Logical Methods in Computer Science
Volume 20, Issue 3, 2024, pp. 23:1–23:38
https://lmcs.episciences.org/

Submitted Aug. 31, 2023
Published Sep. 10, 2024

SHARING PROOFS WITH PREDICATIVE THEORIES THROUGH UNIVERSE-POLYMORPHIC ELABORATION

THIAGO FELICISSIMO AND FRÉDÉRIC BLANQUI

Université Paris-Saclay, INRIA project Deducteam, Laboratoire Méthodes Formelles, ENS Paris-Saclay, 91190 France
e-mail address: thiago.felicissimo@inria.fr, frederic.blanqui@inria.fr

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 to proof assistants based on predicative logics, whenever impredicativity is not used in an essential way.

In this paper we present a transformation for sharing proofs with a core predicative system supporting prenex universe polymorphism. It consists in trying to elaborate each term into a predicative universe-polymorphic term as general as possible. The use of universe polymorphism is justified by the fact that mapping each universe to a fixed one in the target theory is not sufficient in most cases. During the elaboration, we need to solve unification problems in the equational theory of universe levels. In order to do this, we give a complete characterization of when a single equation admits a most general unifier. This characterization is then employed in a partial algorithm which uses a constraint-postponement strategy for trying to solve unification problems.

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 library to Agda, including proofs of Bertrand's Postulate and Fermat's Little Theorem, which (as far as we know) were not available in Agda yet.

1. Introduction

An important achievement 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. Proof assistants do not only help mathematicians to make sure that their proofs are indeed correct, but are also used to verify the correctness of safety-critical software.

Key words and phrases: Type Theory, Impredicativity, Predicativity, Proof Translation, Universe Polymorphism, Universe-Polymorphic Elaboration, Unification for Universe Levels, Agda, Dedukti.

DOI:10.46298/LMCS-20(3:23)2024
© T. Felicissimo and F. Blanqui
Creative Commons

Interoperability of proof assistants. Unfortunately, a proof written in a proof assistant cannot be directly 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 often required to do logical transformations.

A naïve 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 once again in order to be used with A' — the translation is *implementation-dependent*.

Logical Frameworks & Dedukti. A better solution is instead to first define the logics of all proof assistants in a common formalism, a *logical framework*. Then, proof transformations can be defined uniformly *inside* the logical framework. Hence, such transformations do not depend on the implementations anymore, but instead on the *logics* that are implemented.

The logical framework Dedukti [ABC+16] 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 [BDG+23]. Moreover, previous works have shown how proofs can be transformed inside Dedukti. For instance, Thiré describes in [Thi18] 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 [BNW19] to several proof assistants [Gé].

(Im)Predicativity. One of the challenges in proof system interoperability is sharing proofs coming from impredicative proof assistants (the majority of them) with predicative ones such as Agda. Indeed, impredicativity, which states that propositions can refer to entities of arbitrary sizes, is a logical principle absent from predicative systems. It is therefore clear that any proof that uses impredicativity in an essential way cannot be translated to a predicative system. Nevertheless, one can wonder if most proofs written in impredicative systems really use impredicativity and, if not, how one could devise a way for trying to detect this and translate them to predicative systems.

A predicativization transformation. In this paper, we tackle this problem by proposing a transformation that tries to do precisely this. Our translation works by forgetting all the universe information of the initial impredicative term, and then trying to elaborate it into a predicative universe-polymorphic term as general as possible. The need for universe polymorphism arises from the fact that mapping each universe to a unique one in the target theory does not work in most cases — this is explained in details in Section 3.

Universe level unification. During the translation, we need to solve level unification problems which are generated when elaborating the impredicative term into a universe-polymorphic one. We therefore develop a (partial) unification algorithm for the equational theory of universe levels. This is done by first giving a novel and complete characterization

of which single equations admit a most general unifier (m.g.u.), along with an explicit description of a m.g.u. when it exists. This characterization is then employed in an algorithm implementing a *constraint-postponement* strategy: at each step, we look for an equation admitting a m.g.u. and solve it while applying the obtained substitution to the other equations, in the hope of bringing new ones to the fragment admitting a m.g.u. The given algorithm is *partial* in the sense that, when the unification problem is not a singleton, it may fail to find a m.g.u. even in cases that there is one — see for instance Example 6.23. Our practical results show nevertheless that it is sufficiently powerful for our needs.

The implementation. Our predicativization algorithm was implemented on top of the DkCheck type-checker for Dedukti with the tool Predicativize (available at `https://github.com/Deducteam/predicativize`), allowing for the translation of proofs inside Dedukti. Our tool works in a semi-automatic manner: most of the translation is handled by the proposed algorithm, yet some intermediate steps that are harder to automate currently require some user intervention. The translated proofs can then be exported to Agda, the main proof assistant based on predicative type theory.

Translating Matita's arithmetic library. The tool has been used to translate to the proof assistant Agda the *whole* of Matita's arithmetic library, making many important mathematical developments available to Agda users. In particular, this work has led to (as far as we know) the first ever proofs in Agda of Fermat's Little Theorem, stating that for $p \in \mathbb{N}$ prime and $n \in \mathbb{N}$ coprime to p we have n^{p-1} equal to 1 modulo p, and of Bertrand's Postulate,[1] stating that for all positive $n \in \mathbb{N}$ one can always find a prime number p with $n < p \leqslant 2n$.

The proof of Bertrand's Postulate in Matita had even been the subject of a whole journal publication [AR12], evidencing its complexity and importance. Thanks to Predicativize, the same hard work did not have to be repeated to make it available in Agda, as the transformation allowed the translation of the whole proof without *any* need of specialist knowledge about it.

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, in which we present the elaboration algorithm. Section 6 then contributes with a complete characterization of equations admitting a m.g.u., which is then used to give an algorithm for universe level unification. We then introduce the tool Predicativize in Section 7, and describe the translation of Matita's arithmetic library in Section 8. Finally, Section 9 concludes and discusses future work.

Related version. A preliminary version of this work [FBB23] was published in the proceedings of the 31st EACSL Annual Conference on Computer Science Logic. This journal version contains a number of improvements, among which are the following:

(1) A main novelty with respect to [FBB23] is that we provide a complete characterization of when a single equation between universe levels admit a m.g.u. (most general unifier), along with an explicit description of such a m.g.u. This characterization then allows

[1] Which, despite its name, is actually a theorem and not a postulate.

$$\textsc{EmptyCtx}\ \frac{}{\cdot \vdash} \qquad \textsc{ExtCtx}\ \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : A \vdash} \qquad \textsc{ExtCtxC}\ \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, i : A \vdash}$$

$$\textsc{Sort}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}} \qquad \textsc{Var}\ \frac{\Gamma \vdash}{x : A \in \Gamma \quad \Gamma \vdash x : A} \qquad \textsc{VarC}\ \frac{\Gamma \vdash}{i : A \in \Gamma \quad \Gamma \vdash i : A}$$

$$\textsc{Cons}\ \frac{\Gamma \vdash}{\begin{array}{l} c : A \in \Sigma_{\mathbb{T}} \text{ or} \\ c : A := u \in \Sigma_{\mathbb{T}} \end{array} \quad \Gamma \vdash c : A} \qquad \textsc{ConsC}\ \frac{\Gamma \vdash \vec{l} : \Delta}{f : \Delta \to A \in \Sigma_{\mathbb{T}} \quad \Gamma \vdash f(\vec{l}) : A[\vec{i_\Delta} \mapsto \vec{l}]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{A \equiv B \quad \Gamma \vdash t : B} \qquad \textsc{ConvC}\ \frac{\Gamma \vdash l : A \qquad \Gamma \vdash B : \mathsf{Type}}{A \equiv B \quad \Gamma \vdash l : B}$$

$$\textsc{Arrow}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash (x : A) \to B : s} \qquad \textsc{ArrowC}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, i : A \vdash B : s}{\Gamma \vdash (i : A) \to B : s}$$

$$\textsc{Abs}\ \frac{\begin{array}{c} \Gamma \vdash A : \mathsf{Type} \\ \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B \end{array}}{\Gamma \vdash x.t : (x : A) \to B} \qquad \textsc{AbsC}\ \frac{\begin{array}{c} \Gamma \vdash A : \mathsf{Type} \\ \Gamma, i : A \vdash B : s \qquad \Gamma, i : A \vdash t : B \end{array}}{\Gamma \vdash i.t : (i : A) \to B}$$

$$\textsc{App}\ \frac{\Gamma \vdash t : (x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \mapsto u]} \qquad \textsc{AppC}\ \frac{\Gamma \vdash t : (i : A) \to B \qquad \Gamma \vdash l : A}{\Gamma \vdash t\ l : B[i \mapsto l]}$$

Figure 1: Typing rules of Dedukti

us to give a better algorithm for level unification, which in particular is complete for singleton problems, whereas the original algorithm is not.

(2) The confluence proof of $\mathbb{UPP}$ in [FBB23] relied on the *ad hoc* restriction that level variables could only be replaced by levels. We make this condition more precise and integrate it in the definition of Dedukti by adopting a presentation featuring *confinement*, a technique first proposed in [ADJL17] which allows to isolate a first-order subset of terms from the higher-order part of the syntax.

(3) Finally, most of the text has been rewritten in order to improve the presentation.

2. Dedukti

In this work we use Dedukti [ABC+16, BDG+23] as the framework in which we express the various type theories we use and define our proof transformation. Therefore, we start with a quick introduction to this system. For a reader familiar with Dedukti, see Remark 2.1 for

a comparison of our presentation of Dedukti with more standard ones — in particular, note that we use a version with *confinement* [ADJL17].

The syntax of Dedukti is defined by the following grammars. Here, c ranges over a set of constants $\mathcal{C}$ and f ranges over a set of confined constants $\mathcal{F}$. Similarly, x ranges over an infinite set of variables $\mathcal{V}$, whereas i ranges over an infinite set of confined variables $\mathcal{I}$. We assume that the sets $\mathcal{V}$, $\mathcal{I}$, $\mathcal{C}$ and $\mathcal{F}$ are pairwise disjoint, and that each confined constant f comes with an arity $n \in \mathbb{N}$.

Because of confinement, abstraction and application come in two flavors. First, we have $x.t$ and $t\ u$ for the usual abstraction and application. Then, we also have $i.t$ and $t\ l$ for abstracting a confined variable or applying a regular term to a confined term — note therefore that confined terms are not terms, but can appear in the right side of an application. Accordingly, we also have the dependent functions types $(x : A) \to B$ for the regular case, and $(i : A) \to B$ for the confined case. Whenever the variable x does not appear free in B, we abbreviate $(x : A) \to B$ as just $A \to B$.

(Sorts)	$s, s' ::= \mathsf{Type} \mid \mathsf{Kind}$
(Terms and Types)	$A, B, t, u ::= x \mid c \mid s \mid (x : A) \to B \mid x.t \mid t\ u \mid (i : A) \to B \mid i.t \mid t\ l$
(Confined Terms)	$l, l' ::= i \mid f(l_1, ..., l_n)$ where $\mathsf{arity}(f) = n$

A *substitution* θ is a finite set of pairs $x \mapsto t$ or $i \mapsto l$ where each variable occurs at most once — note that regular variables can only be mapped to regular terms, and confined variables only to confined terms. We write $t[\theta]$ or $l[\theta]$ for its application to a term or confined term, and $\mathsf{dom}(\theta)$ for the set of variables that are assigned to a term or confined term by θ.

A *context* Γ is a finite sequence of entries of the form $x : A$ or $i : A$ where each variable occurs at most once. A *signature* Σ is a finite sequence of entries of the form $c : A$ or $c : A := t$ or $f : (i_1 : B_1..i_n : B_n) \to A$, where we must have $\mathsf{arity}(f) = n$. We adopt the convention of writing the names of constants of the signature in blue sans serif font[2].

A *rewrite system* $\mathcal{R}$ is a set of *rewrite rules*, which are pairs of the form $c\ \vec{t} \longrightarrow u$ with $\mathsf{fv}(u) \subseteq \mathsf{fv}(c\ \vec{t})$. Given a signature Σ, we also consider the δ rules allowing for the unfolding of definitions: we have $c \longrightarrow t \in \delta$ for each $c : A := t \in \Sigma$. We then denote by $\longrightarrow_{\mathcal{R}}$ the closure by context and substitution of $\mathcal{R}$, and by $\longrightarrow_{\delta}$ the closure by context of δ. Finally, we define $\longrightarrow_{\beta}$ as the closure by context of $(x.t)u \longrightarrow t[x \mapsto u]$, $\longrightarrow_{\beta_c}$ as the closure by context of $(i.t)l \longrightarrow t[i \mapsto l]$, and we write $\longrightarrow_{\beta\beta_c\mathcal{R}\delta}$ for $\longrightarrow_{\beta} \cup \longrightarrow_{\beta_c} \cup \longrightarrow_{\mathcal{R}} \cup \longrightarrow_{\delta}$.

Rewriting allows us to define equality by computation, but not all equalities can be defined like this in a well-behaved way, e.g. the commutativity of some operator. Therefore, we also consider *rewriting modulo equations* [Hue80, BKdVT03, Bla03]. If $\mathcal{E}$ is a set of equations of the form $l \simeq l'$, we write $\simeq_{\mathcal{E}}$ for its reflexive-symmetric-transitive closure by context and substitution — note that we only allow equations between confined terms. Because $\mathcal{R}$ and $\mathcal{E}$ are usually kept fixed, in the following we write $\longrightarrow$ for $\longrightarrow_{\beta\beta_c\mathcal{R}\delta}$, $\simeq$ for $\simeq_{\mathcal{E}}$ and $\equiv$ for the reflexive-symmetric-transitive closure of $\longrightarrow \cup \simeq$.

One central notion in Dedukti is that of *theory*, which is a triple $\mathbb{T} = (\Sigma_{\mathbb{T}}, \mathcal{R}_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}})$ where $\Sigma_{\mathbb{T}}$ is a signature and $\mathcal{R}_{\mathbb{T}}$ and $\mathcal{E}_{\mathbb{T}}$ are respectively sets of rewrite rules and equations, containing only constants and confined constants declared in $\Sigma_{\mathbb{T}}$. Theories are used to define in Dedukti the object logics in which we work (for instance, predicate logic). When

[2]Note that the letters c and f themselves are not written in blue because they are not really names, but rather variables of the metalanguage for referring to constant names.

working in some theory, we consider untyped terms containing only constants declared in the theory — that is, we assume that the sets $\mathcal{C}$ and $\mathcal{F}$ contain exactly the constants and confined constants declared in $\Sigma_{\mathbb{T}}$. Given a theory $\mathbb{T}$, we define the typing rules of Dedukti as the ones of Figure 1. There, we write $\Gamma \vdash \vec{l} : \Delta$ for $\Gamma \vdash l_k : B_k[i_1 \mapsto l_1..i_{k-1} \mapsto l_{k-1}]$ for all $k = 1..n$ when $\Delta = i_1 : B_1..i_n : B_n$. Note also that the signature and the conversion relation $\equiv$ are the ones defined by the theory $\mathbb{T}$. Finally, whenever the underlying theory is not clear from the context, we write $\Gamma \vdash^{\mathbb{T}} t : A$.

