# Formalizing Representation Theorems for a Logical Framework with Rewriting

Thomas Traversié, Florian Rabe

# Formalizing Representation Theorems for a Logical Framework with Rewriting

Thomas Traversié* Florian Rabe†

**Abstract**

Representation theorems for formal systems often take the form of an inductive translation that satisfies certain invariants, which are proved inductively. Theory morphisms and logical relations are common patterns of such inductive constructions. They allow representing the translation and the proofs of the invariants as a set of translation rules, corresponding to the cases of the inductions. Importantly, establishing the invariants is reduced to checking a finite set of, typically decidable, statements. Therefore, in a framework supporting theory morphisms and logical relations, translations that fit one of these patterns become much easier to formalize and to verify.

The $\lambda\Pi$-calculus modulo rewriting is a logical framework designed for representing and translating between formal systems that has previously not systematically supported such patterns. In this paper, we extend it with theory morphisms and logical relations. We apply these to define and verify invariants for a number of translations between formal systems. In doing so, we identify some best practices that enable us to obtain elegant novel formalizations of some challenging translations, in particular sort-erasure translations from sorted to unsorted languages.

## 1 Introduction

**Motivation and Related Work** Logical frameworks are meta-languages for formalizing deductive systems. The idea originated in Automath [dB80] and was refined, e.g., by the Isabelle system [Pau94] based on higher-order logic and the Edinburgh Logical Framework [HHP93] (LF) based on the dependently-typed $\lambda$-calculus. A variety of LF-based practical logical frameworks have been developed, including extensions with logic programming in Twelf [PS99], abstraction over contexts in Beluga [PD10], monadic side conditions in $\text{LLF}_{\mathcal{P}}$ [HLMS17], and user-definable features in MMT [Rab18]. Many different logics can be encoded in LF-based frameworks including higher-order logics [HHP93], type systems [BDG+23], modal logics [AHMP98], foundations of mathematics [IR11], model theory [HR11], or the calculus of constructions [BDG+23].

The $\lambda\Pi$-calculus modulo rewriting [CD07] ($\lambda\Pi/\mathcal{R}$) extends LF with user-defined rewrite rules, both at the term and the type level. All terms and types are considered modulo the congruence relation induced by the usual $\beta$-reduction rule and by the user-defined rewrite rules. $\lambda\Pi/\mathcal{R}$ was implemented in the Dedukti proof language [ABC+16, Sai15], designed for exchanging proofs between systems. For instance, it was used to translate [Thi20] the Matita arithmetic library to several systems including Rocq and PVS, and to export [Bla24] the HOL Light standard library to Rocq.

*Université Paris-Saclay, CentraleSupélec, MICS, France and Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, LMF, France, thomas.traversie@centralesupelec.fr

†University Erlangen-Nuremberg, Germany, florian.rabe@fau.de

The type-level rewriting includes in particular the ability to rewrite an atomic type into a function type. For example, the type $Prf\ (p \Rightarrow q)$ of proofs of implications can be rewritten into—and thus become equal to—the function type $Prf\ p \to Prf\ q$. Compared to typical encodings of natural deduction without rewriting, this means that the implication introduction and elimination rules, which normally map back and forth between those two types, can be dropped entirely. For example, the modus ponens step $\mathsf{imp_e}\ p\ q\ H_{pq}\ H_p$, with $\mathsf{imp_e} : \Pi p, q : Prop.\ Prf\ (p \Rightarrow q) \to Prf\ p \to Prf\ q$, becomes simply $H_{pq}\ H_p$, thus erasing the explicit application of $\mathsf{imp_e}$ and its arguments $p$ and $q$. This simplifies the work of the formalizer and has a major impact on scalability when representing large libraries of theorems, such as those exported from proof assistants, because it leads to much smaller proof terms. This effect has been leveraged heavily and critically in the existing encodings in Dedukti.

Additionally, the need to obtain fast implementations of rewriting has led the $\lambda\Pi/\mathcal{R}$ community to adopt the design choice of an untyped conversion relation, i.e., both the usual $\beta\eta$-equality and rewriting operate on any terms, not just the well-typed ones.

This raises the question whether meta-theorems about LF carry over, or if these generalizations subtly interfere with established intuitions and results. This is important because reasoning about the meta-theory of deductive systems, such as establishing type or truth preservation of translations, has been a major application of logical frameworks. In this paper we answer this question positively for two such meta-theorems as described below.

Representation theorems often take the form of $\forall\exists$ meta-statements, e.g., expressing that for all terms $t : A$ of language $\mathbb{S}$, there is a term $\mu(t) : \mu(A)$ of language $\mathbb{T}$ (where $\mathbb{S}$ and $\mathbb{T}$ are independently formalized in the framework). There are two approaches to formalizing such representation functions $\mu$. Firstly, $\mu$ can be implemented in a programming language that treats the terms of $\mathbb{S}$ and $\mathbb{T}$ as data. These programs are logic programs in [PS99] or functional programs in [PD10, PS09]. The framework then has to verify the meta-theorem by proving the correctness and termination of the program. Among early big case studies are verifications of cut-elimination [Pfe00] and logic translations [SS06].

Secondly, the framework can provide explicit support for certain restricted classes of representation functions for which correctness and termination are guaranteed. These are usually significantly easier for the user to define, their formalization is more elegant, and their verification is decidable and easy to implement. When applicable, they are usually the superior formalization. However, their expressivity is limited, and intended applications often hit these limitations. *Theory morphisms* (also called signature morphisms) were introduced for LF in [HST94]. Here the function $\mu$ is induced homomorphically on all $\mathbb{S}$-terms from manually supplied translations of all constants of $\mathbb{S}$. The function $\mu$ is guaranteed to be total, to preserve all judgments, and to be compositional, i.e., commute with substitutions. Theory morphisms were added to Isabelle in [KWP99], to Twelf in [RS09], and to MMT in [RK13]. They also allow building a module system in the style of [SW83], and they were used to build a major library of modular logics and representation theorems [CHK+11, Rab14].

To extend the expressivity of theory morphisms while retaining their simplicity, [RS13] introduced *logical relations* for LF. Here the function $\mu$ is coupled with a second function $\rho$ that establishes additional invariants about $\mu$. For example, if $\mu$ is a type-erasure translation, then $\rho$ can be used to state and prove a type-preservation invariant. Parametricity translations [BJP10, BJP12] closely resemble logical relations for pure type systems. They were developed in [KL12] for the calculus of inductive constructions, and [CCM24] builds a parametricity-based Rocq plugin for automated proof transfer.

All of these developments were done in the absence of rewriting. In fact, other than the Maude tool [CELM96], which is based on membership equational logic with rewriting, we are not aware of any tool supporting theory morphisms or related concepts on top of a rewrite system, and none that do so for a dependently-typed $\lambda$-calculus. Felicissimo [Fel22] effectively defined theory morphisms in $\lambda\Pi/\mathcal{R}$ to establish the soundness of an encoding of functional and explicitly-typed pure type systems. Similarly, [Tra24b] encoded individual interpretations in which the morphism and the relation are mutually recursive. However, both lacked a general definition of the concept and a general meta-theorem establishing their properties once and for all. That is critical to fully leverage morphisms

in practice, because it allows shifting most of the work to the framework and leaving only a small amount of work to the formalizer of an individual translation.

**Contribution** Our work follows the morphism-based approach mentioned above in the context of the $\lambda\Pi$-calculus modulo rewriting. Our contribution is threefold.

Firstly, we generalize theory morphisms and logical relations from LF to $\lambda\Pi/\mathcal{R}$, stating all definitions and theorems in full generality. Maybe surprisingly, even though the initial definition of $\lambda\Pi/\mathcal{R}$ is more complex than that of LF, the proofs down the road become significantly simpler. The resulting formalism subsumes the translation templates of [RS13] and simplifies the special interpretations of [Tra24b].

Secondly, we apply this formalism to define translations that have previously been out of reach for morphism-like methods. In particular, we formalize a novel translation from hard-sorted to soft-sorted to unsorted logic. We show that a specific combination of subtle design choices allows representing these translations concisely and then obtaining their soundness from the framework for free. To our knowledge, this is the first time that such translations are defined in a fully declarative way. Notably, some of these design choices involve encoding artefacts that have previously, i.e., in the absence of rewriting, proved to be very cumbersome to use at scale. Here, rewriting enables us to use clever encodings when defining the translations and then eliminate the artefacts when using the translations in practice.

Thirdly, we have implemented the `TranslationTemplates` tool, which realizes theory morphisms and unary logical relations for Dedukti. We have used `TranslationTemplates` to formalize all examples shown in the sequel. The implementation and examples are available at

https://github.com/Deducteam/TranslationTemplates.

**Overview** In Section 2, we recap the syntax and typing rules of $\lambda\Pi/\mathcal{R}$. Then we define theory morphisms and logical relations for $\lambda\Pi/\mathcal{R}$ in Section 3 and Section 4. In Section 5 and Section 6, we apply our framework to challenging sort-erasure translations and datatype embeddings. We describe our implementation in Section 7.

## 2 The $\lambda\Pi$-Calculus Modulo Rewriting

The Edinburgh Logical Framework, also known as LF or $\lambda\Pi$-calculus, corresponds to simply typed $\lambda$-calculus extended with dependent types. $\lambda\Pi/\mathcal{R}$ is an extension of LF with user-defined rewrite rules.

The terms of $\lambda\Pi/\mathcal{R}$ are divided into three levels: objects (denoted by $M$ and $N$), types (denoted by $A$ and $B$), and kinds (denoted by $K$). The syntax of $\lambda\Pi/\mathcal{R}$ is given by the following grammars:

| | |
|---|---|
| *Objects* | $M, N ::= c \mid x \mid \lambda x : A.\ M \mid M\ N$ |
| *Types* | $A, B ::= a \mid \Pi x : A.\ B \mid \lambda x : A.\ B \mid A\ M$ |
| *Kinds* | $K ::= \mathsf{Type} \mid \Pi x : A.\ K$ |
| *Terms* | $t, u, v, T ::= M \mid A \mid K \mid \mathsf{Kind}$ |

where $c$ is an object constant, $a$ is a type constant, and $x$ is a variable. Dependent products $\Pi x : A.\ B$ (respectively $\Pi x : A.\ K$) are simply written $A \to B$ (respectively $A \to K$) when $x$ does not occur in $B$ (respectively $K$).

Substitutions $\theta$ associate terms to variables. We write $t\theta$ for the result of the capture-avoiding substitution of the term $t$ with respect to $\theta$. Contexts $\Gamma$ are used to specify the type of the free variables, and theories $\mathbb{T}$ are used to declare the constants and rewrite rules considered by the users. Substitutions, contexts and theories are finite sequences, and are written $\varnothing$ when empty.

Rewrite rules are pairs $M \hookrightarrow N$ (respectively $A \hookrightarrow B$). The left- and right-hand side of a rewrite rule may contain free variables. In Definition 1, we will restrict the rules in such a way that all variable types can be inferred. Therefore, we omit the context that binds these variables from the notation.

| | |
|---|---|
| *Substitutions* | $\theta ::= \varnothing \mid \theta, x \leftarrow N$ |
| *Contexts* | $\Gamma ::= \varnothing \mid \Gamma, x : A$ |
| *Theories* | $\mathbb{T} ::= \varnothing \mid \mathbb{T},\ c : A \mid \mathbb{T},\ a : K \mid \mathbb{T},\ M \hookrightarrow N \mid \mathbb{T},\ A \hookrightarrow B$ |

We write $\mathbb{T} \vdash$ when the theory $\mathbb{T}$ is well formed, $\vdash_{\mathbb{T}} \Gamma$ when the context $\Gamma$ is well formed, and $\Gamma \vdash_{\mathbb{T}} t : T$ when the term $t$ has type $T$ in the context $\Gamma$. For convenience, $\varnothing \vdash_{\mathbb{T}} t : T$ is simply written $\vdash_{\mathbb{T}} t : T$. We write $\mathsf{dom}(\Gamma)$ for the domain of a context $\Gamma$, and $\mathsf{dom}(\mathbb{T})$ for the domain of a theory $\mathbb{T}$. Well-formed syntax is defined by the typing rules given in Figure 1, by Definition 1 and Definition 2.

**Definition 1.** *Let $\mathbb{T}$ be a theory and $\ell, r$ be two terms. We say that the rewrite rule $\ell \hookrightarrow r$ is* well-formed*, written $\mathbb{T} \vdash \ell \hookrightarrow r$, when the following holds:*

1. *$\ell$ is algebraic but not simply a variable,*
2. *all free variables of $r$ are also free in $\ell$,*
3. *the rule preserves typing, i.e., whenever $\Gamma \vdash_{\mathbb{T}} \ell\theta : T$ for some substitution $\theta$, then $\Gamma \vdash_{\mathbb{T}} r\theta : T$.*

*Here a term is called* algebraic *if it is a variable or the application of a constant to algebraic arguments.*

The first two conditions in Definition 1 are straightforward: they restrict the left-hand side to be a pattern and allow inferring the types of all free variables in a rewrite rule from their occurrences on the left-hand side. But the third condition may appear surprising: $\lambda\Pi/\mathcal{R}$ does not require, as one might expect, that the left- and right-hand side are well-formed and have the same type. Instead, it only requires that every rewrite rule preserves typing whenever a substitution instance of the left-hand side is well-formed. The most common situation where it helps to have this generality is when we linearize rewrite rules.

**Remark 1** (Left-Linear Rules)**.** *Consider an intrinsically typed encoding of polymorphic lists, using constants* $\mathsf{cons}$ *and* $\mathsf{hd}$ *that take the underlying type as a first argument. The natural rewrite rule is* $\mathsf{hd}\ a\ (\mathsf{cons}\ a\ h\ t) \hookrightarrow h$.

*To reuse existing confluence criteria on left-linear rules, it is however preferable to use the linearized version* $\mathsf{hd}\ a\ (\mathsf{cons}\ a'\ h\ t) \hookrightarrow h$*, even though its left-hand side is ill-formed [Bla01, Sai15]. This rule is still type-preserving in the sense of Definition 1, because substitution instances of the left-hand side can only be well-typed if they substitute the same terms for $a$ and $a'$.*

While this type-preservation condition makes it more difficult to check individual rules, the following lemma from [Sai15] states that the usual intuition is a sufficient criterion.

**Lemma 1** (Typed Rewrite Rules)**.** *Let $\ell \hookrightarrow r$ be a rewrite rule that satisfies the first two conditions of Definition 1. If $\Gamma$ declares the variables of $\ell$, and we have $\Gamma \vdash \ell : T$ and $\Gamma \vdash r : T$, then $\ell \hookrightarrow r$ preserves typing.*

Before defining the conversion relation, we explain two design choices.

