EPTCS 404

Proceedings of the

Workshop on Logical Frameworks and Meta-Languages: Theory and Practice

Tallinn, Estonia, 8th July 2024

Edited by: Florian Rabe and Claudio Sacerdoti Coen

Published: 8th July 2024
DOI: 10.4204/EPTCS.404
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

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, implementation and their use in reasoning tasks, ranging from the correctness of software to the properties of formal systems, have been the focus of considerable research over the last three decades.

The LFMTP 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.

The 2024 instance of LFMTP was organized by Florian Rabe and Claudio Sacerdoti Coen in Tallinn, Estonia, the 8th July, as a satellite event of the FSCD conference. We are very grateful to the conference organizers for providing the infrastructure and local coordination for the workshop as well as to EPTCS for providing the logistics of publishing these proceedings.

The member of the programme committee were:

- Mauricio Ayala-Rincón (University of Brasilia)
- Mario Carneiro (Carnegie Mellon University)
- Kaustuv Chaudhuri (Inria Saclay)
- Cyril Cohen (Inria Sophia Antipolis)
- Alberto Momigliano (University of Milan, Italy)
- Florian Rabe (University of Erlangen-Nuremberg), co-chair
- Colin Rothgang (IMDEA, Madrid)
- Claudio Sacerdoti Coen (University of Bologna, Italy), co-chair
- Sophie Tourret (Inria Nancy & Loria)
- Theo Winterhalter (Inria Saclay)

Additionally, Alessio Coltellacci and Chuta Sano provided external reviews. The editors are very grateful for their thorough analysis of all submissions.

The workshop received 8 submissions, of which 6 were presented at the workshop. Of these, 2 were work-in-progress presentations, and 4 were accepted for these formal proceedings. Additionally, Carsten Schürmann of IT University of Copenhagen gave an invited talk on Nominal State Separating Proofs.

July 03, 2024

Florian Rabe and Claudio Sacerdoti Coen
PC chairs of LFMTP 2024

A Beluga Formalization of the Harmony Lemma in the π-Calculus

Gabriele Cecilia
Dipartimento di Matematica,
Università degli Studi di Milano, Italy

Alberto Momigliano
Dipartimento di Informatica,
Università degli Studi di Milano, Italy

The "Harmony Lemma", as formulated by Sangiorgi & Walker, establishes the equivalence between the labelled transition semantics and the reduction semantics in the π-calculus. Despite being a widely known and accepted result for the standard π-calculus, this assertion has never been rigorously proven, formally or informally. Hence, its validity may not be immediately apparent when considering extensions of the π-calculus. Contributing to the second challenge of the Concurrent Calculi Formalization Benchmark — a set of challenges tackling the main issues related to the mechanization of concurrent systems — we present a formalization of this result for the fragment of the π-calculus examined in the Benchmark. Our formalization is implemented in Beluga and draws inspiration from the HOAS formalization of the LTS semantics popularized by Honsell et al. In passing, we introduce a couple of useful encoding techniques for handling telescopes and lexicographic induction.

1 Introduction

At page 51 of their "bible" on the π-calculus [19], Sangiorgi & Walker state the *Harmony Lemma*, regarding the relationship between the reduction semantics and the transitional one (LTS). The sketch of the proof starts as follows:

> *Rather than giving the whole (long) proof, we explain the strategy and invite the reader to check some of the details [. . .]*

While this informal style of proof, akin to the infamous "proof on a napkin" championed by de Millo and colleagues[1], may be suitable for a (long) textbook, it might not be applicable to emerging calculi with more unconventional operational semantics. Although the theorem is undisputed within the well-established framework of the π-calculus, this assurance may not extend to these developing calculi. In such instances, a more rigorous approach, potentially in the form of a machine-checked proof, is advisable.

These considerations are of course not novel: they have been prominently argued for in the POPLMark challenge [2] and subsequent follow-ups [1, 7]. The recent Concurrent Calculi Formalization Benchmark [4] (CCFB in brief) introduces a new collection of benchmarks addressing challenges encountered during the mechanization of models of concurrent and distributed programming languages, with an emphasis on process calculi. As with POPLMark, the idea is to explore the state of the art in the formalization in this subarea, finding the best practices to address their typical issues and improving the tools for their mechanization.

CCFB considers in isolation three aspects that may be problematic when mechanizing concurrency theory: *linearity*, *scope extrusion*, and *coinductive reasoning*. Scope extrusion is, of course, the method by which a process can transfer restricted names to another process, as long as the restriction can be

[1] "Social Processes and Proofs of Theorems and Programs", CACM 22-5, 1979.

safely expanded to include the receiving process. This phenomenon has been captured in two different, yet equivalent ways of formulating the operational semantics of the π-calculus:

1. a reduction system, which avoids explicit reasoning about scope extrusion by using structural congruence;
2. a labelled transition system, which introduces a new kind of action to handle extrusion directly: in doing so, it breaks shared conventions such as α-equivalence.[2]

The second challenge in the Concurrent Calculi Formalization Benchmark (CCFB.2) consists in mechanizing these two operational semantics and relating them via the aforementioned Harmony Lemma.

Obviously, we are not the first to address the mechanization of the π-calculus (although we seem to be the first to tackle the Harmony result): given the challenges that it poses (various kind of binders with somewhat unusual properties compared to the λ-calculus), there is a long tradition starting with [11] and mostly developed with encodings based on first-order syntax such as de Brujin indexes — see [4] for a short review of the literature w.r.t. scope extrusion. As often remarked, concrete encodings will get you there, but not effortlessly: an estimation of 75% of the development being devoted to the infrastructure of names handling is not uncommon [8]:

> "Technical work, however, still represents the biggest part of our implementation, mainly due to the managing of De Bruijn indexes [...] Of our 800 proved lemmas, about 600 are concerned with operators on free names."

It is not surprising that specifications based on higher-order abstract syntax (HOAS) soon emerged, first only as animations, see [12] in λProlog and [9] in LF. Moving to meta-reasoning, we can roughly distinguish two main approaches:

1. "squeezing" HOAS into a general proof assistant: there is a plethora of approaches, but w.r.t the π-calculus this has been investigated by Despeyroux [6] and then systematically by Honsell and his colleagues, starting with [10] and then addressing other calculi;
2. the Pfenning-Miller "two-level approach" of separating the specification from the reasoning logic, whose culmination, as far as the π-calculus is concerned, is the most elegant version presented in [21] and later implemented in Abella.

We fall in the second camp and we offer a Beluga [18] mechanization of CCFB.2 together with a detailed informal proof, filling all the gaps left by the quoted sketch. Along the way, we introduce (or simply rediscover) a couple of Beluga tricks to encode *telescopes* (i.e. n-ary sequences of binders) and to simulate lexicographic induction. We also prove another folk result, namely the equivalence between the early and late LTS, as well as what is sometimes called "internal adequacy" [10], that is the equivalence between the LTS encoding from the Honsell paper with the one in [21].

Informal and formal proofs in all their glory can be found here [5]. In the text, the statements of informal lemmas and theorems are hyperlinked to their formalization in the repository. For reasons of space, we will assume familiarity with the basic notions of the π-calculus as in [15], as well as a working knowledge of Beluga, both of its syntax and more importantly of its approach to proof checking.

2 The π-Calculus and its Operational Semantics

In this section, we quickly recall the main notions involved, so as to make the paper self-contained. For more details see [19].

[2]There are also intermediate approaches that save α-equivalence, such as Parrow's LTS with structural congruence [15] or Milner's notion of abstraction and concretion as formalized for example in [3].

2.1 Syntax

We assume the existence of a countably infinite set of *names*, ranged over by $x, y, \ldots$ We make no other assumption about names, since the syntax of *processes* in CCFB.2 does not consider (mis)match. In fact, to concentrate in isolation on scope extrusion, sums and replications are ignored as well:

$$P, Q \ ::= \ \mathbf{0} \mid x(y).P \mid \bar{x}y.P \mid (P \mid Q) \mid (\nu x)P$$

The input prefix $x(y).P$ and the restriction $(\nu y)P$ both bind the name y in P. Any other occurrence of names in a process is free. The sets of free and bound names occurring in a process ($\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ respectively) are defined as usual.

In the mathematical presentation of the operational semantics, we adopt the following slightly weaker variable convention[3]: 1) given a process, it is *possible* to α-rename the bound occurrences of variables within it; 2) the bound names of any processes or actions under consideration *can* be chosen different from the names occurring free in any other entities under consideration.

2.2 Reduction Semantics

We define *structural congruence* ($\equiv$) and *reduction* ($\rightarrow$) as the smallest binary relations over processes, respectively satisfying the axioms in Fig. 1. The notation $Q\{y/z\}$ represents capture-avoiding substitution of y for z in the process Q. Note that we have chosen to present congruence as the compatible refinement of the six basic axioms, rather than using process *contexts* as in [19], since the latter tend to be problematic w.r.t. a HOAS formalization.

2.3 Labelled Transition System Semantics

The syntax of *actions* is the following:

$$\alpha := x(y) \mid \bar{x}y \mid \bar{x}(y) \mid \tau$$

In the input action $x(y)$ and in the bound output action $\bar{x}(y)$, the name x is free and y is bound; in the free output action $\bar{x}y$, both x and y are free. The sets of free names, bound names and names occurring in an action ($\mathsf{bn}(\alpha)$, $\mathsf{fn}(\alpha)$ and $\mathsf{n}(\alpha)$ respectively) are defined accordingly. The *transition* relation $\cdot \xrightarrow{\cdot} \cdot$ is the smallest relation which satisfies the rules in Fig. 2.

Unlike the reduction semantics, the transitional semantics directly addresses scope extrusion via the two S-CLOSE rules in interaction with S-OPEN: recall how the former rules are *not* closed under α-conversion, since the bound name z must occur free in the other premise.

The LTS introduced here is the *late* semantics, as opposed to the *early* one adopted by the Benchmark. However, as remarked in [15], "it is a matter of taste which semantics to adopt". We indeed prove this equivalence in Appendix A.

2.4 The Harmony Lemma

In [19], the Harmony Lemma reads as:

i. $P \equiv\xrightarrow{\alpha} Q$ implies $P \xrightarrow{\alpha}\equiv Q$.

[3]Variable conventions are used in a rather loose way in the literature, e.g. Parrow and Sangiorgi & Walker adopt the same convention, but end up with different provisos in the operational semantics rules.

$$\textsc{Par-Assoc}\ \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \qquad \textsc{Par-Unit}\ \frac{}{P \mid \mathbf{0} \equiv P} \qquad \textsc{Par-Comm}\ \frac{}{P \mid Q \equiv Q \mid P}$$

$$\textsc{Sc-Ext-Zero}\ \frac{}{(\nu x)\mathbf{0} \equiv \mathbf{0}} \qquad \textsc{Sc-Ext-Par}\ \frac{x \notin \mathsf{fn}(Q)}{(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)} \qquad \textsc{Sc-Ext-Res}\ \frac{}{(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P}$$

$$\textsc{C-In}\ \frac{P \equiv Q}{x(y).P \equiv x(y).Q} \qquad \textsc{C-Out}\ \frac{P \equiv Q}{\bar{x}y.P \equiv \bar{x}y.Q} \qquad \textsc{C-Par}\ \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad \textsc{C-Res}\ \frac{P \equiv Q}{(\nu x)P \equiv (\nu x)Q}$$

$$\textsc{C-Ref}\ \frac{}{P \equiv P} \qquad \textsc{C-Sym}\ \frac{P \equiv Q}{Q \equiv P} \qquad \textsc{C-Trans}\ \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R}$$

$$\textsc{R-Com}\ \frac{}{\bar{x}y.P \mid x(z).Q \;\to\; P \mid Q\{y/z\}} \qquad \textsc{R-Par}\ \frac{P \to Q}{P \mid R \to Q \mid R}$$

$$\textsc{R-Res}\ \frac{P \to Q}{(\nu x)P \to (\nu x)Q} \qquad \textsc{R-Struct}\ \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

Figure 1: Congruence and reduction rules.

ii. $P \xrightarrow{\tau}\equiv Q$ iff $P \to Q$.

The juxtaposition of symbols denotes relational composition (e.g. $P \equiv\xrightarrow{\alpha} Q$ denotes $P \equiv R$ and $R \xrightarrow{\alpha} Q$ for some R). The first assertion is a direct consequence of Lemma 2.6, as detailed at page 6, which is instrumental to prove the right-to-left direction of the equivalence result. The latter breaks down into the following theorems:

1. Every transition through a τ action corresponds to a reduction;
2. Given a reduction of P to Q, P is able to make a τ-transition to some Q' congruent to Q.

In the interest of setting the stage for anybody who wishes to give a solution to CCFB.2, we start by stating a few technical lemmas about substitutions that are used in both directions of the Harmony Lemma, while being often left unsaid.

Lemma S1 $Q\{x/x\} = Q$.

Lemma S2 *If $x \notin fn(Q)$, then $Q\{y/x\} = Q$.*

These two lemmas are proved by induction on the structure of the process Q. A consequence of the latter is the following: if $x \notin \mathsf{fn}(Q)$, then $P\{y/x\} \mid Q = (P \mid Q)\{y/x\}$.

Finally, we state a stability result for structural congruence under substitutions, only used in the second direction of Harmony:

Lemma S3 *If $P \equiv Q$, then $P\{y/x\} \equiv Q\{y/x\}$.*

This lemma is proved by induction on the structure of the given derivation.

$$\textsc{S-In}\quad \frac{}{x(z).P \xrightarrow{x(z)} P} \qquad\qquad \textsc{S-Out}\quad \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}$$

$$\textsc{S-Par-L}\quad \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad\qquad \textsc{S-Par-R}\quad \frac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textsc{S-Com-L}\quad \frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad\qquad \textsc{S-Com-R}\quad \frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P'\{y/z\} \mid Q'}$$

$$\textsc{S-Res}\quad \frac{P \xrightarrow{\alpha} P' \qquad z \notin \mathsf{n}(\alpha)}{(\nu z)P \xrightarrow{\alpha} (\nu z)P'} \qquad\qquad \textsc{S-Open}\quad \frac{P \xrightarrow{\bar{x}z} P' \qquad z \neq x}{(\nu z)P \xrightarrow{\bar{x}(z)} P'}$$

$$\textsc{S-Close-L}\quad \frac{P \xrightarrow{\bar{x}(z)} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \qquad\qquad \textsc{S-Close-R}\quad \frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}$$

Figure 2: Transition rules.

2.4.1 Theorem 1: τ-Transition Implies Reduction

The proof of the first direction relies on three key lemmas which describe rewriting (up to structural congruence) of processes involved in input and output transitions.

Lemma 1.1 *If $Q \xrightarrow{x(y)} Q'$ then there exist a finite (possibly empty) set of names $w_1, \dots, w_n$ (with $x, y \neq w_i \ \forall i = 1, \dots, n$) and two processes R, S such that $Q \equiv (\nu w_1) \dots (\nu w_n)(x(y).R \mid S)$ and $Q' \equiv (\nu w_1) \dots (\nu w_n)(R \mid S)$.*

Lemma 1.2 *If $Q \xrightarrow{\bar{x}y} Q'$ then there exist a finite (possibly empty) set of names $w_1, \dots, w_n$ (with $x, y \neq w_i \ \forall i = 1, \dots, n$) and two processes R, S such that $Q \equiv (\nu w_1) \dots (\nu w_n)(\bar{x}y.R \mid S)$ and $Q' \equiv (\nu w_1) \dots (\nu w_n)(R \mid S)$.*

Lemma 1.3 *If $Q \xrightarrow{\bar{x}(z)} Q'$ then there exist a finite (possibly empty) set of names $w_1, \dots, w_n$ (with $x \notin \{z, w_1, \dots, w_n\}$) and two processes R, S such that $Q \equiv (\nu z)(\nu w_1) \dots (\nu w_n)(\bar{x}z.R \mid S)$ and $Q' \equiv (\nu w_1) \dots (\nu w_n)(R \mid S)$.*

These three lemmas are proved by induction over the structure of the given transition. We observe that the presence of a sequence of binders is not an issue in the informal presentation; on the other hand, from the mechanization point of view, these sequences are challenging to encode in a framework where the meta-level binder is unary.

Theorem 1 *$P \xrightarrow{\tau} Q$ implies $P \rightarrow Q$.*

The theorem is proved by induction on the structure of the given transition. If the latter consists of an explicit interaction of processes in a parallel composition, we apply the aforementioned lemmas to rewrite processes involved in specific transitions up to congruence; we then construct the desired reduction through a chain of congruence and reduction rules.

Corollary 1.1 *$P \xrightarrow{\tau}\equiv Q$ entails $P \to Q$.*

2.4.2 Theorem 2: Reduction Implies τ-Transition

The other direction starts with five technical lemmas regarding free and bound names in specific transitions. They are instrumental, together with the variable convention, to the firing of the appropriate transitions.

Lemma 2.1 *If $P \xrightarrow{\bar{x}y} P'$, then $x, y \in fn(P)$.*

Lemma 2.2 *If $P \xrightarrow{x(y)} P'$, then $x \in fn(P)$.*

Lemma 2.3 *If $P \xrightarrow{\bar{x}(z)} P'$, then $x \in fn(P)$ and $z \in bn(P)$.*

Lemma 2.4 *If $P \xrightarrow{\alpha} P'$, $x \notin n(\alpha)$ and $x \notin fn(P)$, then $x \notin fn(P')$.*

Lemma 2.5 *If $P \equiv P'$, then $x \in fn(P) \Leftrightarrow x \in fn(P')$.*

The first four lemmas follow by induction over the structure of the given transition. The last by induction on the congruence judgment.

The next key ingredient is establishing that structural congruence is a strong late bisimulation.

Lemma 2.6 *Let $P \equiv Q$.*

1. *If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$.*
2. *If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $P' \equiv Q'$.*

The two statements need to be proven at the same time by mutual induction over the derivation of the congruence judgment and case analysis on the given transition.

Finally, a rewriting lemma for reduction, again proven by induction on the structure of the given reduction judgment:

Lemma 2.7 *If $P \to Q$ then there exist three names x, y and z, a finite (possibly empty) set of names $w_1, \ldots, w_n$ and three processes R_1, R_2 and S such that $P \equiv (\nu w_1)\ldots(\nu w_n)((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\nu w_1)\ldots(\nu w_n)((R_1 \mid R_2\{y/z\}) \mid S)$.*

Theorem 2 *$P \to Q$ implies the existence of a Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.*

The proof follows immediately from the application of Lemmas 2.6 and 2.7.

3 Beluga Formalization

This section provides an overview of the formalization of the definitions and proofs introduced in the previous section with the proof assistant Beluga. The complete formalization is accessible at [5].

3.1 Syntax

Fig. 3 displays the syntax of names and processes. Since names are just an infinite set, we encode them with an LF type `names` without any constructor, which will be extended with new inhabitants dynamically in the operational semantics. This is made possible by the declaration

```
schema ctx = names;
```

indicating that all relevant judgments involving open terms (processes) are formulated in contexts categorized by the above schema.

Processes are encoded using (weak) HOAS, as well trodden in the literature: input and restrictions abstract over names, in particular the input process $x(y).P$ is encoded by the term `p_in X \y.(P y)`, where the bound name y in P is represented by the implicit argument of the LF function `\y.(P y)`. As usual, α-renaming and capture-avoiding substitutions are automatically implemented by the meta-language. From now on, semicolons and infix constructor declarations will be omitted from code snippets for brevity.

```
LF names: type =
;
LF proc: type =
  | p_zero: proc
  | p_in: names → (names → proc) → proc
  | p_out: names → names → proc → proc
  | p_par: proc → proc → proc --infix p_par 11 left.
  | p_res: (names → proc) → proc
;
```

Figure 3: Encoding of names and processes.

Recall that LF types are *not* inductive. While the LF type `names` has no constructor, the *contextual* type `[g ⊢ proc]`, for `g` ∈ `ctx`, denotes the set of open processes built out of LF variables ranging over names.

3.2 Reduction Semantics

Congruence and reduction are encoded by the type families `cong` and `red` respectively, as presented in Fig. 4. As usual, we use universal quantification `{x:names}` to descend into binders, e.g. in the compatibility rule for restriction. Scope extension is realized in rule `sc_ext_par` by simply *not* having Q depend on the restricted channel, hence meeting the side condition $x \notin \mathsf{fn}(Q)$ in the SC-EXT-PAR automatically. In the same vein, substitution in rule R-COM is encoded by meta-level β-reduction.

3.3 Labelled Transition System Semantics

We follow Honsell et al. [10] for the encoding of the late LTS semantics. We declare two different relations for transitions via free and bound actions. The result of a free transition is a process, while the result of a bound transition is a process abstraction: instead of explicitly stating the bound name involved in the transition, that name is the argument of the aforementioned function. This is reflected by the encoding of free and bound actions, which only mention the free names involved. Fig. 5 shows the types

```
% Structural Congruence
LF cong: proc → proc → type =
% Abelian Monoid Laws for Parallel Composition
  | par_assoc: cong (P p_par (Q p_par R)) ((P p_par Q) p_par R)
  | par_unit: cong (P p_par p_zero) P
  | par_comm: cong (P p_par Q) (Q p_par P)
% Scope Extension Laws
  | sc_ext_zero: cong (p_res (\x.p_zero)) p_zero
  | sc_ext_par: cong ((p_res P) p_par Q) (p_res (\x.((P x) p_par Q)))
  | sc_ext_res: cong (p_res \x.(p_res \y.(P x y))) (p_res \y.(p_res \x.(P x y)))
% Compatibility Laws
  | c_in: ({y:names} cong (P y) (Q y)) → cong (p_in X P) (p_in X Q)
  | c_out: cong P Q → cong (p_out X Y P) (p_out X Y Q)
  | c_par: cong P P' → cong (P p_par Q) (P' p_par Q)
  | c_res: ({x:names} cong (P x) (Q x)) → cong (p_res P) (p_res Q)
% Equivalence Relation Laws
  | c_ref: cong P P
  | c_sym: cong P Q → cong Q P
  | c_trans: cong P Q → cong Q R → cong P R

% Reduction
LF red: proc → proc → type =
  | r_com: red ((p_out X Y P) p_par (p_in X Q)) (P p_par (Q Y))
  | r_par: red P Q → red (P p_par R) (Q p_par R)
  | r_res: ({x:names} red (P x) (Q x)) → red (p_res P) (p_res Q)
  | r_str: P cong P' → red P' Q' → Q' cong Q → red P Q
```

Figure 4: Encoding of congruence and reduction.

f_act and b_act encoding free and bound actions and the two mutually defined type families fstep and bstep which encode free and bound transitions respectively[4]. Note that none of the side conditions of the transition rules need to be explicitly stated, nor do we need the axioms and additional freshness predicates as in [10].

HOAS encodings customarily come with an (informal) *adequacy* proof, ensuring that there is a compositional bijection between the mathematical model and its encoding (in canonical form). While this is fairly obvious for processes, congruence and reduction, it is less so w.r.t. the LTS semantics. Luckily, this has been carefully proven both in [10] and in [21] for a related version. We refer to those papers for further details and to the repository for a Beluga proof of the "internal" adequacy of those two encodings.

3.4 The Harmony Lemma

One of the expected yet very much appreciated payoffs of a HOAS encoding is that most (in fact all but Lemma 2.4) boilerplate lemmas about names, occurrences and substitution vanish. We are referring to the substitution Lemmas S1—S3, as well as the free/bound names Lemmas 2.1, 2.2, 2.3 and 2.5.

[4]Interestingly, a similar approach is pursued by Cheney in his nominal encoding in αProlog, see https://homepages.inf.ed.ac.uk/jcheney/programs/aprolog/examples/picalc.apl.

```
% Free Actions                             % Bound Actions
LF f_act: type =                           LF b_act: type =
  | f_tau: f_act                             | b_in: names → b_act
  | f_out: names → names → f_act             | b_out: names → b_act

% Transition Relation
LF fstep: proc → f_act → proc → type =
  | fs_out: fstep (p_out X Y P) (f_out X Y) P
  | fs_par1: fstep P A P' → fstep (P p_par Q) A (P' p_par Q)
  | fs_par2: fstep Q A Q' → fstep (P p_par Q) A (P p_par Q')
  | fs_com1: fstep P (f_out X Y) P' → bstep Q (b_in X) Q'
    → fstep (P p_par Q) f_tau (P' p_par (Q' Y))
  | fs_com2: bstep P (b_in X) P' → fstep Q (f_out X Y) Q'
    → fstep (P p_par Q) f_tau ((P' Y) p_par Q')
  | fs_res: ({z:names} fstep (P z) A (P' z)) → fstep (p_res P) A (p_res P')
  | fs_close1: bstep P (b_out X) P' → bstep Q (b_in X) Q'
    → fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))
  | fs_close2: bstep P (b_in X) P' → bstep Q (b_out X) Q'
    → fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))

and bstep: proc → b_act → (names → proc) → type =
  | bs_in: bstep (p_in X P) (b_in X) P
  | bs_par1: bstep P A P' → bstep (P p_par Q) A \x.((P' x) p_par Q)
  | bs_par2: bstep Q A Q' → bstep (P p_par Q) A \x.(P p_par (Q' x))
  | bs_res: ({z:names} bstep (P z) A (P' z))
    → bstep (p_res P) A \x.(p_res \z.(P' z x))
  | bs_open: ({z:names} fstep (P z) (f_out X z) (P' z)) → bstep (p_res P)(b_out X) P'
```

Figure 5: Encoding of actions and transition.

3.4.1 Theorem 1: τ-Transition Implies Reduction

We start with the encoding of Lemmas 1.1, 1.2 and 1.3. There are two issues, one standard, the other slightly more challenging. For one, the conclusion of these lemmas includes an existential quantification, and Beluga lacks such a construct; the usual workaround consists of defining a new type family which encodes the existential quantification. Secondly, and more seriously, the statements refer to sequences of binders (here restrictions), sometimes referred to as *telescopes*. We can give a combined solution to these items by encoding them *inductively*, where the base case describes when the existential holds for the empty sequence and the inductive one adds one more binder. More specifically for Lemma 1.1, we say that the congruences $(\star)$ $(Q \equiv (\nu w_1)\dots(\nu w_n)(x(y).R \mid S)$ and $Q' \equiv (\nu w_1)\dots(\nu w_n)(R \mid S))$ hold for two processes Q and Q' iff one of the following holds:

i. $Q \equiv x(y).R \mid S$ and $Q' \equiv R \mid S$;

ii. $Q \equiv (\nu w)P$, $Q' \equiv (\nu w)P'$ and the congruences $(\star)$ hold for P and P'.

We list the definition of the type family `ex_inp_rew` that encodes the above judgment; the types `ex_fout_rew` and `ex_bout_rew` are defined analogously.

```
LF ex_inp_rew: proc → names → (names → proc) → type =
  | inp_base: Q cong ((p_in X R) p_par S) → ({y:names} (Q' y) cong ((R y) p_par S))
    → ex_inp_rew Q X Q'
```

```
rec bs_in_rew: (g:ctx) [g ⊢ bstep Q (b_in X) \y.Q'[..,y]]
  → [g ⊢ ex_inp_rew Q X \y.Q'[..,y]] =
/ total b (bs_in_rew _ _ _ _ b) /
fn b ⇒ case b of
   | [g ⊢ bs_in] ⇒ [g ⊢ inp_base (c_sym par_unit) \y.(c_sym par_unit)]
   | [g ⊢ bs_par1 B1]:[g ⊢ bstep (P p_par R) (b_in X) (\y.(P' p_par (R[..])))] ⇒
     let [g ⊢ D1] = bs_in_rew [g ⊢ B1] in
     let [g ⊢ D2] = bs_in_rew_par1 [g ⊢ R] [g ⊢ D1] in [g ⊢ D2]
   | [g ⊢ bs_par2 B2]:[g ⊢ bstep (R p_par P) (b_in X) (\y.((R[..]) p_par P'))] ⇒
     let [g ⊢ D1] = bs_in_rew [g ⊢ B2] in
     let [g ⊢ D2] = bs_in_rew_par2 [g ⊢ R] [g ⊢ D1] in [g ⊢ D2]
   | [g ⊢ bs_res \y.B[..,y]] ⇒
     let [g,y:names ⊢ D1[..,y]] = bs_in_rew [g,y:names ⊢ B[..,y]] in
     let [g ⊢ D2] = bs_in_rew_res [g,y:names ⊢ D1[..,y]] in [g ⊢ D2]
```

Figure 6: Proof of Lemma 1.1.

```
  | inp_ind: Q cong (p_res P) → ({y:names} (Q' y) cong (p_res (P' y)))
    → ({w:names} ex_inp_rew (P w) X \y.(P' y w)) → ex_inp_rew Q X Q'
```

We prove each lemma by defining a total recursive function which receives a contextual derivation of an input/free output/bound output transition respectively and returns a contextual object of the corresponding existential type. We provide the proof term of the `bs_in_rew` function, which proves Lemma 1.1, in Fig. 6; the functions encoding Lemmas 1.2 and 1.3 are omitted.