A signature entry $c : A$ or $c : A := t$ or $f : \Delta \to A$ is valid in $\mathbb{T}$ if, respectively, $\vdash^{\mathbb{T}} A : s$ or $\vdash^{\mathbb{T}} t : A$ or $\Delta \vdash^{\mathbb{T}} A : \mathsf{Type}$. A theory $\mathbb{T}$ is then said to be well-formed if each entry in $\Sigma_{\mathbb{T}}$ is valid in $(\Sigma', \mathcal{R}', \mathcal{E}')$, where Σ' is the prefix of $\Sigma_{\mathbb{T}}$ preceding the entry in question, and $\mathcal{R}', \mathcal{E}'$ are the restrictions of $\mathcal{R}_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}}$ to rules and equations only containing constants in Σ'.

Remark 2.1. Compared with most presentations of Dedukti [ABC+16, BDG+23], ours feature three relevant differences:

(1) We consider a syntax with non-annotated abstractions. As shown in [Dow93], this leads to undecidable type checking. However, in this article we only work with encodings in which the only terms of interest are the β-normal forms [Fel22]. For these terms, the omission of such annotations does not jeopardize the decidability of type checking.
(2) Like some other works [ADJL17, Gen20, Bla22], we consider a version of Dedukti with rewriting modulo. This is essential to support some equations which cannot be oriented into rewrite rules.
(3) As mentioned previously, we consider a version with *confinement*. This notion, first introduced in [ADJL17], allows to syntactically isolate a first-order part of the syntax from the higher-order one. In [ADJL17] it is used to provide a confluence criterion for non-left-linear rewriting. In a similar vein, we use it to allow for non-linear equations in $\mathcal{E}$ without jeopardizing the *Church-Rosser modulo* property — see Remark 4.6 for further discussion on this point.

We recall the following basic metaproperties of Dedukti.

Proposition 2.2 (Basic metaproperties)**.** *Let us write $\Gamma \sqsubseteq \Gamma'$ when Γ is a subsequence of Γ', and let $\Gamma \vdash \mathcal{J}$ range over the typing judgment forms $\Gamma \vdash$ or $\Gamma \vdash t : A$ or $\Gamma \vdash l : A$.*

Weakening: *Suppose $\Gamma \sqsubseteq \Gamma'$ and $\Gamma' \vdash$. Then $\Gamma \vdash \mathcal{J}$ implies $\Gamma' \vdash \mathcal{J}$.*
Substitution property: *If $\Gamma, x : B, \Gamma' \vdash \mathcal{J}$ and $\Gamma \vdash u : B$ then $\Gamma, \Gamma'[x \mapsto u] \vdash \mathcal{J}[x \mapsto u]$. If $\Gamma, i : B, \Gamma' \vdash \mathcal{J}$ and $\Gamma \vdash l : B$ then $\Gamma, \Gamma'[i \mapsto l] \vdash \mathcal{J}[i \mapsto l]$.*

In the following points, suppose that the underlying theory is well-formed.

Validity: *If $\Gamma \vdash t : A$ then either $A = \mathsf{Kind}$ or $\Gamma \vdash A : s$ for some sort s. If $\Gamma \vdash l : A$ then $\Gamma \vdash A : \mathsf{Type}$.*
Subject reduction for δ: *If $\Gamma \vdash t : A$ and $t \longrightarrow_\delta t'$ then $\Gamma \vdash t' : A$*

We say that injectivity of dependent products holds when $(x : A) \to B \equiv (x : A') \to B'$ implies $A \equiv A'$ and $B \equiv B'$, and $(i : A) \to B \equiv (i : A') \to B'$ implies $A \equiv A'$ and $B \equiv B'$.

Subject reduction for β and β_c: *If injectivity of dependent products holds, then $\Gamma \vdash t : A$ and $t \longrightarrow_{\beta\beta_c} t'$ imply $\Gamma \vdash t' : A$.*

A rule $l \longrightarrow r \in \mathcal{R}$ is said to preserve typing whenever $\Gamma \vdash l[\theta] : A$ implies $\Gamma \vdash r[\theta] : A$, for every θ, Γ, A.

Subject reduction for $\mathcal{R}$: *If every rule in $\mathcal{R}$ preserves typing, then $\Gamma \vdash t : A$ and $t \longrightarrow_{\mathcal{R}} t'$ imply $\Gamma \vdash t' : A$.*

Proof. We refer to [Bla01, Sai15] for detailed proofs — even if there the definition of the typing system is not exactly the same, the proofs for the variant used here are straightforward adaptions of their proofs. □

2.1. **Defining type theories in Dedukti.** We briefly review how one can define type theory with Russell-style universes and dependent products [Fel22][3]. Given a set $\mathfrak{S}$ of *sorts*, we start by declaring the following constants:

$$\mathsf{Ty}_s : \mathsf{Type} \qquad \text{for } s \in \mathfrak{S}$$
$$\mathsf{Tm}_s : \mathsf{Ty}_s \to \mathsf{Type} \qquad \text{for } s \in \mathfrak{S}$$

Then, an object type A at sort s is represented as an element of Ty_s, and an object term t of type A is represented as an element of $\mathsf{Tm}_s\ A$, where s is the sort of A. That is, we write $A : \mathsf{Ty}_s$ to represent the object-level judgment form $A\ \mathsf{type}_s$, and $t : \mathsf{Tm}_s\ A$ to represent the object-level judgment form $t :_s A$. As an example, if we wanted to add natural numbers at sort $s_0 \in \mathfrak{S}$, we would add constants $\mathsf{Nat} : \mathsf{Ty}_{s_0}$, $0 : \mathsf{Tm}_{s_0}\ \mathsf{Nat}$, $\mathsf{S} : \mathsf{Tm}_{s_0}\ \mathsf{Nat} \to \mathsf{Tm}_{s_0}\ \mathsf{Nat}$, and then a constant for its elimination principle along with its corresponding rewrite rules.

We now add universes. To do this, we suppose we are given a functional relation $\mathfrak{A} \subseteq \mathfrak{S}^2$, relating a sort to its successor, and then postulate a constant U_s in $\mathsf{Ty}_{s'}$ for each $(s, s') \in \mathfrak{A}$ to represent the universe for the sort s. The defining property of the universe for s is that its object terms should correspond somehow to object types at sort s. One way of achieving this is by postulating a definitional isomorphism $\mathsf{Tm}_{s'}\ \mathsf{U}_s \simeq \mathsf{Ty}_s$, which defines *Coquand-style universes* [Coq13, KHS19, ABKT19]. But because in Dedukti we have type-level rewrite rules, we can replace this isomorphism with an identification using a rewrite rule, yielding *Russell-style universes*. In this style of type universes, object terms typed by the universe for s become really the same as the object types at sort s.

$$\mathsf{U}_s : \mathsf{Ty}_{s'} \qquad \text{for } (s, s') \in \mathfrak{A}$$
$$\mathsf{Tm}_{s'}\ \mathsf{U}_s \longrightarrow \mathsf{Ty}_s \qquad \text{for } (s, s') \in \mathfrak{A}$$

Let us now define dependent products. We now suppose we are given a relation $\mathfrak{R} \subseteq \mathfrak{S}^3$, which is functional when seen as $\mathfrak{R} \subseteq \mathfrak{S}^2 \times \mathfrak{S}$. Then, for each triple $(s, s', s'') \in \mathfrak{R}$, we postulate a constant $\Pi_{s,s'}$ mapping $A : \mathsf{Ty}_s$ and $B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}$ to an element of $\mathsf{Ty}_{s''}$. Then we add abstraction $\lambda_{s,s'}$, mapping $t : (x : \mathsf{Tm}_s\ A) \to \mathsf{Tm}_{s'}\ (B\ x)$ to an element of $\mathsf{Tm}_{s''}\ (\Pi_{s,s'}\ A\ B)$, and application $@_{s,s'}$, mapping $t : \mathsf{Tm}_{s''}\ (\Pi_{s,s'}\ A\ B)$ and $u : \mathsf{Tm}_s\ A$ to an element of $\mathsf{Tm}_{s'}\ (B\ u)$.

$$\Pi_{s,s'} : (A : \mathsf{Ty}_s) \to (B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}) \to \mathsf{Ty}_{s''} \qquad \text{for } (s, s', s'') \in \mathfrak{R}$$
$$\lambda_{s,s'} : (A : \mathsf{Ty}_s) \to (B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}) \to ((x : \mathsf{Tm}_s\ A) \to \mathsf{Tm}_{s'}\ (B\ x)) \to \mathsf{Tm}_{s''}\ (\Pi_{s,s'}\ A\ B) \qquad \text{for } (s, s', s'') \in \mathfrak{R}$$
$$@_{s,s'} : (A : \mathsf{Ty}_s) \to (B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}) \to (t : \mathsf{Tm}_{s''}\ (\Pi_{s,s'}\ A\ B)) \to (u : \mathsf{Tm}_s\ A) \to \mathsf{Tm}_{s'}\ (B\ u) \qquad \text{for } (s, s', s'') \in \mathfrak{R}$$

Finally, we need to state the associated computation rule of dependent products. The most natural choice would be to take the following rule.

$$@_{s,s'}\ A\ B\ (\lambda_{s,s'}\ A\ B\ t)\ u \longrightarrow t\ u \qquad \text{for } (s, s', s'') \in \mathfrak{R}$$

[3]Other approaches also exists [CD07], however as argued in [Fel22] they lead to less well-behaved encodings.

$$
\begin{array}{ll}
\mathsf{Ty}_s : \mathsf{Type} & \text{for } s \in \mathfrak{S} \\
\mathsf{Tm}_s : \mathsf{Ty}_s \to \mathsf{Type} & \text{for } s \in \mathfrak{S} \\
\mathsf{U}_s : \mathsf{Ty}_{s'} & \text{for } (s, s') \in \mathfrak{A} \\
\mathsf{Tm}_{s'}\ \mathsf{U}_s \longrightarrow \mathsf{Ty}_s & \text{for } (s, s') \in \mathfrak{A} \\
\Pi_{s,s'} : (A : \mathsf{Ty}_s) \to (B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}) \to \mathsf{Ty}_{s''} & \text{for } (s, s', s'') \in \mathfrak{R} \\
\lambda_{s,s'} : (A : \mathsf{Ty}_s) \to (B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}) \to & \\
\quad ((x : \mathsf{Tm}_s\ A) \to \mathsf{Tm}_{s'}\ (B\ x)) \to \mathsf{Tm}_{s''}\ (\Pi_{s,s'}\ A\ B) & \text{for } (s, s', s'') \in \mathfrak{R} \\
@_{s,s'} : (A : \mathsf{Ty}_s) \to (B : \mathsf{Tm}_s\ A \to \mathsf{Ty}_{s'}) \to & \\
\quad (t : \mathsf{Tm}_{s''}\ (\Pi_{s,s'}\ A\ B)) \to (u : \mathsf{Tm}_s\ A) \to \mathsf{Tm}_{s'}\ (B\ u) & \text{for } (s, s', s'') \in \mathfrak{R} \\
@_{s,s'}\ A\ B\ (\lambda_{s,s'}\ A'\ B'\ t)\ u \longrightarrow t\ u & \text{for } (s, s', s'') \in \mathfrak{R}
\end{array}
$$

Figure 2: Dedukti theory defined by specification $(\mathfrak{S}, \mathfrak{A}, \mathfrak{R})$

However, this rule is non left-linear, and thus the rewrite system it generates is non-confluent on raw terms [Klo63]. Instead, the standard solution is to *linearize* [Bla05, Sai15, MW96] the rule, by replacing the second occurrences of variables A, B by new fresh variables A', B'.

$$@_{s,s'}\ A\ B\ (\lambda_{s,s'}\ A'\ B'\ t)\ u \longrightarrow t\ u \qquad \text{for } (s, s', s'') \in \mathfrak{R}$$

Note that the left-hand side is not well-typed anymore, but this is not a problem. Indeed, the important property we can show is that for all well-typed instances of the left-hand side, A becomes convertible to A' and B becomes convertible to B', which then guarantees that the corresponding instance of the right-hand side is well-typed with the same type as the left one.

This concludes the definition of the theory, which is summarized in Figure 2. In the following, we adopt some conventions to make the notation lighter. First, we write $\Pi_{s,s'} x : A.B$ for $\Pi_{s,s'}\ A\ (x.B)$, or $A \rightsquigarrow_{s,s'} B$ when $x \notin \mathsf{fv}(B)$. Moreover, in order to keep examples readable it will be essential to treat some arguments as implicit: namely, we will write $\mathsf{Tm}\ A$ instead of $\mathsf{Tm}_s\ A$, $\Pi x : A.B$ instead of $\Pi_{s,s'} x : A.B$, $t_{@} u$ instead of $@_{s,s'}\ A\ B\ t\ u$ and $\lambda x.t$ instead of $\lambda_{s,s'}\ A\ B\ (x.t)$. Nevertheless, note that this is just an informal notation used in examples, and in actual Dedukti terms all these arguments should be spelled out — alternatively, one can switch to a framework with support for *erased arguments*, such as [Fel24].

Remark 2.3. In the Pure Type Systems (PTS) [Bar93] literature, the triple $(\mathfrak{S}, \mathfrak{A}, \mathfrak{R})$ is known as a (functional) *PTS specification*, and each such specification defines a PTS. The main result of [Fel22] is that the theory in Figure 2 defines an adequate encoding of the associated PTS in Dedukti.

3. An informal look at 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

simplistic, they showcase real problems we found during our first predicativization attempt of Fermat's little theorem library in HOL — some of them being already noted in [Del20].

We first start by defining the impredicative theory $\mathbb{I}$ and the predicative theory $\mathbb{P}$, which will serve respectively as source and target of our proof transformation. They are defined by instantiating the theory of Figure 2 with the following specifications — following Remark 2.3, they can equivalently be seen as the Pure Type Systems defined by these specifications.

$$\mathfrak{S}_{\mathbb{I}} := \{\Omega, \Box\} \qquad \mathfrak{S}_{\mathbb{P}} := \mathbb{N}$$
$$\mathfrak{A}_{\mathbb{I}} := \{(\Omega, \Box)\} \qquad \mathfrak{A}_{\mathbb{P}} := \{(n, n+1) \mid n \in \mathbb{N}\}$$
$$\mathfrak{R}_{\mathbb{I}} := \{(\Omega, \Omega, \Omega), (\Box, \Omega, \Omega), (\Box, \Box, \Box)\} \qquad \mathfrak{R}_{\mathbb{P}} := \{(n, m, \mathsf{max}\{n, m\}) \mid n, m \in \mathbb{N}\}$$

Note that in the theory $\mathbb{I}$ we have $(\Box, \Omega, \Omega) \in \mathfrak{R}_{\mathbb{I}}$, and thus for $\Gamma \vdash A : \mathsf{Ty}_{\Box}$ and $\Gamma, x : \mathsf{Tm}\ A \vdash B : \mathsf{Ty}_{\Omega}$ we have $\Gamma \vdash \Pi x : A.B : \mathsf{Ty}_{\Omega}$. Therefore, the sort Ω is closed under dependent products indexed over types in $\Box$, a larger sort, so $\mathbb{I}$ is indeed an impredicative theory. Finally, we note that $\mathbb{P}$ is a subtheory of the one implemented in Agda, whereas $\mathbb{I}$ is a subtheory of the ones implemented in Coq, Matita, Isabelle, etc, justifying why they are of interest.

In the following, let Φ be a signature without confined constant declarations. We say that Φ is well-formed in a theory $\mathbb{T}$ when the theory $(\Sigma', \mathcal{R}_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}})$ is well-formed, where $\Sigma' := \Sigma_{\mathbb{T}}, \Phi$. We then call Φ a *local* signature, in contrast to $\Sigma_{\mathbb{T}}$ which is *global*. We write names of constants in the local signature in sans serif black in order to distinguish them from names in the global signature, which are still written in sans serif blue.

