Translating proofs from an impredicative type system to a predicative one

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

Frédéric Blanqui
Université Paris-Saclay, INRIA project, Deducteam, Laboratoire Méthodes Formelles, ENS Paris-Saclay, 91190 France

Ashish Kumar Barnawal
Indian Institute of Technology Guwahati, Guwahati 781039, Assam, India

Abstract

As the development of formal proofs is a time-consuming task, it is important to devise ways of sharing the already written proofs to prevent wasting time redoing them. One of the challenges in this domain is to translate proofs written in proof assistants based on impredicative logics, such as Coq, Matita and the HOL family, to proof assistants based on predicative logics like Agda, whenever impredicativity is not used in an essential way.

In this paper we present an algorithm to do such a translation between a core impredicative type system and a core predicative one allowing prenex universe polymorphism like in Agda. It consists in trying to turn a potentially impredicative term into a universe polymorphic term as general as possible. The use of universe polymorphism is indeed essential since mapping an impredicative universe to a fixed predicative one is not sufficient in most cases.

During the algorithm, we need to solve unification problems modulo the max-successor algebra on universe levels. But, in this algebra, there are solvable problems having no most general solution. We however provide an algorithm whose solutions, when it succeeds, are most general ones.

The proposed translation is of course partial, but in practice allows one to translate many proofs that do not use impredicativity in an essential way. Indeed, it was implemented in the tool Predicativize and then used to translate semi-automatically many non-trivial developments from Matita's arithmetic library to Agda, including Bertrand's Postulate and Fermat's Little Theorem, which were not available in Agda yet.

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

Keywords and phrases type theory, impredicativity, predicativity, proof translation, universe polymorphism, unification modulo max, Agda, Dedukti

Digital Object Identifier 10.4230/LIPIcs.CSL.2023.?

Supplementary Material `https://github.com/Deducteam/predicativize`

Acknowledgements The authors would like to thank François Thiré for the help while developing Predicativize, and Gilles Dowek for the helpful discussions on this paper.

1 Introduction

One of the most important achievements of the research community in logic is the invention of proof assistants. Such tools allow for interactively writing proofs, which are then checked automatically and can then be reused in other developments. Unfortunately, a proof written in a proof assistant cannot be reused in another one, which makes each tool isolated in its own library of proofs. This is specially the case when considering two proof assistants with incompatible logics, as in this case simply translating from one syntax to another would not

work. Therefore, in order to share proofs between systems it is very often required to do logical transformations.

One approach to share proofs from a proof assistant A to a proof assistant B is to define a transformation acting directly on the syntax of A and then implement it using the codebase of A. However, this code would be highly dependent on the implementation of A and can easily become outdated if the codebase of A evolves. Moreover, if there is another proof assistant A' whose logic is very similar to the one of A then this transformation would have to be implemented another time in order to be used with A'.

Dedukti

The logical framework Dedukti [3] is a good candidate for a system where multiple logics can be encoded, allowing for logical transformations to be defined uniformly *inside* Dedukti.

Indeed, first, the framework was already shown to be sufficiently expressive to encode the logics of many proof assistants [7]. Moreover, previous works have shown how proofs can be transformed inside Dedukti. For instance, Thiré describes in [18] a transformation to translate a proof of Fermat's Little Theorem from the Calculus of Inductive Constructions to Higher Order Logic (HOL), which can then be exported to multiple proof assistants such as HOL, PVS, Lean, etc. Géran also used Dedukti to export the formalization of Euclid's Elements Book 1 in Coq [4] to several proof assistants [13].

(Im)Predicativity

One of the challenges in proof interoperability is sharing proofs coming from impredicative proof assistants (the majority of them) to predicative ones such as Agda. Indeed, impredicativity, which is the ability in a logic to quantify over arbitrary entities, regardless of size considerations, is incompatible with predicative systems, in which each entity can only quantify over smaller ones.

Therefore, it is clear that any proof that uses such characteristic in an essential way cannot be translated to a predicative system. Nevertheless, one can wonder if most proofs written in impredicative systems really need impredicativity and, if not, how one could devise a way for detecting and translating them to predicative systems.

Our contribution

In this paper, we tackle this problem by proposing an algorithm that tries to do precisely this. This algorithm was implemented on top of the DkCheck type-checker for Dedukti with the tool Predicativize, allowing for the translation of proofs semi-automatically inside Dedukti. These proofs can then be exported to Agda, the main proof assistant based on predicative type theory.

This tool has been used to translate many proofs semi-automatically to Agda, including Matita's arithmetic library. It contains many-non trivial proofs, and in particular a proof of Bertrand's Postulate which was the subject of a whole publication [1] – thanks to our tool, the same hard work did not have to be repeated in order to make it available in Agda.

Outline

We start in Section 2 with an introduction to Dedukti, before moving to Section 3, where we present informally the problems that appear when translating proofs to predicative systems. We then introduce in Section 4 a predicative universe-polymorphic system, which is a subsystem of Agda and is used as the target of the translation. This is followed by Section 5, the main one, where we present our algorithm. Section 6 then proposes an unification algorithm for universe levels, which is used by the predicativization algorithm. We then

introduce the tool Predicativize in Section 7, and describe the translation of Matita's library in Section 8. We end with some remarks in Section 9. The proofs not given in the main body of the article can be found in the appendix.

2 Dedukti

In this work we use Dedukti [3] as the metatheory in order to express the various logic systems and to define our proof transformation. Therefore, we start with a quick introduction to this system. The logical framework Dedukti has the syntax of the λ-calculus with dependent types ($\lambda\Pi$-calculus).

$$A, B, M, N ::= x \mid c \mid \mathbf{Type} \mid \mathbf{Kind} \mid MN \mid \lambda x : A.M \mid \Pi x : A.B$$

Here, c ranges in an infinite set of constants $\mathcal{C}$, and x ranges in an infinite set of variables $\mathcal{V}$. We call a type 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 use the letter $\mathbf{s}$ to refer to either **Type** or **Kind**.

A *context* Γ is a finite sequence of pairs $x : A$. A *signature* Σ is a finite sequence of constants $c : A$ and definitions $c : A := M$. It can be useful to split the signature into a global one Σ and a local one Δ. The global signature holds the definition of the object logic we are working in, whereas the local one holds axioms and definitions inside the logic. For instance, when working with natural numbers in predicate logic we would have $\wedge : Prop \to Prop \to Prop \in \Sigma$, as $\wedge$ is in the definition of predicate logic, but $+ : Nat \to Nat \to Nat \in \Delta$, given that the natural numbers and addition is not part of predicate logic, but can be defined on top of it.

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 $c\ \vec{l} \hookrightarrow r$. Given a signature, we also consider the δ rules allowing for the unfolding of definitions: we have $c \hookrightarrow M \in \delta$ for each $c : A := M$ appearing in the signature. We then denote by $\hookrightarrow_{\mathcal{R}}$ the closure by context and substitution of $\mathcal{R}$, and by $\hookrightarrow_{\delta}$ the closure by context of δ. Finally, we write $\equiv_{\beta\mathcal{R}\delta}$ for the reflexive, symmetric and transitive closure of $\hookrightarrow_{\beta} \cup \hookrightarrow_{\mathcal{R}} \cup \hookrightarrow_{\delta}$.

One very important notion that we will use in this work is that of a *theory*, which is a pair $(\Sigma, \mathcal{R})$ where Σ is a global signature and all constants appearing in $\mathcal{R}$ are declared in Σ. Theories are used to define in Dedukti the object logics in which we work (for instance, predicate logic).

The typing rules for Dedukti are given in Appendix A, along with some basic metaproperties that we use in the subsequent proofs. We remark in particular that the conversion rule of the system allows to exchange types which are equivalent modulo $\equiv_{\beta\mathcal{R}\delta}$.

2.1 Defining Pure Type Systems in Dedukti

We briefly review how Pure Type Systems (PTS) can be defined in Dedukti, given that these are the most used types in Type Theory and that their definition will be very important in the rest of the article. Recall that in PTSs, universes and function types can be specified by a set $\mathcal{S}$ of universes, and two relations $\mathcal{A} \subseteq \mathcal{S}^2$ and $\mathcal{R} \subseteq \mathcal{S}^3$ – which we suppose to be functional relations here, as is usually the case. These specify that, if $(s_1, s_2) \in \mathcal{A}$, then s_1 is of type s_2, and if $(s_1, s_2, s_3) \in \mathcal{R}$ then when $A : s_1$ and $B : s_2$ we have $\Pi x : A.B : s_3$. Given a PTS specification $(\mathcal{S}_{\mathcal{O}}, \mathcal{A}_{\mathcal{O}}, \mathcal{R}_{\mathcal{O}})$, we can define the corresponding PTS with a Dedukti theory $\mathcal{O}$ in the following manner [8].

We first start with the definition of universes. For each universe $s \in \mathcal{S}$, we declare a Dedukti type $U_s : \mathbf{Type}$ holding the types in the universe s. We then also declare a function

$$
\begin{array}{llll}
U_s : \mathbf{Type} & \text{for } s \in \mathcal{S}_\mathcal{O} & u_{s_1} : U_{s_2} & \text{for } (s_1, s_2) \in \mathcal{A}_\mathcal{O} \\
El_s : U_s \to \mathbf{Type} & \text{for } s \in \mathcal{S}_\mathcal{O} & El_{s_2}\ u_{s_1} \hookrightarrow U_{s_1} & \text{for } (s_1, s_2) \in \mathcal{A}_\mathcal{O} \\
\multicolumn{2}{l}{\pi_{s_1,s_2} : \Pi(A : U_{s_1}).(El_{s_1}\ A \to U_{s_2}) \to U_{s_3}} & \multicolumn{2}{l}{\text{for } (s_1, s_2, s_3) \in \mathcal{R}_\mathcal{O}} \\
\multicolumn{2}{l}{El_{s_3}\ (\pi_{s_1,s_2}\ A\ B) \hookrightarrow \Pi x : El_{s_1}\ A.El_{s_2}\ (B\ x)} & \multicolumn{2}{l}{\text{for } (s_1, s_2, s_3) \in \mathcal{R}_\mathcal{O}}
\end{array}
$$

Figure 1 The DEDUKTI theory $\mathcal{O} = (\Sigma_\mathcal{O}, \mathcal{R}_\mathcal{O})$ which defines the PTS specified by $(\mathcal{S}_\mathcal{O}, \mathcal{A}_\mathcal{O}, \mathcal{R}_\mathcal{O})$

symbol $El_s : U_s \to \mathbf{Type}$ mapping each member of U_s to the type of its elements. We might see the elements of U_s as the codes for the types in s, and El_s as the decoding function, mapping a code to its true type.