**Remark 2** (Untyped Conversion)**.** *Type systems, such as LF and $\lambda\Pi/\mathcal{R}$, can be presented with typed conversion (where $\Gamma \vdash M \equiv N : A$ means that $M$ and $N$ are equal* and *well-typed) or untyped conversion (where $M \equiv N$ only means that $M$ and $N$ are equal).*

*In the presence of rewriting, it is strongly preferable to use the untyped variant [CD07, ABC+16, Sai15, BDG+23]. It allows defining and efficiently implementing the conversion relation on the context-free syntax without*

**Contexts**

$$\frac{}{\vdash_{\mathbb{T}} \varnothing}\ [\textsc{Empty}] \qquad \frac{\vdash_{\mathbb{T}} \Gamma \qquad \Gamma \vdash_{\mathbb{T}} A : \mathsf{Type}}{\vdash_{\mathbb{T}} \Gamma, x : A}\ [\textsc{Decl}]\ x \notin \mathsf{dom}(\Gamma)$$

**Theories**

$$\frac{}{\varnothing \vdash}\ [\textsc{Thy-Empty}] \qquad \frac{\mathbb{T} \vdash \qquad \vdash_{\mathbb{T}} A : \mathsf{Type}}{\mathbb{T}, c : A \vdash}\ [\textsc{Decl-Obj}]\ c \notin \mathsf{dom}(\mathbb{T})$$

$$\frac{\mathbb{T} \vdash \qquad \vdash_{\mathbb{T}} K : \mathsf{Kind}}{\mathbb{T}, a : K \vdash}\ [\textsc{Decl-Type}]\ a \notin \mathsf{dom}(\mathbb{T}) \qquad \frac{\mathbb{T} \vdash \qquad \mathbb{T} \vdash M \hookrightarrow N}{\mathbb{T}, M \hookrightarrow N \vdash}\ [\textsc{Rewr-Obj}]$$

$$\frac{\mathbb{T} \vdash \qquad \mathbb{T} \vdash A \hookrightarrow B}{\mathbb{T}, A \hookrightarrow B \vdash}\ [\textsc{Rewr-Type}]$$

**Objects**

$$\frac{\vdash_{\mathbb{T}} \Gamma}{\Gamma \vdash_{\mathbb{T}} c : A}\ [\textsc{Const-Obj}]\ c : A \in \mathbb{T} \qquad \frac{\vdash_{\mathbb{T}} \Gamma}{\Gamma \vdash_{\mathbb{T}} x : A}\ [\textsc{Var}]\ x : A \in \Gamma$$

$$\frac{\Gamma \vdash_{\mathbb{T}} A : \mathsf{Type} \qquad \Gamma, x : A \vdash_{\mathbb{T}} B : \mathsf{Type} \qquad \Gamma, x : A \vdash_{\mathbb{T}} M : B}{\Gamma \vdash_{\mathbb{T}} \lambda x : A.\ M : \Pi x : A.\ B}\ [\textsc{Abs-Obj}]$$

$$\frac{\Gamma \vdash_{\mathbb{T}} M : \Pi x : A.\ B \qquad \Gamma \vdash_{\mathbb{T}} N : A}{\Gamma \vdash_{\mathbb{T}} M\ N : B[x \leftarrow N]}\ [\textsc{App-Obj}] \qquad \frac{\Gamma \vdash_{\mathbb{T}} M : A \qquad \Gamma \vdash_{\mathbb{T}} B : \mathsf{Type}}{\Gamma \vdash_{\mathbb{T}} M : B}\ [\textsc{Conv-Type}]\ A \equiv_{\beta(\eta)\mathcal{R}} B$$

**Types**

$$\frac{\vdash_{\mathbb{T}} \Gamma}{\Gamma \vdash_{\mathbb{T}} a : K}\ [\textsc{Const-Type}]\ a : K \in \mathbb{T} \qquad \frac{\Gamma \vdash_{\mathbb{T}} A : \mathsf{Type} \qquad \Gamma, x : A \vdash_{\mathbb{T}} B : \mathsf{Type}}{\Gamma \vdash_{\mathbb{T}} \Pi x : A.\ B : \mathsf{Type}}\ [\textsc{Prod-Type}]$$

$$\frac{\Gamma \vdash_{\mathbb{T}} A : \mathsf{Type} \qquad \Gamma, x : A \vdash_{\mathbb{T}} K : \mathsf{Kind} \qquad \Gamma, x : A \vdash_{\mathbb{T}} B : K}{\Gamma \vdash_{\mathbb{T}} \lambda x : A.\ B : \Pi x : A.\ K}\ [\textsc{Abs-Type}]$$

$$\frac{\Gamma \vdash_{\mathbb{T}} A : \Pi x : B.\ K \qquad \Gamma \vdash_{\mathbb{T}} M : B}{\Gamma \vdash_{\mathbb{T}} A\ M : K[x \leftarrow M]}\ [\textsc{App-Type}] \qquad \frac{\Gamma \vdash_{\mathbb{T}} A : K \qquad \Gamma \vdash_{\mathbb{T}} K' : \mathsf{Kind}}{\Gamma \vdash_{\mathbb{T}} A : K'}\ [\textsc{Conv-Kind}]\ K \equiv_{\beta(\eta)\mathcal{R}} K'$$

**Kinds**

$$\frac{\vdash_{\mathbb{T}} \Gamma}{\Gamma \vdash_{\mathbb{T}} \mathsf{Type} : \mathsf{Kind}}\ [\textsc{Sort}] \qquad \frac{\Gamma \vdash_{\mathbb{T}} A : \mathsf{Type} \qquad \Gamma, x : A \vdash_{\mathbb{T}} K : \mathsf{Kind}}{\Gamma \vdash_{\mathbb{T}} \Pi x : A.\ K : \mathsf{Kind}}\ [\textsc{Prod-Kind}]$$

Figure 1: Well-formedness rules for the syntax of $\lambda\Pi/\mathcal{R}$

*any dependency on the context or the type system. To avoid accidentally converting well-typed terms into ill-typed terms, the rules* Conv-Type *and* Conv-Kind *check that the converted term is well-typed before using it.*

*Importantly, this means that proofs of meta-theorems that proceed by induction on derivations can be split into two parts: first establishing the result for conversion and then the result for typing. This is in contrast to presentations with typed conversion as used in [RS13], where both results must be proved jointly in a much more complicated mutually recursive induction. Therefore, the proofs of our main results in Section 3 and Section 4 are simpler than those in [RS13] even though the results are more general.*

**Remark 3** ($\eta$-Conversion and Adequacy)**.** *$\eta$-conversion is necessary in LF-style logical frameworks to obtain adequate encodings of bindings via higher-order abstract syntax. However, the combination of untyped rewriting and $\eta$ is subtle and must be handled carefully [Bla16]. Therefore, $\lambda\Pi/\mathcal{R}$ is often presented without $\eta$, and Dedukti provides a flag to turn $\eta$ on or off.*

*In order to retain compatibility with the existing literature on both $\lambda\Pi/\mathcal{R}$ (often without $\eta$) and other logical frameworks (usually with $\eta$), we systematically develop our results for both variants. In particular, all our meta-theorems hold with and and without $\eta$ unless mentioned otherwise, e.g., in Example 8.*

We finally define the conversion relation on (not necessarily well-formed) terms, also called definitional equality.

**Definition 2** (Conversion)**.** *We assume that all terms are once and for all quotiented by $\alpha$-equality. Let $\mathbb{T}$ be a theory. The relation $\hookrightarrow_{\beta\mathcal{R}}$ (respectively $\hookrightarrow_{\beta\eta\mathcal{R}}$) is the smallest relation, closed by term constructors and substitutions, that is generated by $\beta$-reduction (respectively $\beta$-reduction and $\eta$-expansion) and by the rewrite rules of $\mathbb{T}$. The conversion $\equiv_{\beta\mathcal{R}}$ (respectively $\equiv_{\beta\eta\mathcal{R}}$) is the reflexive, symmetric, and transitive closure of $\hookrightarrow_{\beta\mathcal{R}}$ (respectively $\hookrightarrow_{\beta\eta\mathcal{R}}$).*

*We use the subscript $\beta(\eta)\mathcal{R}$ to indicate that either variant can be used if it globally fixed which variant to use.*

We restrict our attention to theories for which $\hookrightarrow_{\beta(\eta)\mathcal{R}}$ is confluent and terminating, and thus produces normal forms of terms.

**Example 1** (PL)**.** *We define the theory* PL*, a fragment of propositional logic with implication and conjunction. Prop is the type of propositions and Prf maps a proposition to the type of its proof.*

$Prop : \mathsf{Type}$

$Prf : Prop \to \mathsf{Type}$

$\Rightarrow : Prop \to Prop \to Prop$

$\wedge : Prop \to Prop \to Prop$

*Every natural deduction inference rule is encoded by an axiom, that is a typed constant.*

$\mathsf{imp_i} : \Pi p, q : Prop.\ (Prf\ p \to Prf\ q) \to Prf\ (p \Rightarrow q)$

$\mathsf{imp_e} : \Pi p, q : Prop.\ Prf\ (p \Rightarrow q) \to (Prf\ p \to Prf\ q)$

$\mathsf{and_i} : \Pi p : Prop.\ Prf\ p \to \Pi q : Prop.\ Prf\ q \to Prf\ (p \wedge q)$

$\mathsf{and_{e\ell}} : \Pi p, q : Prop.\ Prf\ (p \wedge q) \to Prf\ p$

$\mathsf{and_{er}} : \Pi p, q : Prop.\ Prf\ (p \wedge q) \to Prf\ q$

**Example 2** (PLeq)**.** *The theory* PLeq *extends propositional logic* PL *with equality. $\iota$ is the type of individuals. We*

*define an equality symbol for elements of type $\iota$, along with the reflexivity principle and the Leibniz principle.*

$\iota$ : Type
$= : \iota \to \iota \to \mathit{Prop}$
refl : $\Pi x : \iota.\ \mathit{Prf}\ (x = x)$
leib : $\Pi x, y : \iota.\ \mathit{Prf}\ (x = y) \to \Pi P : \iota \to \mathit{Prop}.\ \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ y)$

**Example 3** (Groups based on multiplication)**.** *The theory* `MulGr` *of multiplicative groups extends* `PLeq` *with a multiplication symbol* $\times$*, an inverse operation* inv*, and a neutral element* 1*.*

| | | | |
|---|---|---|---|
| $\times : \iota \to \iota \to \iota$ | $x \times 1 \hookrightarrow x$ | $x \times (\mathsf{inv}\ x) \hookrightarrow 1$ | $\mathsf{inv}\ 1 \hookrightarrow 1$ |
| $1 : \iota$ | $1 \times x \hookrightarrow x$ | $(\mathsf{inv}\ x) \times x \hookrightarrow 1$ | $\mathsf{inv}\ (\mathsf{inv}\ x) \hookrightarrow x$ |
| $\mathsf{inv} : \iota \to \iota$ | $(x \times y) \times z \hookrightarrow x \times (y \times z)$ | | |

*We use rewrite rules to capture the usual associativity, neutrality, and inverseness axioms. Such axioms therefore derive from the rewrite rules. For instance, the axiom of associativity is proved by simply combining* $\mathsf{all}_\mathsf{i}$ *and* refl*. We benefit from the computational power of* $\lambda\Pi/\mathcal{R}$*. For example, we obtain the conversion* $(\mathsf{inv}\ (\mathsf{inv}\ x)) \times (\mathsf{inv}\ x \times y) \equiv_{\beta\mathcal{R}} y$ *for free.*

**Example 4** (Groups based on division)**.** *The theory* `DivGr` *of division groups extends* `PLeq` *with a division operation* $\div$ *and a neutral element* 1*.*

| | | |
|---|---|---|
| $\div : \iota \to \iota \to \iota$ | $(x \div y) \div z \hookrightarrow x \div (y \div (1 \div z))$ | $x \div 1 \hookrightarrow x$ |
| $1 : \iota$ | $1 \div (1 \div x) \hookrightarrow x$ | $x \div x \hookrightarrow 1$ |

*Using these rewrite rules, we have* $((y \div x) \div y) \div (1 \div x) \equiv_{\beta\mathcal{R}} 1$ *for free.*

*Relative to division, we can define the usual group operation* $x \times y$ *as* $x \div (1 \div y)$*. More generally, we will show in Example 12 that* `DivGr` *is isomorphic to* `MulGr`*.*

**Remark 4** (Proof Irrelevance)**.** *When representing proof systems in logical frameworks, it is occasionally important to impose irrelevance conditions on certain type symbols, e.g., on the symbol Prf from our examples to obtain proof irrelevance for the encoded logic. Whether or not proof irrelevance can be encoded depends on the specific version of* $\lambda\Pi/\mathcal{R}$*. The original version introduced in [CD07] used a very general form of rewrite rules with context that allows declaring arbitrary rewrite rules such as*

| | | | |
|---|---|---|---|
| $\mathit{unit}$ : Type | $\star : \mathit{unit}$ | $[x : \mathit{unit}]\, x \hookrightarrow \star$ | $[p : \mathit{Prop}, H : \mathit{Prf}\ p]\, \mathit{Prf}\ p \hookrightarrow \mathit{unit}$ |

*This introduces a unit type and rewrites every inhabited proof type into it. But more recent versions such as the one from [BDG+23] do not allow such rewrite rules in order to simplify confluence analysis and to obtain more efficient implementations. They disallow rules whose context cannot be inferred from its left-hand side, e.g., because it is just a variable (like* $x$ *above) or because of unused variables (like* $H$ *above). Our Definition 1 follows this approach. However, Example 12 assumes for simplicity that the framework offers some way to encode proof irrelevance.*

# 3 Theory Morphisms

In this section, we define theory morphisms for $\lambda\Pi/\mathcal{R}$, and we prove the basic Judgment Preservation theorem. As a running example, we give theory morphisms that translate between `MulGr` and `DivGr`.

### 3.1 Formal Definition

Theory morphisms from theory $\mathbb{S}$ to theory $\mathbb{T}$ are translations that replace the *constants* of $\mathbb{S}$ by *terms* of $\mathbb{T}$. Such terms are the parameters of the translation and must be provided to perform the translation.

**Definition 3** (Theory morphism)**.** *The mapping $\mu$ defined inductively from a set of parameters $\mu_c$ and $\mu_a$ by*

$$\begin{array}{lcllcl}
\mu(x) & = & x & \mu(\lambda x : A.\ M) & = & \lambda x : \mu(A).\ \mu(M) \\
\mu(c) & = & \mu_c & \mu(\lambda x : A.\ B) & = & \lambda x : \mu(A).\ \mu(B) \\
\mu(a) & = & \mu_a & \mu(\Pi x : A.\ B) & = & \Pi x : \mu(A).\ \mu(B) \\
\mu(M\ N) & = & \mu(M)\ \mu(N) & \mu(\Pi x : A.\ K) & = & \Pi x : \mu(A).\ \mu(K) \\
\mu(A\ M) & = & \mu(A)\ \mu(M) & \mu(\mathsf{Kind}) & = & \mathsf{Kind} \\
\mu(\mathsf{Type}) & = & \mathsf{Type} & & &
\end{array}$$

*is a theory morphism from theory $\mathbb{S}$ to theory $\mathbb{T}$ when:*

1. *for every constant $c : A \in \mathbb{S}$, there exists a term $\mu_c$ such that $\vdash_{\mathbb{T}} \mu_c : \mu(A)$,*
2. *for every constant $a : K \in \mathbb{S}$, there exists a term $\mu_a$ such that $\vdash_{\mathbb{T}} \mu_a : \mu(K)$,*
3. *for every rewrite rule $\ell \hookrightarrow r \in \mathbb{S}$, we have $\mu(\ell) \equiv_{\beta(\eta)\mathcal{R}} \mu(r)$,*

*where $\mu$ is defined on contexts and substitutions by*

$$\begin{array}{lcl}
\mu(\varnothing) & = & \varnothing \\
\mu(\Gamma, x : A) & = & \mu(\Gamma), x : \mu(A) \\
\mu(\theta, x \leftarrow M) & = & \mu(\theta), x \leftarrow \mu(M).
\end{array}$$

The first two conditions are the same as in LF: the constants of $\mathbb{S}$ must be mapped to terms of $\mathbb{T}$ that have the correct type. When extending theory morphisms from LF to $\lambda\Pi/\mathcal{R}$, we require as a third condition that, for every rewrite rule $\ell \hookrightarrow r$ of $\mathbb{S}$, we have the conversion $\mu(\ell) \equiv_{\beta\mathcal{R}} \mu(r)$ in $\mathbb{T}$ (or $\mu(\ell) \equiv_{\beta\eta\mathcal{R}} \mu(r)$ if we use the variant of $\lambda\Pi/\mathcal{R}$ with $\eta$). Under this condition, we will see that *convertibility* is preserved, i.e., if $t \equiv_{\beta(\eta)\mathcal{R}} u$ in $\mathbb{S}$ then $\mu(t) \equiv_{\beta(\eta)\mathcal{R}} \mu(u)$ in $\mathbb{T}$. Intuitively, the three conditions of theory morphisms ensure that the target theory has at least the logical and computational strength as the source theory.

