Encoding Agda Programs using Rewriting

Guillaume Genestier
Université Paris-Saclay, ENS Paris-Saclay, Inria, CNRS, LSV, France
MINES ParisTech, PSL University, France

Abstract

We present in this paper an encoding in an extension with rewriting of the Edimburgh Logical Framework (LF) [13] of two common features: universe polymorphism and eta-convertibility. This encoding is at the root of the translator between AGDA and DEDUKTI developped by the author.

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

Keywords and phrases Logical Frameworks, Rewriting, Universe Polymorphism, Eta Conversion

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.36

Funding Part of this work was carried out during a stay at Chalmers University of Technology, Sweden, funded by the Cost Action CA15123: EUTypes

1 Introduction

With the multiplication of proof assistants, interoperability has became a main obstacle preventing the dissemination of formally verified software among industrial companies.

Indeed, a lot of mathematical results have been formalized, using many different proof assistants. Hence, if one want to use two already proved theorems in her development, there is a high risk that these two proofs are in different systems.

To avoid the community the burden of redevelopping the same proofs in each system, the LOGIPEDIA project aims at building an encyclopedia of formal proofs, agnostic in the system they were developped in. To do so, the logics of the proof assistants can be encoded in the same *Logical Framework*: DEDUKTI, which is based of the $\lambda\Pi$-calculus modulo rewriting. Once all the logics are encoded in the same framework, it becomes easier to compare them, and so to export to a target system proofs originally made in another system.

In this article, we present an encoding of two common features, shared by many proof assistants.

The first one is universe polymorphism. Introduced by Harper and Pollack [14], this allows the user to declare a symbol only once for all universe levels, and then to instantiate it several times with concrete levels.

The second one is equality modulo η. In set theory, a function is identified with its graph, hence two functions outputing the same result when fed with the same data are equal. In type theory, it is not the case. η-conversion is a weak form of this principle of extensionality, which just states that f is equal to the function associating to any x the result of f applied to x.

Developped for twenty years, AGDA is a dependently-typed functional programming language based on an extension Martin-Löf's type theory. Thanks to Curry-Howard correspondence, it is often used as a proof assistant. Furthermore, it features the two ingredients this article focuses on. Hence, the author developed, in collaboration with Jesper Cockx, an automatic translator from a fragment of AGDA to DEDUKTI.

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 36; pp. 36:1–36:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Outline

After a brief presentation of the $\lambda\Pi$-calculus modulo rewriting, Section 2 introduces the Cousineau-Dowek's encoding of *Pure Type Systems*. Section 3 presents a general encoding of universe polymorphism and an instantiation of this encoding in the special case of the predicative two-ladder universe system behind AGDA. The main theorem of this section is the preservation of typability of this encoding. Then, Section 4 explains how to encode η-conversion using rewriting. Preservation of the conversion is the main result of this section. Finally, after a presentation of the implementation in Section 5, Section 6 summarizes our result and provides hints on future extensions.

2 Encoding Pure Type Systems in $\lambda\Pi$-modulo Rewriting

In [3], Barendregt presents the λ-cube, a classification of eight widely used type systems, distinguishing themselves from each other by the possibility they offer (or not) to quantify on a type, a term to construct a type, or a term.

Those constructions of systems in the λ-cube were generalized by Terlouw and Berardi [5], giving birth to what they called "generalized type system", nowadays more often called *Pure Type Systems* (PTS).

Every PTS shares the same typing rules. The only difference between them are the relations $\mathcal{A}$ and $\mathcal{R}$. $\mathcal{A}$, called axioms, states inhabitation between sorts and $\mathcal{R}$, called rules, controls on which sort one can quantify.

▶ **Definition 1** (Syntax and typing of PTS). *Let $\mathcal{X}$ be an infinite set of variables and $\mathcal{S}$ be the set of sorts.*

$$t, u ::= s \mid x \mid (x : t) \to u \mid \lambda x^t.u \mid t\,u \qquad \text{with } s \in \mathcal{S} \text{ and } x \in \mathcal{X}$$

The typing rules include 5 introduction rules related to the syntax, and 2 structural rules.

$$(var)\quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\; x \notin \mathrm{dom}(\Gamma)$$

$$(ax)\quad \frac{}{\vdash s_1 : s_2}\;(s_1, s_2) \in \mathcal{A} \qquad (prod)\quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (x : A) \to B : s_3}\;(s_1, s_2, s_3) \in \mathcal{R}$$

$$(app)\quad \frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B\,[^u/_x]} \qquad (abs)\quad \frac{\Gamma \vdash (x : A) \to B : s \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : (x : A) \to B}$$

$$(conv)\quad \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\; A \leftrightsquigarrow^*_\beta B \qquad (weak)\quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\; x \notin \mathrm{dom}(\Gamma)$$

▶ **Definition 2** (Functional Pure Type System). *A PTS is called* functional *if axioms and rules are functional relations, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$.*

One can be even more restrictive on the class of PTS's considered, by defining a special case of *functional PTS*, the *full PTS*.

▶ **Definition 3** (Full Pure Type System). *A PTS is called* full *if axioms and rules are total functions, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$.*

▶ **Example 4** ($\mathcal{P}^\infty$ and $\mathcal{C}^\infty$). The predicative and impredicative infinite hierarchies, are two full PTS: $\mathcal{P}^\infty$ is $\mathcal{S} = \{*_i | \, i \in \mathbb{N}\}\,;\mathcal{A} = \{(*_i, *_{i+1})\}\,;\mathcal{R} = \{(*_i, *_j, *_k) | \, k = \max(i, j)\}$ whereas $\mathcal{C}^\infty$ is $\mathcal{S} = \{*_i | \, i \in \mathbb{N}\}\,;\mathcal{A} = \{(*_i, *_{i+1})\}\,;\mathcal{R} = \{(*_i, *_j, *_k) | \, j \geqslant 1 \text{ and } k = \max(i, j)\} \cup \{(*_i, *_0, *_0)\}$.

▶ **Definition 5** (Embedding of PTS). *Given $P_1 = (\mathcal{S}_1; \mathcal{A}_1; \mathcal{R}_1)$ and $P_2 = (\mathcal{S}_2; \mathcal{A}_2; \mathcal{R}_2)$ two PTS, $f : \mathcal{S}_1 \to \mathcal{S}_2$ is an embedding of P_1 in P_2 if for all $(s, s') \in \mathcal{A}_1$, we have $(f(s), f(s')) \in \mathcal{A}_2$ and for all $(s, s', s'') \in \mathcal{R}_1$, we have $(f(s), f(s'), f(s'')) \in \mathcal{R}_2$.*

f is extended to terms of P_1, by: $f(x) = x$, *if* $x \in \mathcal{X}$; $f(\lambda x^A.t) = \lambda x^{f(A)}.f(t)$; $f(t\,u) = f(t)\,f(u)$; $f((x : A) \to B) = (x : f(A)) \to f(B)$.

▶ **Proposition 6** (Soundness of the Embedding). *If f is an embedding from a PTS P_1 to P_2, if $\Gamma \vdash_{P_1} t : A$, then $f(\Gamma) \vdash_{P_2} f(t) : f(A)$.*

Proof. By induction on the proof tree. Since f preserves $\mathcal{A}$ and $\mathcal{R}$, the (ax) and $(prod)$ cases are satisfied. All the other cases are direct, since f does not act on the shape of terms. ◀

The Edimburgh Logical Framework [13] (LF), denoted λP in Barendregt's λ-cube is the minimal PTS including dependent types. It has two sorts $\mathcal{S} = \{\star, \square\}$, with the axioms $\mathcal{A} = \{(\star, \square)\}$ and the rules $\mathcal{R} = \{(\star, \star, \star), (\star, \square, \square)\}$. It is well-known to be "a framework for defining logics", since it allows to encode most of the proof systems. One can note, LF is not a Full PTS, since $\square$ is the left-hand side of no axioms.

The logic behind the *Logical Framework* Dedukti is the $\lambda\Pi$-calculus modulo rewriting [2, 6], an extension of the Edimburg Logical Framework with user-defined rewrite rules used not only to define functions, but also types, allowing for shallow embedding of various type systems. Indeed, even if one can encode many logics in LF, those encodings are deep, meaning that applications, λ-abstractions and variables of the encoded system are not translated directly by their equivalent in LF, but by using explicit symbols `App`, `Lam` and `Var`. Using rewriting, the introduction of those extra symbols can be avoided, allowing for more reasonable size translations.