The proof proceeds by induction on the structure of the given assumption. If it has been obtained through the S-In rule, we are in the base case of the existential with no binders: hence, we conclude immediately modulo symmetry of congruence. In case the input transition has been obtained through the S-Par-L rule, it can be rewritten as $P \mid R \xrightarrow{x(y)} P' \mid R$, with the transition `B1`: $P \xrightarrow{x(y)} P'$ as hypothesis. In such situation we can apply a recursive call of the `bs_in_rew` function to `B1`, obtaining an object `D1` of type `ex_inp_rew` encoding the rewriting for P and P'; to conclude, we appeal to the lemma `bs_in_rew_par1` in order to unfold `D1` and build the required existential object encoding the rewriting for $P \mid R$ and $P' \mid R$. Here is the signature of the above lemma:

```
rec bs_in_rew_par1: (g:ctx) {R:[g ⊢ proc]} [g ⊢ ex_inp_rew Q X \y.Q'[..,y]]
  → [g ⊢ ex_inp_rew (Q p_par R) X \y.(Q'[..,y] p_par R[..])] = ...
```

The other two cases of the main proof follow the same pattern and require similar auxiliary lemmas.

We are now ready to discuss the proof of the main result of this section, namely that $P \xrightarrow{\tau} Q$ implies $P \rightarrow Q$ (Theorem 1):

```
rec fstep_impl_red: (g:ctx) [g ⊢ fstep P f_tau Q] → [g ⊢ P red Q] = ...
```

The proof proceeds by induction on the derivation of `[g ⊢ fstep P f_tau Q]`. Mirroring the informal proof, in certain subcases it is enough to apply a recursive call of the function on a structurally smaller τ-transition and return the desired object of type `[g ⊢ P red Q]`. In the other subcases, we apply the functions `bs_in_rew`, `fs_out_rew` and `bs_out_rew` on the given input/output/bound output transitions, obtaining the corresponding objects of existential type, viz. the cited `ex_inp_rew` and its "siblings" `ex_fout_rew` and `ex_bout_rew`. Then, we would like to conclude by applying some auxiliary functions which unfold these objects and build the desired reduction. Here, we face a major hurdle, not so much in writing down the proof terms, but in having them checked for *termination*, which is, after all, what guarantees that the inductive structure of the proof is correct. Let us see one of such lemmas, which emerges in the subcase where a S-COM-L rule is applied:

```
rec fs_com1_impl_red: (g:ctx) [g ⊢ ex_fout_rew P1 X Y Q1]
  → [g ⊢ ex_inp_rew P2 X \x.Q2[..,x]]
  → [g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])] = ...
```

The proof needs to consider both hypotheses, in order to unfold the two existential types – recall that the telescopes force upon us an inductive encoding of those existentials. In other terms, verification of the fact that this function is decreasing would need to appeal to a form of *lexicographic* induction.

Termination checkers are of course incomplete and adopt strict syntactic criteria to enforce it; in particular, Beluga's checker currently does not support lexicographic induction. One way out is to define the `fs_com1_impl_red` function so that it is decreasing on the first argument only, and relies on an auxiliary function addressing its base case that is decreasing on the second argument. The signature of such a function is:

```
rec fs_com1_impl_red_base: (g:ctx) [g ⊢ P2 cong ((p_in X \x.R[..,x]) p_par S)]
   → [g,w:names ⊢ Q2[..,w] cong (R[..,w] p_par S[..])]
   → [g ⊢ ex_fout_rew P1 X Y Q1]
   → [g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])] = ...
```

Once these lemmas are in place, the proof of the first direction of the Harmony Lemma follows without any further drama.

3.4.2 Theorem 2: Reduction Implies τ-Transition

In the other direction, having HOAS disposed of most of the technical lemmas about names and substitutions, the workhorses are the reduction rewriting Lemma 2.7 and the congruence-as-bisimilarity Lemma 2.6. Since the former does not introduce new ideas, we discuss it first.

Given its similarity to Lemmas 1.1–1.3, Lemma 2.7 is implemented with the same strategy: we define an existential type `ex_red_rew` that inductively encodes the existence of the telescopes and the two congruences stated in the conclusion of the lemma.

```
LF ex_red_rew: proc → proc → type =
  | red_base: P cong (((p_out X Y R1) p_par (p_in X R2)) p_par S)
    → Q cong ((R1 p_par (R2 Y)) p_par S) → ex_red_rew P Q
  | red_ind: P cong (p_res P') → Q cong (p_res Q')
    → ({w:names} ex_red_rew (P' w) (Q' w)) → ex_red_rew P Q
```

We then implement some auxiliary functions to unfold objects of the existential type `ex_red_rew` in specific subcases, for example:

```
rec red_impl_red_rew_par: (g:ctx) {R:[g ⊢ proc]} [g ⊢ ex_red_rew P Q]
  → [g ⊢ ex_red_rew (P p_par R) (Q p_par R)] = ...
```

Having established those, it is straightforward to prove

```
rec red_impl_red_rew: (g:ctx) [g ⊢ P red Q] → [g ⊢ ex_red_rew P Q] = ...
```

by induction on the structure of the given reduction judgment.

Moving on to the proof of Lemma 2.6, there is a new technicality to address. We have seen how in a HOAS setting provisos such as "$x \notin \mathsf{fn}(P)$" are realized by P being in the scope of a meta level abstraction that binds x, but *not* actually depending on x. Sometimes (see [20] for other instances), we have to convince our proof environment of this non-dependency, which is basically the content of

Lemma 2.4, in words: "given a transition $P \xrightarrow{\alpha} Q$ where x does not occur free in P, then x does not occur free in α and Q". Since in Beluga judgments over open terms are encapsulated in the context where they make sense, removing these spurious dependencies amounts to "strengthening" such a context, akin to strengthening lemmas in type theory. The lemma more formally reads as:

$$\text{If } \Gamma, x : \mathit{names} \vdash P \xrightarrow{\alpha_x} Q_x, \text{ then there are } \alpha', Q' \text{ such that } \alpha_x = \alpha', Q_x = Q' \text{ and } \Gamma \vdash P \xrightarrow{\alpha'} Q' \qquad (1)$$

Not only Beluga does not have existentials (nor conjunctions), but LF also has no built-in equality notion. However, this is easy to simulate via pattern unification over canonical forms, by defining three type families encoding equality of processes, free actions and bound actions respectively. Process equality is defined as:

```
LF eqp: proc → proc → type =
  | prefl: eqp P P
```

Since we are now dealing with a property about a LF contextual object (the initial transition), the statement in (1) has to be encoded at the computation level as an *inductive* type [17]. We list this encoding, omitting the definition of its counterpart for bound transitions `ex_str_bstep`.

```
inductive ex_str_fstep: (g:ctx) [g,x:names ⊢ fstep P[..] A Q] → ctype =
  | ex_fstep: {F:[g,x:names ⊢ fstep P[..] A Q]} [g ⊢ fstep P A' Q']
    → [g,x:names ⊢ eqf A A'[..]] → [g,x:names ⊢ eqp Q Q'[..]]
    → ex_str_fstep [g,x:names ⊢ F]
```

Note how non-occurrence is realized using *weakening* substitutions, i.e. `P[..]` signals that the process `P` depends only on the variables mentioned in `g`, excluding `x`.

The strengthening lemma is implemented through the two following mutually recursive functions:

```
rec strengthen_fstep: (g:ctx) {F:[g,x:names ⊢ fstep P[..] A Q]}
  → ex_str_fstep [g,x:names ⊢ F] = ...
and rec strengthen_bstep: (g:ctx) {B:[g,x:names ⊢ bstep P[..] A \z.Q[..,x,z]]}
  → ex_str_bstep [g,x:names ⊢ B] = ...
```

The proof follows by a long but straightforward induction on the structure of the given transition.

To state Lemma 2.6, we again need to code the existential in its conclusion; however, since the statement involves transitions through a generic action α, we actually require two new type families: one for free transitions and one for bound transitions.

```
LF ex_fstepcong: proc → proc → f_act → proc → type =
  | fsc: fstep Q A Q' → P' cong Q' → ex_fstepcong P Q A P'
LF ex_bstepcong: proc → proc → b_act → (names → proc) → type =
  | bsc: bstep Q A Q' → ({x:names} (P' x) cong (Q' x)) → ex_bstepcong P Q A P'
```

The reader may wonder why the process `P` is mentioned in the `fsc` and `bsc` constructors, as it does not play any role: indeed, it is a trick to please Beluga's coverage checker when this definition is unfolded in the rest of the development.

Lemma 2.6 is encoded through the definition of four mutually recursive functions: as the statement involves transitions via a generic action α, the first two prove the result for free transitions, while the last two demonstrate the result for bound transitions; moreover, since the proof is carried out by concurrently establishing two symmetrical assertions, the odd functions prove the left-to-right statement, while the even ones demonstrate the right-to-left statement. We present the signature of the first function `cong_fstepleft_impl_stepright`, which proves the left-to-right assertion for free transitions:

```
rec cong_fstepleft_impl_fstepright: (g:ctx) [g ⊢ P cong Q] → [g ⊢ fstep P A P’]
  → [g ⊢ ex_fstepcong P Q A P’] = ...
```

Mirroring the informal proof, this lemma is proved by a long induction on the structure of the given congruence; in most of the subcases, case analysis of the given transition is performed as well.

A final ingredient for the proof of Theorem 2 is the auxiliary lemma:

```
rec red_rew_impl_fstepcong: (g:ctx) [g ⊢ ex_red_rew P Q]
  → [g ⊢ ex_fstepcong P P f_tau Q] = ...
```

The proof proceeds by induction on the structure of the given object of type `[g ⊢ ex_red_rew P Q]`; in both the base case and the inductive case, a key factor consists in the application of the function `cong_fstepright_impl_fstepleft`.

We are ready to prove Theorem 2:

```
rec red_impl_fstepcong: (g:ctx) [g ⊢ P red Q] → [g ⊢ ex_fstepcong P P f_tau Q] =
/ total r (red_impl_fstepcong _ _ _ r) /
fn r ⇒ let [g ⊢ D1] = red_impl_red_rew r in
       let [g ⊢ D2] = red_rew_impl_fstepcong [g ⊢ D1] in [g ⊢ D2]
```

Given `r` representing the reduction $P \longrightarrow Q$, we apply the function `red_impl_red_rew` to it returning an object `D1` of type `[g ⊢ ex_red_rew P Q]` that encodes the following congruences: $P \equiv (\nu w_1)\cdots(\nu w_n)((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\nu w_1)\cdots(\nu w_n)((R_1 \mid R_2\{y/z\}) \mid S)$, for some R_1, R_2, S and $w_1,\ldots,w_n$. Finally, we invoke the auxiliary function `red_rew_impl_fstepcong`, which unfolds the argument `D1` and returns the desired object of type `[g ⊢ ex_fstepcong P P f_tau Q]`.

4 Evaluation and Conclusions

The reader may wonder “Is that it?” We sympathize with the feeling: what is remarkable in this formalization is how (mostly) uneventful it has been. Once we had settled on using (weak) HOAS and a specialized proof environment such as Beluga — which, given our lineage, was not much of a stretch — the Honsell/Miller encoding of the labelled transition system removed all issues related to scope extrusion and Beluga’s remarkable conciseness did the rest, turning 30 pages of LaTeX proofs, which still skip many steps, into some 700 lines of proof terms.

Remarkably, the structure of the formal proof closely mirrors the informal one: having eliminated the 7 technical lemmas thanks to the HOAS encoding, both proofs share the same 6 lemmas, proved in the same fashion. Some parts of the formal proof are covered by 22 additional lemmas which deal with the unfolding of the existential types,[5] while another 4 lemmas result from the mutual recursion induced by the encoding.

In our biased opinion, this uneventfulness does not trivialize the accomplishment: we have provided a compact and elegant solution of a benchmark problem, which, after all, is supposed to be challenging: the fact that this has not been a heroic feat is a testament to the merits of the HOAS encoding and to the long line of research that has made meta-level reasoning over HOAS specifications possible. It also suggests that, once we put on the HOAS “spectacles”, the binders of the π-calculus are not that different from those of the λ-calculus, in the sense that the meta-level binder will gladly model scope extrusion, under the right encoding.

[5]Given how widespread existential (and conjunctive) statements are in this development, it would be helpful if Beluga could provide some syntactic sugar, similarly to Agda, and a way to unfold those definitions automatically.

Beluga has shown to be a reliable system: we did stress the termination checker, with a heavy use of mutually defined recursive functions. We managed to get around the current lack of support for lexicographic induction; our technique could have broader applicability, for instance in verifying totality results such as the admissibility of cut elimination that rely on nested induction for their termination proofs [16]. We also established coverage, again getting around the minor glitch that we have mentioned above in the proof of the right-to-left direction of the Harmony Lemma.

Since the benchmark is amenable to a weak HOAS encoding, this begs the question of why not pursue its solution in a general proof assistant such as Coq. While weak HOAS is indeed consistent with monotonic inductive types, it is well known that the full dependent function space of a theory such as the Calculus of Constructions is incompatible with the intensional quantification featured by LF-like type theories. Workarounds exist: the most successful one is the "Theory of Contexts" [10], where additional predicates concerning freshness and non-occurrence of names are added to specifications such as of the LTS. Further, ToC assumes some axioms regulating the properties of names and abstraction over names (i.e. "contexts") in order to reify what LF-like frameworks provide natively. To be fair, it is unclear to us which role ToC would play in a Coq solution of CCFB.2, but for the rest of the Concurrent Benchmark, the outcome is not pretty (believe us, we tried). Of course, there is no obstacle in abandoning HOAS for concrete encodings and we are looking forward to comparing such solutions to ours.

Although CCFB.2 does not ask for it, we conjecture that it would be easy, albeit tedious, to extend our solution to account for other features of the π-calculus, namely sums, replication and match. Mismatch, which is handled in [10], is rather problematic in HOAS, since the systems that support it have no native notion of negation.

The adoption of Beluga as a proof environment for this formalization is motivated by our endeavor (together with Pientka's group) to give an overall HOAS solution to all the challenges listed in CCFB, which include type safety for (linear) session types and turning strong barbed bisimilarity into a congruence. For this, Beluga is a strong candidate: in fact the type safety challenge is already in the bag, thanks to the techniques developed in [20]. The coinduction part is more challenging, but we have a good track record in a similar benchmark [14]. Solving the rest of CCFB in Beluga may also shed some light on the role of the ∇ quantifier [13] as a meta-reasoning tool, compared to Beluga's use of contextual LF as a specification language.

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark Reloaded: Mechanizing Proofs by Logical Relations*. *J. Funct. Program.* 29:19, doi:10.1017/S0956796819000170.

[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 Joe Hurd & Tom Melham, editors: *Theorem Proving in Higher Order Logics*, Springer, Berlin & Heidelberg, pp. 50–65, doi:10.1007/11541868_4.

[3] Jesper Bengtson & Joachim Parrow (2009): *Formalising the π-Calculus using Nominal Logic*. *Log. Methods Comput. Sci.* 5, doi:10.2168/LMCS-5(2:16)2009.

[4] Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Nobuko Yoshida & Daniel Zackon (2024): *The Concurrent Calculi Formalisation Benchmark*. In Ilaria Castellani & Francesco Tiezzi, editors: *Coordination Models and Languages*, Springer Nature Switzerland, Cham, pp. 149–158, doi:10.1007/978-3-031-62697-5_9.

[5] Gabriele Cecilia (2024): *Formalizing the Operational Semantics of the π-Calculus*. Master's thesis, Università degli Studi di Milano. Available at `https://github.com/GabrieleCecilia/concurrent-benchmark-solution`.

[6] Joëlle Despeyroux (2000): *A Higher-Order Specification of the π-Calculus*. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses & Takayasu Ito, editors: *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, *Lecture Notes in Computer Science* 1872, Springer, pp. 425–439, doi:`10.1007/3-540-44929-9_30`.

[7] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2018): *Benchmarks for Reasoning with Syntax Trees Containing Binders and Contexts of Assumptions*. *Math. Struct. Comput. Sci.* 28(9), pp. 1507–1540, doi:`10.1017/S0960129517000093`.

[8] Daniel Hirschkoff (1997): *A Full Formalisation of π-Calculus Theory in the Calculus of Constructions*. In Elsa L. Gunter & Amy P. Felty, editors: *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, *Lecture Notes in Computer Science* 1275, Springer, pp. 153–169, doi:`10.1007/BFB0028392`.

[9] Furio Honsell, Marina Lenisa, Ugo Montanari & Marco Pistore (1998): *Final Semantics for the π-Calculus*. In David Gries & Willem P. de Roever, editors: *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98) 8-12 June 1998, Shelter Island, New York, USA*, *IFIP Conference Proceedings* 125, Chapman & Hall, pp. 225–243, doi:`10.1007/978-0-387-35358-6_17`.

[10] 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`.

[11] T. F. Melham (1994): *A Mechanized Theory of the π-Calculus in HOL*. *Nordic J. of Computing* 1(1), p. 50–76, doi:`10.48456/tr-244`.

[12] Dale Miller (1994): *Specification of the π-Calculus*. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/examples/pi-calculus/toc.html`.

[13] Dale Miller & Alwen Tiu (2005): *A Proof Theory for Generic Judgments*. *ACM Trans. Comput. Log.* 6(4), pp. 749–783, doi:`10.1145/1094622.1094628`.

[14] Alberto Momigliano, Brigitte Pientka & David Thibodeau (2019): *A Case-Study in Programming Coinductive Proofs: Howe's Method*. *Math. Struct. Comput. Sci.* 29(8), pp. 1309–1343, doi:`10.1017/S0960129518000415`.

[15] Joachim Parrow (2001): *An Introduction to the π-Calculus*. In Jan A. Bergstra, Alban Ponse & Scott A. Smolka, editors: *Handbook of Process Algebra*, North-Holland / Elsevier, pp. 479–543, doi:`10.1016/B978-044482830-9/50026-6`.

[16] Frank Pfenning (2000): *Structural Cut Elimination: I. Intuitionistic and Classical Logic*. *Inf. Comput.* 157(1-2), pp. 84–141, doi:`10.1006/INCO.1999.2832`.

[17] Brigitte Pientka & Andrew Cave (2015): *Inductive Beluga: Programming Proofs*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, *Lecture Notes in Computer Science* 9195, Springer, pp. 272–281, doi:`10.1007/978-3-319-21401-6_18`.

[18] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In Jürgen Giesl & Reiner Hähnle, editors: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, *Lecture Notes in Computer Science* 6173, Springer, pp. 15–21, doi:`10.1007/978-3-642-14203-1_2`.

[19] Davide Sangiorgi & David Walker (2001): *The π-Calculus - a Theory of Mobile Processes*. Cambridge University Press, doi:`10.2178/bsl/1182353926`.

[20] Chuta Sano, Ryan Kavanagh & Brigitte Pientka (2023): *Mechanizing Session-Types Using a Structural View: Enforcing Linearity without Linearity*. *Proc. ACM Program. Lang.* 7(OOPSLA), pp. 235:374–235:399, doi:`10.1145/3622810`.

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

A Appendix: Late vs Early Transitions

An alternative to the late semantics presented in the paper is the *early* semantics. As its name suggests, it is characterized by the fact that substitutions of names received in interaction are performed as soon as possible, namely during the execution of the S-IN rule. The syntax of actions is the same as in the late semantics; however, the name y occurring in an input action $x(y)$ is considered to be free. As for transitions, now denoted as $\xrightarrow{(-)}$, the rules S-IN, the two S-COM rules and the two S-CLOSE are replaced by those in Fig. 7 ("right" rules are omitted for brevity). Note how the S-IN rule now exhibits the substitution of the input name; conversely, in the S-COM rules, both of the names x and y occurring in the actions of the given transitions must coincide and no substitution takes place in the conclusion.

$$\text{S-IN}\quad \frac{}{x(z).P \xrightarrow{x(y)} P\{y/z\}} \qquad \text{S-COM-L}\quad \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{S-CLOSE-L}\quad \frac{P \xrightarrow{\bar{x}(z)} P' \quad Q \xrightarrow{x(z)} Q' \quad z \notin \mathsf{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}$$

Figure 7: Early transition semantics rules.

In order to prove the equivalence of the two semantics, we follow Parrow's approach in [15]: namely, our objective is to demonstrate that the two semantics allow to infer the same τ-transitions. Both directions are achieved by induction on the depth of the inference of the given transitions and rely on some additional lemmas, which state a correspondence between input/output transitions of the two semantics as well. The only non-trivial correspondence lies between input (in the late semantics) and free input (in the early semantics), which is defined as follows:

$$P \xrightarrow{x(y)} Q \text{ iff there are } Q' \text{ and } w \text{ such that } P \xrightarrow{x(w)} Q' \text{ and } Q = Q'\{y/w\}. \tag{2}$$

We begin the Beluga formalization by encoding the two semantics in the same environment (Fig. 8). The type `f_act` presents a new constructor `f_in` for free input actions in the early semantics; as for transitions, the constructors expressing identical rules in the two semantics are omitted for brevity. We observe that the `ebs_in` constructor representing bound input transitions in the early semantics cannot be eliminated, since bound input transitions are needed as premises in the rules introduced by the `efs_close` constructors. For consistency of notation, the types `fstep` and `bstep` for late transitions have been renamed as `late_fstep` and `late_bstep`.

The next ingredient for the formalization of the semantics equivalence is the definition of the type family `ex_latebs`, which encodes the existence of a late transition such as in the correspondence (2):

```
LF ex_latebs: proc → names → names → proc → type =
  | lbs: late_bstep P (b_in X) \w.(Q' w) → eqp Q (Q' Y) → ex_latebs P X Y Q
```

```
% Free Actions                              % Bound Actions
LF f_act: type =                            LF b_act: type =
  | f_in: names → names → f_act               | b_in: names → b_act
  | f_out: names → names → f_act              | b_out: names → b_act
  | f_tau: f_act

% Early Transition Relation
LF early_fstep: proc → f_act → proc → type =
  | efs_in: early_fstep (p_in X P) (f_in X Y) (P Y)
  | efs_com1: early_fstep P (f_out X Y) P’ → early_fstep Q (f_in X Y) Q’
     → early_fstep (P p_par Q) f_tau (P’ p_par Q’)
  | efs_com2: early_fstep P (f_in X Y) P’ → early_fstep Q (f_out X Y) Q’
     → early_fstep (P p_par Q) f_tau (P’ p_par Q’)
  | efs_close1: early_bstep P (b_out X) P’ → early_bstep Q (b_in X) Q’
     → early_fstep (P p_par Q) f_tau (p_res \z.((P’ z) p_par (Q’ z)))
  | efs_close2: early_bstep P (b_in X) P’ → early_bstep Q (b_out X) Q’
     → early_fstep (P p_par Q) f_tau (p_res \z.((P’ z) p_par (Q’ z)))
  ...
and early_bstep: proc → b_act → (names → proc) → type =
  | ebs_in: early_bstep (p_in X P) (b_in X) P
  ...
```

Figure 8: Encoding of actions and early transition.

We then list the signature of the functions `finp_earlytolate` and `finp_latetoearly`, encoding the correspondence between free input transitions in the early semantics and input transitions in the late semantics. The correspondence between the other types of actions is performed analogously with two functions for each case; since their formalization is straightforward, it is omitted.

```
rec finp_earlytolate: (g:ctx) [g ⊢ early_fstep P (f_in X Y) Q]
  → [g ⊢ ex_latebs P X Y Q] = ...

rec finp_latetoearly: (g:ctx) {Y:[g ⊢ names]} [g ⊢ ex_latebs P X Y Q]
  → [g ⊢ early_fstep P (f_in X Y) Q] = ...
```

In the second statement, the input name `Y` needs to be passed as an explicit argument, otherwise Beluga would not be able to reconstruct it during some further calls of this lemma. The proofs of both lemmas are straightforward inductions on the given derivation.

We can now state the signature of the functions `tau_earlytolate` and `tau_latetoearly`, which encode the equivalence of the two semantics:

```
rec tau_earlytolate: (g:ctx) [g ⊢ early_fstep P f_tau Q]
  → [g ⊢ late_fstep P f_tau Q] = ...

rec tau_latetoearly: (g:ctx) [g ⊢ late_fstep P f_tau Q]
  → [g ⊢ early_fstep P f_tau Q] = ...
```

The proof follows by induction on the given transition. In case the transition is obtained through a S-COM or S-CLOSE rule, we apply the previously defined lemmas in order to turn an early input/output transition into a corresponding late input/output transition and then conclude.

Binding Contexts as Partitionable Multisets in Abella

Terrance Gray Gopalan Nadathur
University of Minnesota, Minneapolis, MN 55455, USA
grayx501@umn.edu ngopalan@umn.edu

When reasoning about formal objects whose structures involve binding, it is often necessary to analyze expressions relative to a context that associates types, values, and other related attributes with variables that appear free in the expressions. We refer to such associations as binding contexts. Reasoning tasks also require properties such as the shape and uniqueness of associations concerning binding contexts to be made explicit. The Abella proof assistant, which supports a higher-order treatment of syntactic constructs, provides a simple and elegant way to describe such contexts from which their properties can be extracted. This mechanism is based at the outset on viewing binding contexts as ordered sequences of associations. However, when dealing with object systems that embody notions of linearity, it becomes necessary to treat binding contexts more generally as partitionable multisets. We show how to adapt the original Abella encoding to encompass such a generalization. The key idea in this adaptation is to base the definition of a binding context on a mapping to an underlying ordered sequence of associations. We further show that properties that hold with the ordered sequence view can be lifted to the generalized definition of binding contexts and that this lifting can, in fact, be automated. These ideas find use in the extension currently under development of the two-level logic approach of Abella to a setting where linear logic is used as the specification logic.

1 Introduction

It is often necessary to develop specifications and to reason about formal objects whose structures incorporate some notion of binding. Examples of such objects include formulas, types, proofs, and programs. A recursive analysis of such objects requires the examination of their subparts in which there may be occurrences of free variables. This analysis is usually parameterized by an association of some kind, such as a type, a value, or a property, with each of these variables. This paper concerns support for such associations, which we refer to as *binding contexts*, in reasoning tasks.

The focus of our work is the treatment of binding contexts relative to a particular reasoning system, the Abella proof assistant [1]. A defining characteristic of Abella is that it provides intrinsic support for the higher-order approach to abstract syntax. At the representation level, this support derives from the use of the terms of the simply typed lambda calculus as the means for encoding objects. At the level of the logic, Abella incorporates the special generic quantifier ∇, pronounced as *nabla*, to move binding into the meta-level and the associated nominal constants to encode free variables. Further, it allows properties of binding contexts to be made explicit through the definition of *context predicates* and *context relations* and thereby to be used in proofs.

While Abella provides rich support for working with binding contexts, one aspect that it does not treat adequately with respect to these contexts is *linearity*. This requirement arises, for instance, when bound variables take on the connotation of resources that must be used exactly once within the overall syntactic object. To provide support for this viewpoint, it becomes necessary to encode binding contexts as partitionable entities. We show in this paper how this capability can be built into the Abella system. The key idea underlying our proposal is to view binding contexts as multisets that are *permutation invariant* and that can be constructed from two simpler multisets through multiset union. Thus, if $\sim$ is an infix operator representing the permutation relation between multisets and ++ is an infix multiset union operator, the expression $G \sim (G_1 \text{ ++ } G_2)$ encodes the fact that G_1 and G_2 partition the multiset G.[1]

Unfortunately, the ability to partition a multiset is not by itself sufficient for the usual reasoning tasks. When G_1 and G_2 have been determined to be partitions of a binding context G, we need also to know that each of them independently satisfies the properties needed to be the required kind of binding context. A related issue is that we must be able to define what it means to be a binding context in a particular reasoning task when these contexts may be constructed using multiset unions. A major part of our work here is to outline a systematic method for realizing these requirements. Our proposal in a nutshell is to identify what it means to be a binding context through the definition of a context predicate or relation while initially viewing it as an ordered sequence or list of associations. This definition can then be lifted to arbitrary multisets through the permutation relation. Distributivity of the property over multiset union then factors through the same permutation relation. An auxiliary consequence of what we show is the fact that this scheme is to a substantial extent automatable.

The rest of the paper is structured as follows. In the next section, we identify in more detail the idea of binding contexts and describe their realization in Abella when they are represented in a list-based form; we assume in this presentation, and, indeed, the rest of the paper, a familiarity with the Abella system. Section 3 then identifies the need for linearity with respect to binding contexts in specifications and the additional constructors and definitions that suffice to realize it. Of course, it still remains to be shown how to make things work at the reasoning level. Section 4 explains how context predicates can be defined when binding contexts may be constructed using the multiset union operator and how the properties of such contexts can be extracted into lemmas even in this situation. Section 5 shows that these ideas extend also to the setting of context relations, which embody the simultaneous description of multiple correlated contexts. Section 6 discusses a schematic presentation of context predicates and context relations and explains how the lifting procedure may be automated, describing some tactics for implementing the corresponding algorithms. Section 7 discusses related work and Section 8 closes out the paper by sketching the use of our work in the particular application domain that has motivated it.