**Remark 5.** *Felicissimo [Fel22, see Long version] required as a third condition that, for every rewrite rule $\ell \hookrightarrow r$ of $\mathbb{S}$, we have the rewriting $\mu(\ell) \hookrightarrow^*_{\beta\mathcal{R}} \mu(r)$ in $\mathbb{T}$ (where $\hookrightarrow^*_{\beta\mathcal{R}}$ is the reflexive and transitive closure of $\hookrightarrow_{\beta\mathcal{R}}$). Under this condition,* rewritability *is preserved, i.e., if $t \hookrightarrow^*_{\beta\mathcal{R}} u$ in $\mathbb{S}$ then $\mu(t) \hookrightarrow^*_{\beta\mathcal{R}} \mu(u)$ in $\mathbb{T}$. Felicissimo's condition on rewriting is sufficient to prove our condition on conversion, but it is not necessary. For instance, consider $A_1, A_2, A_3$ of type* Type *in $\mathbb{S}$, with $A_1 \hookrightarrow A_3$, and $B_1, B_2, B_3$ of type* Type *in $\mathbb{T}$, with $B_1 \hookrightarrow B_2$ and $B_3 \hookrightarrow B_2$. We set $\mu(A_i) = B_i$ for $i \in [\![1, 3]\!]$. We indeed have $\mu(A_1) = B_1 \equiv_{\beta\mathcal{R}} B_3 = \mu(A_3)$ in $\mathbb{T}$, but we do not have $\mu(A_1) \hookrightarrow^*_{\beta\mathcal{R}} \mu(A_3)$. For the Judgment Preservation theorem, we only need to preserve convertibility, hence our definition of theory morphisms is more general than Felicissimo's definition.*

**Example 5** (Identity morphism)**.** *Let $\mathbb{T}$ be a theory. The identity morphism from $\mathbb{T}$ to $\mathbb{T}$ maps each constant to itself, and therefore each term to itself. The conditions of theory morphism are trivially satisfied.*

**Example 6** (Morphism `MulDivGr` : `MulGr` $\to$ `DivGr`)**.** *We define a morphism* `MulDivGr` *from* `MulGr` *to* `DivGr`*. All the constants of* `PLeq` *are mapped to themselves.*

$\mu(\times) = \lambda x, y : \iota.\ x \div (1 \div y)$

$\mu(1) = 1$

$\mu(\mathsf{inv}) = \lambda x : \iota.\ 1 \div x$

*For every rewrite rule $\ell \hookrightarrow r$ of the multiplication group, we can easily show that $\mu(\ell)$ and $\mu(r)$ are convertible using the rewrite rules of* `DivGr`*. For the rewrite rule $x \times (\mathsf{inv}\ x) \hookrightarrow 1$, we have $\mu(x \times (\mathsf{inv}\ x)) \equiv_{\beta\mathcal{R}} x \div (1 \div (1 \div x))$. Since $(1 \div (1 \div x)) \hookrightarrow x$ and $x \div x \hookrightarrow 1$, we get $\mu(x \times (\mathsf{inv}\ x)) \equiv_{\beta\mathcal{R}} 1 \equiv_{\beta\mathcal{R}} \mu(1)$.*

**Example 7** (Morphism `DivMulGr` : `DivGr` → `MulGr`)**.** *We define a morphism* `DivMulGr` *from* `DivGr` *to* `MulGr`*. All the constants of* `PLeq` *are mapped to themselves.*

$\mu(\div) = \lambda x, y : \iota.\ x \times (\mathsf{inv}\ y)$
$\mu(1) = 1$

*For every rewrite rule $\ell \hookrightarrow r$ of* `DivGr`*, we can easily show that $\mu(\ell)$ and $\mu(r)$ are convertible using the rewrite rules of the multiplication group. For the the rewrite rule $1 \div (1 \div x) \hookrightarrow x$, we have $\mu(1 \div (1 \div x)) \equiv_{\beta\mathcal{R}} 1 \times (\mathsf{inv}\ (1 \times \mathsf{inv}\ x))$. Since $1 \times x \hookrightarrow x$ and $\mathsf{inv}\ (\mathsf{inv}\ x) \hookrightarrow x$, we get $\mu(1 \div (1 \div x)) \equiv_{\beta\mathcal{R}} x \equiv_{\beta\mathcal{R}} \mu(x)$.*

To show that `MulDivGr` and `DivMulGr` are isomorphisms, we have to show that the composition `MulDivGr`;`DivMulGr` and the identity morphism of `MulGr` are equal, and accordingly for the opposite composition. In Example 8 we show that the former condition indeed holds *definitionally*, i.e., up to $\equiv_{\beta\eta\mathcal{R}}$. Later in Example 12, we show how a *propositional* equality of morphisms can be proved in a situation where definitional equality is not strong enough.

**Example 8** (Morphism `MulGr` → `MulGr`)**.** *The composition* `MulDivGr`;`DivMulGr` *is a theory morphism from the multiplication group to itself. In particular, we get the following parameters.*

$\mu(\times) = \lambda x, y : \iota.\ x \times \mathsf{inv}\ (1 \times \mathsf{inv}\ y)$
$\mu(1) = 1$
$\mu(\mathsf{inv}) = \lambda x : \iota.\ 1 \times \mathsf{inv}\ x$

*Note that, thanks to the rewrite rules of* `MulGr`*, we have $\mu(\times) \equiv_{\beta\mathcal{R}} \lambda x, y : \iota.\ x \times y$ and $\mu(\mathsf{inv}) \equiv_{\beta\mathcal{R}} \lambda x : \iota.\ \mathsf{inv}\ x$. Therefore, the variant of $\lambda\Pi/\mathcal{R}$ with the $\eta$-rule suffices to have $\mu(\times) \equiv_{\beta\eta\mathcal{R}} \times$ and $\mu(\mathsf{inv}) \equiv_{\beta\eta\mathcal{R}} \mathsf{inv}$. In that case, the morphism* `MulDivGr`;`DivMulGr` *and the identity morphism of* `MulGr` *are definitionally equal.*

## 3.2 Judgment Preservation Theorem

The main property of theory morphisms is that this translation preserves convertibility and judgments. Once we have specified the parameters of a theory morphism, we can therefore translate any typing judgment from the source theory to the target theory. In particular, we can transfer proofs between different theories of $\lambda\Pi/\mathcal{R}$.

**Theorem 1** (Judgment Preservation)**.** *Let $\mu$ be a theory morphism from $\mathbb{S}$ to $\mathbb{T}$.*

1. *If $\vdash_{\mathbb{S}} \Gamma$, then $\vdash_{\mathbb{T}} \mu(\Gamma)$.*
2. *If $\Gamma \vdash_{\mathbb{S}} M : A$, then $\mu(\Gamma) \vdash_{\mathbb{T}} \mu(M) : \mu(A)$.*
3. *If $\Gamma \vdash_{\mathbb{S}} A : K$, then $\mu(\Gamma) \vdash_{\mathbb{T}} \mu(A) : \mu(K)$.*
4. *If $\Gamma \vdash_{\mathbb{S}} K : \mathsf{Kind}$, then $\mu(\Gamma) \vdash_{\mathbb{T}} \mu(K) : \mathsf{Kind}$.*

This theorem extends the Judgment Preservation theorem of [RS13] from LF to $\lambda\Pi/\mathcal{R}$, and generalizes the one of [Fel22] as we consider a broader definition of theory morphisms. The theorem relies on the substitution lemma, which states that morphism and substitution application commute with each other, and on the conversion lemma, which states that convertibility is preserved by the morphism.

**Lemma 2** (Substitution)**.** *Let $\mu$ be a theory morphism from $\mathbb{S}$ to $\mathbb{T}$, and $\theta$ be a substitution. Then, up to $\alpha$-renaming, we have:*

1. $\mu(M\theta) = \mu(M)\mu(\theta)$,
2. $\mu(A\theta) = \mu(A)\mu(\theta)$,
3. $\mu(K\theta) = \mu(K)\mu(\theta)$.

*Proof.* We show the first item by induction on $M$. Suppose that $M$ is a variable $x$. If $\theta$ does not substitute $x$, then by definition $\mu(\theta)$ does not substitute $x$ either, so $\mu(x\theta) = \mu(x) = \mu(x)\mu(\theta)$. If $\theta$ substitutes $x$ by $N$, then by definition $\mu(\theta)$ substitutes $x$ by $\mu(N)$, so $\mu(x\theta) = \mu(N) = \mu(x)\mu(\theta)$. The other cases are shown using the induction hypotheses. We prove the second and third items similarly. □

**Lemma 3** (Conversion)**.** *Let $\mu$ be a theory morphism from $\mathbb{S}$ to $\mathbb{T}$.*

1. *If $A \equiv_{\beta(\eta)\mathcal{R}} B$ in $\mathbb{S}$, then $\mu(A) \equiv_{\beta(\eta)\mathcal{R}} \mu(B)$ in $\mathbb{T}$.*
2. *If $K \equiv_{\beta(\eta)\mathcal{R}} K'$ in $\mathbb{S}$, then $\mu(K) \equiv_{\beta(\eta)\mathcal{R}} \mu(K')$ in $\mathbb{T}$.*

*Proof.* The proof proceeds by induction on the formation of $A \equiv_{\beta(\eta)\mathcal{R}} B$ and $K \equiv_{\beta(\eta)\mathcal{R}} K'$.

- By definition, we have $\mu((\lambda x : A.\ M)\ N) = (\lambda x : \mu(A).\ \mu(M))\ \mu(N)$, which $\beta$-reduces to $\mu(M)\mu([x \leftarrow N])$. By Lemma 2, we get $\mu((\lambda x : A.\ M)\ N) \equiv_{\beta\mathcal{R}} \mu(M[x \leftarrow N])$. We prove $\mu((\lambda x : A.\ B)\ M) \equiv_{\beta\mathcal{R}} \mu(B[x \leftarrow M])$ similarly.
- By definition, we have $\mu(\lambda x : A.\ M\ x) = \lambda x : \mu(A).\ \mu(M)\ x$, which $\eta$-reduces to $\mu(M)$.
- Let $\ell \hookrightarrow r \in \mathbb{S}$ and $\theta$ be a substitution. Using Lemma 2, we have $\mu(\ell\theta) = \mu(\ell)\mu(\theta)$ and $\mu(r\theta) = \mu(r)\mu(\theta)$. By definition of theory morphisms, we derive $\mu(\ell\theta) = \mu(\ell)\mu(\theta) \equiv_{\beta(\eta)\mathcal{R}} \mu(r)\mu(\theta) = \mu(r\theta)$.
- Closure by context, reflexivity, symmetry, and transitivity are immediate. □

*Proof of Theorem 1.* The proof proceeds by induction on the typing derivations. The cases App-Obj and App-Type rely on Lemma 2. The cases Conv-Type and Conv-Kind rely on Lemma 3. □

**Remark 6.** *Theory morphisms capture that derivability in the source theory implies derivability in the target theory. The reverse is not necessarily true and indeed often fails, e.g., if the target has greater logical and computational strength than the source. We get back to this in Section 8.*

### 3.3 Applications

Theory morphisms encompass many different translations between logics and between data structures. We give here three examples. The first one is a translation from a theory with axiomatized natural deduction to a theory with computational natural deduction. The second one is a translation from propositional logic to $Q_0$ logic. The third one is a translation from lists to binary trees.

### 3.3.1 From Deduction to Computation

In Example 1, we have encoded the natural deduction rules of the implication and the conjunction via typed constants. Alternatively, we can make use of the computational power of $\lambda\Pi/\mathcal{R}$ and represent natural deduction rules via rewrite rules [BDG+23].

$Prf\ (p \Rightarrow q) \hookrightarrow Prf\ p \to Prf\ q$
$Prf\ (p \wedge q) \hookrightarrow \Pi r : Prop.\ (Prf\ p \to Prf\ q \to Prf\ r) \to Prf\ r$

We can now define a theory morphism $\mu$ from the deductive encoding to the computational encoding of the natural deduction rules, thus showing that the latter is a refinement of the former. The constants shared by both encodings are mapped to themselves. The constants representing natural deduction rules are mapped to theorems proving them by using the rewrite rules. We map the introduction of implication to

$$\mu(\mathsf{imp_i}) = \lambda p, q : Prop.\ \lambda H : Prf\ p \to Prf\ q.\ H$$

of type $\Pi p, q : Prop.\ (Prf\ p \to Prf\ q) \to Prf\ (p \Rightarrow q)$, since $Prf\ p \to Prf\ q$ and $Prf\ (p \Rightarrow q)$ are convertible. Similarly, the left-elimination of the conjunction is mapped to

$$\mu(\mathsf{and_{e\ell}}) = \lambda p, q : Prop.\ \lambda H_{pq} : Prf\ (p \wedge q).\ H_{pq}\ p\ (\lambda H_p : Prf\ p.\ \lambda H_q : Prf\ q.\ H_p)$$

which has type $\Pi p, q : Prop.\ Prf\ (p \wedge q) \to Prf\ p$. The same idea applies for the remaining rules.

### 3.3.2 From Propositional Logic to $Q_0$ Logic

Andrews formulated $Q_0$ [And02], a version of higher-order logic in which the only primitive symbol is equality. All the other connectives and quantifiers are then built upon equality. $Q_0$ is for instance used in the HOL Light proof assistant.

**Example 9** ($Q_0$ logic)**.** *To encode $Q_0$ in $\lambda\Pi/\mathcal{R}$, we have to define a polymorphic equality. However, we cannot quantify on* Type*. We define the type Set of sorts, and El that maps sorts to the type of their elements. In doing so, we can quantify on sorts and embed them into types. The sort o encodes propositions. The arrow $\rightsquigarrow$ encodes function sorts. The rewrite rule on $\rightsquigarrow$ states that the type embedding a function sort indeed corresponds to a function type [BDG+23]. Prf maps propositions to the type of their proofs.*

$Set : \mathsf{Type}$ $\qquad$ $\rightsquigarrow : Set \to Set \to Set$ $\qquad$ $o : Set$
$El : Set \to \mathsf{Type}$ $\qquad$ $El\ (x \rightsquigarrow y) \hookrightarrow El\ x \to El\ y$ $\qquad$ $Prf : El\ o \to \mathsf{Type}$

*We define a polymorphic equality that satisfies the Leibniz rule, functional extensionality and propositional extensionality. For simplicity, $=\ a\ x\ y$ is written $x =_a y$. Contrary to the usual presentation of $Q_0$, we take advantage of $\lambda\Pi/\mathcal{R}$ and define the Leibniz rule through rewriting.*

$= : \Pi a : Set.\ El\ a \to El\ a \to El\ o$
$Prf\ (x =_a y) \hookrightarrow \Pi P : El\ a \to Prop.\ Prf\ (P\ x) \to Prf\ (P\ y)$
$\mathsf{propext} : \Pi p, q : Prop.\ (Prf\ p \to Prf\ q) \to (Prf\ q \to Prf\ p) \to Prf\ (p =_o q)$
$\mathsf{funext} : \Pi a, b : Set.\ \Pi f, g : El\ (a \rightsquigarrow b).\ (\Pi x : El\ a.\ Prf\ (f\ x =_b g\ x)) \to Prf\ (f =_{a \rightsquigarrow b} g)$

*In* funext*, the applications $f\ x$ and $g\ x$ are well-typed thanks to the rewrite rule on $\rightsquigarrow$.*

We define a theory morphism from PL to $Q_0$. The constant *Prf* is mapped to itself, and *Prop* is mapped to *El o*. Each connective is mapped to its encoding in $Q_0$.