In order to represent the fact that a universe s_1 is a member of s_2 when $(s_1, s_2) \in \mathcal{A}$ we add the constant $u_{s_1} : U_{s_2}$. However, now the universe s_1 is represented both by $El_{s_2}\ u_{s_1}$ and U_{s_1}. Therefore, we add the rewrite rule $El_{s_2}\ u_{s_1} \hookrightarrow U_{s_1}$, stating that u_{s_1} decodes to U_{s_1}.

Finally, to define dependent functions, for each $(s_1, s_2, s_3) \in \mathcal{R}$ we add a symbol $\pi_{s_1,s_2} : \Pi A : U_{s_1}.(El_{s_1}\ A \to U_{s_2}) \to U_{s_3}$. Intuitively, the type $El_{s_3}\ (\pi_{s_1,s_2}\ A\ (\lambda x.B))$ should hold the functions from $x : El_{s_1}\ A$ to $El_{s_2}\ B$, where x might occur in B. To make this representation explicit, we add a rewrite rule $\pi_{s_1,s_2}\ A\ B \hookrightarrow \Pi x : El_{s_1}\ A.El_{s_2}\ (B\ x)$ – note the use of *Higher-Order Abstract Syntax* (HOAS) here. Because the type of functions from $x : A$ to B is now represented by the framework's function type, the framework's abstraction and application can be used to represent the ones of the encoded system.

On the following, we allow ourselves to write $\pi_{s_1,s_2}\ A\ (\lambda x.B)$ informally as $\pi_{s_1,s_2}\ x : A.B$ in order to improve clarity. When $x \notin FV(B)$, we might also write $A \rightsquigarrow_{s_1,s_2} B$.

3 An informal look at the challenges of proof predicativization

In this informal section we present the problem of proof predicativization and discuss the challenges that arise through the use of examples. Even though the examples might be unrealistic, they showcase real problems we found during our first predicativization attempt, of Fermat's little theorem library in HOL [18] – some of them being already noted in [9].

We first start by defining the theories **I** and **P**, which we will use to represent the core logics of impredicative and predicative proof assistants. These theories are defined as Pure Type Systems as explained in Subsection 2.1 and are described by the specifications bellow. Remember that a universe s is said to be impredicative when it is closed under dependent products indexed by some bigger sort, that is, for some s' with $(s, s') \in \mathcal{A}$ we have $(s', s, s) \in \mathcal{R}$. Therefore, **I** is an impredicative system and **P** is a predicative one.

$$
\begin{array}{ll}
\mathcal{S}_\mathbf{I} = \{*, \Box\} & \mathcal{S}_\mathbf{P} = \mathbb{N} \\
\mathcal{A}_\mathbf{I} = \{(*, \Box)\} & \mathcal{A}_\mathbf{P} = \{(n, n+1) \mid n \in \mathbb{N}\} \\
\mathcal{R}_\mathbf{I} = \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\} & \mathcal{R}_\mathbf{P} = \{(n, m, max\{n, m\}) \mid n, m \in \mathbb{N}\}
\end{array}
$$

In this setting, the problem of proof predicativization consists in defining a transformation such that, given a local signature Δ with $\Sigma_I, \Delta; - \texttt{well-formed}$, allows to translate it to a local signature Δ' with $\Sigma_P, \Delta'; - \texttt{well-formed}$. Stated informally, we would like to translate constants (which represent axioms) and definitions (which also represent proofs) from I into P. Note in particular that such a transformation is not applied to a single term but to a sequence of constants and definitions, which can be related by dependency – this dependency

turns out to be a major issue as we will see. In the following we represent the local signature Δ in a more readable way as a list of declarations **constant** $c : A$ and **definition** $c : A := M$ – this syntax resembles the concrete syntax of Dedukti and Lambdapi.

Now that our basic notions are explained, let us dive into proof predicativization. For our first step, consider a very simple development showing that, for a certain type T in $*$, we can build an element of $T \rightsquigarrow_{*,*} T$ – if $*$ is a universe of propositions, then this is just a proof that T implies itself.

constant $T : U_*$ **definition** $thm_1 : El_* \ (\pi_{\square,*} \ P : u_*.P \rightsquigarrow_{*,*} P) := \lambda P : U_*.\lambda p : El_* \ P.p$

definition $thm_2 : El_* \ (T \rightsquigarrow_{*,*} T) := thm_1 \ T$

To translate this simple development, the first idea that comes to mind is to define a simple syntactic translation: the sort $*$ is mapped to 0 and the sort $\square$ is mapped to 1. This would then yield the following local signature, which is indeed valid in **P**.

constant $T : U_0$ **definition** $thm_1 : El_1 \ (\pi_{1,0} \ P : u_0.P \rightsquigarrow_{0,0} P) := \lambda P : U_0.\lambda p : El_0 \ P.p$

definition $thm_2 : El_0 \ (T \rightsquigarrow_{0,0} T) := thm_1 \ T$

This naive approach however quickly fails when considering other cases. For instance, suppose now that one adds the following definition – once again, if $*$ is a universe of propositions, then this is just a proof of the proposition $(\forall P.P \Rightarrow P) \Rightarrow \forall P.P \Rightarrow P$.

definition $thm_3 : El_* \ ((\pi_{\square,*} \ P : u_*.P \rightsquigarrow_{*,*} P) \rightsquigarrow_{*,*} \pi_{\square,*} \ P : u_*.P \rightsquigarrow_{*,*} P)$
$:= thm_1 \ (\pi_{\square,*} \ P : u_*.P \rightsquigarrow_{*,*} P)$

If we try to perform the same syntactic translation as before, we get the following result:

definition $thm_3 : El_0 \ ((\pi_{1,0} \ P : u_0.P \rightsquigarrow_{0,0} P) \rightsquigarrow_{1,1} \pi_{1,0} \ P : u_0.P \rightsquigarrow_{0,0} P) := thm_1 \ (\pi_{1,0} \ P : u_0.P \rightsquigarrow_{0,0} P)$

However, one can verify that this term is not well typed. Indeed, in the original term one quantifies over all types in $*$ in the term $\pi_{\square,*} \ P : u_*.P \rightsquigarrow_{*,*} P$, and because of impredicativity this term stays at $*$. However, in **P** quantifying over all elements of the universe 0 in $\pi_{1,0} \ P : u_0.P \rightsquigarrow_{0,0} P$ raises the type to the universe 1. As thm_1 expects a term in the universe 0, the term $thm_1 \ (\pi_{1,0} \ P : u_0.P \rightsquigarrow_{0,0} P)$ is not well-typed.

This suggests that impredicativity introduces a kind of *typical ambiguity*, as it allows us to hide in a single universe $*$ all kinds of bigger types which would have to be placed in bigger universes in a predicative setting. Hence, in order to handle cases like this one, which arise a lot in practice, we should not translate every occurrence of $*$ as 0 naively as we did, but try to compute for each occurrence of $*$ some natural number i such that replacing it by i would produce a valid result.

Thankfully, performing such kind of transformations is exactly the goal of Universo [19]. This tool allows one to transport a typing derivations between two PTS specifications.

To understand how this works, let us come back to the previous example. Universo starts here by replacing all sorts by occurrences of I, where I is a fresh metavariable representing a natural number.

constant $T : U_{I_1}$ **definition** $thm_1 : El_{I_2} \ (\pi_{I_3,I_4} \ P : u_{I_5}.P \rightsquigarrow_{I_6,I_7} P) := \lambda P : U_{I_8}.\lambda p : El_{I_9} \ P.p$

definition $thm_2 : El_{I_{10}} \ (T \rightsquigarrow_{I_{11},I_{12}} T) := thm_1 \ T$

definition $thm_3 : El_{I_{13}} \ ((\pi_{I_{14},I_{15}} \ P : u_{I_{16}}.P \rightsquigarrow_{I_{17},I_{18}} P) \rightsquigarrow_{I_{19},I_{20}} \pi_{I_{21},I_{22}} \ P : \ u_{I_{23}}.P \rightsquigarrow_{I_{24},I_{25}} P)$
$:= thm_1 \ (\pi_{I_{26},I_{27}} \ P : u_{I_{28}}.P \rightsquigarrow_{I_{29},I_{30}} P)$

These of course are not valid proofs in **P**, but in the following step Universo typechecks such development and generates constraints in the process. These constraints are then given to a SMT solver, which is used to compute for each metavariable I a natural number so that

the local signature is valid in **P**. For instance, applying UNIVERSO to our previously example would produce the following valid local signature in **P**.

$$\textbf{constant } T : U_1 \qquad \textbf{definition } thm_1 : El_2\ (\pi_{2,1}\ P : u_1.P \rightsquigarrow_{1,1} P)) := \lambda P : U_1.\lambda p : El_1\ P.p$$

$$\textbf{definition } thm_2 : El_1\ (T \rightsquigarrow_{1,1} T) := thm_1\ T$$

$$\textbf{definition } thm_3 : El_1\ ((\pi_{1,0}\ P : u_0.P \rightsquigarrow_{0,0} P) \rightsquigarrow_{1,1} \pi_{1,0}\ P : u_0.P \rightsquigarrow_{0,0} P) := thm_1\ (\pi_{1,0}\ P : u_0.P \rightsquigarrow_{0,0} P)$$

By using UNIVERSO it is possible to go much further than with the naive syntactical translation. Still, this approach also fails when being employed with real libraries. To see the reason, consider the following minimum example, in which one uses an element of $\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P$ two times to build another element of the same type.

$$\textbf{definition } thm_1 : El_*\ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P) := \lambda P : U_*.\lambda p : El_*\ P.p$$

$$\textbf{definition } thm_2 : El_*\ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P) := thm_1\ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P)\ thm_1$$

If we repeat the same procedure as before, we get the following term, which generates unsolvable constraints.

$$\textbf{definition } thm_1 : El_{l_1}\ (\pi_{l_2,l_3}\ P : u_{l_4}.P \rightsquigarrow_{l_5,l_6} P) := \lambda P : U_{l_7}.\lambda p : El_{l_8}\ P.p$$

$$\textbf{definition } thm_2 : El_{l_9}\ (\pi_{l_{10},l_{11}}\ P : u_{l_{12}}.P \rightsquigarrow_{l_{13},l_{14}} P) := thm_1\ (\pi_{l_{15},l_{16}}\ P : u_{l_{17}}.P \rightsquigarrow_{l_{18},l_{19}} P)\ thm_1$$

This happens because the application $thm_1\ (\pi_{l_{15},l_{16}}\ P : u_{l_{17}}.P \rightsquigarrow_{l_{18},l_{19}} P)\ thm_1$ forces l_4 to be both l_{17} and $l_{17}+1$, which is impossible. This example suggests that impredicativity does not only hide the fact that types are stratified, but also the fact that they can be used at any level of this stratification. For instance, in our example we would like to use thm_1 one time with $l_4 = l_{17}$ and another time with $l_4 = l_{17+1}$. In general, when trying to translate libraries using UNIVERSO we found that at very early stages a translated proof or object was already needed at multiple universes at the same time, causing the translation to fail.

