemma 2 below).

Sub-case: extrusions of different binders. In this case the residuals F'/F and F/F' remain bound outputs. Then, with the $\mathsf{push}^* E$ derivation as before, we can derive:

$$\mathsf{push}^* E \mid^{\tau}_{\nu} \cdot \dfrac{F'/F \dfrac{\vdots}{\Gamma + 1 \vdash S \xrightarrow{\overline{x+1}} \Gamma + 2 \vdash Q'}}{\Gamma + 1 \vdash \mathsf{push}^* P \mid S \xrightarrow{\tau} \Gamma + 1 \vdash \nu((\mathsf{push} + 1)^* R \mid Q')}$$

and for the other residual:

$$\nu^{\overline{u}} \cdot \dfrac{R \mid^{\overline{u+1}} \cdot \dfrac{F/F' \dfrac{\vdots}{\Gamma + 1 \vdash S' \xrightarrow{\overline{u+1}} \Gamma + 2 \vdash Q''}}{\Gamma + 1 \vdash R \mid S' \xrightarrow{\overline{u+1}} \Gamma + 2 \vdash \mathsf{push}^* R \mid Q''}}{\Gamma \vdash \nu(R \mid S') \xrightarrow{\overline{u}} \Gamma + 1 \vdash \nu(\mathsf{swap}^* \mathsf{push}^* R \mid \mathsf{swap}^* Q'')}$$

with $\mathsf{swap}^* Q' \cong Q''$. It remains to establish a $\cong$-path between the two terminal processes. We have $Q' \cong \mathsf{swap}^* Q''$ by functionality and involutivity of swap, and $\mathsf{push} + 1 = \mathsf{swap} \circ \mathsf{push}$ by Lemma 5 and then the rest follows by reflexivity and congruence.

C.2 Cofinality for Theorem 4

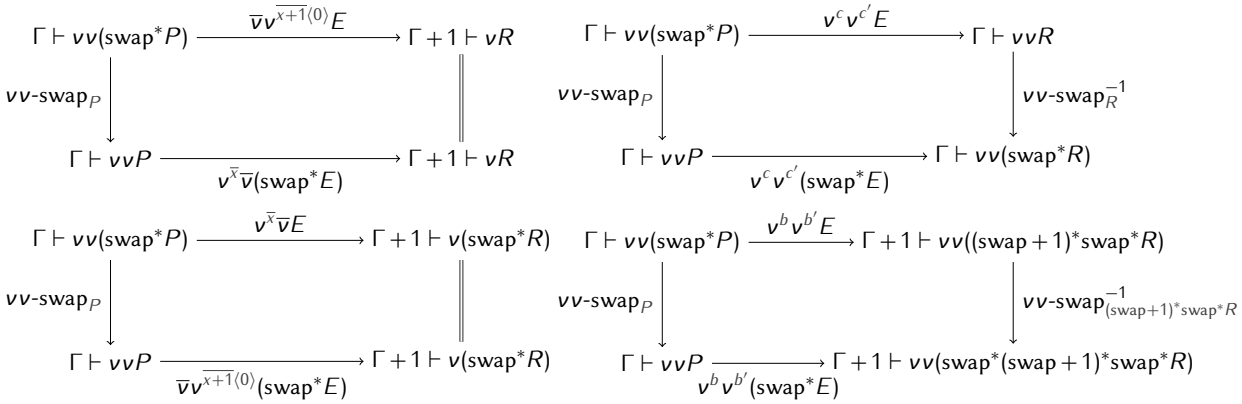


Figure 9: Cofinality of ϕ/E and E/ϕ in the $\nu\nu$-swap cases

Figure 9 illustrates cofinality for the $\nu\nu$-swap cases, omitting the renaming lemmas used as type-level coercions. The $\nu\nu$-swap^{-1} cases are symmetric.

Equations for Hereditary Substitution in Leivant's Predicative System F: a case study

Cyprien Mangin
Univ Paris Diderot & École Polytechnique
Paris, France
cyprien.mangin@m4x.org

Matthieu Sozeau
Inria Paris & PPS, Univ Paris Diderot
Paris, France
matthieu.sozeau@inria.fr

This paper presents a case study of formalizing a normalization proof for Leivant's Predicative System F [6] using the EQUATIONS package. Leivant's Predicative System F is a stratified version of System F, where type quantification is annotated with kinds representing universe levels. A weaker variant of this system was studied by Stump & Eades [5, 3], employing the hereditary substitution method to show normalization. We improve on this result by showing normalization for Leivant's original system using hereditary substitutions and a novel multiset ordering on types. Our development is done in the COQ proof assistant using the EQUATIONS package, which provides an interface to define dependently-typed programs with well-founded recursion and full dependent pattern-matching. EQUATIONS allows us to define explicitly the hereditary substitution function, clarifying its algorithmic behavior in presence of term and type substitutions. From this definition, consistency can easily be derived. The algorithmic nature of our development is crucial to reflect languages with type quantification, enlarging the class of languages on which reflection methods can be used in the proof assistant.

1 Introduction

EQUATIONS [10] is a toolbox built as a plugin on top of the COQ proof assistant for writing dependently-typed programs in COQ. Given a high-level specification of a function, using dependent pattern-matching and complex recursion schemes, its purpose is to compile it to pure COQ terms. This compilation scheme builds on the work of Goguen et al [4], which explains dependent pattern-matching [2] in terms of manipulations of propositional equalities. In essence, dependent pattern-matching is compiled away using a reduction-preserving encoding with eliminators for the equality type between datatypes. In addition to this compilation scheme, EQUATIONS also automatically derives the resulting equations as propositional equalities, abstracting entirely from the encoding of pattern-matching found in the actual compiled definition, and an elimination scheme corresponding to the graph of the function. This elimination scheme can then be used to simplify proofs that directly follow the case-analysis and recursion behavior of the function without repeating it. EQUATIONS supports definitions using arbitrarily complex well-founded recursion schemes, including the nested kind of recursion found in hereditary substitution functions, and the generation of unfolding lemmas and elimination schemes for those as well. Additionally, EQUATIONS plays well with the Program extension of COQ to manipulate subset types (also known as refinement types).

The purpose of this paper is to present a case study of using EQUATIONS to show the normalization of Predicative System F, in an algorithmic way such that the normalization function can actually be run inside the proof assistant.

Predicative System F was introduced by Leivant [6] to study the logical strength of different extensions of arithmetic. Using kinds to represent levels of allowed predicative quantification, he can show

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 71–86, doi:10.4204/EPTCS.185.5

that super-elementary functions can be represented in this system. He employs a Tait-Girard logical relation proof to argue normalization.

Stump and Eades [5] took the system and studied a normalization proof using hereditary substitution. However, they needed to define a variant of the system where the kinding rule of universal quantifications is more restrictive, which makes level $n+1$ not closed by quantifications over types in level n. They claim that the same derivations can be done in their system but give no proof of such fact. We remedy this situation by giving a simple normalization proof still based on hereditary substitution using a novel ordering on types based on multisets of kinds.

The hereditary substitution function ends up being defined as one would do in e.g. ML, but combining Program and EQUATIONS, it can be shown to inhabit a richer type, providing a proof that the function indeed computes normal forms given inputs in normal form. The pre and postconditions of the hereditary subsitution function, which will be explicited later, are actually necessary to justify the termination of this function. From this it is easy to derive normalization and show that the system is consistent (relatively to Coq's theory, *with the K axiom* currently, although we hope to have an axiom-free version working by the time of the workshop).

The paper is organized as follows: in §2 we present a gentle introduction to the EQUATIONS package and its features and quickly explain the main differences with the original presentation from [10]. Then we summarize the standard definitions and metatheoretical results on Predicative System F that we proved and highlight the main differences with the presentation of [5]. We provide the COQ development of our proof supplemented with some commentary to help follow along. First in §3 we present the language definition, with its typing and reduction rules. Then we show in §4 some metatheoretical properties on this language, such as substitution lemmas and regularity. Section 5 is dedicated to showing strong normalization of the calculus, which includes defining a well-founded ordering to justify the termination of the hereditary substitution function, and its definition itself using the EQUATIONS package. We also provide as an appendix the code which is produced from this definition by COQ's extraction mechanism. Finally, we compare with related work and conclude in §6.

2 Equations

EQUATIONS allows one to write recursive functions by specifying a list of clauses with a pattern on the left and a term on the right, à la Agda [9] and Epigram [8]. Here is an example recursive definition on lists, where wildcards corresponds to arbitrary fresh variables in patterns:

```
Equations length {A} (l : list A) : nat :=
length _ [] ⇒ 0;
length _ (cons _ t) ⇒ S (length t).
```

The package starts by building a splitting tree for the definition and then compiles it to a pure COQ term. From the splitting tree, it also derives the equations as propositional equalities, which can be more robust to use than reduction when writing proofs about the constant, although in this particular case the compiled definition is the same as the one from the standard library. Here we have two leaves in the computation tree hence two equations:

```
Check length_equation_1 : ∀ A : Type, length [] = 0.
Check length_equation_2 : ∀ (A : Type) (a : A) (l : list A), length (cons a l) = S (length l).
```

These two equations are automatically added to a rewrite hint database named length and can be used during proofs using the *simp* length tactic. In addition, an elimination principle for length is derived. Note

that length_comp is just a definition of the return type of length in terms of its arguments, i.e. it is $\lambda\ A\ l$, nat here:

```
Check length_elim : ∀ P : ∀ (A : Type) (l : list A), length_comp l → Prop,
    (∀ A : Type, P A [] 0) →
    (∀ (A : Type) (a : A) (l : list A), P A l (length l) → P A (cons a l) (S (length l))) →
    ∀ (A : Type) (l : list A), P A l (length l).
```

This elimination principle can be used in proofs to eliminate *calls* to length and refine at the same time the arguments and results of the call in the goal. For example, to prove the following lemma, one can apply the functional elimination principle using the *funelim* tactic to eliminate the length l call:

```
Lemma length_rev {A} (l : list A) : length (rev l) = length l.
Proof.
  funelim (length l).
```

We get two subgoals, easily solved by simplification and arithmetic.

```
  A : Type
  ============================
   length (rev []) = 0

  A : Type
  a : A
  l : list A
  H : length (rev l) = length l
  ============================
   length (rev (cons a l)) = S (length l)

  - simp length.
  - simpl; rewrite length_app, H; simp length; omega.
Qed.
```

2.1 Dependent Pattern-Matching

EQUATIONS handles not only simple pattern-matching on inductive types, but also dependent pattern-matching on inductive *families*. With respect to the standard COQ match construct, it eases the definition of complex pattern-matchings by compiling in the proof term all the inversion and unification steps that must be witnessed. Here is an example with the le relation on natural numbers.

```
Inductive le : nat → nat → Set :=
| lz : ∀ {n}, 0 ≤ n
| ls : ∀ {m n}, m ≤ n → (S m) ≤ (S n) where "m ≤ n" := (le m n).
```

Proving antisymmetry of this relation requires only two cases, because pattern matching on the first argument determines the endpoints of the second argument:

```
Equations antisym {m n : nat} (x : m ≤ n) (y : n ≤ m) : m = n :=
antisym _ _ x y by rec x ⇒
antisym _ _ lz lz ⇒ eq_refl;
```

```
antisym _ _ (ls x) (ls y) ⇒ f_equal S (antisym x y).
```

More precisely, in the x = lz case, it is possible during the translation to deduce that m must be 0, which implies that y cannot be some application of ls. These deductions are done automatically by EQUATIONS, which allows to reduce this proof to its simplest form. Writing it explicitely in pure COQ would be actually annoying and require explicit mention of impossible cases and surgical rewritings with equalities.

2.2 Recursion

Note that we use a clause by *rec x* ⇒ here in addition to the pattern-matching. This is a different kind of right-hand-side, that allows to specify the recursion scheme of the function. We are using well-founded recursion on the $m \leq n$ hypothesis here. The implicit ordering used is actually automatically derived using a *Derive Subterm* `for` le command, and corresponds to the transitive closure of the direct subterm relation, i.e. the deep structural recursion ordering.

The compiled definition cannot be checked using the built-in structural guardness check of COQ, because the equality manipulations appearing in the term go outside of the subset of recursion schemes recognized by it. It would have to handle commutative cuts and specific constructs on the equality type. Also, the syntactic check can be very slow on medium-sized terms.

The solution here, using the logic to justify the recursive calls, means that we are freed from any syntactic restriction, and any logical justification for termination is allowed. At each recursive call, we must simply provide a proof that the given argument is strictly smaller than the initial one in the subterm relation. An automatic proof search using the constructors of the subterm relation for le solves these subgoals for us here, otherwise they are given as obligations for the user to prove.

As in the case of length, we provide equations and an elimination principle for the definition. In case well-founded recursion is used, we first prove an unfolding lemma for the definition which allows us to remove any reasoning on the termination conditions after the definition. The equations are as expected:

```
Check antisym_equation_1 : antisym lz lz = eq_refl.
Check antisym_equation_2 : ∀ (n1 m0 : nat) (l : n1 ≤ m0) (l0 : m0 ≤ n1),
    antisym (ls l) (ls l0) = f_equal S (antisym l l0).
```

And the elimination principle, with the correct inductive hypothesis in the recursive case:

```
Check antisym_elim : ∀ P : ∀ (m n : nat) (x : m ≤ n) (y : n ≤ m), antisym_comp x y → Prop,
    P 0 0 lz lz eq_refl
    → (∀ (n1 m0 : nat) (l : n1 ≤ m0) (l0 : m0 ≤ n1), P n1 m0 l l0 (antisym l l0) →
       P (S n1) (S m0) (ls l) (ls l0) (f_equal S (antisym l l0))) →
  ∀ (m n : nat) (x : m ≤ n) (y : n ≤ m), P m n x y (antisym x y).
```

The last feature of EQUATIONS necessary to write real definitions is the `with` construct. This construct allows to do pattern-matching on intermediary results in a definition. A typical example is the filter function on lists, which selects all elements of the original list respecting some boolean predicate:

```
Context {A} (p : A → bool).

Equations filter (l : list A) : list A :=
filter [] := [] ;
filter (cons a l) ⇐ p a ⇒ { | true := cons a (filter l) ; | false := filter l }.
```

The ⇐ p a ⇒ right-hand side adds a new pattern to the left-hand side of its subprogram, for an object of type bool here. The subprogram is actually defined as another proxy constant, which takes as

arguments the variables a, p and a new variable of type bool. The clauses of the subprogram can shortcut the filter (cons a l) part of the pattern which is automatically inferred from the enclosing left-hand side.

The generated equations for such definitions go through the proxy constant, hence we have two equations for filter and two for *filter_helper_1*, which is the name of the proxy constant. To generate the elimination principle, a mutually inductive graph is generated, and the predicate applying to the subprogram is defined in terms of the original one, adding an equality between the new variable and the exact term it is applied to in the enclosing program. This way, we cannot forget during proofs that the true or false cases are actually results of a call to p a. Note that there are three leaves in the original program (and splitting tree) hence three cases to consider here.

```
Check ⋆filter_elim : ∀ (A : Type) (p : A → bool) (P : list A → list A → Prop)
 (P0:=λ (a : A) (refine : bool) (l H : list A) , p a = refine → P (cons a l) H) ,
  P [] [] → (∀ (a : A) (l : list A) , P l (filter p l) → P0 a true l (cons a (filter p l))) →
  (∀ (a : A) (l : list A) , P l (filter p l) → P0 a false l (filter p l)) →
  ∀ l : list A , P l (filter p l).
```

In general, the term used as a new discriminee is abstracted from the context and return type at this point of the program before checking the subprogram. In that case the eliminator predicate for the subprogram has a dependent binding for the t = refine hypothesis that is used to rewrite in the type of hypotheses and results. This is examplified in the following classical example:

```
Inductive incl {A} : list A → list A → Prop :=
  stop : incl nil nil
| keep {x : A} {xs ys : list A} : incl xs ys → incl (cons x xs) (cons x ys)
| skip {x : A} {xs ys : list A} : incl xs ys → incl (xs) (cons x ys).
```

We define list inclusion inductively and show that filtering out some elements from a list xs results in a included list.

```
Equations(nocomp) sublist {A} (p : A → bool) (xs : list A) : incl (filter p xs) xs :=
sublist A p nil := stop ;
sublist A p (cons x xs) with p x := {
  | true := keep (sublist p xs) ; | false := skip (sublist p xs) }.
```

Here at the with node, the return type is incl (if p x then cons x (filter p xs) else filter p xs) (cons x xs). We abstract p x from the return type and check the new subprogram in context A P x xs (refine : bool) with return type: incl (if refine then cons x (filter p) xs else filter p xs) (cons x xs).