$\mu(\wedge) = \lambda p, q : El\ o.\ (\lambda f.\ f\ p\ q) =_{(o \rightsquigarrow o \rightsquigarrow o) \rightsquigarrow o} (\lambda f.\ f\ \top\ \top)$

$\mu(\Rightarrow) = \lambda p, q : El\ o.\ (\mu(\wedge)\ p\ q) =_o p$

where $\top$ is an abbreviation for the term $(\lambda x.\ x) =_{o \rightsquigarrow o} (\lambda x.\ x)$. Note that the translation of $\wedge$ is well-typed only because of the rewrite rule on $\rightsquigarrow$. The translations of the natural deduction rules rely on funext, on propext and on intermediate lemmas on equality.

In LF, i.e., without rewriting, we would have to define an application constructor $app : \Pi a, b : Set.\ El\ (a \rightsquigarrow b) \to El\ a \to El\ b$ and an abstraction constructor $lam : \Pi a, b : Set.\ (El\ a \to El\ b) \to El\ (a \rightsquigarrow b)$ instead of the rewrite rule $El\ (x \rightsquigarrow y) \hookrightarrow El\ x \to El\ y$, along with an axiom for $\beta$-reduction. Similarly, the Leibniz rule would have been encoded by an axiom. Therefore, in LF, the user has to explicitly apply these constructors and axioms. In $\lambda\Pi/\mathcal{R}$, this work is discharged by the framework through rewriting, resulting in much simpler proof obligations for the user.

### 3.4 From Lists to Binary Trees

**Example 10** (Lists)**.** *The theory* List *defines the data structure of lists indexed by the sort of their elements. We use Set and El defined in Example 9 to define the polymorphic constructor* list*, which maps any sort $a$ to the type of the lists containing elements of sort $a$.* nil *creates an empty list.* cons *appends an element to a list.* hd *returns the head of a list and* tl *returns the tail of a list.* concat *concatenates two lists.*

$\mathsf{list} : Set \to Set$ $\quad \mathsf{cons} : \Pi a : Set.\ El\ a \to El\ (\mathsf{list}\ a) \to El\ (\mathsf{list}\ a)$ $\quad \mathsf{hd} : \Pi a : Set.\ El\ (\mathsf{list}\ a) \to El\ a$

$\mathsf{nil} : \Pi a : Set.\ El\ (\mathsf{list}\ a)$ $\quad \mathsf{concat} : \Pi a : Set.\ El\ (\mathsf{list}\ a) \to El\ (\mathsf{list}\ a) \to El\ (\mathsf{list}\ a)$ $\quad \mathsf{tl} : \Pi a : Set.\ El\ (\mathsf{list}\ a) \to El\ (\mathsf{list}\ a)$

*The semantic of these symbols is encoded using linearized rewrite rules, as explained in Remark 1.*

$\mathsf{hd}\ a\ (\mathsf{cons}\ a'\ x\ l) \hookrightarrow x$

$\mathsf{tl}\ a\ (\mathsf{cons}\ a'\ x\ l) \hookrightarrow l$

$\mathsf{concat}\ a\ (\mathsf{nil}\ a')\ l \hookrightarrow l$

$\mathsf{concat}\ a\ (\mathsf{cons}\ a'\ x\ l_1)\ l_2 \hookrightarrow \mathsf{cons}\ a\ x\ (\mathsf{concat}\ a\ l_1\ l_2)$

*The left-hand side of these rewrite rules are obviously ill typed, but this is not a problem: the rules preserve typing. Due to the typing constraints of the head symbols, these rewrite rules can only be used when $a$ and $a'$ are instantiated by the same sort.*

**Example 11** (Trees)**.** *The theory* Tree *defines the data structure of binary trees indexed by the sort of their elements.* leaf *creates the empty tree.* node *creates a new binary tree, by taking as arguments its root and the two children.* left *returns the left child,* right *returns the right child, and* root *returns the root.*

$\mathsf{tree} : Set \to Set$ $\quad \mathsf{left} : \Pi a : Set.\ El\ (\mathsf{tree}\ a) \to El\ (\mathsf{tree}\ a)$

$\mathsf{leaf} : \Pi a : Set.\ El\ (\mathsf{tree}\ a)$ $\quad \mathsf{right} : \Pi a : Set.\ El\ (\mathsf{tree}\ a) \to El\ (\mathsf{tree}\ a)$

$\mathsf{node} : \Pi a : Set.\ El\ a \to El\ (\mathsf{tree}\ a) \to El\ (\mathsf{tree}\ a) \to El\ (\mathsf{tree}\ a)$ $\quad \mathsf{root} : \Pi a : Set.\ El\ (\mathsf{tree}\ a) \to El\ a$

*We encode the semantics of* left*,* right *and* root *using rewriting. Again, we use linearized rules.*

$\mathsf{left}\ a\ (\mathsf{node}\ a'\ x\ l\ r) \hookrightarrow l$

$\mathsf{right}\ a\ (\mathsf{node}\ a'\ x\ l\ r) \hookrightarrow r$

$\mathsf{root}\ a\ (\mathsf{node}\ a'\ x\ l\ r) \hookrightarrow x$

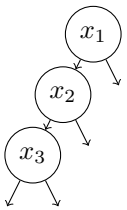
*We define by induction a composition of binary trees. The second tree is inductively composed with the left child of the first tree.*

$\mathsf{compo} : \Pi a : \mathit{Set}.\ \mathit{El}\ (\mathsf{tree}\ a) \to \mathit{El}\ (\mathsf{tree}\ a) \to \mathit{El}\ (\mathsf{tree}\ a)$

$\mathsf{compo}\ a\ (\mathsf{leaf}\ a')\ t \hookrightarrow t$

$\mathsf{compo}\ a\ (\mathsf{node}\ a'\ x\ l\ r)\ t \hookrightarrow \mathsf{node}\ a\ x\ (\mathsf{compo}\ a\ l\ t)\ r$

We can now define a theory morphism from `List` to `Tree`. The idea is to represent lists as binary trees with only left children. For instance, the list $[x_1, x_2, x_3]$ is mapped to the binary tree




Formally, the morphism maps *Set* and *El* to themselves. list, nil and concat are mapped to their counterparts in `Tree`. The parameters for cons, hd and tl are chosen so that we encode lists inside the left of binary trees.

$\mu(\mathsf{list}) = \mathsf{tree}$

$\mu(\mathsf{nil}) = \mathsf{leaf}$

$\mu(\mathsf{cons}) = \lambda a : \mathit{Set}.\ \lambda x : \mathit{El}\ a.\ \lambda l : \mathit{El}\ (\mathsf{tree}\ a).\ \mathsf{node}\ a\ x\ l\ (\mathsf{leaf}\ a)$

$\mu(\mathsf{hd}) = \lambda a : \mathit{Set}.\ \lambda l : \mathit{El}\ (\mathsf{tree}\ a).\ \mathsf{root}\ a\ l$

$\mu(\mathsf{tl}) = \lambda a : \mathit{Set}.\ \lambda l : \mathit{El}\ (\mathsf{tree}\ a).\ \mathsf{left}\ a\ l$

$\mu(\mathsf{concat}) = \lambda a : \mathit{Set}.\ \lambda l_1, l_2 : \mathit{El}\ (\mathsf{tree}\ a).\ \mathsf{compo}\ a\ l_1\ l_2$

The conditions on the rewrite rules of `List` are satisfied in `Tree`. We only show the most interesting case.

$$\begin{array}{rcl} \mu(\mathsf{concat}\ a\ (\mathsf{cons}\ a\ x\ l_1)\ l_2) & \equiv_{\beta\mathcal{R}} & \mathsf{compo}\ a\ (\mathsf{node}\ a\ x\ l_1\ (\mathsf{leaf}\ a))\ l_2 \\ & \equiv_{\beta\mathcal{R}} & \mathsf{node}\ a\ x\ (\mathsf{compo}\ a\ l_1\ l_2)\ (\mathsf{leaf}\ a) \\ & \equiv_{\beta\mathcal{R}} & \mu(\mathsf{cons}\ a\ x\ (\mathsf{concat}\ a\ l_1\ l_2)) \end{array}$$

Note that the conditions on the rewrite rules only need to be satisfied by the versions of the rules *before* linearization. Indeed, we only use instances of the linearized rules where the left-hand side is well typed, i.e., we only use instances of the rules before linearization.

We are therefore able to translate lists into binary trees, where both data structures are encoded using the computational power of $\lambda\Pi/\mathcal{R}$.

# 4 Logical Relations

In this section, we extend logical relations in the sense of [RS13] to $\lambda\Pi/\mathcal{R}$, and we prove the main theorem about them, often called the Basic lemma or Abstraction theorem. The technical details of logical relations are much trickier than those of theory morphisms. But our proof of the Abstraction theorem for $\lambda\Pi/\mathcal{R}$ will be simpler than the proof for LF [RS13] despite the added generality.

## 4.1 Formal Definition

A theory morphism $\mu$ maps the judgment $\Gamma \vdash_{\mathbb{S}} M : A$ to the judgment $\mu(\Gamma) \vdash_{\mathbb{T}} \mu(M) : \mu(A)$. A logical relation $\rho$ on $\mu$ states and proves an additional invariant satisfied by $\mu$: it maps the judgment $\Gamma \vdash_{\mathbb{S}} M : A$ to the judgment $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M) : \rho(A)\ \mu(M)$. Here every type $A$ is mapped to a predicate $\rho(A) : \mu(A) \to \mathsf{Type}$ and every term $M : A$ is mapped to a proof $\rho(M)$ that $\mu(M)$ satisfies $\rho(A)$.

The function $\rho$ duplicates every (free or bound) variable so that every fresh variable $x : A$ yields both its translation $x : \mu(A)$ and an assumption $x^* : \rho(A)\ x$ that $x$ satisfies the invariant. Thus, the translation of an abstraction $\lambda x : A.\ M$ is $\lambda x : \mu(A).\ \lambda x^* : \rho(A)\ x.\ \rho(M)$. Accordingly, the translation of an application $M\ N$ is $\rho(M)\ \mu(N)\ \rho(N)$, i.e., it supplies both the translated argument and the proof of its invariant.

The definition of the logical relation of a function type $\Pi x : A.\ B$ is the well-known condition that functions must preserve the relation: $\rho(\Pi x : A.\ B)$ holds for a function $f : \mu(\Pi x : A.\ B)$ if for every $x : \mu(A)$ satisfying $\rho(A)$, the term $(f\ x)$ satisfies $\rho(B)$. Thus, $\rho(\Pi x : A.\ B)$ is given by $\lambda f : \mu(\Pi x : A.\ B).\ \Pi x : \mu(A).\ \Pi x^* : \rho(A)\ x.\ \rho(B)\ (f\ x)$.

Following the same idea, we would like to define $\rho(\Pi x : A.\ K) = \lambda f : \mu(\Pi x : A.\ K).\ \Pi x : \mu(A).\ \Pi x^* : \rho(A)\ x.\ \rho(K)\ (f\ x)$. This works, with a little extra effort, in type theories with higher universes. However, in $\lambda\Pi/\mathcal{R}$, such a term is ill typed because $\mu(\Pi x : A.\ K)$ is a kind. To get around this issue, we insert an extra parameter $R$ to the translation: if $R$ has type $\Pi x : A.\ K$ then we define $\rho^R(\Pi x : A.\ K) = \Pi x : \mu(A).\ \Pi x^* : \rho(A)\ x.\ \rho^{R\ x}(K)$, and if $R$ has type $\mathsf{Type}$ then we define $\rho^R(\mathsf{Type}) = \mu(R) \to \mathsf{Type}$.

More generally, we can define $n$-ary logical relations for theory morphisms $\mu_1, \ldots, \mu_n$ from $\mathbb{S}$ to $\mathbb{T}$. Such a $n$-ary logical relation $\rho$ maps every type to an $n$-ary predicate and every term $M : A$ to a proof of $\rho(A)\ \mu_1(M)\ \ldots\ \mu_n(M)$.

**Conventions** Let $\mu_1, \ldots, \mu_n$ be $n$ theory morphisms from $\mathbb{S}$ to $\mathbb{T}$. Without loss of generality, we consider that each $\mu_i$ maps variables $x$ to $x_i$. We use the following notations:

- We write $\vec{x} : \vec{\mu}(A)$ for the context $x_1 : \mu_1(A), \ldots, x_n : \mu_n(A)$.
- We write $[\vec{x} \leftarrow \vec{\mu}(M)]$ for the substitution $[x_1 \leftarrow \mu_1(M), \ldots, x_n \leftarrow \mu_n(M)]$.
- We write $\lambda\vec{x} : \vec{\mu}(A).\ t$ for $\lambda x_1 : \mu_1(A).\ \ldots\ \lambda x_n : \mu_n(A).\ t$.
- We write $\Pi\vec{x} : \vec{\mu}(A).\ t$ for $\Pi x_1 : \mu_1(A).\ \ldots\ \Pi x_n : \mu_n(A).\ t$. Similarly, we write $\vec{\mu}(A) \to t$ for $\mu_1(A) \to \ldots \to \mu_n(A) \to t$.
- Given a list of terms $\vec{M} = M_1, \ldots, M_n$, we write $t\ \vec{M}$ for the application $t\ M_1\ \ldots\ M_n$.

**Definition 4** (Logical relation)**.** *Let $\mu_1, \ldots, \mu_n$ be theory morphisms from $\mathbb{S}$ to $\mathbb{T}$. The mapping $\rho$ defined inductively from a set of parameters $\rho_c$ and $\rho_a$ by*

$$
\begin{array}{lcl}
\rho(x) & = & x^* \\
\rho(c) & = & \rho_c \\
\rho(a) & = & \rho_a \\
\rho(M\ N) & = & \rho(M)\ \vec{\mu}(N)\ \rho(N) \\
\rho(A\ M) & = & \rho(A)\ \vec{\mu}(M)\ \rho(M) \\
\rho(\lambda x : A.\ M) & = & \lambda\vec{x} : \vec{\mu}(A).\ \lambda x^* : \rho(A)\ \vec{x}.\ \rho(M) \\
\rho(\lambda x : A.\ B) & = & \lambda\vec{x} : \vec{\mu}(A).\ \lambda x^* : \rho(A)\ \vec{x}.\ \rho(B) \\
\rho(\Pi x : A.\ B) & = & \lambda\vec{f} : \vec{\mu}(\Pi x : A.\ B).\ \Pi\vec{x} : \vec{\mu(A)}.\ \Pi x^* : \rho(A)\ \vec{x}.\ \rho(B)\ (f_1\ x_1)\ \ldots\ (f_n\ x_n) \\
\rho^R(\Pi x : A.\ K) & = & \Pi\vec{x} : \vec{\mu}(A).\ \Pi x^* : \rho(A)\ \vec{x}.\ \rho^{R\ x}(K) \\
\rho^R(\mathsf{Type}) & = & \vec{\mu}(R) \to \mathsf{Type} \\
\rho(\mathsf{Kind}) & = & \mathsf{Kind}
\end{array}
$$

*is a logical relation on $\mu$ when:*

1. *for every constant $c : A \in \mathbb{S}$, there exists a term $\rho_c$ such that $\vdash_{\mathbb{T}} \rho_c : \rho(A)\ \vec{\mu}(c)$,*
2. *for every constant $a : K \in \mathbb{S}$, there exists a term $\rho_a$ such that $\vdash_{\mathbb{T}} \rho_a : \rho^a(K)$,*
3. *for every rewrite rule $\ell \hookrightarrow r \in \mathbb{S}$, we have $\rho(\ell) \equiv_{\beta(\eta)\mathcal{R}} \rho(r)$,*

*where $\rho$ is defined on contexts and substitutions by*

$$\begin{array}{lcl} \rho(\varnothing) & = & \varnothing \\ \rho(\Gamma, x : A) & = & \rho(\Gamma),\ \vec{x} : \vec{\mu}(A),\ x^* : \rho(A)\ \vec{x} \\ \rho(\theta, x \leftarrow N) & = & \rho(\theta),\ \vec{x} \leftarrow \vec{\mu}(N),\ x^* \leftarrow \rho(N). \end{array}$$

The first two conditions are the same as in LF. The third condition, specific to $\lambda\Pi/\mathcal{R}$, is necessary so that convertibility is preserved by logical relations.