Therefore, in order to properly compensate for the lack of impredicativity, we found that the only way to address this is by using *universe polymorphism*, a feature in type theory (and also present in AGDA) that allows defining terms that can later be used at multiple universes [14, 16]. Our translation works by trying to compute for each definition or declaration its most general universe polymorphic type, and use it when translating the subsequent declarations or definitions. To understand how this is done precisely, let us first introduce universe polymorphism, which is the subject of the following section.

4 A Universe-Polymorphic Predicative Type System

In this section, we define the Universe-Polymorphic Predicative Type System (or just **UPP**), which enriches the Predicative PTS **P** with prenex universe polymorphism [14]. This is in particular a subsystem of the one underlying the AGDA proof assistant [17]. As usual, we define this system as a DEDUKTI theory $\mathbf{UPP} = (\Sigma_{UPP}, \mathcal{R}_{UPP})$.

The main change with respect to **P** is that, instead of indexing the constants El_s, U_s, u_s, π_{s_1,s_2} externally, we index them inside the framework [2]. To do this, we first introduce a syntax for *universe levels* inside DEDUKTI. A universe level l is defined as a normal form of the DEDUKTI type *Level*, built with the the constant z, the unary function s, the binary max $\sqcup$, and finally the variables of type *Level*, which we suppose to be taken from a specific set of level variables $\mathcal{I} = \{i, j, ...\}$ in order to distinguish them from the term variables.

$$Level : \mathbf{Type} \qquad \mathsf{s} : Level \to Level$$
$$\mathsf{z} : Level \qquad \sqcup : Level \to Level \to Level \quad \text{(written infix)}$$

The definitions in the theory $\mathbf{P}$ of $El_s, U_s, u_s, \pi_{s_1,s_2}$ and the related rewrite rules are then replaced by the following ones.

$$U : Level \to \mathbf{Type} \qquad \pi : \Pi(i_A\ i_B : Level)\ (A : U\ i_A).(El\ i_A\ A \to U\ i_B) \to U\ (i_A \sqcup i_B)$$
$$El : \Pi i : Level.U\ i \to \mathbf{Type} \qquad El\ i'\ (u\ i) \hookrightarrow U\ i$$
$$u : \Pi i : Level.U\ (\mathtt{s}\ i) \qquad El\ i'\ (\pi\ i_A\ i_B\ A\ B) \hookrightarrow \Pi x : El\ i_A\ A.El\ i_B\ (B\ x)$$

We however still allow ourselves to write $El_l, U_l, u_l, \pi_{l,l'}$ in order to improve clarity. We also reuse the previous convention to write $\pi_{l,l'}\ A\ (\lambda x.B)$ as $\pi_{l,l'}\ x : A.B$, or even $A \rightsquigarrow_{l,l'} B$ when $x \notin FV(B)$.

Now universe polymorphism can be represented directly with the use of the framework's function type. Indeed, if a definition contains free level variables, it can be made universe polymorphic by abstracting over such variables. The following example illustrates this.

▶ **Example 1.** The universe polymorphic identity function is given by

$$id = \lambda(i : Level).\lambda(A : U_i).\lambda(a : El_i\ A).a$$

which has type $\Pi i : Level.El_{(\mathtt{s}\ i)}\ (\pi_{(\mathtt{s}\ i),i}\ A : u_i.A \rightsquigarrow_{i,i} A)$. This then allows to use id at any universe level: for instance, we can obtain the polymorphic identity function at the level $\mathtt{z}$ with the application $id\ \mathtt{z}$, which has type $El_{(\mathtt{s}\ \mathtt{z})}\ (\pi_{(\mathtt{s}\ \mathtt{z}),\mathtt{z}}\ A : u_{\mathtt{z}}.A \rightsquigarrow_{\mathtt{z},\mathtt{z}} A)$.

Finally, in order to finish our definition we need to specify the definition equality satisfied by levels. A (level) valuation is a function $\mathcal{I} \to \mathbb{N}$ giving a natural number to each level variable. Given a valuation σ, one can define the interpretation $[\![l]\!]_\sigma$ of a level l by interpreting the symbols $\mathtt{z}, \mathtt{s}$ and $\sqcup$ in the most obvious way, as zero, successor and max. Like is usually done when dealing with universe-polymorphism, we also consider a conversion relation $\simeq$ which allows one to interchange two universe levels l, l' whenever their interpretations are the same for all valuations. Explicitly, we have $l \simeq l'$ iff for all $\sigma : \mathcal{I} \to \mathbb{N}, [\![l]\!]_\sigma = [\![l']\!]_\sigma$.

We will not detail here how $\simeq$ can be expressed with rewrite rules, and we refer to Appendix B and to [12] for more details. Nevertheless, this will not be needed for understanding the next sections.

Recall that injectivity of dependent products with respect to a relation $\equiv$ states that $\Pi x : A.B \equiv \Pi x : A'.B'$ implies $A \equiv A'$ and $B \equiv B'$. For the Section 5, we assume injectivity of dependent products with respect to $\equiv_{\beta\mathcal{R}_{UPP}\delta}$. While we have still not proved it true, in Appendix B we discuss possible solutions to this, and we also discuss why this property is important to us.

5 The algorithm

We are now ready to define the (partial) translation of a local signature Δ to the theory **UPP**. The idea of the translation is that we traverse the signature Δ and at each step we try to compute the most general universe polymorphic version of a definition or constant. The result of a previously translated definition or declaration can then be used at multiple levels for translating entries occurring later in the signature. In order to understand all the following steps intuitively, we will make use of a running example.

▶ **Example 2.** The last example of Section 3 corresponds to the local signature

$$\Delta_l = thm_1 : El_* \ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P) := \lambda P : U_*.\lambda p : El_*\ P.p\ ,$$
$$thm_2 : El_* \ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P) := thm_1\ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P)\ thm_1$$

which is well-formed in the theory $\mathbf{I}$. Let us suppose that the first entry of the signature has already been translated, giving the following signature Δ_{thm_1}.

$$\Delta_{thm_1} = thm_1 : \Pi i : Level.El_{(\mathtt{s}\ i)}\ (\pi_{(\mathtt{s}\ i),i}\ P : u_i.P \rightsquigarrow_{i,i} P) := \lambda i : Level.\lambda P : U_i.\lambda p : El_i\ P.p$$

Therefore, as a running example, we will translate step by step the second entry thm_2.

Let us start with some basic auxiliary definitions. Given a local signature Δ such that $\Sigma_{UPP}, \Delta; - \ \mathtt{well\text{-}formed}$ and a constant c occurring in Δ, let us define $arity(c)$ as the greatest natural number k such that the type of c is convertible to $\Pi i_1\ ..\ i_k : Level.A$. More informally, it is the number of level arguments that this constant expects. For instance, we have $arity(thm_1) = 1$.

Using this function, let us define $insert\_metas(M)$, by the following equations. This function allows us to insert the fresh level variables that will be used to compute the constraints. We suppose that the inserted variables come from a dedicated subset of level variables $\mathcal{M} \subsetneq \mathcal{I}$ and that each inserted variable is fresh.

$$insert\_metas(El_s) = El_i \qquad insert\_metas(u_s) = u_i$$
$$insert\_metas(U_s) = U_i \qquad insert\_metas(\pi_{s_1,s_2}) = \pi_{i,j}$$
$$insert\_metas(c) = c\ i_1 ... i_k \text{ where } k = arity(c) \text{ and } c \neq El_s, U_s, u_s, \pi_{s_1,s_2}$$
$$insert\_metas(M) = M \text{ if } M \text{ is a variable } x \text{ or } \mathbf{Type} \text{ or } \mathbf{Kind}$$
$$insert\_metas(\Pi x : A.B) = \Pi x : insert\_metas(A).insert\_metas(B)$$
$$insert\_metas(\lambda x : A.M) = \lambda x : insert\_metas(A).insert\_metas(M)$$
$$insert\_metas(MN) = insert\_metas(M)\ insert\_metas(N)$$

▶ **Example 3.** By applying $insert\_metas$ to the type and body of thm_2 we get

$$insert\_metas(El_* \ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P)) = El_{i_1}\ (\pi_{i_2,i_3}\ P : u_{i_4}.P \rightsquigarrow_{i_5,i_6} P)$$
$$insert\_metas(thm_1\ (\pi_{\square,*}\ P : u_*.P \rightsquigarrow_{*,*} P)\ thm_1) = thm_1\ i_7\ (\pi_{i_8,i_9}\ P : u_{i_{10}}.P \rightsquigarrow_{i_{11},i_{12}} P)\ (thm_1\ i_{13})$$

▶ Remark 4. Note that because our first step is erasing the universes that appear in the terms, this translation is defined for all PTS local signatures, and not only those in $\mathbf{I}$. Therefore, it can be applied to proofs coming from systems featuring much more complex universes hierarchies then $\mathbf{I}$, such as the PTS underlying the type systems of Coq and Matita.

