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 (LF), allows for the representation of computation alongside deduction. However, unlike LF 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 result, 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 LF 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, Pure Type Systems

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.25

Related Version *Full Version*: `https://arxiv.org/abs/2205.02883`

Acknowledgements I would like to thank my PhD advisors Frédéric Blanqui and Gilles Dowek for the helpful discussions and comments on this paper. I would also like to thank the anonymous reviewers for their very helpful comments and suggestions.

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* (LF)[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 LF does not achieve.

The logical framework Dedukti[3] addresses this point by extending the LF with rewriting rules, thus allowing for the representation of both deduction and computation. This framework was already proven to be as a very expressive system, and has been used to encode the logics of many proof assistants, such as Coq[14], Agda[15], PVS[18] and others.

However, an unsatisfying aspect persists as, unlike LF encodings, the Dedukti encodings proposed until now are not *adequate*, in the sense that they do not feature a syntactical bijection between the terms of the encoded system and those of the encoding. Such property is key to ensure 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

© Thiago Felicissimo;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 25; pp. 25:1–25:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a type is inhabited, then this type is inhabited) is still a challenge, in particular for recent works such as [14, 23, 15, 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 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, contrarily to LF encodings, theirs is *computational* – that is, represents computation in the encoded system directly as computation. The authors then show that the encoding is 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, the author 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 LF 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 of them. One then needs this process to be terminating, which is non-trivial to show as it involves 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 LF encodings were very natural. They also both rely on intricate properties of the encoded systems, which is unnatural given that logical frameworks should ideally only require the encoded systems to satisfy some basic properties, and be agnostic with respect to more deep ones – for instance, one should not be obliged to show that a given system is consistent in order to encode it in a logical framework. This reason, coupled with the technicality of these proofs, may explain why recent works such as [23], [18] and [15] have left conservativity as conjecture. Moreover, none of these works have addressed the lack of an adequacy theorem, which until now has remained an overlooked problem in the Dedukti community.

Our contribution

We propose to depart from the approach of traditional Dedukti encodings by restoring the separation that existed in LF 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 LF encodings, ours is computational like other Dedukti encodings.

To show conservativity, we leverage the fact that 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 those 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 for the encoding. We then present the 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 infinitely many sorts. Finally, in Section 9 we discuss more practical aspects by showing how the encoding can be instantiated and used in practice.

2 Dedukti

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

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

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

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

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

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.

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

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 $\mathit{length}(\vec{M}) = n_c$. 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 $c[\vec{l}] \hookrightarrow r$ 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 `Cons` we use the 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 refer to Appendix A for a review of some basic metatheorems.

3 Strong Normalization of β in Dedukti

In order to show conservativity of encodings, one often needs to be able to β normalize terms, thus requiring β to be normalizing for well-typed terms. In this section we present a criterion for the normalization of β in Dedukti. More precisely, we will see that if $\beta\mathcal{R}$ is confluent and $\mathcal{R}$ is *arity preserving* (a definition we will introduce in this section), then β is SN (strongly normalizing) in Dedukti for well-typed terms. The proof generalizes the one given in [17]. However, because of space constraints, we only give the intuition of the proof and refer to the long version in [12] for all the details.

Note that, unlike works such as [8], which provide syntactic criteria on the normalization of $\beta\mathcal{R}$ in Dedukti, we only aim at showing 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 showing that this map preserves typing and non-termination of β, thus implying that β is SN in Dedukti. Before proceeding, we introduce the following basic definitions.

▶ **Definition 1.**

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 $c[\vec{l}] \hookrightarrow r$ is type-level if its head symbol c is a type-level constant.*

We can now define the erasure map.

▶ **Definition 2** (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.

