Adequate and computational encodings in the logical framework Dedukti

Thiago Felicissimo ✉
Université Paris-Saclay, INRIA project Deducteam, Laboratoire de Méthodes Formelles, ENS Paris-Saclay, 91190 France

Abstract

DEDUKTI is a very expressive logical framework which unlike most frameworks, such as the Edinburgh Logical Framework (ELF), allows for the representation of computation alongside deduction. However, unlike ELF encodings, DEDUKTI encodings proposed until now do not feature an adequacy theorem — *i.e.*, a bijection between terms in the encoded system and in its encoding. Moreover, many of them also do not have a conservativity one, which compromises the ability of DEDUKTI to check proofs written in such encodings. We propose a different approach for DEDUKTI encodings which do not only allow for simpler conservativity proofs, but which also restore the adequacy of encodings. More precisely, we propose in this work adequate (and thus conservative) encodings for Functional Pure Type Systems. However, in contrast with ELF encodings, ours is computational — that is, represents computation directly as computation. Therefore, our work is the first to present and prove correct an approach allowing for encodings that are both adequate and computational in DEDUKTI.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Equational logic and rewriting

Keywords and phrases Type Theory, Logical Frameworks, Rewriting, Dedukti

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements I would like to thank my PhD advisors Frédéric Blanqui and Gilles Dowek for the helpful discussions and comments on this paper.

1 Introduction

The research on proof-checking naturally leads to the proposal of many logical systems and theories. *Logical frameworks* are a way of addressing this heterogeneity by proposing a common foundation in which systems and theories can be defined. The *Edinburgh Logical Framework* (ELF)[17] is one of the milestones in the history of logical frameworks, and proposes the use of a dependently-typed lambda-calculus to express deduction. However, as modern proof assistants move from traditional logics to type theories, where computation plays an important role alongside deduction, it becomes essential for such frameworks to also be able to express computation, something that the ELF does not achieve.

The logical framework DEDUKTI[3] addresses this point by extending the ELF with rewriting rules, thus allowing for the representation of both deduction and computation. This framework has already proven itself as a very expressive system, and has been used to encode the logics of many proof assistants, such as COQ[13], AGDA[14], PVS[18] and others.

However, an unsatisfying aspect persists as, unlike ELF encodings, the DEDUKTI encodings proposed until now are not *adequate* — that is, feature a syntactical bijection between the terms of the encoded system and those of the encoding. Such property is key in ensuring that the framework faithfully represents the syntax on the encoded system. Moreover, proving that DEDUKTI encodings are *conservative* (*i.e.*, that if the translation of a type is inhabited, then this type is inhabited) is still a challenge, in particular for recent works such as [13][23][14][18]. This is a problem if one intends to use DEDUKTI to check the correctness of proofs coming from proof assistants: if conservativity does not hold then the fact that the translation of a

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

proof is checked correct in Dedukti does not imply that this proof is correct.

In the specific case of Pure Type Systems (PTS), a class of type systems which generalizes many others, [9] was the first to propose an encoding of functional PTSs into Dedukti. One of their main contributions is that, differently from ELF encodings, their one is *computational* — that is, represents computation in the encoded system directly as computation. They then showed that the encoding was conservative under the hypothesis of normalization of their rewrite rules.

To address the issue of this unproven assumption, [11] proposed a notion of model of Dedukti and showed using the technique of *reducibility candidates* that the existence of such a model entails the normalization of the encoding. Using this result, they then showed the conservativity of the encoding of Simple Type Theory and of the Calculus of Constructions. This technique however is not very satisfying as the construction of such models is a very technical task, and needs to be done case by case. One can also wonder why conservativity should rely on normalization.

The cause of this difficulty in [9] and in all other traditional Dedukti encodings comes from a choice made to represent the abstraction and application of the encoded system directly by the abstraction and application of the framework. This causes a confusion as redexes of the encoded system, that represent real computations, get confused with the β redexes of the framework, which in other frameworks such as the ELF are used exclusively to represent binder substitution. As a non-normal term can contain both types of redexes, it is impossible to inverse translate it as some of these redexes are ill-typed in the original system, and the only way of eliminating these ill-typed redexes is by reducing all ones. One then needs this process to be terminating, which is non-trivial to show as it envolves proving that the reduction of the redexes of the encoded system terminates.

The work of [2] first noted this problem and proposed a different approach to show the conservativity of the encoding of PTSs. Instead of relying on the normalization of the encoding, they proposed to directly inverse translate terms without normalizing them. As this creates ill-typed terms, they then used reducibility candidates to show that these ill-typed terms reduce to well-typed ones, thus proving conservativity for the encoding in [9]. Even though this technique is a big improvement over [11], it is still unsatisfying that both of them rely on involved arguments using reducibility, whereas the proofs of ELF encodings were very natural. The technicality of these proofs may be a reason for recent works such as [23], [18] and [14] to have left conservativity as conjecture. Moreover, none of such works have addressed the lack of an adequacy theorem.

Our contribution

We propose to depart from the approach of traditional Dedukti encodings by restoring the separation that existed in ELF encodings. Our paradigm represents the abstractions and applications of the encoded system not by those of the framework, but by dedicated constructions. Using this approach, we propose an encoding of functional PTSs that is not only sound and conservative but also adequate. However, in contrast with ELF encodings, ours is computational like other Dedukti encodings.

To show conservativity, we leverage the fact the computational rules of the encoded system are not represented by β reduction anymore, but by dedicated rewrite rules. This allows us to normalize only the framework's β redexes without touching the ones associated with the encoded system, and thus performing no computation from its point of view.

To be able to β normalize terms, we generalize the proof in [17] to give a general criterion for the normalization of β reduction in Dedukti. This criterion imposes rewriting rules to be *arity preserving* (a definition we introduce). This is not satisfied by traditional DEDUKTI encodings, but poses no problem to ours. The proof uses the simple technique of defining an erasure map into the simply-typed lambda calculus, which is known to be normalizing.

Outline

We start in Section 2 by recalling the preliminaries about DEDUKTI. We proceed in Section 3 by proposing a criterion for the normalization of β in DEDUKTI, which is used in our proofs of conservativity and adequacy. In Section 4 we introduce an explicitly-typed version of Pure Type Systems, which is used in our translation. We then present our encoding in Section 5, and proceed by showing it is sound in Section 6 and that it is conservative and adequate in Section 7. In Section 8 we discuss how our approach can be used together with already known techniques to represent systems with infinite sorts. Finally, in Section 10 we discuss more practical aspects by showing how our encoding can be instantiated and used in practice.

2 Dedukti

$$\frac{}{\Sigma; - \ \mathtt{well\text{-}formed}}\ \mathtt{Empty} \qquad x \notin \Gamma\ \frac{\Sigma; \Gamma \vdash A : \mathtt{TYPE}}{\Sigma; \Gamma, x : A\ \mathtt{well\text{-}formed}}\ \mathtt{Decl}$$

$$c[\Delta] : A \in \Sigma\ \frac{\Sigma; \Delta \vdash A : s \qquad \Sigma; \Gamma \vdash \vec{M} : \Delta}{\Sigma; \Gamma \vdash c[\vec{M}] : A\{\vec{M}\}}\ \mathtt{Cons} \qquad \frac{\Sigma; \Gamma\ \mathtt{well\text{-}formed}}{\Sigma; \Gamma \vdash \mathtt{TYPE} : \mathtt{KIND}}\ \mathtt{Sort}$$

$$x : A \in \Gamma\ \frac{\Sigma; \Gamma\ \mathtt{well\text{-}formed}}{\Sigma; \Gamma \vdash x : A}\ \mathtt{Var} \qquad A \equiv_{\beta\mathcal{R}} B\ \frac{\Sigma; \Gamma \vdash M : A \qquad \Sigma; \Gamma \vdash B : s}{\Sigma; \Gamma \vdash M : B}\ \mathtt{Conv}$$

$$\frac{\Sigma; \Gamma \vdash A : \mathtt{TYPE} \qquad \Sigma; \Gamma, x : A \vdash B : s}{\Sigma; \Gamma \vdash \Pi x : A.B : s}\ \mathtt{Prod} \qquad \frac{\Sigma; \Gamma \vdash M : \Pi x : A.B \qquad \Sigma; \Gamma \vdash N : A}{\Sigma; \Gamma \vdash MN : B\{N/x\}}\ \mathtt{App}$$

$$\frac{\Sigma; \Gamma \vdash A : \mathtt{TYPE} \qquad \Sigma; \Gamma, x : A \vdash B : s \qquad \Sigma; \Gamma, x : A \vdash M : B}{\Sigma; \Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ \mathtt{Abs}$$

Figure 1 Typing rules for DEDUKTI

The logical framework DEDUKTI [3] has the syntax of the λ-calculus with dependent types [17] ($\lambda\Pi$-calculus). Like works such as [18], we consider here a version with arities, with the following syntax.

$$A, B, M, N ::= x \mid c[\vec{M}] \mid \mathtt{TYPE} \mid \mathtt{KIND} \mid MN \mid \lambda x : A.M \mid \Pi x : A.B$$

Here, c ranges in an infinite set of constants $\mathcal{C}$, and x ranges in an infinite set of variables $\mathcal{V}$. Each constant c is assumed to have a fixed arity n_c and for each occurence of $c[\vec{M}]$ we should have $length(\vec{M}) = n_c$. We denote $\Lambda_{\mathtt{DK}}$ the set of terms generated by this grammar. We call a term of the form $\Pi x : A.B$ a *dependent product*, and we write $A \to B$ when x does not appear free in B. We allow ourselves sometimes to write $c\ \vec{M}$ instead of $c[\vec{M}]$ to ease the notation.

A *context* Γ is a finite sequence of pairs $x : A$ with $A \in \Lambda_{\mathtt{DK}}$. A *signature* Σ is a finite set of triples $c[\Delta] : A$ where $A \in \Lambda_{\mathtt{DK}}$ and Δ is a context containing at least all free variables of A. The main difference between DEDUKTI and the $\lambda\Pi$-calculus is that we also consider a set $\mathcal{R}$ of *rewrite rules*, that is, of pairs of the form $c[\vec{l}] \longrightarrow r$ with $l_1, ..., l_k, r \in \Lambda_{\mathtt{DK}}$. A *theory* is a pair $(\Sigma, \mathcal{R})$ such that all constants appearing in $\mathcal{R}$ are declared in Σ.

We write $\hookrightarrow_{\mathcal{R}}$ for the context and substitution closure of the rules in $\mathcal{R}$ and $\hookrightarrow_{\beta\mathcal{R}}$ for $\hookrightarrow_{\beta} \cup \hookrightarrow_{\mathcal{R}}$. We also consider the equivalence relation $\equiv_{\beta\mathcal{R}}$ generated by $\hookrightarrow_{\beta\mathcal{R}}$. Finally, we may refer to $\hookrightarrow_{\beta\mathcal{R}}$ and $\equiv_{\beta\mathcal{R}}$ by just $\hookrightarrow$ and $\equiv$.

Typing in Dedukti is given by the rules in Figure 1. In rule `Conv` we use to usual notation $\Sigma; \Gamma \vdash \vec{M} : \Delta$ meaning that $\Delta = x_1 : A_1, ..., x_n : A_n$ and $\Sigma; \Gamma \vdash M_i : A_i\{M_1/x_1\}...\{M_{i-1}/x_{i-1}\}$ is derivable for $i = 1, ..., n$. We then also allow ourselves to write $A\{\vec{M}\}$ instead of $A\{M_1/x_1\}...\{M_n/x_n\}$.

We recall the following basic metatheorems.

▶ **Proposition 1** (Basic properties). *Suppose $\hookrightarrow_{\beta\mathcal{R}}$ is confluent.*

1. *Weakening: If $\Sigma; \Gamma \vdash M : A$ and $\Gamma \sqsubseteq \Gamma'$ then $\Sigma; \Gamma' \vdash M : A$*
2. *Well-typedeness of contexts: If $\Sigma; \Gamma$ `well-formed` then for all $x : B \in \Gamma$, $\Sigma; \Gamma \vdash B : \mathtt{TYPE}$*
3. *Inversion of typing: Suppose $\Sigma; \Gamma \vdash M : A$*
   - *If $M = x$ then $x : A' \in \Gamma$ and $A \equiv A'$*
   - *If $M = c[\vec{N}]$ then $c[\Delta] : A' \in \Sigma$, $\Sigma; \Delta \vdash A' : s$, $\Sigma; \Gamma \vdash \vec{N} : \Delta$ and $A'(\vec{N}/\Delta) \equiv A$*
   - *If $M = \mathtt{TYPE}$ then $A \equiv \mathtt{KIND}$*
   - *$M = \mathtt{KIND}$ is impossible*
   - *If $M = \Pi x : A_1.A_2$ then $\Sigma; \Gamma \vdash A_1 : \mathtt{TYPE}$, $\Sigma; \Gamma, x : A_1 \vdash A_2 : s$ and $s \equiv A$*
   - *If $M = M_1 M_2$ then $\Sigma; \Gamma \vdash M_1 : \Pi x : A_1.A_2$, $\Sigma; \Gamma \vdash M_2 : A_1$ and $A_2(M_2/x) \equiv A$*
   - *If $M = \lambda x : B.N$ then $\Sigma; \Gamma \vdash B : \mathtt{TYPE}$, $\Sigma; \Gamma, x : B \vdash C : s$, $\Sigma; \Gamma, x : B \vdash N : C$ and $A \equiv \Pi x : B.C$*
4. *Uniqueness of types: If $\Sigma; \Gamma \vdash M : A$ and $\Sigma; \Gamma \vdash M : A'$ then $A \equiv A'$*
5. *Well-sortness: If $\Sigma; \Gamma \vdash M : A$ then $\Sigma; \Gamma \vdash A : s$ or $A = \mathtt{TYPE}, s = \mathtt{KIND}$*

▶ **Theorem 2** (Conv in context for DK). *Let $A \equiv A'$ with $\Sigma; \Gamma \vdash A' : s$. We have*

- $\Sigma; \Gamma, x : A, \Gamma'$ *WF* $\Rightarrow \Sigma; \Gamma, x : A', \Gamma'$ *WF*
- $\Sigma; \Gamma, x : A, \Gamma' \vdash M : B \Rightarrow \Sigma; \Gamma, x : A', \Gamma' \vdash M : B$

Proof. Structural induction on the derivation tree, using weakening for the case `Var`. ◀

▶ **Proposition 3** (Reduce type in judgement). *Suppose $\hookrightarrow_{\beta\mathcal{R}}$ is confluent. Then if $\Sigma; \Gamma \vdash_{\lambda\Pi} M : A$ and $A \hookrightarrow^* A'$ we have $\Sigma; \Gamma \vdash_{\lambda\Pi} M : A'$.*

Proof. By confluence, β satisfies subject reduction. Therefore, this is a direct consequence of well-sortness and subject reduction. ◀

3 Strong Normalization of β in Dedukti

In order to show the conservativity of encodings, one often needs to β normalize terms, thus requiring β to be normalizing for well-typed terms. In this section we generalize the proof of normalization of the λΠ-calculus given in [17] to Dedukti. More precisely, we show that, given that $\beta\mathcal{R}$ is confluent and *arity preserving* (a definition we will introduce in this section), then β is SN (strongly normalizing) in Dedukti for well-typed terms.