Then, the problem of proof predicativization consists in defining a transformation such that, given a local signature Φ well-formed in $\mathbb{I}$, we obtain a local signature Φ' well-formed in $\mathbb{P}$ — a suitable transformation should of course preserve the structure of the statements in Φ, however we leave the precise relationship between Φ and Φ' vague at this point. Stated informally, we would like to translate constants declarations $c : A$ (which represent axioms) and constant definitions $c : A := t$ (which also represent proofs) from $\mathbb{I}$ to $\mathbb{P}$ — confined constant declarations $f : \Delta \to A$ cannot occur in Φ and will be of no interest here. 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, as we shall see, turns out to be a major issue for the translation.

Now that our basic notions are explained, let us try to predicativize proofs. For our first step, consider a very simple development showing that for every object term A in U_{Ω} we can build an object term in $A \rightsquigarrow A$ — this is actually just the polymorphic identity function at sort Ω. Here we adopt an Agda-like syntax to display entries of the local signature.

$$\mathsf{id} : \mathsf{Tm}\ (\Pi A : \mathsf{U}_{\Omega}.A \rightsquigarrow A)$$
$$\mathsf{id} := \lambda A.\lambda x.x$$

To translate this simple development, the first idea that comes to mind is to define a mapping on sorts: the sort Ω is mapped to 0 and the universe $\Box$ is mapped to 1. If this mapping defined a *specification morphism*[4] then this transformation would always produce a valid definition in $\mathbb{P}$ [Geu93, Lemma 4.2.6]. Unfortunately, it is easy to check that it does not define a specification morphism (worse, no function $\mathfrak{S}_{\mathbb{I}} \to \mathfrak{S}_{\mathbb{P}}$ defines a specification morphism). Nevertheless, this does not mean that it cannot produce something well-typed

[4]That is, if $(s, s') \in \mathfrak{A}_{\mathbb{I}}$ implied $(\phi(s), \phi(s')) \in \mathfrak{A}_{\mathbb{P}}$ and $(s, s', s'') \in \mathfrak{R}_{\mathbb{I}}$ implied $(\phi(s), \phi(s'), \phi(s'')) \in \mathfrak{R}_{\mathbb{P}}$, where $\phi : \mathfrak{S}_{\mathbb{I}} \to \mathfrak{S}_{\mathbb{P}}$ is the sort mapping.

in $\mathbb{P}$ in *some* cases. For instance, by applying it to id we get the following entry,[5] which is actually well-typed in $\mathbb{P}$.

$$\begin{aligned}&\mathsf{id} : \mathsf{Tm}\ (\Pi A : \mathsf{U}_0.A \rightsquigarrow A)\\&\mathsf{id} := \lambda A.\lambda x.x\end{aligned}$$

This naïve approach however quickly fails when considering other cases. For instance, suppose now that one adds the following definition.

$$\begin{aligned}&\mathsf{id\text{-}to\text{-}id} : \mathsf{Tm}\ ((\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A))\\&\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A)\end{aligned}$$

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

$$\begin{aligned}&\mathsf{id\text{-}to\text{-}id} : \mathsf{Tm}\ ((\Pi A : \mathsf{U}_0.A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathsf{U}_0.A \rightsquigarrow A))\\&\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathsf{U}_0.A \rightsquigarrow A)\end{aligned}$$

However, one can verify that this term is not well typed. Indeed, in the original term one quantifies over all elements of U_Ω in $\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A$, and because of impredicativity this object term stays at U_Ω. However, in $\mathbb{P}$ quantifying over all elements of the universe U_0 in $\Pi A : \mathsf{U}_0.A \rightsquigarrow A$ lifts its overall object type to U_1. As id expects something of object type U_0, then $\mathsf{id}_{@}(\Pi A : \mathsf{U}_0.A \rightsquigarrow A)$ is not well-typed.

The takeaway lesson from this first try is that impredicativity introduces a kind of *typical ambiguity*, as it allows us to put in a single universe U_Ω types which, in a predicative setting, would have to be stratified and placed in larger universes. Therefore, we should not translate every occurrence of Ω as 0 naively as we did, but try to compute for each occurrence of Ω some natural number n such that replacing it by n would produce a valid term in $\mathbb{P}$. In other words, we should erase all sort information and then *elaborate* it into a well-typed term in $\mathbb{P}$.

Thankfully, performing such kind of transformations is exactly the goal of the tool Universo [Thi20]. To understand how it works, let us come back to the previous example. Universo starts here by replacing each sort by a fresh metavariable representing a natural number.

$$\begin{aligned}&\mathsf{id} : \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A)\\&\mathsf{id} := \lambda A.\lambda x.x\\&\mathsf{id\text{-}to\text{-}id} : \mathsf{Tm}\ ((\Pi A : \mathsf{U}_{i_2}.A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A))\\&\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathsf{U}_{i_4}.A \rightsquigarrow A)\end{aligned}$$

Then, in the following step Universo tries to elaborate the term into a well-typed one in $\mathbb{P}$. To do so, it first tries to typecheck it 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 $\mathbb{P}$. For instance, applying Universo to our previous example would produce the following local signature, which is indeed valid

[5] Modulo the recomputation of some omitted sort annotations.

with respect to $\mathbb{P}$.

$$\begin{aligned}
&\mathsf{id} : \mathsf{Tm}\ (\Pi A : \mathsf{U}_1.A \rightsquigarrow A)\\
&\mathsf{id} := \lambda A.\lambda x.x\\
&\mathsf{id\text{-}to\text{-}id} : \mathsf{Tm}\ ((\Pi A : \mathsf{U}_0.A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathsf{U}_0.A \rightsquigarrow A))\\
&\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathsf{U}_0.A \rightsquigarrow A)
\end{aligned}$$

By using Universo it is possible to go much further than with the naïve method shown before. Still, this approach also fails when being employed with real libraries. To see the reason, consider the following minimum example, in which one uses id twice to build another element of the same type.

$$\begin{aligned}
&\mathsf{id}' : \mathsf{Tm}\ (\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A)\\
&\mathsf{id}' := \mathsf{id}_{@}(\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A)_{@}\mathsf{id}
\end{aligned}$$

If we repeat the same procedure as before, we get the following entries, which when type checked generate unsolvable constraints.

$$\begin{aligned}
&\mathsf{id} : \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A)\\
&\mathsf{id} := \lambda A.\lambda x.x\\
&\mathsf{id}' : \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_2}.A \rightsquigarrow A)\\
&\mathsf{id}' := \mathsf{id}_{@}(\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A)_{@}\mathsf{id}
\end{aligned}$$

The reason is that the application $\mathsf{id}_{@}(\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A)_{@}\mathsf{id}$ forces i_1 to be both i_3 and $i_3 + 1$, which is of course impossible. Therefore, the takeaway lesson from this second try is that impredicativity does not only hide the fact that types need to be stratified, but also the fact that they need to be usable at multiple levels of this stratification. Indeed, in our example we would like to use id at $\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A$ and $\Pi A : \mathsf{U}_{i_3+1}.A \rightsquigarrow A$. In practice, 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 should not translate entries by fixing once and for all their universes, but instead we should let them vary by using *universe polymorphism* [HP91, ST14]. This feature, present in some type theories (and also in the one of Agda [Tea]), allows defining terms containing universe variables, which can later be instantiated at various concrete universes.

Our translation will then work by first computing for each definition or declaration its set of constraints. However, instead of assigning concrete values to metavariables, we perform *unification* which allows us to solve constraints in a symbolic way. The result will then be a universe-polymorphic term, which will be usable at multiple universes when translating the next entries. In order to define this formally, we first start in the next section by refining the target type theory $\mathbb{P}$ with universe polymorphism.

4. A universe-polymorphic predicative theory

In this section we define $\mathbb{UPP}$, a theory which refines $\mathbb{P}$ by internalizing sort annotations and allowing for prenex universe polymorphism [HP91, Gen20]. This is in particular a subsystem of the one underlying the Agda proof assistant [Tea].

The main change with respect to $\mathbb{P}$ is that, instead of indexing constants externally, we index them inside the framework [Ass15, Ste19]. To do this, we first introduce in Dedukti a syntax for *universe levels* (which is just the terminology used in the literature for predicative sorts, i.e. $\mathbb{N}$), using the following declarations. Note that the constants $\mathsf{0}, \mathsf{S}, \sqcup$ are declared as confined, which will be needed later to show the confluence of the theory.

$$\mathsf{Lvl} : \mathsf{Type} \qquad \mathsf{S} : (i : \mathsf{Lvl}) \to \mathsf{Lvl} \quad \text{(confined)}$$
$$\mathsf{0} : () \to \mathsf{Lvl} \quad \text{(confined)} \qquad \sqcup : (i : \mathsf{Lvl}, i' : \mathsf{Lvl}) \to \mathsf{Lvl} \quad \text{(confined)}$$

In the following, we write $\sqcup$ in infix notation, S in curryfied notation, and consider $\sqcup$ as having a lower precedence than S — for instance, $\mathsf{S}\ i \sqcup \mathsf{S}\ j$ should be parsed as $\sqcup\,(\mathsf{S}(i), (\mathsf{S}(j)))$. These constant declarations yield the following grammar of confined terms, which from now on we refer to as *levels*.

$$l, l' ::= i \mid \mathsf{0} \mid \mathsf{S}\ l \mid l \sqcup l'$$

The definitions of Figure 2 are then replaced by the following ones. Note that the two rewrite rules are presented in linearized form, in order for them to be left-linear.

$$\begin{aligned}
&\mathsf{Ty} : (l : \mathsf{Lvl}) \to \mathsf{Type}\\
&\mathsf{Tm} : (l : \mathsf{Lvl}) \to \mathsf{Ty}\ l \to \mathsf{Type}\\
&\mathsf{U} : (l : \mathsf{Lvl}) \to \mathsf{Ty}\ (\mathsf{S}\ l)\\
&\mathsf{Tm}\ l'\ (\mathsf{U}\ l) \longrightarrow \mathsf{Ty}\ l\\
&\Pi : (l\ l' : \mathsf{Lvl}) \to (A : \mathsf{Ty}\ l) \to (B : \mathsf{Tm}\ l\ A \to \mathsf{Ty}\ l') \to \mathsf{Ty}\ (l \sqcup l')\\
&\lambda : (l\ l' : \mathsf{Lvl}) \to (A : \mathsf{Ty}\ l) \to (B : \mathsf{Tm}\ l\ A \to \mathsf{Ty}\ l') \to\\
&\quad ((x : \mathsf{Tm}\ l\ A) \to \mathsf{Tm}\ l'\ (B\ x)) \to \mathsf{Tm}\ (l \sqcup l')\ (\Pi\ l\ l'\ A\ B)\\
&@ : (l\ l' : \mathsf{Lvl}) \to (A : \mathsf{Ty}\ l) \to (B : \mathsf{Tm}\ l\ A \to \mathsf{Ty}\ l') \to\\
&\quad (t : \mathsf{Tm}\ (l \sqcup l')\ (\Pi\ l\ l'\ A\ B)) \to (u : \mathsf{Tm}\ l\ A) \to \mathsf{Tm}\ l'\ (B\ u)\\
&@\ l\ l'\ A\ B\ (\lambda\ l''\ l'''\ A'\ B'\ t)\ u \longrightarrow t\ u
\end{aligned}$$

In the following, we adopt a subscript notation for levels and write Ty_l, Tm_l, U_l, $\Pi_{l,l'}$, $\lambda_{l,l'}$ and $@_{l,l'}$ to improve clarity. We also continue to write $\Pi_{l,l'} x : A.B$ for $\Pi_{l,l'}\ A\ (x.B)$ or $A \rightsquigarrow_{l,l'} B$ when $x \notin \mathsf{fv}(B)$. Finally, in order to keep examples readable we also reuse our convention for implicit arguments and write $\mathsf{Tm}\ A$ for $\mathsf{Tm}_l\ A$, $\Pi x : A.B$ for $\Pi_{l,l'} x : A.B$, $\lambda x.t$ for $\lambda_{l,l'}\ A\ B\ (x.t)$, and $t_{@} u$ for $@_{l,l'}\ A\ B\ t\ u$.

Universe polymorphism can now be represented directly with the use of the framework's function type [Ass15]. 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 4.1. The universe-polymorphic identity function is given by

$$\begin{aligned}
&\mathsf{id} : (i : \mathsf{Lvl}) \to \mathsf{Tm}\ (\Pi A : \mathsf{U}_i.A \rightsquigarrow A)\\
&\mathsf{id} := i.\lambda A.\lambda a.a
\end{aligned}$$

This then allows to use id at any universe level: for instance, we can obtain the polymorphic identity function at the level $\mathsf{0}$ with the application $\mathsf{id}\ \mathsf{0}$, which has type $\mathsf{Tm}\ (\Pi A : \mathsf{U}_0.A \rightsquigarrow A)$.

Remark 4.2. Note that unlike some other proposals [BCDE23] there is no object-level operation for universe level abstraction, which is instead handled by the framework function type. Therefore, universe-polymorphic definitions are best understood as schemes rather than actual terms in the object logic.

In order to finish the definition of the theory we need to specify the definitional equality satisfied by levels, which is the one generated by the following equations [Tea].[6] This then concludes the definition of the theory $\mathbb{UPP}$.

$$i_1 \sqcup (i_2 \sqcup i_3) \simeq (i_1 \sqcup i_2) \sqcup i_3 \qquad \mathsf{S}\ (i_1 \sqcup i_2) \simeq \mathsf{S}\ i_1 \sqcup \mathsf{S}\ i_2 \qquad i \sqcup 0 \simeq i$$
$$i_1 \sqcup i_2 \simeq i_2 \sqcup i_1 \qquad i \sqcup \mathsf{S}\ i \simeq \mathsf{S}\ i \qquad i \sqcup i \simeq i$$

As we will see later with Proposition 6.7, the definition of $\simeq$ ensures us that two levels are convertible exactly when they are arithmetically equivalent, allowing us for instance to exchange $\mathsf{S}\ i \sqcup i \sqcup 0$ and $\mathsf{S}\ i$.

4.1. **Metatheory of $\mathbb{UPP}$.** We now look at the metatheory of $\mathbb{UPP}$.

Proposition 4.3. *The theory $\mathbb{UPP}$ is well-formed.*

Proof. Can be easily verified manually, or with the help of Lambdapi [Deda]. □

In the following, we consider $\mathbb{UPP}$ extended with an arbitrary local signature Φ well-formed in $\mathbb{UPP}$. Therefore, the following δ rules are the ones of Φ.

Proposition 4.4. *The rewrite system $\mathcal{R}_{\mathbb{UPP}}$ is confluent together with the rules β, β_c and δ.*

Proof. Follows from the fact that the rewrite rules define an orthogonal combinatory rewrite system [Kvv93]. □

The following basic property is similar to [Voe14, Lemma 4.1.6] and shows that $\longrightarrow$ and $\simeq$ interact well.

Proposition 4.5. *The relation $\simeq$ is a simulation with respect to $\longrightarrow$. Diagrammatically,*

$$\begin{array}{ccc} t & \simeq & u \\ \downarrow & & \downarrow \\ \exists t' & \simeq & u' \end{array}$$

Proof. By induction on the rewrite context of $u \longrightarrow u'$. The induction steps are easy, we only show the base cases.