Note that for every variable $x$ that occurs in a term $t$, the two variables $x_i$ and $x^*$ occur in the translated term $\rho_i(t)$. Therefore, if $\Gamma$ declares $m$ variables, $\rho(\Gamma)$ declares $m(n+1)$ variables.

In Example 8, we discussed how the equality between `MulDivGr;DivMulGr` and the identity of `MulDivGr` can be offloaded to the definitional equality of $\lambda\Pi/\mathcal{R}$ extended with $\eta$-reduction. In general, however, definitional equality—even with $\eta$-reduction—is not sufficient to show the equality between two morphisms, and we have to resort to a propositional equality. Those situations are one of the applications of logical relations:

**Example 12.** *In Example 8, we showed that* `MulDivGr;DivMulGr` *and the identify of* `MulGr` *are definitionally equal in the $\lambda\Pi/\mathcal{R}$ meta-logic if we use $\eta$. But for the sake of example, we now show how to prove propositional equality up to object-logic equality on terms and an object-logic encoded equivalence on propositions. To formalize that argument, we capture the invariant as a binary logical relation on* `MulDivGr;DivMulGr` *and on the identity of* `MulGr`*.*

*We first define a binary logical relation on* `PLeq` *that captures our proof obligations: the two translations of any element of type $\iota$ must be equal, and the two translations of any proposition must be equivalent.*

$\rho(\iota) = \lambda x_1, x_2 : \iota.\ Prf\ (x_1 = x_2)$
$\rho(Prop) = \lambda p_1, p_2 : Prop.\ Prf\ ((p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1))$

*We also have to define an invariant for proofs. Without proof-irrelevance, our logical relation must use a trivially true relation on proof terms. That is inessential if we assume proof-irrelevance: then we can simply skip the proof obligations for proofs.*

$\rho(Prf) = \lambda p_1, p_2 : Prop.\ \lambda H : Prf((p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)).\ unit$

*Defining $\rho$ for the remaining constants is straightforward.*

*We extend $\rho$ to the constants of* `MulGr` *as follows. The parameter $\rho(\times)$ is a proof that if $x_1 = x_2$ and $y_1 = y_2$ then $x_1 \times y_1 = x_2 \times y_2$.*

$\rho(\times) = \lambda x_1, x_2 : \iota.\ \lambda H_x : Prf\ (x_1 = x_2).\ \lambda y_1, y_2 : \iota.\ \lambda H_y : Prf\ (y_1 = y_2).$
$\quad \mathsf{leib}\ x_1\ x_2\ H_x\ (\lambda z.\ x_1 \times y_1 = z \times y_2)\ (\mathsf{leib}\ y_1\ y_2\ H_y\ (\lambda z.\ x_1 \times y_1 = x_1 \times z)\ (\mathsf{refl}\ (x_1 \times y_1)))$

*The parameter $\rho(1)$ is a proof that $1 = 1$, and the parameter $\rho(\mathsf{inv})$ is a proof that if $x_1 = x_2$ then $\mathsf{inv}\ x_1 = \mathsf{inv}\ x_2$.*

$\rho(1) = \mathsf{refl}\ 1$
$\rho(\mathsf{inv}) = \lambda x_1, x_2 : \iota.\ \lambda H : Prf\ (x_1 = x_2).\ \mathsf{leib}\ x_1\ x_2\ H\ (\lambda z : \iota.\ \mathsf{inv}\ x_1 = \mathsf{inv}\ z)\ (\mathsf{refl}\ (\mathsf{inv}\ x_1))$

*We can easily express the remaining parameters.*

*To ensure this is a well-formed logical relation, we have to check the condition on the rewrite rules. For every rewrite rule $\ell \hookrightarrow r$ of* `MulGr`*, $\ell$ and $r$ have type $\iota$. Therefore $\rho(\ell)$ and $\rho(r)$ are proofs, so that these conditions are trivial under proof irrelevance.*

## 4.2 Abstraction Theorem

We have seen that, if $M$ has type $A$, we want $\rho(M)$ to be of type $\rho(A)\ \mu(M)$, where $\rho(A)$ is intuitively an invariant that must be satisfied by $\mu(M)$. The Abstraction theorem extends this idea to the three-level hierarchy of LF and $\lambda\Pi/\mathcal{R}$.

**Theorem 2** (Abstraction)**.** *Let $\rho$ be a logical relation on $\mu_1, \ldots, \mu_n$.*

1. *If $\vdash_{\mathbb{S}} \Gamma$, then $\vdash_{\mathbb{T}} \rho(\Gamma)$.*
2. *If $\Gamma \vdash_{\mathbb{S}} M : A$ and $\Gamma \vdash_{\mathbb{S}} A : \mathsf{Type}$, then $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M) : \rho(A)\ \vec{\mu}(M)$.*
3. *If $\Gamma \vdash_{\mathbb{S}} A : K$ and $\Gamma \vdash_{\mathbb{S}} K : \mathsf{Kind}$, then $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \rho^A(K)$.*
4. *If $\vdash_{\mathbb{S}} K : \mathsf{Kind}$ and $\Gamma \vdash_{\mathbb{S}} A : K$, then $\rho(\Gamma) \vdash_{\mathbb{T}} \rho^A(K) : \mathsf{Kind}$.*

Note that, in [RS13], the preservation of conversion is part of the Abstraction theorem. This is because in [RS13], the conversion is typed, so that both conversion and typing depend on each other and both proofs must be intertwined. With our untyped conversion, the preservation of conversion can be proved separately and before proving the preservation of typing, making the proofs much simpler. Our proof needs only a simple substitution lemma and a conversion lemma.

**Lemma 4** (Substitution)**.** *Let $\rho$ be a logical relation on $\mu_1, \ldots, \mu_n$, and $\theta$ be a substitution. Then, up to $\alpha$-renaming, we have:*

1. *$\rho(M\theta) = \rho(M)\rho(\theta)$,*
2. *$\rho(A\theta) = \rho(A)\rho(\theta)$,*
3. *$\rho^{A\theta}(K\theta) = \rho^A(K)\rho(\theta)$.*

*Proof.* By induction on the terms $M$, $A$ and $K$. □

**Lemma 5** (Conversion)**.** *Let $\rho$ be a logical relation on $\mu_1, \ldots, \mu_n$.*

1. *If $A \equiv_{\beta(\eta)\mathcal{R}} B$ in $\mathbb{S}$, then $\rho(A) \equiv_{\beta(\eta)\mathcal{R}} \rho(B)$ in $\mathbb{T}$.*
2. *If $K \equiv_{\beta(\eta)\mathcal{R}} K'$ in $\mathbb{S}$ then $\rho^R(K) \equiv_{\beta(\eta)\mathcal{R}} \rho^R(K')$ in $\mathbb{T}$.*

*Proof.* The proof proceeds by induction on the formation of $A \equiv_{\beta(\eta)\mathcal{R}} B$ and $K \equiv_{\beta(\eta)\mathcal{R}} K'$.

- We have $\rho((\lambda x : A.\ M)\ N) = (\lambda \vec{x} : \vec{\mu}(A).\ \lambda x^* : \rho(A)\ \vec{x}.\ \rho(M))\ \vec{\mu}(N)\ \rho(N)$, which $\beta$-reduces to $\rho(M)\rho([x \leftarrow N])$. By Lemma 4, we derive $\rho((\lambda x : A.\ M)\ N) \equiv_{\beta\mathcal{R}} \rho(M[x \leftarrow N])$. Similarly, we have $\rho((\lambda x : A.\ B)\ M) \equiv_{\beta\mathcal{R}} \rho(B[x \leftarrow M])$.
- We have $\rho(\lambda x : A.\ M\ x) = \lambda \vec{x} : \vec{\mu}(A).\ \lambda x^* : \rho(A)\ \vec{x}.\ \rho(M)\ \vec{x}\ x^*$, which $\eta$-reduces to $\rho(M)$.

- Let $\ell \hookrightarrow r \in \mathbb{S}$ and $\theta$ be a substitution. Using Lemma 4, we have $\rho(\ell\theta) = \rho(\ell)\rho(\theta)$ and $\rho(r\theta) = \rho(r)\rho(\theta)$. By definition, we derive $\rho(\ell\theta) = \rho(\ell)\rho(\theta) \equiv_{\beta(\eta)\mathcal{R}} \rho(r)\rho(\theta) = \rho(r\theta)$.
- Closure by context, reflexivity, symmetry, and transitivity are immediate and relies on Lemma 3. □

*Proof of Theorem 2.* We proceed by induction on the derivations.

- EMPTY: Since $\rho(\varnothing) = \varnothing$, we derive $\vdash_{\mathbb{T}} \rho(\varnothing)$ using EMPTY.
- DECL: By induction, we have $\vdash_{\mathbb{T}} \rho(\Gamma)$ and $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \vec{\mu}(A) \to \mathsf{Type}$. Using Theorem 1, we have $\mu_i(\Gamma) \vdash_{\mathbb{T}} \mu_i(A) : \mathsf{Type}$. Since $x \notin \Gamma$, we have $x_i \notin \rho(\Gamma)$ and $x^* \notin \rho(\Gamma)$. We derive $\vdash_{\mathbb{T}} \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x}$ using weakening and DECL several times.
- SORT: Suppose that $\Gamma \vdash_{\mathbb{S}} A : \mathsf{Type}$ for some $A$. We have $\vdash_{\mathbb{T}} \rho(\Gamma)$ by induction hypothesis. Using Theorem 1, we get $\mu_i(\Gamma) \vdash_{\mathbb{T}} \mu_i(A) : \mathsf{Type}$. Using weakening and PROD-KIND several times, we derive $\rho(\Gamma) \vdash_{\mathbb{T}} \vec{\mu}(A) \to \mathsf{Type} : \mathsf{Kind}$.
- CONST-OBJ: We get $\vdash_{\mathbb{T}} \rho(\Gamma)$ by induction hypothesis. We know that $\vdash_{\mathbb{T}} \rho_c : \rho(A)\ \vec{\mu}(c)$. Using weakening, we derive $\rho(\Gamma) \vdash_{\mathbb{T}} \rho_c : \rho(A)\ \mu_1(c)\ \ldots\ \mu_n(c)$.
- CONST-TYPE: We get $\vdash_{\mathbb{T}} \rho(\Gamma)$ by induction hypothesis. We know that $\vdash_{\mathbb{T}} \rho_a : \rho^a(K)$. Using weakening, we derive $\rho(\Gamma) \vdash_{\mathbb{T}} \rho_a : \rho^a(K)$.
- VAR: By induction, we have $\vdash_{\mathbb{T}} \rho(\Gamma)$. Since $x : A \in \Gamma$, we have $x^* : \rho(A)\ \vec{x} \in \rho(\Gamma)$. We derive $\rho(\Gamma) \vdash_{\mathbb{T}} x^* : \rho(A)\ \vec{x}$ using VAR.
- PROD-TYPE: By induction, we have

$$\begin{array}{rl} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \vec{\mu}(A) \to \mathsf{Type} \\ \text{and} & \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho(B) : \vec{\mu}(B) \to \mathsf{Type}. \end{array}$$

  Using weakening and then APP-OBJ and APP-TYPE several times, we get

$$\rho(\Gamma), \vec{f} : \vec{\mu}(\Pi x : A.\ B), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho(B)\ (f_1\ x_1)\ \ldots\ (f_n\ x_n) : \mathsf{Type}.$$

  Using PROD-TYPE and ABS-TYPE several times, we derive

$$\begin{array}{rl} \rho(\Gamma) \vdash_{\mathbb{T}} & \lambda\vec{f} : \vec{\mu}(\Pi x : A.\ B).\ \Pi\vec{x} : \vec{\mu}(A).\ \Pi x^* : \rho(A)\ \vec{x}. \\ & \rho(B)\ (f_1\ x_1)\ \ldots\ (f_n\ x_n) : \vec{\mu}(\Pi x : A.\ B) \to \mathsf{Type}. \end{array}$$

- PROD-KIND: Suppose that $\Gamma \vdash_{\mathbb{S}} R : \Pi x : A.\ K$ for some $R$. By induction, we have

$$\begin{array}{rl} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \vec{\mu}(A) \to \mathsf{Type} \\ \text{and} & \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho^{R\ x}(K) : \mathsf{Kind}. \end{array}$$

  Using PROD-KIND several times, we derive $\rho(\Gamma) \vdash_{\mathbb{T}} \Pi\vec{x} : \vec{\mu}(A).\ \Pi x^* : \rho(A)\ \vec{x}.\ \rho^{R\ x}(K) : \mathsf{Kind}$.
- ABS-OBJ: By induction, we have

$$\begin{array}{rl} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \vec{\mu}(A) \to \mathsf{Type} \\ \text{and} & \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho(B) : \vec{\mu}(B) \to \mathsf{Type} \\ \text{and} & \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho(M) : \rho(B)\ \vec{\mu}(M). \end{array}$$

Using Abs-Obj several times, we derive

$$\rho(\Gamma) \vdash_{\mathbb{T}} \lambda \vec{x} : \vec{\mu}(A).\ \lambda x^* : \rho(A)\ \vec{x}.\ \rho(M) : \Pi \vec{x} : \vec{\mu}(A).\ \Pi x^* : \rho(A)\ \vec{x}.\ \rho(B)\ \vec{\mu}(M).$$

Using Conv-Type, we get $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(\lambda x : A.\ M) : \rho(\Pi x : A.\ B)\ \vec{\mu}(\lambda x : A.\ M)$.

- Abs-Type: By induction, we have

$$\begin{array}{rl} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \vec{\mu}(A) \to \mathsf{Type} \\ \text{and} & \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho^B(K) : \mathsf{Kind} \\ \text{and} & \rho(\Gamma), \vec{x} : \vec{\mu}(A), x^* : \rho(A)\ \vec{x} \vdash_{\mathbb{T}} \rho(B) : \rho^B(K). \end{array}$$

We derive $\rho(\Gamma) \vdash_{\mathbb{T}} \lambda \vec{x} : \vec{\mu}(A).\ \lambda x^* : \rho(A)\ \vec{x}.\ \rho(B) : \Pi \vec{x} : \vec{\mu}(A).\ \Pi x^* : \rho(A)\ \vec{x}.\ \rho^B(K)$ using Abs-Type several times. Using Conv-Kind and the fact that $\rho^{(\lambda x:A.\ B)\ x}(K) \equiv_{\beta\mathcal{R}} \rho^B(K)$, we get $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(\lambda x : A.\ B) : \rho^{\lambda x:A.\ B}(\Pi x : A.\ K)$.

- App-Obj: By induction, we have

$$\begin{array}{rl} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(M) : \Pi \vec{x} : \vec{\mu}(A).\ \Pi x^* : \rho(A)\ \vec{x}.\ \rho(B)\ (\mu_1(M)\ x_1)\ \ldots\ (\mu_n(M)\ x_n) \\ \text{and} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(N) : \rho(A)\ \vec{\mu}(N). \end{array}$$

Using Theorem 1 and weakening, we get $\rho(\Gamma) \vdash_{\mathbb{T}} \mu_i(N) : \mu_i(A)$. Using App-Obj several times, we derive

$$\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M)\ \vec{\mu}(N)\ \rho(N) : (\rho(B)\ (\mu_1(M)\ x_1)\ \ldots\ (\mu_n(M)\ x_n))[\vec{x} \leftarrow \vec{\mu}(N), x^* \leftarrow \rho(N)],$$

that is

$$\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M)\ \vec{\mu}(N)\ \rho(N) : \rho(B)[\vec{x} \leftarrow \vec{\mu}(N), x^* \leftarrow \rho(N)]\ (\mu_1(M)\ \mu_1(N))\ \ldots\ (\mu_n(M)\ \mu_n(N)).$$

Using Lemma 4, we derive $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M\ N) : \rho(B[x \leftarrow N])\ \vec{\mu}(M\ N)$.