Each of the patterns instantiates refine to a constructor, so the return type reduces to the two expected cases matching with conclusions of the incl relation. The (*nocomp*) option indicates that we do not want the return type to be defined using a *_comp* constant. Indeed, the term keep (sublist p xs) would not be well-typed, as it is expected to have type *sublist_comp* p (x :: xs) which is incl (filter p (x :: xs)) (x :: xs), but has type incl (x :: filter p xs) (x :: xs). This is just a technical limitation we hope to remove in the future.

```
Check ⋆sublist_elim : ∀ (P : ∀ (A : Type) (p : A → bool) (xs : list A) , incl (filter p xs) xs → Prop)
(P0:= λ (A : Type) (p : A → bool) (a : A) (refine : bool) (l : list A)
 (H : incl (filter_obligation_2 (filter p) a refine l) (cons a l)) ,
 ∀ Heq : p a = refine , P A p (cons a l)
    (eq_rect_r (λ r : bool , incl (filter_obligation_2 (filter p) a r l) (cons a l)) H Heq)) ,
  (∀ (A : Type) (p : A → bool) , P A p [] stop) →
  (∀ (A : Type) (p : A → bool) (a : A) (l : list A) ,
```

```
    P A p l (sublist p l) → P0 A p a true l (keep (sublist p l))) →
  (∀ (A : Type) (p : A → bool) (a : A) (l : list A),
    P A p l (sublist p l) → P0 A p a false l (skip (sublist p l))) →
  ∀ (A : Type) (p : A → bool) (xs : list A), P A p xs (sublist p xs).
```

The resulting elimination principle, while maybe not so useful in that case as this program constructs a *proof*, shows the explicit rewriting needed in the definition of the subpredicate *P0*.

This concludes our exposition of EQUATIONS and we now turn to the formalization of Predicative System F.

3 Typing and reduction

3.1 Definition of terms

Recall that Predicative System F is a typed lambda calculus with type abstractions and applications. Our type structure is very simple here, with just the function space and universal quantification on kinded type variables. We use an absolutely standard de Bruijn encoding for type and term variables. The kinds (a.k.a universe levels) are represented using natural numbers. Our development is based on Jérôme Vouillon's solution to the POPLmark challenge for System F^{sub} [11].

```
Definition kind := nat.

Inductive typ : Set :=
  | tvar : nat → typ | arrow : typ → typ → typ | all : kind → typ → typ.
```

We will write $\forall\, X :^* k.\ T$ for the quantification over types of kind k.

```
Inductive term : Set :=
  | var : nat → term
  | abs : typ → term → term
  | app : term → term → term
  | tabs : kind → term → term
  | tapp : term → typ → term.
```

Our raw terms are simply the abstract syntax trees.

3.2 Shiftings and substitutions

We define the different operations of shifting and substitutions with the EQUATIONS package, we only show the substitution function here which uses a `with` right-hand side. All the development can be downloaded or browsed at `http://equations-fpred.gforge.inria.fr`.

```
Check shift_typ : ∀ (X : nat) (t : term), term.
Check tsubst : typ → nat → typ → typ.

Equations subst (t : term) (x : nat) (t' : term) : term :=
subst (var y) x t' ⇐ lt_eq_lt_dec y x ⇒ {
  | inleft (left _) ⇒ var y;
  | inleft (right _) ⇒ t';
  | inright _ ⇒ var (y - 1) };
subst (abs T1 t2) x t' ⇒ abs T1 (subst t2 (1 + x) (shift 0 t'));
```

```
subst (app t1 t2) x t' ⇒ app (subst t1 x t') (subst t2 x t');
subst (tabs k t2) x t' ⇒ tabs k (subst t2 x (shift_typ 0 t'));
subst (tapp t1 T2) x t' ⇒ tapp (subst t1 x t') T2.

Check subst_typ : term → nat → typ → term.
```

3.3 Contexts

We define the contexts env and the two functions get_kind and get_var which access the context. A context is an interleaving of types and terms contexts. Vouillon's great idea is to have parallel de Bruijn indexings for type and term variables, which means separate indices for type and term variables. That way, shifting and substitution of one kind does not influence the other, making weakening and substitution lemmas much simpler, we follow this idea here.

```
Inductive env : Set :=
  | empty : env | evar : env → typ → env | etvar : env → kind → env.
```

Note that EQUATIONS allows wildcards and overlapping clauses with a first match semantics, as usual.

```
Equations(nocomp) get_kind (e : env) (X : nat) : option kind :=
get_kind empty _ ⇒ None;
get_kind (evar e _) X ⇒ get_kind e X;
get_kind (etvar _ T) O ⇒ Some T;
get_kind (etvar e _) (S X) ⇒ get_kind e X.
```

We need the functorial map on option types to ease writing these partial lookup functions.

```
Equations opt_map (A B : Set) (f : A → B) (x : option A) : option B :=
opt_map _ _ f (Some x) ⇒ Some (f x);
opt_map _ _ _ None ⇒ None.

Equations(nocomp) get_var (e : env) (x : nat) : option typ :=
get_var empty _ ⇒ None;
get_var (etvar e _) x ⇒ opt_map (tshift 0) (get_var e x);
get_var (evar _ T) O ⇒ Some T;
get_var (evar e _) (S x) ⇒ get_var e x.
```

3.4 Well-formedness conditions

We also define some well-formedness conditions for types, terms and contexts. Namely, in a type (resp. in a term), the variables must all be kinded (resp. typed). We just show the wf_typ definition here, those follow Stump and Haye's work.

```
Equations wf_typ (e : env) (T : typ) : Prop :=
wf_typ e (tvar X) ⇒ get_kind e X ≠ None;
wf_typ e (arrow T1 T2) ⇒ wf_typ e T1 ∧ wf_typ e T2;
wf_typ e (all k T2) ⇒ wf_typ (etvar e k) T2.
```

3.5 Kinding and typing rules

The kinding rules are the main difference between Leivant's and Stump's presentations. We refer to these works for pen and paper presentations of these systems, due to lack of space, we cannot include them here. The case for universal quantification sets the level of a universal type at 1 + max *k k'*, where *k* and *k'* are respectively the domain and codomain kinds, in Stump's case, which allows for a straightforward order on types based on levels, but this means that each level is not closed under products from lower levels anymore. In other words, multiple quantifications at the same level raise the overall level. For example (∀ *X* :* 0. *X*) :* 1 as expected but ∀ *X* :* 0. ∀ *Y* :* 0. *X* :* (1 + max 0 (1 + max 0 0)) = 2. This is a very strange behavior.

We use the standard predicative product rule which sets the product level to max (*k*+1) *k'*, which directly corresponds to Martin-Löf's Predicative Type Theory. Note that the system includes cumulativity through the Var rule which allows to lift a type variable declared at level *k* into any higher level *k'*.

```
Inductive kinding : env → typ → kind → Prop :=
| T_TVar : ∀ (e : env) (X : nat) (k k' : kind), wf_env e →
    get_kind e X = Some k → k ⇐ k' → kinding e (tvar X) k'
| T_Arrow : ∀ e T U k k', kinding e T k → kinding e U k' → kinding e (arrow T U) (max k k')
| T_All : ∀ e T k k', kinding (etvar e k) T k' → kinding e (all k T) (max (k+1) k').
```

The typing relation is straightforward. Just note that we check for well-formedness of environments at the variable case, so typing derivations are always well-formed.

```
Inductive typing : env → term → typ → Prop :=
  | T_Var (e : env) (x : nat) (T : typ) : wf_env e → get_var e x = Some T → typing e (var x) T
  | T_Abs (e : env) (t : term) (T1 T2 : typ) :
      typing (evar e T1) t T2 → typing e (abs T1 t) (arrow T1 T2)
  | T_App (e : env) (t1 t2 : term) (T11 T12 : typ) :
      typing e t1 (arrow T11 T12) → typing e t2 T11 → typing e (app t1 t2) T12
  | T_Tabs (e : env) (t : term) (k : kind) (T : typ) :
      typing (etvar e k) t T → typing e (tabs k t) (all k T)
  | T_Tapp (e : env) (t : term) k (T1 T2 : typ) :
      typing e t (all k T1) → kinding e T2 k → typing e (tapp t T2) (tsubst T1 0 T2).
```

3.6 Reduction rules

To define normalization we must formalize the reduction relation of the calculus. Beta-redexes for this calculus are the application of an abstraction to a term and the application of a type abstraction to a type.

```
Inductive red : term → term → Prop :=
  | E_AppAbs (T : typ) (t1 t2 : term) : red (app (abs T t1) t2) (subst t1 0 t2)
  | E_TappTabs k (T : typ) (t : term) : red (tapp (tabs k t) T) (subst_typ t 0 T).
```

We define the transitive closure of reduction on terms by closing `red` by context.

```
Inductive sred : term → term → Prop :=
  | Red_sred t t' : red t t' → sred t t'
  | sred_trans t1 t2 t3 : sred t1 t2 → sred t2 t3 → sred t1 t3
  | Par_app_left t1 t1' t2 : sred t1 t1' → sred (app t1 t2) (app t1' t2)
  | Par_app_right t1 t2 t2' : sred t2 t2' → sred (app t1 t2) (app t1 t2')
  | Par_abs T t t' : sred t t' → sred (abs T t) (abs T t')
```

```
| Par_tapp t t' T : sred t t' → sred (tapp t T) (tapp t' T)
| Par_tabs k t t' : sred t t' → sred (tabs k t) (tabs k t').

Definition reds t n := clos_refl sred t n.
```

We can prove usual congruence lemmas on reds showing that it indeed formalizes parallel reduction.

4 Metatheory

The metatheory of the system is pretty straightforward and follows the one of F^{sub} closely. We only mention the main idea for type substitution and the statements of the main metatheoretical lemmas.

To formalize type substitution, we use a proposition env_subst that corresponds to the environment operation: $E, X :* k, E' \Rightarrow E, (X \mapsto T')\ E'$ assuming $E \vdash T' :* k$. In other words, env_subst $X\ T\ e\ e'$ holds whenever we can find environments E, E' and a kind k such that $E \vdash T' :* k$ and $e = E, X :* k, E'$ and $e' = E, (X \mapsto T')\ E'$.

```
Inductive env_subst : nat → typ → env → env → Prop :=
| es_here (e : env) (T : typ) (k : kind) : kinding e T k → env_subst 0 T (etvar e k) e
| es_var (X : nat) (T T' : typ) (e e' : env) :
    env_subst X T' e e' → env_subst X T' (evar e T) (evar e' (tsubst T X T'))
| es_kind (X : nat) k (T' : typ) (e e' : env) :
    env_subst X T' e e' → env_subst (1 + X) (tshift 0 T') (etvar e k) (etvar e' k).
```

4.1 Typing and well-formedness

Actually, both kinding and typing imply well-formedness. In other words, it is possible to kind a type T in an environment e only if both the type and the environment are well-formed.

```
Lemma kinding_wf (e : env) (T : typ) (k : kind) : kinding e T k → wf_env e ∧ wf_typ e T.
```

4.2 Weakening

We only show the main weakening lemma for typing: if e' results from e by inserting a type variable at position X with any kind, the term and types of a typing derivation can be shifted accordingly to give a new typing derivation in the extended environment.

```
Lemma typing_weakening_kind_ind (e e' : env) (X : nat) (t : term) (U : typ) :
  insert_kind X e e' → typing e t U → typing e' (shift_typ X t) (tshift X U).
```

Weakening by a term variable preserves typing as well.

```
Lemma typing_weakening_var (e : env) (t : term) (U V : typ) :
  wf_typ e V → typing e t U → typing (evar e V) (shift 0 t) U.
```

4.3 Narrowing

As the system includes a kind of subtyping relation due to level cumulativity, we can prove a narrowing property for derivations. Again we define a judgment formalizing that a context e' is a narrowing of a context e if they are identical but for one type variable binding $(T : k')$ in e' and $(T : k)$ in e with $k' < k$.

```
Inductive narrow : nat → env → env → Set :=
    narrow_0 (e : env) (k k' : kind) : k' < k → narrow 0 (etvar e k) (etvar e k')
  | narrow_extend_kind (e e' : env) (k : kind) (X : nat) :
      narrow X e e' → narrow (1 + X) (etvar e k) (etvar e' k)
  | narrow_extend_var (e e' : env) (T : typ) (X : nat) :
      wf_typ e' T → narrow X e e' → narrow X (evar e T) (evar e' T).
```

Before we can show narrowing, we have to show that kinding respects cumulativity: If it is provable that a type T has kind k in the context e, then we can also prove that it has any kind k' for $k \leq k'$.

```
Lemma kinding_transitive e T k k' : kinding e T k → k ≤ k' → kinding e T k'.
```

Narrowing is a strong property, in the sense that a type T can have in a narrowing of a context e any kind that it can have in e itself.

```
Lemma typing_narrowing_ind (e e' : env) (X : nat) (t : term) (U : typ) : narrow X e e' → typing e t U
→ typing e' t U.
```

4.4 Substitution

Now, substitution lemmas can be proven for the various substitution functions.

```
Lemma subst_preserves_typing (e : env) (x : nat) (t u : term) (V W : typ) :
  typing e t V → typing (remove_var e x) u W → get_var e x = Some W → typing (remove_var e x)
(subst t x u) V.

Lemma subst_typ_preserves_typing (e : env) (t : term) (U P : typ) k :
  typing (etvar e k) t U → kinding e P k → typing e (subst_typ t 0 P) (tsubst U 0 P).
```

Finally, we prove regularity, which is to say that the type of any well-typed term is kinded. This is a consequence of the fact that any well-formed type is kindable. All these results correspond directly to the paper proofs of Stump and Hayes.

```
Theorem regularity (e : env) (t : term) (U : typ) : typing e t U → ∃ k : kind, kinding e U k.
```

5 Normalization

To show that hereditary substitution is well-defined, we must provide an order of termination. In our case, we will have a lexicographic combination of a multiset ordering on kinds. To formalize this, we reuse CoLoR's [1] library of multisets and definition of the multiset order. Those are multisets on ordered types, here natural numbers with the usual ordering, which is well-founded.

```
Notation "X <_m Y" := (MultisetLt gt X Y) (at level 70).

Definition wf_multiset_order : well_founded (MultisetLt gt).
```

The kinds_of function computes the multiset of kinds appearing in a type, which reduces to the bounds of universal quantifications.

```
Equations kinds_of (t : typ) : Multiset :=
kinds_of (tvar _) ⇒ empty; kinds_of (arrow T U) ⇒ union (kinds_of T) (kinds_of U);
kinds_of (all k T) ⇒ union {k} (kinds_of T).
```

Clearly, the singleton multiset built from any valid kind for T bounds the bag of kinds appearing in T, according to the kinding rules. This is proved by induction on the kinding derivation:

```
Lemma kinds_of_kinded e T k : kinding e T k → kinds_of T <mul { k }.
```

Kinds in a type are invariant by shifting or lifting. This is a simple example of a proof by functional elimination. The T argument and result of kinds_of T get refined and we just need to simplify the right hand sides according to the definitions of kinds_of and tshift, using rewriting not computation, and finish by rewriting with the induction hypotheses.

```
Lemma kinds_of_tshift X T : kinds_of (tshift X T) = kinds_of T.
Proof.
  funelim (kinds_of T); simp kinds_of tshift; now rewrite H, ?H0.
Qed.
```

For type substitution of T in U however, an exact arithmetic relation holds. We know that the multiset of kinds of the substituted type can appear a finite number of times in the resulting type, along with the original kinds of U.

```
Lemma kinds_of_tsubst e e' X T U k : env_subst X T e e' → kinding e U k →
  ∃ n : nat, kinds_of (tsubst U X T) =mul= kinds_of U + mul_sum n (kinds_of T).
```

This allows us to derive a general result about kindings of universal types: any well-kinded instance substitution produces a type with a strictly smaller bag of kinds. This is the central result needed to show termination. In Stump's work, the measure considered was solely the depth of types, and only through the stricter kinding invariant could the order be shown well-founded.

```
Lemma kinds_of_tsubst_all e U k k' T : kinding e (all k U) k' →
  kinding e T k → kinds_of (tsubst U 0 T) <mul kinds_of (all k U).
```

5.1 Definition of the measure.

We first define the depth of a type as being the number of universal quantifications and type variables in that type.

```
Equations depth (t : typ) : nat :=
depth (tvar _) ⇒ 1; depth (arrow T U) ⇒ (depth T + depth U)%nat;
depth (all k U) ⇒ S (depth U).
```

Of course, it cannot be zero, which is useful since it allows to have depth T < depth (arrow T U), which will be needed to prove the well-foundedness of the hereditary substitution.

```
Lemma depth_nz t : 0 < depth t.
```

The order that we will use on types is a lexicographical order on the multiset of kinds and the depth. As the first part of the lexicographic product is a multiset, and as those should not be compared with the Leibniz equality but rather a specific setoid equality, we defined a generalized notion of lexicographic product up-to an equivalence relation on the first component, here meq which represents multiset equality.

```
Definition relmd : relation (Multiset × nat) := lexprod (MultisetLt gt) meq lt.
```