▶ **Definition 7** (Signature in $\lambda\Pi$-modulo rewriting). *A signature in $\lambda\Pi$-modulo rewriting is $(\Sigma, \Theta, \mathbb{R})$ where Σ is a set of symbols, disjoint of $\mathcal{X}$, Θ is a function from Σ to terms and $\mathbb{R}$ is a set of rewriting rules, i.e. a set of pair of terms of the form $f\,\vec{l} \hookrightarrow r$, with $f \in \Sigma$ and all l_i's are Miller's pattern [16].*

We say that t rewrites to u, denoted $t \rightsquigarrow u$ if there is a rule $f\,\vec{l} \hookrightarrow r$, a substitution σ and a "term with a hole" $C[]$, such that $t = C[(f\,\vec{l})\sigma]$ and $u = C[r\sigma]$. $\rightsquigarrow$ is the smallest relation containing $\hookrightarrow$ and stable by substitution and context. We denote by $\rightsquigarrow^*$ the reflexive transitive closure of $\rightsquigarrow$ and by $\leftrightsquigarrow^*$ the convertibility relation, which is the reflexive symmetric and transitive closure of $\rightsquigarrow$.

▶ **Definition 8** (Typing rules of $\lambda\Pi$-modulo rewriting). *They are the one of LF (those of Def. 1, instantiated with $\mathcal{S} = \{\star, \square\}$, $\mathcal{A} = \{(\star, \square)\}$ and $\mathcal{R} = \{(\star, \star, \star), (\star, \square, \square)\}$.), but with a rule to introduce symbols of Σ and enrichment of the conversion, to include both β-reduction and the user-defined rewriting rules.*

$$(sig)\ \frac{\Gamma \vdash \Theta(f) : s \quad f \in \Sigma}{\Gamma \vdash f : \Theta(f)} \qquad (conv)\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ A \leftrightsquigarrow^*_{\beta \cup \mathbb{R}} B$$

In 2007, Cousineau and Dowek [8] proposed an encoding of any functional PTS in Dedukti. Their encoding contained two symbols for each sort, and one symbol for each axiom or rule. However, having an infinite number of symbols and rules is not well-suited for implementations. Hence, to encode *Pure Type Systems* with an infinite number of sorts, one prefers to have a type `Sort` for sorts and only one symbol for products [1]. For *full Pure Type Systems*, this extension is quite straightforward. The general encoding of full PTS is:

First the PTS specificification: a type of sorts and two functions for $\mathcal{A}$ and $\mathcal{R}$.

```
constant Sort : TYPE.
symbol axiom : Sort ⇒ Sort.        symbol rule : Sort ⇒ Sort ⇒ Sort.
```

For each sort `s`, a type `Univ s` containing the codes of its elements. Indeed, since the $\lambda\Pi$-calculus, does not allow to quantify over types, one needs to declare the type of the logic we are encoding, not directly as a type, but as a code, which can be decoded to a type using rewriting rules.

```
constant Univ : (s : Sort) ⇒ TYPE.
```

Then a symbol to decode the elements of `Univ s` as type of $\lambda\Pi$-modulo rewriting.

```
symbol Term : (s : Sort) ⇒ Univ s ⇒ TYPE.
```

The encoding of sorts and the rewrite rule to decode it. (Simulates the rule (ax) of a PTS).

```
constant code : (s : Sort) ⇒ Univ (axiom s).
Term _ (code s) ⟶ Univ s.
```

The encoding of products and its decoding rewrite rule. (Simulates the rule ($prod$) of a PTS).

```
constant prod : (s1 : Sort) ⇒ (s2 : Sort) ⇒
        (A : Univ s1) ⇒ (Term s1 A ⇒ Univ s2) ⇒ Univ (rule s1 s2).
Term _ (prod a b A B) ⟶ (x : Term a A) ⇒ Term b (B x).
```

Then the peculiarity of each PTS is reflected in the encoding of the elements of $\mathcal{S}$ as terms of `Sort`, and in the implementation of `axiom` and `rule` to encode $\mathcal{A}$ and $\mathcal{R}$ respectively.

3 Universe Polymorphism and its Encoding

It is quite common to enrich PTS with *Universe Polymorphism* [14], which consists in allowing the user to quantify over *universe levels*, allowing to declare simultaneously a symbol for several sorts. For instance, if the sorts are $\{\, \mathrm{Set}_i |\, i \in \mathbb{N}\}$, then one want to declare `List` in $\forall \ell, (A : \mathrm{Set}_\ell) \to \mathrm{Set}_\ell$. Indeed, just like polymorphism was used to avoid declaring a type of lists for each type of elements, one want to avoid one declaration of a new type of lists for each universe level.

We present here a definition of *universe polymorphism* inspired by the one given by Sozeau and Tabareau [19] for the proof assistant Coq. In this setting, the context contains three lists: a list Σ called signature, a list Θ of level variables, and a list Γ called local context. Both Σ and Γ contain pairs of a variable name and a type, but the variables in Γ can contain free level variables (those occuring in Θ), whereas all the level variables are bound by a prenex quantifier $\forall$ in the signature Σ. Unlike [19], we do not need to store constraints between universe levels, since those constraints are related to cumulativity, a feature we are not trying to encode here.

▶ **Definition 9** (Uniform Universe Polymorphic Full PTS). *We consider a set $\mathbb{L}$ of levels and a finite set $\mathcal{H}$ of sort constructors. Then the sorts are $\{s_\ell\}_{s\in\mathcal{H},\ell\in\mathbb{L}}$.*

In addition to functionality and totality of $\mathcal{A}$ and $\mathcal{R}$, we assume a uniformity in the hierarchy. Meaning that for all $s \in \mathcal{H}$, there is a $s' \in \mathcal{H}$, such that for all $\ell \in \mathbb{L}$, there is a $\ell' \in \mathbb{L}$, such that $(s_\ell, s'_{\ell'}) \in \mathcal{A}$ and for all $s^{(1)}, s^{(2)} \in \mathcal{H}$, there is a $s^{(3)} \in \mathcal{H}$, such that for all $\ell_1, \ell_2, \in \mathbb{L}$, there is $\ell_3 \in \mathbb{L}$ such that $(s^{(1)}_{\ell_1}, s^{(2)}_{\ell_2}, s^{(3)}_{\ell_3}) \in \mathcal{R}$.

We denote by $\bar{A}$ the function $\{\, (s,s') \in \mathcal{H}^2 \,|\, \exists \ell, \ell', (s_\ell, s'_{\ell'}) \in \mathcal{A}\}$ and for all s by $\mathcal{A}_s$ the function $\{\, (\ell,\ell') \in \mathbb{L}^2 \,|\, \exists s', (s_\ell, s'_{\ell'}) \in \mathcal{A}\}$.

Analogously $\bar{\mathcal{R}}$ is the function $\left\{ (s^{(1)}, s^{(2)}, s') \in \mathcal{H}^3 \,\middle|\, \exists \ell_1, \ell_2, \ell', (s^{(1)}_{\ell_1}, s^{(2)}_{\ell_2}, s'_{\ell'}) \in \mathcal{R} \right\}$ and for all $(s^{(1)}, s^{(2)})$, $\mathcal{R}_{s^{(1)}, s^{(2)}}$ is the function $\left\{ (\ell_1, \ell_2, \ell') \in \mathbb{L}^3 \,\middle|\, \exists s', (s^{(1)}_{\ell_1}, s^{(2)}_{\ell_2}, s'_{\ell'}) \in \mathcal{R} \right\}$.

The typing rules are:

$$\text{(lvl)}\quad \frac{}{\Theta \vdash \ell \ \text{ isLvl}}\ \ell \in \mathbb{L} \qquad\qquad \text{(ax)}\quad \frac{\Theta \vdash \gamma \ \text{ isLvl}}{[]; \Theta; [] \vdash s_\gamma : s'_{\mathcal{A}_s(\gamma)}}\ (s, s') \in \bar{\mathcal{A}}$$

$$(\mathbb{L}var)\quad \frac{}{\Theta \vdash i \ \text{ isLvl}}\ i \in \Theta \qquad\qquad \text{(abs)}\quad \frac{\Sigma; \Theta, \Gamma \vdash (x : A) \to B : s_\gamma \quad \Sigma; \Theta; \Gamma, x : A \vdash t : B}{\Sigma; \Theta; \Gamma \vdash \lambda(x : A).t : (x : A) \to B}$$

$$(\mathbb{L}\mathcal{A})\quad \frac{\Theta \vdash \ell \ \text{ isLvl}}{\Theta \vdash \mathcal{A}_s(\ell) \ \text{ isLvl}} \qquad\qquad \text{(app)}\quad \frac{\Sigma; \Theta; \Gamma \vdash t : (x : A) \to B \quad \Sigma; \Theta; \Gamma \vdash u : A}{\Theta; \Gamma \vdash t\, u : B\, [u/_x]}$$

$$(\mathbb{L}\mathcal{R})\quad \frac{\Theta \vdash \ell_1 \ \text{ isLvl} \quad \Theta \vdash \ell_2 \ \text{ isLvl}}{\Theta \vdash \mathcal{R}_{ss'}(\ell_1, \ell_2) \ \text{ isLvl}} \qquad\qquad \text{(conv)}\quad \frac{\Sigma; \Theta; \Gamma \vdash t : A \quad \Sigma; \Theta; \Gamma \vdash B : s_\gamma}{\Sigma; \Theta; \Gamma \vdash t : B}\ A \leftrightsquigarrow^*_\beta B$$