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

We also extend the definition of $\|-\|$ (partially) on contexts and signatures by the following equations.

$$\begin{aligned}
\|-\| &= - \\
\|x : A, \Gamma\| &= x : \|A\|, \|\Gamma\| \\
\|c[x_1 : A_1, ..., x_n : A_n] : A; \Sigma\| &= (c : \|A_1\| \to ... \to \|A_n\| \to \|A\|), \|\Sigma\|
\end{aligned}$$

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 3.** Let El be a type-level constant, and consider the rule

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

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[1] as the number of consecutive arrows (that is, of Πs), then we realize that the problem here is that rules such as $El\ (Prod\ A\ B) \longleftrightarrow \Pi x : El\ A.El\ (B\ x)$ 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[2]. 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 motivates the following definition.

▶ **Definition 4** (Arity preserving). *$\mathcal{R}$ is said to be arity-preserving[3] 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 arbitrary.*

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

[1] Note that this concept is different from the notion of arity of constants, as defined in Section 2.
[2] 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.
[3] More precisely, this definition also depends on the signature Σ, as this is used to define which constants are type-level.

It turns out that this condition, together with confluence of $\beta\mathcal{R}$, is enough to show that the translation preserves typing and non-termination. Using the fact that well-typed terms are strongly normalizing in the simply-typed λ-calculus, we get the following theorem. We refer to the long version in [12] for the proofs.

▶ **Theorem 5** (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.*

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* $\mathcal{S}$ (refereed to by the letter 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 make 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 make explicit the sort parameters. As in PTSs the dependent product is used across multiple 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 direct, 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[4]

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

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

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

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

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

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

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

Figure 2 Typing rules for Explicitly-typed Pure Type Systems.

[4] 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 $B \equiv B'$.

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] and [21] also consider similar variants, though none of them corresponds exactly to ours. Therefore, we had to develop the basic metatheory of our version in [13], and we have found that the usual meta-theoretical properties of functional PTSs are preserved when moving to the explicitly-typed version (see Appendix B). More importantly, by a proof that uses ideas present in [21], we have shown the following equivalence.

Let $|-|$ be the erasure map defined in the most natural way from an EPTS to its corresponding PTS. Moreover, for a system X let $\Lambda(\Gamma \vdash_X _ : A)$ be the set of $M \in \Lambda_X$ with $\Gamma \vdash_X M : A$. Finally, let $\equiv_I$ be defined by $M \equiv_I N$ iff $|M| = |N|$ and $M \equiv N$.

▶ **Theorem 6** (Equivalence between PTSs and EPTSs[13]). *Consider a functional PTS. If $\Gamma \vdash_{PTS} A$ type, then there are Γ', A' with $|\Gamma'| = \Gamma, |A'| = A$ such that we have a bijection*

$$\Lambda(\Gamma \vdash_{PTS} _ : A) \simeq \Lambda(\Gamma' \vdash_{EPTS} _ : A')/\equiv_I$$

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 $(\Sigma_{\mathtt{EPTS}}, \mathcal{R}_{\mathtt{EPTS}})$ 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 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].

$$U_s : \mathtt{TYPE} \qquad El_s[A : U_s] : \mathtt{TYPE} \qquad \text{for } s \in \mathcal{S}$$

$$u_{s_1} : U_{s_2} \qquad El_{s_2}\ u_{s_1} \hookrightarrow_{u_{s_1}\text{-red}} U_{s_1} \qquad \text{for } (s_1, s_2) \in \mathcal{A}$$

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.

$$Prod_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}] : U_{s_3} \qquad \text{for } (s_1, s_2, s_3) \in \mathcal{R}$$

Traditional Dedukti encodings would normally continue here by introducing the rule $El_{s_3}\ (Prod_{s_1,s_2}\ A\ B) \hookrightarrow \Pi x : El_{s_1}\ A.El_{s_2}\ (B\ x)$, 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 encoded, that is, representing abstraction and application.

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

We note that this idea is also hinted in [1], though they did not pursue it further. This approach also resembles the one of the Edinburgh Logical Framework (LF) [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 LF 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.