It is well-founded, relying ultimately on the well-foundedness of lt.

```
Definition wf_relmd : well_founded relmd.
```

The actual order is relmd *on* the kinds_of and depth measures on types.

```
Definition order (x y : typ) := relmd (kinds_of x, depth x) (kinds_of y, depth y).
```

```
Definition wf_order : well_founded order.
```

It is well-founded and clearly transitive. `Lemma order_trans t u v : order t u → order u v → order t v.`

As we expected, we can compare a type with an arrow on that type, on the left and on the right.

```
Lemma order_arrow_l : ∀ A B, order A (arrow A B).
```

```
Lemma order_arrow_r : ∀ A B, order B (arrow A B).
```

We also define the reflexive closure of this order. It will be useful to express the postcondition of the hereditary substitution function, as we will explain below.

```
Definition ordtyp := clos_refl order.
```

Finally, we define the size of a term as usual.

```
Equations(nocomp) term_size (t : term) : nat :=
term_size (var _) ⇒ 0; term_size (abs T t) ⇒ S (term_size t);
term_size (app t u) ⇒ S (term_size t + term_size u);
term_size (tabs k t) ⇒ S (term_size t); term_size (tapp t U) ⇒ S (term_size t).
```

```
Definition wf_term_size : well_founded (MR term_size lt) := wf_inverse_image lt term_size lt_wf.
```

The hereditary substitution order is a lexicographic combination of the order on the multisets of kinds in the substituted term's type, the number of universal quantifiers and type variables in the substituted term's type, and the term size of the substituend. In other words, with U the type of the substituted term and t the substituend, we first compare the multiset of kinds in U, then the depth of U, and ultimately the size of t.

```
Definition her_order : relation (typ × term) :=
  lexprod order (fun x y ⇒ kinds_of x = kinds_of y ∧ depth x = depth y) (MR term_size lt).
```

```
Instance WF_her_order : WellFounded her_order.
```

5.2 The model

We now turn to the interpretation proper. We characterize the normal forms as a subset of the terms using mutually-inductive normal and neutral judgments. The plan is to show that the hereditary substitution function, when given two terms in normal form will produce terms in normal form. We can already expect some complications as normal terms also include neutral ones...

```
Inductive normal : term → Prop :=
| normal_abs T t : normal t → normal (abs T t)
| normal_tabs k t : normal t → normal (tabs k t)
| normal_neutral r : neutral r → normal r
with neutral : term → Prop :=
| neutral_var i : neutral (var i)
| neutral_app t n : neutral t → normal n → neutral (app t n)
| neutral_tapp t T : neutral t → neutral (tapp t T).
```

A term t is said to be a canonical inhabitant of a type T in environment e if $e \vdash t : T$ and t is in normal form. Our goal will be to show that every typeable term can be normalized to a canonical one.

```
Definition canonical e t T := typing e t T ∧ normal t.
```

We define a relation expressing that n is the interpretation of some arbitrary term t of type T and in environment e. `Definition interp e t T n := reds t n ∧ canonical e n T.`

5.3 Hereditary substitution

As we said, hereditary substitution takes two terms t and u in normal form and returns a term which is the result of substituting u in t at some index. From a purely algorithmic point of view, we only need t, u and the index X to compute the result of this function. However, we need more to prove its correctness.

First of all, the well-founded order that we use to justify its termination is an order on the type of the substituted and on the substituend, which is why the function hsubst also takes as an argument the type of the substituted term.

We then need a typing environment for t and u, which is not useful from a computational point of view but will serve to prove the termination and the correctness of hsubst. To this effect, we will decorate the function with a precondition and a postcondition. We define those in the universe of propositions to underline the fact that they are not useful in a computational way.

```
Definition pre (t : typ × term) (u : term) (X : nat) (p : env × typ) : Prop :=
    (get_var (fst p) X = Some (fst t) ∧ canonical (fst p) (snd t) (snd p)
     ∧ canonical (remove_var (fst p) X) u (fst t)).
```

There is one subtle point in the formulation of the postcondition. When we substitute in an application app *t1 t2*, it may be that the result of substituting in *t1* is an abstraction abs *T t*. If that's the case, to preserve the invariant that the result of hsubst is in normal form, we have to call again hsubst to perform the beta-reduction. However to do this, we need to know that the type of *t1* is smaller than the type of the substituted term. Note that we need to add a side-condition to this property, which is not always true: if the substituted variable did not appear at all, then there is no reason to have any relation between those types. There is a relation only if the original term was not an abstraction but the substituted term is.

```
Equations is_abs (t : term) : Prop :=
is_abs (abs _ _) ⇒ True; is_abs (tabs _ _) ⇒ True; is_abs _ ⇒ False.

Definition post (t : typ × term) (u : term) (X : nat) (r : term) (p : env × typ) : Prop :=
  interp (remove_var (fst p) X) (subst (snd t) X u) (snd p) r ∧
  (¬ is_abs (snd t) → is_abs r → ordtyp (snd p) (fst t)).
```

The Program mode proposed by Coq interacts nicely with Equations, in that it allows us to just return a term, and provide later the postcondition, thanks to subtyping of subset types. In the same way, we can treat a value returned by a call as if it was just the term. We use a standard encoding for the ghost p : env × typ variable. The (noind) option disables the generation of the graph and elimination principle for the function, its type and computational behavior is all we need here.

```
Equations(noind) hsubst (t : typ × term) (u : term) (X : nat) (P : ∃ (p : env × typ), pre t u X p) :
  {r : term | ∀ (p : env × typ), pre t u X p → post t u X r p} :=
hsubst t u X P by rec t her_order ⇒
hsubst (pair U t) u X P ⇐ t ⇒ {
```

```
| var i ⇐ lt_eq_lt_dec i X ⇒ {
  | inleft (right p) ⇒ u; | inleft (left p) ⇒ var i;
  | inright p ⇒ var (pred i) };
| abs T t ⇒ abs T (hsubst (U, t) (shift 0 u) (S X) _) ;
| tabs k t ⇒ tabs k (hsubst (tshift 0 U, t) (shift_typ 0 u) X _) ;
| tapp t T ⇐ hsubst (U, t) u X _ ⇒ {
  | exist (tabs k t') P ⇒ subst_typ t' 0 T;
  | exist r P ⇒ tapp r T };
| app t1 t2 ⇐ hsubst (U, t2) u X _ ⇒ {
  | exist r2 P2 ⇐ hsubst (U, t1) u X _ ⇒ {
    | exist (abs T' t') P1 ⇒ hsubst (T', t') r2 0 _;
    | exist r1 P1 ⇒ app r1 r2 } } }.
```

With hsubst defined, it is now easy to implement a normalize function which takes a term and returns its normal form. As for hsubst, we add a precondition and a postcondition which allow to show correctness by construction.

```
Definition pre' (t : term) (p : env × typ) : Prop :=
  typing (fst p) t (snd p).
Definition post' (t : term) (n : term) (p : env × typ) : Prop :=
  interp (fst p) t (snd p) n.

Equations(noind) normalize (t : term) (P : ∃ (p : env × typ), pre' t p) :
  {n : term | ∀ (p : env × typ), pre' t p → post' t n p} :=
normalize (var i) P ⇒ var i;
normalize (abs T1 t) P ⇒ abs T1 (normalize t _) ;
normalize (app t1 t2) P ⇐ normalize t2 _ ⇒ {
  | exist t2' P2' ⇐ normalize t1 _ ⇒ {
    | exist (abs T t) P1' ⇒ hsubst (T, t) t2' 0 _ ;
    | exist t1' P1' ⇒ app t1' t2' } };
normalize (tabs k t) P ⇒ tabs k (normalize t _) ;
normalize (tapp t T) P ⇐ normalize t _ ⇒ {
  | exist (tabs k t') P' ⇒ subst_typ t' 0 T ;
  | exist t' P' ⇒ tapp t' T }.
```

The existence of the normalize function is in itself a proof of the strong normalization of Leivant's Predicative System F.

```
Theorem normalization e t T : typing e t T → ∃ n, reds t n ∧ typing e n T ∧ normal n.
```

5.4 Consistency

It is easy to show consistency based on the normalization function. We just need lemmas showing that neutral terms cannot inhabit any type in an environment with only a type variable, by inversion on the neutrality derivation.

```
Lemma neutral_tvar t k T : neutral t → typing (etvar empty_env k) t T → False.
```

Consistency is then proved using case analysis on an assumed typing derivation of falsehood at any universe level k. Informally, it is showing that $\forall\ X :^* k$, X is not inhabited for any k.

```
Corollary consistency k : ¬ ∃ t, typing empty_env t (all k (tvar 0)).
```

6 Related Work and Conclusion

There are many formalizations of similar calculi, and we do not claim any originality there. However, to our knowledge, the multiset ordering used to show normalization is original. The point of this paper is more to show that the EQUATIONS plugin is ready to handle more consequent developments and showcase its features. Is has similar expressivity w.r.t. Agda and Idris, but derives more principles, and everything is compiled down to vanilla COQ terms, so it does not change the trusted code base except for the use of K, which we are hopeful we can get rid of by the time of the workshop.

It would be interesting to study extensions of the language with type recursion. As shown by Leivant, this would allow to type terms that are not typeable in second-order lambda calculus. We will also need to extend the language with existentials, pairs and a minimal notion of inductive types to be able to handle a larger class of programs. One of the possible venues for generalization is to extend the work of Malecha et al [7] to reflect a larger fragment of GALLINA, the language of COQ.

References

[1] Frédéric Blanqui: *CoLoR, a Coq Library on Rewriting and Termination*. Available at http://color.inria.fr.

[2] Thierry Coquand (1992): *Pattern Matching with Dependent Types*. Available at http://www.cs.chalmers.se/~coquand/pattern.ps. Proceedings of the Workshop on Logical Frameworks.

[3] Harley D Eades III (2014): *The semantic analysis of advanced programming languages*. Ph.D. thesis, The University of Iowa. Available at http://metatheorem.org/wp-content/papers/thesis.pdf.

[4] Conor McBride Healfdene Goguen & James McKinna (2006): *Eliminating Dependent Pattern Matching*. Available at http://cs.ru.nl/~james/RESEARCH/goguen2006.pdf.

[5] Harley D. Eades III & Aaron Stump (2010): *Hereditary Substitution for Stratified System F*. In: *International Workshop on Proof-Search in Type Theories*, A FLoC workshop, Edinburgh, Scotland. Available at http://homepage.divms.uiowa.edu/~astump/papers/pstt-2010.pdf.

[6] Daniel Leivant (1990): *Finitely stratified polymorphism*. Technical Report, Carnegie Mellon University. Available at http://repository.cmu.edu/cgi/viewcontent.cgi?article=2961&context=compsci.

[7] Gregory Malecha, Adam Chlipala & Thomas Braibant (2014): *Compositional Computational Reflection*. In Gerwin Klein & Ruben Gamboa, editors: *ITP'14*, *Lecture Notes in Computer Science* 8558, Springer, pp. 374–389, doi:10.1007/978-3-319-08970-6_24. Available at http://dx.doi.org/10.1007/978-3-319-08970-6.

[8] Conor McBride (2005): *Epigram: Practical Programming with Dependent Types*. *Advanced Functional Programming*, pp. 130–170, doi:10.1007/11546382_3.

[9] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. Available at http://www.cs.chalmers.se/~ulfn/papers/thesis.html.

[10] Matthieu Sozeau (2010): *Equations: A Dependent Pattern-Matching Compiler*. In: *First International Conference on Interactive Theorem Proving*, Springer, doi:10.1007/978-3-642-14052-5_29.

[11] Jérôme Vouillon: *POPLmark challenge solution*. Available at http://www.seas.upenn.edu/~plclub/poplmark/vouillon.html.

A Extracted code

```
let hereditary_subst t u x =
  let rec fix_F x0 =
    let h = fun y -> fix_F y in
    (fun u0 x1 _ ->
    let Pair (t0, t1) = x0 in
    (match t1 with
     | Var n ->
       (match lt_eq_lt_dec n x1 with
        | Inleft s ->
          (match s with
           | Left -> Var n
           | Right -> u0)
        | Inright -> Var (pred n))
     | Abs (t2, refine) ->
       Abs (t2, (h (Pair (t0, refine)) (shift O u0) (S x1) __))
     | App (refine1, refine2) ->
       (match h (Pair (t0, refine1)) u0 x1 __ with
        | Abs (t2, x2) ->
          h (Pair (t2, x2)) (h (Pair (t0, refine2)) u0 x1 __) O __
        | x2 -> App (x2, (h (Pair (t0, refine2)) u0 x1 __)))
     | Tabs (k, refine) ->
       Tabs (k, (h (Pair ((tshift O t0), refine)) (shift_typ O u0) x1 __))
     | Tapp (refine, t2) ->
       (match h (Pair (t0, refine)) u0 x1 __ with
        | Tabs (k, x2) -> subst_typ x2 O t2
        | x2 -> Tapp (x2, t2))))
  in fix_F t u x __

type normalize_comp = term

(** val normalize : term -> normalize_comp **)

let rec normalize = function
| Var n -> Var n
| Abs (t0, t1) -> Abs (t0, (normalize t1))
| App (t1, t2) ->
  (match normalize t1 with
   | Abs (t0, x0) -> hereditary_subst (Pair (t0, x0)) (normalize t2) O
   | x -> App (x, (normalize t2)))
| Tabs (k, t0) -> Tabs (k, (normalize t0))
| Tapp (t0, t1) ->
  (match normalize t0 with
   | Tabs (k, x) -> subst_typ x O t1
   | x -> Tapp (x, t1))
```

Rewriting Modulo β in the $\lambda\Pi$-Calculus Modulo

Ronan Saillard
MINES ParisTech, PSL Research University, France
ronan.saillard@mines-paristech.fr

The $\lambda\Pi$-calculus Modulo is a variant of the λ-calculus with dependent types where β-conversion is extended with user-defined rewrite rules. It is an expressive logical framework and has been used to encode logics and type systems in a shallow way. Basic properties such as subject reduction or uniqueness of types do not hold in general in the $\lambda\Pi$-calculus Modulo. However, they hold if the rewrite system generated by the rewrite rules together with β-reduction is confluent. But this is too restrictive. To handle the case where non confluence comes from the interference between the β-reduction and rewrite rules with λ-abstraction on their left-hand side, we introduce a notion of rewriting modulo β for the $\lambda\Pi$-calculus Modulo. We prove that confluence of rewriting modulo β is enough to ensure subject reduction and uniqueness of types. We achieve our goal by encoding the $\lambda\Pi$-calculus Modulo into Higher-Order Rewrite System (HRS). As a consequence, we also make the confluence results for HRSs available for the $\lambda\Pi$-calculus Modulo.

1 Introduction

The $\lambda\Pi$-calculus Modulo is a variant of the λ-calculus with dependent types ($\lambda\Pi$-calculus or LF) where β-conversion is extended with user-defined rewrite rules. Since its introduction by Cousineau and Dowek [8], it has been used as a logical framework to express different logics and type systems. A key advantage of rewrite rules is that they allow designing *shallow* embeddings, that is embeddings that preserve the computational content of the encoded system. It has been used, for instance, to encode functional Pure Type Systems [8], First-Order Logic [9], Higher-Order Logic [2], the Calculus of Inductive Constructions [4], resolution and superposition proofs [6], and the ς-calculus [7].

The expressive power of the $\lambda\Pi$-calculus Modulo comes at a cost: basic properties such as subject reduction or uniqueness of types do not hold in general. Therefore, one has to prove these properties for each particular set of rewrite rules considered. The usual way to do so is to prove that the rewriting relation generated by the rewrite rules together with β-reduction is confluent. This entails a property called product compatibility (also known as Π-injectivity or injectivity of function types) which, in turn, implies both subject reduction and uniqueness of types. Another important consequence of confluence is that, together with termination, it implies the decidability of the corresponding congruence. Indeed, for confluent and terminating relations, checking congruence boils down to a syntactic equality check between normal forms. As a direct corollary, we get the decidability of type checking in the $\lambda\Pi$-calculus Modulo for the corresponding rewrite relations.