$$\text{(var)}\quad \frac{\Sigma; \Theta; \Gamma \vdash A : s_\gamma}{\Sigma; \Theta; \Gamma, x : A \vdash x : A}\ x \notin \Sigma, \Gamma \qquad\qquad \text{(sig)}\quad \frac{\Sigma; \Theta; [] \vdash A : s_\gamma}{\Sigma, x : \forall \Theta.A; \Theta'; [] \vdash x : \forall \Theta.A}\ x \notin \Sigma, \Gamma$$

$$\text{(inst)}\quad \frac{\Sigma; \Theta; \Gamma \vdash t : \forall [i_1, \dots, i_n], A \quad \Theta \vdash \gamma_1 \ \text{ isLvl} \quad \dots \quad \Theta \vdash \gamma_n \ \text{ isLvl}}{\Sigma; \Theta; \Gamma \vdash t[\gamma_1, \dots, \gamma_n] : A \left[\gamma_k/_{i_k}\right]_k}$$

$$\text{(prod)}\quad \frac{\Sigma; \Theta; \Gamma \vdash A : s_\gamma \quad \Sigma; \Theta; \Gamma, x : A \vdash B : s'_{\gamma'}}{\Sigma; \Theta; \Gamma \vdash (x : A) \to B : s''_{\mathcal{R}_{s,s'}(\gamma, \gamma')}} (s, s', s'') \in \bar{\mathcal{R}}$$

$$\text{(ctx-weak)}\quad \frac{\Sigma; \Theta; \Gamma \vdash A : s_\gamma \quad \Sigma; \Theta; \Gamma \vdash t : B}{\Sigma; \Theta; \Gamma, x : A \vdash t : B}\ x \notin \Sigma, \Gamma$$

$$\text{(sig-weak)}\quad \frac{\Sigma; \Theta; [] \vdash A : s_\gamma \quad \Sigma; \Theta'; [] \vdash t : B}{\Sigma, x : \forall \Theta.A; \Theta'; \Gamma \vdash t : B}\ x \notin \Sigma, \Gamma$$

In all those typing rules, $s, s' \in \mathcal{H}$ and $i, x \in \mathcal{X}$. Furthermore, we allowed ourselves to simply write $x \notin \Sigma, \Gamma$, rather than "for all A, $x : A$ is not in Σ, Γ".

One typical case of use, is to have only one hierarchy: $\mathcal{H} = \{\text{Set}\}$ and to use natural numbers for levels: $\mathbb{L} = \mathbb{N}$. But we do not want to restrict ourselves to have only one hierarchy, since some proof assistants feature several. For instance, in Agda and Coq, there are 2, called Set and Prop, and Type and SProp respectively.

The two rules modifying the signature Σ, allows to completely change the set Θ of names of local variables. Changing this set during the proof is not necessary, however, without this renewal of Θ, all the symbols in the signature would have been quantified over the same set Θ, no matter which variables occur really in it.

The universe polymorphism we are interested in is purely prenex. Furthermore, universally quantified types are not typed themselves and are only inhabited by variables. This form of universe polymorphism only provides ease of use, but it does not allow to prove more, meaning that it does not compromise the consistency of the logic.

To prove this, one can construct a new PTS $(\mathcal{S}^\Theta, \mathcal{A}^\Theta, \mathcal{R}^\Theta)$ simply by adding a brand new sort for every expression containing a level variable (such expressions are in $\mathbb{L}^+_\Theta$). Then embedding this newly-constructed PTS in the original one is defined just by interpreting level variables. Then using this interpretation of the variables, one can mimic the proofs done using universe polymorphism in the original PTS.

▶ **Proposition 10** (Conservativity of the universe polymorphism). *Let $P = (\mathbb{L}, \mathcal{H}, \mathcal{A}, \mathcal{R})$ be a uniform universe polymorphic full PTS and Θ be a subset of $\mathcal{X}$.*

Let $\mathbb{L}^+_\Theta$ *be the smallest subset such that:*

$$\mathbb{L}^+_\Theta = \Theta \cup \left\{ \mathcal{A}_s(l) \middle| s \in \mathcal{H}, l \in \mathbb{L}^+_\Theta \right\} \cup \left\{ \mathcal{R}_{ss'}(l_1, l_2) \middle| s, s' \in \mathcal{H}, (l_1, l_2) \in (\mathbb{L} \cup \mathbb{L}^+)^2 \setminus \mathbb{L}^2 \right\}.$$

Let $\mathcal{X}^+ = \mathcal{X} \cup \left\{ y[l_1, \dots, l_n] \middle| y \in \mathcal{X}, n \in \mathbb{N}, (l_1, \dots, l_n) \in (\mathbb{L} \cup \mathbb{L}^+_{\mathcal{X}})^n \right\}$ *and* P^Θ *be the PTS:*

$$\mathcal{S}^\Theta = \left\{ s_l \middle| s \in \mathcal{H}, l \in \mathbb{L} \cup \mathbb{L}^+_\Theta \right\}; \qquad \mathcal{A}^\Theta = \mathcal{A} \cup \left\{ \left(s_l, s'_{\mathcal{A}_s(l))}\right) \middle| (s, s') \in \bar{\mathcal{A}}, l \in \mathbb{L}^+_\Theta \right\}$$

$$\mathcal{R}^\Theta = \mathcal{R} \cup \left\{ \left(s_{l_1}, s'_{l_2}, s''_{\mathcal{R}_{ss'}(l_1,l_2))}\right) \middle| (s, s', s'') \in \bar{\mathcal{R}}, (l_1, l_2) \in (\mathbb{L} \cup \mathbb{L}^+)^2 \setminus \mathbb{L}^2 \right\}$$

a. *There is an embedding from* P^Θ *to the underlying PTS of* P.

b. *If* $\Sigma; \Theta; \Gamma \vdash t : A$ *in* P *and* A *is not a universal quantification, then there is a* $\bar{\Sigma} \subset \left\{ x[l_1, \dots, l_n] : A' \middle| x : \forall[y_1, \dots, y_n].A \in \Sigma, A' = A\left[{}^{l_i}/_{y_i}\right]_{i=1\dots n} \text{ and all } l_i \in \mathbb{L} \cup \mathbb{L}^+_\Theta \right\}$ *such that* $\bar{\Sigma}, \Gamma \vdash_{P^\Theta} t : A$ *using the enriched set of variables* $\mathcal{X}^+$.

Proof sketch. a. The embedding consists in just chosing a level for each variable in Θ.

b. Since A is not a universal quantification, in the proof of $\Sigma; \Theta; \Gamma \vdash t : A$, all the (*sig*) are followed directly by an arbitrary number of weakenings and a (*inst*). The weakenings can be anticipated and to create a proof in P^Θ, the (*sig*) and (*inst*) are compressed in a single introduction of a variable of $\bar{\Sigma}$. ◀

In a PTS, if $\Gamma \vdash t : A$, then there is a sort s such that $A = s$ or $\Gamma \vdash A : s$. In a full PTS, $\mathcal{A}$ is a total function, hence, all sorts inhabit a sort, allowing us to refer to s as the sort of a A. However, in the presentation of universe polymorphism of Def. 9, this property is lost because universally quantified types have no type. To overcome this issue, we assign artificially a type to those quantified types, using a brand new sort Sort_ω, which is not typable, is the type of no sort and over which one cannot quantify. Its only purpose is to make "the sort of A" well-defined whenever A is inhabited. It must be noted that Sort is not in $\mathcal{H}$ and ω is not a level.

To encode *Universe Polymorphic Full PTS*, one introduce a symbol `sortOmega` and a quantification symbol `∀𝕃` which takes as first argument the sort in which the term will live once instanciated. The definition of the decoding function `Term` is enriched with a new rule, specifying its behaviour when applied to a `∀𝕃`.

▶ **Definition 11** (Encoding).

```
constant sortOmega : Sort.
constant ∀𝕃 : (f:(𝕃⇒Sort)) ⇒ ((l:𝕃) ⇒ Univ (f l)) ⇒ Univ sortOmega.
Term _ (∀𝕃 f t) ⟶ (l : 𝕃) ⇒ Term (f l) (t l).
```

For instance, the encoding of $\forall \ell, \text{Set}_\ell$ is `∀𝕃 (λ l, axiom (set l)) (λ l, code (set l))`, if `set` is a sort constructor in the encoding. And its decoding (when applying `Term sortOmega`) is, as expected, `(l:𝕃) ⇒ Univ (set l)`.