Once the fresh level variables are inserted, the next step is to compute the constraints between levels. To do this, we use an approach similar to [14] and define a bidirectional type checking/inference algorithm which also generates constraints, by the rules in Figures 2 and 3. In these rules, we write $\widehat{M}$ for the weak head normal form of M. As usual, $M \Rightarrow A$ denotes type inference, whereas $M \Leftarrow A$ denotes type checking. We also write $M \Rightarrow_{sort} \mathbf{s}$ or $M \Rightarrow_\Pi \Pi x : A.B$ as a shorthand for $M \Rightarrow A'$ and $\widehat{A'} = \mathbf{s}$ or $\widehat{A'} = \Pi x : A.B$ respectively.

$$\frac{}{I \equiv^? I' \downarrow \{I = I'\}} \qquad \frac{M = x, c, \mathbf{Type}, \mathbf{Kind}}{M \equiv^? M \downarrow \varnothing} \qquad \frac{M \equiv^? M' \downarrow C_1 \qquad \widehat{N} \equiv^? \widehat{N'} \downarrow C_2}{MN \equiv^? M'N' \downarrow C_1 \cup C_2}$$

$$\frac{\widehat{A} \equiv^? \widehat{A'} \downarrow C_1 \qquad \widehat{B} \equiv^? \widehat{B'} \downarrow C_2}{\Pi x : A.B \equiv^? \Pi x : A'.B' \downarrow C_1 \cup C_2} \qquad \frac{\widehat{A} \equiv^? \widehat{A'} \downarrow C_1 \qquad \widehat{M} \equiv^? \widehat{M'} \downarrow C_2}{\lambda x : A.M \equiv^? \lambda x : A'.M' \downarrow C_1 \cup C_2}$$

Figure 2 Inference rules for computing constraints for two terms in whnf to be convertible

Intuitively, these judgments define a conditional typing relation that depends on the constraints being satisfied. This intuition is formalized by the following results – we only provide proof sketches, as these are very similar to the ones in [14].

$$\frac{c : A := M \in \Sigma_{UPP}, \Delta \text{ or } c : A \in \Sigma_{UPP}, \Delta}{\Sigma_{UPP}, \Delta; \Gamma \vdash c \Rightarrow A \downarrow \varnothing}\ \textsc{Cons} \qquad \frac{x : A \in \Gamma}{\Sigma_{UPP}, \Delta; \Gamma \vdash x \Rightarrow A \downarrow \varnothing}\ \textsc{Var}$$

$$\frac{i \in \mathcal{M}}{\Sigma_{UPP}, \Delta; \Gamma \vdash i \Rightarrow \mathit{Level} \downarrow \varnothing}\ \textsc{Lvl-Var} \qquad \frac{}{\Sigma_{UPP}, \Delta; \Gamma \vdash \mathbf{Type} \Rightarrow \mathbf{Kind} \downarrow \varnothing}\ \textsc{Sort}$$

$$\frac{\Sigma_{UPP}, \Delta; \Gamma \vdash A \Leftarrow \mathbf{Type} \downarrow C_1 \qquad \Sigma_{UPP}, \Delta; \Gamma, x : A \vdash B \Rightarrow_{sort} \mathbf{s} \downarrow C_2}{\Sigma_{UPP}, \Delta; \Gamma \vdash \Pi x : A.B \Rightarrow \mathbf{s} \downarrow C_1 \cup C_2}\ \textsc{Prod}$$

$$\frac{\Sigma_{UPP}, \Delta; \Gamma \vdash A \Leftarrow \mathbf{Type} \downarrow C_1 \qquad \Sigma_{UPP}, \Delta; \Gamma, x : A \vdash B \Rightarrow_{sort} \mathbf{s} \downarrow C_2 \qquad \Sigma_{UPP}, \Delta; \Gamma, x : A \vdash M \Rightarrow B \downarrow C_3}{\Sigma_{UPP}, \Delta; \Gamma \vdash \lambda x : A.M \Rightarrow \Pi x : A.B \downarrow C_1 \cup C_2 \cup C_3}\ \textsc{Abs}$$

$$\frac{\Sigma_{UPP}, \Delta; \Gamma \vdash M \Rightarrow_{\Pi} \Pi x : A.B \downarrow C_1 \qquad \Sigma_{UPP}, \Delta; \Gamma \vdash N \Leftarrow A \downarrow C_2}{\Sigma_{UPP}, \Delta; \Gamma \vdash MN \Rightarrow B\{N/x\} \downarrow C_1 \cup C_2}\ \textsc{App}$$

$$\frac{\Sigma_{UPP}, \Delta; \Gamma \vdash M \Rightarrow A \downarrow C_1 \qquad \widehat{A} \equiv^{?} \widehat{B} \downarrow C_2}{\Sigma_{UPP}, \Delta; \Gamma \vdash M \Leftarrow B \downarrow C_1 \cup C_2}\ \textsc{Check}$$

Figure 3 Inference rules for computing constraints for a term to be typable

▶ **Definition 5.** *Given a level substitution θ (sending level variables to levels) and a set of constraints C, containing pairs of levels $l = l'$, we write $\theta \vDash C$ when for all $l = l' \in C$, $l\theta \simeq l'\theta$.*

▶ **Lemma 6.** *Let M, N be terms and θ be a level substitution. If $M \equiv^{?} N \downarrow C$ and $\theta \vDash C$ then $M\theta \equiv N\theta$.*

Proof. By induction on $M \equiv^{?} N \downarrow C$. ◀

Let us write $\vec{i}_X$ for the free level variables in X. We also shorten $\vec{i} : \mathit{Level}$ as $\vec{i}$.

▶ **Lemma 7.** *Given a level substitution θ, suppose $\Sigma_{UPP}, \Delta; \vec{i}_{\Gamma\theta}, \Gamma\theta$* `well-formed`.

- *If $\Sigma_{UPP}, \Delta; \Gamma \vdash M \Rightarrow A \downarrow C$ and $\theta \vDash C$ then $\Sigma_{UPP}, \Delta; \vec{i}_{\Gamma\theta} \cup \vec{i}_{M\theta} \cup \vec{i}_{A\theta}, \Gamma\theta \vdash M\theta : A\theta$*
- *If $\Sigma_{UPP}, \Delta; \Gamma \vdash M \Leftarrow A \downarrow C$, $\theta \vDash C$ and $\Sigma_{UPP}, \Delta; \vec{i}_{\Gamma\theta} \cup \vec{i}_{A\theta}, \Gamma\theta \vdash A\theta : \boldsymbol{s}$ then we have $\Sigma_{UPP}, \Delta; \vec{i}_{\Gamma\theta} \cup \vec{i}_{M\theta} \cup \vec{i}_{A\theta}, \Gamma\theta \vdash M\theta : A\theta$*

Proof. By induction on the derivation, using weakening, the substitution lemma, well-sortedness, inversion and subject reduction of $\hookrightarrow_{\beta\mathcal{R}_{UPP}\delta}$. ◀

▶ **Example 8.** We can use rules for constraint computation with our running example. We first calculate $\Sigma_{UPP}, \Delta_{thm_1}; - \vdash El_{i_1}\ (\pi_{i_2,i_3}\ P : u_{i_4}.P \rightsquigarrow_{i_5,i_6} P) \Rightarrow_{sort} \mathbf{s} \downarrow C_1$. Therefore, any substitution θ with $\theta \vDash C_1$ applied to the previous term results in a valid type. We then calculate

$$\Sigma_{UPP}, \Delta_{thm_1}; - \vdash thm_1\ i_7\ (\pi_{i_8,i_9}\ P : u_{i_{10}}.P \rightsquigarrow_{i_{11},i_{12}} P)\ (thm_1\ i_{13}) \Leftarrow El_{i_1}\ (\pi_{i_2,i_3}\ P : u_{i_4}.P \rightsquigarrow_{i_5,i_6} P) \downarrow C_2$$

This gives $C_1 \cup C_2 = \{i_8 = \mathbf{s}\ i_{10}, i_{11} = i_{10}, i_{12} = i_{10}, i_9 = i_{10} \sqcup i_{12}, i_8 \sqcup i_9 = i_7, i_{13} = i_{10}, i_1 = i_2 \sqcup i_3, \mathbf{s}\ i_4 = i_2, i_5 = i_4, i_6 = i_4, i_3 = i_5 \sqcup i_6, i_4 = i_{10}\}$.

Once the constraints are computed the next step is to solve them. However, as explained in Section 3, we do not want a numerical assignment of level variables that satisfies the constraints, but rather a general symbolic solution which allows the term to be instantiated later at different universe levels. This leads us to use equational unification but, as levels are not purely syntactic entities, one needs to devise a unification algorithm for the equational theory $\simeq$. For now, let us postpone this to the next section and assume we are given a (partial) function *unify* which computes from a set of constraints C a unifier θ – that is, a

$$
\begin{aligned}
|-| &= - \\
|\Delta, c : A| &= \text{let } A' = \mathit{insert\_metas}(A) \\
&\quad \text{let } C \text{ be such that } \Sigma_{\mathit{UPP}}, |\Delta|; - \vdash A' \Rightarrow_{\mathit{sort}} \mathbf{s} \downarrow C \text{ else } \mathit{raise}\ \bot \text{ if no such } C \\
&\quad \text{let } \theta = \mathit{unify}(C) \text{ else } \mathit{raise}\ \bot \text{ if no such } \theta \\
&\quad \text{let } \vec{i} = \vec{i}_{A'\theta} \\
&\quad |\Delta|, c : \Pi \vec{i} : \mathit{Level}.A'\theta \\
|\Delta, c : A := M| &= \text{let } A', M' = \mathit{insert\_metas}(A), \mathit{insert\_metas}(M) \\
&\quad \text{let } C_1 \text{ be such that } \Sigma_{\mathit{UPP}}, |\Delta|; - \vdash A' \Rightarrow_{\mathit{sort}} \mathbf{s} \downarrow C_1 \text{ else } \mathit{raise}\ \bot \text{ if no such } C_1 \\
&\quad \text{let } C_2 \text{ be such that } \Sigma_{\mathit{UPP}}, |\Delta|; - \vdash M' \Leftarrow A' \downarrow C_2, \text{ else } \mathit{raise}\ \bot \text{ if no such } C_2 \\
&\quad \text{let } \theta = \mathit{unify}(C_1 \cup C_2) \text{ else } \mathit{raise}\ \bot \text{ if no such } \theta \\
&\quad \text{let } \vec{i} = \vec{i}_{A'\theta} \cup \vec{i}_{M'\theta} \\
&\quad |\Delta|, c : \Pi \vec{i} : \mathit{Level}.A'\theta := \lambda \vec{i} : \mathit{Level}.M'\theta
\end{aligned}
$$

Figure 4 Pseudocode of the predicativization algorithm

substitution satisfying $\theta \vDash C$. We however do not assume that θ is the most general unifier – as we show later in Theorem 12, such a most general unifier might not exist.