One case where confluence is easily lost is if one allows rewrite rules with λ-abstractions on their left-hand side. For instance, consider the following rewrite rule (which reflects the mathematical equality $(e^f)' = f' * e^f$):

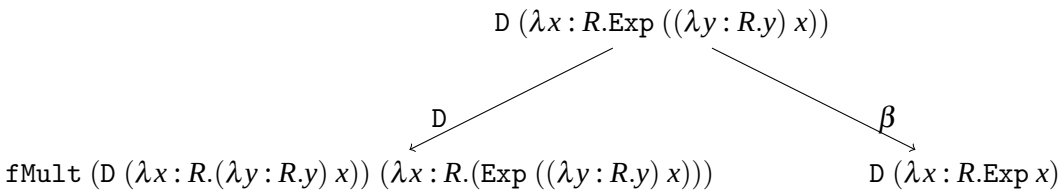
$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ (f\ x)) \hookrightarrow \mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.\, f\ x))\ (\lambda x : R.\mathtt{Exp}\ (f\ x)).$$

This rule introduces a non-joinable critical peak when combined with β-reduction:

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 87–101, doi:10.4204/EPTCS.185.6

x,y,z	$\in$	$\mathscr{V}$	(Variable)
$\underline{c},\underline{f}$	$\in$	$\mathscr{C}_O$	(Object Constant)
C,F	$\in$	$\mathscr{C}_T$	(Type Constant)
$\underline{t},\underline{u},\underline{v}$	::=	$x \mid \underline{c} \mid \underline{u}\ \underline{v} \mid \lambda x:U.\underline{t}$	(Object)
U,V	::=	$C \mid U\ \underline{v} \mid \lambda x:U.V \mid \Pi x:U.V$	(Type)
K	::=	$\mathbf{Type} \mid \Pi x:U.K$	(Kind)
t,u,v	::=	$\underline{u} \mid U \mid K \mid \mathbf{Kind}$	(Term)

Figure 1: The terms of the $\lambda\Pi$-calculus Modulo



A way to recover confluence is to consider a generalized rewriting relation where matching is done modulo β-reduction. In this setting $\mathtt{D}\ (\lambda x:R.\mathtt{Exp}\ x)$ is reducible because it is β-equivalent to the redex $\mathtt{D}\ (\lambda x:R.\mathtt{Exp}((\lambda y:R.y)\ x))$ and, as we will see, this allows closing the critical peak.

In this paper, we formalize the notion of *rewriting modulo* β in the context of the $\lambda\Pi$-calculus Modulo. We achieve this by encoding the $\lambda\Pi$-calculus Modulo into Nipkow's Higher-Order Rewrite Systems [14]. This encoding allows us, first, to properly define matching modulo β using the notion of higher order rewriting and, secondly, to make available, in the $\lambda\Pi$-calculus Modulo, confluence and termination criteria designed for higher-order rewriting. Then we prove that the assumption of confluence for the rewriting modulo β relation can be used, in most proofs, in place of standard confluence. In particular this implies subject reduction (for both standard rewriting and rewriting modulo β) and uniqueness of types.

The paper is organized as follows. First, we define in Section 2 the $\lambda\Pi$-calculus modulo for which we prove subject reduction and uniqueness of types under the assumption of product compatibility and we show that confluence implies this latter property. In Section 3, we show that a naive definition of rewriting modulo β does not work in a typed setting. This leads us to use Higher-Order Rewrite Systems which we present in Section 4 and in which we encode the $\lambda\Pi$-calculus Modulo in Section 5. Then, we use this encoding to properly define rewriting modulo β in Section 6 and generalize the results of the previous sections. We discuss possible applications in Section 7 before concluding in Section 8.

2 The $\lambda\Pi$-Calculus Modulo

The $\lambda\Pi$-calculus Modulo is an extension of the dependently-typed λ-calculus ($\lambda\Pi$-calculus) where the β-conversion is extended by user-defined rewrite rules.

2.1 Terms

The terms of the $\lambda\Pi$-calculus Modulo are the same as the terms of the $\lambda\Pi$-calculus. Their syntax is given in Figure 1.

$$\begin{array}{lcll} \Delta & ::= & \emptyset \mid \Delta(x:U) & \text{(Local Context)} \\ \Gamma & ::= & \emptyset \mid \Gamma(\underline{c}:U) \mid \Gamma(C:K) \mid \Gamma(\underline{u} \hookrightarrow \underline{v}) \mid \Gamma(U \hookrightarrow V) & \text{(Global Context)} \end{array}$$

Figure 2: Syntax for contexts

Definition 2.1 (Object, Type, Kind, Term)**.** *A* term *is either an* object*, a* type*, a* kind *or the symbol* **Kind***.*

An object is either a variable *in the set $\mathscr{V}$, or an* object constant *in the set $\mathscr{C}_O$, or an* application $\underline{u}\ \underline{v}$ *of two objects, or an* abstraction $\lambda x:A.\underline{t}$ *where A is a type and $\underline{t}$ is an object.*

A type is either a type constant *in the set $\mathscr{C}_T$, or an* application $U\ \underline{v}$ *where U is a type and $\underline{v}$ is an object, or an* abstraction $\lambda x:U.V$ *where U and V are types, or a* product $\Pi x:U.V$ *where U and V are types.*

A kind is either a product $\Pi x:U.K$ *where U is a type and K is a kind or the symbol* **Type***.*

Type *and* **Kind** *are called* sorts*.*

The sets $\mathscr{V}$, $\mathscr{C}_O$ and $\mathscr{C}_T$ are assumed to be infinite and pairwise disjoint.

Definition 2.2. *A term is* algebraic *if it is not a variable, it is built from constants, variables and applications and variables do not have arguments.*

Notation 2.1. *In addition to the naming convention of Figure 1, we use A and B to denote types or kinds; T to denote a type, a kind or* **Kind***; s for* **Type** *or* **Kind***.*

Moreover, we write $t\vec{u}$ to denote the application of t to an arbitrary number of arguments $u_1,\ldots,u_n$. We write $u[x/v]$ for the usual (capture-avoiding) substitution of x by v in u. We write $A \longrightarrow B$ for $\Pi x:A.B$ when B does not depend on x.

2.2 Contexts

We distinguish two kinds of context: local and global contexts. A local context is a list of typing declarations corresponding to variables. The syntax for contexts is given in Figure 2.

Definition 2.3 (Local Context)**.** *A* local context *is a list of variable declarations (variables together with their type).*

Following our previous work [17], we give a presentation of the $\lambda\Pi$-calculus Modulo where the rewrite rules are internalized in the system as part of the global context. This is a difference with earlier presentations [8] where the rewrite rules lived *outside* the system and were typed in an external system (either the simply-typed calculus or the $\lambda\Pi$-calculus). The main benefit of this approach is that the typing of the rewrite rules is made explicit and becomes an iterative process: rewrite rules previously added in the system can be used to type new ones.

Definition 2.4. *A* rewrite rule *is a pair of terms. We distinguish* object-level rewrite rules *(pairs of objects) from* type-level rewrite rules *(pairs of types).*

These are the only allowed rewrite rules. We write $(u \hookrightarrow v)$ for the rewrite rule (u,v).

It is left-algebraic *if u is algebraic and* left-linear *if no free variable occurs twice in u.*

Definition 2.5 (Global Context)**.** *A* global context *is a list of object declarations (an object constant together with a type), type declarations (a type constant together with a kind), object-level rewrite rules and type-level rewrite rules.*

$$\textbf{(Sort)}\quad \frac{}{\Gamma;\Delta \vdash \textbf{Type} : \textbf{Kind}}$$

$$\textbf{(Variable)}\quad \frac{(x:A)\in\Delta}{\Gamma;\Delta\vdash x:A}$$

$$\textbf{(Constant)}\quad \frac{(c:A)\in\Gamma}{\Gamma;\Delta\vdash c:A}$$

$$\textbf{(Application)}\quad \frac{\Gamma;\Delta\vdash t:\Pi x:A.B \qquad \Gamma;\Delta\vdash u:A}{\Gamma;\Delta\vdash tu:B[x/u]}$$

$$\textbf{(Abstraction)}\quad \frac{\Gamma;\Delta\vdash A:\textbf{Type} \qquad \Gamma;\Delta(x:A)\vdash t:B \qquad B\neq\textbf{Kind}}{\Gamma;\Delta\vdash \lambda x:A.t:\Pi x:A.B}$$

$$\textbf{(Product)}\quad \frac{\Gamma;\Delta\vdash A:\textbf{Type} \qquad \Gamma;\Delta(x:A)\vdash B:s}{\Gamma;\Delta\vdash \Pi x:A.B:s}$$

$$\textbf{(Conversion)}\quad \frac{\Gamma;\Delta\vdash t:A \qquad \Gamma;\Delta\vdash B:s \qquad A\equiv_{\beta\Gamma} B}{\Gamma;\Delta\vdash t:B}$$

Figure 3: Typing rules for terms in the $\lambda\Pi$-calculus Modulo.

2.3 Rewriting

Definition 2.6 (β-reduction)**.** *The* β-reduction *relation* $\to_\beta$ *is the smallest relation on terms containing* $(\lambda x:A.u)v \to_\beta u[x/v]$*, for all terms* A,u *and* v*, and closed by subterm rewriting.*

Definition 2.7 (Γ-reduction)**.** *Let* Γ *be a global context. The* Γ-reduction *relation* $\to_\Gamma$ *is the smallest relation on terms containing* $u\to_\Gamma v$ *for every rewrite rule* $(u\hookrightarrow v)\in\Gamma$*, closed by substitution and by subterm rewriting. We say that* $\to_\Gamma$ *is* left-algebraic *(respectively* left-linear*) if the rewrite rules in* Γ *are left-algebraic (respectively left-linear).*

Notation 2.2. *We write* $\to_{\beta\Gamma}$ *for* $\to_\beta\cup\to_\Gamma$*,* $\equiv_\beta$ *for the congruence generated by* $\to_\beta$ *and* $\equiv_{\beta\Gamma}$ *the congruence generated by* $\to_{\beta\Gamma}$*.*

It is important to notice that these notions of reduction are defined as relations on all (untyped) terms. In particular, we do not require the substitutions to be well-typed. This allows defining the notion of rewriting independently from the notion of typing (see below). This makes the system closer from what we would implement in practice.

Since the rewrite rules are either object-level or type-level, rewriting preserves the three syntactic categories (object, type, kind). Moreover, sorts are only convertible to themselves.

2.4 Type System

We now give the typing rules for the $\lambda\Pi$-calculus Modulo. We begin by the inference rules for terms, then for local contexts and finally for global contexts.

Definition 2.8 (Well-Typed Term)**.** *We say that a term* t has type A *in the global context* Γ *and the local context* Δ *if the judgment* $\Gamma;\Delta\vdash t:A$ *is derivable by the inference rules of Figure 3. We say that a term is* well-typed *if such* A *exists.*

(**Empty Local Context**) $$\frac{}{\Gamma \vdash^{ctx} \emptyset}$$

(**Variable Declaration**) $$\frac{\Gamma \vdash^{ctx} \Delta \qquad \Gamma;\Delta \vdash U : \textbf{Type} \qquad x \notin dom(\Delta)}{\Gamma \vdash^{ctx} \Delta(x : U)}$$

Figure 4: Typing rules for local contexts

The typing rules only differ from the usual typing rules for the $\lambda\Pi$-calculus by the **(Conversion)** rule where the congruence is extended from β-conversion to $\beta\Gamma$-conversion allowing taking into account the rewrite rules in the global context.

Definition 2.9 (Well-Formed Local Context)**.** *A local context Δ is* well-formed *with respect to a global context Γ if the judgment $\Gamma \vdash^{ctx} \Delta$ is derivable by the inference rules of Figure 4.*

Well-formed local contexts ensure that local declarations are unique and well-typed.

Besides the new conversion relation, the main difference between the $\lambda\Pi$-calculus and the $\lambda\Pi$-calculus Modulo is the presence of rewrite rules in global contexts. We need to take this into account when typing global contexts.

A key feature of any type system is the preservation of typing by reduction: the subject reduction property.

Definition 2.10 (Subject Reduction)**.** *Let Γ be a global context. We say that a rewriting relation $\rightarrow$ satisfies the* subject reduction *property in Γ if, for all terms t_1, t_2, T and local context Δ such that $\Gamma \vdash^{ctx} \Delta$, $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \rightarrow t_2$ imply $\Gamma;\Delta \vdash t_2 : T$.*

In the $\lambda\Pi$-calculus Modulo, we cannot allow adding arbitrary rewrite rules in the context, if we want to preserve subject reduction. In particular, to prove subject reduction for the β-reduction we need the following property:

Definition 2.11 (Product-Compatibility)**.** *We say that a global context Γ satisfies the* product compatibility *property (and we note $\mathbf{PC}(\Gamma)$) if the following proposition is verified:*
if $\Pi x : A_1.B_1$ and $\Pi x : A_2.B_2$ are two well-typed product types in the same well-formed local context such that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ then $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$.

On the other hand, subject reduction for the Γ-reduction requires rewrite rules to be well-typed in the following sense:

Definition 2.12 (Well-typed Rewrite Rules)**.**

- *A rewrite rule $(u \hookrightarrow v)$ is* well-typed *for a global context Γ if, for any substitution σ, well-formed local context Δ and term T, $\Gamma;\Delta \vdash \sigma(u) : T$ implies $\Gamma;\Delta \vdash \sigma(v) : T$.*
- *A rewrite rule is* permanently well-typed *for a global context Γ if it is well-typed for any extension $\Gamma_0 \supset \Gamma$ that satisfies product compatibility. We write $\Gamma \vdash u \hookrightarrow v$ when $(u \hookrightarrow v)$ is permanently well-typed in Γ.*

The notion of permanently well-typed rewrite rule makes possible to typecheck rewrite rules only once and not each time we make new declarations or add other rewrite rules in the context.

We can now give the typing rules for global contexts.

Definition 2.13 (Well-formed Global Context)**.** *A global context is* well-formed *if the judgment Γ **wf** is derivable by the inference rules of Figure 5.*

(**Empty Global Context**) $$\frac{}{\emptyset \textbf{ wf}}$$

(**Object Declaration**) $$\frac{\Gamma \textbf{ wf} \qquad \Gamma;\emptyset \vdash U : \textbf{Type} \qquad \underline{c} \notin dom(\Gamma)}{\Gamma(\underline{c} : U) \textbf{ wf}}$$

(**Type Declaration**) $$\frac{\Gamma \textbf{ wf} \qquad \Gamma;\emptyset \vdash K : \textbf{Kind} \qquad \textbf{PC}(\Gamma(C : K)) \qquad C \notin dom(\Gamma)}{\Gamma(C : K) \textbf{ wf}}$$

(**Rewrite Rules**) $$\frac{\Gamma \textbf{ wf} \qquad (\forall i)\Gamma \vdash u_i \hookrightarrow v_i \qquad \textbf{PC}(\Gamma(u_1 \hookrightarrow v_1) \dots (u_n \hookrightarrow v_n))}{\Gamma(u_1 \hookrightarrow v_1) \dots (u_n \hookrightarrow v_n) \textbf{ wf}}$$

Figure 5: Typing rules for global contexts

The rules **(Object Declaration)** and **(Type Declaration)** ensure that constant declarations are well-typed. One can remark that the premise $\textbf{PC}(\Gamma(\underline{c} : U))$ is *missing* in the **(Object Declaration)** rule. This is because $\textbf{PC}(\Gamma(\underline{c} : U))$ can be proved from $\textbf{PC}(\Gamma)$; to prove product compatibility for $\Gamma(c : U)$ it suffices to emulate the constant c by a fresh variable and use the product compatibility property of Γ. This cannot be done for type declarations since type-level variables do not exist in the $\lambda\Pi$-calculus Modulo. The rule **(Rewrite Rules)** permits adding rewrite rules. Notice that we can add several rewrite rules at once. In this case, only product compatibility for the whole system is required. On the other hand, when a rewrite rule is added it needs to be well-typed independently from the other rules that are added at the same time.

Well-formed global contexts satisfy subject reduction and uniqueness of types. Proofs can be found in the long version of this paper at the author's webpage.

Theorem 2.1 (Subject Reduction)**.** *Let Γ be a well-formed global context. Subject reduction holds for $\rightarrow_{\beta\Gamma}$ in Γ.*

Theorem 2.2 (Uniqueness of Types)**.** *Let Γ be a well-formed global context and let Δ be a local context well-formed for Γ. If $\Gamma;\Delta \vdash t : T_1$ and $\Gamma;\Delta \vdash t : T_2$ then $T_1 \equiv_{\beta\Gamma} T_2$.*

Remark that strong normalization of well-typed terms for the relations $\rightarrow_\Gamma$ and $\rightarrow_\beta$ is not guaranteed.

2.5 Criteria for Product Compatibility and Well-typedness of Rewrite Rules

We now give effective criteria for checking product compatibility and well-typedness of rewrite rules.

The usual way to prove product compatibility is by showing the confluence of the rewrite system.

Theorem 2.3 (Product Compatibility from Confluence)**.** *Let Γ be a global context. If $\rightarrow_{\beta\Gamma}$ is confluent then product compatibility holds for Γ.*

One could think that we can weaken the assumption of confluence requiring only confluence for well-typed terms. This is not a viable option since, without product compatibility, we do not know if reduction preserves typing (subject reduction) and if the set of well-typed terms is closed by reduction. Therefore, it seems unlikely to be able to prove confluence only for well-typed terms before proving the product compatibility property.

The confluence of $\rightarrow_{\beta\Gamma}$ can be obtained from the confluence of $\rightarrow_\Gamma$.