▶ **Example 12.** Consider the system $\mathcal{H} = \{s, \sigma\}$, $\mathcal{A} = \left\{ (A_i, s_{ax_A(i)}) \middle| A \in \mathcal{H} \right\}$ and $\mathcal{R} = \left\{ (A_i, B_j, B_{ru(i,j)}) \middle| A, B \in \mathcal{H} \right\}$, with ax_s, ax_σ and ru three functions remaining abstract here. ru could be indexed by two sorts, for ease of readibility, we have chosen not present such a general case.

```
(; one symbol for each sort constructor ;)
constant s : 𝕃 ⇒ Sort.                constant σ : 𝕃 ⇒ Sort.
(; Function axiom ;)
```

```
symbol axiom : Sort ⇒ Sort.
symbol ax_s : 𝕃 ⇒ 𝕃.
axiom (s i) ⟶ s (ax_s i).
(; Function rule ;)
symbol rule : Sort ⇒ Sort ⇒ Sort.
rule (s i) (s j) ⟶ s (ru i j).
rule (σ i) (s j) ⟶ s (ru i j).

symbol ax_σ : 𝕃 ⇒ 𝕃.
axiom (σ i) ⟶ s (ax_σ i).

symbol ru : 𝕃 ⇒ 𝕃 ⇒ 𝕃.
rule (s i) (σ j) ⟶ σ (ru i j).
rule (σ i) (σ j) ⟶ σ (ru i j).
```

▶ **Definition 13** (Translation). *We translate well-typed terms in a Universe Polymorphic Full Pure Type System by:* $\|x\| = \mathtt{x}$*;* $\|s_\ell\| = \mathtt{code}\ |s_\ell|_S$*;* $\|t\,u\| = \|t\|\ \|u\|$*;*
$\|\lambda x^A.t\| = \lambda(\mathtt{x} : \mathtt{Term}\ \ |s_A|_S\ \|A\|).\|t\|$*;*
$\|(x : A) \to B\| = \mathtt{prod}\,|s_A|_S\ |s_B|_S\ \|A\|\ (\lambda\mathtt{x} : \mathtt{Term}\ |s_1|_S\ \|A\|.\|B\|)$*;*
$\|\forall[\ell_1,\dots,\ell_n], A\| = \forall_{\mathbb{L}}\ (\lambda\ell_1 : \mathbb{L}.\ \mathtt{sortOmega})\ (\lambda\ell_1 \dots \forall_{\mathbb{L}}\ (\lambda\ell_n : \mathbb{L}.|s_A|_S)\ (\lambda\ell_n : \mathbb{L}.\|A\|)\dots)$*;*
$\|A[\gamma_1,\dots,\gamma_n]\| = \|A\|\ |\gamma_1|_L \dots |\gamma_n|_L$*.*
The translation of sorts is $|\mathrm{Sort}_\omega|_S = \mathtt{sortOmega}$*,* $|s_\gamma|_S = \mathtt{s}\ |\gamma|_L$*.*
And the translation of levels is $|i|_L = i$ *if* $i \in \mathcal{X}$*;*
$|\mathcal{A}_s(\ell)|_L = \mathtt{ax\_s}\ |\ell|_L$ *and* $|\mathcal{R}_{ss'}(\ell_1,\ell_2)|_L = \mathtt{ru\_ss'}\ |\ell_1|_L\ |\ell_2|_L$*.*
Wherever they are used, s_A *and* s_B *are respectively the sorts of* A *and* B*.*

It can be noted that the translation $|\ell|_L$ for $\ell \in \mathbb{L}$ is not given, since in general the number of level is infinite, hence, we do not want to introduce one new symbol per level. Furthermore, with universe polymorphism, universe levels are open terms, hence, convertibility between universe levels is now an issue. Fortunately, it is the last one, since once this issue is overcome, the encoding has one of the expected properties: we type check at least as much terms as in the original system.

To state this, we start with two useful lemmas:

▶ **Lemma 14** (Substitution and conversion). *a. If* x *is a free variable in* t *such that* t *and* $t\,[{}^{u}/_{x}]$ *are well-typed,* $\|t\,[{}^{u}/_{x}]\| = \|t\|\left[{}^{\|u\|}/_{x}\right]$*;*
b. If ℓ *is a level variable in* t *such that* t *and* $t\,[{}^{u}/_{\ell}]$ *are well-typed,* $\|t\,[{}^{u}/_{x}]\| = \|t\|\left[{}^{|u|_L}/_{x}\right]$*;*
c. If $t \rightsquigarrow_\beta u$*, then* $\|t\| \rightsquigarrow_\beta \|u\|$*.*

Proof. a and b are proved by induction on the the term t. c is because a β-redex is translated as a β-redex. ◀

The proof of this property is only sketched, since Section 4 will contain detailled proofs on the conversion specifically.

▶ **Lemma 15** (Shape-preservation of type). *a. If* s *is a sort,* $\mathtt{Term}\,|\mathcal{A}(s)|_S\,\|s\| \rightsquigarrow^* \mathtt{Univ}\,|s|_S$*,*
b. If $(x : A) \to B$ *is of sort* s*,* $\mathtt{Term}\,|s|_S\,\|(x : A) \to B\| \rightsquigarrow^* (x : \mathtt{Term}\,|s_A|_S\,\|A\|) \Rightarrow \mathtt{Term}\,|s_B|_S\,\|B\|$*;*
c. If $\ell_1 < \dots < \ell_n$*,* $\mathtt{Term}\ \mathtt{sortOmega}\,\|\forall\,\{\ell_i\}_i\,, A\| \rightsquigarrow^* (\ell_1 : \mathbb{L}) \Rightarrow \dots \Rightarrow (\ell_n : \mathbb{L}) \Rightarrow \|A\|$*.*

Proof. The three rules on `Term` are crafted to ensure those properties. ◀

To state properly the Correctness Theorem, one first has to define the translation of contexts:

▶ **Definition 16** (Context Translation). *If* $\Sigma = x_1 : T_1, \dots, x_l : T_l$*,* $\Theta = i_1, \dots, i_m$ *and* $\Gamma = y_1 : A_1, \dots, y_n : A_n$*, then the translation is* $\|\Sigma;\Theta;\Gamma\| = x_1 : \mathtt{Term}\ \mathtt{sortOmega}\ \|T_1\|, \dots,$ $x_l : \mathtt{Term}\ \mathtt{sortOmega}\ \|T_l\|, i_1 : \mathbb{L}, \dots, i_m : \mathbb{L}, y_1 : \mathtt{Term}\ |s_{A_1}|_S\ \|A_1\|, \dots, y_n : \mathtt{Term}\ |s_{A_n}|_S\ \|A_n\|$*.*

▶ **Theorem 17** (Correctness). *Given a correct criterion for equality of levels (i.e. if two levels ℓ_1 and ℓ_2 are equals, their translations $|\ell_i|_L$ are convertible), for a Universe Polymorphic Full Pure Type System P, if $\Sigma;\Theta;\Gamma \vdash t : A$, then $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|t\| : \texttt{Term}\ |s|_S\ \|A\|$, where s is the sort of A.*

Proof. By induction on the derivation. We assume that if $\Theta \vdash \gamma\,\text{isLvl}$, then $\|[];\Theta;[]\| \vdash_{\lambda\Pi/P} |\gamma|_L : \mathbb{L}$, a property which can be proved by induction on the derivation, with the assumption that for all $\ell \in \mathbb{L}$, $\vdash_{\lambda\Pi/P} |\ell|_L : \mathbb{L}$. We then consider the 10 remaining cases:

(var) By induction hypothesis, $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|A\| : \texttt{Univ}\ |s_\gamma|_S$. Hence $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \texttt{Term}\ |s_\gamma|_S\ \|A\|$: TYPE, so one can introduce a variable of this type.

(ax) The translation of s_γ is code (s $|\gamma|_L$) which lives in Univ (s' (ax_s $|\gamma|_L$)), which is the reduct of the translation as type of $s'_{\mathcal{A}_s(\gamma)}$.

(abs) By induction hypothesis, $\|\Sigma;\Theta;\Gamma\|$,x : Term $|s|_S\ \|A\| \vdash_{\lambda\Pi/P} \|t\|$: Term $|s'|_S\ \|B\|$, hence, one has that $\lambda(x$: Term $|s|_S\ \|A\|).t$ inhabits $(x$: Term $|s|_S\ \|A\|) \to$Term $|s'|_S\ \|B\|$, which is the reduct of the translation as type of $(x : A) \to B$. The other induction hypothesis $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|(x : A) \to B\|$: Univ $|s_\gamma|_S$ ensures us that Term $|s|_S\ \|A\|$ lives in TYPE.

(app) By the induction hypothesis and the Lem. 15, one can apply the translation of t to the translation of u. The result lives in the translation of $B\,[{}^{u}/_{x}]$ thanks to Lem. 14.

(conv) This is a direct consequence of Lem. 14 and the induction hypotheses.

(sig) By induction hypothesis, $\|\Sigma;\Theta;[]\| \vdash_{\lambda\Pi/P} \|A\|$: Univ $|s_\gamma|_S$. Hence, one can use the (prod) rule of $\lambda\Pi$-modulo rewriting to move all the $i : \mathbb{L}$ from the context to the term. By Lem. 15, the product obtained is convertible with $\|\forall\Theta.A\|$, hence one can introduce a variable of this type. One must then use the weakening, to Re-invent the variables of type $\mathbb{L}$ corresponding to the Θ'.