2 Binding Contexts and their Conventional Treatment in Abella

Towards understanding the nature of binding contexts and the kinds of properties that must be associated with them in reasoning tasks, we may consider the example of type assignment for the simply typed lambda calculus. We limit the expressions in the calculus to those constructed from variables using the operations of application, written as $(e_1\ e_2)$, and abstraction, written as $\lambda x : \tau . e$. The rules for associating types with expressions in this calculus are then the following:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1\ e_2) : \tau} \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x : \tau' . e : \tau' \to \tau}\ x \text{ new to } \Gamma$$

Type assignment for closed terms is ultimately a relation between a term and a type. However, a recursive definition of this relation requires us to consider type assignments to open terms under the assumption that the free variables in the term have designated types. Thus, the relation must be formalized as a ternary one, written as $\Gamma \vdash e : \tau$. In this example, Γ constitutes the *typing* or *binding context*. The structure of Γ is governed by the rule for assigning types to abstractions. Based on this rule, we can observe some properties that are implicitly associated with Γ: it assigns types only to variables and there

[1]Partitioning of multisets can be described without the use of a permutation relation; this is mainly a convenient way to do it if we have the relation.

is at most one assignment to any variable. While Γ is built one element at a time and seems to have the structure of an ordered sequence, we are free to think of it a multiset or even a set. Note finally that a *closed-world assumption* applies to the rules: a term may be assigned a type only by virtue of these rules.

Relational presentations of the kind above have a natural translation into Abella specifications. To present this in the particular example under consideration, we must first describe a representation for the types and terms of the simply typed lambda calculus. We will use the Abella types `ty` and `tm` for encodings of expressions in these two categories. We will also use the constant `arrow` of type $\texttt{ty} \rightarrow \texttt{ty} \rightarrow \texttt{ty}$ to represent the function type constructor, and the constants `app` and `abs`, respectively of type $\texttt{tm} \rightarrow \texttt{tm} \rightarrow \texttt{tm}$ and $\texttt{ty} \rightarrow (\texttt{tm} \rightarrow \texttt{tm}) \rightarrow \texttt{tm}$, to represent application and abstraction in the object language. Note the use of a higher-order abstract syntax representation here; for example, the term $\lambda x : \tau_1 \rightarrow \tau_2 . \lambda y : \tau_1 . x\, y$ in the object language would be encoded by the Abella term $(\texttt{abs}\ (\texttt{arrow}\ \overline{\tau_1}\ \overline{\tau_2})\ (\texttt{x}\backslash\texttt{abs}\ \overline{\tau_1}\ (\texttt{y}\backslash\texttt{app}\ \texttt{x}\ \texttt{y})))$, where $\overline{\tau_1}$ and $\overline{\tau_2}$ are the representations of the types τ_1 and τ_2, respectively.[2] In this context, the content of the type assignment rules is captured by the following Abella declarations that ultimately provide an inductive definition for the ternary type assignment relation `type_of`:

```
Kind ty_assoc type.
Type ty_of tm -> ty -> ty_assoc.

Define member : A -> list A -> prop by
member X (X :: L) ;
member X (Y :: L) := member X L.

Define type_of : list ty_assoc -> tm -> ty -> prop by
type_of G X T := member (ty_of X T) G ;
type_of G (app M N) T := exists T', type_of G M (arrow T' T) /\ type_of G N T';
type_of G (abs T E) (arrow T T') := nabla x, type_of (ty_of x T :: G) (E x) T'.
```

Focusing on the first argument of the `type_of` relation, we see that it has the kinds of properties that we observed of binding contexts that arise in type assignment and that it represents. While it has the structure of an Abella list, it can equally be viewed as a multiset or a set; the use of the `member` predicate relative to it is compatible with all these views. In the intended use of the predicate, this collection is constructed one item at a time via the clause for assigning types to abstractions. The use of the `nabla` quantifier also ensures that the associations it provides pertain only to nominal constants—which represent the free variables in object language terms in the logic—and that there is at most one such association in it for any such constant.

The properties we have described for binding contexts in type assignment can be important to reasoning tasks. They are key, for example, to showing the uniqueness of type assignment to any typeable term: the proof of this fact hinges on the observations that the typing context does not assign types to applications or abstractions and that the assignments to variables are unique. However, in a formalized setting, it is not enough that these properties hold. It must also be made explicit that they do. This can be done in Abella by what are commonly referred to as *context definitions*. In the example in question, the following definition serves this purpose:

```
Define ty_ctx : list ty_assoc -> prop by
ty_ctx nil ;
nabla x, ty_ctx (ty_of x T :: G) := ty_ctx G.
```

The `nabla` quantifier in the head of the second clause is to be understood as follows: it must be instantiated by a nominal constant in generating an instance of the clause and the substitutions for `T` and `G` that

[2]We recall that abstraction is written as the infix operator $\backslash$ in Abella, *i.e.*, the expression $\lambda x.F$ is denoted by $x\backslash F$.

generate the instance must not contain that constant. The formula `(ty_ctx G)` now serves to assert that `G` is a typing context with the necessary properties.

It is useful to drill down a little on the last statement. One of the requirements of a typing context is that it associates types only with nominal constants, *i.e.*, the representatives of variables in terms. The following definition identifies the predicate `name` as a recognizer for such constants:

```
Define name : A -> prop by
nabla n, name n.
```

Using it, we can capture the desired property in the following Abella theorem about the "shape" of the entities comprising a typing context:

```
Theorem ty_ctx_mem : forall L X,
ty_ctx L -> member X L -> exists n T, name n /\ X = ty_of n T.
```

Another property that is important is the uniqueness of type association. This can be rendered into the following Abella theorem:

```
Theorem ty_ctx_uniq : forall L X T1 T2,
ty_ctx L -> member (ty_of X T1) L -> member (ty_of X T2) L -> T1 = T2
```

These theorems can both be proved by induction on the definition of `ty_ctx`. Once we have these properties, it is an easy matter to prove the following theorem:

```
Theorem ty_uniq : forall L X T1 T2,
ty_ctx L -> type_of L X T1 -> type_of L X T2 -> T1 = T2.
```

The uniqueness of type assignment for closed terms follows easily from this theorem.

Although our discussion in this section has been oriented around an example, the underlying concepts are quite general. Binding contexts manifest themselves commonly in specifications about syntactic constructs that incorporate binding notions. Context definitions make explicit the structure of such contexts when a higher-order abstract syntax representation is used for the constructs. The properties that we must extract from such definitions to support other reasoning tasks take two forms. First, there are *membership lemmas* like `ty_ctx_mem` that constrain the shape of the elements of the context. Second, there are *uniqueness lemmas* like `ty_ctx_uniq` that assert the uniqueness of bindings. We have seen how context definitions can be played out and the associated lemmas can be proved when contexts are limited to being constructed and analyzed one item at a time. We will next show why this view of the structure of contexts needs to be generalized and then demonstrate how such a generalization may be accommodated.

3 Partitionable Binding Contexts and Multiset Union

The treatment of binding contexts that we have described in the previous section does not support the aspect of *linearity* that is relevant to some applications. An example of such an application is provided by the simply typed *linear* lambda calculus. To be well-formed, terms in this calculus must have the additional property that every bound variable is used exactly once. Under this restriction, the term $\lambda x:\tau_1 \to \tau_2.\lambda y:\tau_1.x\ y$ is well-formed but $\lambda x:\tau_1 \to \tau_1 \to \tau_2.\lambda y:\tau_1.x\ y\ y$ and $\lambda x:\tau_1.\lambda y:\tau_2.y$ are not.

If we are to build a linearity check into the type assignment process, the rule for assigning a type to an application must incorporate the idea of *partitioning* a binding context. To support this possibility, we propose allowing binding contexts to be constructed using one other operation, that of *multiset union*. More specifically, we shall continue to use the type `(list A)` to represent such contexts but now we will interpret this type as that of multisets of elements of type `A` rather than that of ordered sequences. We will continue to use the constant `nil` and the infix operator `::` as constructors of this type, but now interpret the latter as a means for adding an element to a multiset. Additionally, the type now has one other constructor, the infix operator `++` of type `(list A) -> (list A) -> (list A)`. An expression of the form `G1 ++ G2` is intended to represent a multiset whose elements comprise those of `G1` and `G2`.

The `member` predicate must be adapted to this changed syntax. Its definition becomes the following:

```
Define member : A -> list A -> prop by
member X (X :: G) ;
member X (Y :: G) := member X G ;
member X (G1 ++ G2) := member X G1 \/ member X G2.
```

To accommodate linearity, we will need a counterpart to this predicate that represents the *selection* of a member from a multiset that simultaneously yields a smaller multiset. Towards this end, we will use a predicate called `select` that has the definition below.

```
Define select : A -> list A -> list A -> prop by
select X (X :: G) G ;
select X (Y :: G) (Y :: G') := select X G G' ;
select X (G1 ++ G2) (G1' ++ G2) := select X G1 G1' ;
select X (G1 ++ G2) (G1 ++ G2') := select X G2 G2'.
```

We want to be able to treat binding contexts that have the same elements as equivalent, regardless of how they are constructed. Towards this end, we introduce a permutation predicate `perm` for multisets. It is useful to define this, at a high-level, by recursion on the number of elements in each context, defining an auxiliary `no_elems` predicate that holds of a context that is empty. The relevant definitions follow:

```
Define no_elems : list A -> prop by
no_elems nil ;
no_elems (G1 ++ G2) := no_elems G1 /\ no_elems G2.

Define perm : list A -> list A -> prop by
perm G1 G2 := no_elems G1 /\ no_elems G2 ;
perm G1 G2 := exists X G1' G2',
    select X G1 G1' /\ select X G2 G2' /\ perm G1' G2'.
```

The auxiliary definition of `no_elems` also gives us a means of ensuring that all bound variables are used in a specification of a linear system. We can assert that the context satisfies this predicate after we have analyzed the entirety of a term to ensure no variables were introduced by an abstraction but left unused.

We introduce a convenient notational shorthand for the predicate `perm`: we shall write `G1 ~ G2` to represent `(perm G1 G2)`. The `perm` predicate and the `++` operator together give us a means for encoding a partition of *n* multisets into *m* multisets, which we can write as `G1 ++ ... ++ Gn ~ D1 ++ ... ++ Dm`. Note that the permutation component of this expression allows elements to be distributed in any order between the multisets on the other side—so that partitioning does not depend on the elements to partition having been ordered correctly ahead of time.

The components that we have described in this section provide us the necessary means for writing linear specifications. Let us bring this out through the definition of a typing relation for the linear lambda calculus that only assigns types to valid linear lambda terms. The definition of this relation, which we denote by the predicate `ltype_of`, is as follows:

```
Define ltype_of : list ty_assoc -> tm -> ty -> prop by
ltype_of G X T := exists G', select (ty_of X T) G G' /\ no_elems G' ;
ltype_of G (app M N) T := exists T' G1 G2,
    G ~ G1 ++ G2 /\ ltype_of G1 M (arrow T' T) /\ ltype_of G2 N T' ;
ltype_of G (abs T E) (arrow T T') :=
    nabla x, ltype_of (ty_of x T :: G) (E x) T'.
```

It is worth mentioning the differences between the definition of this predicate and that of `type_of` in Section 2 to understand how the linearity constraints are enforced. The use of `select` in the first clause ensures that a particular association for a bound variable cannot be used more than once, and the `no_elems` assertion ensures that every association must have been used. The formula `G ~ G1 ++ G2` realizes a partitioning of the context `G` and thereby ensures that the type assignment to a particular bound variable must be used for typing exactly one of the two subcomponents of an application. The structure of the last clause, which is unchanged from the definition of `type_of`, still ensures that the binding context has associations only for variables and that an association for any variable is unique. However, we must reason now about the effect of partitioning to see that these properties actually hold.

4 Reasoning About Binding Contexts in the Generalized Form

In proving properties of relations whose definitions involve binding contexts in the extended form, we will once again need to establish membership and uniqueness lemmas pertaining to the binding contexts. For example, in showing the uniqueness of type assignment as expressed by the `ltype_of` relation, we will need the counterparts of the `ty_ctx_mem` and `ty_ctx_uniq` lemmas for contexts in the new form. We show here how this can be done. The difficulty that must be addressed is that the introduction of multiset union breaks the view of contexts being constructed one element at a time. The solution that we propose is based on flattening a context with arbitrary structure into one that is constructed in the conventional way. We present the idea relative to an example but its generality should be clear from the discussion.

4.1 Lifting Context Definitions to the Generalized Form

In Section 2, we defined the predicate `ty_ctx` to make explicit the logical structure of typing contexts for the simply typed lambda calculus. This definition must now be extended to cover contexts that are constructed using the multiset union operator. We might think of doing this by adding a third clause akin to the following to the definition of `ty_ctx`:

```
ty_ctx (G1 ++ G2) := ty_ctx G1 /\ ty_ctx G2.
```

Unfortunately, this idea does not work: such a clause would break the property of the binding context that associations for a particular name are unique, as nothing in it enforces that the names associated within `G1` and `G2` are distinct from each other, even if they are distinct within each individual context.

The insight that underlies the solution that we propose is that the properties in question should not depend on the order in which the associations in a binding context are listed or the way in which they are distributed over a multiset, only on what those associations are. Thus, it would suffice if we could restructure the multiset construction and rearrange its elements so as to produce a form that satisfies the `ty_ctx` predicate that we had defined earlier. Further, the kind of projection that is necessary here can be accomplished through the `perm` predicate that relates two multisets with possibly different structures so long as they have the same elements. Thus, in the present example, the context definition might be given by the `ty_ctx'` predicate that is defined as follows:

```
Define ty_ctx' : list ty_assoc -> prop by
ty_ctx' G := exists L, G ~ L /\ ty_ctx L.
```

This predicate applies to typing contexts presented in the generalized form since the `perm` predicate is defined over multisets that could include the `++` operator as well. Note, however, the definition of ty_ctx is dependent on an "ordered sequence" view, *i.e.*, the given context must be projected onto one in this form to assess whether it possesses the necessary properties.

4.2 Proving Membership and Uniqueness Lemmas

The new definition must still enable us to prove lemmas about the shape of the associations in the binding context as well as their uniqueness. These lemmas are the following in the present situation:

```
Theorem ty_ctx_mem' : forall G X,
ty_ctx' G -> member X G -> exists n T, name n /\ X = ty_of n T.

Theorem ty_ctx_uniq' : forall G X T1 T2,
ty_ctx' G -> member (ty_of X T1) G -> member (ty_of X T2) G -> T1 = T2.
```

The proofs of these lemmas also embody a process of "lifting" of properties established based on the ordered sequence view through the projection. First observe that the theorems ty_ctx_mem and ty_ctx_uniq continue to hold despite the change in the definition of the member predicate. Specifically, the definition of this predicate reduces to the original one when the multiset argument is limited to having a list-like structure, a structure that is forced by the ty_ctx predicate. But now we can also prove the following (generic) lemma that states that membership in a multiset is preserved through a permutation:

```
Theorem mem_replace : forall X G G', member X G -> G ~ G' -> member X G'.
```

Since contexts described by ty_ctx' are only a permutation away from those described by ty_ctx, this is sufficient to lift the theorems ty_ctx_mem and ty_ctx_uniq into ty_ctx_mem' and ty_ctx_uniq'. We need only apply the lemma to replace the member predicates in one theorem with those in the other.

4.3 Distributivity of Context Properties over Multiset Unions

The multiset union constructor was introduced originally to facilitate a partitioning of contexts. For this to be useful for the intended purpose, the facet of being a context of the desired kind must distribute over such partitioning. In our example, this translates into the desire that the following theorem be provable:

```
Theorem ty_ctx_distr : forall G G1 G2,
ty_ctx' G -> G ~ G1 ++ G2 -> ty_ctx' G1 /\ ty_ctx' G2.
```

Once again, we can prove the desired property by establishing a corresponding property for list-like contexts, and then lifting that property to contexts that may include the multiset union operator in their formation. One approach to stating the first property involves defining a predicate that encodes an ordered partition relation between three lists:

```
Define partition : list A -> list A -> list A -> prop by
partition nil nil nil ;
partition (X :: L) (X :: L1) L2 := partition L L1 L2 ;
partition (X :: L) L1 (X :: L2) := partition L L1 L2.
```

By exploiting the fact that the relative order of elements in a list L is preserved within the related lists L1 and L2, we can easily prove the following theorem that states that the property of being a typing context is preserved by such partitions:

```
Theorem ty_ctx_distr_part : forall L L1 L2,
ty_ctx L -> partition L L1 L2 -> ty_ctx L1 /\ ty_ctx L2.
```

We can lift this theorem to ty_ctx' and perm-style partitions by relating partition and perm. Towards this end, we first define a predicate that captures the property that a context has a list-like structure:

```
Define is_list : list A -> prop by
is_list nil ;
is_list (X :: L) := is_list L.
```

The following lemma then provides the necessary bridge:

```
Theorem perm_to_part : forall L G1 G2,
is_list L -> L ~ G1 ++ G2 -> exists L1 L2,
    G1 ~ L1 /\ G2 ~ L2 /\ partition L L1 L2.
```

Essentially, the lemma says that a partition of the elements in a list into two arbitrary contexts can be flattened into a `partition` between lists of the same elements. It can be proved by inverting the permutation and using the elements extracted from the multisets `G1` and `G2` to construct the lists `L1` and `L2`. The proof relies critically on the following lemma which allows elements in a multiset `G` that is related by `perm` to `L` to be extracted one at a time in the order they appear in `L`:

```
Theorem sel_replace : forall X G1 G1' G2,
G1 ~ G2 -> select X G1 G1' -> exists G2', G1' ~ G2' /\ select X G2 G2'.
```

Note that this lemma is, in fact, a counterpart to `mem_replace` for `select`.

At this stage, we have all the ingredients in place to prove the `ty_ctx_distr` theorem. Given any context `G` for which `ty_ctx'` holds, there must, by definition, be an `L` such that $\mathtt{G} \sim \mathtt{L}$ and `ty_ctx L`. Since $\mathtt{G} \sim \mathtt{G1} \mathbin{++} \mathtt{G2}$ holds, by properties of `perm`, it must then be the case that $\mathtt{L} \sim \mathtt{G1} \mathbin{++} \mathtt{G2}$ holds. Now, using theorems `perm_to_part` and `ty_ctx_distr_part`, we can conclude that there are contexts `L1` and `L2` such that $\mathtt{G1} \sim \mathtt{L1}$, $\mathtt{G2} \sim \mathtt{L2}$, `ty_ctx L1`, and `ty_ctx L2` hold; we will need to show that `is_list L` holds in order to invoke theorem `perm_to_part`, but this follows easily from the fact that `ty_ctx L` holds. Using the definition of `ty_ctx'`, it is then immediate that `ty_ctx' G1` and `ty_ctx' G2` must hold.

5 Generalization to Context Relations

Typical meta-theoretic reasoning tasks require us to relate different kinds of analyses over the same object-language expression. When the expression embodies binding constructs, these analyses would be parameterized by binding contexts. In the Abella setting, the shape of each of these contexts must be characterized by a definition. When different analyses are involved in the property to be proved, there will generally be an additional requirement: the content of the different binding contexts parameterizing the analyses must be coordinated in an appropriate way. *Context relations* constitute the canonical mechanism in Abella for phrasing context definitions to suit the reasoning needs in such situations. The generalized multiset structure is needed for dealing with linearity in this situation as well and the methods for supporting it bear a remarkable resemblance to those when only one binding context is involved. We bring this observation out in this section through an example.

The example we consider is that of relating typing judgments across a translation. The target language for the translation shall be the linear variant of the simply typed lambda calculus that we introduced in Section 3. The source language, which we will call mini linear ML, shall be similar, except that it shall include an additional `let` construct. To represent such expressions, we introduce the constant `let` that has the type $\mathtt{ty} \to \mathtt{tm} \to (\mathtt{tm} \to \mathtt{tm}) \to \mathtt{tm}$. Observe that higher-order abstract syntax is used again in the encoding of `let` expressions: the expression $\mathtt{let}\ \mathtt{X} : \tau = \mathtt{V}\ \mathtt{in}\ \mathtt{F}$ is represented by $(\mathtt{let}\ \overline{\tau}\ \overline{\mathtt{V}}\ (\mathtt{X}\backslash\overline{\mathtt{F}}))$, where $\overline{\tau}$, $\overline{\mathtt{V}}$, and $\overline{\mathtt{F}}$ are the representations of τ, `V`, and `F`, respectively. The typing relation for the source language is now given by the following definition:

```
Define mltype_of : list ty_assoc -> tm -> ty -> prop by
mltype_of G X T := exists G', select (ty_of X T) G G' /\ no_elems G' ;
mltype_of G (app M N) T := exists T' G1 G2,
    G ~ G1 ++ G2 /\ mltype_of G1 M (arrow T' T) /\ mltype_of G2 N T' ;
mltype_of G (let T' V E) T := exists G1 G2,
    G ~ G1 ++ G2 /\ mltype_of G1 V T' /\
    nabla x, mltype_of (ty_of x T' :: G2) (E x) T ;
mltype_of G (abs T E) (arrow T T') :=
    nabla x, mltype_of (ty_of x T :: G) (E x) T'.
```

The translation of mini linear ML expressions to the linear lambda calculus essentially replaces `let` expressions by applications. It is formalized by the following clauses for the `ltrans` predicate:

```
Kind var_assoc type.
Type trans_to tm -> tm -> var_assoc.

Define ltrans : list var_assoc -> tm -> tm -> prop by
ltrans G X Y := exists G', select (trans_to X Y) G G' /\ no_elems G' ;
ltrans G (app M N) (app M' N') := exists G1 G2,
    G ~ G1 ++ G2 /\ ltrans G1 M M' /\ ltrans G2 N N' ;
ltrans G (let T V E) (app (abs T E') V') := exists G1 G2,
    G ~ G1 ++ G2 /\ ltrans G1 V V' /\
    nabla x y, ltrans (trans_to x y :: G2) (E x) (E' y) ;
ltrans G (abs T E) (abs T E') :=
    nabla x y, ltrans (trans_to x y :: G) (E x) (E' y).
```

We would like to prove that this translation preserves the types of expressions. Since translation and typing are defined by recursion over the structures of expressions and will, in general, encounter open terms, the theorem to be proved must have a form such as the following:

```
Theorem ltrans_pres_ty'' : forall E E' T T' G G' G'',
mltype_of G E T -> ltrans G' E E' -> ltype_of G'' E' T' -> T = T'.
```

However, this formula cannot be proved as stated. The contexts that arise at intermediate points in translation and type assignment have structures and relationships that must be made explicit in the formulation to yield a provable statement. Only names can be associated with other data in these contexts, and these associations must be unique. Further, we will need to relate the types of free variables in a term and its translation to be able to show that the two have the same type.

The canonical way to make the relationship in the content of multiple contexts explicit in Abella is by defining an appropriate context relation as a predicate. Let `trans_rel` be a predicate that encodes the relevant relationship between the three contexts in consideration here. The theorem to be actually proved then becomes the following:

```
Theorem ltrans_pres_ty : forall E E' T T' G G' G'',
trans_rel G G' G'' -> mltype_of G E T -> ltrans G' E E'
                   -> ltype_of G'' E' T' -> T = T'.
```

In proving theorems such as these, there are, once again, certain lemmas about members of the contexts that we must be able to extract from the relevant context relations. In this particular example, we would need to be able to prove the following lemmas that express a uniqueness property and a membership *coordination* property between the related contexts:

```
Theorem trans_rel_uniq : forall G1 G2 G3 X Y Y',
trans_rel G1 G2 G3 -> member (trans_to X Y) G2
                   -> member (trans_to X Y') G2 -> Y = Y'.
```

```
Theorem trans_rel_mem : forall G1 G2 G3 E,
trans_rel G1 G2 G3 -> member E G2 -> exists X Y T,
    E = trans_to X Y /\ name X /\ name Y /\
    member (ty_of X T) G1 /\ member (ty_of Y T) G3.
```

These properties are stated from the perspective of the second of the three contexts. There would be four more similar properties when matters are viewed from either of the other two contexts.

The issue to be addressed, then, is how the context relation should be defined to allow for the extraction of such properties. There is a standard recipe for realizing the described objectives when contexts are limited to a list-like structure. In this example, we may define a list-oriented version of trans_rel following the conventional strategy as follows:

```
Define trans_rel_list : list ty_assoc -> list var_assoc
                                     -> list ty_assoc -> prop by
trans_rel_list nil nil nil ;
nabla x y, trans_rel_list (ty_of x T :: L1)
                          (trans_to x y :: L2)
                          (ty_of y T :: L3)   := trans_rel_list L1 L2 L3.
```

The uniqueness of binding property relativized to trans_rel_list has a proof similar to the one discussed for the typing context in Section 2. The second property follows easily from the fact that the definition is based on a coordinated recursion over the three contexts that in fact ensures that they each contain the right kinds of members.

What we want, though, is a definition of trans_rel that applies to contexts whose structure includes the multiset union constructor. Using the ideas discussed in Section 4, we can accomplish this once again by lifting the list-based definition up to contexts with a more general structure through permutations. The following definition of the relation realizes the desired result:

```
Define trans_rel : list ty_assoc -> list var_assoc -> list ty_assoc -> prop by
trans_rel G1 G2 G3 := exists L1 L2 L3,
    G1 ~ L1 /\ G2 ~ L2 /\ G3 ~ L3 /\ trans_rel_list L1 L2 L3.
```

This definition still requires the associations in each context to correspond with associations in the other contexts, but now the corresponding associations need not be in the same position in each context. Still, since the associations are clearly linked in the list-based context relation, we will be able to lift the necessary membership and uniqueness lemmas from the latter context relation. Indeed, the proof of trans_rel_mem proceeds nearly as in the unary case: we can make use of mem_replace to ensure that E is an element of the underlying translation context, and then make use of this lemma again to ensure that ty_of X T and ty_of Y T are also members of the original typing contexts. The uniqueness lemma can be proved from the corresponding lemma for the underlying context in a similar way, and the lifting process is even simpler: since no conclusions need be drawn about the other contexts, mem_replace is only needed in one direction.

The first clause in the definitions of mltype_of, ltrans, and ltype_of actually *selects* an association from the relevant context rather than simply checking membership. Consequently, we would often need a stronger version of the trans_rel_mem property that is based on the select relation and that additionally asserts that the remaining contexts continue to be in the trans_rel relation:

```
Theorem trans_rel_sel : forall G1 G2 G2' G3 E,
trans_rel G1 G2 G3 -> select E G2 G2' -> exists X Y T G1' G3',
    E = trans_to X Y /\ name X /\ name Y /\ select (ty_of X T) G1 G1' /\
    select (ty_of Y T) G3 G3' /\ trans_rel G1' G2' G3'.
```

The new requirement here is that we must show that `trans_rel G1' G2' G3'` holds for the three new contexts `G1'`, `G2'`, and `G3'` that result from selection from `G1`, `G2`, and `G3`. Most of this lemma can be proved without significant digression from the proof sketched for `trans_rel_mem`. For the lifting step, where we convert the `selects` on multisets to `selects` on lists and vice versa, we can just use `sel_replace` instead of `mem_replace`. This also yields the necessary permutations for concluding `trans_rel G1' G2' G3'`: if `trans_rel G1 G2 G3` holds because `trans_rel_list L1 L2 L3` does, and selecting from `L1`, `L2`, and `L3` yields `L1'`, `L2'`, and `L3'`, then `G2'` ~ `L2'`, `G1'` ~ `L1'`, and `G3'` ~ `L3'` must hold. In the overall scheme, we can think of just proving `trans_rel_sel`. We can get a proof of `trans_rel_mem` from this if it is desired by using the following easily proved theorem that asserts that selecting from a context implies membership in that context:

```
Theorem sel_implies_mem : forall X G G', select X G G' -> member X G.
```

Finally, when multiset union is permitted in the construction of contexts, we will need lemmas that verify the distributivity of context relations over partitions. For example, the definitions of `mltype_of`, `ltrans`, and `ltype_of` will force us to prove lemmas such as the following that are analogous to the distributivity property for `ty_ctx_distr` in the preceding section:[3]

```
Theorem trans_rel_distr : forall G1 G1' G1'' G2 G3,
trans_rel G1 G2 G3 -> G1 ~ G1' ++ G1'' -> exists G2' G2'' G3' G3'',
    G2 ~ G2' ++ G2'' /\ G3 ~ G3' ++ G3'' /\
    trans_rel G1' G2' G3' /\ trans_rel G1'' G2'' G3''.
```

To prove such a distributivity lemma, we can first state and prove an analogous lemma for the related contexts in list form and then lift it to contexts with a more general structure. For this, we may reuse the definition of `partition` and many of its properties, stating the lemma to prove in the case under consideration as

```
Theorem trans_rel_list_distr : forall L1 L1' L1'' L2 L3,
trans_rel_list L1 L2 L3 -> partition L1 L1' L1'' -> exists L2' L2'' L3' L3'',
    trans_rel_list L1' L2' L3' /\ trans_rel_list L1'' L2'' L3'' /\
    partition L2 L2' L2'' /\ partition L3 L3' L3''.
```

The ordered nature of `partition` again is critical to the proof; since the related contexts are also ordered, we can construct new `partitions` using the corresponding elements as in `partition L1 L1' L1''` in the same places for the other contexts. Then, to lift this lemma to arbitrary multiset partitions, we can exploit `perm_to_part` in a first step to transform `G1` ~ `G1'` ++ `G1''` into `partition L1 L1' L1''`, where `G1` ~ `L1`, `G1'` ~ `L1'`, and `G1''` ~ `L1''` hold. After applying `trans_rel_list_distr`, we can make use of a kind of inverse of the `perm_to_part` lemma to convert `partitions` back into permutations:

```
Theorem part_to_perm : forall L L1 L2, partition L L1 L2 -> L ~ L1 ++ L2.
```

Since partitioning a list involves a restricted form of selection, the structure of the proof of this lemma should be easy to visualize. To complete the proof of `trans_rel_distr`, we can then note that the contexts it asserts the existence of can be the same as those asserted by `trans_rel_list_distr`, and that for any `G`, `L`, `L'`, and `L''`, `G` ~ `L'` ++ `L''` follows from `G` ~ `L` and `L` ~ `L'` ++ `L''` by properties of `perm`. Thus, we can conclude that `trans_rel G1' G2' G3'` and `trans_rel G1'' G2'' G3''` hold by definition; we will need to show that each list is a permutation of itself for this, but this follows easily from the fact that each is a list.