This finishes the definition of the theory $(\Sigma_{\text{EPTS}}, \mathcal{R}_{\text{EPTS}})$. Now we ready to define the translation function $[\![-]\!]$.

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

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.

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

▶ Remark 7. Note that in the definition of $[\![-]\!]$ it was essential for λ and $@$ to make 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 6).

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, as is usually done with traditional Dedukti encodings[9]. Nevertheless, this complicates many proofs, as each time we apply $[\![-]\!]_\Gamma$ to a term we need to know it is well-typed in Γ. Moreover, properties which should concern all untyped terms (such as preservation of computation) would then be true only for well-typed ones.

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

▶ **Example 8.** 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

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

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 9.** *We establish the following notations.*

- *We write* $\Sigma; \Gamma \vdash_{\mathtt{DK}} M : A$ *for a* Dedukti *judgment and* $\Gamma \vdash M : A$ *for an EPTS judgment*
- *As the same signature* $\Sigma_{\mathtt{EPTS}}$ *is used everywhere, when referring to* $\Sigma_{\mathtt{EPTS}}; \Gamma \vdash_{\mathtt{DK}} M : A$ *we omit it and write* $\Gamma \vdash_{\mathtt{DK}} M : A$.

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

▶ **Proposition 10** (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* $c[\Delta] : A \in \Sigma_{\mathtt{EPTS}}$, *we have* $\Delta \vdash_{\mathtt{DK}} A : s$.
3. *Subject reduction for* β*: If* $\Gamma \vdash_{\mathtt{DK}} M : A$ *and* $M \longrightarrow_\beta M'$ *then* $\Gamma \vdash_{\mathtt{DK}} M' : A$.
4. *Strong normalization for* β*: If* $\Gamma \vdash_{\mathtt{DK}} M : A$, *the* β *is strongly normalizing for* M.
5. *Compositionality: For all* $M, N \in \Lambda_{EPTS}$ *we have* $[\![M]\!]\{[\![N]\!]/x\} = [\![M\{N/x\}]\!]$.

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 $\beta\mathcal{R}_{\mathtt{EPTS}}$[6].
4. $\mathcal{R}_{\mathtt{EPTS}}$ is arity preserving and $\beta\mathcal{R}_{\mathtt{EPTS}}$ is confluent, thus β *is SN in* Dedukti (Theorem 5) applies.
5. By induction on M. ◀

▶ Remark 11. 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 $\mathcal{R}_{\mathtt{EPTS}}$. Therefore, we conjecture that our proof method can also be adapted to systems that do not satisfy subject reduction.

▶ **Lemma 12** (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. Intuitively, the first part holds because a β step in the source system is represented by a *beta* step followed by a β step in Dedukti. It 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 13** (Equivalence for sort representations)**.** *If* s *is not a top-sort, then*

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

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

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

▶ **Theorem 14** (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. We present here only the case Prod to show the idea, the other cases are detailed in the long version in [12].

Case Prod. The proof ends with

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

By the IH and Lemma 13, 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

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

If s_3 is not a top-sort, we then apply Lemma 13. ◀

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] (with the notable exception of [2]). 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 should not be required to show conservativity.

In this section we will show how conservativity can be proven 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 inconsistent 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 15** (Invertible forms). *We call the terms generated by the following grammar the invertible forms. The s_i are arbitrary sorts in $\mathcal{S}$, whereas the T_1, T_2 are arbitrary terms.*

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

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

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

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 17.** *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 18.** *Let* M *be an 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. The property 1 is shown by induction on M, whereas 2, 3 follow by induction on the rewrite context and 4 follows directly from 2, 3. ◀

▶ Remark 19. 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 represent 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 are reflected in the encoded system.