(inst) Lem. 15 tells us that, after conversion, the induction hypothesis is $\|\Sigma;\Theta;\Gamma\| \vdash \|A\| : (\ell_1 : \mathbb{L}) \to \cdots \to (\ell_n : \mathbb{L}) \to \|X\|$, hence, we can apply the γ_i's without type issues.

(prod) By induction hypothesis, we have $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|A\|$: Univ $\|s_\gamma\|$ and also $\|\Sigma;\Theta;\Gamma, x : A\| \vdash_{\lambda\Pi/P} \|B\|$: Univ $\|s'_{\gamma'}\|$, so $\|\Sigma;\Theta;\Gamma\|, x$: Term $|s_\gamma|_S\ \|A\| \vdash_{\lambda\Pi/P} \|B\|$: Univ $\|s'_{\gamma'}\|$ and we can conclude by introducing the lambda and applying prod.

(ctx-weak) As before, we have $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P} \|A\|$: Univ $\|s_\gamma\|$, so $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi/P}$Term $|s_\gamma|_S\ \|A\|$: TYPE, so one can weaken with a variable of this type.

(∀weak) Like for the (sig) rule, one can empty the context of the variables of type $\mathbb{L}$ by applying the rule (prod) of $\lambda\Pi$-modulo rewriting. Then, one can weaken with a variable of this type and variables of type $\mathbb{L}$ to translate the Θ'. ◀

Now, we will more specifically focus on a specific hierarchy of levels, where $\mathbb{L} = \mathbb{N}$ and all the $\mathcal{A}_s$ are the successor function and all $\mathcal{R}_{ss'}$ are the maximum function. This is the predicative hierarchy of $\mathcal{P}^\infty$ (Expl. 4), used in Agda for instance.

The grammar of universe level we are interested in is: $t, u \in \mathcal{L} ::= x \in \mathcal{X} \mid 0 \mid \mathrm{s}\,t \mid \max\,t\,u$:

```
constant 𝕃 : TYPE.                symbol 0 : 𝕃.
symbol s : 𝕃 ⇒ 𝕃.                 symbol max : 𝕃 ⇒ 𝕃 ⇒ 𝕃.
```

The question which arises in the translation is to have a convergent rewrite system such that for all t and u in $\mathcal{L}$:

$$t{\downarrow} = u{\downarrow} \text{ if and only if } \forall\sigma : \mathcal{X} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$$

where $[\![\_]\!]\_ : \mathcal{L} \to (\mathcal{X} \to \mathbb{N}) \to \mathbb{N}$ is the obvious interpretation in $\mathbb{N}$:

$$[\![0]\!]_\sigma = 0_\mathbb{N} \quad [\![x]\!]_\sigma = \sigma(x), \text{ if } x \in \mathcal{X} \quad [\![\mathrm{s}\,t]\!]_\sigma = [\![t]\!]_\sigma +_\mathbb{N} 1_\mathbb{N} \quad [\![\max\,t\,u]\!]_\sigma = \max{}_\mathbb{N}([\![t]\!]_\sigma, [\![u]\!]_\sigma)$$

Since max is associative and commutative (AC), we will propose an encoding having a weak version of this property: $t{\downarrow} \equiv_{AC} u{\downarrow}$ if and only if $\forall \sigma : \mathcal{X} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$.

Since $[\![\mathrm{s}\,(\max\, t\, u]\!] = [\![\max\,(\mathrm{s}\,t)\,(\mathrm{s}\,u)]\!]$, one can consider having a `Max` acting on a set of terms, which do not contain max.

Furthermore, we have for all n the equality $[\![\max\,(\mathrm{s}^n\, x)\, x]\!] = [\![\mathrm{s}^n\, x]\!]$. To avoid declaring this rule infinitely often (once for every n), we add addition to our encoding. However, since this addition encodes iteration of the application of s, it is not an addition between two levels, but one between a ground natural number and a level. Furthermore, $[\![\max\,(\mathrm{s}^n\, x)\,(\mathrm{s}^m\, 0)]\!] = [\![\mathrm{s}^n\, x]\!]$, if $m < n$. Hence, the symbol `Max` will also collect the value of the smallest possible ground natural that the result can be.

Hence, in our encoding, the normal forms are the Max $i\ \{j_k + x_k\}_k$ where:

(1) $i, j_1, \ldots$ are ground naturals, (2) $x_1, \ldots$ are distinct variables, (3) for all k, $i \geqslant j_k$.

A separate type $\mathbb{N}$, containing only ground natural numbers, is declared, to avoid confusion with levels.

```
constant ℕ : TYPE.          constant 0ℕ : ℕ.        constant sℕ : ℕ ⇒ ℕ.
definition 1ℕ := sℕ 0ℕ.
symbol maxℕ : ℕ ⇒ ℕ ⇒ ℕ.          maxℕ 0ℕ y ⟶ y.
maxℕ x 0ℕ ⟶ x.                    maxℕ (sℕ x) (sℕ y) ⟶ sℕ (maxℕ x y).
infix +ℕ : ℕ ⇒ ℕ ⇒ ℕ.
0ℕ +ℕ y ⟶ y.                     (sℕ x) y ⟶ sℕ (x +ℕ y).
```

Sets can be empty or singleton or union of sets. This union operator is an associative and commutative symbol. Furthermore, since singletons are of the form $\{i + x\}$, the constructor of singletons is denoted $\oplus$.

```
symbol ∅: LSet.    infix ⊕: ℕ⇒𝕃⇒LSet.    infix ac ∪: LSet⇒LSet⇒LSet.
x ∪ ∅ ⟶ x.
```

Since constraint (1) is guaranteed by typing, we still have to implement the two constraints (2) and (3) presented in the description of the normal form:

- The only non-left-linear rule of the encoding eliminates redundancies, ensuring that all variables in the normal forms are distinct, in order to satisfy the invariant (2).

```
(i ⊕ l) ∪ (j ⊕ l) ⟶ (maxℕ i j) ⊕ l.
```

- Intuitively, to flatten the entanglement of max and plus, we would like to have a rule stating that $a + \max(b, c) = \max(a + b, a + c)$.
  However, to fulfill constraint (3), we added the invariant that the first argument of `Max` is larger than all the first arguments of the $\oplus$ occuring directly under it. Hence, we do not declare the expected computation rule of $\oplus$, but enforce this computation to be performed under a `Max`.
  Furthermore, for typing distinction between $\mathbb{L}$ and `LSet`, we introduce an auxiliary function mapping (`i` $\oplus$ _) to all the elements of a set.

```
symbol mapPlus : ℕ ⇒ LSet ⇒ LSet.
mapPlus i ∅ ⟶ ∅.              mapPlus i (j ⊕ l) ⟶ (i +ℕ j) ⊕ l.
mapPlus i (l1 ∪ l2) ⟶ (mapPlus i l1) ∪ (mapPlus i l2).
symbol Max : ℕ ⇒ LSet ⇒ 𝕃          Max 0ℕ (0ℕ ⊕ x) ⟶ x.
Max i  (j ⊕ Max k l)       ⟶ Max (maxℕ i (j +ℕ k)) (mapPlus j l).
Max i ((j ⊕ Max k l) ∪ tl) ⟶
                        Max (maxℕ i (j +ℕ k)) ((mapPlus j l) ∪ tl).
```

And finally we give rewrite rules for the symbols of the syntax:

```
0 ⟶ Max 0ℕ ∅.                    s x ⟶ Max 1ℕ (1ℕ ⊕ x).
max x y ⟶ Max 0ℕ ((0ℕ ⊕ x) ∪ (0ℕ ⊕ y)).
```

This encoding is not confluent, as the following example illustrates:

```
Max i (j ⊕ (Max k (j2 ⊕ (Max k2 l))))
 ⇝o Max (maxℕ i (j +ℕ k)) (mapPlus j (j2 ⊕ (Max k2 l)))
  ⇝ Max (maxℕ i (j +ℕ k)) ((j +ℕ j2) ⊕ (Max k2 l))
  ⇝ Max (maxℕ (maxℕ i (j +ℕ k)) (j +ℕ j2 +ℕ k2)) (mapPlus (j +ℕ j2) l)
 ⇝i Max i (j ⊕ (Max (maxℕ k (j2 +ℕ k2)) (mapPlus j2 l)))
  ⇝ Max (maxℕ i (j +ℕ (maxℕ k (j2 +ℕ k2)))) (mapPlus j (mapPlus j2 l))
```

But this is not an issue, since we are only interested in reducts of elements of the syntax, meaning that all the variables are of type $\mathbb{L}$.

▶ **Proposition 18.** *The absence of variable of type* $\mathbb{N}$ *or* `LvlSet` *ensures the uniqueness of normal form (modulo AC) property.*

Proof. Since there are no variables of type $\mathbb{N}$ and `LSet`, the function $\texttt{max}_{\mathbb{N}}$, $+_{\mathbb{N}}$ and `mapPlus` are fully defined and cannot occur in the normal forms.

Hence, normal forms contain only $0_{\mathbb{N}}$, $\texttt{s}_{\mathbb{N}}$, `Max`, $\emptyset$, $\oplus$ and $\cup$. Among it, the only constructor of a $\mathbb{L}$ is `Max`, hence every level is either a variable or headed by `Max`.