- App-Type: By induction, we have $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \Pi \vec{x} : \vec{\mu}(B).\ \Pi x^* : \rho(B)\ \vec{x}.\ \rho^{A\ x}(K)$ and $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M) : \rho(B)\ \vec{\mu}(M)$. Using Theorem 1 and weakening, we get

$$\rho(\Gamma) \vdash_{\mathbb{T}} \mu_i(M) : \mu_i(B).$$

Using App-Type several times, we derive

$$\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A)\ \vec{\mu}(M)\ \rho(M) : \rho^{A\ x}(K)[\vec{x} \leftarrow \vec{\mu}(M), x^* \leftarrow \rho(M)].$$

Using Lemma 4, we obtain $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A\ M) : \rho^{A\ M}(K[x \leftarrow M])$.

- Conv-Type: By induction, we have

$$\begin{array}{rl} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(M) : \rho(A)\ \vec{\mu}(M) \\ \text{and} & \rho(\Gamma) \vdash_{\mathbb{T}} \rho(B) : \vec{\mu}(B) \to \mathsf{Type}. \end{array}$$

Since we have $A \equiv_{\beta(\eta)\mathcal{R}} B$ in $\mathbb{S}$, we have $\rho(A) \equiv_{\beta(\eta)\mathcal{R}} \rho(B)$ in $\mathbb{T}$ using Lemma 5. We derive $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(M) : \rho(B)\ \vec{\mu}(M)$ using Conv-Type.

- Conv-Kind: By induction, we have $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \rho^A(K)$ and $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(K') : \mathsf{Kind}$. Since $K \equiv_{\beta(\eta)\mathcal{R}} K'$ in $\mathbb{S}$, we have $\rho^A(K) \equiv_{\beta(\eta)\mathcal{R}} \rho^A(K')$ in $\mathbb{T}$ using Lemma 5. We derive $\rho(\Gamma) \vdash_{\mathbb{T}} \rho(A) : \rho^A(K')$ using Conv-Kind. □

# 5 Sort-Erasure Translations

In this section, we represent sort-erasure translations from hard-sorted to soft-sorted to unsorted logic. Here "sort" is the word that we will use for object-logic types to avoid any confusion with the types of $\lambda\Pi/\mathcal{R}$. The first translation erases hard sorting and instead captures the sorting relation via a special judgment. The second translation erases sorts entirely and captures them as unary predicates. Both of these translations have proved challenging, and to our knowledge, this is the first time that such translations are fully defined and verified. Our treatment will reveal several subtle critical design choices.

A key insight to formalize these is to adjust the target theories with additional features that allow carrying the sorting properties throughout the translation. Specifically, we encode dependent pairs to group a term with a proof about it, and we add a dependent variant of implication in order to access such proofs while building propositions.

## 5.1 Hard-Sorted, Soft-Sorted and Unsorted Logic

We first formalize unsorted logic `UFOL`, soft-sorted logic `SFOL`, and hard-sorted logic `HFOL`.

**Example 13** (Unsorted Logic)**.** *In unsorted logic* `UFOL`*, all terms have the generic type tm.*

$tm : \mathsf{Type}$ $\quad$ $Prop : \mathsf{Type}$ $\quad$ $Prf : Prop \to \mathsf{Type}$

*We define an implication* $\Rightarrow$*, along with a rewrite rule that subsumes its natural deduction rules.*

$\Rightarrow : Prop \to Prop \to Prop$
$Prf\ (p \Rightarrow q) \hookrightarrow Prf\ p \to Prf\ q$

*We also add the usual universal quantifier to exemplify the treatment of binders:*

$\forall : (tm \to Prop) \to Prop$
$Prf\ (\forall\ p) \hookrightarrow \Pi x : tm.\ Prf\ (p\ x)$

Sorted logic arises by adding a constant *Set* for object-logic sorts and allows quantification over sorted object-logic terms. There are two variants to define, going back to the definitions by Curry and Church, which we will refer to as soft-sorted logic `SFOL` and hard-sorted logic `HFOL`.

**Example 14** (Soft-Sorted Logic)**.** *The basic structure of the theory* `SFOL` *arises from that of* `UFOL` *by adding a type Set of sorts and an external predicate* $\#$ *to capture the sorting of terms:*

$tm : \mathsf{Type}$ $\quad$ $Set : \mathsf{Type}$ $\quad$ $Prop : \mathsf{Type}$ $\quad$ $Prf : Prop \to \mathsf{Type}$ $\quad$ $\# : tm \to Set \to \mathsf{Type}$

*The definition of the implication is the same as in* `UFOL`*. But we change the universal quantifier, which is unsorted in* `UFOL`*, to a polymorphic version that takes as an additional argument the sort a over which it quantifies.*

$\forall : \Pi a : Set.\ (\Pi x : tm.\ x\ \#\ a \to Prop) \to Prop$
$Prf\ (\forall\ a\ p) \hookrightarrow \Pi x : tm.\ \Pi h : x\ \#\ a.\ Prf\ (p\ x\ h)$

*The body of the quantifier binds two assumptions: the bound variable and a proof that it has the sort a. The latter ensures that bound variables are always well-sorted and thus that variables range over well-sorted objects only. Similarly, any extension of* `SFOL` *to* $\lambda$*-calculus requires guarding the bound variable of the* $\lambda$*-abstraction.*

**Example 15** (Hard-Sorted Logic)**.** *Hard-sorted logic* `HFOL` *uses the type of sorts Set and the injection El that maps a sort to the type of its elements.*

$Set : \mathsf{Type}$ $El : Set \to \mathsf{Type}$ $Prop : \mathsf{Type}$ $Prf : Prop \to \mathsf{Type}$

*Thus, object-logic terms of sort a have type El a.*

*The encoding of the implication is the same as for* `UFOL` *and* `SFOL`*. The polymorphic universal quantifier of* `HFOL` *differs from that of* `SFOL` *by binding a sorted variable as a single variable of the framework:*

$\forall : \Pi a : Set.\ (El\ a \to Prop) \to Prop$

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

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

*Note that we encode the semantics of the universal quantifier here using two axioms as opposed to the rewrite rule used in* `UFOL` *and* `SFOL`*. This is due to a technical issue that we will explain in Section 5.4.*

## 5.2 From Hard-Sorted Logic to Soft-Sorted Logic

**Overview** To translate from `HFOL` to `SFOL`, the first intuition is to proceed in two steps: we erase the sorting information using a theory morphism, and then we recover it using a logical relation. This approach is one of the example of [RS13], with $\lambda$-binding instead of quantifiers.

The first step is to define a morphism, mapping sorts to themselves and *erasing* the hard sort information by mapping every type $El\ a$ to the type $tm$. The constants $Set$, $Prop$ and $Prf$ are mapped to themselves.

$\mu(El) = \lambda a : Set.\ tm$

$\mu(\forall) = \lambda a : Set.\ \lambda p : tm \to Prop.\ \forall\ a\ (\lambda x : tm.\ \lambda h : x \#\ a.\ p\ x)$

Then, in a second step, we recover the sort information by proving that whenever we have $t : El\ a$ in `HFOL`, we can show (i.e., give a term of type) $\mu(t) \#\ \mu(a)$ in `SFOL`. To do so, we define a unary logical relation on the morphism. Terms of type $Set$ are mapped by the morphism to themselves, so $\rho(Set)$ can be any trivially satisfied predicate, for instance $\forall\ (\lambda p.\ p \Rightarrow p)$.

$\rho(Set) = \lambda a : Set.\ Prf\ (\forall\ (\lambda p.\ p \Rightarrow p))$

$\mu(El) = \lambda a : Set.\ \lambda a^* : Prf\ (\forall\ (\lambda p.\ p \Rightarrow p)).\ \lambda x : tm.\ x \#\ a$

So far, we have only translated the *syntax* of the language. To prove the translation sound, we must also translate the *proof rules*. However, we encountered a problem when extending the morphism to the proof rules. The proof rule $\mathsf{all_e}$ takes a term argument $x : El\ a$. Therefore $\mu(\mathsf{all_e})$ takes a term argument $x : tm$. But to define $\mu(\mathsf{all_e})$ in `SFOL`, we must have access to the sort-preservation invariant $\mu(x) \#\ \mu(a)$. Thus, we cannot define the morphism for the proof rules without already using the logical relation on the syntax—whereas the logical relation must be defined after the theory morphism. The translation in [RS13] worked out because it did not cover the proof rules.

One way out of this is to define a mutually recursive morphism and relation. This is essentially the approach followed in [Tra24b] for particular theories of $\lambda\Pi/\mathcal{R}$. We have generalized that idea to a systematic meta-theorem for $\lambda\Pi/\mathcal{R}$. It allowed the morphism to flexibly make use of the invariant established by the relation whenever it was needed. But the use of this formalism became very complex.

Instead, we have identified a third approach as the most scalable: we pair up the morphism and the translation, and define both at once. For example, we map $El\ a$ to the dependent pair type $\Sigma x : tm.\ x \#\ a$. Thus, every term

in the image of the translation always carries its well-sortedness proof. While seemingly heavyweight in the use of pairs, this approach has the key advantage that it can technically be represented as a theory morphism, thus keeping the framework simple.

Of course, the syntax of $\lambda\Pi/\mathcal{R}$ does not feature dependent pairs. We could extend $\lambda\Pi/\mathcal{R}$, but that would cut us off from implementations, like Dedukti, that do not have dependent pairs. Alternatively, we could construct another kind of translation that eliminates dependent pairs, but that would again complicate the framework. However, it turns out we only need very specific instances of dependent pairs for the translation that we can encode in `SFOL`.

**Extending the Target Language** For our specific translation we only need the types $\Sigma x : tm.\ x \# a$ for every $a : Set$. The declarations below extend `SFOL` to `SFOL′` by adding this type, called $\mathsf{pair}\ a$.

$\mathsf{pair} : Set \to \mathsf{Type}$
$\mathsf{mk\_pair} : \Pi a : Set.\ \Pi x : tm.\ x \# a \to \mathsf{pair}\ a$
$\mathsf{fst} : \Pi a : Set.\ \Pi m : \mathsf{pair}\ a.\ tm$
$\mathsf{snd} : \Pi a : Set.\ \Pi m : \mathsf{pair}\ a.\ (\mathsf{fst}\ a\ m) \# a$

We also specify rewrite rules, effectively making $\mathsf{pair}\ a$ behave like $\Sigma x : tm.\ x \# a$.

$\mathsf{fst}\ a\ (\mathsf{mk\_pair}\ a\ x\ h) \hookrightarrow x$
$\mathsf{snd}\ a\ (\mathsf{mk\_pair}\ a\ x\ h) \hookrightarrow h$
$\mathsf{mk\_pair}\ a\ (\mathsf{fst}\ a\ m)\ (\mathsf{snd}\ a\ m) \hookrightarrow m$

In particular, the second rewrite rule only preserves typing because of the first rewrite rule: the left-hand side is of type $(\mathsf{fst}\ a\ (\mathsf{mk\_pair}\ a\ x\ h)) \# a$ while the right-hand side is of type $x \# a$.

**Remark 7.** *The combination of these rewrite rules with $\beta$-reduction is not confluent on untyped terms [Klo80], but is confluent in several typed cases [Pot81, CC96]. We conjecture that it is also confluent on typed terms of $\lambda\Pi/\mathcal{R}$.*

Note that `SFOL′` is a conservative extension of `SFOL`, in the sense that there is no `SFOL`-type that is uninhabited over `SFOL` but inhabited over `SFOL′`. Thus, `SFOL′` cannot prove any `SFOL`-proposition that `SFOL` cannot prove. In fact, we could even define pair if we worked in an $\lambda\Pi/\mathcal{R}$-like framework with dependent pairs.

**Defining the Translation** We can now give a morphism $\mathtt{hs} : \mathtt{HFOL} \to \mathtt{SFOL'}$.

$\mu(Set) = Set$
$\mu(El) = \lambda a : Set.\ \mathsf{pair}\ a$
$\mu(Prop) = Prop$
$\mu(Prf) = Prf$

Mapping the implication is straightforward, and the condition on the rewrite rule of $\Rightarrow$ is trivially satisfied. To define $\mu(\forall)$, we have a predicate $p$ that takes a pair as an argument, but we need to use the universal quantifier of soft-sorted logic, in which the predicate takes two arguments sequentially. The translations for $\mathsf{all}_\mathsf{i}$ and $\mathsf{all}_\mathsf{e}$ can be

easily derived, as it suffices to pack elements into a pair or unpack them.

$$\begin{array}{lcl}
\mu(\forall) & = & \lambda a : \mathit{Set}.\ \lambda p : \mathsf{pair}\ a \to \mathit{Prop}.\ \forall\ a\ (\lambda x.\ \lambda h.\ p\ (\mathsf{mk_pair}\ a\ x\ h)) \\
\mu(\mathsf{all_i}) & = & \lambda a : \mathit{Set}.\ \lambda p : \mathsf{pair}\ a \to \mathit{Prop}. \\
 & & \lambda H : (\Pi m : \mathsf{pair}\ a.\ \mathit{Prf}\ (p\ m)). \\
 & & \lambda x : \mathit{tm}.\ \lambda h : x \# a. \\
 & & H\ (\mathsf{mk_pair}\ a\ x\ h) \\
\mu(\mathsf{all_e}) & = & \lambda a : \mathit{Set}.\ \lambda p : \mathsf{pair}\ a \to \mathit{Prop}. \\
 & & \lambda H : \mathit{Prf}\ (\forall\ a\ (\lambda x.\ \lambda h.\ p\ (\mathsf{mk_pair}\ a\ x\ h))) \\
 & & \lambda m : \mathsf{pair}\ a. \\
 & & H\ (\mathsf{fst}\ a\ m)\ (\mathsf{snd}\ a\ m)
\end{array}$$

Note that we built a term of type $\mathit{Prf}\ (p\ (\mathsf{mk\_pair}\ a\ (\mathsf{fst}\ a\ m)\ (\mathsf{snd}\ a\ m)))$, although $\mu(\mathsf{all_e})$ must return an object of type $\mathit{Prf}\ (p\ m)$. These two types are convertible thanks to the third rewrite rule.

Without rewriting, we would have to represent the rewrite rules of dependent pairs as equality axioms in the object-logic. But these rewrite rules are used extensively in practice: because our translation pairs every term with the proof of its invariant, every operation on terms must unpair its arguments, operate on the components, and pair the results, thus introducing reducible expressions all over the place. For example, [IR11] used a similar pairing approach to translate type theory into set theory. But it used LF, and practical applications of that translation were bogged down by the resulting overhead of object-logic deduction steps to reduce these expressions. With rewriting, we can minimize this overhead.

## 5.3 From Soft-Sorted Logic to Unsorted Logic

**Overview** To translate from `SFOL` to `UFOL`, the key intuition is to map every *sort* to a unary *predicate* on unsorted terms, and to use that predicate to relativize the quantifiers. Then the external sorting relation $t \# a$ can be mapped to the proposition $\mu(a)\ \mu(t)$.