Theorem 2.4 (Müller [12])**.** *If $\rightarrow_\Gamma$ is left-algebraic, left-linear and confluent, then $\rightarrow_{\beta\Gamma}$ is confluent.*

To show that a rewrite rule is well-typed, one can use the following result:

Theorem 2.5. *Let Γ be a well-formed global context and $(u \hookrightarrow v)$ be a rewrite rule. If u is algebraic and there exist Δ and T such that $\Gamma \vdash^{ctx} \Delta$, $dom(\Delta) = FV(u)$, $\Gamma;\Delta \vdash u : T$ and $\Gamma;\Delta \vdash v : T$ then $(u \hookrightarrow v)$ is permanently well-typed for Γ.*

2.6 Example

As an example, we define the map function on lists of integers. We first define the type of *Peano integers* by the three successive global declarations:

$\texttt{Nat} : \textbf{Type}.$
$0 : \texttt{Nat}.$
$\texttt{S} : \texttt{Nat} \longrightarrow \texttt{Nat}.$

For readability, we will write n instead of $\overbrace{S\ (S\ \ldots (S}^{n\ times}\ 0))$. We now define a type for lists:

$\texttt{List} : \textbf{Type}.$
$\texttt{Nil} : \texttt{List}.$
$\texttt{Cons} : \texttt{Nat} \longrightarrow \texttt{List} \longrightarrow \texttt{List}.$

and the function map on lists:

$\texttt{Map} : (\texttt{Nat} \longrightarrow \texttt{Nat}) \longrightarrow \texttt{List} \longrightarrow \texttt{List}.$
$\texttt{Map}\ f\ \texttt{Nil} \hookrightarrow \texttt{Nil}.$
$\texttt{Map}\ f\ (\texttt{Cons}\ hd\ tl) \hookrightarrow \texttt{Cons}\ (f\ hd)\ (\texttt{Map}\ f\ tl).$

For instance, we can use this function to add some value to the elements of a list. First, we define addition:

$\texttt{plus} : \texttt{Nat} \longrightarrow \texttt{Nat} \longrightarrow \texttt{Nat}.$
$\texttt{plus}\ 0\ n \hookrightarrow n.$
$\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

Then, we have the following reduction:

$\texttt{Map}\ (\texttt{plus}\ 3)\ (\texttt{Cons}\ 1\ (\texttt{Cons}\ 2\ (\texttt{Cons}\ 3\ \texttt{Nil}))) \to^*_\Gamma \texttt{Cons}\ 4\ (\texttt{Cons}\ 5\ (\texttt{Cons}\ 6\ \texttt{Nil}))$.

This global context is well-formed. Indeed, one can check that each global declaration is well-typed. Moreover, each time we add a rewrite rule, it verifies the hypotheses of Theorem 2.5 and it preserves the confluence of the relation $\to_{\beta\Gamma}$. Therefore, the rewrite rules are permanently well-typed and, by Theorem 2.3, product compatibility is always guaranteed.

3 A Naive Definition of Rewriting Modulo β

As already mentioned, our goal is to give a notion of rewriting modulo β in the setting of $\lambda\Pi$-calculus Modulo. We first exhibit the issues arising from a naive definition of this notion.

In an untyped setting, we could define rewriting modulo β in this manner: t_1 rewrites to t_2 if, for some rewrite rule $(u \hookrightarrow v)$ and substitution σ, $\sigma(u) \equiv_\beta t_1$ and $\sigma(v) \equiv_\beta t_2$. This definition is not satisfactory for several reasons.

It breaks subject reduction. For the rewrite rule of Section 1, taking $\sigma = \{f \mapsto \lambda y : \Omega.y\}$ where Ω is some ill-typed term, we have

$$\texttt{D}\ (\lambda x : R.\texttt{Exp}\ x) \longrightarrow \texttt{fMult}\ (\texttt{D}\ (\lambda x : R.(\lambda y : \Omega.y)\ x)\ (\lambda x : R.\texttt{Exp}\ ((\lambda y : \Omega.y)\ x)))$$

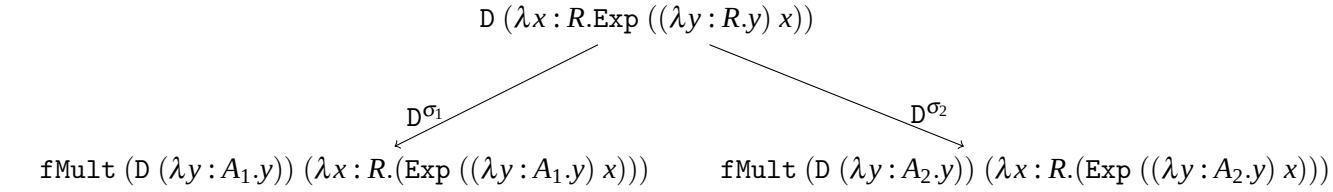
and, even if $\texttt{D}\ (\lambda x : R.\texttt{Exp}\ x)$ is well-typed, its reduct is ill-typed since it contains an ill-typed subterm.

It may introduce free variables. In the example above, Ω has no reason to be closed.

It does not provide confluence. If we consider the following variant of the rewrite rule

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ (f\ x)) \hookrightarrow \mathtt{fMult}\ (\mathtt{D}\ f)\ (\lambda x : R.\mathtt{Exp}\ (f\ x))$$

and take $\sigma_1 = \{f \mapsto \lambda y : A_1.y\}$ and $\sigma_2 = \{f \mapsto \lambda y : A_2.y\}$ where A_1 and A_2 are two non convertible types then we have:



and the peak is not joinable.

Therefore, we need to find a definition that takes care of these issues. We will achieve this using an embedding of $\lambda\Pi$-calculus Modulo into Higher-Order Rewrite Systems.

4 Higher-Order Rewrite Systems

In 1991, Nipkow [14] introduced Higher-Order Rewrite Systems (HRS) in order to lift termination and confluence results from first-order rewriting to rewriting over λ-terms. More generally, the goal was to study rewriting over terms with bound variables such as programs, theorem and proofs.

Unlike the $\lambda\Pi$-calculus Modulo, in HRSs β-reduction and rewriting do not operate at the same level. Rewriting is defined as a relation between the $\beta\eta$-equivalence classes of simply typed λ-terms: the λ-calculus is used as a meta-language.

Higher-Order Rewrite Systems are based upon the (pre)terms of the simply-typed λ-calculus built from a signature. A signature is a set of base types $\mathscr{B}$ and a set of typed constants. A simple type is either a base type $b \in \mathscr{B}$ or an arrow $A \longrightarrow B$ where A and B are simple types.

Definition 4.1 (Preterm)**.** *A* preterm *of type A is*

- *either a* variable *x of type A (we assume given for each simple type A an infinite number of variables of this type),*
- *or a* constant *f of type A,*
- *or an* application *$t(u)$ where t is a preterm of type $B \longrightarrow A$ and u is a preterm of type B,*
- *or, if $A = B \longrightarrow C$, an* abstraction *$\underline{\lambda}x.t$ where x is a variable of type B and t is a preterm of type C.*

In order to distinguish the abstraction of HRSs from the abstraction of $\lambda\Pi$-calculus Modulo, we use the underlined symbol $\underline{\lambda}$ instead of λ. Similarly, we write the application $t(u)$ for HRSs (instead of tu). We use the abbreviation $t(u_1,\ldots,u_n)$ for $t(u_1)\ldots(u_n)$. If A is a simple type, we write A^1 for A and A^{n+1} for $A \longrightarrow A^n$.

Notice also that HRSs abstractions do not have type annotations because variables are typed.

β-reduction and η-expansion are defined as usual on preterms. We write $\updownarrow^{\eta}_{\beta} t$ for the long $\beta\eta$-normal form of t.

Definition 4.2 (Term)**.** *A* term *is a preterm in long $\beta\eta$-normal form.*

Definition 4.3 (Pattern)**.** *A term t is a* pattern *if every free occurrence of a variable F is in a subterm of t of the form $F\vec{u}$ such that $\vec{u}$ is η-equivalent to a list of distinct bound variables.*

The crucial result about patterns (due to Miller [11]) is the decidability of higher-order unification (unification modulo $\beta\eta$) of patterns. Moreover, if two patterns are unifiable then a most general unifier exists and is computable.

The notion of rewrite rule for HRSs is the following:

Definition 4.4 (Rewrite Rules)**.** *A* rewrite rule *is a pair of terms $(l \hookrightarrow r)$ such that l is a pattern not η-equivalent to a variable, $FV(r) \subset FV(l)$ and l and r have the same base type.*

The restriction to patterns for the left-hand side ensures that matching is decidable but also that, when it exists, the resulting substitution is unique. This way, the situation is very close to first-order (*i.e.* syntactic) matching.

Definition 4.5 (Higher-Order Rewriting System (HRS))**.** *A* Higher-Order Rewriting System *is a set R of rewrite rules.*

The rewrite relation $\to_R$ is the smallest relation on terms closed by subterm rewriting such that, for any $(l \hookrightarrow r) \in R$ and any well-typed substitution σ, $\updownarrow_\beta^\eta \sigma(l) \to_R \updownarrow_\beta^\eta \sigma(r)$.

The standard example of an HRS is the untyped λ-calculus. The signature involves a single base type `Term` and two constants:

$$\texttt{Lam} : (\texttt{Term} \longrightarrow \texttt{Term}) \longrightarrow \texttt{Term}$$

$$\texttt{App} : \texttt{Term} \longrightarrow \texttt{Term} \longrightarrow \texttt{Term}$$

and a single rewrite rule for β-reduction:

$$(beta) \quad \texttt{App}(\texttt{Lam}(\underline{\lambda} x.X(x)), Y) \hookrightarrow X(Y)$$

5 An Encoding of the $\lambda\Pi$-calculus Modulo into Higher-Order Rewrite Systems

5.1 Encoding of Terms

We now mimic the encoding of the untyped λ-calculus as an HRS and encode the terms of the $\lambda\Pi$-calculus Modulo. First we specify the signature.

Definition 5.1. *The signature* **Sig**$(\lambda\Pi)$ *is composed of a single base type* `Term`*, the constants* `Type` *and* `Kind` *of atomic type* `Term`*, the constant* `App` *of type* $\texttt{Term} \longrightarrow \texttt{Term} \longrightarrow \texttt{Term}$*, the constants* `Lam` *and* `Pi` *of type* $\texttt{Term} \longrightarrow (\texttt{Term} \longrightarrow \texttt{Term}) \longrightarrow \texttt{Term}$ *and the constants* `c` *of type* `Term` *for every constant $c \in \mathscr{C}_O \cup \mathscr{C}_T$.*

Then we define the encoding of $\lambda\Pi$-terms.

Definition 5.2 (Encoding of $\lambda\Pi$-term)**.** *The function $\|.\|$ from $\lambda\Pi$-terms to HRS-terms in the signature* **Sig**$(\lambda\Pi)$ *is defined as follows:*

$\|\mathbf{Kind}\|$	$:=$	`Kind`	$\|\mathbf{Type}\|$	$:=$	`Type`
$\|x\|$	$:=$	x *(variable of type* `Term`*)*	$\|c\|$	$:=$	`c`
$\|uv\|$	$:=$	$\texttt{App}(\|u\|, \|v\|)$	$\|\lambda x : A.t\|$	$:=$	$\texttt{Lam}(\|A\|, \underline{\lambda} x.\|t\|)$
$\|\Pi x : A.B\|$	$:=$	$\texttt{Pi}(\|A\|, \underline{\lambda} x.\|B\|)$			

Lemma 5.1. *The function $\|.\|$ is a bijection from the $\lambda\Pi$-terms to HRS-terms of type* `Term`*.*

Note that this is a bijection between the untyped terms of the $\lambda\Pi$-calculus Modulo and well-typed terms of the corresponding HRS.

5.2 Higher-Order Rewrite Rules

We have faithfully encoded the terms. The next step is to encode the rewrite rules. The following rule corresponds to β-reduction at the HRS level:

$$(beta) \quad \mathtt{App}(\mathtt{Lam}(X, \underline{\lambda} x.Y(x)), Z) \hookrightarrow Y(Z)$$

We have the following correspondence:

Lemma 5.2.

- *If* $t_1 \to_\beta t_2$ *then* $\|t_1\| \to_{(beta)} \|t_2\|$.
- *If* $t_1 \to_{(beta)} t_2$ *and* t_1, t_2 *have type* `Term` *then* $\|t_1\|^{-1} \to_\beta \|t_2\|^{-1}$ *(where* $\|.\|^{-1}$ *is the inverse of* $\|.\|$*).*

By encoding rewrite rules in the obvious way (translating $(u \hookrightarrow v)$ by $(\|u\| \hookrightarrow \|v\|)$), we would get a similar result for Γ-reduction. But, since we want to incorporate rewriting modulo β, we proceed differently.

First, we introduce the notion of uniform terms. These are terms verifying an arity constraint on their free variables.

Definition 5.3 (Uniform Terms)**.** *A term t is* uniform *for a set of variables V if all occurrences of a variable free in t not in V is applied to the same number of arguments.*

Now, we define an encoding for uniform terms.

Definition 5.4 (Encoding of uniform terms)**.** *Let V be a set of variables and t be a term uniform in V. The HRS-term $\|u\|_V$ of type* `Term` *is defined as follows:*

$\|\mathbf{Kind}\|_V$	:=	`Kind`
$\|\mathbf{Type}\|_V$	:=	`Type`
$\|x\|_V$	:=	x if $x \in V$ (variable of type `Term`)
$\|c\|_V$	:=	`c`
$\|\lambda x : A.u\|_V$	:=	$\mathtt{Lam}(\|A\|_V, \underline{\lambda} x.\|u\|_{V \cup \{x\}})$
$\|\Pi x : A.B\|_V$	:=	$\mathtt{Pi}(\|A\|_V, \underline{\lambda} x.\|B\|_{V \cup \{x\}})$
$\|x\vec{v}\|_V$	:=	$x(\|\vec{v}\|_V)$ if $x \notin V$ (x of type $\mathtt{Term}^{n+1}$ where $n = \|\vec{v}\|$)
$\|uv\|_V$	:=	$\mathtt{App}(\|u\|_V, \|v\|_V)$ if $uv \neq x\,\vec{w}$ for $x \notin V$

Now, we define an equivalent of patterns for the $\lambda\Pi$-calculus Modulo.

Definition 5.5 ($\lambda\Pi$-patterns)**.** *Let V_0 be a set of variables, $\mathscr{A}$ be a function giving an arity to variables and let $V = (V_0, \mathscr{A})$. The subset $\mathscr{P}_V$ of $\lambda\Pi$-terms is defined inductively as follows:*

- *if c is a constant, then $c \in \mathscr{P}_V$;*
- *if $p, q \in \mathscr{P}_V$, then $p\ q \in \mathscr{P}_V$;*
- *if $x \in V_0$, then $x \in \mathscr{P}_V$;*
- *if $p \in \mathscr{P}_V$, $x \notin V_0$ and $\vec{y}$ is a vector of pairwise distinct variables in V_0 such that $|\vec{y}| = \mathscr{A}(x)$, then $p\ (x\,\vec{y}) \in \mathscr{P}_V$;*
- *if $p \in \mathscr{P}_V$, $FV(A) \subset V_0$ and $q \in \mathscr{P}_{(V_0 \cup \{x\}, \mathscr{A})}$, then $p\ (\lambda x : A.q) \in \mathscr{P}_V$;*

*A term t is a $\lambda\Pi$-*pattern *if, for some arity function $\mathscr{A}$, $t \in \mathscr{P}_{(\emptyset, \mathscr{A})}$.*

Remark that the encoding of a $\lambda\Pi$-pattern as a uniform term is a pattern.

We now define the encoding of rewrite rules.

Definition 5.6 (Encoding of Rewrite Rules)**.** *Let $(u \hookrightarrow v)$ be a rewrite rule such that*

- *u is a $\lambda\Pi$-pattern;*
- *$FV(v) \subset FV(u)$;*
- *all free occurrences of a variable in u and v are applied to the same number of arguments.*

The encoding of $(u \hookrightarrow v)$ is $\|u \hookrightarrow v\| = \|u\|_\emptyset \hookrightarrow \|v\|_\emptyset$.

Remark that the first assumption ensures that the left-hand side is a pattern and the third assumption ensures that the HRS-term is well-typed.

Definition 5.7 (HRS(Γ))**.** *Let Γ a global context whose rewrite rules satisfy the condition of Definition 5.6. We write HRS(Γ) for the HRS $\{\|u \hookrightarrow v\| \mid (u \hookrightarrow v) \in \Gamma\}$ and HRS($\beta\Gamma$) for HRS$(\Gamma) \cup \{(beta)\}$.*

6 Rewriting Modulo β

6.1 Definition

We are now able to properly define rewriting modulo β. As for usual rewriting, rewriting modulo β is defined on all (untyped) terms.