[3]Note that these lemmas do not let us specify the partition used for multiple contexts at once; they assert only the existence of some partitions that work. However, they suffice for many reasoning examples or can be worked around by exploiting properties of other predicates—such as the typing and translation relations here.

6 Schematic Context Specifications and Automated Proofs

The idea of defining a multiset-based context specification—via a context predicate or context relation—by lifting from a list-based one has a general applicability and can be deployed in other developments as well. We present in this section a general form for such specifications for which we can write *schematic* proofs of several distributivity lemmas and of a lifting procedure for a reasonably large class of lemmas based on the `member` predicate which includes our membership and uniqueness lemmas. This works since the distributivity lemmas and lifting procedure depend only on the general structure of the context specifications defined and not on the particular elements of the context(s). Hence, a user need only state and prove the `member` lemmas that require explicit reference to the elements of the binding context(s) for an underlying specification and can leave the rest of the work to an automated procedure.

Let us begin by introducing a command that might be used to succinctly generate a pair of context specifications—one based on lists and the other based on multisets. The syntax of this command should take the following general form, with each `FORMULA` referring to an expression of type `prop`, each `TERM` referring to a term of some other type, each `VAR` referring to a variable identifier, and `CTX-NAME` referring to the name of the context specification to be defined:

```
Context CTX-NAME with elems as
    nabla VAR11 ... VAR1k_1 (TERM11 _|_ ... _|_ TERM1n -| FORMULA1) \/ ... \/
    nabla VARm1 ... VARmk_m (TERMm1 _|_ ... _|_ TERMmn -| FORMULAm).
```

The use of the formula is to provide an additional means for encoding the relationship between elements of each of the related contexts in a context relation. Also note that there may be zero variables, in which case the `nabla` may be omitted, and we can omit the formula if it is `true`. However, there must be at least one clause and at least one term denoting some element of a context. As examples of the intended usage of this command, we present the commands that, following the process we describe next, would generate the definitions of `ty_ctx'` and `trans_rel` from Sections 4 and 5:

```
Context ty_ctx' with elems as nabla x (ty_of x T).
Context trans_rel with elems as
    nabla x y (ty_of x T _|_ trans_to x y _|_ ty_of y T').
```

Our command schema is meant to define a pair of context specifications of the following forms, where each `TYPE` is an Abella type inferred from the types of each contexts' elements:

```
Define CTX-NAME_list : list TYPE1 -> ... -> list TYPEn -> prop by
CTX-NAME_list nil ... nil ;
nabla VAR11 ... VAR1k_1, CTX-NAME_list (TERM11 :: L1) ... (TERM1n :: Ln) :=
    CTX-NAME_list L1 ... Ln /\ FORMULA1 ;
...
nabla VARm1 ... VARmk_m, CTX-NAME_list (TERMm1 :: L1) ... (TERMmn :: Ln) :=
    CTX-NAME_list L1 ... Ln /\ FORMULAm.

Define CTX-NAME : list TYPE1 -> ... -> list TYPEn -> prop by
CTX-NAME G1 ... Gn := exists L1 ... Ln,
    G1 ~ L1 /\ ... /\ Gn ~ Ln /\ CTX-NAME_list L1 ... Ln.
```

Once we have a context specification of the aforementioned form, a suite of lemmas can be automatically generated about it. First, a distributivity lemma can be generated for each index of the specification that allows the context specification to be distributed over partitions of the corresponding context while generating corresponding partitions of the other context(s) as needed for the other indices. The general form of the ith such lemma may be represented as follows:

```
Theorem CTX-NAME_distri : forall G1 ... Gi Gi' Gi'' ... Gn,
CTX-NAME G1 ... Gn -> Gi ~ Gi' ++ Gi'' -> exists G1' G1'' ... Gn' Gn'',
    CTX-NAME G1' ... Gn' /\ CTX-NAME G1'' ... Gn'' /\
    G1 ~ G1' ++ G1'' /\ ... /\ Gn ~ Gn' ++ Gn''.
```

For instance, for `trans_rel`, we might automatically generate the lemma for i = 2 in this form as:

```
Theorem trans_rel_distr2 : forall G1 G2 G2' G2'' G3,
trans_rel G1 G2 G3 -> G2 ~ G2' ++ G2'' -> exists G1' G1'' G3' G3'',
    trans_rel G1' G2' G3' /\ trans_rel G1'' G2'' G3'' /\
    G1 ~ G1' ++ G1'' /\ G3 ~ G3' ++ G3''.
```

Each lemma can be proved automatically as well. The generated proofs follow the structure that we have already seen in Sections 4 and 5. In short, a corresponding lemma involving `partition` is first automatically generated and proved by a routine inductive argument that depends only on the number of clauses and names in the definition. Then, `perm_to_part` is applied to interface the desired lemma's hypotheses with this lemma's, and finally `part_to_perm` is applied to obtain results in the right form.

For lemmas involving `member`, an algorithm exists to automatically lift lemmas proved for traditional context specifications to their multiset versions. Suppose the user proves a lemma of the following form:

```
Theorem USER-LEMMA : forall L1 ... Ln VAR*,
CTX-NAME_list L1 ... Ln -> [member TERM Li ->]* [exists VAR*, ]
    [member TERM Lj /\]* [FORMULA /\]* [TERM = TERM /\]* true.
```

Suppose also that each `FORMULA` and `TERM` does not depend on any of the context variables `Li`, so that any non-`member` assertions are only about the elements of the context(s). Then, a corresponding lemma for multiset-based contexts, of the following form, can be automatically generated and proved:

```
Theorem USER-LEMMA-MSET : forall G1 ... Gn VAR*,
CTX-NAME G1 ... Gn -> [member TERM Gi ->]* [exists VAR*, ]
    [member TERM Gj /\]* [FORMULA /\]* [TERM = TERM /\]* true.
```

The automatically generated proof involves three main steps:

1. The context specification hypothesis `CTX-NAME G1 ... Gn` is unfolded and appropriate instances of `mem_replace` are applied to each of the other hypotheses.
2. The user-provided lemma is applied to the hypotheses constructed in the first step.
3. The conclusions obtained using the user-provided lemma are converted into the desired forms. Nothing needs to be done for the `FORMULA` and equality conclusions, but any obtained instances of `member` are converted to the correct form via appropriate uses of `mem_replace`. Since the original lemma's form was restricted to only allow `member` to pull from contexts described by the context specification, it is certain that the requisite permutations will be available; they are necessarily the same permutations as obtained by unfolding `CTX-NAME G1 ... Gn`.

We envision tactics in Abella for handling the aforementioned automation. A `subst` tactic would implement the `mem_replace` lemma, a `distr` tactic would implement the distributivity lemmas, and a `lift` tactic would implement the procedure for lifting `member`-based lemmas. For example, calling `subst Hi into Hj` would apply the `mem_replace` lemma, as long as `Hi` is an appropriate permutation and `Hj` is an instance of the `member` predicate. On the other hand, calling `distr Hi over Hj` with `Hi` being a context specification defined via the `Context` command and `Hj` being a `perm`-style partition would prove and apply a distributivity lemma for whichever index of the context specification could be matched with the input permutation. And, finally, calling `lift USER-LEMMA` would generate and prove the corresponding `USER-LEMMA-MSET`, adding it as a new hypothesis.

7 Related Work

Our focus in this paper has been on the special problems that arise when binding contexts must be accorded a resource interpretation. While this concern is original to our work, we have superimposed it on a treatment of resources, which is an issue that has received the attention of other researchers. A particular situation in which the need for such a treatment has arisen is in the encoding of linear logic [7] within proof assistants towards mechanizing reasoning about the meta-theoretic properties of this logic. Chaudhuri *et al.* have undertaken this task using the Abella system [4]. They too have used multisets to encode resources, which, in their case, are linear collections of formulas. They observe that the representation of multisets must support the ability to add an element to a multiset and to partition a multiset, and it must ensure that multisets are considered to be equivalent under permutations. These observations underlie our work as well, with the key difference that we have taken permutation equivalence to be fundamental to the representation. This allows us to introduce the ++ constructor that renders partitioning into a syntactic operation rather than needing it to be defined, as is done by the `merge` predicate in [4]. Our approach has the benefit of succinctness, at least in presentation; for example, it accommodates a simple rendition of the partitioning of a multiset into several subcomponents. On the negative side, the definitions of permutation and the addition (or, dually, the selection) of an element are marginally more complex. Similar concerns arise in the encoding of linear logic in the Coq system developed by Olivier Laurent [8]. In that work, the choice was made to use lists to represent linear collections of formulas and to realize the multiset interpretation via an explicit "exchange" rule that is implemented via permutations. While this approach supports a simple encoding, it separates partitioning from permutations, an aspect that can make the analysis of the derivability of particular sequents in linear logic more complex.

The notion of partitionable contexts is also relevant to the LINCX framework [6] that allows the user to define functions whose types correspond to typing judgments in the linear logical framework LLF [3]; theorems given by the type of the function are considered proved if the function can be shown to be total. Unlike our scheme that uses the explicit definition of a permutation relation for treating partitions, LINCX provides a built-in operator $\bowtie$ for *context joins*, whose definition is hidden from the user. One significant difference between these schemes is that, since LLF contexts are inherently ordered, context joins must preserve the relative order of elements whereas `perm`-style partitions need not. Each element of `G = G1` $\bowtie$ `G2` remains in the same order in `G1` and `G2` but is only made *available* in exactly one of them—with only a placeholder in the other for order-preservation and type checking purposes. Since context joins are built-in and system-manipulated, a user need not explicitly drive the functionality of these. However, by the same token, they also cannot affect the functionality. In contrast, we expose the definition of `perm` and allow users to reason about it and prove additional lemmas if needed—though the user also typically *must* reason explicitly about `perm` in order to make use of it.

The encoding that we have used for type assignment in the simply typed linear lambda calculus is based on superimposing linearity explicitly on typing contexts. This choice has been motivated by the eventual application for our work that we discuss in the next section; the simply typed linear lambda calculus figures mainly as an example to highlight the issues that have to be considered in this setting. If the focus is instead on a specific example, then an encoding of a different style could be used to circumvent the issues discussed. For instance, the simply typed linear lambda calculus could have been treated by specifying type assignment and linearity separately; uniqueness of typing in this case would, for example, be a simple consequence of the result for the regular simply typed lambda calculus. This style in fact underlies the encoding of linear logic and other substructural logics described in [5].

The idea of schematically extracting context properties, useful for minimizing the burden of reasoning explicitly about contexts, has also been explored in other settings besides ours. Savary Bélanger and Chaudhuri [2] define a plugin for Abella for concisely defining and extracting properties from what they call *regular context relations*, which describe the structure of LF contexts. This structure is noticeably similar to the structure of context specifications without the addition of the lifting procedure using `perm`. Specifically, in their framework, a user can define a *context schema* that fully specifies the form of the elements in the desired context(s). Then, they can make use of provided *tacticals* for extracting properties of the corresponding context relation. For example, the *inversion* tactical extracts the form of (corresponding) elements in the context(s), much like our membership lemmas. Though both our and their developments define a general form for contexts of interest that capture some desired properties and then provide tools for extracting those properties, the specific goals and the actual form of the contexts differ significantly.

8 Conclusion

In this paper, we have discussed our scheme for specifying binding contexts that must be partitionable in Abella. We have illustrated our ideas by using typing judgments and a translation relation that are encoded directly as definitions in the *reasoning logic* of Abella. However, reasoning in Abella is often done using a two-level logic approach, in which the reasoning logic is augmented with an auxiliary *specification logic* that is well-suited for computation. This paradigm is realized by embedding the specification logic in the reasoning logic via a definition that encapsulates the proof system of the specification logic; one then describes object systems in the specification logic and reasons about them via the ability the embedding provides to reason about derivability in the specification logic. In ongoing work, we are exploring the possibility of using a variant of linear logic called *Forum* [9] that has a computational interpretation as the specification logic in this framework. The motivation for doing so is that linear object systems, such as the linear lambda calculus that we have considered here, can be specified in a logical way by making use of *linear implication* ($\multimap$) to encode resources and their usage, a move that enables metatheoretic properties of the specification logic to be used in simplifying the reasoning process. To support this idea, we must provide an embedding of Forum in Abella. Since formulas in one category in such a logic must be used exactly once, their encoding and usage in the embedding necessitates a treatment of linear contexts with an associated capability for considering their partitioning in the reasoning process.[4] Moreover, when the object system embodies notions of binding, such linear contexts take on the attributes of binding contexts that have been the topic of interest in this paper. Many of the ideas we have discussed remain applicable in this situation and we are in fact incorporating the automation techniques described in Section 6 in our implementation towards providing the user a tool to simplify reasoning developments that make use of the new specification logic.

Acknowledgements

This paper has benefitted greatly from the feedback provided by its reviewers. The work underlying it was supported in its early stages by the National Science Foundation under Grant No. CCF-1617771. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[4]Note that the kind of embedding we are interested in here makes it necessary to treat linear contexts explicitly, unlike what is done, for example, in [5].

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*. Journal of Formalized Reasoning 7(2), doi:10.6092/issn.1972-5787/4650.

[2] Olivier Savary Bélanger & Kaustuv Chaudhuri (2014): *Automatically Deriving Schematic Theorems for Dynamic Contexts*. In: Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP '14, Association for Computing Machinery, New York, NY, USA, doi:10.1145/2631172.2631181.

[3] Iliano Cervesato & Frank Pfenning (2002): *A Linear Logical Framework*. Information & Computation 179(1), pp. 19–75, doi:10.1006/inco.2001.2951.

[4] Kaustuv Chaudhuri, Leonardo Lima & Giselle Reis (2019): *Formalized meta-theory of sequent calculi for linear logics*. Theoretical Computer Science 781, pp. 24–38, doi:10.1016/j.tcs.2019.02.023.

[5] Karl Crary (2010): *Higher-order representation of substructural logics*. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, Association for Computing Machinery, New York, NY, USA, p. 131–142, doi:10.1145/1863543.1863565.

[6] Aina Linn Georges, Agata Murawska, Shawn Otis & Brigitte Pientka (2017): LINCX*: A Linear Logical Framework with First-Class Contexts*. In H. Yang, editor: Programming Languages and Systems, ESOP17, Lecture Notes in Computer Science 10201, Springer, pp. 530–555, doi:10.1007/978-3-662-54434-1_20.

[7] Jean-Yves Girard (1987): *Linear Logic*. Theoretical Computer Science 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[8] Olivier Laurent: YALLA*: an LL library for Coq*. Available from https://perso.ens-lyon.fr/olivier.laurent/yalla/.

[9] Dale Miller (1996): *Forum: A Multiple-Conclusion Specification Logic*. Theoretical Computer Science 165(1), pp. 201–232, doi:10.1016/0304-3975(96)00045-X.

Kuroda's Translation for the $\lambda\Pi$-Calculus Modulo Theory and Dedukti

Thomas Traversié
Université Paris-Saclay, CentraleSupélec, MICS
Gif-sur-Yvette, France
Université Paris-Saclay, Inria, CNRS, ENS-Paris-Saclay, LMF
Gif-sur-Yvette, France
thomas.traversie@centralesupelec.fr

Kuroda's translation embeds classical first-order logic into intuitionistic logic, through the insertion of double negations. Recently, Brown and Rizkallah extended this translation to higher-order logic. In this paper, we adapt it for theories encoded in higher-order logic in the $\lambda\Pi$-calculus modulo theory, a logical framework that extends λ-calculus with dependent types and user-defined rewrite rules. We develop a tool that implements Kuroda's translation for proofs written in DEDUKTI, a proof language based on the $\lambda\Pi$-calculus modulo theory.

1 Introduction

The $\lambda\Pi$-calculus modulo theory [6] is an extension of simply typed λ-calculus with dependent types and user-defined rewrite rules. It is a logical framework, meaning that one can express many theories in it—through the definitions of typed constants and rewrite rules. For instance, it is possible to encode Predicate Logic, Simple Type Theory and the Calculus of Constructions in the $\lambda\Pi$-calculus modulo theory [2]. In particular, theories from other proof systems can be expressed inside this logical framework [20]. The $\lambda\Pi$-calculus modulo theory has been implemented in the concrete language DEDUKTI [1, 15]. Besides automatic proof checking, DEDUKTI can be used as a common language to exchange proofs between different systems. However, if one wants to translate proofs from the *classical* proof assistant HOL LIGHT to the *intuitionistic* proof assistant COQ *via* DEDUKTI, one must transform classical proofs into intuitionistic proofs *inside* DEDUKTI.

Classical logic corresponds to intuitionistic logic extended with the principle of excluded middle $A \vee \neg A$, or equivalently the double-negation elimination $\neg\neg A \Rightarrow A$. Classical logic can be embedded into intuitionistic logic, using double-negations translations. Glivenko [12] proved that any propositional formula A is provable in classical logic if and only if its double negation $\neg\neg A$ is provable in intuitionistic logic. Kolmogorov [17], Gödel [13], Gentzen [10] and Kuroda [18] developed double-negation translations $A \mapsto A^*$, which transforms any first-order formula A such that:

(i) if A is provable in classical logic then its translation A^* is provable in intuitionistic logic,

(ii) A and A^* are classically equivalent.

More recently, Brown and Rizkallah [4] showed that Kolmogorov's and Gödel-Gentzen's translations cannot be extended to higher-order logic. They proved that, in higher-order logic, Kuroda's translation satisfies Property (i), but that it fails in the presence of functional extensionality. In fact [21], Property (i) holds in the presence of functional extensionality under some specific condition, and Property (ii) holds when assuming functional extensionality and propositional extensionality.

Contribution. In this paper, we express Kuroda's translation for theories of the $\lambda\Pi$-calculus modulo theory that are encoded in higher-order logic. It is both an encoding—into a logical framework that features proofs as terms—and an extension—to a logical framework that features dependent types and user-defined rewrite rules—of Kuroda's translation. We implement such translation inside CONSTRUKTI, a tool that translates DEDUKTI files. CONSTRUKTI is tested on a benchmark of a hundred formal proofs. This tool and this benchmark are available at `https://github.com/Deducteam/Construkti`.

Outline of the paper. In Section 2, we present the $\lambda\Pi$-calculus modulo theory and we detail an encoding of higher-order logic in it. In Section 3, we define Kuroda's translation for theories of $\lambda\Pi$-calculus modulo theory that are encoded in higher-order logic, and we prove the embedding of classical logic into intuitionistic logic. In Section 4, we implement CONSTRUKTI and test it on DEDUKTI proofs.

2 Higher-Order Logic in the $\lambda\Pi$-Calculus Modulo Theory

In this section, we present the $\lambda\Pi$-calculus modulo theory, and we detail an encoding of higher-order logic in this logical framework. We characterize the theories considered in the rest of this paper—theories encoded in higher-order logic.

2.1 The $\lambda\Pi$-Calculus Modulo Theory

The Edinburgh Logical Framework [14], also called $\lambda\Pi$-calculus, is an extension of simply typed λ-calculus with dependent types. The $\lambda\Pi$-calculus modulo theory [6] corresponds to the Edinburgh Logical Framework extended with user-defined rewrite rules [7]. Its syntax is given by:

$$\begin{array}{ll}
\textit{Sorts} & s ::= \mathtt{TYPE} \mid \mathtt{KIND} \\
\textit{Terms} & t,u,A,B ::= c \mid x \mid s \mid \Pi x : A.\, B \mid \lambda x : A.\, t \mid t\ u \\
\textit{Contexts} & \Gamma ::= \langle\rangle \mid \Gamma, x : A \\
\textit{Signatures} & \Sigma ::= \langle\rangle \mid \Sigma, c : A \\
\textit{Rewrite systems} & \mathscr{R} ::= \langle\rangle \mid \mathscr{R}, \ell \hookrightarrow r
\end{array}$$

where c is a constant and x is a variable (ranging over disjoint sets). TYPE and KIND are two sorts: terms of type TYPE are called types, and terms of type KIND are called kinds. $\Pi x : A.\, B$ is a dependent product (simply written $A \to B$ if x does not occur in B), $\lambda x : A.\, t$ is an abstraction, and $t\ u$ is an application. Contexts, signatures and rewrite systems are finite sequences, and are written $\langle\rangle$ when empty. Signatures Σ are composed of typed constants $c : A$, where A is a closed term (that is a term with no free variables). Rewrite systems $\mathscr{R}$ are composed of rewrite rules $\ell \hookrightarrow r$, where the head symbol of ℓ is a constant. The $\lambda\Pi$-calculus modulo theory is a logical framework, in which Σ and $\mathscr{R}$ are fixed by the users depending on the logic they are working in. The relation $\hookrightarrow_{\beta\mathscr{R}}$ is generated by β-reduction and by the rewrite rules of $\mathscr{R}$. The conversion $\equiv_{\beta\mathscr{R}}$ is the reflexive, symmetric, and transitive closure of $\hookrightarrow_{\beta\mathscr{R}}$.

The typing rules for the $\lambda\Pi$-calculus modulo theory are given in Figure 1. We write $\vdash \Gamma$ when the context Γ is well formed, and $\Gamma \vdash t : A$ when the term t is of type A in the context Γ. For convenience, $\langle\rangle \vdash t : A$ is simply written $\vdash t : A$. The standard weakening rule is admissible.

We write $\Lambda(\Sigma)$ for the set of terms whose constants belong to Σ. We say that $(\Sigma, \mathscr{R})$ is a theory when: (i) for each rule $\ell \hookrightarrow r \in \mathscr{R}$, both ℓ and r belongs to $\Lambda(\Sigma)$, (ii) $\hookrightarrow_{\beta\mathscr{R}}$ is confluent on $\Lambda(\Sigma)$, and (iii) each

$$\frac{}{\vdash \langle\rangle}\ [\textsc{Empty}] \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : s}{\vdash \Gamma, x : A}\ [\textsc{Decl}]\ x \notin \Gamma \qquad \frac{\vdash \Gamma}{\Gamma \vdash \texttt{TYPE} : \texttt{KIND}}\ [\textsc{Sort}]$$

$$\frac{\vdash \Gamma \qquad \vdash A : s}{\Gamma \vdash c : A}\ [\textsc{Const}]\ c : A \in \Sigma \qquad \frac{\vdash \Gamma}{\Gamma \vdash x : A}\ [\textsc{Var}]\ x : A \in \Gamma$$

$$\frac{\Gamma \vdash A : \texttt{TYPE} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\, B : s}\ [\textsc{Prod}] \qquad \frac{\Gamma \vdash A : \texttt{TYPE} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}\ [\textsc{Abs}]$$

$$\frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \leftarrow u]}\ [\textsc{App}] \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ [\textsc{Conv}]\ A \equiv_{\beta\mathscr{R}} B$$

Figure 1: Typing rules of the $\lambda\Pi$-calculus modulo theory.

rule $\ell \hookrightarrow r \in \mathscr{R}$ preserves types (for all context Γ, substitution θ, and term $A \in \Lambda(\Sigma)$, if $\Gamma \vdash \ell\theta : A$ then $\Gamma \vdash r\theta : A$).

In the $\lambda\Pi$-calculus modulo theory, if $\Gamma \vdash t : A$ then Γ is well-formed and A is well-typed. To prove this, we use the two following properties.

Lemma 1. *If $\Gamma \vdash t : A$, then either $A = \texttt{KIND}$ or $\Gamma \vdash A : s$ for $s = \texttt{TYPE}$ or $s = \texttt{KIND}$. If $\Gamma \vdash \Pi x : A.\, B : s$, then $\Gamma \vdash A : \texttt{TYPE}$.*

2.2 An Encoding of Higher-Order Logic

It is possible to express higher-order logic in the $\lambda\Pi$-calculus modulo theory [2]. For this, we have to introduce the notions of proposition and proof. We declare the constant *Set*, which represents the universe of sorts, along with the injection *El* that maps sorts to the type of its elements. The constant *Prop* defines the universe of propositions, and the injection *Prf* maps propositions into the type of its proofs. In this encoding, we say that P of type *Prop* is a proposition, that *Prf* P is a formula and that a term of type *Prf* P is a proof of P.

$Set : \texttt{TYPE}$	$El : Set \to \texttt{TYPE}$	$\rightsquigarrow\ : Set \to Set \to Set$	$o : Set$
$Prop : \texttt{TYPE}$	$Prf : Prop \to \texttt{TYPE}$	$El\ (x \rightsquigarrow y) \hookrightarrow El\ x \to El\ y$	$El\ o \hookrightarrow Prop$

The arrow $\rightsquigarrow$ (written infix) is used to represent function types between terms of type *Set*. Propositions are considered as objects, using the sort o and the rewrite rule $El\ o \hookrightarrow Prop$.

Now that we have introduced the notions of proposition and proof, we can define the logical connectives and quantifiers of predicate logic.

$\Rightarrow\ : Prop \to Prop \to Prop$	$\top : Prop$	$\forall : \Pi x : Set.\ (El\ x \to Prop) \to Prop$
$\wedge : Prop \to Prop \to Prop$	$\bot : Prop$	$\exists : \Pi x : Set.\ (El\ x \to Prop) \to Prop$
$\vee : Prop \to Prop \to Prop$	$\neg : Prop \to Prop$	$\Leftrightarrow\ : Prop \to Prop \to Prop$

Remark that $\forall$ and $\exists$ are polymorphic quantifiers that can be applied to the sort of proposition o. Hence the higher-order feature directly derives from the rewrite rule $El\ o \hookrightarrow Prop$.

In natural deduction, each connective and quantifier comes with an introduction and an elimination inference rule. The encoding of the notions of proposition and proof is well-suited for representing inference rules: logical consequences are represented by arrow types, and parameters are represented by dependent types. For instance, the inference rule for the elimination of disjunction

$$\frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R \qquad \Gamma, Q \vdash R}{\Gamma \vdash R}$$

is simply expressed by the constant $\mathsf{or_e}$ of type

$$\Pi p,q : Prop.\ Prf\ (p \vee q) \to \Pi r : Prop.\ (Prf\ p \to Prf\ r) \to (Prf\ q \to Prf\ r) \to Prf\ r$$

that can be used for any context Γ. The constants representing the natural deduction rules for the logical connectives are:

$\mathsf{imp_i} : \Pi p,q : Prop.\ (Prf\ p \to Prf\ q) \to Prf\ (p \Rightarrow q)$
$\mathsf{imp_e} : \Pi p,q : Prop.\ Prf\ (p \Rightarrow q) \to Prf\ p \to Prf\ q$
$\mathsf{and_i} : \Pi p : Prop.\ Prf\ p \to \Pi q : Prop.\ Prf\ q \to Prf\ (p \wedge q)$
$\mathsf{and_{e\ell}} : \Pi p,q : Prop.\ Prf\ (p \wedge q) \to Prf\ p$
$\mathsf{and_{er}} : \Pi p,q : Prop.\ Prf\ (p \wedge q) \to Prf\ q$
$\mathsf{or_{i\ell}} : \Pi p : Prop.\ Prf\ p \to \Pi q : Prop.\ Prf\ (p \vee q)$
$\mathsf{or_{ir}} : \Pi p,q : Prop.\ Prf\ q \to Prf\ (p \vee q)$
$\mathsf{or_e} : \Pi p,q : Prop.\ Prf\ (p \vee q) \to \Pi r : Prop.\ (Prf\ p \to Prf\ r) \to (Prf\ q \to Prf\ r) \to Prf\ r$
$\mathsf{neg_i} : \Pi p : Prop.\ (Prf\ p \to Prf\ \bot) \to Prf\ (\neg p)$
$\mathsf{neg_e} : \Pi p : Prop.\ Prf\ (\neg p) \to Prf\ p \to Prf\ \bot$

For convenience, the semantic of the logical biconditional is encoded through the rewrite rule $p \Leftrightarrow q \hookrightarrow (p \Rightarrow q) \wedge (q \Rightarrow p)$. The introduction of tautology and the elimination of contradiction are encoded by:

$\mathsf{top_i} : Prf\ \top$
$\mathsf{bot_e} : Prf\ \bot \to \Pi p : Prop.\ Prf\ p$

The natural deduction rules for the quantifiers are represented by the following constants:

$\mathsf{all_i} : \Pi a : Set.\ \Pi p : El\ a \to Prop.\ (\Pi x : El\ a.\ Prf\ (p\ x)) \to Prf\ (\forall\ a\ p)$
$\mathsf{all_e} : \Pi a : Set.\ \Pi p : El\ a \to Prop.\ Prf\ (\forall\ a\ p) \to \Pi x : El\ a.\ Prf\ (p\ x)$
$\mathsf{ex_i} : \Pi a : Set.\ \Pi p : El\ a \to Prop.\ \Pi x : El\ a.\ Prf\ (p\ x) \to Prf\ (\exists\ a\ p)$
$\mathsf{ex_e} : \Pi a : Set.\ \Pi p : El\ a \to Prop.\ Prf\ (\exists\ a\ p) \to \Pi r : Prop.\ (\Pi x : El\ a.\ Prf\ (p\ x) \to Prf\ r) \to Prf\ r$

All those constants and rewrite rules define the encoding of intuitionistic higher-order logic in the $\lambda\Pi$-calculus modulo theory. We write Σ^i_{HOL} for its constants and $\mathscr{R}_{HOL}$ for its rewrite rules. The principle of excluded middle is represented by:

$\mathsf{pem} : \Pi p : Prop.\ Prf\ (p \vee \neg p)$

Classical higher-order logic is encoded in the $\lambda\Pi$-calculus modulo theory by the constants Σ^c_{HOL} (that is Σ^i_{HOL} along with pem) and by the rewrite rules $\mathscr{R}_{HOL}$.