(1) If $u \longrightarrow u'$ with a δ rule, then u is a non-confined constant, so $u = t$ and $t \longrightarrow u'$.
(2) If $u = \mathsf{Tm}_{l_1}\ \mathsf{U}_{l_2} \longrightarrow \mathsf{Ty}_{l_2} = u'$, then we have $t = \mathsf{Tm}_{l'_1}\ \mathsf{U}_{l'_2}$ with $l_k \simeq l'_k$ for $k = 1, 2$. Therefore, $t \longrightarrow \mathsf{Ty}_{l'_2} \simeq \mathsf{Ty}_{l_2} = u'$.
(3) If $u = @_{l_1,l_2}\ A_1\ B_1\ (\lambda_{l_3,l_4}\ A_2\ B_2\ v)\ w \longrightarrow v\ w = u'$, then $t = @_{l'_1,l'_2}\ A'_1\ B'_1\ (\lambda_{l'_3,l'_4}\ A'_2\ B'_2\ v')\ w'$ with $v \simeq v'$, $w \simeq w'$ and other relations that we will not need. Therefore, $t \longrightarrow v'\ w' \simeq v\ w = u'$.
(4) If $u = (x.v)w \longrightarrow v[x \mapsto w]$, then $t = (x.v')w'$ with $v \simeq v'$ and $w \simeq w'$. Therefore, $t \longrightarrow v'[x \mapsto w']$ and $v'[x \mapsto w'] \simeq v[x \mapsto w]$ follows by stability under substitution.

[6] Some authors consider a version of this theory without the neutral element 0 [BCDE23]; here we stick to the variant used in Agda, which includes the 0.

(5) If $u = (i.v)l \longrightarrow v[i \mapsto l]$, then $t = (i.v')l'$ with $v \simeq v'$ and $l \simeq l'$. Therefore, $t \longrightarrow v'[i \mapsto l']$ and $v'[i \mapsto l'] \simeq v[i \mapsto l]$ follows by stability under substitution. □

Remark 4.6. We remark that the use of confinement was essential in the previous proof. Indeed, had $\sqcup, \mathsf{S}, 0$ been declared as regular constants, their associated equations would also have to be declared with regular variables, and we would have $x \simeq x \sqcup x$. Then, this equation could be used in cases in which x is instantiated by redexes: for instance, we could have

$$\mathsf{Tm}_0\ \mathsf{U}_0 \simeq \mathsf{Tm}_0\ \mathsf{U}_0 \sqcup \mathsf{Tm}_0\ \mathsf{U}_0 \longrightarrow \mathsf{Ty}_0 \sqcup \mathsf{Tm}_0\ \mathsf{U}_0$$

But then Ty_0 is the only reduct of $\mathsf{Tm}_0\ \mathsf{U}_0$, yet $\mathsf{Ty}_0 \simeq \mathsf{Ty}_0 \sqcup \mathsf{Tm}_0\ \mathsf{U}_0$ does not hold, showing that Proposition 4.5 fails in this setting. One could argue that this is not a problem because the term $\mathsf{Tm}_0\ \mathsf{U}_0 \sqcup \mathsf{Tm}_0\ \mathsf{U}_0$ is ill-typed, however in Proposition 4.5 we do not ask terms to be well-typed. This is because this property is the main lemma for showing Church-Rosser, which in turn is needed for proving subject reduction. So without having subject reduction at this point, we cannot yet rely on the fact that terms are well-typed.

We will need the following simple property about abstract rewriting. Recall that an abstract equational rewrite system $(\blacktriangleright, \sim)$ is given by a binary relation $\blacktriangleright \subseteq X^2$ and an equivalence relation $\sim \subseteq X^2$. Then $(\blacktriangleright, \sim)$ is said to be *Church-Rosser modulo* if $x\ (\blacktriangleright \cup \blacktriangleleft \cup \sim)^*\ y$ implies $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft\ y$, where we write juxtaposition for composition of relations and $\blacktriangleleft$ for the inverse of $\blacktriangleright$.

Proposition 4.7. *Let $(\blacktriangleright, \sim)$ be an abstract equational rewrite system. If $\blacktriangleright$ is confluent and $\sim$ is a simulation for $\blacktriangleright$, then $(\blacktriangleright, \sim)$ is Church-Rosser modulo.*

Proof. If $x\ (\blacktriangleright \cup \blacktriangleleft \cup \sim)^*\ y$, then we have $x\ (\blacktriangleright \cup \blacktriangleleft \cup \sim)^n\ y$ for some n. We prove the result by induction on n, the base case being trivial. For the inductive step, we have

$$x\ (\blacktriangleright \cup \blacktriangleleft \cup \sim)^n\ z\ (\blacktriangleright \cup \blacktriangleleft \cup \sim)\ y$$

for some z. First note that by i.h. we have $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft\ z$. We now have three possibilities:

(1) $z \blacktriangleright y$: We have $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft \blacktriangleright\ y$, and so by confluence we have $x \blacktriangleright^* \sim \blacktriangleright^* {}^*\!\blacktriangleleft\ y$. Using the fact that $\sim$ is a simulation with respect to $\blacktriangleright$, we then get $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft\ y$.
(2) $z \blacktriangleleft y$: We have $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft \blacktriangleleft\ y$, and thus $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft\ y$.
(3) $z \sim y$: We have $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft \sim y$, thus using the fact that $\sim$ is a simulation with respect to $\blacktriangleright$, we get $x \blacktriangleright^* \sim {}^*\!\blacktriangleleft\ y$. □

Corollary 4.8 (Church-Rosser modulo)**.** *If $t \equiv u$ then $t \longrightarrow^* t' \simeq u'\ {}^*\!\longleftarrow u$.*

Proof. Direct consequence of Propositions 4.4 and 4.5 and 4.7. □

The previous corollary has two important consequences. First, in order to check $t \equiv u$ we do not need to employ matching modulo $\simeq$, but instead only regular syntactic matching. This is important because it means that in order to decide $\equiv$ we do not need to design a specific matching algorithm for $\simeq$, but only to decide $\simeq$ (which is indeed decidable by Theorem 6.6) and to show $\longrightarrow$ to be strongly normalizing for well-typed terms (a property we conjecture to be true). Second, using Church-Rosser modulo we can prove subject reduction, a property that will be essential to show soundness of elaboration (in particular, it is used in Theorem 5.5).

Proposition 4.9 (Subject Reduction)**.** *If $\Gamma \vdash t : A$ and $t \longrightarrow t'$ then $\Gamma \vdash t' : A$.*

Proof. From Church-Rosser modulo we get the injectivity of the framework's dependent function type, so from Proposition 2.2 we conclude subject reduction of δ, β and β_c.

By Proposition 2.2 once again, to show subject reduction for $\mathcal{R}_{\mathbb{UPP}}$ it suffices to prove that all of its rewrite rules preserve typing, which easily follows from inversion of typing and Church-Rosser modulo. □

Remark 4.10. The proof of subject reduction relies on Church-Rosser modulo, which in turn is shown using Proposition 4.5. As discussed in Remark 4.6, the proof of this proposition crucially relies on the use of confinement, so it is less clear how to derive Church-Rosser modulo in a setting without it. Because $\mathcal{R}_{\mathbb{UPP}}$ is left-linear and there are no critical pairs between $\mathcal{R}_{\mathbb{UPP}}$ and $\mathcal{E}_{\mathbb{UPP}}$, nor between $\mathcal{R}_{\mathbb{UPP}}$ and itself, one possibility would be to apply [MN98, Theorem 5.11] to show the restriction of $\equiv$ to strongly normalizing (s.n.) terms to be Church-Rosser modulo. However, to show Church-Rosser modulo for the well-typed restriction of $\equiv$ and then subject reduction we would then need to show $\longrightarrow$ to be s.n. for well-typed terms. The use of confinement is thus an interesting alternative to this option, as it allows us to establish the main correctness property of elaboration (Theorem 5.5) without relying on strong normalization.

5. Universe-polymorphic elaboration

We are now ready to define the (partial) transformation of a local signature Φ to the theory $\mathbb{UPP}$. As hinted at the end of Section 3, our translation works by incrementally trying to elaborate each entry of Φ into a universe-polymorphic one in $\mathbb{UPP}$, such that at the end we get a local signature Φ' well-formed in $\mathbb{UPP}$, if no errors are produced in the process.

In order to explain it, we now suppose that we have already translated a local signature Φ to a local signature Φ' well-formed in $\mathbb{UPP}$, and try to translate a new entry either of the form $c : A$ or $c : A := t$. To illustrate the steps of the process, we will make use of a running example. Note that, following our previously established convention, we keep some arguments implicit in order to improve readability.

Example 5.1. The last example of Section 3 is the following local signature, which is well-formed in $\mathbb{I}$.

$$\begin{aligned}
&\mathsf{id} : \mathsf{Tm}\ (\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A)\\
&\mathsf{id} := \lambda A.\lambda x.x\\
&\mathsf{id}' : \mathsf{Tm}\ (\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A)\\
&\mathsf{id}' := \mathsf{id}_{@}(\Pi A : \mathsf{U}_\Omega.A \rightsquigarrow A)_{@}\mathsf{id}
\end{aligned}$$

Let us call Φ the local signature containing only the first entry and suppose that it has already been translated, giving the following local signature Φ'.

$$\begin{aligned}
&\mathsf{id} : (i : \mathsf{Lvl}) \to \mathsf{Tm}\ (\Pi A : \mathsf{U}_i.A \rightsquigarrow A)\\
&\mathsf{id} := i.\lambda A.\lambda x.x
\end{aligned}$$

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

$$
\begin{aligned}
&|x| := x \\
&|t\ u| := |t||u| \\
&|x.t| := x.|t| \\
&|(x : A) \to B| := (x : |A|) \to |B| \\
&|\mathsf{Type}| := \mathsf{Type} \\
&|\mathsf{Kind}| := \mathsf{Kind} \\
&|c| := c\ i_1..i_k \quad \text{if } c : (j_1..j_k : \mathsf{Lvl}) \to A \in \Phi'
\end{aligned}
\qquad
\begin{aligned}
&|\mathsf{Ty}_s| := \mathsf{Ty}\ i \\
&|\mathsf{Tm}_s| := \mathsf{Tm}\ i \\
&|\mathsf{U}_s| := \mathsf{U}\ i \\
&|\Pi_{s,s'}| := \Pi\ i\ i' \\
&|\lambda_{s,s'}| := \lambda\ i\ i' \\
&|@_{s,s'}| := @\ i\ i'
\end{aligned}
$$

Figure 3: Translation to the syntax of schematic terms
(each inserted level variable is assumed to be different from the previous ones)

5.1. **Schematic terms.** The source syntax of our elaborator will be a subset of the one defined by $\mathbb{UPP}$ extended with Φ'. The only allowed levels will be confined variables i, appearing as arguments of constant applications, and there should be no occurrences of confined abstractions $i.t$ or confined function types $(i : A) \to B$ — we henceforth call such terms *schematic terms*. Therefore, in order to translate proofs from $\mathbb{I}$ the first step is defining a translation to the syntax of schematic terms, given by Figure 3. Note that the translation is not defined for terms of the form $t\ l$ or $i.t$ or $(i : A) \to B$ because these are not used in the theory $\mathbb{I}$. Moreover, because the first step is erasing all sort information, this actually defines a translation starting from any theory defined by instantiating Figure 2 with a specification. Therefore, it can also be applied to theories using much more complex universe hierarchies, such as those of the proof assistants Matita or Coq.

Let us explain the translation of Figure 3. First, all of the constants in the definition of $\mathbb{I}$ are mapped to their correspondents in $\mathbb{UPP}$, except that they are also applied to fresh level variables which are to be solved during elaboration. Then, constants from Φ are translated as they are, except that they are also given fresh level variables in order to fill in the number of level arguments that they expect. Note that this is necessary because the translation of the entries preceding the current one introduced new dependencies on level variables which were not present in $\mathbb{I}$.

Example 5.2. When applying $|-|$ to id' we get the following entry.

$$
\begin{aligned}
&\mathsf{id}' : \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A) \\
&\mathsf{id}' := (\mathsf{id}\ i_2)_{@}(\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)
\end{aligned}
$$

Note that the occurrences of U_Ω have been replaced by U_i (which is just a notation for $\mathsf{U}\ i$) for some fresh i. Moreover, because the type of id in Φ' is $(i : \mathsf{Lvl}) \to \mathsf{Tm}\ (\Pi A : \mathsf{U}_i.A \rightsquigarrow A)$, each occurrence of id is replaced by $\mathsf{id}\ i$ for some fresh i. Finally, note that because of implicit arguments we are hiding many new variables that were also inserted. For instance, the subterm $\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A$ is an implicit notation for $\Pi_{i_5,i_6} A : \mathsf{U}_{i_1}.A \rightsquigarrow_{i_7,i_8} A$. However, the term $(\mathsf{id}\ i_2)_{@}(\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)$ without implicit arguments would not even fit into a line, so for readability reasons we will not write it fully.

5.2. **Computing constraints.** Our next step is then to define a type system for computing constraints. This is done by adapting the seminal work of Harper and Pollack [HP91], with the difference that our system will be *bidirectional*.

In regular bidirectional type systems, the typing judgment $\Gamma \vdash t : A$ is split into modes infer $\Gamma \vdash t \Rightarrow A$ and check $\Gamma \vdash t \Leftarrow A$. In mode infer we start with Γ, t and we should find a type A for t in Γ, whereas in mode check we are given Γ, t, A and we should check that t indeed has type A in Γ. Crucial in bidirectional typing is the proper bookkeeping of pre-conditions and post-conditions, which are summarized in the following table — there, we mark inputs with $-$ and outputs with $+$.

Judgment	Pre-condition	Post-condition
$\Gamma^- \vdash t^- \Rightarrow A^+$	$\Gamma \vdash$	$\Gamma \vdash t : A$
$\Gamma^- \vdash t^- \Leftarrow A^-$	$\Gamma \vdash A : s$	$\Gamma \vdash t : A$

In order to refine these judgments with constraints, let us start with some preliminary definitions. A *(level unification) problem* $\mathcal{C}$ is a set containing *(level) constraints* of the form $l \stackrel{?}{=} l'$, referred to also as *equations*. In the following definitions, let θ be a *level substitution*, that is, one whose domain only contains confined variables. We write $\theta \models \mathcal{C}$ when $l[\theta] \simeq l'[\theta]$ for all $l \stackrel{?}{=} l' \in \mathcal{C}$, in which case θ is called a *unifier* (or *solution*) for $\mathcal{C}$. Let Ξ_θ be a context containing $j : \mathsf{Lvl}$ for each level variable appearing in $i[\theta]$, for each level variable i inserted in the schematic terms. We then also write $\Gamma \vdash_{\mathcal{C}}$ when $\theta \models \mathcal{C}$ implies $\Xi_\theta, \Gamma[\theta] \vdash$ for all θ, and $\Gamma \vdash_{\mathcal{C}} t : A$ when $\theta \models \mathcal{C}$ implies $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : A[\theta]$ for all θ.

Intuitively, just like the context Γ in $\Gamma \vdash t : A$ allows us to state a typing judgment $t : A$ with typing hypotheses of the form $x : B \in \Gamma$, the set of constraints $\mathcal{C}$ in $\Gamma \vdash_{\mathcal{C}} t : A$ refines this with equational hypotheses of the form $l \simeq l'$. With this in mind, we can now give the new typing judgments in the following table. Compared with the previous table, we now start with a set of constraints $\mathcal{D}$, which guarantees that the pre-condition holds, and in the process we must also find constraints $\mathcal{C}$ ensuring that the post-condition holds.