Note that, unlike works such as [8], which provide syntactic criteria on the normalization of $\beta\mathcal{R}$ in Dedukti, we only aim to show the normalization of β. In particular $\beta\mathcal{R}$ may not be SN in our setting. Our work has more similar goals to [4], which provides criteria for the SN of β in the Calculus of Constructions when adding object-level rewrite rules. However, our work also allows for type-level rewrite rules, which will be needed in our encoding.

Our proof works by defining an erasure map into the simply-typed λ-calculus, which is known to be SN, and then show that this map preserves typing and non-termination of

β, thus implying that β is SN in Dedukti. To do this, the erasure map must remove the dependency inside types, but as in Dedukti terms and types are all mixed together, we will first need to be able to separate them syntactically.

The syntactic stratification theorem (Theorem 9) is a standard property of Dedukti and is exactly what we need here. However, unlike the known variants in the literature, such as in [6], we prove a more general version not requiring subject reduction of $\beta\mathcal{R}$. The proof draws inspiration from a similar one in [4].

We proceed as follows. First we start by proving our generalization of the stratification theorem (Thereon 9). This is followed by the definition of the erasure function from Dedukti into the simply-typed λ-calculus (Definition 11). We then introduce our definition of *arity preserving* rewrite systems (Definition 13), and give some motivation of why the proof works. We then show that this function preserves both typing (Proposition 16) and non-termination (Proposition 17). Finally, by putting all this together, we will conclude by showing our main result (Theorem 18).

3.1 Syntactic stratification

▶ **Definition 4.** *We introduce the following basic definitions.*

1. *Given a signature Σ, a constant c is type-level (and referred by α, γ) if $c[\Delta] : A \in \Sigma$ with A of the form $\Pi\vec{x} : \vec{B} : \mathtt{TYPE}$, otherwise it is object-level (and referred by a, b).*
2. *A rewrite rule $c[\vec{l}] \longrightarrow r$ is type-level if its head symbol c is a type-level constant.*

As previously mentioned, our proof of the stratification theorem will not need subject reduction of $\mathcal{R}$. Instead, we will only need the following syntactic property.

▶ **Definition 5.** *We say that $\mathcal{R}$ is well-formed (with respect to Σ) if for all type-level rules, its right-hand side is in the following grammar, where $\vec{M}, N, B$ are any.*

$$R ::= \alpha[\vec{M}] \mid RN \mid \lambda x : B.R \mid \Pi x : R.R$$

We are now ready to define the syntactic classes of terms in Dedukti, through the following grammars. We will show in Theorem 9 that every typed term belong to one of these classes.

$$K ::= \mathtt{TYPE} \mid \Pi x : T.K \qquad \text{(Kinds)}$$

$$T ::= \alpha[\vec{O}] \mid TO \mid \lambda x : T.T \mid \Pi x : T.T \qquad \text{(Type Families)}$$

$$O ::= x \mid a[\vec{O}] \mid OO \mid \lambda x : T.O \qquad \text{(Objects)}$$

These grammars can easily be shown closed under object-level substitution.

▶ **Lemma 6** (Closure under object-level substitution)**.** *For all objects O, we have*

- *If M is an object, then $M\{O/x\}$ is an object*
- *If M is an type family, then $M\{O/x\}$ is a type family*
- *If M is a kind, then $M\{O/x\}$ is a kind*

A property that one would find natural is for these grammars to be closed under reduction. However, as it is shown by the following example, this is not the case.

▶ **Example 7.** Consider the rule $\alpha[\lambda x : y.x] \longrightarrow \alpha[y]$. With the substitution $y \mapsto \gamma$ we have $\alpha[\lambda x : \gamma.x] \longrightarrow \alpha[\gamma]$. The left-hand side is a type family, whereas the right-hand side is not in any of the grammars.

However, we can still define a weaker notion of pre-kinds and pre-type families for which closure under rewriting holds. This property will be key when proving the stratification theorem.

▶ **Lemma 8.** *Define the grammars*

$$L ::= \texttt{TYPE} \mid \Pi x : R.L \qquad \textit{(Pre-Kinds)}$$

$$R ::= \alpha[\vec{M}] \mid RN \mid \lambda x : A.R \mid \Pi x : R.R \qquad \textit{(Pre-Type Families)}$$

where $A, N, \vec{M}$ are any. If $\mathcal{R}$ is well-formed, then they are disjoint and closed under $\beta\mathcal{R}$.

Proof. Both grammars are clearly disjoint. Before showing closure under rewriting, we first show closure under substitution: for every pre-kind L, pre-type family R and term N, $L\{N/x\}$ is a pre-kind and $R\{N/x\}$ is a pre-type family (by induction on R and L). Then, by induction on the rewrite context and using closure under substitution we show the result. ◀

We are now ready to show the stratification theorem.

▶ **Theorem 9** (Syntactical stratification). *Suppose that $\beta\mathcal{R}$ is confluent and $\mathcal{R}$ is well formed. If $\Sigma; \Gamma \vdash M : A$ then exactly one of the following hold:*
1. *M is a kind and $A = \texttt{KIND}$*
2. *M is a type family and A is a kind*
3. *M is an object and A is a type family*

Proof. First note that the grammars are clearly disjoint, so only one of the cases can hold. We proceed by showing the rest by induction over $\Sigma; \Gamma \vdash M : A$.

Sort: Trivial.

Var: We have

$$x : A \in \Gamma \; \frac{\Sigma; \Gamma \; \texttt{well-formed}}{\Sigma; \Gamma \vdash x : A} \, \texttt{Var}$$

For some $\Gamma' \sqsubseteq \Gamma$, we have $\Sigma; \Gamma' \vdash A : \texttt{TYPE}$ with a smaller derivation tree. By IH, A is a type family, hence the result follows.

Cons: We have

$$c[\Delta] : A \in \Sigma \; \frac{\Sigma; \Delta \vdash A : s \qquad \Sigma; \Gamma \vdash \vec{M} : \Delta}{\Sigma; \Gamma \vdash c[\vec{M}] : A\{\vec{M}\}} \, \texttt{Cons}$$

We first show the following claim.

▷ Claim 10. $\vec{M}$ is made of objects.

Proof. Write $\Delta = x_1 : A_1, ..., x_n : A_n$. First note that $\Sigma; \Delta \vdash A : s$ implies $\Sigma; x_1 : A_1, ..., x_{i-1} : A_{i-1} \vdash A_i : \texttt{TYPE}$ with a smaller derivation tree, hence by the IH each A_i is a type family. We now show by induction on i that for $i = 1, ..., n$, M_i is an object[1].

For the case $i = 1$ this follows from $\Sigma; \Gamma \vdash M_1 : A_1$, by the outer IH and the fact that A_1 is a type family. For the induction step, we have $\Sigma; \Gamma \vdash M_i : A_i\{M_1/x_1\}...\{M_{i-1}/x_{i-1}\}$. We know that A_i is a type family, and by the inner IH we have that $M_1, ..., M_{i-1}$ are objects. By closure under object-level substitution, $A_i\{M_1/x_1\}...\{M_{i-1}/x_{i-1}\}$ is also a type family. Hence the outer IH implies that M_i is an object. ◁

[1] We will use the terms “inner IH” for the IH corresponding to this claim and “outer IH” for the IH corresponding to the whole thoerem.

We now proceed with the main proof obligation. If $s = \mathtt{KIND}$ by IH A is a kind, of the form $\Pi \vec{x} : \vec{B}.\mathtt{TYPE}$. Hence c is a type-level constant. Because c is type-level and $\vec{M}$ is made of objects, then $c[\vec{M}]$ is a type family. Finally, as $\vec{M}$ are objects, then by closure under object-level substitution $A\{\vec{M}\}$ is a kind. Hence we are in case 2.

If $s = \mathtt{TYPE}$ by IH A is a type family. Hence c is an object-level constant. Because c is object-level and $\vec{M}$ is made of objects, then $c[\vec{M}]$ is an object. Finally, as $\vec{M}$ are objects, then by closure under object-level substitution $A\{\vec{M}\}$ is a type family. Hence we are in case 3.

Conv: We have

$$A \equiv B \ \frac{\Sigma; \Gamma \vdash M : A \qquad \Sigma; \Gamma \vdash B : s}{\Sigma; \Gamma \vdash M : B}\ \mathtt{Conv}$$

By confluence, there is C with $A \longleftrightarrow^* C \longleftrightarrow^* B$. Note that type families and kinds are also respectively pre-type families and pre-kinds, which are disjoint and closed under rewriting. Therefore, an important remark is that both situations in which A is a type-family and B a kind, or A a kind and B a type family, are impossible.

We now proceed with the proof and consider the cases $s = \mathtt{TYPE}$ and $s = \mathtt{KIND}$.

If $s = \mathtt{TYPE}$ then B is a type family. Applying the IH to $M : A$, then by the previous remark we only need to consider the case in which A is a type family, and thus M is an object as required.

If $s = \mathtt{KIND}$, then B is a kind. Applying the IH to $M : A$, then by previous remark we only need to consider the case in which A is a kind, and thus M is a type family as required.

Prod: We have

$$\frac{\Sigma; \Gamma \vdash A : \mathtt{TYPE} \qquad \Sigma; \Gamma, x : A \vdash B : s}{\Sigma; \Gamma \vdash \Pi x : A.B : s}\ \mathtt{Prod}$$

We have either $s = \mathtt{TYPE}$ or $s = \mathtt{KIND}$.

If $s = \mathtt{TYPE}$, then by IH both A, B are type families, hence $\Pi x : A.B$ is a type family and we are in case (2).

If $s = \mathtt{KIND}$, then A is a type family and B a kind, hence $\Pi x : A.B$ is a kind and we are in case (1).

Abs: We have

$$\frac{\Sigma; \Gamma \vdash A : \mathtt{TYPE} \qquad \Sigma; \Gamma, x : A \vdash B : s \qquad \Sigma; \Gamma, x : A \vdash M : B}{\Sigma; \Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ \mathtt{Abs}$$

We have either $s = \mathtt{TYPE}$ or $s = \mathtt{KIND}$.

If $s = \mathtt{TYPE}$, then by IH both A, B are type families and M is an object. Hence $\lambda x : A.M$ is an object, $\Pi x : A.B$ is a type family and we are in case (3).

If $s = \mathtt{KIND}$, then B is a kind and A, M are type families. Hence $\Pi x : A.M$ is a type family, $\Pi x : A.B$ is a kind and we are in case (2).

App: We have

$$\frac{\Sigma; \Gamma \vdash M : \Pi x : A.B \qquad \Sigma; \Gamma \vdash N : A}{\Sigma; \Gamma \vdash MN : B\{N/x\}}\ \mathtt{App}$$

By the IH applied to $M : \Pi x : A.B$, $\Pi x : A.B$ is either a type family or a kind, hence in all cases A is a type family, and thus by the IH applied to $N : A$, N is an object.

If $\Pi x : A.B$ is a type family, then M is an object, and thus MN is also. As B is a type family and the grammars are closed by object-level substitution, $B\{N/x\}$ is also a type family. Hence we are in case (2).

If $\Pi x : A.B$ is a kind, then M is a type family, and thus MN is also. As B is a kind and the grammars are closed by object-level substitution, $B\{N/x\}$ is a kind. Hence we are in case (1). ◀

3.2 Erasure map

We are now ready to give the definition of the erasure map into the simply-typed λ-calculus.

▸ **Definition 11** (Erasure map). *Consider the simple types generated by the grammar*

$$\sigma ::= * \mid \sigma \to \sigma\,.$$

Moreover, let Γ_π be the context containing for each σ the declaration $\pi_\sigma : * \to (\sigma \to *) \to *$. We define the partial functions $\|-\|, |-|$ by the following equations.

$$\begin{aligned} \|\mathtt{TYPE}\| &= * \\ \|\alpha[\vec{M}]\| &= * \\ \|\Pi x : A.B\| &= \|A\| \to \|B\| \\ \|AN\| &= \|A\| \\ \|\lambda x : A.B\| &= \|B\| \end{aligned} \qquad \begin{aligned} |x| &= x \\ |a[\vec{M}]| &= a\ |\vec{M}| \\ |\alpha[\vec{M}]| &= \alpha\ |\vec{M}| \\ |MN| &= |M||N| \\ |\lambda x : A.M| &= (\lambda z.\lambda x.|M|)|A| \textit{ where } z \notin FV(M) \\ |\Pi x : A.B| &= \pi_{\|A\|}\ |A|\ (\lambda x.|B|) \end{aligned}$$

In particular, note that $|-|$ is defined for all objects and type families, and that $\|-\|$ is defined for all type-families and kinds. We also extend the definition of $\|-\|$ (partially) on contexts and signatures by the following equations.

$$\|-\| = -$$

$$\|x : A, \Gamma\| = x : \|A\|, \|\Gamma\|$$

$$\|c[x_1 : A_1, ..., x_n : A_n] : A; \Sigma\| = (c : \|A_1\| \to ... \to \|A_n\| \to \|A\|), \|\Sigma\|$$

In order to show the normalization of β, we need the erasure to preserve typing. The main obstacle when showing this is dealing with the `Conv` rule. To make the proof go through, we would need to show that if $A \equiv B$ then $\|A\| = \|B\|$. In the $\lambda\Pi$-calculus this can be easily shown, however because in DEDUKTI the relation $\equiv$ also takes into account the rewrite rules in $\mathcal{R}$, we can easily build counterexamples in which this does not hold.

▸ **Example 12.** Let $El, Prod, Nat$ be type-level constants, and consider the rule

$$El\ (Prod\ A\ B) \longrightarrow \Pi x : El\ A.El\ (B\ x)$$

traditionally used to build DEDUKTI encodings (as in [9]). Note that here we write $\alpha\ \vec{l}$ for $\alpha[\vec{l}]$, to ease the notation. We then have

$$El\ (Prod\ Nat\ (\lambda x.Nat)) \equiv \Pi x : El\ Nat.El\ ((\lambda x.Nat)\ x) \equiv El\ Nat \to El\ Nat$$

but $\|El\ (Prod\ Nat\ (\lambda x : .Nat))\| = *$ and $\|El\ Nat \to El\ Nat\| = * \to *$.