After an unifier θ is found, the final step is then to apply it.

▶ **Example 9.** Given the previous computed constraints $C_1 \cup C_2$ we can compute the substitution θ which sends all variables to i_4, except for i_1, i_2, i_7, i_8, which are sent to $\mathbf{s}\ i_4$, and verify that $\theta \vDash C_1 \cup C_2$. By applying θ, and by Lemma 7, we have

$$\Sigma_{\mathit{UPP}}, \Delta_{\mathit{thm}_1}; i_4 \vdash \mathit{thm}_1\ (\mathbf{s}\ i_4)\ (\pi_{(\mathbf{s}\ i_4), i_4}\ P : u_{i_4}.P \rightsquigarrow_{i_4, i_4} P)\ (\mathit{thm}_1\ i_4) : \mathit{El}_{(\mathbf{s}\ i_4)}\ (\pi_{(\mathbf{s}\ i_4), i_4}\ P : u_{i_4}.P \rightsquigarrow_{i_4, i_4} P) \downarrow C_2$$

Note that in this term, the constant thm_1 is used at two different universe levels.

The final algorithm can now be described by the pseudocode in Figure 4. The algorithm might fail at any point when either it is not able to compute the constraints, or if the unification algorithm is not capable of inferring a substitution from the constraints. However, if the algorithm returns, its correctness is guaranteed by the following theorem:

▶ **Theorem 10.** *If* $|\Delta|$ *is defined, then* $\Sigma_{\mathit{UPP}}, |\Delta|; - \texttt{well-formed}$.

Proof. By induction on Δ, the base case being trivial. For the induction step, we have either $\Delta = \Delta'; c : A$ or $\Delta = \Delta'; c : A := M$. In both cases, if $|\Delta|$ is defined, then so is $|\Delta'|$, and thus by induction hypothesis we have $|\Delta'|; - \texttt{well-formed}$. We proceed with a case analysis on the entry.

Definition: As $\Sigma_{\mathit{UPP}}, |\Delta'|; - \vdash A' \Rightarrow T \downarrow C_1$ and $\theta \vDash C_1$, by Lemma 7 we get $\Sigma_{\mathit{UPP}}, |\Delta'|; \vec{i}_{A'\theta} \cup \vec{i}_{T\theta} : \mathit{Level} \vdash A'\theta : T\theta$. Because $\widehat{T} = \mathbf{s}$, we also have $T\theta \hookrightarrow^* \mathbf{s}$, hence by well-sortedness, subject reduction and conversion we get $\Sigma_{\mathit{UPP}}, |\Delta'|; \vec{i}_{A'\theta} \cup \vec{i}_{T\theta} \vdash A'\theta : \mathbf{s}$. By applying the substitution lemma with the substitution sending every variable i in $\vec{i}_{T\theta}$ but not in $\vec{i}_{A'\theta}$ to $\mathbf{z}$, we get $\Sigma_{\mathit{UPP}}, |\Delta'|; \vec{i}_{A'\theta} \vdash A'\theta : \mathbf{s}$. Because we also have $\Sigma_{\mathit{UPP}}, |\Delta'|; - \vdash M' \Leftarrow A' \downarrow C_2$ and $\theta \vDash C_2$, by Lemma 7 again we get $\Sigma_{\mathit{UPP}}, |\Delta'|; \vec{i}_{M'\theta} \cup \vec{i}_{A'\theta} : \mathit{Level} \vdash M'\theta : A'\theta$. By abstracting each free level variable, we get $\Sigma_{\mathit{UPP}}, |\Delta'|; - \vdash \lambda \vec{i} : \mathit{Level}.M'\theta : \Pi \vec{i} : \mathit{Level}.A'\theta$. Hence, we can derive $\Sigma_{\mathit{UPP}}, |\Delta'|, c : \Pi \vec{i} : \mathit{Level}.A'\theta := \lambda \vec{i} : \mathit{Level}.M'\theta; - \texttt{well-formed}$.

Declaration: Similar to the previous case. ◀

▶ Remark 11. One could also wonder if the algorithm always terminates and produces a result (be it a valid signature or $\bot$). By supposing strong normalization for **UPP**, and by checking at each step of the rules in Figure 3 that the constraints are consistent, one could

show termination of the algorithm by using a similar technique as in [14]. As we do not investigate strong normalization of **UPP** in this paper, we leave termination of the algorithm for future work. However, as we will see in Section 8, when using it in practice we were able to translate many proofs without non-termination issues.

Our algorithm relies on an unspecified function *unify* in order to solve the constraints. In order to fully specify it, we thus present an unification algorithm for $\simeq$ in the next section. As we will discuss, the unification algorithm we propose is not guaranteed to always find a most general unifier whenever there is one. However, note that our predicativization algorithm can in principle be used with any unification algorithm for $\simeq$. Therefore, if we have a better unification algorithm in the future, we do not have the modify the algorithm of Figure 4.

6 Solving level constraints

Before addressing the problem of how to solve level constraints, the first natural question that comes to mind is if one can always find a most general unifier (mgu). The following result answers this negatively.

▶ **Theorem 12.** *Not all problems of unification modulo $\simeq$ have most general unifiers.*

Proof. Consider the equation $\mathtt{s}\ i_1 = i_2 \sqcup i_3$ and suppose it had a mgu θ. Note that $\theta_1 = i_1 \mapsto \mathtt{z}, i_2 \mapsto \mathtt{s}\ \mathtt{z}, i_3 \mapsto \mathtt{z}$ is also a solution, thus there is some τ such that $i_3\theta\tau \simeq \mathtt{z}$. Therefore, there can be no occurrence of s in $i_3\theta$. By taking $\theta_2 = i_1 \mapsto \mathtt{z}, i_2 \mapsto \mathtt{z}, i_3 \mapsto \mathtt{s}\ \mathtt{z}$ we can show similarly that there can be no occurrence of $\mathtt{s}$ in $i_2\theta$. But by taking the substitution $\theta' = \_ \mapsto \mathtt{z}$ mapping all variables to $\mathtt{z}$, we get $(i_2 \sqcup i_3)\theta\theta' \simeq \mathtt{z}$, which cannot be equivalent to $(\mathtt{s}\ i_1)\theta\theta'$. Hence, $\mathtt{s}\ i_1 = i_2 \sqcup i_3$ has no mgu. ◀

Therefore, no unification algorithm for $\simeq$ can always produces a mgu. Hence, our algorithm will produce three kinds of results: either it produces a substitution, in which case it is a mgu; or it produces $\bot$, in which case there is no solution to the constraints; or it gets stuck on a set of constraints that it cannot handle. Still, it is not guaranteed to compute a mgu whenever there is one.

Before presenting the algorithm, the first issue we have to address is the fact that levels can have multiple equivalent representations. It would be convenient if we had a syntactical way to compare them. Thankfully, previous works have already addressed this problem:

Let us assume from this point on that level variables in $\mathcal{I}$ admit a total order $\leqslant$. Given a strictly increasing sequence of level variables $V = (i_1, ..., i_k)$, and an V-indexed family of levels $\{l_i\}_{i\in V}$, let $\sqcup_{i\in V} l_i$ denote the term $l_1 \sqcup (l_2 \sqcup ...(l_{k-1} \sqcup l_k)...)$. Moreover, given a natural number k, let $\mathtt{s}^k\ l$ be inductively defined by $\mathtt{s}^0\ l = l$ and $\mathtt{s}^{n+1}\ l = \mathtt{s}\ (\mathtt{s}^n\ l)$.

▶ **Definition 13** (Level normal form)**.** *A level is in normal form when it is of the form $\mathtt{s}^k\ \mathtt{z} \sqcup (\sqcup_{i\in V} \mathtt{s}^{n_i}\ i)$ with $n_i \leqslant k$ for all i.*

Previous works [12, 11, 5] have established that for every level l there is a unique level in normal form, which we refer to as $\widehat{l}$, with $l \simeq \widehat{l}$ – see for instance Lemma 6.2.5 of [11]. We will not describe explicitly here the algorithm for computing normal forms, as this has already been thoroughly explained in previous works, such as in [12, 5].

We also define a notion of normal forms for constraints.

▶ **Definition 14.** *A constraint $l_1 = l_2$ is said to be in normal form if*

1. *Both l_1, l_2 are in normal form – so we write $l_p = \mathtt{s}^{k_p}\ \mathtt{z} \sqcup (\sqcup_{i\in V_p} \mathtt{s}^{n_i^p}\ i)$ for $p = 1, 2$*
2. *If $i \in V_1 \cap V_2$, then $n_i^1 = n_i^2$*

3. *At least one of the numbers in* $\{k_1, k_2\} \cup \{n_i^1\}_{i \in V_1} \cup \{n_i^2\}_{i \in V_2}$ *is equal to* 0

Every constraint can be put in normal form, and for this we can use the algorithm in Figure 5. From the second line on, we use k^p, n_i^p to refer to the indices in the normal forms of l_1, l_2. Moreover, the pseudocode should be read imperatively, in the sense that l_1, l_2, V_1, V_2 are updated at each step.

Given a set of constraints C, let $\widetilde{C}$ denote the result of putting all constraints of C in normal form by applying the algorithm of Figure 5.

let $l_1, l_2 = \widehat{l_1}, \widehat{l_2}$
for each $i \in V_1 \cap V_2$
  if $n_i^1 < n_i^2$ then remove $\mathtt{s}^{n_i^1}\ i$ from l_1
  else if $n_i^1 > n_i^2$ then remove $\mathtt{s}^{n_i^2}\ i$ from l_2
substract the minimum value of the set $\{k_1, k_2\} \cup \{n_i^1\}_{i \in V_1} \cup \{n_i^2\}_{i \in V_2}$ from all of its elements

Figure 5 Imperative algorithm for putting a constraint in normal form

▶ **Lemma 15.** *For all substitutions* θ*, we have* $\theta \vDash C$ *iff* $\theta \vDash \widetilde{C}$*.*

By putting constraints in normal form, we can help our unification algorithm to find a solution, as shown by the following example.