▶ **Corollary 20** (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 18 and Proposition 17.
2. Follows from confluence of $\beta\mathcal{R}_{\mathtt{EPTS}}$ and also Proposition 18 and Proposition 17. ◀

Note that for part 2 we really need $\beta\mathcal{R}_{EPTS}$ to be confluent. Indeed, If $[\![M]\!] \hookleftarrow N$ 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 21.** *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 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 30) 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 later, we also need to show that $|-|$ is a kind of right inverse to $[\![-]\!]$. But because the inverse translation does not capture the information in the type annotations of binders, $[\![|M|]\!] = M$ does not hold.

▶ **Example 22.** 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 sent 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 23** (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 24** (Conservativity of β normal forms)**.** *Suppose $\Gamma \vdash A$ type and let $M \in \Lambda_{\mathtt{DK}}$ be a β normal form such that $[\![\Gamma]\!] \vdash_{\mathtt{DK}} M : T$, 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 $T \equiv \Pi x : A'_1.A'_2$. This then implies that T reduces to a dependent product, but because T is of the form $El_{s_A}\ [\![A]\!]$ or U_A and $\mathcal{R}_{\mathtt{EPTS}}$ is arity preserving, this 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 $c[\Delta] : C \in \Sigma_{\mathtt{EPTS}}$ or $x : C \in \Gamma$ with $C\{\vec{M}\}$ convertible to a dependent product type, for some $\vec{M}$. 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. We present only case $c = Prod_{s_1,s_2}$ here and refer to the long version in [12] for all the details.

▶ Note 25. In the following, to improve readability we omit the typing hypothesis when applying `Conv`. However, all such uses can be justified.

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 21 applied to 3, 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

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

By *Basic properties* (Proposition 10), β is strongly normalizing and type preserving. Therefore from the previous result we can immediately get full conservativity.

▶ **Theorem 26** (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 $M \in \Lambda_{EPTS}$ such that $\Gamma \vdash M : A$ and $\Lambda_{NF}(\Gamma \vdash_{\mathtt{DK}} _ : T)$ for the set of $M \in \Lambda_{\mathtt{DK}}$ in β normal form such that $\Gamma \vdash_{\mathtt{DK}} M : T$, we can show our adequacy theorem. This result follows by simply putting together *Basic properties* (Proposition 10), *Preservation of computation* (Lemma 12), *Soundness* (Theorem 14), *Reflection of computation* (Corollary 20) and *Conservativity* (Theorem 26).

▶ **Theorem 27** (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*

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

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

8 Representing systems with infinitely many 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 infinitely many 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. Indeed, every proof in an infinite sort systems only uses a finite number of sorts, and thus does not need all of them to be properly represented.

A different approach proposed in [1] is to internalize the indices of $Prod_{s_1,s_2}$, El_{s_1}, ... 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 $(\Sigma^{\mathcal{S}}_{\mathtt{EPTS}}, \mathcal{R}^{\mathcal{S}}_{\mathtt{EPTS}})$.

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

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

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

Figure 3 Definition of the theory $(\Sigma^{\mathcal{S}}_{\mathtt{EPTS}}, \mathcal{R}^{\mathcal{S}}_{\mathtt{EPTS}})$.

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 28.** 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 $\widehat{\mathcal{R}}\ (s\ x)\ (s\ y) \hookrightarrow s\ (\widehat{\mathcal{R}}\ x\ y)$.

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 $(\Sigma^{\mathcal{S}}_{\mathtt{EPTS}}, \mathcal{R}^{\mathcal{S}}_{\mathtt{EPTS}})$ as a "hidden implementation" of $(\Sigma_{\mathtt{EPTS}}, \mathcal{R}_{\mathtt{EPTS}})$. In this case, it should be possible to take a proof written in the $(\Sigma_{\mathtt{EPTS}}, \mathcal{R}_{\mathtt{EPTS}})$ and "implement" it in the $(\Sigma^{\mathcal{S}}_{\mathtt{EPTS}}, \mathcal{R}^{\mathcal{S}}_{\mathtt{EPTS}})$. Following this intuition, we define in the long version in [12] a notion of *theory morphism* which allows us to show the soundness of this new encoding by means of morphism from $(\Sigma_{\mathtt{EPTS}}, \mathcal{R}_{\mathtt{EPTS}})$ to $(\Sigma^{\mathcal{S}}_{\mathtt{EPTS}}, \mathcal{R}^{\mathcal{S}}_{\mathtt{EPTS}})$.