If we were to define the arity of a type[2] as the number of consecutive arrows (that is, of Πs), then we realize that the problem here is that rules such as $El\ (Prod\ A\ B) \longrightarrow \Pi x : El\ A.El\ (B\ x)$ do not preserve the arity. Indeed, $El\ (Prod\ A\ B)$ has arity 0 because it has no arrows, whereas $\Pi x : El\ A.El\ (B\ x)$ has arity 1 as it has one arrow[3]. As the left-hand side of a type-level rule always has arity 0 (because it is of the form $\alpha[\vec{l}]$), to remove these unwanted cases we need for their right-hand sides to also have arity 0. This motives the following definition.

[2] Note that this concept is different from the notion of arity of constants, as defined in Section 2.
[3] Using a different notation for the dependent product, we can write this type as $(x : El\ A) \to El\ (B\ x)$, which may help to clarify this assertion.

▶ **Definition 13** (Arity preserving). *$\mathcal{R}$ is said to be arity-preserving*[4] *if, for every type-level rewrite rule in $\mathcal{R}$, the right-hand side is in the following grammar, where $\vec{M}, N, A$ are any.*

$$R ::= \alpha[\vec{M}] \mid R\ N \mid \lambda x : A.R$$

It turns out that this definition, together with confluence of $\beta\mathcal{R}$, will be enough to show that the translation preserves typing, and also non-termination. Therefore, throughout the rest of this section we suppose the following assumptions.

▶ **Assumption 14.** *$\beta\mathcal{R}$ is confluent and $\mathcal{R}$ is arity preserving.*

Note that if $\mathcal{R}$ is arity preserving then it is also well-formed, and thus we can in particular also use the stratification theorem.

3.3 Proof of Strong Normalization of β in Dedukti

We start with the following key lemma, which ensures that convertible types are erased into the same simple type by $\|-\|$.

▶ **Lemma 15** (Key property).

1. *If $\|A\|$ is defined, then for all N, $\|A(N/x)\|$ is also defined and $\|A\| = \|A(N/x)\|$.*
2. *If $A \longrightarrow A'$ and $\|A\|$ is defined, then $\|A'\|$ is also and $\|A\| = \|A'\|$.*
3. *If $A \equiv A'$ and $\|A\|, \|A'\|$ are well defined, then $\|A\| = \|A'\|$.*

Proof. 1. By induction on A.
2. By induction on the rewrite context. For the base case of β, we use part 1. For the base case of a rule in $\mathcal{R}$, this rule needs to be type-level, of the form $\alpha[\vec{l}] \longrightarrow r$. Note that for every substitution σ, we have $\|\alpha[\vec{l}\{\sigma\}]\| = *$. Thus, it suffices to show that for every σ, $\|r\{\sigma\}\|$ is defined and equal to $*$, which is done by induction on the grammar of Definition 13.
3. Follows from confluence and part 2. ◀

With the key property in hand, we can show that the erasure preserves typing.

▶ **Theorem 16** (Preservation of typing). *If $\Sigma; \Gamma \vdash M : A$ and $A \neq \texttt{KIND}$, then there is $\Sigma' \subseteq \Sigma$ such that $\Gamma_\pi, \|\Sigma'\|, \|\Gamma\| \vdash_\lambda |M| : \|A\|$*

Proof. First note that for all $x : A \in \Gamma$, we have $\Sigma; \Gamma \vdash A : \texttt{TYPE}$, thus by syntactic stratification A is a type-family, and thus $\|\Gamma\|$ is well-defined. We proceed by induction on the derivation. The base cases Var and Sort are trivial.

Cons: We have

$$c[\Delta] : A \in \Sigma\ \frac{\Sigma; \Gamma\ \texttt{well-formed} \qquad \Sigma; \Delta \vdash A : s \qquad \Sigma; \Gamma \vdash \vec{M} : \Delta}{\Sigma; \Gamma \vdash c[\vec{M}] : A\{\vec{M}\}}\ \texttt{Cons}$$

We first show that $\|c[\Delta] : A\|$ is defined. Write $\Delta = x_1 : A_1, ..., x_n : A_n$. First note that $\Sigma; \Delta \vdash A : s$ implies that for all $x_i : A_i \in \Delta$ we have $A_i : \texttt{TYPE}$, hence by stratification each A_i is a type family and $\|-\|$ is defined for all of them. Moreover, by stratification $A : s$ implies that A is either a type family or kind, hence $\|A\|$ is defined. Hence, $\|c[\Delta] : A\| = c : \|A_1\| \to ...\|A_n\| \to \|A\|$ is well-defined.

[4] More precisely, this definition also depends on the signature Σ, as this is used to define which constants are type-level.

We now proceed with the main proof obligation. By IH for $i = 1, ..., n$ we have $\Sigma_i \subseteq \Sigma$ such that $\Gamma_\pi, \|\Sigma_i\|, \|\Gamma\| \vdash |M_i| : \|A_i\{M_1/x_1\}...\{M_{i-1}/x_{i-1}\}\|$. Then, by Lemma 15 we have $\|A_i\{M_1/x_1\}...\{M_{i-1}/x_{i-1}\}\| = \|A_i\|$. Therefore, by taking

$$\Sigma' = c[\Delta] : A \cup \Sigma_1 \cup ... \cup \Sigma_n$$

we can derive $\Gamma_\pi, \|\Sigma'\|, \|\Gamma\| \vdash_\lambda c\ |\vec{M}| : \|A\|$. Because $\|A\| = \|A\{\vec{M}\}\|$, the result follows.

Conv: We have

$$\frac{\Sigma; \Gamma \vdash M : A \qquad \Sigma; \Gamma \vdash B : s \qquad A \equiv B}{\Sigma; \Gamma \vdash M : B}\ \texttt{Conv}$$

First note that A cannot be KIND. Indeed, by confluence we would have $B \longrightarrow^* \texttt{KIND}$, but by syntactic stratification B is either a kind or a type family. As kinds and type families are in particular pre-kinds and pre-type families, B is one those. But as they are closed under rewriting, this would imply that KIND is a pre-kind or a pre-type family, absurd.

Therefore, by IH we have $\Gamma_\pi, \|\Sigma'\|, \|\Gamma\| \vdash_\lambda |M| : \|A\|$ for some $\Sigma' \subseteq \Sigma$. Moreover, by syntactic stratification A, B are kinds or type families, thus $\|-\|$ is defined for them. By the Key Property $A \equiv B$ implies $\|A\| = \|B\|$ and thus the result follows.

Prod: We have

$$\frac{\Sigma; \Gamma \vdash A : \texttt{TYPE} \qquad \Sigma; \Gamma, x : A \vdash B : s}{\Sigma; \Gamma \vdash \Pi x : A.B : s}\ \texttt{Prod}$$

If $s = \texttt{KIND}$ there is nothing to show, thus we consider $s = \texttt{TYPE}$. By IH, for some $\Sigma', \Sigma'' \subseteq \Sigma$ we have $\Gamma_\pi, \|\Sigma'\|, \|\Gamma\| \vdash_\lambda |A| : *$ and $\Gamma_\pi, \|\Sigma''\|, \|\Gamma\|, x : \|A\| \vdash_\lambda |B| : *$. We thus get $\Gamma_\pi, \|\Sigma''\|, \|\Gamma\| \vdash_\lambda \lambda x.|B| : \|A\| \to *$, and therefore $\Gamma_\pi, \|\Sigma' \cup \Sigma''\|, \|\Gamma\| \vdash_\lambda \pi_{\|A\|}\ |A|\ (\lambda x.|B|) : *$

Abs: We have

$$\frac{\Sigma; \Gamma \vdash A : \texttt{TYPE} \qquad \Sigma; \Gamma, x : A \vdash B : s \qquad \Sigma; \Gamma, x : A \vdash M : B}{\Sigma; \Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ \texttt{Abs}$$

By IH, for some $\Sigma', \Sigma'' \subseteq \Sigma$ we have $\Gamma_\pi, \|\Sigma'\|, \|\Gamma\| \vdash_\lambda |A| : *$ and $\Gamma_\pi, \|\Sigma''\|, \|\Gamma\|, x : \|A\| \vdash_\lambda |M| : \|B\|$, from which we deduce $\Gamma_\pi, \|\Sigma''\|, \|\Gamma\| \vdash_\lambda \lambda x.|M| : \|A\| \to \|B\|$. By adding some spurious variable z of type $*$ to the context and abstracting over it, we get $\Gamma_\pi, \|\Sigma''\|, \|\Gamma\| \vdash_\lambda \lambda z.\lambda x.|M| : * \to \|A\| \to \|B\|$. Finally, we use application to conclude $\Gamma_\pi, \|\Sigma' \cup \Sigma''\|, \|\Gamma\| \vdash_\lambda (\lambda z.\lambda x.|M|)|A| : \|A\| \to \|B\|$.

App: We have

$$\frac{\Sigma; \Gamma \vdash M : \Pi x : A.B \qquad \Sigma; \Gamma \vdash N : A}{\Sigma; \Gamma \vdash MN : B\{N/x\}}\ \texttt{App}$$

By IH, for some $\Sigma', \Sigma'' \subseteq \Sigma$ we deduce $\Gamma_\pi, \|\Sigma'\|, \|\Gamma\| \vdash_\lambda |M| : \|A\| \to \|B\|$ and $\Gamma_\pi, \|\Sigma''\|, \|\Gamma\| \vdash_\lambda |N| : \|A\|$. By application, we get $\Gamma_\pi, \|\Sigma' \cup \Sigma''\|, \|\Gamma\| \vdash_\lambda |N||M| : \|B\|$, and as $\|B\| = \|B\{N/x\}\|$ the result follows. ◂

▸ **Proposition 17** (Preservation of non-termination)**.** *Let M be an object or type family.*

1. *If N is an object, then $M\{N/x\}$ is an object or type family and $|M\{N/x\}| = |M|\{|N|/x\}$.*
2. *If $M \longrightarrow_\beta N$ then N is an object or type family and $|M| \longrightarrow_\beta^+ |N|$.*

Proof. **1.** By induction on M, and using Lemma 15 for the case $M = \Pi x : A.B$.

2. By induction on the rewriting context. For the base case, we have $M = (\lambda x : A.M_1)M_2 \longrightarrow M_1\{M_2/x\}$ and thus $|M| = (\lambda z.\lambda x.|M_1|)|A||M_2|$. As z is not free in $|M_1|$, we have $|M| = (\lambda z.\lambda x.|M_1|)|A||M_2| \longrightarrow (\lambda x.|M_1|)|M_2| \longrightarrow |M_1|\{|M_2|/x\}$. By part 1, $|M_1\{M_2/x\}|$ is well-defined and equal to $|M_1|\{|M_2|/x\}$.

The induction steps are all similar, we present two of them to show the idea. If $M = \lambda x : A.M' \longrightarrow \lambda x : A'.M' = N$, where $A \longrightarrow A'$, then by IH we have $|A| \longrightarrow^{+} |A'|$, and thus $|M| = (\lambda z.\lambda x.|M'|)|A| \longrightarrow^{+} (\lambda z.\lambda x.|M'|)|A'| = |N|$. If $M = \Pi x : A.B \longrightarrow \Pi x : A'.B = N$, where $A \longrightarrow A'$, then by IH we have $|A| \longrightarrow^{+} |A'|$. By the key property, this implies $\|A\| = \|A'\|$, and thus $|M| = \pi_{\|A\|}\ |A|\ (\lambda x.|B|) \longrightarrow^{+} \pi_{\|A'\|}\ |A'|\ (\lambda x.|B|)$. ◂

▸ **Theorem 18** (β is SN in DEDUKTI). *If $\beta\mathcal{R}$ is confluent and $\mathcal{R}$ is arity-preserving, then β is strongly normalizing for well-typed terms in* DEDUKTI.

Proof. Suppose that M satisfies $\Sigma; \Gamma \vdash_{\mathtt{DK}} M : A$ and there is an infinite sequence $M = M_1 \longrightarrow_{\beta} M_2 \longrightarrow_{\beta} M_3 \longrightarrow_{\beta} \ldots$ starting from M. We now show that for some N, $|N|$ is well-typed in the simply-typed λ-calculus and an infinite sequence starts from $|N|$.

If $A \neq \mathtt{KIND}$, then this follows directly from Proposition 16 by taking $N = M$. If $A = \mathtt{KIND}$, then M is of the form $\Pi\vec{x} : \vec{B}.\mathtt{TYPE}$, and as there are finitely many Bs and they are all type families, we conclude that there is a type family B_i from which an infinite sequence starts. We can thus take $N = B_i$ and apply Proposition 16 to get the result.

Now note that as objects and type-families are closed under β, then $|-|$ is defined for all elements in the sequence. Therefore, by taking the image of this infinite sequence under $|-|$ we also get an infinite sequence, by Proposition 17. This is a contradiction with the strong normalization of β in the simply typed λ-calculus, hence the result follows. ◂

4 Pure Type Systems

Pure type systems (or PTSs) is a class of type systems that generalizes many other systems, such as the Calculus of Constructions and System F. They are parameterized by a set of *sorts* $\mathcal{S}$ and two relations $\mathcal{A} \subseteq \mathcal{S}^2, \mathcal{R} \subseteq \mathcal{S}^3$. In this work we restrict ourselves to functional PTSs, for which $\mathcal{A}$ and $\mathcal{R}$ are functional relations. This restriction covers almost all of PTSs used in practice, and gives a much more well behaved metatheory.

In this paper we consider a variant of PTSs with explicit parameters. That is, just like when taking the projection of a pair $\pi^1(p)$ we can explicit all parameters and write $\pi^1(A, B, p)$ where $p : A \times B$, we can also write $\lambda(A, [x]B, [x]M)$ instead of $\lambda x : A.M$ and $@(A, [x]B, M, N)$ instead of MN. Moreover, if $- \times -$ is a universe-polymorphic definition, we should also write $\pi^1_{s_A, s_B}(A, B, p)$ to explicit the sort parameters. As in PTSs the dependent product is used across sorts, we then should also write $\lambda_{s_A, s_B}(A, [x]B, [x]M)$, $@_{s_A, s_B}(A, [x]B, M, N)$ and $\Pi_{s_A, s_B}(A, [x]B)$. To be more technical, we render explicit the parameters on the dependent product type and on its constructor (abstraction) and eliminator (application). Because of this interpretation in which we are rendering the parameters of λ and $@$ explicit, we name this version of PTSs as Explicitly-typed Pure Type Systems (EPTSs).

Reduction is then defined by the context closure of the β rules[5]

$$@_{s_1, s_2}(A, [x]B, \lambda_{s_1, s_2}(A', [x]B', [x]M), N) \longrightarrow M\{N/x\}$$

given for each $(s_1, s_2, s_3) \in \mathcal{R}$. Typing is given by the rules in Figure 2.