Judgment	Pre-condition	Post-condition
$\Gamma^- \uparrow \mathcal{D}^- \vdash t^- \Rightarrow A^+ \downarrow \mathcal{C}^+$	$\Gamma \vdash_{\mathcal{D}}$	$\Gamma \vdash_{\mathcal{C} \cup \mathcal{D}} t : A$
$\Gamma^- \uparrow \mathcal{D}^- \vdash t^- \Leftarrow A^- \downarrow \mathcal{C}^+$	$\Gamma \vdash_{\mathcal{D}} A : s$	$\Gamma \vdash_{\mathcal{C} \cup \mathcal{D}} t : A$

We now come to the definition of the bidirectional type system for computing constraints, given in Figure 4 — we mark inputs with $-$ and outputs with $+$. Note that it also relies on two new conversion judgments $A \equiv B \downarrow \mathcal{C}$ and $A \equiv_{\text{whnf}} B \downarrow \mathcal{C}$ used to compute a set of constraints needed for the conversion to hold. In the given rules, we write $t \longrightarrow^{\text{wh}} u$ for the reduction of t to a weak-head normal form (whnf) u, meaning that $t \longrightarrow^* u$ and, if $u \longrightarrow^* u'\ a_1\ \ldots\ a_k$ with k being possibly 0 and each of the $a_1, \ldots, a_k$ being a regular or confined term, then u' matches no rewrite rule left-hand side (including β, β_{c} and δ).

Example 5.3. In order to compute the constraints of the entry

$$\mathsf{id}' : \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A)$$
$$\mathsf{id}' := (\mathsf{id}\ i_2)_{@}(\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)$$

we first compute the constraints necessary for its type to be of type Type:

$$() \uparrow \varnothing \vdash \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A) \Leftarrow \mathsf{Type} \downarrow \mathcal{C}_1$$

$\boxed{A^- \equiv B^- \downarrow \mathcal{C}^+}$

$$\frac{A \longrightarrow^{\text{wh}} A' \qquad B \longrightarrow^{\text{wh}} B' \qquad A' \equiv_{\text{whnf}} B' \downarrow \mathcal{C}}{A \equiv B \downarrow \mathcal{C}}$$

$\boxed{A^- \equiv_{\text{whnf}} B^- \downarrow \mathcal{C}^+}$

$$\frac{t = x, c, s}{t \equiv_{\text{whnf}} t \downarrow \varnothing} \qquad \frac{t \equiv t' \downarrow \mathcal{C}}{x.t \equiv_{\text{whnf}} x.t' \downarrow \mathcal{C}} \qquad \frac{A \equiv A' \downarrow \mathcal{C}_1 \qquad B \equiv B' \downarrow \mathcal{C}_2}{(x : A) \to B \equiv_{\text{whnf}} (x : A') \to B' \downarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{t \equiv_{\text{whnf}} t' \downarrow \mathcal{C}_1 \qquad u \equiv u' \downarrow \mathcal{C}_2}{t\ u \equiv_{\text{whnf}} t'\ u' \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \qquad \frac{t \equiv_{\text{whnf}} t' \downarrow \mathcal{C}}{t\ l \equiv_{\text{whnf}} t'\ l' \downarrow \mathcal{C} \cup \{l \stackrel{?}{=} l'\}}$$

$\boxed{\Gamma^- \uparrow \mathcal{D}^- \vdash t^- \Leftarrow A^- \downarrow \mathcal{C}^+}$

$$\textsc{Switch}\ \frac{\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow A \downarrow \mathcal{C}_1 \qquad A \equiv B \downarrow \mathcal{C}_2}{\Gamma \uparrow \mathcal{D} \vdash t \Leftarrow B \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \qquad \textsc{Abs}\ \frac{C \longrightarrow^{\text{wh}} (x : A) \to B \qquad \Gamma, x : A \uparrow \mathcal{D} \vdash t \Leftarrow B \downarrow \mathcal{C}}{\Gamma \uparrow \mathcal{D} \vdash x.t \Leftarrow C \downarrow \mathcal{C}}$$

$\boxed{\Gamma^- \uparrow \mathcal{D}^- \vdash t^- \Rightarrow A^+ \downarrow \mathcal{C}^+}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \uparrow \mathcal{D} \vdash x \Rightarrow A \downarrow \varnothing} \qquad \textsc{Cons}\ \frac{c : A \in \Sigma_{\mathbb{UPP}}, \Phi' \text{ or } c : A := t \in \Sigma_{\mathbb{UPP}}, \Phi' \qquad A = (j_1..j_k : \mathsf{Lvl}) \to B}{\Gamma \uparrow \mathcal{D} \vdash c\ i_1..i_k \Rightarrow B[\vec{j} \mapsto \vec{i}] \downarrow \varnothing}$$

$$\textsc{Sort}\ \frac{}{\Gamma \uparrow \mathcal{D} \vdash \mathsf{Type} \Rightarrow \mathsf{Kind} \downarrow \varnothing} \qquad \textsc{Pi}\ \frac{\Gamma \uparrow \mathcal{D} \vdash A \Leftarrow \mathsf{Type} \downarrow \mathcal{C}_1 \qquad \Gamma, x : A \uparrow \mathcal{D} \cup \mathcal{C}_1 \vdash B \Rightarrow C \downarrow \mathcal{C}_2 \qquad C \longrightarrow^{\text{wh}} s}{\Gamma \uparrow \mathcal{D} \vdash (x : A) \to B \Rightarrow s \downarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\textsc{App}\ \frac{\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow C \downarrow \mathcal{C}_1 \qquad C \longrightarrow^{\text{wh}} (x : A) \to B \qquad \Gamma \uparrow \mathcal{D} \cup \mathcal{C}_1 \vdash u \Leftarrow A \downarrow \mathcal{C}_2}{\Gamma \uparrow \mathcal{D} \vdash t\ u \Rightarrow B[x \mapsto u] \downarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

Figure 4: Bidirectional type system for computing universe level constraints

Then, once we know that the type is valid under the constraints $\mathcal{C}_1$, we can do the same with the term:

$$() \uparrow \mathcal{C}_1 \vdash (\mathsf{id}\ i_2)_{@}(\Pi A : \mathsf{U}_{i_3}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4) \Leftarrow \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_1}.A \rightsquigarrow A) \downarrow \mathcal{C}_2$$

In the end we get the constraints

$$\mathcal{C}_1 \cup \mathcal{C}_2 := \{\mathsf{S}\ i_1 \stackrel{?}{=} i_2, i_1 \stackrel{?}{=} i_3, i_1 \stackrel{?}{=} i_4, \ldots\}$$

where the hidden constraints concern level variables appearing in implicit arguments.

We can show the soundness of the new type system by verifying that each rule locally preserves the invariants of the table — this is the essence of the proof of Theorem 5.5. Before showing this, we first need a lemma establishing the soundness of conversion checking.

Lemma 5.4. *Suppose that* $A \equiv B \downarrow \mathcal{C}$ *or* $A \equiv_{\text{whnf}} B \downarrow \mathcal{C}$. *If* $\theta \vDash \mathcal{C}$ *then* $A[\theta] \equiv B[\theta]$.

Proof. By an easy mutual induction on $A \equiv B \downarrow \mathcal{C}$ and $A \equiv_{\text{whnf}} B \downarrow \mathcal{C}$. □

Theorem 5.5 (Soundness of type system for computing constraints)**.** *Consider the rules of Figure 4 where* Φ' *is an arbitrary local signature well-formed in* $\mathbb{UPP}$.

- *If* $\Gamma \vdash_{\mathcal{D}} A : s$ *and* $\Gamma \uparrow \mathcal{D} \vdash t \Leftarrow A \downarrow \mathcal{C}$ *then* $\Gamma \vdash_{\mathcal{C} \cup \mathcal{D}} t : A$
- *If* $\Gamma \vdash_{\mathcal{D}}$ *and* $\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow A \downarrow \mathcal{C}$ *then* $\Gamma \vdash_{\mathcal{C} \cup \mathcal{D}} t : A$

Proof. By mutual induction on the elaborator judgments.

- Case Var : Let $\theta \vDash \varnothing \cup \mathcal{D}$. By hypothesis we have $\Xi_\theta, \Gamma[\theta] \vdash$, and hence $\Xi_\theta, \Gamma[\theta] \vdash x : A[\theta]$.
- Case Sort : Similar to the previous case.
- Case Cons : Let $\theta \vDash \varnothing \cup \mathcal{D}$. By hypothesis we have $\Xi_\theta, \Gamma[\theta] \vdash$, and hence $\Xi_\theta, \Gamma[\theta] \vdash c : (j_1..j_k : \mathsf{Lvl}) \to B$. Because $i_n[\theta]$ is a level with free variables among Ξ_θ, we then also have $\Xi_\theta, \Gamma[\theta] \vdash i_n[\theta] : \mathsf{Lvl}$, for all $n = 1..k$. So we get $\Xi_\theta, \Gamma[\theta] \vdash c\ i_1[\theta]..i_k[\theta] : B[\vec{j} \mapsto \vec{i}[\theta]]$, and because $B[\vec{j} \mapsto \vec{i}[\theta]] = B[\vec{j} \mapsto \vec{i}][\theta]$ we conclude.
- Case Switch : Let $\theta \vDash \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}$. By hypothesis we have $\Gamma \vdash_{\mathcal{D}} B : s$ and thus $\Gamma \vdash_{\mathcal{D}}$, therefore by i.h. we have $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t : A$, from which we get $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : A[\theta]$. Moreover, from Lemma 5.4 we also get $A[\theta] \equiv B[\theta]$. Finally, from $\Gamma \vdash_{\mathcal{D}} B : s$ we have $\Xi_\theta, \Gamma[\theta] \vdash B[\theta] : s$, and therefore we conclude $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : B[\theta]$ by rule Conv.
- Case Pi : Let $\theta \vDash \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}$. By hypothesis we have $\Gamma \vdash_{\mathcal{D}}$ and thus $\Gamma \vdash_{\mathcal{D}} \mathsf{Type} : \mathsf{Kind}$, therefore by i.h. we have $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} A : \mathsf{Type}$, from which we get $\Xi_\theta, \Gamma[\theta] \vdash A[\theta] : \mathsf{Type}$ and $\Gamma, x : A \vdash_{\mathcal{C}_1 \cup \mathcal{D}}$. Therefore, by i.h. once again we also have $\Gamma, x : A \vdash_{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}} B : C$ and thus $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash B[\theta] : C[\theta]$.

  We now claim that $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash B[\theta] : s$. Indeed, by validity we have either $C[\theta] = \mathsf{Kind}$ or $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash C[\theta] : s'$ for some s'. If $C[\theta] = \mathsf{Kind}$ then we must have $s = \mathsf{Kind}$, and thus the claim follows from $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash B[\theta] : C[\theta]$. If $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash C[\theta] : s'$ for some s', then by subject reduction we have $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash s : s'$, in which case we must have $s = \mathsf{Type}$ and $s' = \mathsf{Kind}$. Therefore, by the conversion rule with $C[\theta] \equiv \mathsf{Type}$ we get $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash B[\theta] : \mathsf{Type}$.

  Finally, we conclude $\Xi_\theta, \Gamma[\theta] \vdash (x : A[\theta]) \to B[\theta] : s$ from $\Xi_\theta, \Gamma[\theta] \vdash A[\theta] : \mathsf{Type}$ and $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash B[\theta] : s$.
- Case App : By hypothesis we have $\Gamma \vdash_{\mathcal{D}}$, therefore by i.h. we have $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t : C$.

  We claim that $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t : (x : A) \to B$. Indeed, let $\theta \vDash \mathcal{C}_1 \cup \mathcal{D}$, in which case we have $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : C[\theta]$. By validity we have $\Xi_\theta, \Gamma[\theta] \vdash C[\theta] : s$ for some s, so by subject reduction and $C[\theta] \longrightarrow^* (x : A[\theta]) \to B[\theta]$ we have $\Xi_\theta, \Gamma[\theta] \vdash (x : A[\theta]) \to B[\theta] : s$. Applying conversion with $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : C[\theta]$, we get $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : (x : A[\theta]) \to B[\theta]$.

  Using validity and inversion of typing, we can also derive $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} A : \mathsf{Type}$ from $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t : (x : A) \to B$, and thus we can apply the i.h. once again to get $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}} u : A$.

  Now let $\theta \vDash \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}$. We thus have $\Xi_\theta, \Gamma[\theta] \vdash t[\theta] : (x : A[\theta]) \to B[\theta]$ and $\Xi_\theta, \Gamma[\theta] \vdash u[\theta] : A[\theta]$, so by the application rule we get $\Xi_\theta, \Gamma[\theta] \vdash t[\theta]\ u[\theta] : B[\theta][x \mapsto u[\theta]]$. Because $B[\theta][x \mapsto u[\theta]] = (B[x \mapsto u])[\theta]$, the result follows.
- Case Abs : By hypothesis we have $\Gamma \vdash_{\mathcal{D}} C : s$, from which we derive $\Gamma \vdash_{\mathcal{D}} (x : A) \to B : s$ using subject reduction, and then $\Gamma, x : A \vdash_{\mathcal{D}} B : s$ using inversion.

Now let $\theta \models \mathcal{C} \cup \mathcal{D}$. From $\Gamma \vdash_{\mathcal{D}} (x : A) \to B : s$ we get $\Xi_\theta, \Gamma[\theta] \vdash (x : A[\theta]) \to B[\theta] : s$, thus by inversion we have $\Xi_\theta, \Gamma[\theta] \vdash A[\theta] : \mathsf{Type}$ and $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash B[\theta] : s$. By the i.h. we also get $\Gamma, x : A \vdash_{\mathcal{C}\cup\mathcal{D}} t : B$, and thus $\Xi_\theta, \Gamma[\theta], x : A[\theta] \vdash t[\theta] : B[\theta]$, and therefore we conclude $\Xi_\theta, \Gamma[\theta] \vdash x.t[\theta] : (x : A[\theta]) \to B[\theta]$ by the abstraction rule. □

Remark 5.6. One could also wonder if the computation of constraints always terminates, either with a valid set of constraints or with an error indicating the term cannot be elaborated. By supposing strong normalization for $\mathbb{UPP}$, and by checking at each step of the rules in Figure 4 that the constraints are consistent, one could show termination of the algorithm by using a similar technique as in [HP91]. As we do not investigate strong normalization of $\mathbb{UPP}$ in this paper — and doing so would further deviate us from the goals of this work —, we leave this 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.

5.3. **Solving the constraints.** Once the constraints are computed, the next step is solving 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 thus requires unification, but because levels are not purely syntactic entities, one needs to devise a unification algorithm specific for the equational theory of universe levels. 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 $\mathcal{C}$ a unifier θ.