Nevertheless, this definition has too strong requirements and cannot be used to define a morphism in the other direction to show conservativity. Therefore, it is still an open problem for us to find a notion of morphism allowing to show the equivalence between the encodings. For the time being, in order to show conservativity (and then adequacy) of this new encoding one has to redo the work of Section 7.

9 The encoding in practice

Our encoding satisfies 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 natural way, and we refer to `https://github.com/thiagofelicissimo/examples-encodigs` 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 `https://github.com/thiagofelicissimo/encoding-benchmarking` 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[5]. 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 16 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 within 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, using more sharing in Dedukti would probably reduce the time for typechecking, as the parameter annotations in app_{s_A,s_B} and abs_{s_A,s_B} carry a lot of repetition.

10 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 shown in a much more straightforward way and which feature adequacy theorems, something that was missing from traditional Dedukti encodings. However, differently from the LF approach, our encoding is also computational. Therefore, our method combines the adequacy of LF 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 $Type : Type$. This sets our work apart from [9], whose conservativity proof requires the encoded system to be normalizing.

[5] 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.

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 5) 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.* Thesis, 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 λ π-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. `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. Adequate and computational encodings in the logical framework Dedukti. Draft at `https://lmf.cnrs.fr/Perso/ThiagoFelicissimo`, 2022.
13. Thiago Felicissimo. No need to be implicit! Draft at `https://lmf.cnrs.fr/Perso/ThiagoFelicissimo`, 2022.
14. Gaspard Ferey. *Higher-Order Confluence and Universe Embedding in the Logical Framework.* Thesis, Université Paris-Saclay, June 2021. URL: `https://tel.archives-ouvertes.fr/tel-03418761`.

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

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

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. `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. `doi: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.

A Typing rules and metatheory of Dedukti

▶ **Proposition 29** (Basic properties). *Suppose* $\longrightarrow_{\beta\mathcal{R}}$ *is confluent.*

1. *Weakening: If* $\Sigma;\Gamma \vdash M : A$, $\Gamma \sqsubseteq \Gamma'$ *and* $\Sigma;\Gamma'$ `well-formed` *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* $A = \mathtt{KIND}$

▶ **Theorem 30** (Conv in context for DK). *Let* $A \equiv A'$ *with* $\Sigma;\Gamma \vdash A' : s$. *We have*

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

▶ **Proposition 31** (Reduce type in judgement). *Suppose* $\longrightarrow_{\beta\mathcal{R}}$ *is confluent and satisfies subject reduction. Then if* $\Sigma;\Gamma \vdash M : A$ *and* $A \longrightarrow^* A'$ *we have* $\Sigma;\Gamma \vdash M : A'$.

B Metatheory of Explicitly-typed Pure Type Systems

We have the following properties for functional EPTSs. We refer to [13] for the proofs.

▶ **Proposition 32** (Weakening). *Let* $\Gamma \sqsubseteq \Gamma'$ *with* Γ' well-formed. *If* $\Gamma \vdash M : A$ *then* $\Gamma' \vdash M : A$.

▶ **Proposition 33** (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 34** (Uniqueness of types). *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *we have* $A \equiv B$.

▶ **Corollary 35** (Uniqueness of sorts). *If* $\Gamma \vdash M : s$ *and* $\Gamma \vdash M : s'$ *we have* $s = s'$.

▶ **Proposition 36** (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 37** (Substitution in judgment). *Let* $\Gamma \vdash N : A$. *We have*

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