This modification is just a technical change that will help us during the translation, as our encoding needs the data of such parameters often left implicit. Other works such as [20]

[5] We consider a linearized variant of the expected non-left linear rule $@_{s_1, s_2}(A, [x]B, \lambda_{s_1, s_2}(A, [x]B, [x]M), N) \longrightarrow M\{N/x\}$, which is non-confluent in untyped terms. By linearizing it, we get a much more well-behaved rewriting system, where confluence holds for all terms. Moreover, whenever the left hand side is well-typed, the typing constraints impose $A \equiv A'$ and $B \equiv B'$.

$$\frac{}{- \text{ well-formed}}\ \textsc{Empty} \qquad x \notin \Gamma\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \text{ well-formed}}\ \textsc{Decl}$$

$$A \equiv B\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \textsc{Conv} \qquad (s_1, s_2) \in \mathcal{A}\ \frac{\Gamma \text{ well-formed}}{\Gamma \vdash s_1 : s_2}\ \textsc{Sort}$$

$$x : A \in \Gamma\ \frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A}\ \textsc{Var} \qquad (s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi_{s_1,s_2}(A, [x]B) : s_3}\ \textsc{Prod}$$

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda_{s_1,s_2}(A, [x]B, [x]M) : \Pi_{s_1,s_2}(A, [x]B)}\ \textsc{Abs}$$

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma \vdash N : A \qquad \Gamma \vdash M : \Pi_{s_1,s_2}(A, [x]B)}{\Gamma \vdash @_{s_1,s_2}(A, [x]B, M, N) : B\{N/x\}}\ \textsc{App}$$

Figure 2 Typing rules for Explicitly-typed Pure Type Systems

and [21] also consider similar variants, thought none of them correspond exactly to ours. Therefore, we had to develop the basic metatheory of our version in [12], and we have found that the usual meta-theoretical properties of functional PTSs are preserved when moving to the explicitly-typed version. More importantly, by a proof that uses ideas present in [21], we have shown the following equivalence.

▸ **Theorem 19** (Equivalence between PTSs and EPTSs[12])**.** *Let $|-|$ be the erasure map defined in the most natural way from a functional EPTS to its corresponding functional PTS. Given Γ, M, A in a PTS, we have*

$$\Gamma \vdash_{PTS} M : A \iff \exists \Gamma', M', A' st\ \Gamma' \vdash_{EPTS} M' : A' \wedge |\Gamma'| = \Gamma \wedge |M'| = M \wedge |A'| = A\,.$$

We note that functional EPTSs satisfy the following basic properties, whose proofs can be found in [12].

▸ **Proposition 20** (Weakening)**.** *Let $\Gamma \sqsubseteq \Gamma'$. We have*

- $\Gamma, x : A$ well-formed $\Rightarrow \Gamma', x : A$ well-formed *when* $x \notin \Gamma'$
- $\Gamma \vdash M : A \Rightarrow \Gamma' \vdash M : A$

▸ **Proposition 21** (Inversion)**.** *If $\Gamma \vdash M : C$ then*

- *If $M = x$, then*
  - Γ well-formed *with a smaller derivation tree*
  - *there is x with $x : A \in \Gamma$ and $C \equiv A$*
- *If $M = s$, then there is s' with $(s, s') \in \mathcal{A}$ and $C \equiv s'$*
- *If $M = \Pi_{s_1,s_2}(A, [x]B)$ then*
  - $\Gamma \vdash A : s_1$ *with a smaller derivation tree*
  - $\Gamma, x : A \vdash B : s_2$ *with a smaller derivation tree*
  - *there is s_3 with $(s_1, s_2, s_3) \in \mathcal{R}$ and $C \equiv s_3$*
- *If $M = \lambda_{s_1,s_2}(A, [x]B, [x]N)$ then*
  - $\Gamma \vdash A : s_1$ *with a smaller derivation tree*
  - $\Gamma, x : A \vdash B : s_2$ *with a smaller derivation tree*
  - *there is s_3 with $(s_1, s_2, s_3) \in \mathcal{R}$*
  - $\Gamma, x : A \vdash N : B$ *with a smaller derivation tree*
  - $C \equiv \Pi_{s_1,s_2}(A, [x]B)$
- *If $M = @_{s_1,s_2}(A, [x]B, N_1, N_2)$ then*

- $\Gamma \vdash A : s_1$ *with a smaller derivation tree*
- $\Gamma, x : A \vdash B : s_2$ *with a smaller derivation tree*
- *there is* s_3 *with* $(s_1, s_2, s_3) \in \mathcal{R}$
- $\Gamma \vdash N_1 : A$ *with a smaller derivation tree*
- $\Gamma \vdash N_2 : \Pi_{s_1,s_2}(A, [x]B)$ *with a smaller derivation tree*
- $C \equiv B(N_2/x)$

▸ **Proposition 22** (Conv in context). *Let* $A \equiv A'$ *and* $\Gamma \vdash A' : s$. *We have*

- $\Gamma, x : A, \Gamma'$ well-formed $\Rightarrow \Gamma, x : A', \Gamma'$ well-formed
- $\Gamma, x : A, \Gamma' \vdash M : B \Rightarrow \Gamma, x : A', \Gamma' \vdash M : B$

▸ **Proposition 23** (Substitution in judgment). *Let* $\Gamma \vdash N : A$. *We have*

- $\Gamma, x : A, \Gamma'$ well-formed $\Rightarrow \Gamma, \Gamma'(N/x)$ well-formed
- $\Gamma, x : A, \Gamma' \vdash M : B \Rightarrow \Gamma, \Gamma'(N/x) \vdash M(N/x) : B(N/x)$

▸ **Proposition 24** (Uniqueness of types). *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *we have* $A \equiv B$.

▸ **Corollary 25** (Uniqueness of sorts). *If* $\Gamma \vdash M : s$ *and* $\Gamma \vdash M : s'$ *we have* $s = s'$.

5 Encoding EPTSs in Dedukti

This section presents our encoding of functional EPTSs in Dedukti. In order to ease the notation, from now one we write $c\ \vec{M}$ for $c[\vec{M}]$. The basis for the encoding is given by a theory $(\Sigma_{\mathrm{EPTS}}, \mathcal{R}_{\mathrm{EPTS}})$ which we will construct step by step here.

Pure Type Systems (explicitly-typed or not) feature two kinds of types: dependent products and universes. We start by building the representation of the latter. For each $s \in \mathcal{S}$ we declare a type U_s to represent the type of elements of s. However, as the terms A with $\Gamma \vdash_{EPTS} A : s$ are themselves types, we also need to declare a function El_{s_1} which maps each such A to its corresponding type. As for each $(s_1, s_2) \in \mathcal{A}$ we have $\vdash_{EPTS} s_1 : s_2$, we also declare a constant u_{s_1} in U_{s_2} to represent this. Finally, as the sorts s_1 with $(s_1, s_2) \in \mathcal{A}$ now can be represented by both U_{s_1} and $El_{s_2}\ u_{s_1}$, we add a rewrite rule to identify these representations. This encoding resembles the definition of universes in type theories *à la* Tarski, and also follows traditional representations of universes in Dedukti as in [9].

$$\begin{array}{|ll|} U_s : \mathtt{TYPE} & \\ El_s[A : U_s] : \mathtt{TYPE} & \text{for } s \in \mathcal{S} \end{array} \qquad \begin{array}{|ll|} u_{s_1} : U_{s_2} & \\ El_{s_2}\ u_{s_1} \longrightarrow_{u_{s_1}\text{-red}} U_{s_1} & \text{for } (s_1, s_2) \in \mathcal{A} \end{array}$$

We now move to the representation of the dependent product type. We first declare a constant to represent the type formation rule for the dependent product.

$$\begin{array}{|ll|} Prod_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}] : U_{s_3} & \text{for } (s_1, s_2, s_3) \in \mathcal{R} \end{array}$$

Traditional Dedukti encodings would normally continue here by introducing the rule $El_{s_3}\ (Prod_{s_1,s_2}\ A\ B) \longrightarrow \Pi x : El_{s_1}\ A. El_{s_2}\ (B\ x)$, identifying the dependent product of the encoded theory with the one of Dedukti, thus allowing for the use of the framework's abstraction, application and β to represent the ones of the encoded system. We instead keep them separate and declare constants representing the introduction and elimination rules for the dependent product being encoding, that is, representing abstraction and application.

$$\mathit{abs}_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}; M : \Pi x : El_{s_1}\ A.El_{s_2}\ (B\ x)] : El_{s_3}(Prod_{s_1,s_2}\ A\ B)$$
$$\mathit{app}_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}; M : El_{s_3}(Prod_{s_1,s_2}\ A\ B); N : El_{s_1}\ A] : El_{s_2}(B\ N)$$
$$\mathit{app}_{s_1,s_2}\ A\ B\ (\mathit{abs}_{s_1,s_2}\ A'\ B'\ M)\ N \longrightarrow_{beta_{s_1,s_2}} M\ N \qquad \text{for } (s_1, s_2, s_3) \in \mathcal{R}$$

We note that this idea is also hinted in [1], thought they did not pursued it further. This approach also reassembles the one of the Edinburgh Logical Framework (ELF) [17] in which the framework's abstraction is used exclusively for binding. We are however able to encode computation directly as computation with the rule $beta_{s_1,s_2}$, whereas the ELF handles computation by encoding it as an equality judgment, thus introducing explicit coercions in the terms. Some other variants such as [16] prevent the introduction of such coercions, but computation is still represented by an equality judgment instead of being represented by computation.

We are now ready to define the translation function $[\![-]\!]$.

$$\begin{aligned}
[\![x]\!] &= x\\
[\![s]\!] &= u_s\\
[\![\Pi_{s_1,s_2}(A, [x]B)]\!] &= Prod_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\\
[\![\lambda_{s_1,s_2}(A, [x]B, [x]M)]\!] &= abs_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ (\lambda x : El_{s_1}\ [\![A]\!].[\![M]\!])\\
[\![@_{s_1,s_2}(A, [x]B, M, N)]\!] &= app_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ [\![M]\!]\ [\![N]\!]
\end{aligned}$$

We also extend $[\![-]\!]$ to well-formed contexts by the following definition. Note that because we are dealing with functional EPTSs, the sort of A in Γ is unique, hence the following definition makes sense.

$$\begin{aligned}
[\![-]\!] &= -\\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : El_{s_A}\ [\![A]\!] \quad \text{where } \Gamma \vdash A : s_A
\end{aligned}$$

▸ Remark 26. Note that in the definitions of $[\![-]\!]$ it was essential for λ and $@$ to explicit the types A and B, as the constants abs_{s_1,s_2} and app_{s_1,s_2} require their translations. Had we had for instance just $\lambda x : A.M$, we could then make the translation dependent on Γ and take a B such that $\Gamma, x : A \vdash M : B$. However, because $[\![-]\!]$ is defined by induction and B is not a subterm of $\lambda x : A.M$, we cannot apply $[\![-]\!]$ to B. Therefore, when doing an encoding in Dedukti one should first render explicit the needed data before translating, and then show an equivalence theorem between the explicit and implicit versions (in our case, Theorem 19).

Moreover, note that by also making the sorts explicit in $\lambda_{s_1,s_2}, @_{s_1,s_2}, \Pi_{s_1,s_2}$ our translation can be defined purely syntactically. If this information were not in the syntax, we could still define $[\![-]\!]$ by making it dependent on Γ. Nevertheless, this complicates many proofs, as each time we apply $[\![-]\!]_\Gamma$ to a term we need to know it is well-typed in Γ.

In order to understand more intuitively how the encoding works, let's look at an example.

▸ **Example 27.** Recall that System F can be defined by the sort specification $\mathcal{S} = \{Type, Kind\}, \mathcal{A} = \{Type : Kind\}, \mathcal{R} = \{(Type, Type, Type), (Kind, Type, Type)\}$. In this EPTS, we can express the polymorphic identity function, traditionally written as $\lambda A : Type.\lambda x : A.x$, by

$$\lambda_{Kind,Type}(Type, [A]\Pi(A, [x]A), [A]\lambda_{Type,Type}(A, [x]A, [x]x))$$

This term is represented in our encoding by

$$abs_{Kind,Type},\ u_{Type}\ (\lambda A.Prod_{Type,Type}\ A\ (\lambda x.A))\ (\lambda A.abs_{Type,Type}\ A\ (\lambda x.A)\ (\lambda x.x))$$

where we omit the type annotations in the abstractions, to improve readability.

6 Soundness

An encoding is said to be sound when it preserves the typing relation of the original system. In this section we will see that our encoding has this fundamental property. We start by establishing some conventions in order to ease notations.

▶ **Convention 28.** *We establish the following notations.*

- *We write $\Sigma; \Gamma \vdash_{\mathsf{DK}} M : A$ for a* Dedukti *judgment and $\Gamma \vdash M : A$ (not $\vdash_{EPTS}$) for an EPTS judgment*
- *As the same signature Σ_{EPTS} is used everywhere, when writing $\Sigma_{\mathsf{EPTS}}; \Gamma \vdash_{\mathsf{DK}} M : A$ we omit it and write $\Gamma \vdash_{\mathsf{DK}} M : A$.*

Before showing soundness, we start by establishing some basic results.

▶ **Proposition 29** (Basic properties). *We have the following basic properties.*

1. *Confluence: The rewriting rules of the encoding are confluent with β.*
2. *Well-formedness of the signature: For all $c[\Delta] : A \in \Sigma_{\mathsf{EPTS}}$, we have $\Delta \vdash_{\mathsf{DK}} A : s$.*
3. *Subject reduction for β: If $\Gamma \vdash_{\mathsf{DK}} M : A$ and $M \longrightarrow_{\beta} M'$ then $\Gamma \vdash_{\mathsf{DK}} M' : A$.*
4. *Strong normalization for β: If $\Gamma \vdash_{\mathsf{DK}} M : A$, the β is strongly normalizing for M.*
5. *Compositionality: For all $M, N \in \Lambda_{EPTS}$ we have $[\![M]\!]\{[\![N]\!]/x\} = [\![N\{N/x\}]\!]$.*

Proof. 1. The considered rewrite rules form an orthogonal combinatory reduction system, and therefore are confluent[19].
2. Can be shown for instance with Lambdapi[10], an implementation of Dedukti.
3. Subject reduction of β is implied by confluence of $\beta\mathcal{R}_{\mathsf{EPTS}}$[6].
4. $\mathcal{R}_{\mathsf{EPTS}}$ is arity preserving and $\beta\mathcal{R}_{\mathsf{EPTS}}$ is confluent, thus *β is SN in Dedukti* (Theorem 18) applies.
5. By induction on M. ◀

▶ Remark 30. We could also show subject reduction of our encoding, either using the method in [7] or Lambdapi[10]. However, we will see that our proof does not actually require subject reduction of $\mathcal{R}_{\mathsf{EPTS}}$. Therefore, we conjecture that our proof method can also be adapted to systems that do not satisfy subject reduction.