The critical part of the translation is the treatment of bound variables in the universal quantifier, and this is an open problem for machine-verified language translations. The intuitive idea is to translate a quantification $\forall\ a\ p$ over some sort $a$ with body $p$ to a relativized unsorted quantifier of the form $\forall\ (\lambda x.\ \mu(a)\ x \Rightarrow \mu(p)\ x)$. But in `SFOL`, the body $p$ takes two arguments—a term $x$ and a proof of $x \# a$. The translation of $p$ therefore takes two arguments—a term $x$ and a proof of $\mu(a)\ x$. The straightforward relativization fails here: the application of $\mu(p)$ is ill-typed.

**Remark 8.** *We have identified this as a very general issue. Let us call an object-language expression* proof-carrying *when it has a proof as a sub-expression. Many languages use a stratified design where the syntax does not depend on the proof calculus and therefore proof-carrying expressions cannot arise. If the source language allows proof-carrying expressions while the target language does not, it is impossible for the syntax translation to ever make use of any proofs carried by expressions.*

We could try to erase the assumption $x \# a$, e.g., by translating it to a trivial type. But then we would run into the same problems as discussed above for `HFOL` $\to$ `SFOL`: we need these assumptions when translating the proof rules. After much trial and error, we have identified one way to realize the translation at minimal cost: it again involves a subtle extension of the target language.

**Extending the Target Language** We extend `UFOL` to `UFOL′` by adding *dependent* implication [IR11] where the construction of the second argument may assume the truth of the first. Normally, dependent implication is

useless in UFOL because it is not possible to construct proof-carrying terms anyway. But it makes a difference when translating SFOL-terms that already carry sort assumptions.

UFOL′ replaces the declarations regarding implication by an encoding of dependent implication [BDG+23].

$\Rightarrow' : \Pi p : Prop.\ (Prf\ p \to Prop) \to Prop$
$Prf\ (p \Rightarrow' q) \hookrightarrow \Pi h : Prf\ p.\ Prf\ (q\ h)$

Dependent implication is the Curry-Howard analogue of dependent functions, just like plain implication is the analogue of simple functions. It is easy to give a theory morphism from UFOL to UFOL′ that shows that the usual implication is a special case of the dependent one.

**Defining the Translation** We now define a theory morphism su : SFOL → UFOL′. The translation of most constants is straightforward:

$\mu(tm) = tm$
$\mu(Set) = tm \to Prop$
$\mu(Prop) = Prop$
$\mu(Prf) = Prf$
$\mu(\#) = \lambda x : tm.\ \lambda a : tm \to Prop.\ Prf\ (a\ x)$
$\mu(\Rightarrow) = \Rightarrow$

The condition on the rewrite rule of ⇒ is trivially satisfied.

The translation of the universal quantifier is the key case. It succeeds now because we can assume the relativizing predicate to hold when building the body:

$\mu(\forall) = \lambda a : tm \to Prop.\ \lambda p : (\Pi x : tm.\ Prf\ (a\ x) \to Prop).\ \forall\ (\lambda x : tm.\ (a\ x) \Rightarrow' (\lambda h.\ p\ x\ h))$

The condition on the rewrite rule of ∀ is satisfied in UFOL′.

$$\begin{array}{rcl} \mu(Prf\ (\forall\ a\ p)) & \equiv_{\beta\mathcal{R}} & Prf\ (\forall\ (\lambda x : tm.\ (a\ x) \Rightarrow' (\lambda h.\ p\ x\ h))) \\ & \equiv_{\beta\mathcal{R}} & \Pi x : tm.\ Prf\ ((a\ x) \Rightarrow' (\lambda h.\ p\ x\ h)) \\ & \equiv_{\beta\mathcal{R}} & \Pi x : tm.\ \Pi h : Prf\ (a\ x).\ Prf\ (p\ x\ h) \\ & \equiv_{\beta\mathcal{R}} & \mu(\Pi x : tm.\ \Pi h : x\ \#\ a.\ Prf\ (p\ x\ h)) \end{array}$$

**Remark 9.** *If we extend* SFOL *with a function sort constructor* $\rightsquigarrow : Set \to Set \to Set$ *and with term constructors for* $\lambda$*-abstraction and application, then the morphism can only be completed if* UFOL *is also extended with appropriate unsorted* $\lambda$*-abstraction and application. Critically, this unsorted* $\lambda$*-abstraction must have type* $lam : \Pi a : tm \to Prop.\ (\Pi x : tm.\ Prf\ (a\ x) \to tm) \to tm$*, where the first argument defines the domain of application, and where the bound variables are guaranteed to be from that domain. The morphism fails with variants of* UFOL $\lambda$*-abstraction that do not take into account the well-sortedness of* $x$*.*

## 5.4 Axioms vs. Rewrites

Along the lines of Section 3.3.1, we can give both a computational and a deductive variant for HFOL, SFOL, and UFOL. And in each case, we can obtain a morphism $\mu_{\mathbb{T}}$ from the deductive to the computational variant of $\mathbb{T}$.

Typically, if the source theory is deductive, it does not matter if the target theory is deductive or computational—the translations can be established in essentially the same way. In particular, if the target theory $\mathbb{T}$ is deductive,

we can simply compose the translation with $\mu_{\mathbb{T}}$ to obtain a translation into the computational variant of $\mathbb{T}$. But giving the translation into the computational variant directly can be much simpler because the rewriting in the target theory makes the necessary proofs a lot easier.

It would be unreasonable to expect a translation from a computational to a deductive variant, e.g., from computational `SFOL` to deductive `UFOL`: if the source theory already identifies certain types, they cannot be distinguished in the translation.

But we were surprised to find out that translation between computational variants can also pose subtle issues. Indeed, we failed to give the translation from computational `HFOL` to computational `SFOL`, and that is why we used deductive `HFOL` above. To understand what goes wrong, consider the computational variant of the universal quantifier using the rewrite rule

$$\mathit{Prf}\ (\forall\ a\ p) \hookrightarrow \Pi x : \mathit{El}\ a.\ \mathit{Prf}\ (p\ x)$$

After applying `hs`, the translations of the left-hand side and the right-hand side are not convertible in $\texttt{SFOL}'$ and therefore the theory morphism is not well-formed:

$$\begin{array}{lll} \mu(\mathit{Prf}\ (\forall\ a\ p)) & \equiv_{\beta\mathcal{R}} & \Pi x : \mathit{tm}.\ \Pi h : x \#\ \mu(a).\ \mathit{Prf}\ (\mu(p)\ (\mathsf{mk\_pair}\ \mu(a)\ x\ h)) \\ \mu(\Pi x : \mathit{El}\ a.\ \mathit{Prf}\ (p\ x)) & \equiv_{\beta\mathcal{R}} & \Pi x : \mathsf{pair}\ a.\ \mathit{Prf}\ (\mu(p)\ x) \end{array}$$

The problem here is, essentially, that the former is uncurried while the latter is curried. To our knowledge, there is no rewrite-based system that would allow making these two types convertible. Even in a hypothetical variant of $\lambda\Pi/\mathcal{R}$ with $\Sigma$-types, it is not obvious how such an identification up-to currying could be achieved efficiently.

It is of course possible to add $\Sigma$-types to $\lambda\Pi/\mathcal{R}$, define the translation, and then systematically eliminate the $\Sigma$-types. But the result can become very unwieldy because the elimination must split all functions that return $\Sigma$-types into two. To fill this technical gap, we could consider an extension of $\lambda\Pi/\mathcal{R}$ with $\Sigma$-types available, but extend the conversion relation in such a way that they are rewritten into plain $\lambda\Pi/\mathcal{R}$.

# 6 Embedding Data Types

Our use of dependent implication and dependent pairs in the previous translations is not a one-off trick. They are a general techniques for realizing subtly difficult translations. One common application is embedding theorems that identify a smaller data type as a fragment of a larger one. Here the difficulty is that a binding over the smaller source data type must be relativized when translating it to a binding of the larger target data type. As an example, we give an embedding translation from natural numbers to integers, in which we need to relativize bindings of integer variables with the property of being non-negative.

This property serves as an invariant the translated terms must satisfy. Similar to the sort-erasure translations, it is critical to prove this invariant by induction over terms, and this is difficult because the translation must be intertwined with the proof of the invariant.

We start by encoding both theories as sorted languages, i.e., as extensions of `HFOL`.

**Example 16** (Natural Numbers)**.** *The theory* $\texttt{HFOL} + \texttt{Nat}$ *of natural numbers is built on hard-sorted logic. But for the sake of example, we encode the proofs of universally quantified propositions computationally, i.e., via a rewrite rule:*

$\forall : \Pi a : \mathit{Set}.\ (\mathit{El}\ a \to \mathit{Prop}) \to \mathit{Prop}$
$\mathit{Prf}\ (\forall\ a\ p) \hookrightarrow \Pi x : \mathit{El}\ a.\ \mathit{Prf}\ (p\ x)$

*We define the sort* nat *for natural numbers, with the two constructors* $0$ *and* succ*. The relation* $\geq$ *is reflexive and transitive.*

$\mathsf{nat} : \mathit{Set}$

$0 : \mathit{El}\ \mathsf{nat}$

$\mathsf{succ} : \mathit{El}\ \mathsf{nat} \to \mathit{El}\ \mathsf{nat}$

$\geq : \mathit{El}\ \mathsf{nat} \to \mathit{El}\ \mathsf{nat} \to \mathit{Prop}$

$\mathsf{ax}_1 : \Pi x : \mathit{El}\ \mathsf{nat}.\ \mathit{Prf}\ (x \geq x)$

$\mathsf{ax}_2 : \Pi x, y, z : \mathit{El}\ \mathsf{nat}.\ \mathit{Prf}\ (x \geq y) \to \mathit{Prf}\ (y \geq z) \to \mathit{Prf}\ (x \geq z)$

*For any natural number* $x$*,* succ $x$ *is greater than* $x$*. In other words, we have a proof of* succ $x \geq x$*, and any proof of* $x \geq$ succ $x$ *leads to an inconsistency.*

$\mathsf{ax}_3 : \Pi x : \mathit{El}\ \mathsf{nat}.\ \mathit{Prf}\ (\mathsf{succ}\ x \geq x)$

$\mathsf{ax}_4 : \Pi x : \mathit{El}\ \mathsf{nat}.\ \mathit{Prf}\ (x \geq \mathsf{succ}\ x) \to \Pi P : \mathit{Prop}.\ \mathit{Prf}\ P$

*Finally, we have the induction principle on natural numbers:*

$$\begin{aligned}\mathsf{rec} : \quad & \Pi P : \mathit{El}\ \mathsf{nat} \to \mathit{Prop}.\ \mathit{Prf}\ (P\ 0) \to \\ & [\Pi x : \mathit{El}\ \mathsf{nat}.\ \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ (\mathsf{succ}\ x))] \to \\ & \Pi x : \mathit{El}\ \mathsf{nat}.\ \mathit{Prf}\ (P\ x)\end{aligned}$$

**Example 17** (Integers)**.** *The theory of integers* $\mathtt{HFOL} + \mathtt{Int}$ *is like* $\mathtt{HFOL} + \mathtt{Nat}$*, with the sort* nat *renamed to* int*. Additionally, we introduce a predecessor symbol* pred*, such that* pred *and* succ *are inverses.*

$\mathsf{pred} : \mathit{El}\ \mathsf{int} \to \mathit{El}\ \mathsf{int}$

$\mathsf{succ}\ (\mathsf{pred}\ x) \hookrightarrow x$

$\mathsf{pred}\ (\mathsf{succ}\ x) \hookrightarrow x$

*The normal terms of type* $\mathit{El}$ int *are in bijection to the integers.*

*For any integer* $x$*,* pred $x$ *is lower than* $x$*.*

$\mathsf{ax}_5 : \Pi x : \mathit{El}\ \mathsf{int}.\ \mathit{Prf}\ (x \geq \mathsf{pred}\ x)$

$\mathsf{ax}_6 : \Pi x : \mathit{El}\ \mathsf{int}.\ \mathit{Prf}\ (\mathsf{pred}\ x \geq x) \to \Pi P : \mathit{Prop}.\ \mathit{Prf}\ P$

*The induction principle on integers*

$$\begin{aligned}\mathsf{rec} : \quad & \Pi P : \mathit{El}\ \mathsf{int} \to \mathit{Prop}.\ \mathit{Prf}\ (P\ 0) \to \\ & [\Pi x : \mathit{El}\ \mathsf{int}.\ \mathit{Prf}\ (x \geq 0) \to \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ (\mathsf{succ}\ x))] \to \\ & [\Pi x : \mathit{El}\ \mathsf{int}.\ \mathit{Prf}\ (0 \geq x) \to \mathit{Prf}\ (P\ x) \to \mathit{Prf}\ (P\ (\mathsf{pred}\ x))] \to \\ & \Pi x : \mathit{El}\ \mathsf{int}.\ \mathit{Prf}\ (P\ x)\end{aligned}$$

*involves an additional induction step for* pred*.*

The translation $\mathtt{HFOL} + \mathtt{Nat} \to \mathtt{HFOL} + \mathtt{Int}$ is structurally very similar to the one $\mathtt{HFOL} \to \mathtt{SFOL}$. We intuitively map the sort nat to the sort int, but we need to recover the information that any natural number is mapped to a non-negative integer. Here the invariant of the translation is the predicate $x \geq 0$. Like the translation $\mathtt{HFOL} \to \mathtt{SFOL}$,

we will employ dependent pairs. The only dependent pairs we need are of the form $\Sigma x : \mathit{El}\ a.\ \mathit{Prf}\ (p\ x)$. The definitions below conservatively extend $\texttt{HFOL} + \texttt{Int}$ to $\texttt{HFOL} + \texttt{Int}'$ with an axiomatization of those dependent pairs.

$\mathsf{pair} : \Pi a : \mathit{Set}.\ (\mathit{El}\ a \to \mathit{Prop}) \to \mathit{Set}$
$\mathsf{mk\_pair} : \Pi a : \mathit{Set}.\ \Pi p : \mathit{El}\ a \to \mathit{Prop}.\ \Pi x : \mathit{El}\ a.\ \mathit{Prf}\ (p\ x) \to \mathit{El}\ (\mathsf{pair}\ a\ p)$
$\mathsf{fst} : \Pi a : \mathit{Set}.\ \Pi p : \mathit{El}\ a \to \mathit{Prop}.\ \mathit{El}\ (\mathsf{pair}\ a\ p) \to \mathit{El}\ a$
$\mathsf{snd} : \Pi a : \mathit{Set}.\ \Pi p : \mathit{El}\ a \to \mathit{Prop}.\ \Pi m : \mathit{El}\ (\mathsf{pair}\ a\ p).\ \mathit{Prf}\ (p\ (\mathsf{fst}\ a\ p\ m))$
$\mathsf{fst}\ a\ p\ (\mathsf{mk\_pair}\ a\ p\ x\ h) \hookrightarrow x$
$\mathsf{snd}\ a\ p\ (\mathsf{mk\_pair}\ a\ p\ x\ h) \hookrightarrow h$
$\mathsf{mk\_pair}\ a\ p\ (\mathsf{fst}\ a\ p\ m)\ (\mathsf{snd}\ a\ p\ m) \hookrightarrow m$

Then we can define $\texttt{NI} : \texttt{HFOL} + \texttt{Nat} \to \texttt{HFOL} + \texttt{Int}'$. The constants of hard-sorted logic are mapped to themselves. The sort of natural numbers is mapped to the sort that pairs an integer and a proof that it is non-negative.