▶ **Example 16.** Consider the constraint $i \sqcup \mathtt{s}\ (i \sqcup \mathtt{s}\ j) = j \sqcup \mathtt{s}\ (\mathtt{s}\ i)$ – which, as we will see, cannot be treated by our unification algorithm if it is not normalized first. By first computing the level normal form of each side, we get $\mathtt{s}^2\ \mathtt{z} \sqcup \mathtt{s}\ i \sqcup \mathtt{s}^2\ j = \mathtt{s}^2\ \mathtt{z} \sqcup \mathtt{s}^2\ i \sqcup j$. As both variables appear in both sides, we remove from each of the sides the occurrence with the smaller index, giving $\mathtt{s}^2\ \mathtt{z} \sqcup \mathtt{s}^2\ j = \mathtt{s}^2\ \mathtt{z} \sqcup \mathtt{s}^2\ i$. Finally, as the minimum among all indices is 2, we subtract this from all of them, giving $\mathtt{z} \sqcup j = \mathtt{z} \sqcup i$ – a constraint that can be treated by our unification algorithm.

We are now ready to present the unification algorithm, whose rules are given in Figure 6. Steps are represented by rules of the form $C; \theta \rightsquigarrow C'; \theta'$, with the pre-conditions that constraints in C are in normal form, the image of θ contains only levels in normal form, and the domain of θ is disjoint from the free variables of C – as we will see, these properties are preserved by each step. Given a substitution θ, we write $\widehat{\theta}$ for the substitution $i \mapsto \widehat{\theta(i)}$, and we write $\theta\{l/j\}$ for the substitution $i \mapsto \theta(i)\{l/j\}$.

$$\{l = l\} \cup C; \theta \rightsquigarrow C; \theta \qquad \textbf{(Trivial)}$$

$$\{l = l'\} \cup C; \theta \rightsquigarrow \{l' = l\} \cup C; \theta \qquad \text{if } l' = \mathtt{z} \text{ or } \mathtt{z} \sqcup i \quad \textbf{(Orient)}$$

$$\{\mathtt{z} \sqcup i = l\} \cup C; \theta \rightsquigarrow \widetilde{C\{l/i\}}; \widehat{\theta\{l/i\}} \cup \{i \mapsto l\} \qquad \text{if } i \notin l \quad \textbf{(Eliminate 1)}$$

$$\{\mathtt{z} \sqcup i = l\} \cup C; \theta \rightsquigarrow \text{let } l' = l\{i'/i\} \text{ in} \qquad \text{if } \mathtt{s}^m\ i \in l \text{ with } m = 0$$

$$\widetilde{C\{l'/i\}}; \widehat{\theta\{l'/i\}} \cup \{i \mapsto l'\} \qquad \text{where } i' \text{ is taken fresh} \quad \textbf{(Eliminate 2)}$$

$$\{\mathtt{z} = \mathtt{s}^k\ z \sqcup (\sqcup_{i \in V} \mathtt{s}^{n_i}\ i)\} \cup C; \theta \rightsquigarrow \{\mathtt{z} \sqcup i = \mathtt{z}\}_{i \in V} \cup C; \theta \qquad \text{if } k = 0, n_i = 0 \text{ for all } i \quad \textbf{(Decompose)}$$

$$\{\mathtt{z} = \mathtt{s}^k\ z \sqcup (\sqcup_{i \in V} \mathtt{s}^{n_i}\ i)\} \cup C; \theta \rightsquigarrow \bot \qquad \text{if } k \neq 0 \text{ or } n_i \neq 0 \text{ for some } i \quad \textbf{(Clash)}$$

Figure 6 Unification algorithm for $\simeq$

On the following, given a substitution θ we write $dom\ \theta$ for its domain. We write $\theta_1 \subseteq \theta_2$ when for all $i \in dom\ \theta_1$, $\theta_1(i) = \theta_2(i)$.

The following lemma is key in showing the main properties of our algorithm.

▶ **Lemma 17** (Key lemma). *Suppose* $C;\theta \rightsquigarrow C';\theta'$. *For all* τ, *if* $\tau \vDash C$ *and* $\tau \simeq \tau \circ \theta$ *then there is a substitution* τ' *with* $\tau \subseteq \tau'$ *such that (1)* $\tau' \vDash C'$ *and (2)* $\tau' \simeq \tau' \circ \theta'$. *Conversely, for all* τ, *if* $\tau \simeq \tau \circ \theta'$ *and* $\tau \vDash C'$, *then* $\tau \simeq \tau \circ \theta$ *and* $\tau \vDash C$.

It is not evident that $\rightsquigarrow$ always terminates, but we can restrict it so that termination becomes trivial. Let $\rightsquigarrow_0$ be the relation consisting in applying either one of the rules among (Trivial), (Eliminate1), (Eliminate2) and (Clash), or else (Orient) or (Decompose) followed by as many occurrences of (Eliminate 1), (Eliminate 2), (Clash) and (Trivial) as possible.

▶ **Lemma 18.** $\rightsquigarrow_0$ *terminates.*

▶ **Theorem 19.** *If* $C;\mathit{id} \rightsquigarrow_0^* \bot$, *then for no* θ *we have* $\theta \vDash C$.

▶ **Theorem 20.** *If* $C;\mathit{id} \rightsquigarrow_0^* \varnothing;\theta$, *then* θ *is a most general unifier.*

Proof. Let τ be a unifier. We thus have $\tau \vDash C$. Moreover, we have $\tau \simeq \tau \circ \mathit{id}$. By iterating Lemma 17, we get a substitution τ' such that $\tau' \simeq \tau' \circ \theta$ and $\tau(i) = \tau'(i)$ for $i \in \mathit{dom}\ \tau$. Therefore, for $i \in \mathit{dom}\ \tau$ we have $\tau(i) \simeq \theta(i)\tau'$, showing that τ is an instance of θ.

To show that θ is a unifier, first note that the set of variables in its image is disjoint from $\mathit{dom}\ \theta$, which in particular implies $\theta = \theta \circ \theta$. Moreover, $\theta \vDash \varnothing$ trivially, hence by iterating Lemma 17 in the inverse direction, we get $\theta \vDash C$. ◀

We have seen that when the algorithm finishes with $\varnothing;\theta$, then θ is a mgu, and when it finishes with $\bot$, then there is no solution to the constraints. However, the algorithm can also get stuck on constraints that it does not know how to solve. In practice, it is very unsatisfying for the unification to get stuck, as this means that the whole predicativization algorithm has to halt. Thus, in order to prevent this, in our implementation we extended the unification with heuristics that are *only* applied when none of the presented rules applies. Then, whenever the heuristics are applied, the universe polymorphic definition or declaration that is produced might not be the most general one. However, in practice, our algorithm succeeds most of the time without having to use such heuristics.

7 Predicativize, the implementation

In this section we present Predicativize, an implementation of our algorithm. It is publicly available at `https://github.com/Deducteam/predicativize/`.

Our tool is implemented on top of DkCheck [15], a type-checker for Dedukti, and thus does not rely neither on the code of Agda, nor on the code of any other proof assistant. Like Universo [19], our implementation instruments DkCheck's conversion checking in order to implement the constraint computation algorithm described in Section 5.

To understand how everything works in practice, we invite the reader to download the code and run `make running-example`, which translates our running example and produces two files: a Dedukti file `output/running_example.dk` and an Agda file `agda_output/running-example.agda`. In order to test the tool with a more realistic example, the reader can also run `make test_agda`, which translates a proof of Fermat's little theorem from the Dedukti encoding of HOL [18] to **UPP**.

In the following, let us go through some important particularities of how the tool works.

User added constraints

As we have seen, our tool tries to compute the most general type for a definition or declaration to be typable. However, it is not always desirable to have the most general type, as shown by the following example.

▶ **Example 21.** Consider the local signature

$$\Delta = Nat : U_\square; zero : El_\square\ Nat; succ : El_\square\ (Nat \rightsquigarrow_{\square,\square} Nat)$$

defining the natural numbers in **I**. The translation of this signature by our algorithm is

$$|\Delta| = Nat : \Pi i : Level.U_i; zero : \Pi i : Level.El_i\ (Nat\ i); succ : \Pi i\ j : Level.El_{(i \sqcup j)}\ ((Nat\ i) \rightsquigarrow_{i,j} (Nat\ j))$$

However, we normally would like to impose i to be equal to j in the type of *succ*, or even to impose *Nat* to not be universe polymorphic.

In order to solve this problem, we added to Predicativize the possibility of adding constraints by the user, in such a way that we can for instance impose *Nat* to be in U_z, or $i = j$ in the type of the successor.

Rewrite rules

The algorithm that we presented and proved correct covers two types of entries: definitions and constants. This is enough for translating proofs written in higher-order logic or similar systems, in which every step either poses an axiom or makes a definition or proof.

However, when dealing with full-fledged type theories, such as those implemented by Coq or Matita, which also feature inductive types, it is customary to use rewrite rules to encode recursion and pattern matching. If we simply ignore these rules when performing the translation, we would run into problems as the entries that appear after may need those rewrite rules to typecheck.

Therefore, our implementation extends the presented algorithm and also translate rewrite rules. In order to do this, we use DkCheck's subject reduction checker to generate constraints and proceed similarly as in the algorithm. Because this feature is still work in progress, this step can require user intervention in some cases. In this case, the user has to manually add constraints over some symbols to help the translation.

Agda output

Predicativize produces files in **UPP**, which is a subsystem of the encoding of Agda. In order to translate these files to Agda itself, we also integrated in Predicativize a translator that performs a simple syntactical translation from the Agda encoding in Dedukti to Agda. For instance, `make test_agda_with_typecheck` translates Fermat's Little Theorem proof from HOL to Agda and typechecks it.