Once a unifier for the constraints is found, we can just apply it to the entry and generalize over all free level variables $\vec{i}$, and get either $c : (\vec{i} : \mathsf{Lvl}) \to A[\theta]$ or $c : (\vec{i} : \mathsf{Lvl}) \to A[\theta] := \vec{i}.t[\theta]$. However, a last optimization can be made: let us write $\vec{i}_{A[\theta]}$ for the free level variables occurring in $A[\theta]$, and $\vec{i}_{t[\theta]\setminus A[\theta]}$ for the free level variables occurring in $t[\theta]$ but not in $A[\theta]$. Then we can reduce the number of level arguments of the entry by setting all the $\vec{i}_{t[\theta]\setminus A[\theta]}$ to 0, without changing the type $A[\theta]$ in the entry: we then get $c : (\vec{i}_{A[\theta]} : \mathsf{Lvl}) \to A[\theta]$ or $c : (\vec{i}_{A[\theta]} : \mathsf{Lvl}) \to A[\theta] := \vec{i}_{A[\theta]}.t[\theta][\vec{i}_{t[\theta]\setminus A[\theta]} \mapsto 0]$. While this does not impact the correctness of the elaboration, this optimization is still useful because reducing the number of level arguments empirically leads to unification problems that are easier to solve in practice.

Example 5.7. Recall that when calculating the constraints of entry id' we found

$$\mathcal{C}_1 \cup \mathcal{C}_2 := \{\mathsf{S}\ i_1 \stackrel{?}{=} i_2, i_1 \stackrel{?}{=} i_3, i_1 \stackrel{?}{=} i_4, \ldots\}$$

The algorithm of the next section is able to compute the unifier

$$\theta = i_1 \mapsto i_4,\ i_2 \mapsto \mathsf{S}\ i_4,\ i_3 \mapsto i_4, \ldots$$

We can now apply the unifier to the entry and generalize over the free level variables of the type, while mapping the other ones to 0, which gives at the end

$$\mathsf{id}' : (i_4 : \mathsf{Lvl}) \to \mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_4}.A \rightsquigarrow A)$$
$$\mathsf{id}' := i_4.(\mathsf{id}\ (\mathsf{S}\ i_4))_{@}(\Pi A : \mathsf{U}_{i_4}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)$$

Note that in the resulting term, the two occurrences of id are applied to different universe levels. This illustrates the importance of the use of universe polymorphism in the translation.

Let us now show the final correctness theorem for elaboration.

Theorem 5.8 (Correctness of elaboration)**.** *Let Φ' be a local signature well-formed in $\mathbb{UPP}$, and A, t schematic terms.*

- *Suppose* $() \uparrow \varnothing \vdash A \Leftarrow \mathsf{Type} \downarrow \mathcal{C}_1$ *and* $() \uparrow \mathcal{C}_1 \vdash t \Leftarrow A \downarrow \mathcal{C}_2$ *and* $\theta = \textsc{Unify}(\mathcal{C}_1 \cup \mathcal{C}_2)$*. Then* $\Phi', c : (\vec{i}_{A[\theta]} : \mathsf{Lvl}) \to A[\theta] := \vec{i}_{A[\theta]}.t[\theta][\vec{i}_{t[\theta]\setminus A[\theta]} \mapsto 0]$ *is well-formed in* $\mathbb{UPP}$*.*
- *Suppose* $() \uparrow \varnothing \vdash A \Leftarrow \mathsf{Type} \downarrow \mathcal{C}_1$ *and* $\theta = \textsc{Unify}(\mathcal{C}_1)$*. Then* $\Phi', c : (\vec{i}_{A[\theta]} : \mathsf{Lvl}) \to A[\theta]$ *is well-formed in* $\mathbb{UPP}$*.*

Proof. We show the first point, the proof of the second one being similar. By Theorem 5.5, we have $() \vdash_{\mathcal{C}_1 \cup \mathcal{C}_2} t : A$, so because $\theta \vDash \mathcal{C}_1 \cup \mathcal{C}_2$ we get $\Xi_\theta \vdash t[\theta] : A[\theta]$. Consider a substitution mapping all level variables in Ξ_θ not in $A[\theta]$ to 0. By the substitution property, we get $\vec{i}_{A[\theta]} : \mathsf{Lvl} \vdash t[\theta][\vec{i}_{t[\theta]\setminus A[\theta]} \mapsto 0] : A[\theta]$, and then by abstracting $\vec{i}_{A[\theta]}$ we get

$$() \vdash \vec{i}_{A[\theta]}.t[\theta][\vec{i}_{t[\theta]\setminus A[\theta]} \mapsto 0] : (\vec{i}_{A[\theta]} : \mathsf{Lvl}) \to A[\theta]$$

Therefore, $\Phi', c : (\vec{i}_{A[\theta]} : \mathsf{Lvl}) \to A[\theta] := \vec{i}_{A[\theta]}.t[\theta][\vec{i}_{t[\theta]\setminus A[\theta]} \mapsto 0]$ is well-formed in $\mathbb{UPP}$. □

6. Solving universe level unification problems

Our elaborator relies on an unspecified algorithm for universe level unification, which we now present. Before going any further, let us recall that a unifier θ for a problem $\mathcal{C}$ is said to be a *most general unifier* (abbreviated as m.g.u.) when, for any other unifier τ of $\mathcal{C}$, there is a substitution θ' such that $i[\theta][\theta'] \simeq i[\tau]$ for all i appearing in $\mathcal{C}$. When studying unification in an equational theory, the first natural question that comes to mind is whether all solvable unification problems have a most general unifier. Our first important observation is that, in the case of the equational theory of levels used in $\mathbb{UPP}$, this property does not hold.[7]

Theorem 6.1. *Not all solvable problems of universe level unification have a most general unifier.*

Proof. Consider the equation $\mathsf{S}\ i_1 = i_2 \sqcup i_3$, which is solvable, and suppose it had a m.g.u. θ. Note that $\theta_1 = i_1 \mapsto 0,\ i_2 \mapsto \mathsf{S}\ 0,\ i_3 \mapsto 0$ is also a unifier, thus for some τ we have $i_3[\theta][\tau] \simeq 0$. Therefore, there can be no occurrence of S in $i_3[\theta]$. By taking $\theta_2 = i_1 \mapsto 0, i_2 \mapsto 0, i_3 \mapsto \mathsf{S}\ 0$ we can show similarly that there can be no occurrence of S in $i_2[\theta]$. But by taking the substitution $\theta' = _ \mapsto 0$ mapping all variables to 0, we get $(i_2 \sqcup i_3)[\theta][\theta'] \simeq 0$, which cannot be equivalent to $(\mathsf{S}\ i_1)[\theta][\theta']$. Hence, $\mathsf{S}\ i_1 = i_2 \sqcup i_3$ has no m.g.u. □

Therefore, one cannot expect to be able to compute a m.g.u. for all solvable problems of universe level unification. One can then also wonder if, by restricting to the fragment of problems generated by the elaborator, one can expect to recover the property that all solvable problems admit a m.g.u. The following result also answers this negatively.

Theorem 6.2. *There is a schematic term whose constraints computed by the elaborator are solvable but have no most general unifier.*

Proof. Consider the following term (once again, we reuse our convention of keeping some arguments implicit).

$$\lambda A : \mathsf{U}_{i_1}.\lambda B : \mathsf{U}_{i_2}.\lambda R : (\Pi C : \mathsf{U}_{i_3}.C \rightsquigarrow C \rightsquigarrow \mathsf{U}_{i_4}).R_{@}\mathsf{U}_{i_6}{}_{@}\mathsf{U}_{i_7}{}_{@}(A \rightsquigarrow B)$$

[7] It follows that universe level unification is not *unitary* [BS94]. Whether it is *finitary*, *infinitary* or of *type zero* is still to be determined, and left for future work.

If we try to elaborate it in mode infer in the empty context, we get a unification problem that can be simplified to $\{\mathsf{S}\ i_7 \stackrel{?}{=} i_1 \sqcup i_2\}$, after solving some easy equations.[8] This is because the application of $R_{@}\mathsf{U}_{i_6} : \mathsf{Tm}\ (\mathsf{U}_{i_6} \rightsquigarrow \mathsf{U}_{i_6} \rightsquigarrow \mathsf{U}_{i_4})$ to U_{i_7} and $(A \rightsquigarrow B)$ requires the last two to be in the same universe level, which are respectively $\mathsf{S}\ i_7$ and $i_1 \sqcup i_2$. This equation is solvable but, by Theorem 6.1, does not admit a most general unifier. □

Remark 6.3. In the above proof, one can alternatively verify the calculation of constraints automatically in Agda by typechecking the code

$$\mathsf{test} : (A : \mathsf{Set}\ _) \to (B : \mathsf{Set}\ _) \to (R : (C : \mathsf{Set}\ _) \to C \to C \to \mathsf{Set}\ _) \to \mathsf{Set}\ _$$
$$\mathsf{test} = \lambda A\ B\ R \to R\ (\mathsf{Set}\ _)\ (\mathsf{Set}\ _)\ (A \to B)$$

which returns the error

```
Failed to solve the following constraints:  _0 ⊔ _1 = lsuc _10
```

showing that Agda's elaborator also simplifies the problem to find the same constraint.

In other words, some schematic terms may not admit a most general universe-polymorphic instance, even when they admit some well-typed instances. A possible strategy would be to look not for a m.g.u., but instead for a minimal set of incomparable unifiers, as is often done in the equational unification literature [BS94]. However, this would not only require to duplicate each term being translated, one for each incomparable unifier, but this strategy would also risk of growing the output size exponentially. Indeed, a term using a previous translated entry that was duplicated n times would then need to be elaborated multiple times, once with each of these n variants.

Therefore, we instead insist in looking only for m.g.u.s, even if by Theorem 6.2 this approach can fail when the problem is solvable but does not admit a m.g.u. To do this, we proceed as follows in this section.

Our main contribution, given in Subsection 6.2, is a complete characterization of the equations $l \stackrel{?}{=} l'$ that admit a most general unifier. More precisely, our result says exactly when such an equation (1) admits a m.g.u., in which case we also have an explicit description of one, (2) does not admit any unifier, or (3) admits some unifier but no most general one.

Our characterization yields an algorithm for solving equations which is complete, in the sense that we can always find some m.g.u. when the equation admits one. However, because we are interested in unification problems that may contain multiple equations, in Subsection 6.3 we then apply this characterization in the design of a partial algorithm using a *constraint-postponing* strategy [ZS17, DHKP96, Ree09]: at each step, we look for an equation which admits a m.g.u. and eliminate it, while applying the obtained substitution to the other constraints. This can then bring new equations to the fragment admitting a m.g.u., allowing us to solve them next. This is similar to how most proof assistants handle higher-order unification problems, by trying to solve the equations that are in the *pattern* fragment, in the hope of unblocking some other ones in the process.

6.1. **Properties of levels.**

Before presenting our main results, we first start by reviewing some important properties about universe levels that will be useful in our proofs.

Notation 6.4. We adopt new notation conventions to improve the readability of large level expressions. In the following, we write $n + l$ for the level $\mathsf{S}^n\ l$, n for the level $\mathsf{S}^n\ 0$, and we

[8] For instance, by using the algorithm of Figure 6.

drop the blue color in $\sqcup$. For instance, the level $\mathsf{S}\ 0 \sqcup \mathsf{S}\ (\mathsf{S}\ i \sqcup 0)$ will henceforth be written $1 \sqcup 1 + (1 + i \sqcup 0)$ — note that $+$ binds tighter then $\sqcup$, and that the left argument of $+$ is always a natural number, so this expression can be parsed unambiguously.

In order to be able to compare levels syntactically, it is useful to introduce a notion of canonical form. A level is said to be in *canonical form* [Voe14, Gen20] when it is of the form

$$p \sqcup n_1 + i_1 \sqcup \ldots \sqcup n_m + i_m$$

with $n_k \leqslant p$ for all $k = 1..m$, and each variable occurs only once. In this case we call p the *constant coefficient*, and n_k the *coefficient of* i_k. We recall the following fundamental property, which appears in [Voe14, Gen20, Bla22], and which we reprove here for completeness reasons.

Theorem 6.5. *Every level is equivalent to a canonical form, which is unique modulo associativity-commutativity. Moreover, there is a computable function mapping each level to one of its canonical forms.*

Proof. Given a level l, we first replace each variable i by $i \sqcup 0$ (which are convertible levels). Then, by repeatedly applying $1+(l \sqcup l') \simeq 1+l \sqcup 1+l'$, we get a level of the form $l_1 \sqcup \ldots \sqcup l_p$, in which each l_k is either of the form $n_k + i_k$ or n_k. Note that we can easily show $n+i \sqcup i \simeq n+i$ for all $n \in \mathbb{N}$, by induction on n. Using this equation, we can merge all constant coefficients, and then all coefficients of a same variable, by always taking the maximum between them. Because in the beginning we started by replacing each variable i by $i \sqcup 0$, it follows that the constant coefficient of the resulting level must be greater or equal to all variable coefficients, hence it is in canonical form.

To see that the canonical form is unique modulo associativity-commutativity, it suffices to note that if two canonical forms have different coefficients for a variable i, then by applying a substitution mapping i to some n large enough and the other variables to 0 we get two levels which are not convertible, hence the canonical forms we started with could not have been convertible. Similarly, if the constant coefficients are different, it suffices to take the substitution mapping all variables to 0, which then also yields non-convertible levels. □

We hence get the following theorem, also in [Voe14, Gen20, Bla22].

Corollary 6.6. *The equational theory $\simeq$ is decidable for levels.*

In view of Theorem 6.5 and the notion of canonical form, we introduce the following notation: given a level l, we write $l\langle i\rangle$ for the coefficient of i in its canonical form, and set it to $-\infty$ if $i \notin \mathsf{fv}(l)$. We extend this notation to $l\langle 0\rangle$, which denotes the constant coefficient of the canonical form of l. Note that from the definition of canonical forms, we always have $l\langle 0\rangle \neq -\infty$, and $l\langle 0\rangle \geqslant l\langle i\rangle$ for all i, and $l\langle i\rangle \neq -\infty$ only for finitely many i — and moreover, an assignment $\mathcal{I} \cup \{0\} \to \mathbb{N} \cup \{-\infty\}$ defines a valid canonical form exactly when these conditions are met.

In the following, let j_0 stand for either a variable j or the constant 0. Then Theorem 6.5 says exactly that $l \simeq l'$ iff for all j_0 we have $l\langle j_0\rangle = l'\langle j_0\rangle$. This principle will be very useful when proving or disproving that two levels are equivalent.

The definition of $\simeq$ can now be justified by the following property. Given a function ϕ mapping each confined variable to a natural number, define the interpretation $[\![l]\!]_\phi$ of a level l by interpreting the symbols $0, \mathsf{S}$ and $\sqcup$ as zero, successor and max, and by interpreting each variable i by $\phi(i)$.

Proposition 6.7. *We have $l_1 \simeq l_2$ iff $\forall \phi, [\![l_1]\!]_\phi = [\![l_2]\!]_\phi$.*

Proof. Note that for each $l \simeq l' \in \mathcal{E}_{\mathbb{UPP}}$ we have $[\![l]\!]_\phi = [\![l']\!]_\phi$ for all ϕ, and thus the direction $\Rightarrow$ can be showed by an easy induction on $l_1 \simeq l_2$.

For the other direction, let us take the canonical forms l_1' of l_1 and l_2' of l_2. By the left to right implication, we have $[\![l_1]\!]_\phi = [\![l_1']\!]_\phi$ and $[\![l_2]\!]_\phi = [\![l_2']\!]_\phi$ for all ϕ, hence $[\![l_1']\!]_\phi = [\![l_2']\!]_\phi$ for all ϕ. By varying ϕ over suitable valuations we can show that l_1' and l_2' have the same constant coefficients, and that each variable appearing in one also appears in the other with the same coefficient. Therefore, l_1' and l_2' are equal modulo associativity-commutativity, and thus $l_1 \simeq l_2$. □

In other words, $\simeq$ allows one to simplify level expressions which are semantically the same — for instance, $1 + i \sqcup i \sqcup 0$ and $1 + i$. This also shows that our definition of $\simeq$, which is also used in [Tea], agrees with the one used in other works about universe levels [Gen20, Voe14, Fer21, Bla22].