Definition 6.1 (Rewriting Modulo β)**.** *Let Γ be a global context. We say that t_1* rewrites to *t_2* modulo β *(written $t_1 \to_{\Gamma^b} t_2$) if $\|t_1\|$ rewrites to $\|t_2\|$ in HRS(Γ). Similarly, we write $t_1 \to_{\beta\Gamma^b} t_2$ if $\|t_1\|$ rewrites to $\|t_2\|$ in HRS($\beta\Gamma$).*

Lemma 6.1.

- $\to_{\beta\Gamma^b} = \to_{\Gamma^b} \cup \to_\beta$.
- *If $t_1 \to_\Gamma t_2$ then $t_1 \to_{\Gamma^b} t_2$.*

6.2 Example

Let us look at the example from the introduction. Now we have :

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x) \to_{\Gamma^b} \mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)$$

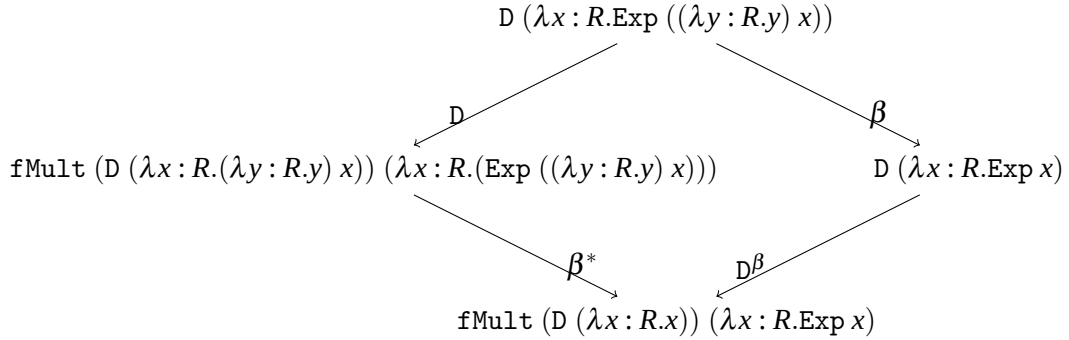
Indeed, for $\sigma = \{f \mapsto \underline{\lambda} y.y\}$ we have

$$\|\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x)\| = \mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, x))) =^{\updownarrow^\eta_\beta} \sigma(\mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, f(x)))))$$

and

$$\begin{aligned}\|\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)\| &= \mathtt{App}(\mathtt{fMult}, \mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.x)), \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, x))) \\ &=^{\updownarrow^\eta_\beta} \sigma(\mathtt{App}(\mathtt{fMult}, \mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.f(x))), \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, f(x)))))\end{aligned}$$

Therefore, the peak is now joinable.



In fact the rewriting relation can be shown confluent [15].

6.3 Properties

Rewriting modulo β also preserves typing.

Theorem 6.1 (Subject Reduction for $\rightarrow_{\Gamma^b}$)**.** *Let Γ a well-formed global context and Δ a local context well-formed for Γ. If $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \rightarrow_{\Gamma^b} t_2$ then $\Gamma;\Delta \vdash t_2 : T$.*

It directly follows from the following lemma:

Lemma 6.2. *If $t_1 \rightarrow_{\Gamma^b} t_2$ then, for some t_1' and t_2', we have $t_1 \leftarrow_\beta^* t_1' \rightarrow_\Gamma t_2' \rightarrow_\beta^* t_2$. Moreover, if t_1 is well-typed then we can choose t_1' such that it is well-typed in the same context.*

Proof. The idea is to lift the β-reductions that occur at the HRS level to the $\lambda\Pi$-calculus Modulo. Suppose $t_1 \rightarrow_{\Gamma^b} t_2$. For some rewrite rule $(u \hookrightarrow v)$ and (HRS) substitution σ, we have $\updownarrow_\beta^\eta \sigma(u) = \|t_1\|$ and $\updownarrow_\beta^\eta \sigma(v) = \|t_2\|$. We define the ($\lambda\Pi$) substitution $\hat{\sigma}$ as follows: $\hat{\sigma}(x) = \|\sigma(x)\|^{-1}$ if $\sigma(x)$ has type $\mathtt{Term}$; $\hat{\sigma}(x) = \lambda\vec{x} : \vec{A}.\|u\|^{-1}$ if $\sigma(x) = \underline{\lambda}\vec{x}.u$ has type $\mathtt{Term}^n \longrightarrow \mathtt{Term}$ where the A_i are arbitrary types. We have, at the $\lambda\Pi$ level, $\hat{\sigma}(u) \rightarrow_\Gamma \hat{\sigma}(v)$, $\hat{\sigma}(u) \rightarrow_\beta^* t_1$ and $\hat{\sigma}(v) \rightarrow_\beta^* t_2$. If t_1 is well-typed then the A_i can be chosen so that $\hat{\sigma}(u)$ is also well-typed. □

Another consequence of this lemma is that the rewriting modulo β does not modify the congruence.

Theorem 6.2. *The congruence generated by $\rightarrow_{\beta\Gamma^b}$ is equal to $\equiv_{\beta\Gamma}$.*

Proof. Follows from Lemma 6.1 and Lemma 6.2. □

6.4 Generalized Criteria for Product Compatibility and Well-Typedness of Rewrite Rules

Using our new notion of rewriting modulo β, we can generalize the criteria of Section 2.5.

Theorem 6.3. *Let Γ be a global context. If HRS($\beta\Gamma$) is confluent, then product compatibility holds for Γ.*

Proof. Assume that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ then, by Theorem 6.2, $\Pi x : A_1.B_1 \equiv_{\beta\Gamma^b} \Pi x : A_2.B_2$. By confluence, there exist A_0 and B_0 such that $A_1 \rightarrow_{\beta\Gamma^b}^* A_0$, $A_2 \rightarrow_{\beta\Gamma^b}^* A_0$, $B_1 \rightarrow_{\beta\Gamma^b}^* B_0$ and $B_2 \rightarrow_{\beta\Gamma^b}^* B_0$. It follows, by Theorem 6.2, that $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$. □

To prove the confluence of a HRS, one can use van Oostrom's development-closed theorem [15].
Theorem 2.5 can also be generalized to deal with $\lambda\Pi$-patterns.

Theorem 6.4. *Let Γ be a well-formed global context and $(u \hookrightarrow v)$ be a rewrite rule. If u is a $\lambda\Pi$-pattern and there exist Δ and T such that $\Gamma \vdash^{ctx} \Delta$, $FV(u) = dom(\Delta)$, $\Gamma;\Delta \vdash u : T$ and $\Gamma;\Delta \vdash v : T$ then $(u \hookrightarrow v)$ is permanently well-typed for Γ.*

This theorem is a corollary of the following lemma.

Lemma 6.3. *Let $\Gamma \subset \Gamma_2$ be two well-formed global contexts. If $t \in \mathscr{P}_{dom(\Sigma)}$, $dom(\sigma) = dom(\Delta)$, for all $(x : A) \in \Sigma$, $\sigma(A) = A$, $\Gamma;\Delta\Sigma \vdash t : T$ and $\Gamma_2;\Delta_2\Sigma \vdash \sigma(t) : T_2$ then $T_2 \equiv_{\beta\Gamma_2} \sigma(T)$ and, for all $x \in FV(t) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ for $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.*

Proof. We proceed by induction on $t \in \mathscr{P}_{dom(\Sigma)}$.

- if $t = c$ is a constant, then $FV(t) = \emptyset$ and, by inversion on $\Gamma;\Delta\Sigma \vdash t : T$, there exists a (closed term) A such that $(c : A) \in \Gamma \subset \Gamma_2$, $T \equiv_{\beta\Gamma} A$ and $T_2 \equiv_{\beta\Gamma_2} A$. Since $A = \sigma(A)$, we have $\sigma(T) \equiv_{\beta\Gamma_2} T_2$.

- if $t = x \in dom(\Sigma)$, then, by inversion, there exists A such that $(x : A) \in \Sigma$, $T \equiv_{\beta\Gamma} A$ and $T_2 \equiv_{\beta\Gamma_2} A$. Since $A = \sigma(A)$, we have $\sigma(T) \equiv_{\beta\Gamma_2} T_2$.
- if $t = p\ q$, then, by inversion, on the one hand, $\Gamma;\Delta\Sigma \vdash p : \Pi x : A.B$, $\Gamma;\Delta\Sigma \vdash q : A$ and $T \equiv_{\beta\Gamma} B[x/q]$. On the other hand, $\Gamma_2;\Delta_2\Sigma \vdash \sigma(p) : \Pi x : A_2.B_2$, $\Gamma_2;\Delta_2\Sigma \vdash \sigma(q) : A_2$ and $T_2 \equiv_{\beta\Gamma_2} B_2[x/\sigma(q)]$.

  By induction hypothesis on p, we have $\sigma(\Pi x : A.B) \equiv_{\beta\Gamma_2} \Pi x : A_2.B_2$ and for all $x \in FV(p) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.

  By product-compatibility of Γ_2, $\sigma(A) \equiv_{\beta\Gamma_2} A_2$ and $\sigma(B) \equiv_{\beta\Gamma_2} B_2$. It follows that $\sigma(T) \equiv_{\beta\Gamma_2} \sigma(B[x/q]) \equiv_{\beta\Gamma_2} B_2[x/\sigma(q)] \equiv_{\beta\Gamma_2} T_2$.

  Now, we distinguish three sub-cases:
  - either $q \in \mathscr{P}_{dom(\Sigma)}$ and by induction hypothesis on q, for all $x \in FV(q) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.
  - Or $q = \lambda x : A.q_0$ with $FV(A) \in dom(\Sigma)$ and $q_0 \in \mathscr{P}_{dom(\Sigma(x:A))}$ and by induction hypothesis on q_0, for all $x \in FV(q_0) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.
  - Or $q = x\vec{y}$ with $x \notin dom(\Sigma)$ and $\vec{y} \subset dom(\Sigma)$. By inversion, on the one hand, $\Delta(x) \equiv_{\beta\Gamma} \Pi\vec{y} : \Sigma(\vec{y}).C$ for $C \equiv_{\beta\Gamma} A$. On the other hand, $\Gamma_2;\Delta_2 \vdash \sigma(x) : \Pi\vec{y} : \Sigma(\vec{y}).C_2$ for $C_2 \equiv_{\beta\Gamma_2} A_2$. Since $\sigma(A) \equiv_{\beta\Gamma_2} A_2$, we have $\Pi\vec{y} : \Sigma(\vec{y}).C_2 \equiv_{\beta\Gamma_2} \Pi\vec{y} : \Sigma(\vec{y}).\sigma(C) = \sigma(\Delta(x))$.

□

Proof of Theorem 6.4. Let Γ_2 be a well-formed extension of Γ. Suppose that $\Gamma_2;\Delta_2 \vdash \sigma(u) : T_2$.

By Lemma 6.3 and $FV(u) = dom(\Delta)$, we have, for all $x \in dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ for $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$ and $T_2 \equiv_{\beta\Gamma_2} \sigma(T)$.

By induction on $\Gamma;\Delta \vdash v : T$, we deduce $\Gamma_2;\Delta_2 \vdash \sigma(v) : T_3$, for $T_3 \equiv_{\beta\Gamma_2} \sigma(T) \equiv_{\beta\Gamma_2} T_2$. It follows, by conversion, that $\Gamma_2;\Delta_2 \vdash \sigma(v) : T_2$. □

7 Applications

7.1 Parsing and Solving Equations

The context declarations and rewrite rules of Figure 6 define a function `to_expr` which parses a function of type `Nat` to `Nat` into an expression of the form $a * x + b$ (represented by the term `mk_expr` a b) where a and b are constants. The left-hand sides of the rewrite rules on `to_expr` are $\lambda\Pi$-patterns. This allows defining `to_expr` by pattern matching in a way which looks under the binders.

The function `solve` can then be used to solve the linear equation $a * x + b = 0$. The answer is either `None` if there is no solution, or `All` if any x is a solution or `One` m n if $-m/(n+1)$ is the only solution.

For instance, we have (writing `One` $-\frac{1}{3}$ for `One` 1 2):

$$\texttt{solve}\ (\texttt{to\_expr}(\lambda x : \texttt{Nat.plus}\ x\ (\texttt{plus}\ x\ (\texttt{S}\ x)))) \to^*_{\beta\Gamma} \texttt{One}\ -\frac{1}{3}.$$

By Theorem 6.3 and Theorem 6.4 the global context of Figure 6 is well-formed.

7.2 Universe Reflection

In [1], Assaf defines a version of the calculus of construction with explicit universe subtyping thanks to an extended notion of conversion generated by a set of rewrite rules. This work can easily be adapted to fit in the framework of the $\lambda\Pi$-calculus Modulo. However, the confluence of the rewrite system holds only for rewriting modulo β.

$$
\begin{array}{lcl}
\texttt{expr} & : & \textbf{Type}. \\
\texttt{mk\_expr} & : & \texttt{Nat} \longrightarrow \texttt{Nat} \longrightarrow \texttt{expr}. \\
\texttt{expr\_S} & : & \texttt{expr} \longrightarrow \texttt{expr}. \\
\texttt{expr\_S}\ (\texttt{mk\_expr}\ a\ b) & \hookrightarrow & \texttt{mk\_expr}\ a\ (\texttt{S}\ b). \\
\texttt{expr\_P} & : & \texttt{expr} \longrightarrow \texttt{expr} \longrightarrow \texttt{expr}. \\
\texttt{expr\_P}\ (\texttt{mk\_expr}\ a_1\ b_1)\ (\texttt{mk\_expr}\ a_2\ b_2) & \hookrightarrow & \texttt{mk\_expr}\ (\texttt{plus}\ a_1\ a_2)\ (\texttt{plus}\ b_1\ b_2). \\
\texttt{to\_expr} & : & (\texttt{Nat} \longrightarrow \texttt{Nat}) \longrightarrow \texttt{expr}. \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.0) & \hookrightarrow & \texttt{mk\_expr}\ 0\ 0. \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.\texttt{S}\ (f\ x)) & \hookrightarrow & \texttt{expr\_S}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x)). \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.x) & \hookrightarrow & \texttt{mk\_expr}\ (\texttt{S}\ 0)\ 0. \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.\texttt{plus}\ (f\ x)\ (g\ x)) & \hookrightarrow & \\
\multicolumn{3}{r}{\texttt{expr\_P}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x))\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.g\ x)).} \\
\texttt{Solution} & : & \textbf{Type}. \\
\texttt{All} & : & \texttt{Solution}. \\
\texttt{One} & : & \texttt{Nat} \longrightarrow \texttt{Nat} \longrightarrow \texttt{Solution}. \\
\texttt{None} & : & \texttt{Solution}. \\
\texttt{solve}\ (\texttt{mk\_expr}\ 0\ 0) & \hookrightarrow & \texttt{All}. \\
\texttt{solve}\ (\texttt{mk\_expr}\ 0\ (S\ n)) & \hookrightarrow & \texttt{None}. \\
\texttt{solve}\ (\texttt{mk\_expr}\ (S\ n)\ m) & \hookrightarrow & \texttt{One}\ m\ n.
\end{array}
$$

Figure 6: Parsing and solving linear equations

8 Conclusion

We have defined a notion of rewriting modulo β for the $\lambda\Pi$-calculus Modulo. We achieved this by encoding the $\lambda\Pi$-calculus Modulo into the framework of Higher-Order Rewrite Systems. As a consequence we also made available for the $\lambda\Pi$-calculus Modulo the confluence criteria designed for the HRSs (see for instance [14] or [15]). We proved that rewriting modulo β preserves typing. We generalized the criterion for product compatibility, by replacing the assumption of confluence by the confluence of the rewriting relation modulo β. We also generalized the criterion for well-typedness of rewrite rules to allow left-hand to be $\lambda\Pi$-patterns. These generalizations permit proving subject reduction and uniqueness of types for more systems.

A natural extension of this work would be to consider rewriting modulo $\beta\eta$ as in Higher-Order Rewrite Systems. This requires extending the conversion with η-reduction. But, as remarked in [10] (attributed to Nederpelt), $\rightarrow_{\beta\eta}$ is not confluent on untyped terms as the following example shows:

$$\lambda y : B.y \leftarrow_{\eta} \lambda x : A.(\lambda y : B.y)x \rightarrow_{\beta} \lambda x : A.x$$

Therefore properties such as product compatibility need to be proved another way. We leave this line of research for future work.

For the $\lambda\Pi$-calculus a notion of higher-order pattern matching has been proposed [16] based on Contextual Type Theory (CTT) [13]. This notion is similar to our. However, it is defined using the notion of meta-variable (which is native in CTT) instead of a translation into HRSs.

In [3], Blanqui studies the termination of the combination of β-reduction with a set of rewrite rules with matching modulo $\beta\eta$ in the polymorphic λ-calculus. His definition of rewriting modulo $\beta\eta$ is

direct and does not use any encoding. This leads to a slightly different notion a rewriting modulo β. For instance, $\mathtt{D}(\lambda : R.\mathtt{Exp}\ x)$ would reduce to $\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.(\lambda y : R.y)\ x))\ (\lambda x : R.\mathtt{Exp}\ ((\lambda y : R.y)\ x))$ instead of $\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)$. It would be interesting to know whether the two definitions are equivalent with respect to confluence.