8 Translating Matita's arithmetic library to Agda

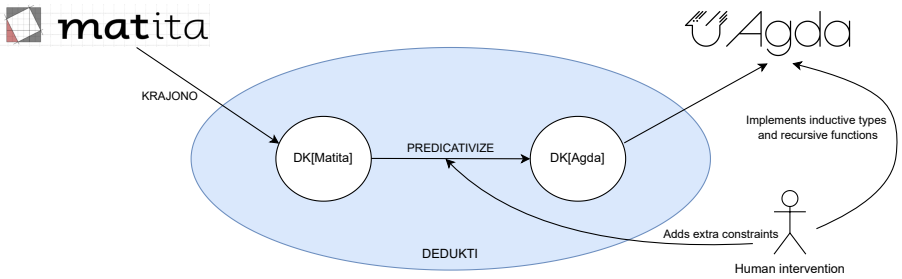

Figure 7 Diagram representing the translation of Matita's arithmetic library into Agda

We now discuss how we used Predicativize to translate Matita's arithmetic library to Agda. The translation is summarized in Figure 7, where **DK**[X] stands for the encoding of system X in Dedukti. Matita's arithmetic library was already available in Dedukti thanks to the Krajono tool [2], a translator from Matita to the encoding **DK**[Matita] in Dedukti. Therefore, the first step of the translation was already done for us.

Then, using Predicativize we translated the library from **DK**[Matita] to **DK**[Agda] (which is a supersystem of **UPP**). As the encoding of Matita's recursive functions uses rewrite rules, their translation required some user intervention to add constraints over certain symbols, as mentioned in the previous section. Once this step is done, the library is known to be predicative, as it typechecks in **DK**[Agda].

We then used Predicativize to translate these files to Agda files. However, because the rewrite rules in the Dedukti encoding cannot be translated to Agda, and given that they are needed for typechecking the proofs, the library does not typecheck directly.

Therefore, to finish our translation we had to define the inductive types and recursive functions manually in Agda. This step of our translation admittedly requires some time, however the effort is orders of magnitude less than rewriting the whole library in Agda, specially given that the great majority of the library is made of proofs, whose translations we did not need to change. Note that this manual step is not exclusive to our work as it is also needed in [18].

Defining inductive types also required us to add constraints. For instance, we saw in Example 21 that the successor symbol is translated as $\mathit{succ} : \Pi i\ j : \mathit{Level}.\mathit{El}_{(i \sqcup j)}\ ((\mathit{Nat}\ i) \rightsquigarrow_{i,j} (\mathit{Nat}\ j))$, but in order to be able to implement this symbol as a constructor of an inductive type, we need to impose $i = j$. If one then wishes to align Nat with the built-in type of natural numbers in Agda, we would also have to impose $i = \mathtt{z}$, which would then allow us to replace Nat by the built-in type in the result of the translation.

The result of this translation is available at `https://github.com/thiagofelicissimo/matita_lib_in_agda` and, as far as we know, contains the very first proofs in Agda of Bertrand's Postulate and Fermat's Little Theorem. It also contains a variety of other interesting results such as the Binomial Law, the Chinese Remainder Theorem, and the Pigeonhole Principle. Moreover, this library typechecks with Agda's `--safe` flag, attesting that it does not use any unsafe features.

9 Conclusion

We made an important step at sharing proofs with predicative systems, by proposing an algorithm for this problem. Our implementation allowed to translate many non-trivial proofs from Matita's arithmetic library to Agda, showing that our algorithm indeed works well in practice.

Our solution uses unification modulo arithmetic equivalence on universe levels. We designed an incomplete algorithm for this problem which is powerful enough for our needs. Still, one can wonder if there is an algorithm which always finds a most general solution when there is one. Agda also features an algorithm for solving level metavariables which uses an approach different from ours, but it does not seem to have been formalized in the literature. Therefore, the problem of finding such an algorithm seems to be open.

For future work, we would like to look also at possible ways of making Predicativize less dependent on user intervention. In particular, the translation of inductive types and recursive functions involves some considerable manual work. Thus if we want to be able to translate larger libraries, there is definitely a need for automating this step.

References

1. Andrea Asperti and Wilmer Ricciotti. A proof of bertrand's postulate. *Journal of Formalized Reasoning*, 5(1):37–57, 2012.
2. Ali Assaf. *A framework for defining computational higher-order logics*. These, École polytechnique, September 2015. URL: https://pastel.archives-ouvertes.fr/tel-01235303.
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. Michael Beeson, Julien Narboux, and Freek Wiedijk. Proof-checking Euclid. *Annals of Mathematics and Artificial Intelligence*, page 53, January 2019. URL: https://hal.archives-ouvertes.fr/hal-01612807, doi:10.1007/s10472-018-9606-x.
5. F. Blanqui. Encoding type universes without using matching modulo AC. In *Proceedings of the 7th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 228, 2022.
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, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSCD.2021.20.
8. 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.
9. Tristan Delort. Importer les preuves de Logipedia dans Agda. Internship report, Inria Saclay Ile de France, November 2020. URL: https://hal.inria.fr/hal-02985530.
10. Thiago Felicissimo. Adequate and Computational Encodings in the Logical Framework Dedukti. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16306, doi:10.4230/LIPIcs.FSCD.2022.25.
11. Gaspard Ferey. *Higher-Order Confluence and Universe Embedding in the Logical Framework*. These, Université Paris-Saclay, June 2021. URL: https://tel.archives-ouvertes.fr/tel-03418761.
12. Guillaume Genestier. Encoding agda programs using rewriting. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 31:1–31:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.31.
13. Yoan Géran. Euclid's elements book 1 in dedukti. URL: https://github.com/Karnaj/sttfa_geocoq_euclid [cited 2022].
14. Robert Harper and Robert Pollack. Type checking with universes. *Theor. Comput. Sci.*, 89(1):107–136, aug 1991. doi:10.1016/0304-3975(90)90108-T.
15. R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.
16. Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In *International Conference on Interactive Theorem Proving*, pages 499–514. Springer, 2014.
17. Agda Development Team. Agda 2.6.2.1 documentation. URL: https://agda.readthedocs.io/en/v2.6.2.1/index.html [cited 2022].

18 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.

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

A Typing rules for Dedukti and basic metaproperties

$$\frac{}{-;- \texttt{ well-formed}}\ \texttt{Empty} \qquad c \notin \Sigma\ \frac{\Sigma; - \vdash A : \mathbf{s}}{\Sigma, c : A; - \texttt{ well-formed}}\ \texttt{Decl-cons}$$

$$c \notin \Sigma\ \frac{\Sigma; - \vdash M : A}{\Sigma, c : A := M; - \texttt{ well-formed}}\ \texttt{Decl-def} \qquad x \notin \Gamma\ \frac{\Sigma; \Gamma \vdash A : \mathbf{Type}}{\Sigma; \Gamma, x : A \texttt{ well-formed}}\ \texttt{Decl-var}$$

$$c : A \text{ or } c : A := M \in \Sigma\ \frac{\Sigma; \Gamma \texttt{ well-formed}}{\Sigma; \Gamma \vdash c : A}\ \texttt{Cons} \qquad x : A \in \Gamma\ \frac{\Sigma; \Gamma \texttt{ well-formed}}{\Sigma; \Gamma \vdash x : A}\ \texttt{Var}$$

$$\frac{\Sigma; \Gamma \texttt{ well-formed}}{\Sigma; \Gamma \vdash \mathbf{Type} : \mathbf{Kind}}\ \texttt{Sort} \qquad A \equiv_{\beta\mathcal{R}\delta} B\ \frac{\Sigma; \Gamma \vdash M : A \qquad \Sigma; \Gamma \vdash B : \mathbf{s}}{\Sigma; \Gamma \vdash M : B}\ \texttt{Conv}$$

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

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

Figure 8 Typing rules for Dedukti

We recall the following basic metaproperties of Dedukti. Proofs can be found in [15, 6].

▶ **Theorem 22** (Basic metaproperties).

1. *Weakening: If* $\Sigma; \Gamma \vdash M : A$, $\Gamma \subseteq \Gamma'$ *and* $\Sigma; \Gamma' \texttt{ well-formed}$ *then* $\Sigma; \Gamma' \vdash M : A$
2. *Substitution Lemma: If* $\Sigma; \Gamma, x : B, \Gamma' \vdash M : A$ *and* $\Sigma; \Gamma \vdash N : B$ *then* $\Sigma; \Gamma, \Gamma'\{N/x\} \vdash M\{N/x\} : A\{N/x\}$
3. *Well-sortedness: If* $\Sigma; \Gamma \vdash M : A$ *then either* $A = \boldsymbol{Kind}$ *or* $\Sigma; \Gamma \vdash A : \boldsymbol{s}$ *for* $\boldsymbol{s} = \boldsymbol{Type}$ *or* $\boldsymbol{Kind}$.
4. *Subject reduction of* δ*: If* $\Sigma; \Gamma \vdash M : A$ *and* $M \longhookrightarrow_\delta M'$ *then* $\Sigma; \Gamma \vdash M' : A$
5. *Subject reduction of* β*: If injectivity of dependent product holds, then* $\Sigma; \Gamma \vdash M : A$ *and* $M \longhookrightarrow_\beta M'$ *implies* $\Sigma; \Gamma \vdash M' : A$.
6. *Contexts are well typed: If* $x : A \in \Gamma$ *then* $\Sigma; \Gamma \vdash A : \boldsymbol{Type}$
7. *Signature are well typed: If* $c : A \in \Sigma$ *then* $\Sigma; - \vdash A : \boldsymbol{s}$ *and if* $c : A := M \in \Sigma$ *then* $\Sigma; - \vdash M : A$
8. *Inversion of typing: Suppose* $\Sigma; \Gamma \vdash M : A$
   - *If* $M = x$ *then* $x : A' \in \Gamma$ *and* $A \equiv A'$
   - *If* $M = c$ *then* $c : A' \in \Sigma$ *and* $A \equiv A'$
   - *If* $M = \boldsymbol{Type}$ *then* $A \equiv \boldsymbol{Kind}$
   - $M = \boldsymbol{Kind}$ *is impossible*
   - *If* $M = \Pi x : A_1.A_2$ *then* $\Sigma; \Gamma, x : A_1 \vdash A_2 : \boldsymbol{s}$ *and* $\boldsymbol{s} \equiv A$
   - *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* $\Sigma; \Gamma, x : B \vdash C : \boldsymbol{s}$, $\Sigma; \Gamma, x : B \vdash N : C$ *and* $A \equiv \Pi x : B.C$

B More details on UPP

B.1 Representing $\simeq$ with a rewrite system

In Section 4 we gave part of the definition of the DEDUKTI theory **UPP**, but we choose to leave out the fragment of the theory that defines $\simeq$ with a rewrite system. Indeed, this encoding is very technical and uses rewriting modulo AC, and its comprehension is not necessary to understand our contribution. However, the full definition of **UPP** can be found in the file `theory/pts.dk` in the repository `https://github.com/Deducteam/predicativize/`. For a formal presentation of this rewrite system and a proof that it indeed implements $\simeq$, we refer to [12, 5].