6.2. **Characterizing equations that admit a m.g.u.** With the preliminaries now set up, we can move to the main contribution of this section: a characterization of the equations that admit a most general unifier, along with an explicit description of a m.g.u. in these cases. Our first step is to introduce a notion of canonical form for equations.

Definition 6.8. An equation $l_1 \stackrel{?}{=} l_2$ is said to be in canonical form if

(1) Both l_1, l_2 are in canonical form.
(2) If $i \in \mathsf{fv}(l_1) \cap \mathsf{fv}(l_2)$, then $l_1\langle i\rangle = l_2\langle i\rangle$
(3) At least some coefficient in l_1 or l_2 is equal to 0

The main motivation for introducing this notion is the following result, stating that in our analysis it suffices to consider only equations in canonical form.

Proposition 6.9. *For all equations $l_1 \stackrel{?}{=} l_2$, there is an equation $l_1' \stackrel{?}{=} l_2'$ in canonical form, such that for all θ, $l_1[\theta] \simeq l_2[\theta]$ iff $l_1'[\theta] \simeq l_2'[\theta]$.*

Proof. Let $l_1 \stackrel{?}{=} l_2$ be any equation. We apply transformations so that properties (1)-(3) that define canonical forms are satisfied one by one, and we argue that they do not change the set of unifiers.

(1) We put each level l_p in canonical form l_p'. It is clear that this preserves the set of unifiers, as any level is convertible to its canonical form.
(2) If some variable i appears in l_1' and l_2' with different coefficients, we remove it from the side with smaller coefficient, and we name the resulting equation $l_1'' \stackrel{?}{=} l_2''$ — this step is then repeated until condition (2) of the canonical form definition is met. By decomposing $l_1' \simeq l_a \sqcup n + i$ and $l_2' \simeq l_b \sqcup m + i$ with $n < m$ (or the symmetric), the correctness of this step follows from $l_1'[\theta] \simeq l_2'[\theta]$ iff $l_a[\theta] \sqcup n + i[\theta] \simeq l_b[\theta] \sqcup m + i[\theta]$ iff $l_a[\theta] \simeq l_b[\theta] \sqcup m + i[\theta]$, where the last equivalence follows from the fact that
$$\max\{k_a, n+q\} = \max\{k_b, m+q\} \quad \Longleftrightarrow \quad k_a = \max\{k_b, m+q\}$$
for all $k_a, k_b, n, m, q \in \mathbb{N}$ with $n < m$, and then by Proposition 6.7.
(3) Finally, if no coefficient in l_1'' or l_2'' is equal to zero, we subtract from all coefficients the value of the current minimal coefficient, and we name the resulting equation $l_1''' \stackrel{?}{=} l_2'''$. If we call this value k, then the correctness of this step follows from the fact that $l_p'' \simeq k + l_p'''$ for $p = 1, 2$, and so $l_1''[\theta] \simeq l_2''[\theta]$ iff $k + l_1'''[\theta] \simeq k + l_2'''[\theta]$ iff $l_1'''[\theta] \simeq l_2'''[\theta]$, where the last equivalence follows by applying Proposition 6.7.

It is clear that $l_1''' \stackrel{?}{=} l_2'''$ is in canonical form, and we have shown that each step of the transformation preserves the set of unifiers. □

Example 6.10. Consider the equation $i \sqcup 1 + (i \sqcup 1 + j) \stackrel{?}{=} j \sqcup 2 + i$ and let us show how it can be put in canonical form using the underlying algorithm of the above proof. First, we compute the level canonical forms of each side, yielding

$$2 \sqcup 1 + i \sqcup 2 + j \stackrel{?}{=} 2 \sqcup 2 + i \sqcup j$$

As the variables i and j appear in the two sides with different coefficients, we then remove from each of the sides the occurrence with the smaller coefficient, yielding

$$2 \sqcup 2 + j \stackrel{?}{=} 2 \sqcup 2 + i$$

Finally, as the minimum among all coefficients is 2, we subtract this from all of them, giving

$$0 \sqcup j \stackrel{?}{=} 0 \sqcup i$$

We are now able to state the main theorem that we are going to show. In the following, if $k \in \mathbb{N}$ we write $[k]$ for the set $\{1, ..., k\}$, and we call an equation $l_1 \stackrel{?}{=} l_2$ *trivial* when $l_1 \simeq l_2$. We also call $l_2 \stackrel{?}{=} l_1$ the *symmetric* of the equation $l_1 \stackrel{?}{=} l_2$. Finally, if $\vec{i} = i_1 ... i_k$ is a list of level variables, we sometimes identify it with the level $i_1 \sqcup ... \sqcup i_k$.

Theorem 6.11. *A non-trivial equation has*

(A) *a most general unifier iff its canonical form (or its symmetric) is of the form*
  (i) $n \sqcup i \stackrel{?}{=} l$ *with* $n < l\langle 0 \rangle$*, in which case* $\theta = i \mapsto l$ *is a m.g.u.*
  (ii) $0 \sqcup \vec{i_0} \sqcup \vec{i_1} \stackrel{?}{=} 0 \sqcup \vec{i_0} \sqcup \vec{i_2}$ *with* $\vec{i_0}$*,* $\vec{i_1}$ *and* $\vec{i_2}$ *disjoint, in which case a m.g.u. is given by*

$$\begin{aligned} \theta = \quad & i_0^k \mapsto x_k \sqcup (\sqcup_{n \in [p_1]} y_{k,n}) \sqcup (\sqcup_{m \in [p_2]} z_{k,m}) && (k \in [p_0]) \\ & i_1^n \mapsto (\sqcup_{k \in [p_0]} y_{k,n}) \sqcup (\sqcup_{m \in [p_2]} v_{n,m}) && (n \in [p_1]) \\ & i_2^m \mapsto (\sqcup_{k \in [p_0]} z_{k,m}) \sqcup (\sqcup_{n \in [p_1]} v_{n,m}) && (m \in [p_2]) \end{aligned}$$

  where p_0, p_1, p_2 *are the lengths of* $\vec{i_0}$ *and* $\vec{i_1}$ *and* $\vec{i_2}$ *respectively, and where* $\{x_k\}_{k \in [p_0]}$*,* $\{y_{k,n}\}_{k \in [p_0], n \in [p_1]}$*,* $\{z_{k,m}\}_{k \in [p_0], m \in [p_2]}$ *and* $\{v_{n,m}\}_{n \in [p_1], m \in [p_2]}$ *are disjoint sets of variables.*

(B) *no unifier iff its canonical form (or its symmetric) is of the form* $n \stackrel{?}{=} l$ *with* $n < l\langle 0 \rangle$

(C) *some unifier but no most general one iff its canonical form (or its symmetric) is not of any of the previous forms*

Before proving the result, let us consider some examples to see how it can be used.

Example 6.12.

- The equation $i_0 \sqcup i_1 \stackrel{?}{=} i_0 \sqcup i_2$ has the canonical form

$$0 \sqcup i_0 \sqcup i_1 \stackrel{?}{=} 0 \sqcup i_0 \sqcup i_2$$

  and therefore by point (A.ii) it admits the m.g.u.

$$\theta = \{i_0 \mapsto x \sqcup y \sqcup z,\ i_1 \mapsto y \sqcup v,\ i_2 \mapsto z \sqcup v\}$$

- The equation $i \sqcup 1 + (j \sqcup 2) \stackrel{?}{=} 1 + (2 \sqcup i \sqcup j)$ has the canonical form

$$2 \sqcup j \stackrel{?}{=} 2 \sqcup i \sqcup j$$

  and therefore by point (C) it is solvable but admits no m.g.u.

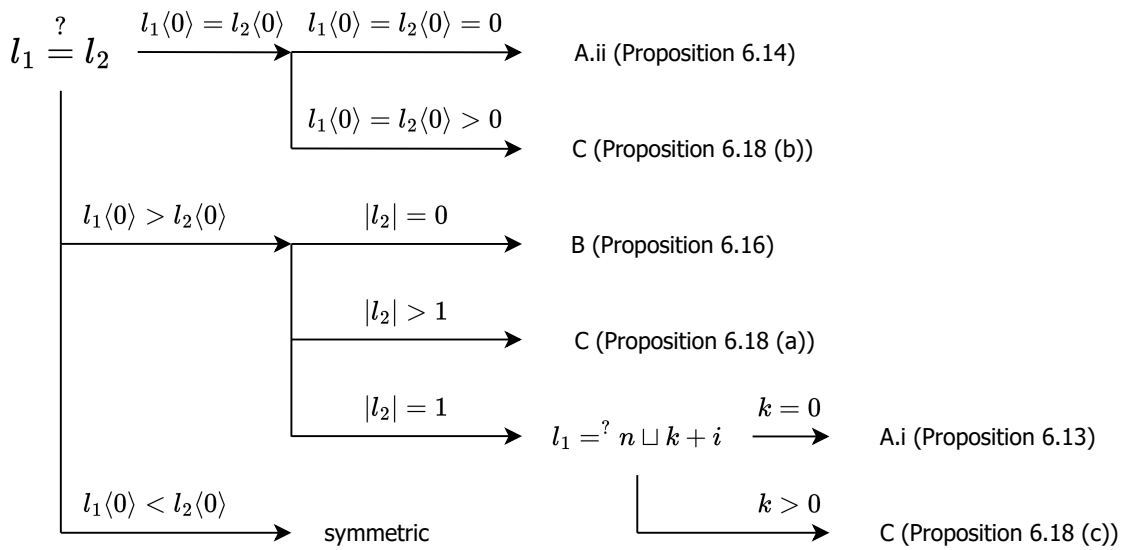

Figure 5: Structure of proof of Theorem 6.11

- The equation $i \sqcup 1 + (j \sqcup 1) \stackrel{?}{=} 1 + (2 \sqcup i \sqcup j)$ has the canonical form
$$1 \sqcup j \stackrel{?}{=} 2 \sqcup i \sqcup j$$
and therefore by point (A.i) it admits the m.g.u.
$$\theta = j \mapsto 2 \sqcup i \sqcup j$$
- The equation $i \sqcup 1 + (j \sqcup 1) \stackrel{?}{=} 2 + (1 \sqcup i \sqcup j)$ has the canonical form
$$0 \stackrel{?}{=} 1 \sqcup i \sqcup j$$
and therefore by point (B) it admits no unifier.

Let us now move to the proof of Theorem 6.11. Figure 5 shows its structure: we take a non-trivial equation in canonical form and consider its possible forms. Each leaf is annotated with the proposition associated with its proof, along with the case of Theorem 6.11 which we are in. We also write $|l_2|$ for the number of free variables occurring in l_2.

6.2.1. *Equations with m.g.u.s.*

Proposition 6.13. *The equation in canonical form $n \sqcup i \stackrel{?}{=} l$ with $n < l\langle 0\rangle$ has the mgu $\tau = i \mapsto l$.*

Proof. It is easy to verify that τ is a unifier. Now let θ be an arbitrary unifier and let us first show that $i[\theta] \simeq l[\theta]$. To do this we show $i[\theta]\langle j_0\rangle = l[\theta]\langle j_0\rangle$ for all j_0. Because the canonical forms of $i[\theta]$ and $n \sqcup i[\theta]$ can only differ on their constant coefficients, and because $n \sqcup i[\theta] \simeq l[\theta]$, then it follows that $i[\theta]\langle j\rangle = l[\theta]\langle j\rangle$ for all variables j. For the case $j_0 = 0$, we have $\max\{n, i[\theta]\langle 0\rangle\} = l[\theta]\langle 0\rangle$, and because $l\langle 0\rangle > n$ then $l[\theta]\langle 0\rangle > n$, so the only possibility is $i[\theta]\langle 0\rangle = l[\theta]\langle 0\rangle$.

Now we can show that τ is more general than θ: we have $j[\tau][\theta] \simeq j[\theta]$ for all $j \in \mathsf{fv}(l) \cup \{i\}$. Indeed, the equation holds trivially for $i \neq j$, and for $i = j$ it follows from $l[\theta] \simeq i[\theta]$. $\square$

Proposition 6.14. *The equation* $0 \sqcup \vec{i_0} \sqcup \vec{i_1} \stackrel{?}{=} 0 \sqcup \vec{i_0} \sqcup \vec{i_2}$*, with* $\vec{i_0}$*,* $\vec{i_1}$ *and* $\vec{i_2}$ *disjoint, has the m.g.u.*

$$\begin{aligned}
\theta = \quad & i_0^k \mapsto x_k \sqcup (\sqcup_{n\in[p_1]} y_{k,n}) \sqcup (\sqcup_{m\in[p_2]} z_{k,m}) && (k \in [p_0]) \\
& i_1^n \mapsto (\sqcup_{k\in[p_0]} y_{k,n}) \sqcup (\sqcup_{m\in[p_2]} v_{n,m}) && (n \in [p_1]) \\
& i_2^m \mapsto (\sqcup_{k\in[p_0]} z_{k,m}) \sqcup (\sqcup_{n\in[p_1]} v_{n,m}) && (m \in [p_2])
\end{aligned}$$

where p_0, p_1, p_2 *are the lengths of* $\vec{i_0}$ *and* $\vec{i_1}$ *and* $\vec{i_2}$ *respectively, and* $\{x_k\}_{k\in[p_0]}$*,* $\{y_{k,n}\}_{k\in[p_0],n\in[p_1]}$*,* $\{z_{k,m}\}_{k\in[p_0],m\in[p_2]}$ *and* $\{v_{n,m}\}_{n\in[p_1],m\in[p_2]}$ *are disjoint sets of variables.*

Proof. It is easy to see that θ is a unifier: all introduced variables appear in both sides, with coefficient 0. Given a unifier τ, define τ' by setting for each j_0

$$\begin{aligned}
x_k[\tau']\langle j_0\rangle &:= i_0^k[\tau]\langle j_0\rangle \\
y_{k,n}[\tau']\langle j_0\rangle &:= \min\{i_0^k[\tau]\langle j_0\rangle, i_1^n[\tau]\langle j_0\rangle\} \\
z_{k,m}[\tau']\langle j_0\rangle &:= \min\{i_0^k[\tau]\langle j_0\rangle, i_2^m[\tau]\langle j_0\rangle\} \\
v_{n,m}[\tau']\langle j_0\rangle &:= \min\{i_1^n[\tau]\langle j_0\rangle, i_2^m[\tau]\langle j_0\rangle\}
\end{aligned}$$

Note that this assignment indeed defines for each i' a canonical form $i'[\tau']$: the constant coefficient of $i'[\tau']$ is never equal to $-\infty$, it is always greater or equal than the variable coefficients of $i'[\tau']$, and $i'[\tau']\langle j\rangle$ is different from $-\infty$ only for finitely many j. Let us now show that $i'[\tau] \simeq i'[\theta][\tau']$ for all i' among $\vec{i_0}, \vec{i_1}, \vec{i_2}$.