We implemented rewriting modulo β in Dedukti [5], our type-checker for the $\lambda\Pi$-calculus Modulo.

Acknowledgments. The author thanks very much Ali Assaf, Olivier Hermant, Pierre Jouvelot and the reviewers for their very careful reading and many suggestions.

References

[1] A. Assaf (2015): *A calculus of constructions with explicit subtyping*. In: *The 20th International Conference on Types for Proofs and Programs (TYPES '14)*.

[2] A. Assaf & G. Burel (2014): *Translating HOL to Dedukti*. Available at `https://hal.archives-ouvertes.fr/hal-01097412`.

[3] F. Blanqui (2015): *Termination of rewrite relations on lambda-terms based on Girard's notion of reducibility. Theoretical Computer Science*. To appear.

[4] M. Boespflug & G. Burel (2012): *CoqInE : Translating the calculus of inductive constructions into the $\lambda\Pi$-calculus modulo*. In: *The Second International Workshop on Proof Exchange for Theorem Proving (PxTP)*.

[5] M. Boespflug, Q. Carbonneaux, O. Hermant & R. Saillard: *Dedukti*. Available at `http://dedukti.gforge.inria.fr`.

[6] G. Burel (2013): *A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo*. In: *The Third International Workshop on Proof Exchange for Theorem Proving (PxTP '13)*.

[7] R. Cauderlier & C. Dubois (2015): *Objects and Subtyping in the $\lambda\Pi$-Calculus Modulo*.

[8] D. Cousineau & G. Dowek (2007): *Embedding Pure Type Systems in $\lambda\Pi$-Calculus Modulo*. In: *The 8th International Conference on Typed Lambda Calculi and Applications (TLCA '07)*, doi:10.1007/978-3-540-73228-0_9.

[9] A. Dorra: *Equivalence de Curry-Howard entre le lambda-Pi calcul et la logique intuitionniste*. Report.

[10] H. Geuvers (1992): *The Church-Rosser Property for beta-eta-reduction in Typed lambda-Calculi*. In: *The Seventh Annual Symposium on Logic in Computer Science (LICS '92)*, doi:10.1109/LICS.1992.185556.

[11] D. Miller (1991): *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*. *Journal of Logic and Computation*, doi:10.1093/logcom/1.4.497.

[12] F. Müller (1992): *Confluence of the Lambda Calculus with Left-Linear Algebraic Rewriting*. *Information Processing Letters*, doi:10.1016/0020-0190(92)90155-O.

[13] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory*. *ACM Trans. Comput. Log.* 9(3), doi:10.1145/1352582.1352591.

[14] T. Nipkow (1991): *Higher-Order Critical Pairs*. In: *The Sixth Annual Symposium on Logic in Computer Science (LICS '91)*, doi:10.1109/LICS.1991.151658.

[15] V. van Oostrom (1995): *Development Closed Critical Pairs*. In: *The Second International Workshop on Higher-Order Algebra, Logic, and Term Rewriting, (HOA '95)*, doi:10.1007/3-540-61254-8_26.

[16] Brigitte Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In: *Symposium on Principles of Programming Languages, (POPL '08)*, doi:10.1145/1328438.1328483.

[17] R. Saillard (2013): *Towards explicit rewrite rules in the $\lambda\Pi$-calculus modulo*. In: *The 10th International Workshop on the Implementation of Logics (IWIL '13)*.

Sequent Calculus and Equational Programming

(work in progress)

Nicolas Guenot and Daniel Gustafsson
IT University of Copenhagen
{ngue,dagu}@itu.dk

Proof assistants and programming languages based on type theories usually come in two flavours: one is based on the standard natural deduction presentation of type theory and involves eliminators, while the other provides a syntax in *equational* style. We show here that the equational approach corresponds to the use of a focused presentation of a type theory expressed as a sequent calculus. A typed functional language is presented, based on a sequent calculus, that we relate to the syntax and internal language of Agda. In particular, we discuss the use of patterns and case splittings, as well as rules implementing inductive reasoning and dependent products and sums.

1 Programming with Equations

Functional programming has proved extremely useful in making the task of writing correct software more abstract and thus less tied to the specific, and complex, architecture of modern computers. This, is in a large part, due to its extensive use of types as an abstraction mechanism, specifying in a crisp way the intended behaviour of a program, but it also relies on its *declarative* style, as a mathematical approach to functions and data structures. However, the vast gain in expressivity obtained through the development of *dependent types* makes the programming task more challenging, as it amounts to the question of proving complex theorems — as illustrated by the double nature of proof assistants such as Coq [11] and Agda [18]. Keeping this task as simple as possible is then of the highest importance, and it requires the use of a clear declarative style.

There are two main avenues for specifying a language of proofs, or programs, that is abstract enough to support complex developments involving dependent types. The first approach, chosen by the Coq project, is to have a language of *tactics* that partially automate the construction of proofs — that is, to mechanically construct complex programs based on the composition of a few generic commands. While this takes the development task closer to the usual idea of proving a mathematical theorem, the second approach is to take the programming viewpoint: although Coq allows to directly write proof terms, this is better illustrated by Agda, where a syntax inspired by Haskell [1] provides a clear *equational* style.

Our goal here is to investigate the relations between the equational style of dependently-typed functional programming as found in Agda to the proof-theoretical description of intuitionistic logic given in the sequent calculus. In particular, we claim that a *focused* sequent calculus, akin to the **LJF** system of Liang and Miller [15], offers a logical foundation of choice for the development of a practical dependently-typed language. We intend to support this claim by showing how the equational syntax of Agda and the internal structure of its implementation correspond to a computational interpretation of such a calculus — for an extended for of intuitionistic logic including dependencies and (co)induction. As it turns out, the use of left rules rather than eliminations for *positive* connectives such as disjunction, in sequent calculus, yields a simpler syntax. In general, beyond the use of *spines* in applications, as in **LJT** [13] and quite common in the implementation of functional programming languages or proof

I. Cervesato and K. Chaudhuri (Eds.): Tenth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
EPTCS 185, 2015, pp. 102–109, doi:10.4204/EPTCS.185.7

assistants, the structure of the sequent calculus is much closer to the equational style of programming than natural deduction, the standard formalism in which type theory is usually expressed [16]. Using a focused system rather than a plain sequent calculus based on **LJ** provides a stronger structure, and emphasizes the importance of *polarities*, already observed in type theory [2].

Beyond the definition of a logical foundation for a functional language in equational style, giving a proof-theoretical explanation for the way Agda is implemented requires to accomodate in the sequent calculus both dependent types and a notion of inductive definition. This is not an easy task, although there has been some work on dependent types in the sequent calculus [14] and there is a number of approaches to inductive definitions in proof theory, including focused systems [5]. For example, the system found in [14] is based on **LJT** but is limited to Π and does not support Σ, while [12] has both, but requires an intricate mixture of natural deduction and sequent calculus to handle Σ. Induction is even more complex to handle, since there are several approaches, including definitions [19] or direct least and greatest fixpoints as found in μ**MALL** [5] and μ**LJ** [4]. From the viewpoint of proof-theory, the least fixpoint operator μ seems to be well-suited, as it embodies the essence of induction, while the greatest fixpoint ν allows to represent coinduction. However, these operators are not used the same way as inductive definitions found in Agda or other languages or proof assistants — they seem more primitive, but the encoding of usual constructs in terms of fixpoints is not obvious. Even more complicated is the question of using fixpoints in the presence of dependent types, and this has only been studied from the type-theoretic viewpoint in complex systems such as the *Calculus of Inductive Constructions* [10]. In the end, what we would like to obtain is a proof-theoretical understanding of the equational style of dependent and (co)inductive programming, related to the goals of the Epigram project. In particular, we consider that the sequent calculus, with its use of left rules, provides access to the *"left"* of equations in a sense similar to what is described in [17].

Here, we will describe the foundamental ideas for using a variant of **LJF** as the basis for the design of a dependently-typed programming language. We start in Section 2 by considering a propositional system and show how the shape of sequent calculus rules allows to type terms in equational style. This is made even more obvious by the use of pattern in the binding structure of the calculus. Then, in Section 3 we discuss the extension of this system to support dependent types and induction, problems related to patterns in this setting, as well as the question of which proof-theoretical approach to induction and coinduction is better suited for use in a such a language. Finally, we conclude by the review of some research problems opened by this investigation, and an evaluation of the possible practical applications to languages and proofs assistants.

2 Focusing and Polarities in the Sequent Calculus

We start our investigation with a propositional intuitionistic system presented as a focused sequent calculus. It is a variant of **LJF** [15] to which we assign a term language extending the $\overline{\lambda}$-calculus of Herbelin [13]. Unlike the calculus based on **LJT**, this system has positive disjunctions and conjunctions $\vee$ and $\times$, but it has no positive atoms. We use the following grammar of formulas:

$$N,M ::= a \mid \uparrow P \mid P \to N \mid N \wedge M \qquad\qquad P,Q ::= \downarrow N \mid P \vee Q \mid P \times Q$$

where $\uparrow$ and $\downarrow$ are called *polarity shifts* and are meant to maintain an explicit distinction between the two categories of formulas, negatives and positives. This is not absolutely necessary, but it clarifies the definition of a focused system by linking the *focus* and *blur* rules to actual connectives. Note that this was also used in the presentation of a computational interpretation of the full **LJF** system [7].

$$\frac{}{\Psi,[N] \vDash \varepsilon : N} \qquad \frac{\Psi \vDash d : [P]}{\Psi \mid \cdot \vdash \triangleright\, d : \uparrow P} \qquad \frac{\Psi \mid \cdot \vdash t : N}{\Psi \vDash \triangleleft\, t : [\downarrow N]}$$

$$\frac{\Psi, x : \downarrow N, [N] \vDash k : M}{\Psi, x : \downarrow N \mid \cdot \vdash x\, k : M} \qquad \frac{\Psi, x : \downarrow N \mid \Gamma \vdash t : M}{\Psi \mid \Gamma, x : \downarrow N \vdash t : M} \qquad \frac{\Psi \mid p : P \vdash t : N}{\Psi, [\uparrow P] \vDash \kappa p.t : N}$$

$$\frac{\Psi \mid \Gamma, p : P \vdash t : N}{\Psi \mid \Gamma \vdash \lambda p.t : P \to N} \qquad \frac{\Psi, [N] \vDash k : L}{\Psi, [N \wedge M] \vDash \mathtt{prl}\ k : L} \qquad \frac{\Psi, [M] \vDash k : L}{\Psi, [N \wedge M] \vDash \mathtt{prr}\ k : L}$$

$$\frac{\Psi \vDash d : [P] \qquad \Psi, [N] \vDash k : M}{\Psi, [P \to N] \vDash d :: k : M} \qquad \frac{\Psi \mid \Gamma \vdash t : N \qquad \Psi \mid \Gamma \vdash u : M}{\Psi \mid \Gamma \vdash \langle t, u \rangle : N \wedge M}$$

$$\frac{\Gamma \vDash d : [P] \qquad \Gamma \vDash e : [Q]}{\Gamma \vDash (d, e) : [P \times Q]} \qquad \frac{\Gamma \vDash d : [P]}{\Gamma \vDash \mathtt{inl}\ d : [P \vee Q]} \qquad \frac{\Gamma \vDash d : [Q]}{\Gamma \vDash \mathtt{inr}\ d : [P \vee Q]}$$

$$\frac{\Psi \mid \Gamma, p : P, q : Q \vdash t : N}{\Psi \mid \Gamma, (p, q) : P \times Q \vdash t : N} \qquad \frac{\Psi \mid \Gamma, p : P \vdash t : N \qquad \Psi \mid \Gamma, q : Q \vdash u : N}{\Psi \mid \Gamma, x[p \mid q] : P \vee Q \vdash x[t \mid u] : N}$$

Figure 1: Typing rules for a pattern-based λ-calculus based on $\overline{\lambda}$

The rules we use in this system are shown in Figure 1, where the term assignment is indicated in red and several turnstiles are used to distinguish an inversion phase $\vdash$ from a focused phase $\vDash$. In this syntax, brackets are used to pinpoint the precise formula under focus. The extended λ-calculus we use to represent proofs is based on the following grammar:

$$\begin{array}{rcl} t, u & ::= & \triangleright\, d \mid \lambda p.t \mid x\, k \mid \langle t, u \rangle \mid x[t \mid u] \\ p, q & ::= & x \mid (p, q) \mid x[p \mid q] \\ d, e & ::= & \triangleleft\, t \mid (d, e) \mid \mathtt{inl}\ d \mid \mathtt{inr}\ d \\ k, m & ::= & \varepsilon \mid t :: k \mid \mathtt{prl}\ k \mid \mathtt{prr}\ k \mid \kappa p.t \end{array}$$

where t denotes a *term*, p a binding *pattern*, d a *data* structure and k an application *context*. In terms of programming, terms are describing computation, mostly by means of functions, while data structures implement pairs and constructors. Note that computations can use *case splittings* $x[t \mid u]$ to choose between the subterms t or u depending on the contents of the data bound to x. The use of patterns rather than plain variables to annotate formulas in the context of typing judgement is taken from [8] and allows to express more directly the equational style found in Agda. For example, we could write:

$$\begin{array}{l} \mathtt{f} : (\mathbb{N} \times \mathbb{N}) \uplus \mathbb{N} \to \mathbb{N} \\ \mathtt{f}\ (\mathtt{inl}\ (\mathtt{x}, \mathtt{y})) = \mathtt{x} + \mathtt{y} \\ \mathtt{f}\ (\mathtt{inr}\ \mathtt{z}) = \mathtt{z} \end{array}$$

to define a function f that uses pattern-matching on its argument and computes the result based on the components of the data structure it received. Such a function can be written in our calculus as the following

term: $\lambda w[(x,y) \mid z].w[\text{add}\ ((x\ \varepsilon) :: (y\ \varepsilon) :: \varepsilon) \mid z\ \varepsilon]$, where add is the name of the addition function. This makes the compilation of the code written above to the adequate representation in our calculus relatively easy, since different parts of a definition can be aggregated into a term with a pattern and a case splitting. This is very much related to the question of compiling pattern-matching into a specific *splitting tree* where case constructs are used [3].

The idea of the logical approach is that *cut elimination* in this system yields a reduction system implementing the dynamics of computation in the corresponding calculus. In such a focused calculus, a number of cut rules are needed to complete the proof of completeness of the cut-free fragment, but only two of them really need to be considered as rules — the other cuts can simply be stated as principles, and their reduction will correspond to a big step of computation. These two rules are:

$$\frac{\Psi \vDash d : [P] \qquad \Psi \mid \Gamma, p : P \vdash t : N}{\Psi \mid \Gamma \vdash p = d\ \mathtt{in}\ t : N} \qquad \frac{\Psi \mid \Gamma \vdash t : N \qquad \Psi, [N] \vDash k : M}{\Psi \mid \Gamma \vdash t\ k : M}$$

the first one being the binding of a data structure to a matching pattern, and the second a simple application of a term to a list of arguments. The latter is already part of the **LJT** system [13], but the former is specific to **LJF** in the sense that it appears only when formulas can be focused on the right of a sequent. The main reduction rule extracted from cut elimination is the $\overline{\lambda}$ variant of β-reduction:

$$(\lambda p.t)\ (d :: k) \quad \to \quad (p = d\ \mathtt{in}\ t)\ k$$

but there are a number of other reduction rules generated by the use of other connectives than implication. In particular, conjunction yields a form of pairing where a term $\langle t, u\rangle$ has to be applied to a list $\mathtt{prl}\ k$ to reduction to $t\ k$. The binding cut is simpler in a certain sense, since its reduction corresponds to a decomposition of the data structure d according to the shape of the pattern p, and a simple substitution when p is just a variable. Moreover, other cuts encountered during reduction usually amount to a form of substitution, except for the one, already present in **LJT**, that yields lists concatenation in the argument of an application.