Remark that we have decided to encode the natural deduction rules via *typed constants*, while they are often expressed via *rewrite rules* in the $\lambda\Pi$-calculus modulo theory [2]. For instance, both the introduction and the elimination of implication can be derived from the rewrite rule $Prf\ (p \Rightarrow q) \hookrightarrow Prf\ p \to Prf\ q$. So as to perform the translation from classical logic to intuitionistic logic, the natural deduction steps must be *explicit* deduction steps, and cannot be *implicit* computation steps. That is why we encode the natural deduction rules with a deep embedding—via typed constants—instead of a shallow embedding—via rewrite rules.

2.3 Theories Encoded in Higher-Order Logic

When working with the encoding of higher-order logic in the $\lambda\Pi$-calculus modulo theory, it is possible to mix sorts, propositions and proofs—which is not expected in higher-order logic. For example, propositions can be inserted in sorts when we have a term of type $Prop \to Set$, and proofs can be inserted in propositions when we have a term of type $\Pi p : Prop.\ Prf\ p \to Prop$. To avoid such behavior, we introduce five grammars:

$$\begin{aligned}
\kappa_1 &::= Set \mid \kappa_1 \to \kappa_1 \\
\kappa_2 &::= Prop \mid El\ a \mid \Pi x : \kappa_i.\ \kappa_2 \text{ with } i \in \{1,2\} \\
\kappa_3 &::= Prf\ p \mid \kappa_3 \to \kappa_3 \mid \Pi x : \kappa_i.\ \kappa_3 \text{ with } i \in \{1,2\} \\
\kappa_4 &::= \texttt{TYPE} \mid \Pi x : \kappa_i.\ \kappa_4 \text{ with } i \in \{1,2\} \\
\kappa_5 &::= \texttt{KIND}
\end{aligned}$$

The grammar κ_3 generates formulas and inference rules. The grammar κ_4 generates a subclass of kinds, and κ_5 only generates `KIND`. We characterize the judgments of the $\lambda\Pi$-calculus modulo theory to ensure that types and kinds are generated by one of those grammars.

Definition 1 (κ-property)**.** *The judgment $\Gamma \vdash t : A$ satisfies the κ-property when $A \in \kappa_i$ for some $i \in [\![1,5]\!]$. The judgment $\vdash \Gamma$ satisfies the κ-property when for each $(x : A) \in \Gamma$ we have $A \in \kappa_i$ for some $i \in [\![1,5]\!]$. A derivation satisfies the κ-property when each of its judgments satisfies the κ-property.*

Theories encoded in higher-order logic are theories that feature the base higher-order encoding and in which the user-defined constants satisfy the κ-property.

Definition 2 (Theory encoded in higher-order logic)**.** *Let $\mathscr{T} = (\Sigma, \mathscr{R})$ be a theory in the $\lambda\Pi$-calculus modulo theory. $\mathscr{T}$ is encoded in higher-order logic when:*

1. *$\Sigma = \Sigma_{HOL}^k \cup \Sigma_{\mathscr{T}}$ with $k \in \{i, c\}$ and $\Sigma_{HOL} \cap \Sigma_{\mathscr{T}} = \emptyset$,*
2. *$\mathscr{R} = \mathscr{R}_{HOL} \cup \mathscr{R}_{\mathscr{T}}$ with $\mathscr{R}_{HOL} \cap \mathscr{R}_{\mathscr{T}} = \emptyset$,*
3. *for every $c : A \in \Sigma_{\mathscr{T}}$, the judgment $\vdash c : A$ satisfies the κ-property,*
4. *for every $\ell \hookrightarrow r \in \mathscr{R}_{\mathscr{T}}$, ℓ is neither Prf nor $\forall$.*

The fourth condition will ensure that the translation of a rewrite rule is a well-defined rewrite rule. Theories encoded in higher-order logic extend higher-order logic with user-defined rewrite rules and inference rules. The introduction of rewrite rules is part and parcel of deduction modulo theory [8], while the introduction of inference rules has been developed in superdeduction modulo theory [3, 16].

When considering a theory encoded in higher-order logic, all the user-defined constants satisfy the κ-property. In that respect, the only way to mix sorts, propositions and proofs is through λ-abstractions. For instance, $(\lambda P : Prop.\ o)$ is a term taking as input a proposition and returning a sort. The type

$El\ ((\lambda P : Prop.\ o)\ \bot)$ mixes propositions and sorts, but it is β-convertible to $El\ o$, in which no proposition occurs. Using this principle, we can transform every derivation of a theory encoded in higher-order logic into a derivation that satisfies the κ-property, by applying β-reduction on fragments of the derivation. When a derivation satisfies the κ-property, the rewrite rules $\ell \hookrightarrow r$ with ℓ and r of type $A \in \kappa_3$ cannot be used. In the rest of this paper and without loss of generality, we only consider derivations that satisfy the κ-property and rewrite rules $\ell \hookrightarrow r$ with ℓ and r of type $A \in \kappa_i$ for $i \neq 3$.

Example 1 (Equational theory)**.** *Consider the theory* $\mathscr{T} = (\Sigma_{HOL} \cup \Sigma_{eq}, \mathscr{R}_{HOL} \cup \mathscr{R}_{eq})$*, with a polymorphic equality symbol* $= \ : \Pi a : Set.\ El\ a \to El\ a \to Prop$*, and a rewrite rule for the Leibniz principle* $Prf\ (= a\ x\ y) \hookrightarrow \Pi P : El\ a \to Prop.\ Prf\ (P\ x) \to Prf\ (P\ y)$*. This theory is encoded in higher-order logic. We can prove that the equality is reflexive, symmetric and transitive. For instance, the proof of reflexivity is given by* $\lambda a : Set.\ \mathsf{all_i}\ a\ (\lambda x : El\ a. = a\ x\ x)\ (\lambda x : El\ a.\ \lambda P : El\ a \to Prop.\ \lambda P_x : Prf\ (P\ x).\ P_x)$ *which is of type* $\Pi a : Set.\ Prf\ (\forall\ a\ (\lambda x : El\ a. = a\ x\ x))$*.*

3 Kuroda's Translation in the $\lambda\Pi$-Calculus Modulo Theory

In this section, we adapt Kuroda's double-negation translation to the $\lambda\Pi$-calculus modulo theory, when working in theories encoded in higher-order logic. Kuroda's translation [18] inserts a double negation in front of formulas and one after every universal quantifier. More formally, we have $A^{Ku} := \neg\neg A_{Ku}$ where A_{Ku} is defined by induction:

$$\begin{array}{lll}
(A \Rightarrow B)_{Ku} := A_{Ku} \Rightarrow B_{Ku} & (\neg A)_{Ku} := \neg A_{Ku} & P_{Ku} := P \text{ if } P \text{ atomic} \\
(A \wedge B)_{Ku} := A_{Ku} \wedge B_{Ku} & \top_{Ku} := \top & (\forall x\, A)_{Ku} := \forall x\, \neg\neg A_{Ku} \\
(A \vee B)_{Ku} := A_{Ku} \vee B_{Ku} & \bot_{Ku} := \bot & (\exists x\, A)_{Ku} := \exists x\, A_{Ku}
\end{array}$$

This translation embeds classical logic into intuitionistic logic, as for any first-order formula A we have $\Gamma \vdash A$ in classical logic if and only if $\Gamma^{Ku} \vdash A^{Ku}$ in intuitionistic logic.

3.1 Translation of Terms and Theories

When working inside a theory encoded in higher-order logic in the $\lambda\Pi$-calculus modulo theory, every formula has head symbol Prf. Inserting a double negation in front of every formula is therefore equivalent to inserting it after every Prf symbol. In that respect, we define a single translation $t \mapsto t^{Ku}$ by induction on the terms of the $\lambda\Pi$-calculus modulo theory. The translation of Prf is $\lambda p.\ Prf\ (\neg\neg p)$, and the translation of the universal quantifier $\forall$ is $\lambda a.\ \lambda p.\ \forall\ a\ (\lambda z.\ \neg\neg(p\ z))$. The translation of λ-abstraction $\lambda x : A.\ t$ is naturally given by $\lambda x : A^{Ku}.\ t^{Ku}$, the one of dependent type $\Pi x : A.\ B$ is given by $\Pi x : A^{Ku}.\ B^{Ku}$ and the one of application $t\ u$ is defined by $t^{Ku}\ u^{Ku}$.

As we are in the $\lambda\Pi$-calculus modulo theory with the *proofs-as-terms* paradigm, we have to translate proofs as well. Kuroda's translation relies on the fact that the translation of each natural deduction rule is admissible in intuitionistic logic. For instance, the introduction of implication allows to derive $\Gamma \vdash P \Rightarrow Q$ from $\Gamma, P \vdash Q$. In intuitionistic logic, $\Gamma^{Ku} \vdash (P \Rightarrow Q)^{Ku}$ is derivable from $\Gamma^{Ku}, P^{Ku} \vdash Q^{Ku}$. In the $\lambda\Pi$-calculus modulo theory, the *constant* $\mathsf{imp_i}$ is of type $\Pi p, q : Prop.\ (Prf\ p \to Prf\ q) \to Prf\ (p \Rightarrow q)$, and we can build a *term* $\mathsf{imp_i}^i$ of type $\Pi p, q : Prop.\ (Prf\ \neg\neg p \to Prf\ \neg\neg q) \to Prf\ \neg\neg(p \Rightarrow q)$, that only depends on the constants representing *intuitionistic* natural deduction rules. Each constant c of type A representing a natural deduction rule is translated by the term c^i of type A^{Ku}, where c^i is an intuitionistic proof term of A^{Ku}.

Definition 3 (Translation of terms)**.** *Kuroda's translation is inductively defined on the terms of the $\lambda\Pi$-calculus modulo theory by:*

$$
\begin{aligned}
&x^{Ku} := x\\
&c^{Ku} := \begin{cases} \lambda p.\, Prf\ (\neg\neg p) & \text{if } c = Prf\\ \lambda x.\, \lambda p.\, \forall\, x\ (\lambda z.\, \neg\neg(p\ z)) & \text{if } c = \forall\\ c^{i} & \text{if } c \text{ is a constant representing a natural deduction rule}\\ c & \text{otherwise}\end{cases}\\
&s^{Ku} := s\\
&(\lambda x : A.\, t)^{Ku} := \lambda x : A^{Ku}.\, t^{Ku}\\
&(\Pi x : A.\, B)^{Ku} := \Pi x : A^{Ku}.\, B^{Ku}\\
&(t\ u)^{Ku} := t^{Ku}\ u^{Ku}
\end{aligned}
$$

Proposition 1. *For every constant $c : A \in \Sigma_{HOL}$ representing a natural deduction rule, we have $\vdash c^{i} : A^{Ku}$ in the theory $(\Sigma^{i}_{HOL}, \mathscr{R}_{HOL})$.*

Proof. We have formalized the proof terms c^{i} in DEDUKTI[1]. For instance, $\mathsf{top_i}^{i}$ is given in Section 4. $\square$

As we are not mixing sorts, propositions and proofs, we know that the symbol $\forall$, the symbol Prf and the constants representing the natural deduction rules only occur in the grammar κ_3. Therefore, any type $A \in \kappa_i$ is modified by Kuroda's translation for $i = 3$, whereas $A^{Ku} = A$ for $i \neq 3$.

We have defined the translation for terms, and we now want to define it for theories. Intuitively, we would like to translate a rewrite rule $\ell \hookrightarrow r$ by $\ell^{Ku} \hookrightarrow r^{Ku}$. However, if the head constant of ℓ is Prf or $\forall$, then the head symbol of ℓ^{Ku} is Prf^{Ku} or $\forall^{Ku}$, that is a λ-abstraction and not a constant. Hence $\ell^{Ku} \hookrightarrow r^{Ku}$ may not be a valid rewrite rule in the $\lambda\Pi$-calculus modulo theory. We write $\lfloor \ell^{Ku} \rfloor$ for the term obtained by β-reducing the head symbol of ℓ^{Ku} if it is Prf^{Ku} or $\forall^{Ku}$.

Definition 4. *The translation $t \mapsto t^{Ku}$ is extended to contexts, signatures and rewrite systems by:*

$$
\begin{aligned}
&\langle\rangle^{Ku} ::= \langle\rangle\\
&(\Gamma, x : A)^{Ku} := \Gamma^{Ku}, x : A^{Ku}\\
&(\Sigma, c : A)^{Ku} := \Sigma^{Ku}, c : A^{Ku}\\
&(\mathscr{R}, \ell \hookrightarrow r)^{Ku} := \mathscr{R}^{Ku}, \lfloor \ell^{Ku} \rfloor \hookrightarrow r^{Ku}
\end{aligned}
$$

When translating a theory encoded in higher-order logic, we replace Σ^{c}_{HOL} by Σ^{i}_{HOL}, and we translate the user-defined signature $\Sigma_{\mathscr{T}}$ and rewrite system $\mathscr{R}_{\mathscr{T}}$.

Definition 5 (Translation of theories)**.** *Let $\mathscr{T} = (\Sigma^{c}_{HOL} \cup \Sigma_{\mathscr{T}}, \mathscr{R}_{HOL} \cup \mathscr{R}_{\mathscr{T}})$ be a theory encoded in higher-order logic. The translation of $\mathscr{T}$ is $\mathscr{T}^{Ku} = (\Sigma^{i}_{HOL} \cup \Sigma^{Ku}_{\mathscr{T}}, \mathscr{R}_{HOL} \cup \mathscr{R}^{Ku}_{\mathscr{T}})$.*

Remark that $\mathscr{T}^{Ku}$ is a theory. Specifically, rewrite rules $\lfloor \ell^{Ku} \rfloor \hookrightarrow r^{Ku} \in \mathscr{R}^{Ku}_{\mathscr{T}}$ are always well-defined, since ℓ is neither Prf nor $\forall$, and by definition of $\lfloor \ell^{Ku} \rfloor$.

3.2 Embedding Classical Logic into Intuitionistic Logic

We aim at proving that the extension of Kuroda's translation in the $\lambda\Pi$-calculus modulo theory indeed embeds classical logic into intuitionistic logic. In other words, we want to show that $\Gamma \vdash t : A$ in $\mathscr{T}$ entails $\Gamma^{Ku} \vdash t^{Ku} : A^{Ku}$ in $\mathscr{T}^{Ku}$. To do so, we translate the derivations step by step. In particular, when the CONV rule is used with $A \equiv_{\beta\mathscr{R}} B$ in $\mathscr{T}$, we want to have $A^{Ku} \equiv_{\beta\mathscr{R}} B^{Ku}$ in $\mathscr{T}^{Ku}$.

[1]See `https://github.com/Deducteam/Construkti/blob/master/kuroda.dk`.

Lemma 2 (Translation of substitutions)**.** $(t[z \leftarrow w])^{Ku} = t^{Ku}[z \leftarrow w^{Ku}]$

Proof. By induction on the term t. We have $(c[z \leftarrow w])^{Ku} = c^{Ku} = c^{Ku}[z \leftarrow w^{Ku}]$ since c^{Ku} is a closed term. Similarly, $(s[z \leftarrow w])^{Ku} = s^{Ku} = s^{Ku}[z \leftarrow w^{Ku}]$. If $x \neq z$, then $(x[z \leftarrow w])^{Ku} = x^{Ku} = x^{Ku}[z \leftarrow w^{Ku}]$. If $x = z$, then $(x[z \leftarrow w])^{Ku} = w^{Ku} = x[z \leftarrow w^{Ku}] = x^{Ku}[z \leftarrow w^{Ku}]$. The cases for λ-abstractions, dependent types, and applications follow from the induction hypotheses. □

Lemma 3 (Translation of conversions)**.** *If $A \equiv_{\beta\mathscr{R}} B$ in $\mathscr{T}$, then $A^{Ku} \equiv_{\beta\mathscr{R}} B^{Ku}$ in $\mathscr{T}^{Ku}$.*

Proof. By induction on the construction of $A \equiv_{\beta\mathscr{R}} B$.

- If $\ell \hookrightarrow r$ in $\mathscr{T}$, then we show $(\ell\theta)^{Ku} \equiv_{\beta\mathscr{R}} (r\theta)^{Ku}$ in $\mathscr{T}^{Ku}$ for any substitution θ. For $\ell \hookrightarrow r \in \mathscr{R}_{HOL}$, we have $\ell^{Ku} = \ell$ and $r^{Ku} = r$, and we use Lemma 2. For $\ell \hookrightarrow r \in \mathscr{R}_{\mathscr{T}}$, we have $\lfloor \ell^{Ku} \rfloor \hookrightarrow r^{Ku} \in \mathscr{R}_{\mathscr{T}}^{Ku}$, which entails that $(\ell\theta)^{Ku} = \ell^{Ku}\theta^{Ku} \equiv_{\beta\mathscr{R}} \lfloor \ell^{Ku} \rfloor \theta^{Ku} \equiv_{\beta\mathscr{R}} r^{Ku}\theta^{Ku} = (r\theta)^{Ku}$ by Lemma 2.
- If $(\lambda x : A.\ t)\ u \hookrightarrow t[x \leftarrow u]$ in $\mathscr{T}$, then we have $((\lambda x : A.\ t)\ u)^{Ku} = (\lambda x : A^{Ku}.\ t^{Ku})\ u^{Ku}$, which β-reduces to $t^{Ku}[x \leftarrow u^{Ku}]$, that is $(t[x \leftarrow u])^{Ku}$ using Lemma 2.
- Closure by context, reflexivity, symmetry, and transitivity are immediate.

□

Theorem 1 (Translation of judgments)**.** *Let $\mathscr{T}$ be a theory encoded in higher-order logic.*

- *If $\vdash \Gamma$ in $\mathscr{T}$ then $\vdash \Gamma^{Ku}$ in $\mathscr{T}^{Ku}$.*
- *If $\Gamma \vdash t : A$ in $\mathscr{T}$ then $\Gamma^{Ku} \vdash t^{Ku} : A^{Ku}$ in $\mathscr{T}^{Ku}$.*

Proof. We proceed by induction on the derivation. We present the most interesting cases, the others follow the definition and the induction hypotheses.

- CONST: By induction we have $\vdash \Gamma^{Ku}$ and $\Gamma^{Ku} \vdash A^{Ku} : s^{Ku}$ in $\mathscr{T}^{Ku}$.

  If $c : A \in \Sigma_{\mathscr{T}}$, then $c : A^{Ku} \in \Sigma_{\mathscr{T}}^{Ku}$ and we derive $\Gamma^{Ku} \vdash c : A^{Ku}$ using CONST.

  Suppose that $c = \mathit{Prf}$. We simply derive $\Gamma^{Ku} \vdash \lambda p.\ \mathit{Prf}\ (\neg\neg p) : \mathit{Prop} \to \mathtt{TYPE}$, that is $\Gamma^{Ku} \vdash \mathit{Prf}^{Ku} : (\mathit{Prop} \to \mathtt{TYPE})^{Ku}$, in $\mathscr{T}^{Ku}$.

  Suppose that $c = \forall$. We simply derive $\Gamma^{Ku} \vdash \lambda x.\ \lambda p.\ \forall\ x\ (\lambda z.\ \neg\neg(p\ z)) : \Pi x : \mathit{Set}.\ (\mathit{El}\ x \to \mathit{Prop}) \to \mathit{Prop}$, that is $\Gamma^{Ku} \vdash \forall^{Ku} : (\Pi x : \mathit{Set}.\ (\mathit{El}\ x \to \mathit{Prop}) \to \mathit{Prop})^{Ku}$, in $\mathscr{T}^{Ku}$.

  Suppose that c is a constant representing a natural deduction rule. Using Proposition 1, we have $\Gamma^{Ku} \vdash c^i : A^{Ku}$ in $\mathscr{T}^{Ku}$, that is $\Gamma^{Ku} \vdash c^{Ku} : A^{Ku}$. In particular, we replace the classical axiom $\mathsf{pem} : \Pi p : \mathit{Prop}.\ \mathit{Prf}\ (p \vee \neg p)$ by the intuitionistic term $\mathsf{pem}^i : \Pi p : \mathit{Prop}.\ \mathit{Prf}\ (\neg\neg(p \vee \neg p))$.

  Otherwise, $c : A \in \Sigma_{HOL}$ but is not Prf, not $\forall$, and not a constant representing a natural deduction rule. Then A does not contain Prf and $\forall$, so $A^{Ku} = A$. We derive $\Gamma^{Ku} \vdash c : A^{Ku}$ using CONST.
- CONV: By induction we have $\Gamma^{Ku} \vdash t^{Ku} : A^{Ku}$ in $\mathscr{T}^{Ku}$ and $\Gamma^{Ku} \vdash B^{Ku} : s^{Ku}$ in $\mathscr{T}^{Ku}$. From Lemma 3, we know that $A^{Ku} \equiv_{\beta\mathscr{R}} B^{Ku}$, and we conclude that $\Gamma^{Ku} \vdash t^{Ku} : B^{Ku}$ in $\mathscr{T}^{Ku}$ using CONV.

□

Example 2 (Translated equational theory)**.** *The translation of the theory $\mathscr{T} = (\Sigma_{HOL} \cup \Sigma_{eq}, \mathscr{R}_{HOL} \cup \mathscr{R}_{eq})$ of Example 1 is obtained by taking the equality symbol $=\ : \Pi a : \mathit{Set}.\ \mathit{El}\ a \to \mathit{El}\ a \to \mathit{Prop}$ (which remains unchanged), and by transforming the rewrite rule $\mathit{Prf}\ (=\ a\ x\ y) \hookrightarrow \Pi P : \mathit{El}\ a \to \mathit{Prop}.\ \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ y)$ into $\mathit{Prf}\ (\neg\neg(=\ a\ x\ y)) \hookrightarrow \Pi P : \mathit{El}\ a \to \mathit{Prop}.\ \mathit{Prf}\ (\neg\neg(P\ x)) \to \mathit{Prf}\ (\neg\neg(P\ y))$. The proof of reflexivity is now given by $\lambda a : \mathit{Set}.\ \mathsf{all}_{\mathsf{i}}^{i}\ a\ (\lambda x : \mathit{El}\ a.\ =\ a\ x\ x)\ (\lambda x : \mathit{El}\ a.\ \lambda P : \mathit{El}\ a \to \mathit{Prop}.\ \lambda P_x : \mathit{Prf}\ (\neg\neg(P\ x)).\ P_x)$ which is of type $\Pi a : \mathit{Set}.\ \mathit{Prf}\ (\neg\neg(\forall\ a\ (\lambda x : \mathit{El}\ a.\ \neg\neg(=\ a\ x\ x))))$.*

3.3 Back to the Original Theory

We have shown that, in the $\lambda\Pi$-calculus modulo theory, $\Gamma \vdash t : A$ in $\mathscr{T}$ implies $\Gamma^{Ku} \vdash t^{Ku} : A^{Ku}$ in $\mathscr{T}^{Ku}$. We now want to prove the reverse implication: if there exists an intuitionistic proof of A^{Ku} in $\mathscr{T}^{Ku}$, then there exists a classical proof of A in $\mathscr{T}$. To do so, we reason in two steps: first we show that it is possible to build a proof of A from a proof of A^{Ku} in classical logic, and then we show that any result in $\mathscr{T}^{Ku}$ can also be derived in $\mathscr{T}$.

The first step consists in proving that, for any $A \in \kappa_3$, it is possible to derive A^{Ku} from A. For this, we show that any proposition P and its translation P^{Ku} are classically equivalent. Such a result is not necessarily true in higher-order logic. We assume some property, called the Kuroda equivalence.

Definition 6 (Kuroda equivalence)**.** *Let Γ be a context, t be a constant or a variable such that $\Gamma \vdash t : T_1 \to \ldots \to T_n \to Prop$, and $u_1, \ldots, u_n$ be terms such that $\Gamma \vdash u_i : T_i$. There exists some p such that $\Gamma \vdash p : Prf\ ((t\ u_1\ \ldots\ u_n)^{Ku} \Leftrightarrow t\ u_1\ \ldots\ u_n)$.*

The Kuroda equivalence property is derivable from functional extensionality and propositional extensionality in classical logic [21]. Remark that it is satisfied for the usual logical connectives and quantifiers. For instance, we have $A_{Ku} \wedge B_{Ku} \Leftrightarrow A \wedge B$ and $\forall x\ \neg\neg A_{Ku} \Leftrightarrow \forall x\ A$ in classical logic. In the rest of this paper, we work assuming the Kuroda equivalence.

Lemma 4. *Any proposition P is β-convertible to a variable x, a constant c, or an application $t\ u_1 \ldots\ u_n$ where t is a constant or a variable of type $T_1 \to \ldots \to T_n \to Prop$ and $u_1, \ldots, u_n$ are terms of type $T_1, \ldots, T_n$.*

The constant c may be $\top$ or $\bot$, and the head symbol of the application may be any connective, quantifier or predicate.

Proposition 2. *Let $\Gamma \vdash P : Prop$. In the theory $(\Sigma^c_{HOL} \cup \Sigma, \mathscr{R}_{HOL} \cup \mathscr{R})$, there exists some proof term m_P such that $\Gamma \vdash m_P : Prf\ (P^{Ku} \Leftrightarrow P)$.*

Proof. We distinguish cases thanks to Lemma 4.

- Suppose that P is β-convertible to a variable x. We have $x^{Ku} = x$ so we build some m_x such that $\Gamma \vdash m_x : Prf\ (x^{Ku} \Leftrightarrow x)$. Since P is β-convertible to x, P^{Ku} is β-convertible to x^{Ku} (see Lemma 3) and we conclude that $\Gamma \vdash m_x : Prf\ (P^{Ku} \Leftrightarrow P)$.
- If P is β-convertible to a constant c, then we are in the case where $c^{Ku} = c$ and we proceed similarly.
- Suppose that P is β-convertible to an application $t\ u_1 \ldots\ u_n$ where t is a constant or a variable. P^{Ku} is β-convertible to $(t\ u_1 \ldots\ u_n)^{Ku}$ and we conclude using the Kuroda equivalence.

$\square$

Lemma 5. *Let $A \in \kappa_3$ and ℓ be a strict subterm of A. In the theory $(\Sigma^c_{HOL} \cup \Sigma, \mathscr{R}_{HOL} \cup \mathscr{R})$, for any context Γ, there exists some t such that $\Gamma \vdash t : A[\ell]$ if and only if there exists some t' such that $\Gamma \vdash t' : A[\ell^{Ku}]$.*

Proof. We proceed by induction on the term A using the fact that A is generated by κ_3.

- Suppose that $A = Prf\ P$. If $\forall$ does not occur in ℓ, then $\ell^{Ku} = \ell$ and $P[\ell^{Ku}] = P[\ell]$, so we directly conclude. Otherwise, we use Proposition 2 on the right proposition.
- Suppose that $A = \Pi x : B.\ C$ with $B \in \kappa_1$ or $B \in \kappa_2$. If ℓ occurs in B, then by definition $B[\ell^{Ku}] = B[\ell]$, so $\ell^{Ku} = \ell$ and we directly conclude. Suppose that ℓ only occurs in C and that there exists some t such that $\Gamma \vdash t : \Pi x : B.\ C[\ell]$. By induction on C with $\Gamma, x : B \vdash t\ x : C[\ell]$ (obtained by weakening), we get some t'_C such that $\Gamma, x : B \vdash t'_C : C[\ell^{Ku}]$. Therefore, we have $\Gamma \vdash \lambda x : B.\ t'_C : \Pi x : B.\ C[\ell^{Ku}]$. We proceed similarly for the reverse implication.

- Suppose that $A = B \to C$ with $B, C \in \kappa_3$. Suppose that we have $\Gamma \vdash t : B[\ell] \to C[\ell]$. By induction on B with $\Gamma, x : B[\ell^{Ku}] \vdash x : B[\ell^{Ku}]$, we get some t_B such that $\Gamma, x : B[\ell^{Ku}] \vdash t_B : B[\ell]$. By induction on C with $\Gamma, x : B[\ell^{Ku}] \vdash t\ t_B : C[\ell]$, we get some t'_C such that $\Gamma, x : B[\ell^{Ku}] \vdash t'_C : C[\ell^{Ku}]$. We conclude that $\Gamma \vdash \lambda x : B[\ell^{Ku}].\ t'_C : B[\ell^{Ku}] \to C[\ell^{Ku}]$. We proceed similarly for the reverse implication.