B.2 Subject reduction in the theory UPP

The main metaproperty of **UPP** that we use in the proofs of Section 5 is subject reduction of $\hookrightarrow_{\beta\mathcal{R}_{UPP}\delta}$. To show this, one could use DKCHECK to verify that all the rules in $\mathcal{R}_{UPP}$ preserve typing – this can be done by running DKCHECK on the file `theory/pts.dk` in `https://github.com/Deducteam/predicativize/`, which is the file that defines **UPP**. However, the algorithm implemented in DKCHECK to check subject reduction assumes that injectivity of dependent products with respect to $\equiv_{\beta\mathcal{R}_{UPP}\delta}$ holds [15], and this is also needed to ensure that $\hookrightarrow_{\beta}$ preserves typing.

The standard way to show injectivity of dependent products is by showing that $\hookrightarrow_{\beta\mathcal{R}_{UPP}\delta}$ is confluent, which would imply the result [15]. However, the encoding of the relation $\simeq$ as a rewrite system requires a non-(left-)linear rewrite rule [12, 5], which destroys confluence on pre-terms.

Still, it is know that if a rewrite system is not Π-producing – that is, if for no $l \hookrightarrow r \in \mathcal{R}$ we have a subterm of the form $\Pi x : A_1.A_2$ in r – then injectivity of depended products automatically holds [15]. The rewrite system for $\simeq$ actually satisfies this property, but unfortunately its union with the other rules

$$El\ i'\ (\pi\ i\ j\ A\ B) \hookrightarrow \Pi x : El\ i\ A.El\ j\ (B\ x)$$

$$El\ i'\ (u\ i) \hookrightarrow U\ i$$

does not, given that the first rule is Π-producing.

Because these two rules (which are confluent by themselves) have no common symbols with the ones used to encode $\simeq$, we still think that the Π-producing criterion can be extended to prove injectivity of dependent products for our system.

If this turns out to be false, another possibility which would actually be simpler would be to switch to an encoding without the rule $El\ i'\ (\pi\ i\ j\ A\ B) \hookrightarrow \Pi x : El\ i\ A.El\ j\ (B\ x)$, so that the Π-producing criterion would apply. This could be done by adapting the work of [10].

In any case, we note that in practice we have translated and typechecked a large number of developments in **UPP** and have not found any errors due to lack of subject reduction. Therefore, we also have practical reasons to believe that it indeed holds.

C Proofs of Section 5

C.1 Lemma 15

It suffices to show that for each step transforming $l_1 = l_2$ into $l_1' = l_2'$, we have $l_1\theta \simeq l_2\theta$ iff $l_1'\theta \simeq l_2'\theta$. For the first part, which transforms $l_1 = l_2$ into $\widehat{l_1} = \widehat{l_2}$, because we have $l_1 \simeq \widehat{l_1}$, $l_2 \simeq \widehat{l_2}$, we thus deduce $l_1\theta \simeq \widehat{l_1}\theta$ and $l_2\theta \simeq \widehat{l_2}\theta$, from which the result follows.

For the second part, it suffices to show that for any l_1, l_2, l, n with $n > 0$, we have (*) $l_1 \sqcup l \simeq l_2 \sqcup (\mathtt{s}^n\ l) \iff l_1 \simeq l_2 \sqcup (\mathtt{s}^n\ l)$. Note that for any $p_1, p_2, m, q \in \mathbb{N}$ with $q > 0$ we have $max\{p_1, m\} = max\{p_2, q + m\} \iff p_1 = max\{p_2, q + m\}$. Indeed, the direction $\Leftarrow$ is clear, whereas for $\Rightarrow$ if we had $m = max\{p_2, q + m\}$ then we would have $m > m$, contradiction. We then can show ($\star$) by unfolding the definition of $\simeq$ and applying this fact.

For the third part, it suffices to note that $\mathtt{s}^m\ l \sqcup \mathtt{s}^m\ l' \simeq \mathtt{s}^m\ (l \sqcup l')$ and that $\mathtt{s}^m\ l \simeq \mathtt{s}^m\ l'$ iff $l \simeq l'$.

C.2 Lemma 17

We first start with the following auxiliary lemma:

▶ **Lemma 23.** *Suppose $i\theta \simeq l\theta$. Then we have*

1. *$l'\theta \simeq l'\{l/i\}\theta$, for all l'*
2. *$\theta \models C$ iff $\theta \models C\{l/i\}$.*

Proof. Part (1) can be shown by a simple induction on l'. For part (2), given $l_1 = l_2 \in C$, we have $l_1\theta \simeq l_1\{l/i\}\theta$ and $l_2\theta \simeq l_2\{l/i\}\theta$, and thus $l_1\theta \simeq l_2\theta$ iff $l_1\{l/i\}\theta \simeq l_2\{l/i\}\theta$. ◀

We now continue with the proof of Lemma 17. It is done by case analysis on the rule, the cases **Trivial** and **Orient** being trivial. On the following, we might use Lemma 23 implicitly.

Eliminate 1: Suppose $\tau \models C$ and $\tau \simeq \tau \circ \theta$, and pose $\tau' = \tau$. First note that we have $i\tau \simeq l\tau$, and thus $i\tau' \simeq l\tau'$.

1. As $\tau \models C$, we have $\tau' \models C$. By $i\tau' \simeq l\tau'$ and Lemma 23, $\tau' \models C\{l/i\}$. Finally, by Lemma 15, $\tau \models \widetilde{C\{l/i\}}$.
2. For all $j \in dom\ \tau$, we have $\tau(j) \simeq \theta(j)\tau$, thus by Lemma 23 we get $\tau(j) \simeq \theta(j)\{l/i\}\tau$. But because $\theta(j)\{l/i\} \simeq \widehat{\theta(j)\{l/i\}}$, we have $\tau(j) \simeq \widehat{\theta(j)\{l/i\}}\tau$. Because $\tau' = \tau$ and $\theta' = \widehat{\theta\{l/i\}}$, this establishes $\tau'(j) \simeq \theta'(j)\tau'$.

Conversely, suppose now $\tau \models \widetilde{C\{l/i\}}$ and $\tau = \tau \circ \theta'$. As $\theta'(i) = l$, it follows that $\tau(i) \simeq l\tau$, showing $(\mathtt{z} \sqcup i)\tau \simeq l\tau$. Thus, $\tau \models \widetilde{C\{l/i\}}$ implies $\tau \models C\{l/i\}$, which then implies $\tau \models C$. Finally, for $j \neq i$, $\tau(j) \simeq \theta'(j)\tau = \widehat{\theta(j)\{l/i\}}\tau \simeq \theta(j)\{l/i\}\tau$, which is equivalent to $\theta(j)\tau$.

Eliminate 2: Suppose $\tau \models C$ and $\tau \simeq \tau \circ \theta$. Pose $\tau' = \tau \cup \{i' \mapsto \tau(i)\}$. First note $(\mathtt{z} \sqcup i)\tau \simeq l\tau$ implies $i\tau' \simeq l'\tau'$.

1. For all $l_1 = l_2 \in C$ we have $l_1\tau \simeq l_2\tau$, and thus $l_1\tau' \simeq l_2\tau'$ (as i' is fresh and does not appear in l_1, l_2). By Lemma 23 and $i\tau' \simeq l'\tau'$ we have $l_1\{l'/i\}\tau' \simeq l_2\{l'/i\}\tau'$, for all $l_1 = l_2 \in C$. By Lemma 15, $\tau \models \widetilde{C\{l'/i\}}$.
2. For all $j \in dom\ \tau$, we have $\tau(j) \simeq \theta(j)\tau$, and in particular $\tau'(j) \simeq \theta(j)\tau'$. By Lemma 23 and $i\tau' \simeq l'\tau'$, we get $\tau'(j) \simeq \theta(j)\{l'/i\}\tau'$, and thus $\tau'(j) \simeq \widehat{\theta(j)\{l'/i\}}\tau'$. Therefore, this establishes $\tau'(j) \simeq \theta'(j)\tau'$ for all $j \in dom\ \tau$. Finally, for $j = i'$ this holds trivially as $\theta'(i') = i'$.

Conversely, suppose now $\tau \models \widetilde{C\{l'/i\}}$ and $\tau = \tau \circ \theta'$. As $\theta'(i) = l'$, it follows that $\tau(i) \simeq l'\tau$, from which we can show $(\mathtt{z} \sqcup i)\tau \simeq l\tau$. Moreover, $\tau \models \widetilde{C\{l'/i\}}$ implies $\tau \models C\{l'/i\}$, which then implies $\tau \models C$. Finally, for $j \neq i$, $\tau(j) \simeq \theta'(j)\tau = \widehat{\theta(j)\{l'/i\}}\tau \simeq \theta(j)\{l'/i\}\tau$, which is equivalent to $\theta(j)\tau$.

Decompose: Suppose $\tau \models C$ and $\tau \simeq \tau \circ \theta$. Pose $\tau' = \tau$. Point (2) is trivial. For (1), as $\mathtt{z} \simeq \sqcup_{i \in V}\tau(i)$, we must have $\tau(i) \simeq \mathtt{z}$ for all $i \in V$. Thus $(\mathtt{z} \sqcup i)\tau' \simeq \mathtt{z}\tau'$ for all $i \in V$. Finally, for all $l_1 = l_2 \in C$, we have $l_1\tau' \simeq l_2\tau'$ by hypothesis. A symmetric reasoning shows also the converse statement.

C.3 Lemma 18

Each step of $\rightsquigarrow_0$ decreases the number of constraints in C.

C.4 Lemma 19

If $\theta \vDash C$, then by iterating Lemma 17 we get that for some θ', $\mathtt{z}\theta' \simeq (\mathtt{s}^k\ \mathtt{z} \sqcup (\sqcup_{i \in V} \mathtt{s}^{n_i}\ i))\theta'$. But for any σ we have $[\![\mathtt{z}\theta']\!]_\sigma = 0$ and $[\![(\mathtt{s}^k\ \mathtt{z} \sqcup (\sqcup_{i \in V} \mathtt{s}^{n_i}\ i))\theta']\!]_\sigma > 0$, contradiction.