▶ **Lemma 31** (Preservation of computation). *Let $M, N \in \Lambda_{EPTS}$. We have*

1. *$M \longrightarrow N$ implies $[\![M]\!] \longrightarrow^* [\![N]\!]$*
2. *$M \equiv N$ implies $[\![M]\!] \equiv [\![N]\!]$*

Proof. The first part is shown by induction on the rewriting context, using compositionality of $[\![-]\!]$ for the base case. The second part follows by induction on $\equiv$ and uses part 1. ◀

Recall that a sort $s \in \mathcal{S}$ is said to be a top-sort if there is no s' with $(s, s') \in \mathcal{A}$. The following auxiliary lemma allows us to switch between sort representations and is heavily used in the proof of soundness.

▶ **Lemma 32** (Equivalence for sort representations). *If s is not a top-sort, then*

$$\Gamma \vdash_{\mathsf{DK}} M : U_s \iff \Gamma \vdash_{\mathsf{DK}} M : El_{s'}\ u_s$$

where $(s, s') \in \mathcal{A}$.

With all these results in hand, we can now show the soundness of our encoding.

▶ **Theorem 33** (Soundness). *Let Γ be a context and M, A terms in an EPTS. We have*

- *If* Γ well-formed *then* $[\![\Gamma]\!]$ `well-formed`
- *If* $\Gamma \vdash M : A$ *then*
  - *if* A *is a top-sort then* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![M]\!] : U_A$
  - *else* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![M]\!] : El_{s_A}\ [\![A]\!]$, *where* $\Gamma \vdash A : s_A$

Proof. By structural induction on the proof of the judgment. Easy for the cases Empty and Var.

Case Decl: The proof ends with

$$x \notin A \ \frac{\Gamma \vdash A : s}{\Gamma \vdash x : A}\ \textsc{Decl}$$

From the IH we can derive $[\![\Gamma]\!] \vdash_{\mathtt{DK}} El_s\ [\![A]\!] : \mathtt{TYPE}$, therefore we can apply `Decl` to get $[\![\Gamma]\!], x : El_s\ [\![A]\!]$ `well-formed`.

Case Sort: The proof ends with

$$(s_1, s_2) \in \mathcal{A}\ \frac{\Gamma \text{ well-formed}}{\Gamma \vdash s_1 : s_2}\ \textsc{Sort}$$

By IH we have $[\![\Gamma]\!]$ `well-formed`, therefore we can show $[\![\Gamma]\!] \vdash_{\mathtt{DK}} u_{s_1} : U_{s_2}$ using `Cons`. If s_2 is not a top-sort, we use Lemma 32 to show $[\![\Gamma]\!] \vdash_{\mathtt{DK}} u_{s_1} : El_{s_3}\ u_{s_2}$, where $(s_2, s_3) \in \mathcal{A}$.

Case Prod: The proof ends with

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi_{s_1,s_2}(A, [x]B) : s_3}\ \textsc{Prod}$$

By the IH and Lemma 32, we have $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![A]\!] : U_{s_1}$ and $[\![\Gamma]\!], x : El_{s_1}\ [\![A]\!] \vdash_{\mathtt{DK}} [\![B]\!] : U_{s_2}$. By `Abs` we get $[\![\Gamma]\!] \vdash_{\mathtt{DK}} \lambda x : El_{s_1}\ [\![A]\!].[\![B]\!] : El_{s_1}\ [\![A]\!] \to U_{s_2}$, therefore it suffices to apply `Cons` with $Prod_{s_1,s_2}$ to conclude

$$[\![\Gamma]\!] \vdash_{\mathtt{DK}} Prod_{s_1,s_2}\ [\![A]\!]_\Gamma\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!]) : U_{s_3}$$

If s_3 is not a top-sort, we then apply Lemma 32.

Case App: The proof ends with

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma \vdash M : \Pi_{s_1,s_2}(A, [x]B) \qquad \Gamma \vdash N : A}{\Gamma \vdash @_{s_1,s_2}(A, [x]B, M, N) : B(N/x)}\ \textsc{App}$$

By the IH and Lemma 32, we have $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![A]\!] : U_{s_1}$, $[\![\Gamma]\!], x : El_{s_1}\ [\![A]\!] \vdash_{\mathtt{DK}} [\![B]\!] : U_{s_2}$, $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![M]\!] : El_{s_3}\ (Prod_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!]))$ and $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![N]\!] : El_{s_1}\ [\![A]\!]$. By `Abs` we get $[\![\Gamma]\!] \vdash_{\mathtt{DK}} \lambda x : El_{s_1}\ [\![A]\!].[\![B]\!] : El_{s_1}\ [\![A]\!] \to U_{s_2}$, therefore we can apply `Cons` with app_{s_1,s_2} to get

$$[\![\Gamma]\!] \vdash_{\mathtt{DK}} app_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ [\![M]\!]\ [\![N]\!] : El_{s_2}\ ((\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ [\![N]\!])$$

Therefore, from *Reduce type in judgement* (Proposition 3) with $(\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])[\![N]\!] \longrightarrow [\![B]\!]\{[\![N]\!]/x\}$ and compositionality of $[\![-]\!]$ we get

$$[\![\Gamma]\!] \vdash_{\mathtt{DK}} app_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ [\![M]\!]\ [\![N]\!] : El_{s_2}\ [\![B\{N/x\}]\!]\,.$$

Finally, note that $\Gamma \vdash N : A$ and $\Gamma, x : A \vdash B : s_2$ imply $\Gamma \vdash B\{N/x\} : s_2$, thus $B\{N/x\}$ is not a top-sort.

Case Conv: The derivation ends with

$$A \equiv B\ \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : s}{\Gamma \vdash M : A}\ \textsc{Conv}$$

First note that by confluence and subject reduction of rewriting in the EPTS, $\Gamma \vdash B : s$, thus B is not a top sort. Therefore, by the IH we have $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![M]\!] : El_s\ [\![B]\!]$. By the IH applied to $\Gamma \vdash A : s$ we can show $[\![\Gamma]\!] \vdash_{\mathtt{DK}} El_s\ [\![A]\!] : \mathtt{TYPE}$, and by *Preservation of computation* (Lemma 31) applied to $A \equiv B$ we get $[\![A]\!] \equiv [\![B]\!]$. Therefore, it suffices to apply `Conv` to conclude $[\![\Gamma]\!] \vdash [\![M]\!] : El_s\ [\![A]\!]$.

Case Abs: The derivation ends with

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda_{s_1,s_2}(A, [x]B, [x]N) : \Pi_{s_1,s_2}(A, [x]B)}\ \textsc{Abs}$$

By the IH and Lemma 32, we have $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![A]\!] : U_{s_1}$, $[\![\Gamma]\!], x : El_{s_1}\ [\![A]\!] \vdash_{\mathtt{DK}} [\![B]\!] : U_{s_2}$ and $[\![\Gamma]\!], x : El_{s_1}\ [\![A]\!] \vdash_{\mathtt{DK}} [\![N]\!] : El_{s_2}\ [\![B]\!]$. By `Abs` we get $[\![\Gamma]\!] \vdash_{\mathtt{DK}} \lambda x : El_{s_1}\ [\![A]\!].[\![B]\!] : El_{s_1}\ [\![A]\!] \to U_{s_2}$ and $[\![\Gamma]\!] \vdash_{\mathtt{DK}} \lambda x : El_{s_1}\ [\![A]\!].[\![N]\!] : \Pi x : El_{s_1}\ [\![A]\!].El_{s_2}\ [\![B]\!]$.

Using inversion of typing, it is not difficult to show that

$$[\![\Gamma]\!] \vdash_{\mathtt{DK}} \Pi x : El_{s_1}\ A.El_{s_2}\ ((\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ x) : \mathtt{TYPE}\,.$$

Hence, because $\Pi x : El_{s_1}\ A.El_{s_2}\ ((\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ x) \equiv \Pi x : El_{s_1}\ [\![A]\!].El_{s_2}\ [\![B]\!]$, by `Conv` we can get

$$[\![\Gamma]\!] \vdash_{\mathtt{DK}} \lambda x : El_{s_1}\ [\![A]\!].[\![N]\!] : \Pi x : El_{s_1}\ A.El_{s_2}\ ((\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ x)\,.$$

Therefore, we can apply `Conv` to conclude

$$\begin{aligned}&[\![\Gamma]\!] \vdash_{\mathtt{DK}} abs_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!]_{\Gamma,x:A})\ (\lambda x : El_{s_1}\ [\![A]\!].[\![N]\!])\\ &\quad : El_{s_3}\ (Prod_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!]))\end{aligned}$$ ◂

7 Conservativity and Adequacy

Many works proposing Dedukti encodings often stop after showing soundness and leave conservativity as a conjecture. This is because, when mixing the rules β with *beta*, as done in traditional Dedukti encodings, one needs to show the termination of both, given that to show conservativity one often considers terms in normal form [9]. However this problem is non-trivial, and in particular the normalization of $\beta \cup beta$ implies the termination (and thus normally also the consistency) of the encoded system. This is also unnatural, as logical frameworks should be agnostic to the fact that a system is consistent or not, and thus this shouldn't be required to show conservativity.

In this section we will show how conservativity can be shown without difficulties when we distinguish the rules β and *beta*. In particular, our proof does not need $\beta \cup beta$ to be normalizing, and thus also applies to non-normalizing and non-consistent systems.

We start by defining a notion of invertible forms and an inverse translation which allows to invert them into the original system. After proving some basic properties about them, we then proceed with the proof of conservativity.

7.1 The inverse translation

▸ **Definition 34** (Invertible forms). *We call the terms generated by the following grammar the invertible forms. The s_i are any sorts in $\mathcal{S}$, whereas the T_1, T_2 are any terms.*

$$\begin{aligned}M, N, A, B ::= {}& x \mid c \mid u_s \mid abs_{s_1,s_2}\ A\ (\lambda x : T_1.B)\ (\lambda x : T_2.M) \mid (\lambda x : T.M)\ N\\ &\mid Prod_{s_1,s_2}\ A\ (\lambda x : T_1.B) \mid app_{s_1,s_2}\ A\ (\lambda x : T_1.B)\ M\ N\end{aligned}$$

Note that this definition includes some terms which are not in β normal form. The next definition justifies the name of invertible forms: we know how to invert them.

▶ **Definition 35.** *We define the inverse translation function* $|-| : \Lambda_{\text{DK}} \to \Lambda_{EPTS}$ *on invertible forms by structural induction.*

$$|x| = x \qquad |c| = c \qquad |u_s| = s$$

$$\begin{aligned}
|(\lambda x : _.M)\ N| &= |M|\{|N|/x\} \\
|Prod_{s_1,s_2}\ A\ (\lambda x : _.B)| &= \Pi_{s_1,s_2}(|A|, [x]|B|) \\
|abs_{s_1,s_2}\ A\ (\lambda x : _.B)\ (\lambda x : _.M)| &= \lambda_{s_1,s_2}(|A|, [x]|B|, [x]|M|) \\
|app_{s_1,s_2}\ A\ (\lambda x : _.B)\ M\ N| &= @_{s_1,s_2}(|A|, [x]|B|, |M|, |N|)
\end{aligned}$$

We can show, as expected, that the terms in the image of the translation are invertible forms and that $|-|$ is a left inverse of $[\![-]\!]$. The proof is a simple induction on M.

▶ **Proposition 36.** *For all* $M \in \Lambda_{EPTS}$*,* $[\![M]\!]$ *is an invertible form and* $|[\![M]\!]| = M$*.*

The following lemma shows that invertible forms are closed under rewriting and that this rewriting can also be inverted into the EPTS.

▶ **Proposition 37.** *Let* M *be a invertible form.*

1. *If* N *is an invertible form, then* $M(N/x)$ *is also and* $|M|\{|N|/x\} = |M\{N/x\}|$*.*
2. *If* $M \hookrightarrow_{beta_{s_1,s_2}} N$ *then* N *is an invertible form and* $|M| \hookrightarrow^*_\beta |N|$
3. *If* $M \hookrightarrow_{\beta,u_{s_1}\text{-}red} N$ *then* N *is an invertible form and* $|M| = |N|$*.*
4. *If* $M \hookrightarrow^* N$ *then* N *is an invertible form and* $|M| \hookrightarrow^* |N|$*.*

Proof. **1.** By induction on M.

2. By induction on the rewrite context. For the base case, we have

$$app_{s_1,s_2}\ A_1\ (\lambda x : T_1.B_1)\ (abs_{s_1,s_2}\ A_2\ (\lambda x : T_2.B_2)\ (\lambda x : T_3.M'))\ N' \hookrightarrow (\lambda x : T_3.M')\ N'$$

whose right hand side is in the grammar. Moreover, we have

$$@_{s_1,s_2}(|A_1|, [x]|B_1|, \lambda_{s_1,s_2}(|A_2|, [x]|B_2|, [x]|M'|), |N'|) \hookrightarrow |M'|\{|N'|/x\} = |(\lambda x : T_3.M')\ N'|$$

and thus the reduction is reflected by the inverse translation.

3. By induction on the rewrite context. Note that there is no base case for u_{s_1}-red, as there is no term of the form $El_{s_2}\ u_{s_1}$ in the grammar. For the base case of β, we have $(\lambda x : T.M')\ N' \hookrightarrow M'\{N'/x\}$. Hence the resulting term is in the grammar and we have $|(\lambda x : T.M')\ N'| = |M'|\{|N'|/x\} = |M'\{N'/x\}|$ by part 1.

4. Immediate consequence of the previous parts. ◀

▶ Remark 38. Note that this last proposition explains the difference between the β and $beta_{s_1,s_2}$ steps. Whereas $beta_{s_1,s_2}$ steps represents the real computation steps that take place in the encoded system, β steps are invisible because they correspond to the framework's substitution, an administrative operation that is implicit in the encoded system. Therefore, it was expected that $beta_{s_1,s_2}$ steps would be reflected into the original system, whereas β steps would be silent.

Putting all this together, we deduce that computation and conversion in Dedukti is reflected in the encoded system.

▶ **Corollary 39** (Reflection of computation)**.** *For* $M, N \in \Lambda_{EPTS}$*, we have*

1. *If* $[\![M]\!] \hookrightarrow^* [\![N]\!]$ *then* $M \hookrightarrow^* N$*.*

2. *If* $[\![M]\!] \equiv [\![N]\!]$ *then* $M \equiv N$.