□

Lemma 6. *Let $A \in \kappa_3$. In the theory $(\Sigma^c_{HOL} \cup \Sigma, \mathscr{R}_{HOL} \cup \mathscr{R})$, for any context Γ, there exists some t such that $\Gamma \vdash t : A$ if and only if there exists some t' such that $\Gamma \vdash t' : A^{Ku}$.*

Proof. We proceed by induction on the term A using the fact that A is generated by κ_3. We use Lemma 5 and the double-negation elimination. □

We have shown that it is possible to build a proof of A in $\mathscr{T}^{Ku}$ using a proof of A^{Ku} and the principle of excluded middle. The next step is to derive a proof of A in the original theory $\mathscr{T}$. In particular, it requires to replace each use of $\lfloor \ell^{Ku} \rfloor \hookrightarrow r^{Ku} \in \mathscr{R}^{Ku}_{\mathscr{T}}$ by a use of $\ell \hookrightarrow r \in \mathscr{R}_{\mathscr{T}}$.

Lemma 7. *Let $A \in \kappa_3$ such that $\Gamma \vdash t : A[\ell^{Ku}]$. Using $\ell \hookrightarrow r$, there exists some t' such that $\Gamma \vdash t' : A[r^{Ku}]$.*

Proof. Using Lemma 5, there exists some t' such that $\Gamma \vdash t' : A[\ell]$. Using $\ell \hookrightarrow r$, we have $\Gamma \vdash t' : A[r]$. We use Lemma 5 to obtain some t'' such that $\Gamma \vdash t'' : A[r^{Ku}]$. □

Lemma 8. *Let $(\Sigma^c_{HOL} \cup \Sigma, \mathscr{R}_{HOL} \cup \mathscr{R}^{Ku})$ and $(\Sigma^c_{HOL} \cup \Sigma, \mathscr{R}_{HOL} \cup \mathscr{R})$ be two theories, abbreviated $\mathscr{R}^{Ku}$ and $\mathscr{R}$.*

- *If $\vdash \Gamma$ in $\mathscr{R}^{Ku}$ then $\vdash \Gamma$ in $\mathscr{R}$.*
- *If $\Gamma \vdash t : A$ in $\mathscr{R}^{Ku}$ and $A \in \kappa_i$ with $i \in \{1,2,4,5\}$, then $\Gamma \vdash t : A$ in $\mathscr{R}$.*
- *If $\Gamma \vdash t : A$ in $\mathscr{R}^{Ku}$ and $A \in \kappa_3$, then there exists some t' such that $\Gamma \vdash t' : A$ in $\mathscr{R}$.*

Proof. We proceed by induction on the typing derivation. We only present the relevant cases.

- <u>ABS</u>: Suppose that $\Gamma \vdash A : \texttt{TYPE}$ and $\Gamma, x : A \vdash B : s$ and $\Gamma, x : A \vdash t : B$ in $\mathscr{R}^{Ku}$. By induction we have $\Gamma \vdash A : \texttt{TYPE}$ and $\Gamma, x : A \vdash B : s$ in $\mathscr{R}$.

  If $B \in \kappa_i$ with $i \in \{1,2\}$, then by induction we have $\Gamma, x : A \vdash t : B$ in $\mathscr{R}$, and we derive $\Gamma \vdash \lambda x : A.\ t : \Pi x : A.\ B$ in $\mathscr{R}$.

  If $B \in \kappa_3$, then by induction we have $\Gamma, x : A \vdash t' : B$ in $\mathscr{R}$. We derive $\Gamma \vdash \lambda x : A.\ t' : \Pi x : A.\ B$ in $\mathscr{R}$.

- <u>APP</u>: Suppose that $\Gamma \vdash t : \Pi x : A.\ B$ and $\Gamma \vdash u : A$ in $\mathscr{R}^{Ku}$.

  If $\Pi x : A.\ B \in \kappa_i$ with $i \in \{1,2,4\}$, then by induction we have $\Gamma \vdash t : \Pi x : A.\ B$ and $\Gamma \vdash u : A$ in $\mathscr{R}$. We derive $\Gamma \vdash t\ u : B[x \leftarrow u]$ in $\mathscr{R}$.

  If $\Pi x : A.\ B \in \kappa_3$, then by induction we have $\Gamma \vdash t' : \Pi x : A.\ B$ in $\mathscr{R}$. If $A \in \kappa_i$ with $i \in \{1,2\}$, then by induction we have $\Gamma \vdash u : A$ in $\mathscr{R}$, and we derive $\Gamma \vdash t'\ u : B[x \leftarrow u]$ in $\mathscr{R}$. If $A \in \kappa_3$ (x does not occur in B), then by induction we have $\Gamma \vdash u' : A$ in $\mathscr{R}$, and we conclude that $\Gamma \vdash_c t'\ u' : B$.

- <u>CONV</u>: If $A \equiv_{\beta\mathscr{R}} B$ is obtained using β-conversion or the rewrite rules of $\mathscr{R}_{HOL}$, then we conclude using the induction hypothesis and the CONV rule. Otherwise, and without loss of generality, we consider that we only use one rewrite rule of $\mathscr{R}^{Ku}$ per CONV rule.

  Suppose that $A \equiv_{\beta\mathscr{R}} B$ is obtained using the rewrite rule $\ell^{Ku} \hookrightarrow r^{Ku} \in \mathscr{R}^{Ku}$. In that case, we have $A = C[\ell^{Ku}]$ and $B = C[r^{Ku}]$ (the case $A = C[r^{Ku}]$ and $B = C[\ell^{Ku}]$ is treated similarly). By assumption, we have $\Gamma \vdash t : C[\ell^{Ku}]$ and $\Gamma \vdash C[r^{Ku}] : s$ in R^{Ku}.

If $A, B \in \kappa_i$ with $i \in \{1,2,4,5\}$, then $\ell^{Ku} = \ell$ and $r^{Ku} = r$. By induction we have $\Gamma \vdash t : C[\ell^{Ku}]$ and $\Gamma \vdash C[r^{Ku}] : s$ in $\mathscr{R}$. We apply Conv with $C[\ell] \equiv_{\beta\mathscr{R}} C[r]$.

If $A, B \in \kappa_3$, then by induction we have $\Gamma \vdash t' : C[\ell^{Ku}]$ and $\Gamma \vdash C[r^{Ku}] : s$ in $\mathscr{R}$. We conclude using Lemma 7.

□

We now have all the tools to show that, for any intuitionistic proof of A^{Ku} in the translated theory $\mathscr{T}^{Ku}$, there exists a classical proof of A in the original theory $\mathscr{T}$.

Theorem 2. *Let $\mathscr{T}$ be a theory encoded in higher-order logic and $A \in \kappa_3$. If $\Gamma^{Ku} \vdash t : A^{Ku}$ in $\mathscr{T}^{Ku}$, then under the Kuroda equivalence there exists some term t' such that $\Gamma \vdash t' : A$ in $\mathscr{T}$.*

Proof. We directly have $\Gamma^{Ku} \vdash t : A^{Ku}$ in $(\Sigma^c_{HOL} \cup \Sigma^{Ku}_{\mathscr{T}}, \mathscr{R}_{HOL} \cup \mathscr{R}^{Ku}_{\mathscr{T}})$.

- By Lemma 6, there exists some t' such that $\Gamma^{Ku} \vdash t' : A$ in $(\Sigma^c_{HOL} \cup \Sigma^{Ku}_{\mathscr{T}}, \mathscr{R}_{HOL} \cup \mathscr{R}^{Ku}_{\mathscr{T}})$ and under the Kuroda equivalence.
- Using Lemma 8, there exists some t'' such that $\Gamma^{Ku} \vdash t'' : A$ in $(\Sigma^c_{HOL} \cup \Sigma^{Ku}_{\mathscr{T}}, \mathscr{R}_{HOL} \cup \mathscr{R}_{\mathscr{T}})$.
- We replace the signature $\Sigma^{Ku}_{\mathscr{T}}$ by $\Sigma_{\mathscr{T}}$. For each constant $c : C \in \Sigma_{\mathscr{T}}$ with $C \in \kappa_3$, we replace c by t_c (provided by Lemma 6) in t''. We obtain $\Gamma^{Ku} \vdash t''[c \leftarrow t_c] : A$ in $(\Sigma^c_{HOL} \cup \Sigma_{\mathscr{T}}, \mathscr{R}_{HOL} \cup \mathscr{R}_{\mathscr{T}})$, that is in $\mathscr{T}$. These substitutions work since c cannot occur in a dependent type.
- We replace the context Γ^{Ku} by Γ. For each variable $x : B \in \Gamma$ with $B \in \kappa_3$, we replace x by t_x (provided by Lemma 6) in $t''[c \leftarrow t_c]$. We obtain $\Gamma \vdash t''[c \leftarrow t_c][x \leftarrow t_x] : A$ in $\mathscr{T}$, which achieves the proof.

□

The extension of Kuroda's translation to the $\lambda\Pi$-calculus modulo theory is a generalization of Brown and Rizkallah's translation for simple type theory [4]. Indeed, if $\mathscr{R}_{\mathscr{T}} = \langle\rangle$, then we obtain the result in higher-order logic, at the only difference that proofs are represented by terms.

4 Construkti, an Implementation for Dedukti Proofs

Dedukti. The $\lambda\Pi$-calculus modulo theory has been implemented in the Dedukti proof language. Abstractions $\lambda x : A.\, t$ are represented by `x : A => t`, and dependent types $\Pi x : A.\, B$ are represented by `x : A -> B`. Constants $c : A$ are specified by `c : A`, prefixed with the keyword `def` if the constant can be defined using rewrite rules. Rewrite rules $\ell \hookrightarrow r$, where x and y are the free variables of ℓ and r, are represented by `[x,y] l --> r`. For instance, using the encoding of the notions of proposition and proof, we can encode the addition on natural numbers via rewrite rules.

```
nat : Set.
0 : El nat.
S : El nat -> El nat.
def add : El nat -> El nat -> El nat.
[x] add x 0 --> x.
[x, y] add x (S y) --> S (add x y).
```

Theorems are represented by `thm n : T := p`, where `n` is its name, `T` its statement and `p` its proof term. For checking that `p` is indeed a proof of `T`, we can use one of the type checkers of Dedukti, for instance DKCheck [19] or Lambdapi [15].

Construkti. We have implemented CONSTRUKTI[2], a tool that performs Kuroda's translation on DEDUKTI proofs. CONSTRUKTI takes as input a DEDUKTI file containing the specification of a user-defined theory encoded in higher-order logic, as well as proofs in this theory. It returns a DEDUKTI file containing the specification of the translated theory, as well as the translated proofs.

In this implementation, we insert one double negation after every *Prf* and $\forall$ symbols, and we replace the constants c representing natural deduction rules by the terms c^i. For instance, the constant $\mathsf{top_i}$ of type $\mathit{Prf}\ \top$, representing the introduction of tautology, is replaced in the formal proofs by the term $\mathsf{top_i}^i$ of type $\mathit{Prf}\ (\neg\neg\top)$. The proof term $\mathsf{top_i}^i$ relies on the proof of $\Pi p : \mathit{Prop}.\ \mathit{Prf}\ (p \Rightarrow \neg\neg p)$.

```
top_i : Prf top.

thm prop_double_neg : p : Prop -> Prf (imp p (not (not p)))
:= p => imp_i p (not (not p))
  (pP => neg_i (not p) (pNP => neg_e p pNP pP)).

thm top_i_i : Prf (not (not top))
:= imp_e top (not (not top)) (prop_double_neg top) top_i.
```

So as to obtain readable theorems, we directly β-reduce every application of Prf^{Ku} and $\forall^{Ku}$.

Benchmark. We have tested CONSTRUKTI on a benchmark of 101 DEDUKTI proofs, available in the file `hol-lib.dk`. These proofs encompass results related to connectives and quantifiers, classical formulas, De Morgan's laws, polymorphic equality, and basic arithmetic. The proofs are expressed in propositional, first-order and higher-order logics. This library of proofs includes user-defined rewrite rules—a feature of the $\lambda\Pi$-calculus modulo theory—and inference rules—thanks to the encoding of the notions of proposition and proof. We compare in Table 1 the different characteristics of the library: the number of proofs, the number of classical proofs, the number of results expressed in higher-order logic, and the number of results that are expressed via admissible inference rules.

Content of	Number of ...			
the library	proofs	classical proofs	higher-order results	admissible inference rules
Basic logic	38	0	15	26
Classical results	12	12	9	3
De Morgan	8	6	4	8
Equality	10	0	6	4
Arithmetic	33	0	0	16
All	101	18	34	57

Table 1: Comparison of the different libraries.

After running CONSTRUKTI, all the translated proofs of the translated theorems typecheck, and are expressed in intuitionistic logic.

[2]Available at `https://github.com/Deducteam/Construkti`.

5 Conclusion

In this paper, we have extended Kuroda's translation to the theories encoded in higher-logic in the $\lambda\Pi$-calculus modulo theory, that is λ-calculus extended with dependent types and user-defined rewrite rules. In this logical framework, proofs are terms following the Curry-Howard correspondence, and have to be effectively translated. Due to the encoding of the notions of proposition and proof in the $\lambda\Pi$-calculus modulo theory, we can assume, prove, and translate inference rules. We have implemented CONSTRUKTI, a tool that transforms DEDUKTI proofs following Kuroda's translation. Both DEDUKTI and CONSTRUKTI pave the way for interoperability between classical proof systems—such as HOL LIGHT or MIZAR—and intuitionistic proof systems—such as COQ, LEAN or AGDA.

Future work. There exist large libraries of proofs in higher-order logic, for instance the HOL LIGHT standard library. Blanqui [9] recently translated it to COQ via DEDUKTI, taking the excluded middle as an axiom. Future work would be to obtain an intuitionistic version of the HOL LIGHT standard library, by applying Kuroda's translation and CONSTRUKTI.

Related work. Double-negation translations aim at embedding classical logic into intuitionistic logic. As such, double-negation translations *always* transform classical proofs into intuitionistic ones, but they modify the formulas during the process. Proof constructivization aims at transforming classical proofs into intuitionistic ones *without* translating the formulas, but such a process does not necessarily succeed. Cauderlier [5] developed heuristics to constructivize proofs in DEDUKTI, via rewrite systems that try to remove instances of the principle of excluded middle or of the double-negation elimination. Gilbert [11] designed a constructivization algorithm for first-order logic, that was tested in DEDUKTI and works in practice for large fragments of first-order logic.

Acknowledgments

The author would like to thank Marc Aiguier, Gilles Dowek and Olivier Hermant for helpful discussions and valuable remarks about this work.

References

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory*. Manuscript.

[2] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet & François Thiré (2023): *A modular construction of type theories*. *Logical Methods in Computer Science* Volume 19, Issue 1, doi:10.46298/lmcs-19(1:12)2023. Available at https://lmcs.episciences.org/10959.

[3] Paul Brauner, Clement Houtmann & Claude Kirchner (2007): *Principles of Superdeduction*. In: *LICS 2007 - 22nd Annual IEEE Symposium on Logic in Computer Science*, Wroclaw, Poland, pp. 41–50, doi:10.1109/LICS.2007.37. Available at https://ieeexplore.ieee.org/abstract/document/4276550. ISSN: 1043-6871.

[4] Chad E. Brown & Christine Rizkallah (2014): *Glivenko and Kuroda for simple type theory*. *The Journal of Symbolic Logic* 79(2), pp. 485–495, doi:10.1017/jsl.2013.10. Available at http://www.jstor.org/stable/43303744.

[5] Raphaël Cauderlier (2016): *A Rewrite System for Proof Constructivization*. In: *LFMTP 2016 - International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Porto, Portugal, pp. 1 – 7, doi:`10.1145/2966268.2966270`. Available at `https://inria.hal.science/hal-01420634`.

[6] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo*. In Simona Ronchi Della Rocca, editor: *Typed Lambda Calculi and Applications*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 102–117, doi:`10.1007/978-3-540-73228-0_9`.

[7] Nachum Dershowitz & Jean-Pierre Jouannaud (1991): *Rewrite Systems*. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, doi:`10.1016/B978-0-444-88074-1.50011-1`.

[8] Gilles Dowek, Thérèse Hardin & Claude Kirchner (2003): *Theorem Proving Modulo*. *Journal of Automated Reasoning* 31, pp. 33–72, doi:`10.1023/A:1027357912519`.

[9] Frédéric Blanqui (2024): *Translating HOL-Light proofs to Coq*. In: *LPAR 2024 - 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Balaclava, Mauritius, pp. 1–18, doi:`10.29007/6k4x`. Available at `https://easychair.org/publications/paper/mtFT`.

[10] Gerhard Gentzen (1936): *Die Widerspruchsfreiheit der Reinen Zahlentheorie*. *Mathematische Annalen* 112, pp. 493–565, doi:`10.1007/BF01565428`.

[11] Frédéric Gilbert (2017): *Automated Constructivization of Proofs*. In: *FOSSACS 2017 - 20th International Conference on Foundations of Software Science and Computation Structures*, Uppsala, Sweden, pp. 480–495, doi:`10.1007/978-3-662-54458-7_28`.

[12] Valery Glivenko (1928): *Sur quelques points de la logique de M. Brouwer*. *Bulletins de la classe des sciences* 15, p. 183–188.

[13] Kurt Gödel (1933): *Zur intuitionistischen Arithmetik und Zahlentheorie*. *Ergebnisse eines Mathematischen Kolloquiums* 4, p. 34–38.

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

[15] Gabriel Hondet & Frédéric Blanqui (2020): *The New Rewriting Engine of Dedukti*. In: *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*, 167, Paris, France, p. 16, doi:`10.4230/LIPIcs.FSCD.2020.35`. Available at `https://inria.hal.science/hal-02981561`.

[16] Clément Houtmann (2010): *Représentation et interaction des preuves en superdéduction modulo*. Ph.D. thesis, Université Henri Poincaré - Nancy I. Available at `https://theses.hal.science/tel-00553219`.

[17] Andrey Nikolaevich Kolmogorov (1925): *O principe tertium non datur*. *Matematicheskiĭ Sbornik* 32, p. 646–667.

[18] Sigekatu Kuroda (1951): *Intuitionistische Untersuchungen der formalistischen Logik*. *Nagoya Mathematical Journal* 2, p. 35–47, doi:`10.1017/S0027763000010023`.

[19] Ronan Saillard (2015): *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice*. Ph.D. thesis, Ecole Nationale Supérieure des Mines de Paris. Available at `https://pastel.hal.science/tel-01299180`.

[20] François Thiré (2020): *Interoperability between proof systems using the logical framework Dedukti*. Ph.D. thesis, Université Paris-Saclay. Available at `https://hal.science/tel-03224039`.

[21] Thomas Traversié (2024): *Kuroda's translation for higher-order logic*. Available at `https://hal.science/hal-04561757`. Manuscript.

Proofs for Free in the $\lambda\Pi$-Calculus Modulo Theory

Thomas Traversié
Université Paris-Saclay, CentraleSupélec, MICS
Gif-sur-Yvette, France
Université Paris-Saclay, Inria, CNRS, ENS-Paris-Saclay, LMF
Gif-sur-Yvette, France
thomas.traversie@centralesupelec.fr

Parametricity allows the transfer of proofs between different implementations of the same data structure. The $\lambda\Pi$-calculus modulo theory is an extension of the λ-calculus with dependent types and user-defined rewrite rules. It is a logical framework, used to exchange proofs between different proof systems. We define an interpretation of theories of the $\lambda\Pi$-calculus modulo theory, inspired by parametricity. Such an interpretation allows to transfer proofs for free between theories that feature the notions of proposition and proof, when the source theory can be embedded into the target theory.

1 Introduction

Many proof assistants have been developed during the past decades, such as AGDA, COQ, HOL LIGHT, ISABELLE, LEAN or MIZAR. All those systems have their own theoretical foundations and proof language. If a library of proofs has been formalized in some proof assistant, one would ideally like to export it automatically to any other proof assistant. That is why the question of the *interoperability* between proof systems arises. Exchanging formal proofs between different proof systems strengthen reusability, re-checking and preservation of libraries. For this purpose, Cousineau and Dowek developed the $\lambda\Pi$-calculus modulo theory [8], that combines λ-calculus with dependent types and user-defined rewrite rules. It is a logical framework, in which theories are defined by typed constants and rewrite rules, specified by the users. Many theories can be expressed in the $\lambda\Pi$-calculus modulo theory [4], such as Predicate Logic, Simple Type Theory and the Calculus of Constructions. Most of all, theories from various proof assistants can be expressed in this logical framework. As a consequence, it can be used as a common framework for exchanging proofs between proof systems [17]. The $\lambda\Pi$-calculus modulo theory has been implemented in the concrete language DEDUKTI [1, 14] and in the LAMBDAPI proof assistant, which features user-friendly proof tactics.

The problem of the exchange of proofs also emerges when it comes to the different implementations of a same data structure. One would like to share the theorems proved for one implementation to all the other implementations of the same data structure, without additional efforts. One method to derive theorems for free is to use *parametricity*. Reynolds [16] originally introduced an abstraction theorem, stating that the different implementations of a polymorphic function behave similarly. Wadler [18] used this result to derive properties satisfied by polymorphic functions, depending on their types. In other words, all functions of the same abstract type satisfy the same theorems. Bernardy *et al.* [2, 3] later extended parametricity to Pure Type Systems. Keller and Lasson [15] investigated parametricity for the Calculus of Inductive Constructions, the language behind the COQ proof assistant. More recently, Cohen *et al.* [7] developed a parametricity framework and implemented TROCQ, a COQ plugin for proof transfer based on parametricity. The exchange of proofs—the very purpose of the $\lambda\Pi$-calculus modulo theory—is therefore an important application of the parametricity translations.

Transferring databases of proofs is relevant when working with related mathematical structures. For instance, if we have proved theorems in a theory of natural numbers and we want to use them in a theory of integers, we would like to export the proofs for non-negative integers. The same issue arises concerning various mathematical structures and databases of proofs, as we can *embed* natural numbers into reals, reals into reals extended with infinity elements, or sets into pointed graphs [5]. It would therefore be interesting to exchange proofs between theories of the $\lambda\Pi$-calculus modulo theory, when the source theory can be embedded into the target theory.

Contribution. In this paper, we define an interpretation of theories of the $\lambda\Pi$-calculus modulo theory, when they feature a prelude encoding of the notions of proposition and proof. Such an interpretation, inspired by parametricity, applies when we can embed the source theory $\mathbb{S}$ into the target theory $\mathbb{T}$. The interpretation depends on parameters, given by the user for representing each constant of the source theory by a term in the target theory. We provide the parameters necessary for interpreting the prelude encoding. We show that if $\mathbb{S}$ has an interpretation in $\mathbb{T}$, then the proofs written inside $\mathbb{S}$ can be transformed into proofs written inside $\mathbb{T}$. This interpretation comes with a relative consistency theorem: if $\mathbb{T}$ is consistent, then $\mathbb{S}$ is consistent too.

In order to illustrate this interpretation, we embed a theory of natural numbers into a theory of integers. This example, as well as the parameters for the prelude encoding, are given in DEDUKTI, and are available at `https://github.com/thomastraversie/InterpDK`.

Outline of the paper. In Section 2, we give a formal presentation of the $\lambda\Pi$-calculus modulo theory, and we detail a prelude encoding of the notions of proposition and proof. In Section 3, we define an interpretation of theories of the $\lambda\Pi$-calculus modulo theory. In particular, we specify the parameters required for interpreting the prelude encoding. We prove the interpretation theorem and the relative consistency theorem. At the end, we show how this interpretation can be used to derive theorems for free, taking the running example of natural numbers and integers.

2 Theories in the $\lambda\Pi$-Calculus Modulo Theory

In this section, we give a formal definition of the syntax and type system of the $\lambda\Pi$-calculus modulo theory. We present a standard way of expressing the notions of proposition and proof in it—called prelude encoding—and we emphasize the theories that will be considered in the rest of the paper.

2.1 The $\lambda\Pi$-Calculus Modulo Theory

The Edinburgh Logical Framework [13], also known as $\lambda\Pi$-calculus, is an extension of simply typed λ-calculus with dependent types. The $\lambda\Pi$-calculus modulo theory [8] is an extension of the Edinburgh Logical Framework, in which user-defined rewrite rules [9] have been added. Its syntax is given by:

Sorts	$s ::= \mathtt{TYPE} \mid \mathtt{KIND}$
Terms	$t,u,A,B ::= c \mid x \mid s \mid \Pi(x:A).\,B \mid \lambda(x:A).\,t \mid t\ u$
Contexts	$\Gamma ::= \langle\rangle \mid \Gamma, x:A$
Signatures	$\Sigma ::= \langle\rangle \mid \Sigma, c:A \mid \Sigma, \ell \hookrightarrow r$

where c is a constant and x is a variable (ranging over disjoint sets), $\Pi(x:A).\,B$ is a dependent product (simply written $A \to B$ if x does not occur in B), $\lambda(x:A).\,t$ is an abstraction, and $t\ u$ is an application. For convenience, $\lambda(x_1 : A_1).\dots\lambda(x_n : A_n).\,t$ is written $\lambda(x_1 : A_1)\dots(x_n : A_n).\,t$ and $\Pi(x_1 : A_1).\dots\Pi(x_n : A_n).\,B$ is written $\Pi(x_1 : A_1)\dots(x_n : A_n).\,B$. Terms of type $\mathtt{TYPE}$ are called types, and terms of type $\mathtt{KIND}$ are called kinds. Signatures and contexts are finite sequences, and are written $\langle\rangle$ when empty. The $\lambda\Pi$-calculus modulo theory is a logical framework, in which Σ is fixed by the users depending on the theory they are working in. Signatures are composed of typed constants $c : A$ (such that A is a closed term, that is a term with no free variables) and rewrite rules $\ell \hookrightarrow r$ (such that the head-symbol of ℓ is a constant). The relation $\hookrightarrow_{\beta\Sigma}$ is the smallest relation, closed by context, such that if t rewrites to u for some rule in Σ or by β-reduction, then $t \hookrightarrow_{\beta\Sigma} u$. The conversion $\equiv_{\beta\Sigma}$ is the reflexive, symmetric, and transitive closure of the relation $\hookrightarrow_{\beta\Sigma}$.

$$\frac{}{\vdash \langle\rangle}\,[\textsc{Empty}] \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathtt{TYPE}}{\vdash \Gamma, x : A}\,[\textsc{Decl}]\ x \notin \Gamma \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE} : \mathtt{KIND}}\,[\textsc{Sort}]$$

$$\frac{\vdash \Gamma \qquad \vdash A : s}{\Gamma \vdash c : A}\,[\textsc{Const}]\ c : A \in \Sigma \qquad \frac{\vdash \Gamma}{\Gamma \vdash x : A}\,[\textsc{Var}]\ x : A \in \Gamma$$

$$\frac{\Gamma \vdash A : \mathtt{TYPE} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi(x : A).\,B : s}\,[\textsc{Prod}] \qquad \frac{\Gamma \vdash A : \mathtt{TYPE} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).\,t : \Pi(x : A).\,B}\,[\textsc{Abs}]$$

$$\frac{\Gamma \vdash t : \Pi(x : A).\,B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \leftarrow u]}\,[\textsc{App}] \qquad \frac{\Gamma \vdash t : A \qquad \vdash B : s}{\Gamma \vdash t : B}\,[\textsc{Conv}]\ A \equiv_{\beta\Sigma} B$$

Figure 1: Typing rules of the $\lambda\Pi$-calculus modulo theory.

The judgment $\vdash \Gamma$ means that the context Γ is well-formed, and $\Gamma \vdash t : A$ means that t is of type A in the context Γ. When the context is empty, we simply write $\vdash t : A$. The typing rules for the $\lambda\Pi$-calculus modulo theory are given in Figure 1. The standard weakening rule is admissible.

A signature is a theory when its rewrite rules satisfy certain properties. We write Λ_Σ for the set of terms whose constants belong to Σ.