$\mu(\mathsf{nat}) = \mathsf{pair}\ \mathsf{int}\ (\lambda x.\ x \geq 0)$
$\mu(0) = \mathsf{mk\_pair}\ \mathsf{int}\ (\lambda x.\ x \geq 0)\ 0\ (\mathsf{ax}_1\ 0)$
$\mu(\geq) = \lambda m_1, m_2 : \mathsf{pair}\ \mathsf{int}\ (\lambda x.\ x \geq 0).\ (\mathsf{fst}\ \mathsf{int}\ (\lambda x.\ x \geq 0)\ m_1) \geq (\mathsf{fst}\ \mathsf{int}\ (\lambda x.\ x \geq 0)\ m_2)$
$\mu(\mathsf{ax}_1) = \lambda m : \mathsf{pair}\ \mathsf{int}\ (\lambda x.\ x \geq 0).\ \mathsf{ax}_1\ (\mathsf{fst}\ \mathsf{int}\ (\lambda x.\ x \geq 0)\ m)$

Most of the remaining parameters are defined similarly. The parameter $\mu(\mathsf{rec})$ is trickier: it requires a dependent implication, for the same reason as the translation $\texttt{HFOL} \to \texttt{SFOL}$. It also requires proof irrelevance—the principle stating that two proofs of the same proposition are equal. We add dependent implication and an axiom for proof irrelevance to $\texttt{HFOL} + \texttt{Int}'$.

$\Rightarrow' : \Pi p : \mathit{Prop}.\ (\mathit{Prf}\ p \to \mathit{Prop}) \to \mathit{Prop}$
$\mathit{Prf}\ (p \Rightarrow' q) \hookrightarrow \Pi h : \mathit{Prf}\ p.\ \mathit{Prf}\ (q\ h)$
$\mathsf{proof\_irr} : \Pi p : \mathit{Prop}.\ \Pi h_1, h_2 : \mathit{Prf}\ p.\ \Pi q : \mathit{Prf}\ p \to \mathit{Prop}.\ \mathit{Prf}\ (q\ h_1) \to \mathit{Prf}\ (q\ h_2)$

The translation from natural numbers to integers was the running example of [Tra24b], where it was formalized using an intricate construction involving mutually recursive definitions of morphism and logical relation. It also required more boilerplate such as trivially true invariants that must be carried through the induction. In contrast, the present translation is itself much simpler and can be expressed in a simpler framework.

Our embedding shows that natural numbers are mapped to non-negative integers. We can extend the theories $\texttt{Nat}$ and $\texttt{Int}$ with operations such as addition, and extend the morphism accordingly. Then the morphism additionally shows that the corresponding operations on integers preserve the invariant.

Moreover, we can use a logical relation to show that the denotations of natural numbers are preserved by the embedding. First, assume we have a theory $\texttt{Den}$ in which denotations of numbers are defined. The semantics of numbers can be represented by morphisms $\texttt{natden} : \texttt{Nat} \to \texttt{Den}$ and $\texttt{intden} : \texttt{Int} \to \texttt{Den}$. To show the preservation of denotations, we can then give a binary logical relation on the morphisms $\texttt{natden}$ and $\texttt{NI}; \texttt{intden}$, that maps the type $\mathsf{nat}$ to the statement of equality of the two denotations.

# 7 Implementation for Dedukti

**Implementation** Dedukti[1] is a proof language based on $\lambda\Pi/\mathcal{R}$. We developed a tool, called `TranslationTemplates`[2], that implements the new features developed here.

Because the main applications of Dedukti are the batch processing of large sets of theorems, our design makes the same trade-offs and optimizes for the batch transport of theorems from a source theory $\mathbb{S}$ to a source theory $\mathbb{T}$. `TranslationTemplates` takes two files representing $\mathbb{S}$ and $\mathbb{T}$, and outputs a new file that contains a copy of $\mathbb{S}$ as an extension of $\mathbb{T}$. All primitive declarations of $\mathbb{S}$ result in gaps that the user needs to fill in—these are the parameters of a theory morphism/logical relation. Then all defined declarations of $\mathbb{S}$ can simply be copied over. An additional argument controls if the generated file should capture a theory morphism or a logical relation. The resulting file can be rechecked by Dedukti so that our code does not have to be a part of the trusted code base. The tool is written in OCaml in less than 400 lines of code and benefits from the Dedukti kernel and parser. All the examples of theory morphisms given here have been implemented in Dedukti and mechanically checked. As an additional example, we have implemented a theory morphism from higher-order classical logic to higher-order intuitionistic logic, following [Tra24a].

**Demonstration** We illustrate `TranslationTemplates` on the theory morphism from Section 3.3.1. For simplicity, we only consider the implication symbol. The source file `deduction.dk` contains the theory where natural deduction rules are encoded via axioms.

```
Prop : Type.
Prf : Prop -> Type.
imp : Prop -> Prop -> Prop.
imp_i : p : Prop -> q : Prop -> (Prf p -> Prf q) -> Prf (imp p q).
imp_e : p : Prop -> q : Prop -> Prf (imp p q) -> Prf p -> Prf q.
```

We prove the theorem $\Pi p : \mathit{Prop}.\ \mathit{Prf}\ (p \Rightarrow p)$ by taking the proof term $\lambda p : \mathit{Prop}.\ \mathsf{imp_i}\ p\ p\ (\lambda H : \mathit{Prf}\ p.\ H)$.

```
thm lemma_imp : p : Prop -> Prf (imp p p)
  := p => imp_i p p (H => H).
```

The target file `computation.dk` contains the theory where natural deduction rules are encoded via rewrite rules. Note that the symbol `Prf` is now declared with `def`, because it is definable with rewrite rules.

```
Prop : Type.
def Prf : Prop -> Type.
imp : Prop -> Prop -> Prop.
[p, q] Prf (imp p q) --> Prf p -> Prf q.
```

The theory morphism from `deduction.dk` to `computation.dk` generates the following file.

```
#REQUIRE computation.

def Prop_mu : Type := TODO.
def Prf_mu : Prop_mu -> Type := TODO.
def imp_mu : Prop_mu -> Prop_mu -> Prop_mu := TODO.
def imp_i_mu : p : Prop_mu -> q : Prop_mu ->
                (Prf_mu p -> Prf_mu q) -> Prf_mu (imp_mu p q)
```

[1] Available at https://github.com/Deducteam/Dedukti.
[2] Available at https://github.com/Deducteam/TranslationTemplates.

```
  := TODO.
def imp_e_mu : p : Prop_mu -> q : Prop_mu ->
                Prf_mu (imp_mu p q) -> Prf_mu p -> Prf_mu q
  := TODO.

thm lemma_imp_mu : p : Prop_mu -> Prf_mu (imp_mu p p)
  := p => imp_i_mu p p (H => H).
```

This is a skeleton of the translation, where the parameters must be filled in by the user instead of the `TODO`s. To help the user, the type of each parameter is displayed. Such parameters must be expressed in the target theory `computation.dk`. We can do so following the blueprint of Section 3.3.1. The statement and the proof of the theorem `lemma_imp` of the source theory have been *automatically* translated to the target theory. The file typechecks provided that the conditions of the theory morphism are fulfilled.

# 8 Conclusion

**Summary** We have introduced two translation templates—based on theory morphisms and logical relations—for representing meta-theorems for the $\lambda\Pi$-calculus modulo rewriting. Barring an example of a theory morphism in [Fel22], this is the first time that such translation templates are used systematically for a logical framework with rewriting.

Throughout our examples, we have dealt with the subtle trade-off between axioms and rewrite rules. On the one hand, when using rewrite rules instead of axioms, parts of the proofs are directly discharged by the framework through computation. In particular, the presence of rewrite rules in the target theory simplifies the proof obligations generated by the translation templates. On the other hand, the presence of rewrite rules in the source theory creates additional constraints that need to be satisfied by the translation templates. These observations allow us to fully leverage the computational power of the $\lambda\Pi$-calculus modulo rewriting to formalize translations between theories.

Moreover, we have identified two subtle practices that allow representing meta-theorems that have previously proved challenging: the use of dependent pairs (as a primitive of the framework or as an ad-hoc conservative extension) and of dependent implication. This observation is independent of rewriting and applies to other logical frameworks as well. More generally, it indicates that efficient translations across languages may be critically enabled by deep technical tweaks to the framework or the target language.

We have implemented both templates in the Dedukti proof language, and we have applied them to mechanically check translations between a number of logics. These kinds of templates are critical for the interoperability of proof systems, a major goal of the Dedukti project, as they allow for the batch translation of large libraries along a theory morphism or a logical relation.

**Reflection Properties** Both morphisms and logical relations capture preservation-style meta-theorems ("if it holds in the source, it is preserved in the target"). These correspond to soundness proofs if we think of the source as the syntax and the morphism as the interpretation function that maps the syntax to its semantics. Such proofs typically proceed by induction on derivations, and that recipe is exactly what morphisms and relations capture.

The dual, reflection-style meta-theorems ("if it holds in the target, it already held in the source") are typically much harder to prove and would therefore be more interesting to verify in logical frameworks. Such theorems correspond to completeness theorems. They are called conservativity of morphisms in [Rab24].

Morphisms can only express that the target theory is strong enough to express the source theory. Sometimes a pair of isomorphisms can be used to express preservation and reflection and thus show that two theories can define

each other. But in most cases a morphism showing preservation is not an isomorphism even if some reflection property holds.

This is because the reflection property often holds only in some restricted form and not for all expressions. For example, we can give a morphism $\mu$ from `HFOL` to set theory that captures the model-theoretical semantics of the logic as well as its soundness proof. The completeness of the semantics is a restricted reflection property: $\mu$ reflects provability in the sense that if $\mu(\mathit{Prf}\ p)$ is inhabited then so is $\mathit{Prf}\ p$. But there is no inverse morphism $\mu^{-1}$ that maps all of set theory back to `HFOL`.

One way to formalize such restricted reflection theorems is to give a partial translation from the target theory to the source theory. For example, [Pfe00] gives and verifies such a translation to reflect derivability from a calculus with cut to one without. If both the source and the target theory have a sound and complete semantics, an alternative is to relate those semantics to each other, as illustrated in [RS13]. For both kinds of arguments, it is difficult to provide general formalization support at the logical framework level. We see this as a major challenge problem for logical frameworks.

**Acknowledgments.** This publication is based upon work from the action CA20111 EuroProofNet supported by COST (European Cooperation in Science and Technology).

We thank the reviewers for their relevant comments and suggestions.

# References

[ABC+16] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory. Manuscript, 2016.

[AHMP98] Arnon Avron, Furio Honsell, Marino Miculan, and Cristian Paravano. Encoding modal logics in logical frameworks. *Studia Logica: An International Journal for Symbolic Logic*, 60(1):161–208, 1998.

[And02] Peter B. Andrews. *Type Theory*, pages 201–256. Springer Netherlands, Dordrecht, 2002.

[BDG+23] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. A modular construction of type theories. *Logical Methods in Computer Science*, Volume 19, Issue 1, February 2023.

[BJP10] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *ICFP 2010 - 15th ACM SIGPLAN International Conference on Functional Programming*, page 345–356, Baltimore, USA, 2010. Association for Computing Machinery.

[BJP12] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22(2):107–152, 2012.

[Bla01] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 9–18, 2001.

[Bla16] Frédéric Blanqui. Termination of rewrite relations on λ-terms based on girard's notion of reducibility. *Theoretical Computer Science*, 611:50–86, 2016.

[Bla24] Frédéric Blanqui. Translating HOL-Light proofs to Coq. In *LPAR 2024 - 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 1–18, Balaclava, Mauritius, 2024.

[CC96] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction for the $\lambda$-calculus with surjective pairing and terminal object. *Journal of Functional Programming*, 6(2):299–327, 1996.

[CCM24] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof Transfer for Free, With or Without Univalence. In *ESOP 2024 - 33rd European Symposium on Programming*, pages 239–268, Luxembourg, Luxembourg, 2024. Springer Nature Switzerland.

[CD07] Denis Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CELM96] Manuel Clavel, Steven Eker, Patrick D. Lincoln, and José Meseguer. Principles of Maude. volume 4, pages 65–89, 1996. RWLW96, First International Workshop on Rewriting Logic and its Applications.

[CHK+11] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, and Florian Rabe. Project abstract: Logic atlas and integrator (latin). In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics*, pages 289–291, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[dB80] Nicolaas G. de Bruijn. *A survey of the project Automath*, pages 579–606. Academic Press Inc., United States, 1980.

[Fel22] Thiago Felicissimo. Adequate and Computational Encodings in the Logical Framework Dedukti. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[HLMS17] Furio Honsell, Luigi Liquori, Petar Maksimovic, and Ivan Scagnetto. LLFP: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads. *Logical Methods in Computer Science*, 2017.

[HR11] Fulya Horozal and Florian Rabe. Representing model theory in a type-theoretical logical framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011. Logical and Semantic Frameworks with Applications (LSFA 2008 and 2009).

[HST94] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structured theory presentations and logic representations. *Annals of Pure and Applied Logic*, 67(1):113–160, 1994.

[IR11] Mihnea Iancu and Florian Rabe. Formalising foundations of mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.

[KL12] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In *CSL 2012 - 26th EACSL Annual Conference on Computer Science Logic*, volume 16, pages 381–395, Fontainebleau, France, 2012. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*. Mathematisch centrum, Amsterdam, 1980.

[KWP99] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A Sectioning Concept for Isabelle. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 149–165, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1994.

[PD10] Brigitte Pientka and Jana Dunfield. Beluga: a framework for programming and reasoning with deductive systems (system description). In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, page 15–21, Berlin, Heidelberg, 2010. Springer-Verlag.

[Pfe00] Frank Pfenning. Structural Cut Elimination: I. Intuitionistic and Classical Logic. *Information and Computation*, 157(1):84–141, 2000.

[Pot81] Garrel Pottinger. The Church-Rosser theorem for the typed $\lambda$-calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264 – 268, 1981.

[PS99] Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Automated Deduction — CADE-16*, pages 202–206, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[PS09] Adam Poswolsky and Carsten Schürmann. System Description: Delphin – A Functional Programming Language for Deductive Systems. *Electronic Notes in Theoretical Computer Science*, 228:113–120, 2009. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

[Rab14] Florian Rabe. How to identify, translate and combine logics? *Journal of Logic and Computation*, 27(6):1753–1798, 12 2014.

[Rab18] Florian Rabe. A modular type reconstruction algorithm. *ACM Trans. Comput. Logic*, 19(4), December 2018.

[Rab24] Florian Rabe. A Logical Framework Perspective on Conservativity. In A. Kohlhase and L. Kovacs, editors, *Intelligent Computer Mathematics*, pages 203–219. Springer, 2024.

[RK13] Florian Rabe and Michael Kohlhase. A scalable module system. *Information and Computation*, 230:1–54, 2013.

[RS09] Florian Rabe and Carsten Schürmann. A practical module system for LF. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, page 40–48, New York, NY, USA, 2009. Association for Computing Machinery.

[RS13] Florian Rabe and Kristina Sojakova. Logical relations for a logical framework. *ACM Transactions on Computational Logic*, 14(4), November 2013.

[Sai15] Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, September 2015.

[SS06] Carsten Schürmann and Mark-Oliver Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 150–166, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[SW83] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation extended abstract. In Marek Karpinski, editor, *Foundations of Computation Theory*, pages 413–427, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[Thi20] François Thiré. *Interoperability between proof systems using the logical framework Dedukti.* Theses, Université Paris-Saclay, December 2020.

[Tra24a] Thomas Traversié. Kuroda's Translation for the λΠ-Calculus Modulo Theory and Dedukti. In Florian Rabe and Claudio Sacerdoti Coen, editors, Proceedings Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice,* Tallinn, Estonia, 8th July 2024, volume 404 of *Electronic Proceedings in Theoretical Computer Science*, pages 35–48. Open Publishing Association, 2024.

[Tra24b] Thomas Traversié. Proofs for Free in the λΠ-Calculus Modulo Theory. In Florian Rabe and Claudio Sacerdoti Coen, editors, Proceedings Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice,* Tallinn, Estonia, 8th July 2024, volume 404 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–63. Open Publishing Association, 2024.