Proof. 1. Immediate consequence of Proposition 37 and Proposition 36.
2. Follows from confluence of $\beta\mathcal{R}_{\mathtt{EPTS}}$ and also Proposition 37 and Proposition 36. ◂

Note that for part 2 we really need $\beta\mathcal{R}_{EPTS}$ to be confluent. Indeed, If $[\![M]\!] \longleftrightarrow N$ then we cannot apply $|-|$ to N because it might not be an invertible form.

7.2 Conservativity

Before showing conservativity, we show the following auxiliary result, saying that every β normal term M that has type $\Pi x : A.B$ in $[\![\Gamma]\!]$ is an abstraction.

▸ **Lemma 40.** *Let* M *be in* β*-normal form. If* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M : \Pi x : A.B$ *then* $M = \lambda x : A'.N$ *with* $A' \equiv A$ *and* $[\![\Gamma]\!], x : A \vdash_{\mathtt{DK}} N : B$.

Proof. By induction on M. M cannot neither be a variable or constant, as there is no $x : C \in [\![\Gamma]\!]$ or $c[\Delta] : C \in \Sigma_{\mathtt{EPTS}}$ with $C \equiv \Pi x : A.B$. If $M = M_1 M_2$, then M_1 has a type of the form $\Pi x' : A'.B'$. By IH we get that M_1 is an abstraction, which contradicts the fact that M is in β normal form.

Therefore, M is an abstraction, of the form $M = \lambda x : A'.N$. By inversion of typing, we thus have $[\![\Gamma]\!], x : A' \vdash_{\mathtt{DK}} N : B'$ with $A' \equiv A$ and $B' \equiv B$. We can then use *Conv in context for DK* (Theorem 2) and `Conv` to derive $[\![\Gamma]\!], x : A \vdash_{\mathtt{DK}} N : B$. ◂

We are now ready to show conservativity for β normal forms. However, if we also want to show adequacy latter, we also need to show that $|-|$ is a kind of right inverse to $[\![-]\!]$. However, because the inverse translation does not capture the information in the type annotations of binders, $[\![|M|]\!] = M$ does not hold.

▸ **Example 41.** Take any invertible forms A, B and a term T with $T \neq El_{s_1}\ A$. Then the term $M = Prod_{s_1,s_2}\ A\ (\lambda x : T.B)$ is send by $|-|$ into $\Pi_{s_1,s_2}(|A|, [x]|B|)$, which is then sent by $[\![-]\!]$ into $Prod_{s_1,s_2}\ [\![|A|]\!]\ (\lambda x : El_{s_1}\ [\![|A|]\!].[\![|B|]\!])$. Therefore, even if have $[\![|B|]\!] = B$ and $[\![|A|]\!] = A$, we still have $T \neq El_{s_1}\ A$, implying $M \neq [\![|M|]\!]$. However, if M is typable, then by typing constraints we should nevertheless have $T \equiv El_{s_1}\ A$.

Therefore, while proving conservativity we will show a weaker property: for the well-typed terms we are interested in, $|-|$ is a right inverse up to the following "hidden" conversion.

▸ **Definition 42** (Hidden step). *We say that a rewriting step* $M \longrightarrow N$ *is hidden when it happens on the type annotation of a binder. More formally, we should have a rewriting context* $C(-)$ *and terms* A, A', P *such that* $A \longrightarrow A'$, $M = C(\lambda x : A.P)$ *and* $N = C(\lambda x : A'.P)$. *We denote the conversion generated by such rules by* $\equiv_H$.

We now have all ingredients to show that the encoding is conservative for β normal forms.

▸ **Theorem 43** (Conservativity of β normal forms). *Suppose* $\Gamma \vdash A$ type *and let* $M \in \Lambda_{\mathtt{DK}}$ *be a* β *normal form st* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M : T$, *with* $T = El_{s_A}\ [\![A]\!]$ *or* $T = U_A$. *Then* M *is an invertible form,* $\Gamma \vdash |M| : A$ *and* $[\![|M|]\!] \equiv_H M$.

Proof. By induction on M.

Case $M = \lambda x : A'.M'$: By inversion we have $M : \Pi x : A'_1.A'_2$ with $T \equiv \Pi x : A'_1.A'_2$. Because $\mathcal{R}_{EPTS}$ is arity preserving, this implies that T is of the form $\Pi x : A_1.A_2$, which cannot hold. Thus, this case is impossible.

Case $M = M_1\ M_2$: As M is in beta normal form, its head symbol is a constant or variable. However, there is no $c[\Delta] : C \in \Sigma$ or $x : C \in \Gamma$ with C convertible to a dependent product type. Hence, this case is impossible.

Case $M = x$: If $M = x$, by inversion of typing there is $x : El_{s_B}\ [\![B]\!] \in [\![\Gamma]\!]$ with $T \equiv El_{s_B}\ [\![B]\!]$. Therefore, we deduce $A \equiv B$ and thus we can derive $\Gamma \vdash x : A$ by applying Var with $x : B \in \Gamma$, then Conv with $A \equiv B$ and $\Gamma \vdash A\ type$.

Case $M = c[\vec{M}]$: We proceed by case analysis on c. Note that for $c = El_s\ M'$ or $c = U_s$ the resulting type is $\mathtt{TYPE}$, which is not convertible to T. Hence, these cases are impossible.

▸ Note 44. In the following, to improve readability we omit the typing hypothesis when applying $\mathtt{Conv}$. However, all such uses can be justified.

Case $c = u_{s_1}$: As we have $[\![\Gamma]\!] \vdash_{\mathtt{DK}} u_{s_1} : U_{s_2}$, by uniqueness of types we have $T \equiv U_{s_2}$, and therefore we get $A \equiv s_2$. We can thus deduce $\Gamma \vdash s_1 : A$ by using Conv with $\Gamma \vdash s_1 : s_2$.

Case $c = Prod_{s_1,s_2}$: By inversion of typing, we have

1. $\vec{M} = M_1\ M_2$
2. $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_1 : U_{s_1}$
3. $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_2 : El_{s_1}\ M_1 \to U_{s_2}$
4. $T \equiv U_{s_3}$

As M_1 is in β normal form, by IH M_1 is an invertible form, $\Gamma \vdash |M_1| : s_1$ and $[\![|M_1|]\!] \equiv_H M_1$.

By Lemma 40 applied to 2, we get $M_2 = \lambda x : B.N$ and $B \equiv El_{s_1}\ M_2$ with $[\![\Gamma]\!], x : El_{s_1}\ M_1 \vdash N : U_{s_2}$. Because $M_1 \equiv [\![|M_1|]\!]$, we have $[\![\Gamma]\!], x : El_{s_1}\ [\![|M_1|]\!] \vdash N : U_{s_2}$. As $\Gamma \vdash |M_1| : s_1$ we have $\Gamma, x : |M_1|$ well-formed and thus by IH N is an invertible form and we have $[\![|N|]\!] \equiv_H N$ and $\Gamma, x : |M_1| \vdash |N| : s_2$.

Therefore, by Prod we have $\Gamma \vdash \Pi_{s_1,s_2}(|M_1|, [x]|N|) : s_3$, and then by Conv with $A \equiv s_3$ we conclude $\Gamma \vdash \Pi_{s_1,s_2}(|M_1|, [x]|N|) : A$. Finally, as $M_1 \equiv_H [\![|M_1|]\!]$, $N \equiv_H [\![|N|]\!]$ and $B \equiv El_{s_1}\ M_1 \equiv El_{s_1}\ [\![|M_1|]\!]$, we conclude

$$\begin{aligned} M &= Prod_{s_1,s_2}\ M_1\ M_2 = Prod_{s_1,s_2}\ M_1\ (\lambda x : B.N) \\ &\equiv_H Prod_{s_1,s_2}[\![|M_1|]\!]\ (\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!]) = [\![\Pi_{s_1,s_2}(|M_1|, [x]|N|)]\!] = [\![|M|]\!] \end{aligned}$$

Case $c = abs_{s_1,s_2}$: By inversion of typing, we have

1. $\vec{M} = M_1\ M_2\ M_3$
2. $[\![\Gamma]\!] \vdash M_1 : U_{s_1}$
3. $[\![\Gamma]\!] \vdash M_2 : El_{s_1}\ M_2 \to U_{s_2}$
4. $[\![\Gamma]\!] \vdash M_3 : \Pi x : El_{s_1}\ M_1.El_{s_2}\ (M_2\ x)$
5. $T \equiv El_{s_3}\ (Prod_{s_1,s_2}\ M_1\ M_2)$

By the same arguments as in case $M = Prod_{s_1,s_2}\ \vec{M}$, we have that

- M_1 is an invertible form, $[\![|M_1|]\!] \equiv_H M_1$ and $\Gamma \vdash |M_1| : s_1$.
- $M_2 = \lambda x : B.N$, N is an invertible form, $[\![|N|]\!] \equiv_H N$, $\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!] \equiv_H \lambda x : B.N$ and $\Gamma, x : |M_1| \vdash |N| : s_2$.

By Lemma 40 applied to 4 we have $M_3 = \lambda x : C.P$, $C \equiv El_{s_1}\ M_1$ and $[\![\Gamma]\!], x : El_{s_1}\ M_1 \vdash_{\mathtt{DK}} P : El_{s_2}\ (M_2\ x)$. Using $M_2 = \lambda x : B.N$ and *Reduce type in judgement* (Proposition 3), we get $[\![\Gamma]\!], x : El_{s_1}\ M_1 \vdash_{\mathtt{DK}} P : El_{s_2}\ N$. Because $M_1 \equiv [\![|M_1|]\!]$ and $N \equiv [\![|N|]\!]$, we then get $[\![\Gamma]\!], x : El_{s_1}\ [\![|M_1|]\!] \vdash_{\mathtt{DK}} P : El_{s_2}\ [\![|N|]\!]$.

Therefore, by IH P is an invertible form, $\Gamma, x : |M_1| \vdash |P| : |N|$ and $[\![|P|]\!] \equiv_H P$. Putting this together with $\Gamma \vdash |M_1| : s_1$ and $\Gamma, x : |M_1| \vdash |N| : s_2$ we can derive $\Gamma \vdash$

$\lambda_{s_1,s_2}(|M_1|, [x]|N|, [x]|P|) : \Pi_{s_1,s_2}(|M_1|, [x]|N|)$. From 5 we can also show $\Pi_{s_1,s_2}(|M_1|, [x]|N|) \equiv A$, which allows us to apply Conv to get $\Gamma \vdash \lambda_{s_1,s_2}(M_1, [x]|N|, [x]|P|) : A$. Finally, from $[\![|M_1|]\!] \equiv_H M_1$, $\lambda x : B.N \equiv_H \lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!]$, $P \equiv_H [\![|P|]\!]$ and $C \equiv El_{s_1}\ M_1 \equiv El_{s_1}\ [\![|M_1|]\!]$ we get

$$\begin{aligned} M &= abs_{s_1,s_2,}\ M_1\ M_2\ M_3 = abs_{s_1,s_2,}\ M_1\ (\lambda x : B.N)\ (\lambda x : C.P) \\ &\equiv_H abs_{s_1,s_2,}\ [\![|M_1|]\!]\ (\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!])\ (\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|P|]\!]) \\ &= [\![\lambda_{s_1,s_2}(|M_1|, [x]|N|, [x]|P|)]\!] = [\![|M|]\!] \end{aligned}$$

Case $c = app_{s_1,s_2}$: By inversion of typing, we have

1. $\vec{M} = M_1\ M_2\ M_3\ M_4\ M_5$
2. $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_1 : U_{s_1}$
3. $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_2 : El_{s_1}\ M_1 \to U_{s_2}$
4. $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_3 : El_{s_3}\ (Prod_{s_1,s_2}\ M_1\ M_2)$
5. $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_4 : El_{s_1}\ M_1$
6. $T \equiv El_{s_2}\ (M_2\ M_4)$

By the same arguments as in case $M = Prod_{s_1,s_2}\ \vec{M}$, we have that

- M_1 is an invertible form, $[\![|M_1|]\!] \equiv_H M_1$ and $\Gamma \vdash |M_1| : s_1$.
- $M_2 = \lambda x : B.N$, N is an invertible form, $[\![|N|]\!] \equiv_H N$, $\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!] \equiv_H \lambda x : B.N$ and $\Gamma, x : |M_1| \vdash |N| : s_2$.

As $Prod_{s_1,s_2}\ M_1\ M_2 \equiv [\![\Pi_{s_1,s_2}(|M_1|, [x]|M_2|)]\!]$, from 4 we get $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_3 : El_{s_3}\ [\![\Pi_{s_1,s_2}(|M_1|, [x]|M_2|)]\!]$. Therefore, we deduce by the IH that M_3 is an invertible form, $\Gamma \vdash |M_3| : \Pi_{s_1,s_2}(|M_1|, [x]|N|)$ and $[\![|M_3|]\!] \equiv_H M_3$.

Moreover, as $M_1 \equiv [\![|M_1|]\!]$, from 5 we get $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M_4 : El_{s_1}\ [\![|M_1|]\!]$, therefore by IH we deduce that M_4 is an invertible form, $\Gamma \vdash |M_4| : |M_1|$ and $[\![|M_4|]\!] \equiv_H M_4$.

Putting together $\Gamma \vdash |M_1| : s_1$, $\Gamma, x : |M_1| \vdash |N| : s_2$, $\Gamma \vdash |M_3| : \Pi_{s_1,s_2}(|M_1|, [x]|N|)$ and $\Gamma \vdash |M_4| : |M_1|$ we derive $\Gamma \vdash @_{s_1,s_2}(|M_1|, [x]|N|, |M_3|, |M_4|) : |N|(|M_4|/x)$.

From 8 we get $T \equiv El_{s_2}\ (M_2\ M_4) \equiv El_{s_2}\ ((\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!])\ [\![|M_4|]\!]) \equiv El_{s_2}\ [\![|N|\{|M_4|/x\}]\!]$, thus we deduce $A \equiv |N|\{|M_4|/x\}$. Hence, we can apply Conv to get $\Gamma \vdash @_{s_1,s_2}(|M_1|, [x]|N|, |M_3|, |M_4|) : A$.

From $[\![|M_1|]\!] \equiv_H M_1$, $\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!] \equiv_H \lambda x : B.N$, $[\![|M_4|]\!] \equiv_H M_4$ and $[\![|M_3|]\!] \equiv_H M_3$ we can then conclude

$$\begin{aligned} M &= app_{s_1,s_2}\ M_1\ M_2\ M_3\ M_4 = app_{s_1,s_2}\ M_1\ (\lambda x : B.N)\ M_3\ M_4 \\ &\equiv_H app_{s_1,s_2}\ [\![|M_1|]\!]\ (\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!])\ [\![|M_3|]\!]\ [\![|M_4|]\!] \\ &= [\![@_{s_1,s_2}(|M_1|, [x]|N|, |M_3|, |M_4|)]\!] = [\![|M|]\!] \end{aligned}$$

◂

By *Basic properties* (Proposition 29), β is strongly normalizing and type preserving. Therefore from the previous result we can immediately get full conservativity.