Definition 1 (Theory)**.** *A theory $\mathbb{T}$ in the $\lambda\Pi$-calculus modulo theory is given by a signature Σ such that:*

1. *for each rule $\ell \hookrightarrow r \in \Sigma$, we have ℓ and r in Λ_Σ,*
2. *$\hookrightarrow_{\beta\Sigma}$ is confluent on Λ_Σ,*
3. *for each rule $\ell \hookrightarrow r \in \Sigma$, for all context Γ, term $A \in \Lambda_\Sigma$ and substitution θ, if $\Gamma \vdash \ell\theta : A$ then $\Gamma \vdash r\theta : A$.*

Lemma 1. *If $\Gamma \vdash t : A$, then either $A = \mathtt{KIND}$ or $\Gamma \vdash A : s$ for $s = \mathtt{TYPE}$ or $s = \mathtt{KIND}$. If $\Gamma \vdash \Pi(x : A).\,B : s$, then $\Gamma \vdash A : \mathtt{TYPE}$.*

2.2 A Prelude Encoding

It is possible to formalize the notions of proposition and proof in the $\lambda\Pi$-calculus modulo theory [4]. In particular, this encoding—called prelude encoding—gives the possibility to quantify over certain propositions through codes, which is not possible inside the standard $\lambda\Pi$-calculus modulo theory. This encoding is defined by the following signature, written Σ_{pre}.

$$\begin{array}{ll}
Set : \texttt{TYPE} & o : Set \\
El : Set \to \texttt{TYPE} & Prf : El\ o \to \texttt{TYPE} \\
\leadsto_d : \Pi(x : Set).\ (El\ x \to Set) \to Set & \Rightarrow_d : \Pi(x : El\ o).\ (Prf\ x \to El\ o) \to El\ o \\
El\ (x \leadsto_d y) \hookrightarrow \Pi(z : El\ x).\ El\ (y\ z) & Prf\ (x \Rightarrow_d y) \hookrightarrow \Pi(z : Prf\ x).\ Prf\ (y\ z) \\
\pi : \Pi(x : El\ o).\ (Prf\ x \to Set) \to Set & \forall : \Pi(x : Set).\ (El\ x \to El\ o) \to El\ o \\
El\ (\pi\ x\ y) \hookrightarrow \Pi(z : Prf\ x).\ El\ (y\ z) & Prf\ (\forall\ x\ y) \hookrightarrow \Pi(z : El\ x).\ Prf\ (y\ z)
\end{array}$$

We declare the constant *Set*, which represents the universe of sorts, along with the injection *El* that maps terms of type *Set* to the type of its elements. We define a sort o, such that $El\ o$ corresponds to the universe of propositions. The injection *Prf* maps propositions to the type of its proof. In other words, a term P of type $El\ o$ is a proposition, and a term of type $Prf\ P$ is a proof of P. The infix symbol $\leadsto_d$ (respectively $\Rightarrow_d$) is used to represent dependent function types between terms of type *Set* (respectively $El\ o$). Remark that the symbols $\leadsto_d$ and $\Rightarrow_d$ are generalizations of the usual functionality $\leadsto$ and implication $\Rightarrow$ in the case of dependent types. The symbol π (respectively $\forall$) is used to represent dependent function types between elements of type $El\ o$ and *Set* (respectively *Set* and $El\ o$).

While it is not possible to quantify over TYPE in the $\lambda\Pi$-calculus modulo theory, this encoding allows to quantify over propositions—objects of type $El\ o$—and then inject them into TYPE using *Prf*. Similarly, we can quantify over sorts—objects of type *Set*—and then inject them into TYPE using *El*.

2.3 Theories with Prelude Encoding

In this paper, we consider theories that feature those basic notions of proposition and proof. More formally, we take theories of the form $\mathbb{T} = \Sigma_{pre} \cup \Sigma_{\mathbb{T}}$, where the user-defined constants $c : A \in \Sigma_{\mathbb{T}}$ have to be expressed in the prelude encoding.

Definition 2 (Theories with prelude encoding). *We say that a theory $\mathbb{T} = \Sigma_{pre} \cup \Sigma_{\mathbb{T}}$ is a theory with prelude encoding when for every $c : A \in \Sigma_{\mathbb{T}}$, we have $\vdash A : \texttt{TYPE}$.*

The condition guarantees that the user-defined constants of $\Sigma_{\mathbb{T}}$ are indeed encoded in the prelude encoding. For instance, we cannot define nat : TYPE, but are forced to take nat : *Set*. Consequently inside a theory with prelude encoding, the only constants $c : A \in \Sigma$ with A a kind are *Set* (of type TYPE), *El* (of type $Set \to$ TYPE) and *Prf* (of type $El\ o \to$ TYPE).

For each rewrite rule $\ell \hookrightarrow r \in \Sigma$, the head-symbol of ℓ is a constant. It follows that, if $\Gamma \vdash \ell : A$, then A cannot be KIND. We thus have $\Gamma \vdash A : s$ with $s =$ TYPE or $s =$ KIND. In particular, TYPE cannot occur in ℓ and r.

Example 1 (Natural numbers). *We define a theory with prelude encoding $\mathbb{T}_n = \Sigma_{pre} \cup \Sigma_n$ for natural numbers.* nat *is the sort of natural numbers. We declare two constructors 0_n and* succ_n*, a relation $\geq_n$,* *and an induction principle* rec_n.

$$\begin{array}{ll}
\mathsf{nat}: & \mathit{Set} \\
0_n: & \mathit{El}\ \mathsf{nat} \\
\mathsf{succ}_n: & \mathit{El}\ \mathsf{nat} \to \mathit{El}\ \mathsf{nat} \\
\geq_n\ : & \mathit{El}\ \mathsf{nat} \to \mathit{El}\ \mathsf{nat} \to \mathit{El}\ o \\
\mathsf{ax}_n^1: & \Pi(x : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (x \geq_n x) \\
\mathsf{ax}_n^2: & \Pi(x : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (\mathsf{succ}_n\ x \geq_n x) \\
\mathsf{ax}_n^3: & \Pi(x,y,z : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (x \geq_n y) \to \mathit{Prf}\ (y \geq_n z) \to \mathit{Prf}\ (x \geq_n z) \\
\mathsf{rec}_n: & \Pi(P : \mathit{El}\ \mathsf{nat} \to \mathit{El}\ o).\ \mathit{Prf}\ (P\ 0_n) \to \\
 & [\Pi(x : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ (\mathsf{succ}_n\ x))] \to \\
 & \Pi(x : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (P\ x)
\end{array}$$

In this theory, we can prove $\Pi(x : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (x \geq_n 0_n)$ *and* $\Pi(x : \mathit{El}\ \mathsf{nat}).\ \mathit{Prf}\ (\mathsf{succ}_n\ x \geq_n 0_n)$.

Example 2 (Integers)**.** *We define a theory with prelude encoding* $\mathbb{T}_i = \Sigma_{pre} \cup \Sigma_i$ *for integers.* int *is the sort of integers. We declare three constructors* 0_i, succ_i *and* pred_i, *a relation* $\geq_i$ *and a generalized induction principle* rec_i.

$$\begin{array}{ll}
\mathsf{int}: & \mathit{Set} \\
0_i: & \mathit{El}\ \mathsf{int} \\
\mathsf{succ}_i: & \mathit{El}\ \mathsf{int} \to \mathit{El}\ \mathsf{int} \\
\mathsf{pred}_i: & \mathit{El}\ \mathsf{int} \to \mathit{El}\ \mathsf{int} \\
\geq_i\ : & \mathit{El}\ \mathsf{int} \to \mathit{El}\ \mathsf{int} \to \mathit{El}\ o \\
\mathsf{ax}_i^1: & \Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (x \geq_i x) \\
\mathsf{ax}_i^2: & \Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (\mathsf{succ}_i\ x \geq_i x) \\
\mathsf{ax}_i^3: & \Pi(x,y,z : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (x \geq_i y) \to \mathit{Prf}\ (y \geq_i z) \to \mathit{Prf}\ (x \geq_i z) \\
\mathsf{ax}_i^4: & \Pi(x : \mathit{El}\ \mathsf{int}).\ \Pi(P : \mathit{El}\ \mathsf{int} \to \mathit{El}\ o).\ \mathit{Prf}\ (P\ (\mathsf{succ}_i\ (\mathsf{pred}_i\ x))) \to \mathit{Prf}\ (P\ x) \\
\mathsf{ax}_i^5: & \Pi(x : \mathit{El}\ \mathsf{int}).\ \Pi(P : \mathit{El}\ \mathsf{int} \to \mathit{El}\ o).\ \mathit{Prf}\ (P\ (\mathsf{pred}_i\ (\mathsf{succ}_i\ x))) \to \mathit{Prf}\ (P\ x) \\
\mathsf{rec}_i: & \Pi(c : \mathit{El}\ \mathsf{int})(P : \mathit{El}\ \mathsf{int} \to \mathit{El}\ o).\ \mathit{Prf}\ (P\ c) \to \\
 & [\Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (x \geq_i c) \to \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ (\mathsf{succ}_i\ x))] \to \\
 & \Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (x \geq_i c) \to \mathit{Prf}\ (P\ x)
\end{array}$$

In this theory, we cannot prove $\Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (x \geq_i 0_i)$ *and* $\Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (\mathsf{succ}_i\ x \geq_i 0_i)$, *but we can prove* $\Pi(x : \mathit{El}\ \mathsf{int}).\ \mathit{Prf}\ (x \geq_i 0_i) \to \mathit{Prf}\ (\mathsf{succ}_i\ x \geq_i 0_i)$.

3 Interpretation in the $\lambda\Pi$-Calculus Modulo Theory

In this section, we define an interpretation of theories with prelude encoding. To do so, we first define the interpretation for the terms of the $\lambda\Pi$-calculus modulo theory, and then we extend it to theories with prelude encoding. Such an interpretation requires external parameters. In particular, we provide the parameters necessary for interpreting the prelude encoding. We show how the interpretation of a source theory $\mathbb{S}$ in a target theory $\mathbb{T}$ can be used to derive in $\mathbb{T}$ the theorems proved in $\mathbb{S}$. We conclude with an example: we provide the formal parameters for interpreting the theory of natural numbers $\mathbb{T}_n$ in the theory of integers $\mathbb{T}_i$.

3.1 Interpretation of Terms

Intuition. When we interpret the source theory $\mathbb{S}$ in the target theory $\mathbb{T}$, we want to represent every term t of $\mathbb{S}$ by a term t^* in $\mathbb{T}$, such that if t is of type A in $\mathbb{S}$ then t^* is of type A^* in $\mathbb{T}$. For instance, when interpreting the theory of natural numbers $\mathbb{T}_n$ in the theory of integers $\mathbb{T}_i$, we have to represent $El\ \mathsf{nat}$ by $(El\ \mathsf{nat})^*$. We would like to take $(El\ \mathsf{nat})^* := \Sigma(z : El\ \mathsf{int}).\ Prf\ (z \geq_i 0_i)$. However, the $\lambda\Pi$-calculus modulo theory does not feature Σ-types, and it is therefore difficult to express $(El\ \mathsf{nat})^*$ in $\mathbb{T}_i$.

An alternative is to interpret the type of natural numbers $El\ \mathsf{nat}$ by the type of integers $El\ \mathsf{int}$, but we must guarantee that every integer representing a natural number is indeed non-negative. We naturally interpret the sort nat by int, 0_n by 0_i, succ_n by succ_i, and $\geq_n$ by $\geq_i$. The interpretation of the theorem $\Pi(x : El\ \mathsf{nat}).\ Prf\ (\mathsf{succ}_n\ x \geq_n 0_n)$ should not be $\Pi(x^* : El\ \mathsf{int}).\ Prf\ (\mathsf{succ}_i\ x^* \geq_i 0_i)$, which is generally false for integers. Instead, we must ensure that x^* is an integer corresponding to a natural number, meaning that we suppose a proof of $Prf\ (x^* \geq_i 0_i)$. Thus the interpretation of the theorem $\Pi(x : El\ \mathsf{nat}).\ Prf\ (\mathsf{succ}_n\ x \geq_n 0_n)$ should be $\Pi(x^* : El\ \mathsf{int}).\ Prf\ (x^* \geq_i 0_i) \to Prf\ (\mathsf{succ}_i\ x^* \geq_i 0_i)$.

Formal definition. Following this intuition, when interpreting a term t of type A in $\mathbb{S}$ by a term t^* of type A^* in $\mathbb{T}$, we must take into account that A^* is a type that encompasses A, but may be larger than A. In that respect, we introduce another term t^+ of type $A^+\ t^*$, where A^+ is a predicate asserting that a given object of type A^* satisfies the semantic of type A.

The interpretation of every constant c is given by two parameters c^* and c^+. The translation of an application $(t\ u)^*$ is $t^*\ u^*\ u^+$, since t^* takes as arguments u^* but also the witness u^+. Similarly, $(t\ u)^+$ is given by $t^+\ u^*\ u^+$. If the variable x occurs in t, then x^* and x^+ may occur in t^* and t^+. Hence $(\lambda(x:A).\ t)^*$ is given by $\lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^*$ and $(\lambda(x:A).\ t)^+$ is given by $\lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^+$.

The same intuition holds for dependent types $(\Pi(x:A).\ B)^*$. The predicate $(\Pi(x:A).\ B)^+$ asserts that an object f of type $(\Pi(x:A).\ B)^*$ corresponds to the semantic of $\Pi(x:A).\ B$. In other words, for every x^* of type A^* and x^+ of type $A^+\ x^*$, the term $f\ x^*\ x^+$ should satisfy the predicate B^+. When B is of type $\mathtt{TYPE}$, we take $(\Pi(x:A).\ B)^+ := \lambda(f : (\Pi(x:A).\ B)^*).\ \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\ (f\ x^*\ x^+)$. However, we cannot do the same when B is of type $\mathtt{KIND}$, because this term would be ill-typed. Indeed, $(\Pi(x:A).\ B)^*$ has type $\mathtt{KIND}$, while the type of the bound variable f must have type $\mathtt{TYPE}$. To get around this issue, we introduce metavariables. We write $T\{X\}$ when the metavariable X occurs in T, and we write $T\{t\}$ for the term obtained when substituting X by t in T. When B has type $\mathtt{KIND}$, we take $(\Pi(x:A).\ B)^+\{X\} := \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\{X\ x^*\ x^+\}$. Metavariables are only used for this purpose. In particular, they are always substituted and they never appear in typed terms.

Definition 3 (Interpretation of terms)**.** *The interpretation of terms of the $\lambda\Pi$-calculus modulo theory is given by the function $t \mapsto t^*$ defined inductively by*

$(x)^* := x^*$ *(variable)*
$(c)^* := c^*$ *(parameter)*
$\mathtt{TYPE}^* := \mathtt{TYPE}$
$\mathtt{KIND}^* := \mathtt{KIND}$
$(t\ u)^* := t^*\ u^*\ u^+$
$(\lambda(x:A).\ t)^* := \lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^*$
$(\Pi(x:A).\ B)^* := \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^*$

and by the function $t \mapsto t^+$ defined inductively by

$(x)^+ := x^+$ *(variable)*
$(c)^+ := c^+$ *(parameter)*
$\mathtt{TYPE}^+\{X\} := X \to \mathtt{TYPE}$
$\mathtt{KIND}^+ := \mathtt{KIND}$
$(t\ u)^+ := t^+\ u^*\ u^+$
$(\lambda(x:A).\ t)^+ := \lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^+$
$(\Pi(x:A).\ B)^+ := \lambda(f : (\Pi(x:A).\ B)^*).\ \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\ (f\ x^*\ x^+)$ *if* $B : \mathtt{TYPE}$
$(\Pi(x:A).\ B)^+\{X\} := \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\{X\ x^*\ x^+\}$ *if* $B : \mathtt{KIND}$.

where the X is a metavariable. The interpretation is extended to contexts with

$\langle\rangle^{*,+} := \langle\rangle$
$(\Gamma, x : A)^{*,+} := \Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^*$.

When the free variable x occurs in t, then x^* and x^+ may both occur in t^* and t^+. As such, we do not define distinct translations Γ^* and Γ^+, but a single translation $\Gamma^{*,+}$, such that if $(x : A) \in \Gamma$ then $(x^* : A^*) \in \Gamma^{*,+}$ and $(x^+ : A^+\ x^*) \in \Gamma^{*,+}$.

Parametricity. Remark that our interpretation is intuitively related to the parametricity translation [2]. Using parametricity, the translation $(t\ u)^*$ is given by $t^*\ u^*$, the translation $(\lambda(x : A).\ t)^*$ is given by $\lambda(x^* : A^*).\ t^*$, and the translation $(\Pi(x : A).\ B)^*$ is given by $\Pi(x^* : A^*).\ B^*$. In our interpretation, we focus on embeddings and we want to represent every type A of the source theory by a type A^* of the target theory. While Σ-types are well-suited for expressing such A^*, they are not defined in the $\lambda\Pi$-calculus modulo theory. That is why we have applied a *currying* operation on Σ-types. We therefore represent type A using a more general type A^*, and we guarantee that each term of type A^* representing a term of type A enjoys the predicate A^+. Consequently, the translation $(\Pi(x : A).\ B)^*$ is given by $\Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^*$, the translation $(\lambda(x : A).\ t)^*$ is given by $\lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^*$, and the translation $(t\ u)^*$ is given by $t^*\ u^*\ u^+$. The formal relation between the parametricity translation and our interpretation remains to be investigated.

3.2 Parameters for the Prelude Encoding

We aim at interpreting a source theory $\mathbb{S}$ in a target theory $\mathbb{T}$, when $\mathbb{S}$ and $\mathbb{T}$ are theories with prelude encoding. Such an interpretation is parametrized by the terms of $\mathbb{T}$ that correspond to the constants of $\mathbb{S}$. In particular, we have to provide the parameters for the constants of the prelude encoding.

When $\vdash t : A$ in $\mathbb{S}$, we want to have $\vdash t^* : A^*$ in $\mathbb{T}$. Moreover, we want $\vdash A^+ : A^* \to \mathtt{TYPE}$ in $\mathbb{T}$ when $A = \mathtt{TYPE}$. These conditions lead to the definition of Set^*, Set^+, El^*, El^+, Prf^*, Prf^+ and o^*. When t is of type $Prf\ p$, we need a witness t^+ of type $(Prf\ p)^+\ t^*$ asserting that t^* is indeed a proof of p^*. Since t^* is of type $Prf\ p^*$, it is necessarily a proof of p^*, and we define Prf^+ so that we can always choose t^+ to be t^*. The predicate o^+ asserts that an object p^* of type $El\ o$ is indeed a proposition, so we choose o^+ to be $\lambda(z : El\ o).\ z \Rightarrow_d (\lambda(x : Prf\ z).\ z)$. Consequently, it is is always possible to find a witness p^+ of type

$\mathit{Prf}\ (o^+\ p^*)$, that is $\mathit{Prf}\ p^* \to \mathit{Prf}\ p^*$.

$$\begin{array}{l}
\mathit{Set}^* := \mathit{Set}\\
\mathit{Set}^+ := \lambda(x : \mathit{Set}).\ \mathit{El}\ x \to \mathit{El}\ o\\
o^* := o\\
o^+ := \lambda(z : \mathit{El}\ o).\ z \Rightarrow_d (\lambda(x : \mathit{Prf}\ z).\ z)\\
\mathit{El}^* := \lambda(x^* : \mathit{Set})(x^+ : \mathit{El}\ x^* \to \mathit{El}\ o).\ \mathit{El}\ x^*\\
\mathit{El}^+ := \lambda(u^* : \mathit{Set})(u^+ : \mathit{El}\ u^* \to \mathit{El}\ o)(x : \mathit{El}\ u^*).\ \mathit{Prf}\ (u^+\ x)\\
\mathit{Prf}^* := \lambda(x^* : \mathit{El}\ o)(x^+ : \mathit{Prf}\ (o^+\ x^*)).\ \mathit{Prf}\ x^*\\
\mathit{Prf}^+ := \lambda(u^* : \mathit{El}\ o)(u^+ : \mathit{Prf}\ (o^+\ u^*))(x : \mathit{Prf}\ u^*).\ \mathit{Prf}\ u^*
\end{array}$$

Parameters $\rightsquigarrow_d{}^*$ and $\rightsquigarrow_d{}^+$ are defined so that $(\mathit{El}\ (a \rightsquigarrow_d b))^@ \equiv_{\beta\Sigma} (\Pi(x : \mathit{El}\ a).\ \mathit{El}\ (b\ x))^@$ for $@ \in \{*, +\}$.

$$\begin{array}{ll}
\rightsquigarrow_d{}^* := & \lambda(a^* : \mathit{Set})(a^+ : \mathit{El}\ a^* \to \mathit{El}\ o)(b^* : \Pi(x^* : \mathit{El}\ a^*).\ \mathit{Prf}\ (a^+\ x^*) \to \mathit{Set}).\\
& \lambda(b^+ : \Pi(x^* : \mathit{El}\ a^*)(x^+ : \mathit{Prf}\ (a^+\ x^*)).\ \mathit{El}\ (b^*\ x^*\ x^+) \to \mathit{El}\ o).\\
& a^* \rightsquigarrow_d (\lambda(x^* : \mathit{El}\ a^*).\ \pi\ (a^+\ x^*)\ (b^*\ x^*))\\
\rightsquigarrow_d{}^+ := & \lambda(a^* : \mathit{Set})(a^+ : \mathit{El}\ a^* \to \mathit{El}\ o)(b^* : \Pi(x^* : \mathit{El}\ a^*).\ \mathit{Prf}\ (a^+\ x^*) \to \mathit{Set}).\\
& \lambda(b^+ : \Pi(x^* : \mathit{El}\ a^*)(x^+ : \mathit{Prf}\ (a^+\ x^*)).\ \mathit{El}\ (b^*\ x^*\ x^+) \to \mathit{El}\ o).\\
& \lambda(f : \mathit{El}\ (a \rightsquigarrow_d b)^*).\\
& \forall\ a^*\ (\lambda(x^* : \mathit{El}\ a^*).\ (a^+\ x^*) \Rightarrow_d (\lambda(x^+ : \mathit{Prf}\ (a^+\ x^*)).\ b^+\ x^*\ x^+\ (f\ x^*\ x^+)))
\end{array}$$

Parameter $\Rightarrow_d{}^*$ is defined so that $(\mathit{Prf}\ (a \Rightarrow_d b))^* \equiv_{\beta\Sigma} (\Pi(x : \mathit{Prf}\ a).\ \mathit{Prf}\ (b\ x))^*$. Because the condition $(\mathit{Prf}\ (a \Rightarrow_d b))^+ \equiv_{\beta\Sigma} (\Pi(x : \mathit{Prf}\ a).\ \mathit{Prf}\ (b\ x))^+$ holds regardless of the definition of $\Rightarrow_d{}^+$, we choose $\Rightarrow_d{}^+$ so that $\vdash \Rightarrow_d{}^+ : (\Pi(a : \mathit{El}\ o).\ (\mathit{Prf}\ a \to \mathit{El}\ o) \to \mathit{El}\ o)^+ \Rightarrow_d{}^*$.

$$\begin{array}{ll}
\Rightarrow_d{}^* := & \lambda(a^* : \mathit{El}\ o)(a^+ : \mathit{Prf}\ (o^+\ a^*))(b^* : \Pi(x^* : \mathit{Prf}\ a^*).\ \mathit{Prf}\ a^* \to \mathit{El}\ o).\\
& \lambda(b^+ : \Pi(x^* : \mathit{Prf}\ a^*)(x^+ : \mathit{Prf}\ a^*).\ \mathit{Prf}\ (o^+\ (b^*\ x^*\ x^+))).\\
& a^* \Rightarrow_d (\lambda(x^* : \mathit{Prf}\ a^*).\ a^* \Rightarrow_d\ (b^*\ x^*))\\
\Rightarrow_d{}^+ := & \lambda(a^* : \mathit{El}\ o)(a^+ : \mathit{Prf}\ (o^+\ a^*))(b^* : \Pi(x^* : \mathit{Prf}\ a^*).\ \mathit{Prf}\ a^* \to \mathit{El}\ o).\\
& \lambda(b^+ : \Pi(x^* : \mathit{Prf}\ a^*)(x^+ : \mathit{Prf}\ a^*).\ \mathit{Prf}\ (o^+\ (b^*\ x^*\ x^+))).\\
& \lambda(p : \mathit{Prf}\ (a \Rightarrow_d b)^*).\ p
\end{array}$$

Parameters π^* and π^+ are defined so that $(\mathit{El}\ (\pi\ a\ b))^@ \equiv_{\beta\Sigma} (\Pi(x : \mathit{Prf}\ a).\ \mathit{El}\ (b\ x))^@$ for $@ \in \{*, +\}$.

$$\begin{array}{ll}
\pi^* := & \lambda(a^* : \mathit{El}\ o)(a^+ : \mathit{Prf}\ (o^+\ a^*))(b^* : \Pi(x^* : \mathit{Prf}\ a^*).\ \mathit{Prf}\ a^* \to \mathit{Set}).\\
& \lambda(b^+ : \Pi(x^* : \mathit{Prf}\ a^*)(x^+ : \mathit{Prf}\ a^*).\ \mathit{El}\ (b^*\ x^*\ x^+) \to \mathit{El}\ o).\\
& \pi\ a^*\ (\lambda(x^* : \mathit{Prf}\ a^*).\ \pi\ a^*\ (b^*\ x^*))\\
\pi^+ := & \lambda(a^* : \mathit{El}\ o)(a^+ : \mathit{Prf}\ (o^+\ a^*))(b^* : \Pi(x^* : \mathit{Prf}\ a^*).\ \mathit{Prf}\ a^* \to \mathit{Set}).\\
& \lambda(b^+ : \Pi(x^* : \mathit{Prf}\ a^*)(x^+ : \mathit{Prf}\ a^*).\ \mathit{El}\ (b^*\ x^*\ x^+) \to \mathit{El}\ o).\\
& \lambda(f : \mathit{El}\ (\pi\ a\ b)^*).\\
& a^* \Rightarrow_d\ (\lambda(x^* : \mathit{Prf}\ a^*).\ a^* \Rightarrow_d (\lambda(x^+ : \mathit{Prf}\ a^*).\ b^+\ x^*\ x^+\ (f\ x^*\ x^+)))
\end{array}$$

Parameter $\forall^*$ is defined so that $(\mathit{Prf}\ (\forall\ a\ b))^* \equiv_{\beta\Sigma} (\Pi(x : \mathit{El}\ a).\ \mathit{Prf}\ (b\ x))^*$. Because the condition $(\mathit{Prf}\ (\forall\ a\ b))^+ \equiv_{\beta\Sigma} (\Pi(x : \mathit{El}\ a).\ \mathit{Prf}\ (b\ x))^+$ holds regardless of the definition of $\forall^+$, we choose $\forall^+$ so that $\vdash \forall^+ : (\Pi(a : \mathit{Set}).\ (\mathit{El}\ a \to \mathit{El}\ o) \to \mathit{El}\ o)^+\ \forall^*$.

$$\begin{array}{ll}
\forall^* := & \lambda(a^* : \mathit{Set})(a^+ : \mathit{El}\ a^* \to \mathit{El}\ o)(b^* : \Pi(x^* : \mathit{El}\ a^*).\ \mathit{Prf}\ (a^+\ x^*) \to \mathit{El}\ o).\\
& \lambda(b^+ : \Pi(x^* : \mathit{El}\ a^*)(x^+ : \mathit{Prf}\ (a^+\ x^*)).\ \mathit{Prf}\ (o^+\ (b^*\ x^*\ x^+))).\\
& \forall\ a^*\ (\lambda(x^* : \mathit{El}\ a^*).\ (a^+\ x^*) \Rightarrow_d (b^*\ x^*))
\end{array}$$

$$\begin{aligned}\forall^+ := \;& \lambda(a^* : Set)(a^+ : El\ a^* \to El\ o)(b^* : \Pi(x^* : El\ a^*).\ Prf\ (a^+\ x^*) \to El\ o).\\ & \lambda(b^+ : \Pi(x^* : El\ a^*)(x^+ : Prf\ (a^+\ x^*)).\ Prf\ (o^+\ (b^*\ x^*\ x^+))).\\ & \lambda(p : Prf\ (\forall\ a\ b)^*).\ p\end{aligned}$$

The parameters chosen for the constants of the prelude encoding satisfy the expected properties. For any $c : A \in \Sigma_{pre}$, we have $\vdash c^* : A^*$ and $\vdash c^+ : A^+\ c^*$. Moreover, the interpretation respects the conversion relation, meaning that for each rewrite rule $\ell \hookrightarrow r$ of Σ_{pre}, we have both $\ell^* \equiv_{\beta\Sigma} r^*$ and $\ell^+ \equiv_{\beta\Sigma} r^+$.