If it contains a `Max`, there is one at the head. Hence the terms are of the form `Max n s` with `n` a closed natural and `s` a `LSet`. If there are more than one `Max`, it means that the `LSet` contains a level which is not a variable. This one, is headed by `Max`, so one of the rewrite rule regarding the interaction between `Max` and $\oplus$ can be applied.

Hence all normal forms are either a variable or of the form `Max n s`, with `n` closed natural and `s` a `LSet` where all levels are variable. The non-linear rule ensures us that the variables are all distinct.

One can check that the invariant that every natural which is the first argument of a $\oplus$ is smaller or equal to the first argument of the `Max` directly above the $\oplus$ is preserved by every rule and verified by the reducts of the syntax.

So, we can conclude that the normal forms have the shape announced.

To check that a term cannot have two distinct normal forms, the definition of the interpretation is extended to the symbols we introduced and one can verify that all the rules preserve the interpretation and that all the terms of the shape we decribed have a different interpretation. ◀

4 Eta-conversion

Many proof assistants implement, among other conversion rules, the η rule, which state that if f is a function, $f \equiv_\eta \lambda x.f\,x$.

At first sight, this conversion might look quite harmless, and one can hope to just add the corresponding rewrite rule. However, this conversion is an important issue for translation of systems in Dedukti. Indeed, the contraction rule cannot be stated, since $\lambda x.f\,x$ is not a Miller pattern: It requires to match on the fact that $f\,x$ is an application, which would be “meta-matching” and is not in the definition of λΠ-modulo rewriting. Furthermore, we could replace it by $\lambda x.f[x]$, but f is not a valid right-hand side anymore, since it is of arity one. On the other hand, to preserve typing, the expansion rule has to match on the type of a variable, and is not syntax-directed anymore.

Another natural solution could be to define $\lambda\Pi$-modulo rewriting as a logical framework with η hard-coded in the conversion (just like β is). But this is a path *logical frameworks* want to avoid. Indeed, if η is hard-coded, it is impossible to have a shallow encoding of the λ-calculus without η-conversion.

One could expect that η-expanding every term during the translation phase, could allow us to completely ignore η-conversion in the $\lambda\Pi$-calculus modulo rewriting. Indeed, with dependent types it might happen than an η-long term has a non-η-long type. A situation that often breaks the *type preservation of the translation*.

▶ **Example 19.** To illustrate this, we start by defining a type, whose number of arrows depends on a natural number, with a constructor for this type.

```
symbol D : (x : ℕ) ⇒ TYPE.        constant d : (x : ℕ) ⇒ D x.
D 0 ⟶ ℕ.                          D (s x) ⟶ ℕ ⇒ D x.
```

We then define a new type depending on the first one and its constructor.

```
symbol E : (x : ℕ) ⇒ D x ⇒ TYPE.     symbol e : (x : ℕ) ⇒ E x (d x).
```

Now, the term `e 1` is η-long and has type `E 1 (d 1)`, but not `E 1 (λ x, d 1 x)` which is the η-long form of the type.

To overcome this issue, we propose to postpone η-expansion, until the type is fully instantiated. For this, we introduce in the translation a symbol ηE, which purpose is to tag with their types the subterms which may become η-expandable. Then some rewrite rules pattern match on this type annotation to decide when and how the expansion can be performed.

▶ **Definition 20** (Eta-expansion rewrite rules). *ηE annotates terms with their types, to do so, it takes as arguments a sort, a code of type in this sort and the term to annotate. The rules state that η-expansion is the identity for inhabitant of sorts (ηS), and genesrates λ's for inhabitants of products (ηP). Furthermore, a rule state that η-expansion is an idempotent operation (ηI).*

```
symbol ηE : (s : Sort) ⇒ (A : Univ s) ⇒ Term s A ⇒ Term s A.
"ηS" ηE _ (code _)         t ⟶ t.
"ηP" ηE _ (prod a b A B) t ⟶
          λ (x : Term a A), ηE b (B (ηE a A x)) (t (ηE a A x)).
"ηI" ηE _ _ (ηE a A t)       ⟶ ηE a A t.
```

To prove that adding those annotations in the encoding enriches enough the conversion to simulate η-equality, we will also add those annotations in the system we are translating, just like what is done in [12, 11].

For sake of readibility, we will study in this section, terms typed in a full PTS embeddable in $\mathcal{C}^\infty$, like $\mathcal{P}^\infty$ and $\mathcal{C}^\infty$ defined in Expl. 4, in order to directly reuse the induction principle defined in [4].

Performing η-expansion can be required for variables or if an application instantiated a type, allowing it to reduce to a product. Hence, we will add those tags on the variable and application rules. Hence, one could imagine having the rules:

$$\text{(var')} \quad \frac{\Gamma \vdash A : s_i}{\Gamma, x : A \vdash x^A : A}\ x \notin \mathrm{dom}(\Gamma) \qquad \text{(app')} \quad \frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\,u)^{B[^u/_x]} : B\,[^u/_x]}$$

But those rules, do not have the property that if a term is well-type, its subterms are well-typed with a smaller tree, because of the substitution performed on B. Fortunately, the induction principle defined by Barthe, Hatcliff and Sørensen [4] ensures us that, if we annotate the applications with normal form, this property is verified, leading to:

$$\text{(app")}\quad \frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\,u)^{B[u/x]\downarrow} : B\,[u/x]}$$

One must note here that the same tags can be added to the universe polymorph version of the full PTS considered. Indeed, Prop. 10 ensures us that the set of typable terms are the same in both systems. However, it would require to annotate the $x[l_1, \ldots, l_n]$, generating an overweight in the proof, without introducing technicality.

▶ **Definition 21** (Translation). *Given an annotated well-typed term t in a Full Pure Type System, with the rules (var') and (app'') and the conversion enriched with η, we translate t by:* $\left\|x^A\right\| = \eta E\ |s_A|_S\ \|A\|\ \mathtt{x}$; $\quad \|s\| = \mathtt{code}\ |s|_S$; $\quad \left\|(t\,u)^A\right\| = \eta E\ |s_A|_S\ \|A\|\ (\|t\|\ \|u\|)$;

$\left\|\lambda x^A.t\right\| = \lambda(\mathtt{x}\ :\ \mathtt{Term}\ \ |s_A|_S\ \|A\|).\|t\|$;

$\|(x : A) \to B\| = \mathtt{prod}\,|s_A|_S\ |s_B|_S\ \|A\|\ (\lambda\mathtt{x}\ :\ \mathtt{Term}\ |s_1|_S\ \|A\|.\|B\|)$;

s_A and s_B are respectively the sorts of A and B, and $|.|_S$ is the translation of sorts.

The correctness of our translation relies on the preservation of conversion. This result comes from the three following lemmas:

▶ **Lemma 22** (No ηE on translation). *If $\Gamma \vdash t : A$, then $\eta E\ |s_A|_S\ \|A{\downarrow}\|\ \|t\| \leftrightsquigarrow^* \|t\|$.*

▶ **Lemma 23** (Substitution). *If t is well-typed in the context $\Gamma, x_1 : A_1, \ldots, x_n : A_n, \Gamma'$ and if $\Gamma \vdash u_1 : A_1, \ldots, \Gamma \vdash u_n : A_n$ then* $\|t\| \left[\|u_i\|/x_i\right]_{i\in\{1,\ldots,n\}} \leftrightsquigarrow^* \left\|t\left[u_i/x_i\right]_{i\in\{1,\ldots,n\}}\right\|$.

▶ **Lemma 24** (Reduction). *If $\Gamma \vdash t : A$ and $t \rightsquigarrow u$, then $\|t\| \leftrightsquigarrow^* \|u\|$.*

We prove those three lemmas, in this order, by a mutual induction on the combination of the subterm ordering and reduction on a multiset of terms (this multiset is of size at most 2), called "measure" in the proofs.