▸ **Theorem 45** (Conservativity). *Let* $\Gamma \vdash A$ *type,* $M \in \Lambda_{\mathtt{DK}}$ *such that* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M : T$*, with* $T = El_{s_A}\ [\![A]\!]$ *or* $T = U_A$*. We have* $\Gamma \vdash |NF_\beta(M)| : A$ *and* $M \longrightarrow_\beta^* NF_\beta(M) \equiv_H [\![|NF_\beta(M)|]\!]$*.*

Note that this also gives us a straightforward algorithm to invert terms: it suffices to normalize with β and then apply $|-|$.

7.3 Adequacy

If we write $\Lambda(\Gamma \vdash_{EPTS} \_ : A)$ for the set of $M \in \Lambda_{EPTS}$ st $\Gamma \vdash M : A$ and $\Lambda_{NF}(\Gamma \vdash_{\mathrm{DK}} \_ : T)$ for the set of $M \in \Lambda_{\mathrm{DK}}$ in β normal form st $\Gamma \vdash_{\mathrm{DK}} M : T$, we can show our adequacy theorem. This result follows by simply putting together *Basic properties* (Proposition 29), *Preservation of computation* (Lemma 31), *Soundness* (Theorem 33), *Reflection of computation* (Corollary 39) and *Conservativity* (Theorem 45).

▸ **Theorem 46** (Computational adequacy). *For A, Γ with $\Gamma \vdash A$ type, let $T = U_A$ if A is a top sort, otherwise $T = El_{s_A}$ $[\![A]\!]$. We have a bijection*

$$\Lambda(\Gamma \vdash_{EPTS} \_ : A) \simeq \Lambda_{NF}([\![\Gamma]\!] \vdash_{\mathrm{DK}} \_ : T)/\equiv_H$$

given by $[\![-]\!]$ and $|-|$. It is compositional in the sense that $[\![-]\!]$ commutes with substitution. It is computational in the sense that $M \hookrightarrow^* N$ iff $[\![M]\!] \hookrightarrow^* [\![N]\!]$. Moreover, any M satisfying $[\![\Gamma]\!] \vdash_{\mathrm{DK}} M : T$ has such a β normal form.

8 Representing systems with infinite sorts

We have presented an encoding of EPTSs in Dedukti that is sound, conservative and adequate. However when using it in practice with Dedukti implementations we run into problems when representing systems with infinite sorts, such as in Martin-Löf's Type Theory or the Extended Calculus of Constructions. Indeed, in this case our encoding needs an infinite number of constant and rule declarations, which cannot be made in practice.

One possible solution is to approximate the infinite sort structure by a finite one. Most proofs in infinite sort systems probably do not use more then 10 sorts, therefore we could truncate the representation while still providing a good encoding of the system.

A different approach proposed in [1][23] is to internalize the indices of $Prod_{s_1,s_2}, El_{s_1}, \ldots$ and represent them inside Dedukti. In order to apply this method, we chose to stick with systems in which $\mathcal{A}, \mathcal{R}$ are total functions $\mathcal{S} \to \mathcal{S}$ and $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$ respectively. Note that this is true for almost all infinite sort systems used in practice, and this will greatly simplify our presentation.

We can now declare a constant $\widehat{\mathcal{S}}$ to represent the type of sorts in $\mathcal{S}$ and two constants $\widehat{\mathcal{A}}, \widehat{\mathcal{R}}$ to represent the functions $\mathcal{A}, \mathcal{R}$. Then, each of our previously declared families of constants now becomes a single one, by taking arguments of type $\widehat{\mathcal{S}}$. The same happens with the rewrite rules. This leads to the theory presented in Figure 3, which we call $(\Sigma^{\mathcal{S}}_{\mathrm{EPTS}}, \mathcal{R}^{\mathcal{S}}_{\mathrm{EPTS}})$.

This theory needs of course to be completed case by case, so that $\widehat{\mathcal{S}}, \widehat{\mathcal{A}}, \widehat{\mathcal{R}}$ correctly represent $\mathcal{S}, \mathcal{A}, \mathcal{R}$. For this to hold, each sort $s \in \mathcal{S}$ should have a representation $\dot{s} : \widehat{\mathcal{S}}$, and this should restrict to a bijection when considering only the closed normal forms of type $\widehat{\mathcal{S}}$. Moreover, we should add rewrite rules such that $\mathcal{A}(s_1) = s_2$ iff $\widehat{\mathcal{A}}\ \dot{s}_1 \equiv \dot{s}_2$ and $\mathcal{R}(s_1, s_2) = s_3$ iff $\widehat{\mathcal{R}}\ \dot{s}_1\ \dot{s}_2 \equiv \dot{s}_3$.

In order to understand intuitively these conditions, let's look at an example.

▸ **Example 47.** The sort structure of Martin-Löf's Type Theory is given by the specification $\mathcal{S} = \mathbb{N}$, $\mathcal{A}(x) = x + 1$ and $\mathcal{R}(x, y) = max\{x, y\}$. We can represent this in Dedukti by declaring constants $z : \widehat{\mathcal{S}}$, $s[n : \widehat{\mathcal{S}}] : \widehat{\mathcal{S}}$ and rewrite rules $\widehat{\mathcal{A}}\ x \hookrightarrow s\ x$, $\widehat{\mathcal{R}}\ z\ x \hookrightarrow x$, $\widehat{\mathcal{R}}\ x\ z \hookrightarrow x$ and $\widehat{\mathcal{R}}\ (s\ x)\ (s\ y) \hookrightarrow s\ (\widehat{\mathcal{R}}\ x\ y)$. Then the closed normal forms of type $\widehat{\mathcal{S}}$ are in bijection with $\mathcal{S}$, and we have $\mathcal{A}(s_1) = s_2$ iff $\widehat{\mathcal{A}}\ s_1 \equiv s_2$ and $\mathcal{R}(s_1, s_2) = s_3$ iff $\widehat{\mathcal{R}}\ s_1\ s_2 \equiv s_3$.

With this representation, we can revisit the example of the polymorphic identity function.

$$\widehat{\mathcal{S}} : \mathtt{TYPE}$$
$$\widehat{\mathcal{A}}[s_1 : \widehat{\mathcal{S}}] : \widehat{\mathcal{S}}$$
$$\widehat{\mathcal{R}}[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}] : \widehat{\mathcal{S}}$$

$$U[s : \widehat{\mathcal{S}}] : \mathtt{TYPE}$$
$$El[s : \widehat{\mathcal{S}}; A : U\ s] : \mathtt{TYPE}$$
$$u[s : \widehat{\mathcal{S}}] : U\ (\widehat{\mathcal{A}}\ s)$$
$$El\ s'\ (u\ s) \longleftrightarrow_{u\text{-red}} U\ s$$

$$Prod[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}; A : U\ s_1; B : El\ s_1\ A \to U\ s_2] : U\ (\widehat{\mathcal{R}}\ s_1\ s_2)$$
$$abs[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}; A : U\ s_1; B : El\ s_1\ A \to U\ s_2; N : \Pi x : El\ s_1\ A.El\ s_2\ (B\ x)]$$
$$: El\ (\widehat{\mathcal{R}}\ s_1\ s_2)\ (Prod\ s_1\ s_2\ A\ B)$$
$$app[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}; A : U\ s_1; B : El\ s_1\ A \to U\ s_2; M : El\ (\widehat{\mathcal{R}}\ s_1\ s_2)\ (Prod\ s_1\ s_2\ A\ B); N : El\ s_1\ A]$$
$$: El\ s_2\ (B\ N)$$
$$app\ s_1\ s_2\ A\ B\ (abs\ s_1'\ s_2'\ A'\ B'\ M)\ N \longleftrightarrow_{beta} M\ N$$

Figure 3 Definition of the theory $(\Sigma^S_{\mathrm{EPTS}}, \mathcal{R}^S_{\mathrm{EPTS}})$

▸ **Example 48.** The (predicative and at sort 0) polymorphic identity function in Martin Löf's Type Theory is given by the term

$$\lambda_{1,0}(0, {}_{[A]}\Pi_{0,0}(A, {}_{[x]}A), {}_{[A]}\lambda_{0,0}(A, {}_{[x]}A, {}_{[x]}x))\,.$$

It can be represented in the encoding by

$$abs\ (s\ z)\ z\ (u\ z)\ (\lambda A.Prod\ z\ z\ A\ (\lambda x.A))\ (\lambda A.abs\ z\ z\ A\ (\lambda x.A)\ (\lambda x.x))\,.$$

Let us now define the encoding function formally, by the following equations.

$$[\![x]\!]_S = x$$
$$[\![s]\!]_S = u\ \dot{s}$$
$$[\![\Pi_{s_1,s_2}(A, {}_{[x]}B)]\!]_S = Prod\ \dot{s}_1\ \dot{s}_2\ [\![A]\!]\ (\lambda x : El\ \dot{s}_1\ [\![A]\!]_S.[\![B]\!]_S)$$
$$[\![\lambda_{s_1,s_2}(A, {}_{[x]}B, {}_{[x]}M)]\!]_S = abs\ \dot{s}_1\ \dot{s}_2\ [\![A]\!]_S\ (\lambda x : El\ \dot{s}_1\ [\![A]\!]_S.[\![B]\!]_S)\ (\lambda x : El\ \dot{s}_1\ [\![A]\!]_S.[\![M]\!]_S)$$
$$[\![@_{s_1,s_2}(A, {}_{[x]}B, M, N)]\!]_S = app\ \dot{s}_1\ \dot{s}_2\ [\![A]\!]_S\ (\lambda x : El\ \dot{s}_1\ [\![A]\!]_S.[\![B]\!]_S)\ [\![M]\!]_S\ [\![N]\!]_S$$
$$[\![-]\!]_S = -$$
$$[\![\Gamma, x : A]\!]_S = [\![\Gamma]\!]_S, x : El\ \dot{s}_A\ [\![A]\!]_S \quad \text{where } \Gamma \vdash A : s_A$$

Now one can proceed as before with the proofs of soundness, conservativity and adequacy, which follow the same idea as the previously presented ones. However, it is quite unsatisfying that we have to redo all the work of Sections 6 and 7 another time, and therefore one can wonder if we can reuse the results we already have about the first encoding.

Note that one may intuitively think of the $(\Sigma^S_{\mathrm{EPTS}}, \mathcal{R}^S_{\mathrm{EPTS}})$ as a "hidden implementation" of $(\Sigma_{\mathrm{EPTS}}, \mathcal{R}_{\mathrm{EPTS}})$. In this case, it should be possible to take a proof written in the $(\Sigma_{\mathrm{EPTS}}, \mathcal{R}_{\mathrm{EPTS}})$ and "implement" it in the $(\Sigma^S_{\mathrm{EPTS}}, \mathcal{R}^S_{\mathrm{EPTS}})$. To formalize this intuition, we will define a notion of *theory morphism* which will allows us to establish the soudness of this new encoding using a morphism from $(\Sigma_{\mathrm{EPTS}}, \mathcal{R}_{\mathrm{EPTS}})$ to $(\Sigma^S_{\mathrm{EPTS}}, \mathcal{R}^S_{\mathrm{EPTS}})$.

9 Theory morphisms

To define our notion of theory morphism, we start by defining an auxiliary weaker notion of pre-morphism. In the following, we write $\mathcal{C}(\Sigma_i)$ for the constants appearing in Σ_i and $\Lambda(\Sigma_i)$

for the terms built using such constants.

▸ **Definition 49** (Theory pre-morphism). *A theory pre-morphism $F : (\Sigma_1, \mathcal{R}_1) \to (\Sigma_2, \mathcal{R}_2)$ is for each $c \in \mathcal{C}(\Sigma_1)$ a term $F_c \in \Lambda(\Sigma_2)$ with free variables in Δ_c. Each such F defines a map on terms $|-|_F$ given by*

$$|c[\vec{M}]|_F = F_c\{|\vec{M}|_F\} \qquad |\Pi x : A.B|_F = \Pi x : |A|_F.|B|_F$$
$$|x|_F = x \qquad |\lambda x : A.M|_F = \lambda x : |A|_F.|M|_F$$
$$|\mathtt{TYPE}|_F = \mathtt{TYPE} \qquad |MN|_F = |M|_F|N|_F$$
$$|\mathtt{KIND}|_F = \mathtt{KIND}$$

Given a term $c[\vec{M}]$ defined in the signature Σ_1, one should understand $F_c\{|\vec{M}|_F\}$ as the implementation in Σ_2 of this term. With this interpretation, we can see F_c as the body of the implementation. This also explains why F_c should have free variables in Δ_c, as these corresponds to the arguments that are supplied to c.

In order to understand intuitively the definition, let's define a theory pre-morphism from $(\Sigma_{\mathtt{EPTS}}, \mathcal{R}_{\mathtt{EPTS}})$ to $(\Sigma^S_{\mathtt{EPTS}}, \mathcal{R}^S_{\mathtt{EPTS}})$, which will then be used to show soundness of $[\![-]\!]_S$.

▸ **Example 50.** We define the pre-morphism $\phi : (\Sigma_{\mathtt{EPTS}}, \mathcal{R}_{\mathtt{EPTS}}) \to (\Sigma^S_{\mathtt{EPTS}}, \mathcal{R}^S_{\mathtt{EPTS}})$ by the following data. We recall in the right the variables in the context of each constant (we write $\mathcal{V}(\Delta)$ for the variables in Δ).

$$\phi_{U_s} = U\ \dot{s} \qquad \mathcal{V}(\Delta_{U_s}) = -$$
$$\phi_{El_s} = El\ \dot{s}\ A \qquad \mathcal{V}(\Delta_{El_s}) = A$$
$$\phi_{u_s} = u\ \dot{s} \qquad \mathcal{V}(\Delta_{u_s}) = -$$
$$\phi_{Prod_{s_1,s_2}} = Prod\ \dot{s_1}\ \dot{s_2}\ A\ B \qquad \mathcal{V}(\Delta_{Prod_{s_1,s_2}}) = A, B$$
$$\phi_{abs_{s_1,s_2}} = abs\ \dot{s_1}\ \dot{s_2}\ A\ B\ N \qquad \mathcal{V}(\Delta_{abs_{s_1,s_2}}) = A, B, N$$
$$\phi_{app_{s_1,s_2}} = app\ \dot{s_1}\ \dot{s_2}\ A\ B\ M\ N \qquad \mathcal{V}(\Delta_{app_{s_1,s_2}}) = A, B, M, N$$

We can then calculate for instance that value of $|Prod_{s_1,s_2}\ T_1\ T_2|_\phi$ as

$$|Prod_{s_1,s_2}\ T_1\ T_2\ |_\phi = (Prod\ \dot{s_1}\ \dot{s_2}\ A\ B)\{|T_1|_\phi/A, |T_2|_\phi/B\} = Prod\ \dot{s_1}\ \dot{s_2}\ |T_1|_\phi\ |T_2|_\phi$$

More generally, we can prove that $|[\![M]\!]|_\phi = [\![M]\!]_S$ by induction on $M \in \Lambda_{EPTS}$.