Note that the $\triangleright\, d$ construct is present in the internal language of Agda, but the constructs $\triangleleft\, t$ and $\kappa p.t$ are not, although they can be obtained indirectly using a cut. While $\triangleleft\, t$ should simply be understood as a *thunk*, which is a term made into data, the list $\kappa p.t$ is slightly more complex. This construct, already present in [6], is more a *context* than a list in the sense that it stops the application of a term to $\kappa p.t$ and enforces the execution of t, where the original term applied is bound to p. This can be understood by considering the reduction extracted from cut elimination:

$$(\triangleright\, d)\ (\kappa p.t) \quad \to \quad p = d\ \mathtt{in}\ t$$

Finally, note that we could have an explicit contraction rule in the system, that would appear in terms under the form of a pattern $p @ q$ indicating that p and q will be the patterns associated to two copies of the same assumption P. The associated typing rule is:

$$\frac{\Psi \mid \Gamma, p : P, q : P \vdash t : N}{\Psi \mid \Gamma, p @ q : P \vdash t : N}$$

and it is reminiscent of the pattern using the same syntax in Haskell — which is meant to exist in Agda as well, but this not yet implemented. However, in Haskell, this is restricted to the form $x @ p$ so that it can only serve to name an assumption before decomposing it, and we could allow for such a use by avoiding maximal inversion, which is not strictly necessary in a focused system [7]. This rule is not necessary for the completeness of the calculus, and there are other ways to obtain the same result. Of course, in a very similar way, the pattern $\_$ can be associated to the weakening rule, also admissible.

$$\frac{}{\Psi,[N] \vDash \varepsilon : N} \qquad \frac{\Psi \vDash d : [P]}{\Psi \mid \cdot \vdash \triangleright\, d : \uparrow P} \qquad \frac{\Psi \mid \cdot \vdash t : N}{\Psi \vDash \triangleleft\, t : [\downarrow N]} \qquad \frac{\Psi,[N] \vDash k : L}{\Psi,[N \wedge M] \vDash \mathtt{prl}\ k : L}$$

$$\frac{\Psi,x : \downarrow N,[N] \vDash k : M}{\Psi,x : \downarrow N \mid \cdot \vdash x\, k : M} \qquad \frac{\Psi,x : \downarrow N \mid \Gamma \vdash t : M}{\Psi \mid \Gamma,x : \downarrow N \vdash t : M} \qquad \frac{\Psi \mid x : P \vdash t : N}{\Psi,[\uparrow P] \vDash \kappa x.t : N} \qquad \frac{\Psi,[M] \vDash k : L}{\Psi,[N \wedge M] \vDash \mathtt{prr}\ k : L}$$

$$\frac{\Psi \mid \Gamma,x : P \vdash t : N}{\Psi \mid \Gamma \vdash \lambda x.t : \Pi(x : P).N} \qquad \frac{\Psi \vDash d : [P] \qquad \Psi,[N\{d/x\}] \vDash k : M}{\Psi,[\Pi(x : P).N] \vDash d :: k : M} \qquad \frac{\Psi \mid \Gamma \vdash t : N \qquad \Psi \mid \Gamma \vdash u : M}{\Psi \mid \Gamma \vdash \langle t,u \rangle : N \wedge M}$$

$$\frac{\Psi \mid \Gamma,y : P,z : Q \vdash t : N\{(y,z)/x\}}{\Psi \mid \Gamma,x : \Sigma(y : P).Q \vdash y,z = x\ \mathtt{in}\ t : N} \qquad \frac{\Gamma \vDash d : [P] \qquad \Gamma \vDash e : [Q\{d/x\}]}{\Gamma \vDash (d,e) : [\Sigma(x : P).Q]} \qquad \frac{\Gamma \vDash d : [P]}{\Gamma \vDash \mathtt{inl}\ d : [P \vee Q]}$$

$$\frac{\Psi \mid \Gamma,y : P \vdash t : N\{\mathtt{inl}\ y/x\} \qquad \Psi \mid \Gamma,z : Q \vdash u : N\{\mathtt{inr}\ z/x\}}{\Psi \mid \Gamma,x : P \vee Q \vdash x[y.t \mid z.u] : N} \qquad \frac{\Gamma \vDash d : [Q]}{\Gamma \vDash \mathtt{inr}\ d : [P \vee Q]}$$

$$\frac{\Psi \vDash d : [A] \qquad \Psi \mid \Gamma,x : A,\Delta \vdash t : B}{\Psi \mid \Gamma,\Delta\{d/x\} \vdash x = d\ \mathtt{in}\ t : B\{d/x\}} \qquad \frac{\Psi \mid \Gamma \vdash t : A \qquad \Psi,[A] \vDash k : B}{\Psi \mid \Gamma \vdash t\, k : B}$$

Figure 2: Typing rules for a dependent λ-calculus based on $\overline{\lambda}$

3 Adding Dependent Types and Induction

We continue our investigation by adapting our variant of **LJF** to dependent types, but this unveils some issues that we will now discuss. On problem we immediately encounter is the adaptation of the pattern machinery to the dependent setting, mostly due to the substitutions involved in the types, where patterns should have appeared. For the dependent implication $\Pi(x : P).N$, using a pattern p rather than a binding variable x yields the question of substituting a data structure d for p: this becomes a much more complicated operation than the traditional substitution. Moreover, keeping the patterns and variables synchronised between their use in terms and in types is a challenging task, that would probably require heavy syntactic mechanisms. For this reason, the system shown above in Figure 2 has no patterns, but rather falls back to the traditional style of typing using only variables to label assumptions. The language used in this variant can still be related to the equational approach to functional programming, but the translation between equations and terms is more involved.

The generalisation of the implication into the dependent product $\Pi(x : P).N$ is a straightforward operation, and the rules we use are essentially the ones found in [14] — except that it involves a data structure, corresponding to a focus on the right-hand side of a sequent. Now, the case of Σ is more complicated, as it is *a priori* unclear whether it should be obtained as a generalisation of the negative conjunction $\wedge$ or of the positive product $\times$ and both solutions might even be possible. But a generalisation of the negative disjunction seems to be problematic, when it comes to the specification of the second left rule, typing the `prr` operation. Indeed, when focusing on $\Sigma(x : N).M$ we would need to plug a term of type N for x in M, but this would require to maintain some *"natural deduction version"* of the term currently

being transformed, and to plug at adequate locations some translation between natural deduction style and our sequent calculus syntax — as done in [12]. This is quite unsatisfactory and will not help us build a proper understanding of dependent types in a pure sequent calculus setting. The solution we adopt here is to obtain $\Sigma(x:P).Q$ as a generalisation of the positive product $\times$ and simply update the corresponding rules as shown in Figure 2. The left rule is simple to define in this case, because the decomposition of the Σ in the context preserves the binding of y in the type Q.

There is a particularly interesting benefit to the use of the sequent calculus to handle splitting as done in the left Σ rule. Consider the elimination rule in natural deduction:

$$\vee\boldsymbol{e}\ \frac{\Gamma, x:A\vee B\vdash C:\mathtt{type}\quad \Gamma\vdash t:A\vee B\quad \Gamma, y:A\vdash u:C\{\mathtt{inl}\, y/x\}\quad \Gamma, z:B\vdash v:C\{\mathtt{inr}\, z/x\}}{\Gamma\vdash \mathtt{match}\ [x.C]\ (t\,;y.u\,;z.v):C\{t/x\}}$$

and observe that it is necessary to be explicit about the return type, since obtaining C from $C\{t/x\}$ is a complicated process, that *reverses* a substitution. This makes the term syntax heavy, while the problem is avoided in the sequent calculus, where no substitution is needed in the conclusion. Note that in Coq, the natural deduction style is used for the proof language, but tactics are written in a style that is much closer to the sequent calculus — as this is the framework of choice for proof search — so that tactics have to perform some kind of translation between the two formalisms.

At the level of dependent types, there is a number of tricks used in the Agda implementation that diverge from the proof-theoretical viewpoint. For example, substitutions in types are treated in a complex way and may be grouped together. Although some of the design choices can be justified by a similarity to the focused sequent calculus, there is probably a number of implementation techniques that have no proof-theoretical foundation. Moreover, we have chosen here a particularly precise framework where formulas are explicitly polarised, but in practice types in a programming language should not always require these annotations: the question of the presence of specific terms corresponding to shifts is therefore not obvious, as it depends if some interesting programming constructs require their presence or their absence. One can observe, for example, that in the system proposed here, dependencies are subject to the presence of delays, because of the contraction present in the left focus rule and of the treatment of names in the $\kappa x.t$ operation.

The problem of generalising the equational style of programming associated to the focused sequent calculus at the propositional level to the level of dependent types is parametrised by a choice: using patterns seems to require a complex tracking mechanism, but provides a relatively direct logical representation of equations, while using simple variables leads to a translation overhead. Notice however that one could think of an implementation based on variables in which equations are easily obtained, since the language would already be expressed in the style of the sequent calculus — this is the approach suggested by Epigram, where equations are meant to clarify the meaning of programs but are not their internal representation. But we now turn to the most challenging task of our whole enterprise: the accomodation of induction in the framework of a focused sequent calculus in a form that can be exploited to design declarative programs.

Induction can be expressed in Agda in a concise manner and enjoys the benefits of the equational presentation. Consider for example the following inductive scheme for natural numbers:

$$\begin{aligned}&\mathtt{ind}_{\mathbb{N}}\ :\ \mathtt{P}\ \mathtt{zero}\ \to\ (\Pi(\mathtt{x}:\mathbb{N}).\ \mathtt{P}\ \mathtt{x}\ \to\ \mathtt{P}\ (\mathtt{suc}\ \mathtt{x}))\ \to\ \Pi(\mathtt{n}:\mathbb{N}).\ \mathtt{P}\ \mathtt{n}\\ &\mathtt{ind}_{\mathbb{N}}\ \mathtt{base}\ \mathtt{ih}\ \mathtt{zero}\ =\ \mathtt{base}\\ &\mathtt{ind}_{\mathbb{N}}\ \mathtt{base}\ \mathtt{ih}\ (\mathtt{suc}\ \mathtt{n})\ =\ \mathtt{ih}\ \mathtt{n}\ (\mathtt{ind}_{\mathbb{N}}\ \mathtt{base}\ \mathtt{ih}\ \mathtt{n})\end{aligned}$$

where the code essentially relies on the matching of a natural number, that can be either zero or the successor of another number. It is not obvious to see through this program and select a particular approach

to induction that would be a good candidate for a proof-theoretical description. The natural candidate for a representation of induction in the sequent calculus would be the μ operator as studied in [4] in the setting of intuitionistic logic. The unfocused rules for this operator would be, from a purely logical viewpoint:

$$\frac{\Gamma \vdash B\{\mu a.B/a\}}{\Gamma \vdash \mu a.B} \qquad \frac{B\{C/a\} \vdash C}{\Gamma, \mu a.B \vdash C}$$

but the presence of fixpoints has consequences for cut elimination, as it prevents some cuts to be reduced. The usual technique applied to avoid this problem is to build the cut rule into the left rule for μ and to consider the result as cut free. This way, all the cuts that cannot be reduced further are explicitly attached to the blocking rule instance. However, the use of these rules in terms of computation is not obvious to specify, in part because of the complexity of the associated cut reduction, that involves the creation of several other cuts and appeals to the functoriality of the body B of any fixpoint $\mu a.B$ — ensured by a positivity condition. In addition, these rules seem to interact poorly with dependent types, as dealing with fixpoints will require a complex handling of terms appearing inside types. It is unclear as of now if fixpoints as expressed by μ — and ν in the case of induction — can fit our scheme of explaining the implementation of a language such as Agda, but at the same time there is no obvious *proof-theoretical* approach that accounts in a straightforward way for the pervasive nature of inductive definitions in the internal language of Agda, where they are handled by expansion of names with the body of the definition.

4 Conclusion and Future Work

As we have seen here, the $\overline{\lambda}$-calculus proposed by Herbelin as an interpretation of the **LJT** focused sequent calculus can be extended beyond its original scope to include positive connectives, leading to a full-fledged intuitionistic system where we can focus on the right-hand side of sequents to decompose positives. The language we obtain is well-suited to represent programs written in the kind of equational style found in Haskell or Agda, the relation to equations can be made even tighter by using patterns as labels for assumptions in the type system. The opens up the possibility to select focused sequent calculus as a logical framework of choice for the implementation of such languages — as evidenced by the current state of the implementation of Agda, containing many elements that can be explained as sequent calculus constructs. The benefit could not only be a simplication of such an implication, but possibly an improvement in terms of efficiency if advanced techniques from proof theory are transferred and made practical. Moreover, one of the strength of the logical approach is that generalisations and extensions of all kinds are usually made simpler by the strong principles at work: any kind of progress made on the side of proof theory could translate into more expressive languages using the clear equational style of Haskell and Agda — that could be modalities, linearity or many other elements studied in the field of computational logic.

The generalisation of this idea to handle dependent types has already been partially investigated, but some question are left unresolved as to the specific rules used in such a system, and the possibility of making the system more equational by exploiting patterns. But the most difficult task at hand is the explanation of the various treatments of induction available in language and proofs assistants in terms of the sequent calculus. As observed previously [2], the notion of polarity seems to be important in the understanding of this question, but unfortunately the proper polarised handling of fixpoints in proof theory has yet to be found — a number of choices are left open when it comes to the definition of a focused system using fixpoints [5]. Note that our enterprise also yields the question of the treatment of the identity type in proof theory, as it makes dependent pattern matching admit the axiom ***K***. This axiom is undesirable

in homotopy type theory, and thus the restriction of dependent pattern matching has been studied [9]. But this was achieved by restricting unification in the splitting rules, and as Agda has no explicit calculus for splitting, this was somewhat hidden. The framework we want to develop provides a calculus and could thus help making this restriction simpler.

Acknowledgements. This work was funded by the grant number 10-092309 from the Danish Council for Strategic Research to the *Demtech* project.

References

[1] *Haskell, an advanced purely-functional programming language: http://www.haskell.org*.

[2] Andreas Abel, Brigitte Pientka, David Thibodeau & Anton Setzer (2013): *Copatterns: programming infinite structures by observations*. In: *POPL'13*, pp. 27–38, doi:10.1145/2429069.2429075.

[3] Lennart Augustsson (1985): *Compiling Pattern Matching*. In: *FPCA'85*, pp. 368–381, doi:10.1007/3-540-15975-4_48.

[4] David Baelde (2008): *A linear approach to the proof-theory of least and greatest fixed points*. Ph.D. thesis, Ecole Polytechnique.

[5] David Baelde (2012): *Least and Greatest Fixed Points in Linear Logic*. *ACM Transactions on Computational Logic* 13(1), p. 2, doi:10.1145/2071368.2071370.

[6] Henk Barendregt & Silvia Ghilezan (2000): *Lambda-terms for natural deduction, sequent calculus and cut elimination*. *Journal of Functional Programming* 10(1), pp. 121–134.

[7] Taus Brock-Nannestad, Nicolas Guenot & Daniel Gustfasson (2015): *Computation in Focused Intuitionistic Logic*. In: *PPDP'15*, pp. 43–54, doi:10.1145/2790449.2790528.

[8] Serenella Cerrito & Delia Kesner (1999): *Pattern Matching as Cut Elimination*. In: *LICS'99*, pp. 98–108, doi:10.1109/LICS.1999.782596.

[9] Jesper Cockx, Dominique Devriese & Frank Piessens (2014): *Pattern Matching Without K*. In: *ICFP'14*, pp. 257–268, doi:10.1145/2628136.2628139.

[10] Thierry Coquand & Christine Paulin (1988): *Inductively defined types*. In: *Conference on Computer Logic*, *LNCS* 417, pp. 50–66, doi:10.1007/3-540-52335-9_47.

[11] Gilles Dowek, Amy Felty, Gérard Huet, Hugo Herbelin, Chet Murthy, Catherine Parent, Christine Paulin-Mohring & Benjamin Werner (1993): *The Coq proof assistant user's guide*. Technical Report, INRIA.

[12] Roy Dyckhoff & Luís Pinto (1998): *Sequent Calculi for the Normal Terms of the $\lambda\Pi$- and $\lambda\Pi\Sigma$-Calculi*. *Electronic Notes in Theoretical Computer Science* 17, pp. 1–14, doi:10.1016/S1571-0661(05)01182-5.

[13] Hugo Herbelin (1994): *A λ-Calculus Structure Isomorphic to Gentzen-Style Sequent Calculus Structure*. In L. Pacholski & J. Tiuryn, editors: *CSL'94*, *LNCS* 933, pp. 61–75, doi:10.1007/BFb0022247.

[14] Stéphane Lengrand, Roy Dyckhoff & James McKinna (2011): *A Focused Sequent Calculus Framework for Proof Search in Pure Type Systems*. *Logical Methods in Computer Science* 7(1), doi:10.2168/LMCS-7(1:6)2011.

[15] Chuck Liang & Dale Miller (2009): *Focusing and Polarization in Linear, Intuitionistic, and Classical Logics*. *Theoretical Computer Science* 410(46), pp. 4747–4768, doi:10.1016/j.tcs.2009.07.041.

[16] Per Martin-Löf (1984): *Intuitionistic Type Theory*. Studies in Proof Theory, Bibliopolis.

[17] Conor McBride & James McKinna (2004): *The view from the left*. *Journal of Functional Programming* 14(1), pp. 69–111, doi:10.1017/S0956796803004829.

[18] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

[19] Peter Schroeder-Heister (1993): *Rules of Definitional Reflection*. In M. Vardi, editor: *LICS'93*, pp. 222–232, doi:10.1109/LICS.1993.287585.