Proof of Lem. 22. We use $\{\!|t|\!\}$ as the measure. If the normal form of A is a sort, then one can conclude using the rule ηS. We proceed by case on t for the remaining cases:

- If $t = x^B$, then $\eta E\ |s_A|_S\ \|A{\downarrow}\|\ \|t\| = \eta E\ |s_A|_S\ \|A{\downarrow}\|\ (\eta E\ |s_B|_S\ \|B\|\ x) \rightsquigarrow_{\eta I} \|t\|$.
- If $t = (u\,v)^B$, then it is again a direct consequence of the rule ηI
- If $t = \lambda x_1^{B_1} \ldots \lambda x_n^{B_n}.u$, with u not a λ-abstraction.
  There is a C such that: $A{\downarrow} = (x_1 : B_1{\downarrow}) \to \cdots \to (x_n : B_n{\downarrow}) \to C$. We denote by s_i the sort of $(x_i : B_i{\downarrow}) \to \cdots \to (x_n : B_n{\downarrow}) \to C$. We have:

$$
\begin{aligned}
&\eta E\ |s_A|_S\ \|A{\downarrow}\|\ \|t\| \\
&\quad = \eta E\ |s_A|_S\ (\texttt{prod}\ |s_{B_1}|_S\ |s_2|_S\ \|B_1{\downarrow}\|\ (\lambda(x_1 : \texttt{Term}\ |s_{B_1}|_S\ \|B_1{\downarrow}\|). \\
&\qquad \texttt{prod} \ldots |s_{B_n}|_S\ |s_C|_S\ \|B_n{\downarrow}\|\ (\lambda(x_n : \texttt{Term}\ |s_{B_n}|_S\ \|B_n{\downarrow}\|).\ \|C\|) \ldots)) \\
&\qquad (\lambda(x_1 : \texttt{Term}\ |s_{B_1}|_S\ \|B_1\|) \ldots \lambda(x_n : \texttt{Term}\ |s_{B_n}|_S\ \|B_n\|).\ \|u\|) \\
&\quad \rightsquigarrow_{\eta P}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|).\eta E\ |s_2|_S\ ((\lambda \ldots \|C\|)(\eta E\ |s_1|_S\ \|B_1{\downarrow}\|\ x_1)) \\
&\qquad ((\lambda x_1 \ldots \|u\|)(\eta E\ |s_1|_S\ \|B_1{\downarrow}\|\ x_1)) \\
&\quad \rightsquigarrow_\beta^2\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|).\eta E\ |s_2|_S\ (\texttt{prod}\ |s_{B_2}|_S\ |s_3|_S\ \|B_2{\downarrow}\| \ldots \|C\|)\,\sigma\,(\lambda x_2 \ldots \|u\|)\sigma \\
&\quad \text{with } \sigma = \left[\eta E\ |s_1|_S\ \|B_1{\downarrow}\|\ x_1/_{x_1}\right] \\
&\quad (\rightsquigarrow_{\eta P}\rightsquigarrow_\beta^2)^{n-1}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|) \ldots \lambda(x_n : \texttt{Term}\ |s_n|_S\ \|B_n{\downarrow}\|).\eta E\ |s_C|_S\ \|C\|\,\tau\ \|u\|\,\tau \\
&\quad \text{with } \tau = \left[\eta E\ |s_i|_S\ \|B_i{\downarrow}\|\ x_i/_{x_i}\right]_{i\in\{1,\ldots,n\}} \\
&\quad \leftrightsquigarrow^*_{Lem.23}\ \lambda(x_1 : \texttt{Term}\ |s_1|_S\ \|B_1{\downarrow}\|) \ldots \lambda(x_n : \texttt{Term}\ |s_n|_S\ \|B_n{\downarrow}\|).\eta E\ |s_C|_S\ \|C\tau'\|\ \|u\tau'\| \\
&\quad \text{with } \tau' = \left[x_i^{B_i\downarrow}/_{x_i}\right]_{i\in\{1,\ldots,n\}} \\
&\quad \leftrightsquigarrow^*_{IH}\ \left\|\lambda x_1^{B_1} \ldots \lambda x_n^{B_n}.u\right\|
\end{aligned}
$$ ◀

Proof of Lem. 23. There, the measure is $\{\!|t, t\left[u_i/_{x_i}\right]_{i\in\{1,\ldots,n\}}|\!\}$. Depending on the shape of t, we have:

- If t is a sort, the substitution does not have any impact.
- If $t = x_i^{A_i}$, $\|t\| = \eta E\ |s_{A_i}|_S\ \|A_i\|\ x_i$, so $\|t\|\left[\|u_i\|/_{x_i}\right]_i = \eta E\ |s_{A_i}|_S\ \|A_i\|\ \|u_i\|$. By Lem. 24, $\|A_i\| \leftrightsquigarrow^* \|A_i{\downarrow}\|$ and one can conclude by Lem. 22 that $\|t\|\left[\|u_i\|/_{x_i}\right]_i \leftrightsquigarrow^* \|u_i\|$.
- If $t = y^B$ with $y \notin \{x_i\}_i$, then $\|t\| = \eta E\ |s_B|_S\ \|B\|\ y$,so

$$\|t\|\left[\|u_i\|/_{x_i}\right]_i = \eta E\ |s_B|_S\ \|B\|\left[\|u_i\|/_{x_i}\right]_i\ y \leftrightsquigarrow^*_{IH} \left\|y^{B\left[u_i/_{x_i}\right]_i}\right\| = \left\|t\left[u_i/_{x_i}\right]_i\right\|.$$

- If $t = \lambda y^B.v$, then $\|t\| = \lambda(y : \texttt{Term}\ |s_B|_S\ \|B\|).\ \|v\|$, so

$$
\begin{aligned}
\|t\|\left[\|u_i\|/_{x_i}\right]_i &= \lambda(y : \texttt{Term}\ |s_B|_S\ \|B\|\left[\|u_i\|/_{x_i}\right]_i).\ \|v\|\left[\|u_i\|/_{x_i}\right]_i \\
&\leftrightsquigarrow^*_{IH} \lambda(y : \texttt{Term}\ |s_B|_S\ \left\|B\left[u_i/_{x_i}\right]_i\right\|).\ \left\|v\left[u_i/_{x_i}\right]_i\right\| = \left\|(\lambda y^B.v)\left[u_i/_{x_i}\right]_i\right\|
\end{aligned}
$$

The other cases are straightforward, just like the previous two. ◀

Proof of Lem. 24. We use $\{\!|t|\!\}$ as the measure. If the reduction is not at the head of t, then the result follows by the induction hypothesis.

Otherwise, the reduction occurs at the head of the term. It can be either η or β reduction.

(η) Then $t = \lambda x^A.(u\,x^A)^B$ and u is either a variable, an application or a λ-abstraction. In every case $\|t\| = \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\|u\|\ (\eta E\ |s_A|_S\ \|A\|\ x))$.

- If $u = y^C$, then $C{\downarrow} = (x : A{\downarrow}) \to B$.

$$
\begin{aligned}
\|u\| = \eta E\ |s_C|_S\ \|C\|\ y &\leftrightsquigarrow^*_{IH} \eta E\ |s_C|_S\ \|(x : A{\downarrow}) \to B\|\ y \\
&= \eta E\ |s_C|_S\ (\texttt{prod}\ |s_A|_S\ |s_B|_S\ \|A{\downarrow}\|\ (\lambda(x : \texttt{Term}\ |s_A|_S\ \|A{\downarrow}\|).\ \|B\|))\,y \\
&\rightsquigarrow_{\eta P} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A{\downarrow}\|).\eta E\ |s_B|_S\ \|B\|\ (y\,(\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x))
\end{aligned}
$$

When we instantiate $\|t\|$ in this case, we get:

$$\|t\| \rightsquigarrow_\beta \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|$$
$$(\eta E\ |s_B|_S\ \|B\| \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right] (y\,(\eta E\ |s_A|_S\ \|A{\downarrow}\|\ (\eta E\ |s_A|_S\ \|A\|\ x))))$$
$$\rightsquigarrow_{\eta I} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (y\,(\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x)) \leftrightsquigarrow^*_{IH}\ \|u\|$$

- If $u = (v\, w)^{(x:A\downarrow)\to B}$.

$$\|u\| = \eta E\ |C|_S\ \|(x : A{\downarrow}) \to B\|\ (\|v\|\ \|w\|)$$
$$\rightsquigarrow_{\eta P} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A{\downarrow}\|).\eta E\ |s_B|_S\ \|B\|\ (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x))$$

Instantiating $\|t\|$ in this case give:

$$\|t\| \rightsquigarrow_\beta\ \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\eta E\ |s_B|_S\ \|B\| \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right]$$
$$(\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A{\downarrow}\|\ (\eta E\ |s_A|_S\ \|A\|\ x))))$$

Since v and w do not contain x free.

$$\rightsquigarrow_{\eta I}\ \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A{\downarrow}\|\ x)) \leftrightsquigarrow^*_{IH}\ \|u\|$$

- If $u = \lambda y^C.v$, then $C{\downarrow} = A{\downarrow}$, then $\|u\| = \lambda(y : \texttt{Term}\ |s_C|_S\ \|C\|).\ \|v\|$. Then,

$$\|t\| \rightsquigarrow_\beta \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \|v\| \left[(\eta E\ |s_A|_S\ \|A\|\ x)/_y\right]$$
$$\leftrightsquigarrow^*_{Lem.23} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \left\|v \left[x/_y\right]\right\|$$

$(\lambda y.v)\, x$ is a subterm of t.

$$\leftrightsquigarrow^*_{Lem.22} \lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\ \left\|v \left[x/_y\right]\right\| =_\alpha \|u\|$$

(β) Then $t = ((\lambda x^A.v)\, w)^B$ and $u = v\,[w/_x]$. We have :

$$\|t\| = \eta E\ |s_B|_S\ \|B\|\ ((\lambda(x : \texttt{Term}\ |s_A|_S\ \|A\|).\ \|v\|)\ \|w\|)$$
$$\rightsquigarrow_\beta \eta E\ |s_B|_S\ \|B\|\ \|v\| \left[\|w\|/_x\right]$$
$$\leftrightsquigarrow^*_{Lem.23} \eta E\ |s_B|_S\ \|B\|\ \|v\,[w/_x]\| \leftrightsquigarrow^*_{Lem.22}\ \|v\,[w/_x]\|$$

v and $v\,[w/_x]$ are respectively subterm and reduct of t, hence Lem. 23 applies. ◀

From those three lemmas, one can conclude that

▶ **Theorem 25** (Correctness of the translation). *If $\Gamma \vdash t : A$ and $t \leftrightsquigarrow^* u$, then $\|t\| \leftrightsquigarrow^* \|u\|$.*

5 Implementation

Agda [18, 17] is a dependently-typed programming languages, based on an extension of Martin-Löf type theory, Luo's Unifying Theory of dependent Types [15, Chapter 9], which features both universe polymorphism and η-conversion. Dedukti [10, 2] is an implementation of the $\lambda\Pi$-calculus modulo rewriting, which was recently enriched with conversion modulo associativity and commutativity.