Proposition 1. *Let $c : A \in \Sigma_{pre}$.*

1. *We have $\vdash c^* : A^*$.*
2. (a) *If $\vdash A :$ `TYPE` then $\vdash c^+ : A^+\ c^*$.*
   (b) *If $\vdash A :$ `KIND` then $\vdash c^+ : A^+\{c^*\}$.*

Proof. By simple verification. The result has been checked in DEDUKTI, see the definitions of the parameters in the file `lo_sp.dk`[1]. ☐

Proposition 2. *For every $\ell \hookrightarrow r \in \Sigma_{pre}$, we have $\ell^* \equiv_{\beta\Sigma} r^*$ and $\ell^+ \equiv_{\beta\Sigma} r^+$.*

Proof. We only show the case $El\ (a \leadsto_d b) \hookrightarrow \Pi(x : El\ a).\ El\ (b\ x)$.

$$\begin{aligned}\text{We have } (El\ (a \leadsto_d b))^* &\equiv_{\beta\Sigma} El\ (a \leadsto_d b)^*\\ &\equiv_{\beta\Sigma} El\ (a^* \leadsto_d (\lambda x^*.\ \pi\ (a^+\ x^*)\ (b^*\ x^*)))\\ &\equiv_{\beta\Sigma} \Pi(x^* : El\ a^*).\ El\ (\pi\ (a^+\ x^*)\ (b^*\ x^*))\\ &\equiv_{\beta\Sigma} \Pi(x^* : El\ a^*)(x^+ : Prf\ (a^+\ x^*)).\ El\ (b^*\ x^*\ x^+)\\ &\equiv_{\beta\Sigma} \Pi(x^* : (El\ a)^*)(x^+ : (El\ a)^+\ x^*).\ (El\ (b\ x))^*\\ &\equiv_{\beta\Sigma} (\Pi(x : El\ a).\ El\ (b\ x))^*\end{aligned}$$

$$\begin{aligned}\text{and } (El\ (a \leadsto_d b))^+ &\equiv_{\beta\Sigma} \lambda(f : El\ (a \leadsto_d b)^*).\ Prf\ ((a \leadsto_d b)^+\ f)\\ &\equiv_{\beta\Sigma} \lambda(f : El\ (a \leadsto_d b)^*).\\ &\qquad Prf\ (\forall\ a^*\ (\lambda x^*.\ (a^+\ x^*) \Rightarrow_d (\lambda x^+.\ b^+\ x^*\ x^+\ (f\ x^*\ x^+))))\\ &\equiv_{\beta\Sigma} \lambda(f : El\ (a \leadsto_d b)^*).\ \Pi(x^* : El\ a^*).\\ &\qquad Prf\ ((a^+\ x^*) \Rightarrow_d (\lambda x^+.\ b^+\ x^*\ x^+\ (f\ x^*\ x^+)))\\ &\equiv_{\beta\Sigma} \lambda(f : (El\ (a \leadsto_d b))^*).\ \Pi(x^* : El\ a^*)(x^+ : Prf\ (a^+\ x^*)).\\ &\qquad Prf\ (b^+\ x^*\ x^+\ (f\ x^*\ x^+))\\ &\equiv_{\beta\Sigma} \lambda(f : (\Pi(x : El\ a).\ El\ (b\ x))^*).\ \Pi(x^* : (El\ a)^*)(x^+ : (El\ a)^+\ x^*).\\ &\qquad (El\ (b\ x))^+\ (f\ x^*\ x^+)\\ &\equiv_{\beta\Sigma} (\Pi(x : El\ a).\ El\ (b\ x))^+.\end{aligned}$$

The result has been checked in DEDUKTI for the four rewrite rules, see the `#ASSERT` commands in the file `lo_sp.dk`. ☐

3.3 Interpretation of Theories

The interpretation of a source theory $\mathbb{S}$ in a target theory $\mathbb{T}$ is given by the parameters c^* and c^+, for each constant c of Σ. We have provided the parameters for the constants of Σ_{pre}, but the parameters for the constants of $\Sigma_{\mathbb{S}}$ remain to be given by the user.

[1] All the DEDUKTI files are available at `https://github.com/thomastraversie/InterpDK`.

Definition 4 (Interpretation of theories)**.** *Let $\mathbb{S}$ and $\mathbb{T}$ be two theories with prelude encoding. We say that $\mathbb{S}$ has an interpretation in $\mathbb{T}$ when:*

1. *for each constant $c : A \in \Sigma_{\mathbb{S}}$, we have a term c^* such that $\vdash c^* : A^*$ in $\mathbb{T}$,*
2. *for each constant $c : A \in \Sigma_{\mathbb{S}}$, we have a term c^+ such that $\vdash c^+ : A^+ \; c^*$ in $\mathbb{T}$,*
3. *for each rewrite rule $\ell \hookrightarrow r \in \Sigma_{\mathbb{S}}$, we have $\ell^* \equiv_{\beta\Sigma} r^*$ and $\ell^+ \equiv_{\beta\Sigma} r^+$ in $\mathbb{T}$.*

Remark that, in the third item, ℓ^+ and r^+ do not contain metavariables, as we have seen that `TYPE` cannot occur in ℓ and r.

If we cannot interpret the rewrite rules of $\mathbb{S}$ into conversions in $\mathbb{T}$, we can nonetheless replace the rewrite rules of $\mathbb{S}$ by equational axioms—that is by typed constants—and then interpret such constants in $\mathbb{T}$. So as to replace user-defined rewrite rules by equational axioms [6], we add an equality in our signature, and we use functional extensionality, uniqueness of identity proofs, and the congruence of equality on applications.

The $\lambda\Pi$-calculus modulo theory features substitutions in the type of an application—in the case of dependent types—and features user-defined rewrite rules. So that the translation of a provable judgment remains provable, it is important to maintain substitution and conversion through the translations $t \mapsto t^*$ and $t \mapsto t^+$. For each variable z occurring in a term t, the two variables z^* and z^+ occur in the translated terms t^* and t^+. The translation $(t[z \leftarrow w])^*$ is thus given by $t^*[z^* \leftarrow w^*][z^+ \leftarrow w^+]$.

Proposition 3 (Substitution)**.** *Let t and w be two terms and z be a variable. We have:*

- $(t[z \leftarrow w])^* = t^*[z^* \leftarrow w^*][z^+ \leftarrow w^+]$.
- $(t[z \leftarrow w])^+ = t^+[z^* \leftarrow w^*][z^+ \leftarrow w^+]$.

Proof. By induction on the term t. □

Proposition 4 (Conversion)**.** *If $A \equiv_{\beta\Sigma} B$ in $\mathbb{S}$, then $A^* \equiv_{\beta\Sigma} B^*$ and $A^+ \equiv_{\beta\Sigma} B^+$ in $\mathbb{T}$.*

Proof. We prove the result by induction on the formation of $A \equiv_{\beta\Sigma} B$.

- We have $(\lambda(x : A).\, t)\; u)^* = (\lambda(x^* : A^*)(x^+ : A^+\; x^*).\, t^*)\; u^*\; u^+$, which β-reduces to $t^*[x^* \leftarrow u^*][x^+ \leftarrow u^+]$, that is $(t[x \leftarrow u])^*$ following Proposition 3. Similarly, $((\lambda(x : A).\, t)\; u)^+ \equiv_{\beta\Sigma} (t[x \leftarrow u])^+$.
- For each $\ell \hookrightarrow r \in \Sigma$ and any substitution θ, we have $\ell^* \equiv_{\beta\Sigma} r^*$ by definition and Proposition 2. Using Proposition 3, we have $(\ell\theta)^* = \ell^*\theta^{*,+}$ and $(r\theta)^* = r^*\theta^{*,+}$, where $\theta^{*,+}$ is defined so that if θ substitutes z by w, then $\theta^{*,+}$ substitutes z^* by w^* and z^+ by w^+. Therefore $(\ell\theta)^* = \ell^*\theta^{*,+} \equiv_{\beta\Sigma} r^*\theta^{*,+} = (r\theta)^*$. Similarly, we have $(\ell\theta)^+ = \ell^+\theta^{*,+} \equiv_{\beta\Sigma} r^+\theta^{*,+} = (r\theta)^+$.
- For closure by context, we only show the λ-abstraction case. Suppose that $\lambda(x : A).\, t \equiv_{\beta\Sigma} \lambda(x : B).\, u$ derives from $A \equiv_{\beta\Sigma} B$ and $t \equiv_{\beta\Sigma} u$. By induction, we have $A^* \equiv_{\beta\Sigma} B^*$, and $A^+ \equiv_{\beta\Sigma} B^+$, and $t^* \equiv_{\beta\Sigma} u^*$, and $t^+ \equiv_{\beta\Sigma} u^+$. We derive that $\lambda(x^* : A^*)(x^+ : A^+\; x^*).\, t^* \equiv_{\beta\Sigma} \lambda(x^* : B^*)(x^+ : B^+\; x^*).\, u^*$, that is $(\lambda(x : A).\, t)^* \equiv_{\beta\Sigma} (\lambda(x : B).\, u)^*$. Similarly, $(\lambda(x : A).\, t)^+ \equiv_{\beta\Sigma} (\lambda(x : B).\, u)^+$.
- Reflexivity, symmetry and transitivity are immediate.

□

We have at hand all the tools allowing us to prove that, when $\mathbb{S}$ has an interpretation in $\mathbb{T}$, any provable judgment in $\mathbb{S}$ is interpreted as a provable judgment in $\mathbb{T}$. The first item of the theorem concerns well-formedness judgments. The second item concerns typing judgments with respect to the translation $t \mapsto t^*$, and the third item concerns typing judgments with respect to the translation $t \mapsto t^+$.

Theorem 1 (Interpretation)**.** *Let $\mathbb{S}$ and $\mathbb{T}$ be two theories with prelude encoding, such that $\mathbb{S}$ has an interpretation in $\mathbb{T}$.*

1. *If $\vdash \Gamma$ in $\mathbb{S}$, then $\vdash \Gamma^{*,+}$ in $\mathbb{T}$.*
2. *If $\Gamma \vdash t : A$ in $\mathbb{S}$ then $\Gamma^{*,+} \vdash t^* : A^*$ in $\mathbb{T}$.*
3. *(a) If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \texttt{TYPE}$ in $\mathbb{S}$, then $\Gamma^{*,+} \vdash t^+ : A^+\ t^*$ in $\mathbb{T}$.*
   (b) If $\Gamma \vdash t : A$ and $\Gamma \vdash A : \texttt{KIND}$ in $\mathbb{S}$, then $\Gamma^{*,+} \vdash t^+ : A^+\{t^*\}$ in $\mathbb{T}$.
   (c) If $\Gamma \vdash A : \texttt{KIND}$ in $\mathbb{S}$, then for every t such that $\Gamma^{*,+} \vdash t : A^*$ in $\mathbb{T}$, we have $\Gamma^{*,+} \vdash A^+\{t\} : \texttt{KIND}$.

Proof. We proceed by induction on the derivation. We only show the most interesting cases.

- <u>CONST:</u> By induction, we have $\vdash \Gamma^{*,+}$ and $\vdash A^* : s^*$. Since $c : A \in \Sigma$, we have $\vdash c^* : A^*$. We derive $\Gamma^{*,+} \vdash c^* : A^*$ by weakening. If $s = \texttt{TYPE}$, then $\vdash c^+ : A^+\ c^*$ and we derive $\Gamma^{*,+} \vdash c^+ : A^+\ c^*$ by weakening. If $s = \texttt{KIND}$, then $\vdash c^+ : A^+\{c^*\}$ and we derive $\Gamma^{*,+} \vdash c^+ : A^+\{c^*\}$ by weakening.
- <u>PROD:</u> By induction, we have $\Gamma^{*,+} \vdash A^* : \texttt{TYPE}$, and $\Gamma^{*,+} \vdash A^+ : A^* \to \texttt{TYPE}$, and $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash B^* : s^*$. Using PROD, we get $\Gamma^{*,+} \vdash \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^* : s^*$.

  Suppose that $s = \texttt{TYPE}$. By induction, $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash B^+ : B^* \to \texttt{TYPE}$. By weakening, we have $\Gamma^{*,+}, f : (\Pi(x : A).\ B)^*, x^* : A^*, x^+ : A^+\ x^* \vdash B^+ : B^* \to \texttt{TYPE}$. Since $\Gamma^{*,+}, f : (\Pi(x : A).\ B)^*, x^* : A^*, x^+ : A^+\ x^* \vdash B^+\ (f\ x^*\ x^+) : \texttt{TYPE}$, we derive $\Gamma^{*,+} \vdash \lambda(f : (\Pi(x : A).\ B)^*).\ \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\ (f\ x^*\ x^+) : (\Pi(x : A).\ B)^* \to \texttt{TYPE}$, which corresponds to $\Gamma^{*,+} \vdash (\Pi(x : A).\ B)^+ : \texttt{TYPE}^+\{(\Pi(x : A).\ B)^*\}$.

  Suppose that $s = \texttt{KIND}$ and that we have $\Gamma^{*,+} \vdash t : (\Pi(x : A).\ B)^*$. Since $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash t\ x^*\ x^+ : B^*$, by induction we get $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash B^+\{t\ x^*\ x^+\} : \texttt{KIND}$. We derive $\Gamma^{*,+} \vdash \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\{t\ x^*\ x^+\} : \texttt{KIND}$, that is $\Gamma^{*,+} \vdash (\Pi(x : A).\ B)^+\{t\} : \texttt{KIND}$.
- <u>ABS:</u> By induction, we have $\Gamma^{*,+} \vdash A^* : \texttt{TYPE}$, and $\Gamma^{*,+} \vdash A^+ : A^* \to \texttt{TYPE}$, and $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash B^* : s^*$, and $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash t^* : B^*$, . We derive $\Gamma^{*,+} \vdash \lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^* : \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^*$, that is $\Gamma^{*,+} \vdash (\lambda(x : A).\ t)^* : (\Pi(x : A).\ B)^*$.

  Suppose that $s = \texttt{TYPE}$. By induction, we have $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash B^+ : B^* \to \texttt{TYPE}$ and $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash t^+ : B^+\ t^*$. We derive $\Gamma^{*,+} \vdash \lambda(x^* : A^*)(x^+ : A^+\ x^*).\ t^+ : \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\ t^*$. Using CONV, we conclude that $\Gamma^{*,+} \vdash (\lambda(x : A).\ t)^+ : (\Pi(x : A).\ B)^+\ (\lambda(x : A).\ t)^*$.

  Suppose that $s = \texttt{KIND}$. By induction, we have $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash B^+\{t^*\} : \texttt{KIND}$ and $\Gamma^{*,+}, x^* : A^*, x^+ : A^+\ x^* \vdash t^+ : B^+\{t^*\}$. We derive $\Gamma^{*,+} \vdash \lambda(x^* : A^*)(x^+ : (A^+\ x^*)).\ t^+ : \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\{t^*\}$, that is $\Gamma^{*,+} \vdash (\lambda(x : A).\ t)^+ : (\Pi(x : A).\ B)^+\{(\lambda(x : A).\ t)^*\}$ using CONV.
- <u>APP:</u> By induction, we have $\Gamma^{*,+} \vdash t^* : \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^*$, and $\Gamma^{*,+} \vdash u^* : A^*$, and $\Gamma^{*,+} \vdash u^+ : A^+\ u^*$. We derive $\Gamma^{*,+} \vdash t^*\ u^*\ u^+ : B^*[x^* \leftarrow u^*][x^+ \leftarrow u^+]$. Using Proposition 3, we conclude that $\Gamma^{*,+} \vdash (t\ u)^* : (B[x \leftarrow u])^*$.

  Suppose that $\Gamma \vdash \Pi(x : A).\ B : \texttt{TYPE}$ (and thus $\Gamma \vdash B : \texttt{TYPE}$). By induction, we have $\Gamma^{*,+} \vdash t^+ : \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\ (t^*\ x^*\ x^+)$. It follows that $\Gamma^{*,+} \vdash t^+\ u^*\ u^+ : B^+[x^* \leftarrow u^*][x^+ \leftarrow u^+]\ (t^*\ u^*\ u^+)$. Using Proposition 3, we conclude that $\Gamma^{*,+} \vdash (t\ u)^+ : (B[x \leftarrow u])^+\ (t\ u)^*$.

  Suppose that $\Gamma \vdash \Pi(x : A).\ B : \texttt{KIND}$ (and thus $\Gamma \vdash B : \texttt{KIND}$). By induction, we have $\Gamma^{*,+} \vdash t^+ : \Pi(x^* : A^*)(x^+ : A^+\ x^*).\ B^+\{t^*\ x^*\ x^+\}$. It follows that $\Gamma^{*,+} \vdash t^+\ u^*\ u^+ : (B^+\{t^*\ x^*\ x^+\})[x^* \leftarrow u^*][x^+ \leftarrow u^+]$. Using Proposition 3, we conclude that $\Gamma^{*,+} \vdash (t\ u)^+ : (B[x \leftarrow u])^+\{(t\ u)^*\}$.
- <u>CONV:</u> We conclude using the induction hypotheses and Proposition 4.

□

Given an interpretation of a source theory $\mathbb{S}$ in a target theory $\mathbb{T}$, the results proved in $\mathbb{S}$ are automatically transported to $\mathbb{T}$. The interpretation of $\mathbb{S}$ in $\mathbb{T}$ only requires the parameters c^* and c^+ in $\mathbb{T}$ for each user-defined constant c of $\mathbb{S}$. Once we have an interpretation of $\mathbb{S}$ in $\mathbb{T}$, it is possible to prove that $\mathbb{S}$ is consistent provided that $\mathbb{T}$ is so. In the $\lambda\Pi$-calculus modulo theory, we say that a theory is inconsistent when we can build a term that takes a proposition and returns one of its proofs, that is when there exists a term t such that $\vdash t : \Pi(P : El\ o).\ Prf\ P$.

Theorem 2 (Relative consistency)**.** *Let $\mathbb{S}$ and $\mathbb{T}$ be two theories with prelude encoding, such that $\mathbb{S}$ has an interpretation in $\mathbb{T}$. If $\mathbb{T}$ is consistent, then $\mathbb{S}$ is consistent too.*

Proof. Assume that $\mathbb{S}$ is inconsistent, meaning that we have a term $\vdash t : \Pi(P : El\ o).\ Prf\ P$. By applying Theorem 1, we get $\vdash t : \Pi(P^* : El\ o)(P^+ : Prf\ P^* \to Prf\ P^*).\ Prf\ P^*$. We take the term $t' := \lambda(P^* : El\ o).\ t\ P^*\ (\lambda(x : Prf\ P^*).\ x)$ and we have $\vdash t' : \Pi(P^* : El\ o).\ Prf\ P^*$. It follows that $\mathbb{T}$ is inconsistent. $\square$

3.4 Examples of Interpretation

We illustrate the interpretation with two examples. First, we detail the embedding of the theory of natural numbers into the theory of integers. This example has been implemented in DEDUKTI. Second, we give an informal presentation of the embedding of Zermelo set theory into a theory where sets are represented by graphs. These two examples exemplify the practicality and limitations of this interpretation.

3.4.1 Natural Numbers and Integers

We aim at interpreting the theory of natural numbers $\mathbb{T}_n$ in the theory of integers $\mathbb{T}_i$. We intuitively take $\mathsf{nat}^* := \mathsf{int}$. An integer is a non-negative natural number, so the predicate asserting that an integer is a natural number is defined by $\mathsf{nat}^+ := \lambda z.\ z \geq_i 0_i$. The interpretation of 0_n is given by $0_n^* := 0_i$, and we choose $0_n^+ := \mathsf{ax}_i^1\ 0_i$ for the proof of $0_n^* \geq_i 0_i$. We take $\mathsf{succ}_n^* := \lambda x^*.\ \lambda x^+.\ \mathsf{succ}_i\ x^*$ and $\mathsf{succ}_n^* := \lambda x^*.\ \lambda x^+.\ \mathsf{ax}_i^3\ (\mathsf{succ}_i\ x^*)\ x^*\ 0_i\ (\mathsf{ax}_i^2\ x^*)\ x^+$. For the interpretation of $\geq_n$, we choose $\geq_n^* := \lambda x^*.\ \lambda x^+.\ \lambda y^*.\ \lambda y^+.\ x^* \geq_i y^*$. Given that $\geq_n$ returns a proposition, the parameter $\geq_n^+$ must have type $\Pi x^*.\ \Pi x^+.\ \Pi y^*.\ \Pi y^+.\ Prf\ (x^* \geq_i y^*) \to Prf\ (x^* \geq_i y^*)$, which has an immediate inhabitant. The interpretation of ax_i^1 is given by $(\mathsf{ax}_i^1)^* := \lambda x^*.\ \lambda x^+.\ \mathsf{ax}_i^1\ x^*$. Since ax_i^1 returns a proof, and by definition of Prf^+, both $(\mathsf{ax}_i^1)^*$ and $(\mathsf{ax}_i^1)^+$ have the same type, so we can take $(\mathsf{ax}_i^1)^+ := (\mathsf{ax}_i^1)^*$. The parameters for ax_i^2 and ax_i^3 are chosen correspondingly.

When defining the parameter rec_n^*, we assume P^* of type $\Pi(x^* : El\ \mathsf{nat}^*).\ Prf\ (x^* \geq_i 0_i) \to El\ o$. We must apply rec_i to a predicate of type $El\ \mathsf{nat}^* \to El\ o$, which asserts that an integer z is non-negative and that, given a proof h_z of its non-negativity, it holds $P^*\ z\ h_z$. Such a predicate can be encoded using $\forall$ and $\Rightarrow_d$. At some point in the proof, we want to show $P^*\ z\ h_z$, but we can only derive $P^*\ z\ h'_z$, where h_z and h'_z are two proofs of $z \geq_i 0_i$. To overcome this problem, we suppose *proof irrelevance*

$$\mathsf{proof\_irr} : \Pi(p : El\ o)(h\ h' : Prf\ p)(Q : Prf\, p \to El\ o).\ Prf\ (Q\ h) \to Prf\ (Q\ h')$$

which states that two proofs of the same proposition are equal.

Using this interpretation of natural numbers into integers, we can derive for free the theorems of $\mathbb{T}_n$ in $\mathbb{T}_i$. For instance, we can show in $\mathbb{T}_n$ that $\vdash \mathsf{thm} : \Pi(x : El\ \mathsf{nat}).\ Prf\ (\mathsf{succ}_n\ x \geq_n 0_n)$, where thm is a proof that uses rec_n, ax_n^1, ax_n^2 and ax_n^3. The interpretation of $\mathbb{T}_n$ in $\mathbb{T}_i$ allows us to directly derive $\vdash \mathsf{thm}^* : \Pi(x^* : El\ \mathsf{int}).\ Prf\ (x^* \geq_i 0_i) \to Prf\ (\mathsf{succ}_i\ x^* \geq_i 0_i)$ in $\mathbb{T}_i$.

The complete interpretation of natural numbers into integers has been formalized in DEDUKTI, and is available in the file `nat_sp.dk`.

3.4.2 Sets and Pointed Graphs

Sets can be represented by a more primitive notion of pointed graphs, such that this encoding satisfies Zermelo set theory [11]. Pointed graphs are directed graphs with a distinguished node—the root. In the $\lambda\Pi$-calculus modulo theory, pointed graphs are implemented [5] thanks to sorts graph and node of type *Set*. The predicate $\mathsf{eta} : El\ \mathsf{graph} \to El\ \mathsf{node} \to El\ \mathsf{node} \to El\ o$ is such that $\mathsf{eta}\ a\ x\ y$ is the proposition asserting that there is an edge in pointed graph a from node y to node x. The operator $\mathsf{root} : El\ \mathsf{graph} \to El\ \mathsf{node}$ returns the root of a pointed graph, and $\mathsf{cr} : El\ \mathsf{graph} \to El\ \mathsf{node} \to El\ \mathsf{graph}$ is such that $\mathsf{cr}\ a\ x$ corresponds to the pointed graph a in which the root is now at node x.

The different constructors on sets—unions, pairs, powersets and comprehension—are defined via rewrite rules using the structure of pointed graphs. At the end, every axiom of Zermelo set theory is a theorem in the theory of pointed graphs. Hence we can naturally interpret Zermelo set theory in the theory of pointed graphs. Remark that every pointed graph represents a set. It follows that the predicates asserting that an object of type $El\ \mathsf{graph}$ is indeed a set are not necessary.

The theory of pointed graphs is more computational than the usual Zermelo set theory. In particular, it satisfies a normalization theorem in deduction modulo theory [11]. Using such an interpretation, the theorems proved in Zermelo set theory can be transferred to the theory of pointed graphs.

4 Conclusion

In this paper, we have defined an interpretation of theories of the $\lambda\Pi$-calculus modulo theory with prelude encoding, given well-suited parameters for interpreting the constants of the source theory. If a source theory $\mathbb{S}$ has an interpretation in a target theory $\mathbb{T}$, then the theorems proved in $\mathbb{S}$ come for free in $\mathbb{T}$. At the end, we obtain a relative consistency result, establishing that the consistency of the theory $\mathbb{T}$ entails the consistency of the theory $\mathbb{S}$.

This interpretation applies when $\mathbb{S}$ can be embedded into $\mathbb{T}$. In particular, we allow the interpretation of a type A of $\mathbb{S}$ by a more general type A^* of $\mathbb{T}$. As a consequence, we ensure that, for every term t of type A in $\mathbb{S}$, its interpretation t^* of type A^* in $\mathbb{T}$ indeed satisfies the predicate A^+. Such an interpretation is well-suited when we embed a source theory into a more general target theory, as we have seen with natural numbers and integers. However, if the target theory encompasses exactly the source theory, then the translation introduces unnecessary predicates, as we have seen with sets and pointed graphs.

Practical application. The $\lambda\Pi$-calculus modulo theory has been implemented in the DEDUKTI proof language and in the LAMBDAPI proof assistant. Future work would be to implement this interpretation in DEDUKTI. It would allow *effective* proof transfers between different DEDUKTI theories, and would therefore strengthen the interoperability between proof assistants via DEDUKTI.

Theoretical application. Dowek and Miquel [12] developed a method for interpreting theories of first-order logic. They showed that this interpretation can be used to prove a relative normalization result for theories in deduction modulo theory [10], that is first-order logic extended with user-defined rewrite rules. An application of this paper would be to prove a relative normalization result for the $\lambda\Pi$-calculus modulo theory. We would therefore be able to show that the encoding of set theory via pointed graphs in the $\lambda\Pi$-calculus modulo theory [5] satisfies a relative normalization result, just like this encoding in deduction modulo theory [11] does.

Acknowledgments

The author is grateful to Valentin Blot, Gilles Dowek and Théo Winterhalter for their insightful feedback on this work, and thanks the reviewers for their relevant comments.

References

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory*. Manuscript.

[2] Jean-Philippe Bernardy, Patrik Jansson & Ross Paterson (2010): *Parametricity and dependent types*. In: *ICFP 2010 - 15th ACM SIGPLAN International Conference on Functional Programming*, Association for Computing Machinery, Baltimore, USA, p. 345–356, doi:10.1145/1863543.1863592.

[3] Jean-Philippe Bernardy, Patrik Jansson & Ross Paterson (2012): *Proofs for free: Parametricity for dependent types*. *Journal of Functional Programming* 22(2), p. 107–152, doi:10.1017/S0956796812000056.

[4] Frédéric Blanqui, Gilles Dowek, Emilie Grienenberger, Gabriel Hondet & François Thiré (2023): *A modular construction of type theories*. *Logical Methods in Computer Science* Volume 19, Issue 1, doi:10.46298/lmcs-19(1:12)2023. Available at https://lmcs.episciences.org/10959.

[5] Valentin Blot, Gilles Dowek & Thomas Traversié (2022): *An Implementation of Set Theory with Pointed Graphs in Dedukti*. In: *LFMTP 2022 - International Workshop on Logical Frameworks and Meta-Languages : Theory and Practice*, Haïfa, Israel. Available at https://inria.hal.science/hal-03740004.

[6] Valentin Blot, Gilles Dowek, Thomas Traversié & Théo Winterhalter (2024): *From Rewrite Rules to Axioms in the $\lambda\Pi$-Calculus Modulo Theory*. In: *FoSSaCS 2024 - 27th International Conference on Foundations of Software Science and Computation Structures*, Springer Nature Switzerland, Luxembourg, Luxembourg, pp. 3–23, doi:10.1007/978-3-031-57231-9_1.

[7] Cyril Cohen, Enzo Crance & Assia Mahboubi (2024): *Trocq: Proof Transfer for Free, With or Without Univalence*. In: *ESOP 2024 - 33rd European Symposium on Programming*, Springer Nature Switzerland, Luxembourg, Luxembourg, pp. 239–268, doi:10.1007/978-3-031-57262-3_10.

[8] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo*. In: *TLCA 2007 - 8th International Conference on Typed Lambda Calculi and Applications*, Springer Berlin Heidelberg, Paris, France, pp. 102–117, doi:10.1007/978-3-540-73228-0_9.

[9] Nachum Dershowitz & Jean-Pierre Jouannaud (1991): *Rewrite Systems*. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, doi:10.1016/B978-0-444-88074-1.50011-1.

[10] Gilles Dowek, Thérèse Hardin & Claude Kirchner (2003): *Theorem Proving Modulo*. *Journal of Automated Reasoning* 31, pp. 33–72, doi:10.1023/A:1027357912519.

[11] Gilles Dowek & Alexandre Miquel (2007): *Cut elimination for Zermelo set theory*. Manuscript.

[12] Gilles Dowek & Alexandre Miquel (2007): *Relative normalization*. Available at https://arxiv.org/abs/2310.20248. Manuscript.

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

[14] Gabriel Hondet & Frédéric Blanqui (2020): *The New Rewriting Engine of Dedukti*. In: *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*, 167, Paris, France, p. 16, doi:10.4230/LIPIcs.FSCD.2020.35. Available at https://inria.hal.science/hal-02981561.

[15] Chantal Keller & Marc Lasson (2012): *Parametricity in an Impredicative Sort*. In: *CSL 2012 - 26th EACSL Annual Conference on Computer Science Logic*, 16, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Fontainebleau, France, pp. 381–395, doi:10.4230/LIPIcs.CSL.2012.381. Available at https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2012.381.

[16] John C. Reynolds (1983): *Types, Abstraction and Parametric Polymorphism*. In: *Information Processing 83 - IFIP 9th World Computer Congress*, North-Holland/IFIP, Paris, France, pp. 513–523.

[17] François Thiré (2020): *Interoperability between proof systems using the logical framework Dedukti*. Ph.D. thesis, Université Paris-Saclay. Available at `https://hal.science/tel-03224039`.

[18] Philip Wadler (1989): *Theorems for free!* In: *FPCA 1989 - 4th International Conference on Functional Programming Languages and Computer Architecture*, Association for Computing Machinery, New York, USA, p. 347–359, doi:`10.1145/99370.99404`.