Not every theory pre-morphism should be called a morphism, as there are some properties which one should enforce. In the following, we write $\vdash_i$ for a judgment in the theory $(\Sigma_i, \mathcal{R}_i)$.

▸ **Definition 51** (Theory morphism). *A theory morphism $F : (\Sigma_1, \mathcal{R}_1) \to (\Sigma_2, \mathcal{R}_2)$ is a theory pre-morphism satisfying the following conditions*
1. *for all $c[A_c] : \Delta_c \in \Sigma_1$, we have $|\Delta_c|_F \vdash_2 F_c : |A_c|_F$*
2. *for all $l \hookrightarrow_1 r \in \mathcal{R}_1$ we have $|l|_F \hookrightarrow^*_2 |r|_F$*

We have the following basic properties about compositionality and preservation of computation and of conversion.

▸ **Lemma 52.** *For each morphism F, we have the following properties.*
1. *Compositionality: $|M|_F\{|N|_F/x\} = |M\{N/x\}|_F$*
2. *Preservation of computation: if $M \hookrightarrow_1 N$ then $|M|_F \hookrightarrow^*_2 |N|_F$*
3. *Preservation of conversion: if $M \equiv_1 N$ then $|M|_F \equiv_2 |N|_F$*

We can now show the main result about theory morphisms.

▸ **Theorem 53** (Preservation of typing). *Let $F : (\Sigma_1, \mathcal{R}_1) \to (\Sigma_2, \mathcal{R}_2)$ be a theory morphism.*

1. *If* Γ $\texttt{well-formed}_1$ *then* $\vdash_2 |\Gamma|_F$ $\texttt{well-formed}_2$
2. *If* $\Gamma \vdash_1 M : A$ *then* $|\Gamma|_F \vdash_2 |M|_F : |A|_F$

Proof. By induction on the judgment tree. We do only cases `Conv` and `Cons`, as they are the only interesting ones.

Case Cons: The proof ends with

$$c[\Delta_c] : A_c \in \Sigma_1 \ \frac{\Delta_c \vdash_1 A_c : s \qquad \Gamma \vdash_1 \vec{M} : \Delta_c}{\Gamma \vdash_1 c[\vec{M}] : A_c\{\vec{M}\}} \ \texttt{Cons}$$

By IH we have $|\Gamma| \vdash_2 |\vec{M}| : |\Delta_c|$. Moreover, because F is a morphism we have $|\Delta_c| \vdash_2 F_c : |A_c|$. By substitution we thus deduce $|\Gamma| \vdash_2 F_c\{|\vec{M}|\} : |A_c|\{|\vec{M}|\}$. Finally, as $|A_c|\{|\vec{M}|\} = |A_c\{\vec{M}\}|$ we get the result.

Case Conv: The proof ends with

$$A \equiv_1 B \ \frac{\Gamma \vdash_1 M : A \qquad \Gamma \vdash_1 B : s}{\Gamma \vdash_1 M : B} \ \texttt{Conv}$$

By the IH, we have $|\Gamma| \vdash_2 |M| : |A|$ and $|\Gamma| \vdash_2 |B| : s$. Moreover, as $A \equiv_1 B$, by Lemma 52 we have $|A| \equiv_2 |B|$, and thus we can apply `Conv` to conclude. ◂

Using this result, one can also show, as expected, that theories and their morphisms assemble into a category. However, as we will not need this result here, we will not show it. Instead, let's now come back to our pre-morphism ϕ and show that it is indeed a morphism.

▸ **Example 54.** We show that ϕ verifies the conditions of Definition 51, and is thus a morphism. Condition 2 can be easily verified, so we concentrate in the first one. As an example, we show the propertiy only for constant $Prod_{s_1,s_2}$. We need to show

$$A : |U_{s_1}|_\phi, B : |El_{s_1}\ A|_\phi \vdash U\ \dot{s_1}\ \dot{s_2}\ A\ B : |U_{s_3}|$$

where $(s_1, s_2, s_3) \in \mathcal{R}$. Because $|U_{s_1}|_\phi = U\ \dot{s_1}$ and $|El_{s_1}\ A|_\phi = El\ \dot{s_1}\ A$ we can show, using rule `Cons`, that $A : U\ \dot{s_1}, B : El\ \dot{s_1}\ A \vdash U\ \dot{s_1}\ \dot{s_2}\ A\ B : U\ (\widehat{\mathcal{R}}\ \dot{s_1}\ \dot{s_2})$. However, as we have $\widehat{\mathcal{R}}\ \dot{s_1}\ \dot{s_2} \equiv \dot{s_3}$ and $U\ \dot{s_3} : \texttt{TYPE}$, using `Conv` we deduce the required result.

We can now use this to show that $[\![-]\!]_S$ is sound. Indeed, because $[\![-]\!]$ is sound and we have $|[\![M]\!]|_\phi = [\![M]\!]_S$, by Theorem 53 we immediately get the following result.

▸ **Corollary 55** ($[\![-]\!]_S$ is sound). *Let Γ be a context and M, A terms in an EPTS. We have*

- *If* Γ well-formed *then* $[\![\Gamma]\!]_S$ $\texttt{well-formed}_2$
- *If* $\Gamma \vdash M : A$ *then*
  - *if $A = s$ is a top-sort then* $[\![\Gamma]\!]_S \vdash_2 [\![M]\!]_S : U\ \dot{s}$
  - *else* $[\![\Gamma]\!]_S \vdash_2 [\![M]\!]_S : El\ \dot{s_A}\ [\![A]\!]_S$*, where* $\Gamma \vdash A : s_A$

In a sense, our notion of theory morphism allows us to embed a theory that is more fined grained into a theory that is less. For instance, to build our morphism ϕ, we map all the constants of the form $Prod_{s_1,s_2}$ to the same one. We then could try to build an inverse morphism ϕ^{-1} to show conservativity of $[\![-]\!]_S$, but this is not possible with our definition. Indeed, the same constant $Prod$ should be sent into $Prod_{s_1,s_2}$ when it is applied to $\dot{s_1}, \dot{s_2}$ and into into $Prod_{s_3,s_4}$ when it is applied to $\dot{s_3}, \dot{s_4}$. However, in our definition the body of the implementation F_c only depends on the initial constant c, and not on its arguments $\vec{M}$.

Therefore, it is still an open problem for us to find a notion of morphism that would allows to build morphisms in both directions between $(\Sigma_{\text{EPTS}}, \mathcal{R}_{\text{EPTS}})$ and $(\Sigma^{S}_{\text{EPTS}}, \mathcal{R}^{S}_{\text{EPTS}})$, and then show the equivalence between the encodings. For the time being, in order to show conservativity of $[\![-]\!]_S$ one unfortunately has to redo the work of Section 7, which is doable but unsatisfying.

We note nevertheless that our definition of morphism can have many other applications. For instance, if we consider two DEDUKTI theories that express classical logic, one using the axiom of the excluded middle $A \vee \neg A$ and the other using the double negation axiom $A \Leftrightarrow \neg\neg A$, one could define morphisms in both directions in order to be able to transport proofs from a theory to another. It would suffice to map the constant representing the excluded middle exm to a proof of it F_{exm} which uses the double negation, and map the constant representing the double negation axiom $nnpp$ to a proof of it F_{nnpp} which uses the excluded middle.

10 The encoding in practice

Our encoding satisfies very nice theoretical properties, but when using it in practice it becomes quite annoying to have to explicit all the information needed in app_{s_1,s_2} and abs_{s_1,s_2}. Worst, when performing translations from other systems where those parameters are not explicit we would then have to compute them during the translation. Thankfully, LAMBDAPI[10], an implementation of DEDUKTI, allows us to solve this by declaring some arguments as implicit, so they are only calculated internally.

Using the encoding of Figure 3 we can mark for instance the arguments s_1, s_2, A of $Prod$ as implicit. We can then also rename $Prod$ into Π', abs into λ', app into $\blacksquare$ and use another LAMBDAPI feature allowing to mark Π', λ' as quantifier and $\blacksquare$ as infix left. This then allows us to represent $\Pi x : A.B$ as $\Pi' x : El\ [\![A]\!].[\![B]\!]$, $\lambda x : A.B$ as $\lambda' x : El\ [\![A]\!].[\![B]\!]$ and $M\ N$ as $[\![M]\!] \blacksquare [\![N]\!]$. Using these notations, we can write terms in the encoding in a very natural way, and we refer to [6] for a set of examples of this.

However, as DEDUKTI also aims to be used in practice for sharing real libraries between proof assistants, we also tested how our approach copes with more practical scenarios. We provide in [7] a benchmark of Fermat's little theorem library in DEDUKTI[22], where we compare the traditional encoding with an adequate version that applies the ideas of our approach.[8] As we can see, the move from the traditional to the adequate version introduces a considerable performance hit. The standard DEDUKTI implementation, which is our reference here, takes 17 times more time to typecheck the files. This is probably caused by the insertion of type parameters A and B in abs_{s_A,s_B} and app_{s_A,s_B}, which are not needed in traditional encodings.

Nevertheless, DEDUKTI is still able to typecheck our encoding in a very reasonable time, showing that our approach is indeed usable in practical scenarios, even if it is not the most performing one. Moreover, as our encoding is mainly intended to be used to check proofs, and not with interactive proof development, immediacy of the result is not essential and thus it can be reasonable to trade performance for better theoretical properties. Still, we plan in the future to look at techniques to improve our performances. In particular, DEDUKTI does

[6] https://github.com/thiagofelicissimo/examples-encodigs
[7] https://github.com/thiagofelicissimo/encoding-benchmarking
[8] Because the underlying logic of the library is not a PTS, this encoding is not exactly the one we present here. However, it uses the same ideas discussed, and the same proof strategy to show adequacy applies.

not implement sharing, and doing so would probably improve drastically our performance, as the parameter annotations in app_{s_A,s_B} and abs_{s_A,s_B} carry a lot of repetition.

11 Conclusion

By separating the framework's abstraction and application from the ones of the encoded system, we have proposed a new paradigm for Dedukti encodings. Our approach offers much more well-behaved encodings, whose conservativity can be showed in a much more straightforward way and which feature adequacy theorems, something that was missing from traditional Dedukti encodings. However, differently from the ELF approach, our encoding is also computational. Therefore, our method combines the adequacy of ELF encodings with the computational aspect of Dedukti encodings.

By decoupling the framework's β from the rewriting of the encoded system, our approach allows to show the expected properties of the encoding without requiring to show that the encoded system terminates. Indeed, our adequacy result concerns all functional EPTS, even non terminating ones, such as the one with $\mathit{Type} : \mathit{Type}$. This sets our work apart from [9], whose conservativity proof requires the encoded system to be normalizing.

This work opens many other directions we would like to explore. We believe that our technique can be extended to craft adequate and computational encodings of type theories with much more complex features, such as (co)inductive types, universe polymorphism, predicate subtyping and others. For instance, in the case of inductive types no type-level rewriting rules need to be added, thus *β is SN in Dedukti* (Theorem 18) would apply. Therefore, we could repeat the same technique of normalizing only with β to show conservativity.

However, we would be particularly interested to see if we could take a general definition of type theories covering most of these features (maybe in the lines of [5]). This would allow us to define a single encoding which could be applied to encode various features, and thus would saves us from redoing similar proofs multiple times.

References

1. Ali Assaf. *A framework for defining computational higher-order logics*. Theses, École polytechnique, September 2015. URL: `https://pastel.archives-ouvertes.fr/tel-01235303`.
2. Ali Assaf. Conservativity of embeddings in the lambda pi calculus modulo rewriting. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
3. Ali Assaf, Guillaume Burel, Raphaël Cauderlier, D Delahaye, G Dowek, C Dubois, F Gilbert, P Halmagrand, O Hermant, and R Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory. Manuscript, 2016.
4. Gilles Barthe. The relevance of proof-irrelevance. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 755–768, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
5. Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories, 2020. `arXiv:2009.05539`.
6. Frédéric Blanqui. *Théorie des types et réécriture. (Type theory and rewriting)*. PhD thesis, University of Paris-Sud, Orsay, France, 2001. URL: `https://tel.archives-ouvertes.fr/tel-00105522`.
7. Frédéric Blanqui. Type safety of rewrite rules in dependent types. In *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*, volume 167, page 14, Paris, France, June 2020. URL: `https://hal.inria.fr/hal-02981528`, `doi:10.4230/LIPIcs.FSCD.2020.13`.

8 Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 9:1–9:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.9.

9 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.

10 Deducteam. Lambdapi. https://github.com/Deducteam/lambdapi.

11 Gilles Dowek. Models and termination of proof reduction in the lambda pi-calculus modulo theory. In *ICALP*, 2017.

12 Thiago Felicissimo. No need to be implicit!

13 Gaspard Ferey. *Higher-Order Confluence and Universe Embedding in the Logical Framework*. Theses, Université Paris-Saclay, June 2021. URL: https://tel.archives-ouvertes.fr/tel-03418761.

14 Guillaume Genestier. *Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting*. PhD thesis, 2020. Thèse de doctorat dirigée par Blanqui, Frédéric et Hermant, Olivier Informatique université Paris-Saclay 2020. URL: http://www.theses.fr/2020UPASG045.

15 Jan Herman Geuvers. *Logics and type systems*. [Sl: sn], 1993.

16 Robert Harper. An equational logical framework for type theories. *arXiv preprint arXiv:2106.01484*, 2021.

17 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. doi:10.1145/138027.138060.

18 Gabriel Hondet and Frédéric Blanqui. Encoding of Predicate Subtyping with Proof Irrelevance in the λΠ-Calculus Modulo Theory. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13885, doi:10.4230/LIPIcs.TYPES.2020.6.

19 Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1):279–308, 1993. URL: https://www.sciencedirect.com/science/article/pii/0304397593900917, doi:https://doi.org/10.1016/0304-3975(93)90091-7.

20 Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs*, pages 254–276, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

21 Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22:153 – 180, 2012.

22 François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. In Frédéric Blanqui and Giselle Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, volume 274 of *EPTCS*, pages 57–71, 2018. doi:10.4204/EPTCS.274.5.

23 François Thiré. *Interoperability between proof systems using the logical framework Dedukti*. PhD thesis, ENS Paris-Saclay, 2020.