Developping a prototypical translator [7] from Agda to Dedukti allowed the author to give a concrete application to the ideas presented in Sections 3 and 4.

However, Agda offers its users a logic much richer than a universe polymorphic pure type system with η-conversion. First of all, Agda permits to declare inductive types and then to define functions using dependent pattern-matching on the constructors of this type. This behaviour can easily be replicated in Dedukti, by declaring new symbols for inductive types, constructors and functions and rewrite rules for each case of the dependent pattern-matching. Just like sorts and products have an encoded and a decoded version, linked by the application of the function `Term`, the type has two translation, one as code and one decoded, linked by a rewrite rule enriching the definition of `Term`. Analogously, one rewrite rule is added to enrich the definition of ηE.

▶ **Example 26.** The Agda declaration of the addition of natural numbers:

```
data Nat : Set where                    _+_ : Nat → Nat → Nat
  zero : Nat                            zero   + m = m
  suc  : (n : Nat) → Nat                suc n + m = suc (n + m)
```

is translated in Dedukti by:

```
constant TYPE__Nat : TYPE.              constant Nat : Univ (set 0).
Term _ Nat ⟶ TYPE__Nat.                 ηE _ Nat t ⟶ t.
constant Nat__zero: Term (set 0) Nat.
constant Nat__suc: Term (set 0) (prod (set 0) (set 0) Nat (λ n, Nat)).
symbol {|_+_|} : Term (set 0) (prod (set 0) (set 0) Nat
                  (λ _0, prod (set 0) (set 0) Nat (λ _1, Nat))).
{|_+_|} Nat__zero    m ⟶ m.
{|_+_|} (Nat__suc n) m ⟶ Nat__suc ({|_+_|} n m).
```

We can observe, that `Nat` in Agda became `TYPE__Nat` and `Nat` in Dedukti, and two rules have been added: one to state that `TYPE__Nat` is the decoding of `Nat` and the other to extend the definition of ηE.

Each declaration of a new type consists in adding a new constructor to the type `Univ s`. The new rules on ηE and `Term` are here to ensure that the pattern-matching on this type remains exhaustive, in order to completely get rid of administrative encoding operators on the normal forms of values.

One can note, that the enrichment of the functions `Term` and ηE are left to the will of the author of the translation. This proves to be a good feature, since the η-conversion of Agda does not restrict to product types, but also concerns records (η-conversion of records is also sometimes called "surjective pairing" and means that if t lives in $\sum_{x:A} B$, then t and $(\mathrm{fst}\, t, \mathrm{snd}\, t)$ are convertible). This does not require to introduce a new symbol for this enrichment of the conversion, but just to define adequate rules on ηE.

▶ **Example 27.** The declaration of this record:

```
record r : Set1 where                   constructor cons
  field A : Set                         field b : A
```

is translated by:

```
constant TYPE__r : TYPE.              constant r : Univ (set (s 0)).
Term _ r ⟶ TYPE__r.
ηE _ r y ⟶ r__cons (r__A y) (ηE 0 (r__A y) (r__b y)).
constant r__cons : Term (set (s 0)) (prod (set (s 0)) (set (s 0))
            (code (set 0)) (λ A, prod (set 0) (set (s 0)) A (λ b, r))).
symbol r__A : Term (set (s 0))
               (prod (set (s 0)) (set (s 0)) r (λ r, code (set 0))).
symbol r__b : Term (set (s 0))
               (prod (set (s 0)) (set 0) r (λ r, r__A r)).
r__A (r__cons A b) ⟶ A.        r__b (r__cons A b) ⟶ ηE 0 A b.
```

The rule to define the η-expansion of an element of `r` states that if y is of type `r`, then $y \equiv \{a = y.a; b = y.b\}$.

This translator is available at `https://github.com/Deducteam/Agda2Dedukti`, the directory `theory/` contains the encoding presented in Sections 3 and 4. It is able to translate and type-check 162 files of AGDA's standard library [9].

6 Conclusion and Future Work

We presented in this article a correct encoding of universe polymorphism in $\lambda\Pi$-modulo rewriting, meaning that every term typable in the original system is translated to a typable term. We also presented a rewrite system to decide equality in the max-plus algebra, which is a comon universe algebra.

Furthermore, we proposed an operator ηE to encode shallowly a type-directed rule, like η-conversion, since the translation of an application really involves the application of the translation of a term to the other one, reducing the interleaving between the computation steps coming from the original system and the steps related to the encoding.

Finally, we applied those results to the practical case of the translation of the proof system AGDA, which offers, among others, the features we targeted, allowing us to provide DEDUKTI users with more than 500 declarations of types, constructors or functions, originating from AGDA's standard library.

We proved that translation of well-typed terms remain typable in our encoding. However, it could be that our encoding is over-permissive and type-checks much more terms than the original system. Hence, one could envision a conservativity theorem, stating that if the translation of a type is inhabited, then the type is also inhabited in the original system. For implementability purposes, we have chosen an encoding with finitely many symbols. Such a theorem has only been proved [8, 1], for encodings of PTS with as many symbols as sorts, axioms and rules. Extending those theorems to our setting is a short-term goal.

Regarding the implementation, making the translator more complete is naturally an objective, however, it involves more theoretical problems, which are long run research programs. For instance, how size types or co-inductive types can be encoded in the $\lambda\Pi$-calculus modulo rewriting is not known yet.

Now that proofs have been translated to the logical framework DEDUKTI, they can be analysed, and (when it is possible) exported to other proof assistants, like what was done with proofs originating from the arithmetic library of MATITA [20].

Acknowledgments I am grateful to Jesper Cockx and Andreas Abel, who received me in Chalmers and helped me to find my way in the hostile jungle the Agda implementation would have been without their help. Then I would like to thank Frédéric Blanqui, Olivier Hermant, Kostia Chardonnet and the anonymous reviewers for their thorough comments, both on the form and content of this article, which greatly improved its quality.

References

1 Ali Assaf. *A Framework for Defining Computational Higher-Order Logics*. PhD thesis, École polytechnique, France, 2015.

2 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory. 2019.

3 Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

4 Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. An induction principle for pure type systems. *Theoretical Computer Science*, 266(1-2):773–818, 2001.

5 Stefano Berardi. *Type Dependence and Constructive Mathematics*. Phd, Carnegie Mellon University, Dept. Comp. Sci., Pittsburgh, Pennsylvania, USA and Torino University, Dipartimento di Informatica, Torino, Italy, 1990.

6 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The $\lambda\Pi$-calculus Modulo as a Universal Proof Language. *PxTP*, page 16, 2012.

7 Jesper Cockx and Guillaume Genestier. Agda2dedukti. `https://github.com/Deducteam/Agda2Dedukti`, 2019.

8 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007.

9 Nils Anders Danielsson, Matthew Daggitt, and Guillaume Allais. Agda standard library. `https://github.com/agda/agda-stdlib`, 2010-.

10 Deducteam. Dedukti. `https://deducteam.github.io/`, 2011-.

11 Gilles Dowek, Gérard Huet, and Benjamin Werner. On the Definition of the Eta-long Normal Form in Type Systems of the Cube. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 115–130, 1993.

12 Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

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

14 Robert Harper and Randy Pollack. Type Checking with Universes. *Theoretical Computer Science*, 89:107–136, 1991.

15 Zhaohui Luo. *Computation and reasoning - a type theory for computer science*, volume 11 of *International series of monographs on computer science*. Oxford University Press, 1994.

16 Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.*, 1(4):497–536, 1991.

17 Ulf Norell. *Towards a practical programming language based on dependent type theory*. Phd, Chalmers University of Technology, Gothenburg, Sweden, 2007.

18 Ulf Norell, Andreas Abel, Niels Anders Danielsson, Makoto Takeyama, and Catarina Coquand. Agda. `https://github.com/agda/agda`, 2007-, v1.0 : 1999.

19 Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 499–514. Springer, 2014.

20 François Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP*, pages 57–71, 2018.