As $i_0^k[\tau]\langle j_0\rangle$ is greater or equal than $\min\{i_0^k[\tau]\langle j_0\rangle, i_1^n[\tau]\langle j_0\rangle\}$ and $\min\{i_0^k[\tau]\langle j_0\rangle, i_2^m[\tau]\langle j_0\rangle\}$ for all n, m, we have

$$\begin{aligned}
i_0^k[\theta][\tau']\langle j_0\rangle &= \max(\{i_0^k[\tau]\langle j_0\rangle\} \\
&\qquad \cup \{\min\{i_0^k[\tau]\langle j_0\rangle, i_1^n[\tau]\langle j_0\rangle\} \mid n \in [p_1]\} \\
&\qquad \cup \{\min\{i_0^k[\tau]\langle j_0\rangle, i_2^m[\tau]\langle j_0\rangle\} \mid m \in [p_2]\}) \\
&= i_0^k[\tau]\langle j_0\rangle
\end{aligned}$$

for all j_0 and $k \in [p_0]$. Therefore, we get $i_0^k[\theta][\tau'] \simeq i_0^k[\tau]$ for all $k \in [p_0]$.

Because τ is a unifier, we have $0 \sqcup \vec{i_0}[\tau] \sqcup \vec{i_1}[\tau] \simeq 0 \sqcup \vec{i_0}[\tau] \sqcup \vec{i_2}[\tau]$, from which we get

$$\max\{\vec{i_0}[\tau]\langle j_0\rangle, \vec{i_1}[\tau]\langle j_0\rangle\} = \max\{\vec{i_0}[\tau]\langle j_0\rangle, \vec{i_2}[\tau]\langle j_0\rangle\}$$

for all j_0. Therefore, for every $n \in [p_1]$, there is some $k \in [p_0]$ st $i_0^k[\tau]\langle j_0\rangle \geqslant i_1^n[\tau]\langle j_0\rangle$, or there is some $m \in [p_2]$ st $i_2^m[\tau]\langle j_0\rangle \geqslant i_1^n[\tau]\langle j_0\rangle$. Hence, either we have $\min\{i_0^k[\tau]\langle j_0\rangle, i_1^n[\tau]\langle j_0\rangle\} = i_1^n[\tau]\langle j_0\rangle$ for some $k \in [p_1]$, or we have $\min\{i_1^n[\tau]\langle j_0\rangle, i_2^m[\tau]\langle j_0\rangle\} = i_1^n[\tau]\langle j_0\rangle$ for some $m \in [p_2]$. Therefore, we get

$$\begin{aligned}
i_1^n[\theta][\tau']\langle j_0\rangle &= \max(\{\min\{i_0^k[\tau]\langle j_0\rangle, i_1^n[\tau]\langle j_0\rangle\} \mid k \in [p_0]\} \\
&\qquad \cup \{\min\{i_1^n[\tau]\langle j_0\rangle, i_2^m[\tau]\langle j_0\rangle\} \mid m \in [p_2]\}) \\
&= i_1^n[\tau]\langle j_0\rangle
\end{aligned}$$

for all j_0 and $n \in [p_1]$. Therefore, we get $i_1^n[\theta][\tau'] \simeq i_1^n[\tau]$ for all $n \in [p_1]$.

Finally, a symmetrical reasoning shows $i_2^m[\theta][\tau'] \simeq i_2^m[\tau]$ for all $m \in [p_2]$. $\square$

Remark 6.15. Proposition 6.14 shows that, when there is no occurrence of S in the equation, it can be solved as an ACUI unification problem. Indeed, the m.g.u. given there is also a m.g.u. of the equation when seen as a unification problem in the theory ACUI [BB88].

6.2.2. *Unsolvable equations.*

Proposition 6.16. *A non-trivial equation in canonical form has no solution iff it (or its symmetric) is of the form $m \stackrel{?}{=} l$ with $m < l\langle 0\rangle$.*

Proof. It is clear that $m \stackrel{?}{=} l$ with $m < l\langle 0\rangle$ has no solution. For the other direction, we show that any equation not of this form has a solution.

First note that if $l_1 \stackrel{?}{=} l_2$ has variables in both sides then it is easy to build a solution. Indeed, if $i_1 \in \mathsf{fv}(l_1)$, $i_2 \in \mathsf{fv}(l_2)$ are (not necessarily distinct) variables, then $\theta = i_1 \mapsto p - l_1\langle i_1\rangle, i_2 \mapsto p - l_2\langle i_2\rangle, _ \mapsto 0$, where $p = \mathsf{max}\{l_1\langle 0\rangle, l_2\langle 0\rangle\}$, is a solution — note that this is also well-defined in the case $i_1 = i_2$, because for equations in canonical form this implies $l_1\langle i_1\rangle = l_2\langle i_2\rangle$.

We can thus restrict our analysis to equations with one of the sides constant, of the form $m \stackrel{?}{=} l$. Note that we can suppose that l has some variable: indeed, if l is constant and $l\langle 0\rangle = m$ then the equation is trivial, and if $l\langle 0\rangle < m$ then its symmetric is of the form $m' = l'$ with $m' < l'\langle 0\rangle$, and indeed has no solution. Finally, it is easy to see that for $m \stackrel{?}{=} l$ with $m \geqslant l\langle 0\rangle$ and where l has some variable i, we have the unifier $\theta = i \mapsto m - l\langle i\rangle, _ \mapsto 0$. □

6.2.3. *Solvable equations not admitting a m.g.u.* The last ingredient for our proof is showing that in all other cases there is no m.g.u. In order to show this, we will use the following auxiliary lemma. In the following, we refer to a level not containing any occurrence of S as *flat*.

Lemma 6.17 (Auxiliary lemma)**.** *Let $l_1 \stackrel{?}{=} l_2$ be an equation admitting a unifier θ and a m.g.u. τ.*

(1) *If $i[\theta] = 0$ with $i \in \mathsf{fv}(l_1) \cup \mathsf{fv}(l_2)$ then $i[\tau]$ is flat.*
(2) *If $j[\theta] = m > 0$ with $j \in \mathsf{fv}(l_1) \cup \mathsf{fv}(l_2)$, and for $p = 1..k$ we have $i_p[\theta] = n_p < m$ with $i_p \in \mathsf{fv}(l_1) \cup \mathsf{fv}(l_2)$, then if $j[\tau]$ is flat it must contain one variable not in any $i_p[\tau]$.*
(3) *If $i \in \mathsf{fv}(l_1)$ and $j \in \mathsf{fv}(i[\theta])$, then for some $i' \in \mathsf{fv}(l_2)$ we must have $j \in \mathsf{fv}(i'[\theta])$.*

Proof. We show each point separately.

(1) Because τ is a m.g.u., for some θ' we have $i[\tau][\theta'] \simeq i[\theta] = 0$, so if $i[\tau]$ contains an occurrence of S then $i[\tau][\theta']$ will also contain one, and therefore will not be convertible to 0.
(2) Because τ is a m.g.u., for some θ' we have $j[\tau][\theta'] \simeq j[\theta] = m$ and $i_p[\tau][\theta'] \simeq i_p[\theta] = n_p$ for $p = 1..k$. Now if we suppose that $j[\tau]$ is flat, then the only way to have $j[\tau][\theta'] \simeq m > 0$ is if some variable i' in $j[\tau]$ is mapped to m by θ'. But because $i_p[\tau][\theta'] \simeq n_p < m$, it is clear that i' cannot appear in any of the $i_p[\tau]$.
(3) Follows from the fact that, if θ is a unifier, then the variables that appear in $l_1[\theta]$ must also appear in $l_2[\theta]$. □

Proposition 6.18 (Equations with no mgu)**.** *The following non-trivial equations in canonical form do not admit a m.g.u.:*

(a) $l_1 \stackrel{?}{=} l_2$ *with* $|l_2| > 1$ *and* $l_1\langle 0\rangle > l_2\langle 0\rangle$.
(b) $l_1 \stackrel{?}{=} l_2$ *with* $l_1\langle 0\rangle = l_2\langle 0\rangle > 0$.
(c) $l \stackrel{?}{=} n \sqcup k + i$ *with* $k > 0$ *and* $n < l\langle 0\rangle$.

Proof. The structure of the proof is the same in all cases: we suppose the existence of a most general unifier τ which we use to obtain a contradiction.

(a) Let i, j be two different variables in l_2. By Lemma 6.17 (1), the unifiers $\theta_1 = i \mapsto l_1\langle 0\rangle - l_2\langle i\rangle, _ \mapsto 0$ and $\theta_2 = j \mapsto l_1\langle 0\rangle - l_2\langle j\rangle, _ \mapsto 0$ show that $i'[\tau]$ is flat for all i'. But then we have $l_1[\tau][_ \mapsto 0] \simeq l_1\langle 0\rangle$ and $l_2[\tau][_ \mapsto 0] \simeq l_2\langle 0\rangle$, and because τ is a unifier we must then have $l_1\langle 0\rangle = l_2\langle 0\rangle$, a contradiction with $l_1\langle 0\rangle > l_2\langle 0\rangle$.
(b) First note that $_ \mapsto 0$ is a unifier, so by Lemma 6.17 (1), $i'[\tau]$ is flat for all i'. Because the equation is supposed to be in canonical form and non-trivial, some variable i appears in only one side. Take such a i with a minimal coefficient, which we henceforth call p.

If $p < l_1\langle 0\rangle$, then the unifier $\theta_1 = i \mapsto l_1\langle 0\rangle - p, _ \mapsto 0$ shows, by Lemma 6.17 (2), that $i[\tau]$ contains a variable not in any $i'[\tau]$ with $i' \neq i$, a contradiction with Lemma 6.17 (3), as i appears in only one side.

Suppose now that $p = l_1\langle 0\rangle$. Because the equation is in canonical form and the constant coefficient of each side is different from 0, then some variable j must appear with coefficient 0. Moreover, because the minimal coefficient of a variable occurring in only one side is $p \neq 0$, it follows that j must appear in both sides (both occurrences, of course, with coefficient 0). Because $p = l_1\langle 0\rangle > 0$, by Lemma 6.17 (2) the unifier $\theta_2 = i \mapsto 1, j \mapsto p + 1, _ \mapsto 0$ shows that some variable $i' \in \mathsf{fv}(i[\tau])$ does not appear in any $j'[\tau]$ with j' different from j and i. Therefore, because i' can only also occur in $j[\tau]$, and because the coefficient of i is p and the coefficient of j is 0, by composing τ with $i' \mapsto 1, _ \mapsto 0$ we get $p + 1$ at the side in which i occurs but p at the other side, a contradiction.
(c) Because we suppose the equation is in canonical form, some j different from i must occur in l with coefficient 0. By Lemma 6.17 (1), the unifier $\theta_1 = i \mapsto l\langle 0\rangle - k, _ \mapsto 0$ shows that $j[\tau]$ is flat, and by Lemma 6.17 (2) the unifier $\theta_2 = i \mapsto l\langle 0\rangle - k, j \mapsto l\langle 0\rangle, _ \mapsto 0$ shows that some variable in $j[\tau]$ does not occur in $i[\tau]$, given that $l\langle 0\rangle - k < l\langle 0\rangle$. Because i is the only variable that appears in the right side, this establishes a contradiction with Lemma 6.17 (3). □

6.2.4. *Putting everything together.*

Proof of Theorem 6.11. We proceed as illustrated in Figure 5. The case $l_1\langle 0\rangle = l_2\langle 0\rangle$ is covered by Proposition 6.14 when $l_1\langle 0\rangle = l_2\langle 0\rangle = 0$, and by Proposition 6.18 (b) when $l_1\langle 0\rangle = l_2\langle 0\rangle \neq 0$. In the case $l_1\langle 0\rangle \neq l_2\langle 0\rangle$ we suppose w.l.o.g. that $l_1\langle 0\rangle > l_2\langle 0\rangle$, the other case being symmetric. Then we branch on the number of variables occurring in l_2: the case of no variables is covered by Proposition 6.16, and the case of more than one variable is covered by Proposition 6.18 (a). For the case of exactly one variable, we branch on the coefficient of this only variable. If the coefficient is zero, the result follows from Proposition 6.13, otherwise it follows by Proposition 6.18 (c). □

6.3. **The unification algorithm.** We can now apply Theorem 6.11 in the design of a partial algorithm for universe level unification. The *configurations* of our algorithm are either of the form $\bot$, or $\mathcal{C};\theta$ where θ is idempotent and $\mathsf{dom}(\theta)$ and $\mathsf{fv}(\mathcal{C})$ are disjoint. Configurations are then rewritten according to the rules of Figure 6, where $\mathcal{C}[\sigma] := \{l_1[\sigma] \stackrel{?}{=} l_2[\sigma] \mid l_1 \stackrel{?}{=} l_2 \in \mathcal{C}\}$ and $\theta[\sigma] := \{i \mapsto i[\theta][\sigma] \mid i \in \mathsf{dom}(\theta)\}$. We also define the free variables of a configuration $\mathcal{C};\theta$ by $\mathsf{fv}(\mathcal{C};\theta) := \mathsf{fv}(\mathcal{C}) \cup \mathsf{dom}(\theta) \cup \mathsf{vrange}(\theta)$, where $\mathsf{vrange}(\theta) := \cup_{i\in\mathsf{dom}(\theta)}\mathsf{fv}(i[\theta])$.

(Solve)	$\{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C};\theta \rightsquigarrow \mathcal{C}[\sigma];\sigma \cup \theta[\sigma]$	if $\sigma = \mathsf{mgu}(l_1, l_2)$
(Fail)	$\{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C};\theta \rightsquigarrow \bot$	if l_1 and l_2 are not unifiable

Figure 6: Unification algorithm for universe levels

Theorem 6.11 is used in (Solve) to detect if some equation admits a m.g.u., and in (Fail) to detect if some equation is unsolvable. We assume that for each σ chosen in step (Solve) we have $\mathsf{dom}(\sigma) = \mathsf{fv}(l_1, l_2)$, which guarantees that $\mathsf{fv}(\mathcal{C};\theta) \subseteq \mathsf{fv}(\mathcal{C}';\theta')$ whenever $\mathcal{C};\theta \rightsquigarrow \mathcal{C}';\theta'$[9]. This assumption can always be satisfied by removing useless entries $i \mapsto l$ in σ for which $i \notin \mathsf{fv}(l_1, l_2)$, and adding trivial entries $i \mapsto i'$ for $i \in \mathsf{fv}(l_1, l_2)\backslash\mathsf{dom}(\sigma)$ and i' fresh, and the resulting substitution is still a m.g.u. We also suppose that the set $\mathsf{vrange}(\sigma)$ only contains fresh variables, which guarantees that steps preserve idempotency of θ and disjointness of $\mathsf{dom}(\theta)$ and $\mathsf{fv}(\mathcal{C})$. This assumption can always be satisfied by composing σ with a bijective renaming, as the composition of a m.g.u. with a bijective renaming is also a m.g.u.

The algorithm succeeds if it reaches a configuration of the form $\varnothing;\theta$, it fails if it reaches the configuration $\bot$ and it gets stuck if it reaches any other configuration in which no rule applies. Moreover, the following straightforward result guarantees that the algorithm cannot run forever, so these are the only options.

Proposition 6.19. *The algorithm always terminates.*

Proof. Each step of (Solve) decreases the cardinality of $\mathcal{C}$, and a step (Fail) leads to a final state. □

In the following, we write $l_1 \stackrel{?}{=} l_2 \in \mathcal{C};\theta$ when either $l_1 \stackrel{?}{=} l_2 \in \mathcal{C}$ or $l_1 = i$ and $l_2 = l$ for some $i \mapsto l \in \theta$. We then write $\tau \vDash \mathcal{C};\theta$ when $l_1[\tau] \simeq l_2[\tau]$ for every $l_1 \stackrel{?}{=} l_2 \in \mathcal{C};\theta$. Finally, given substitutions τ, τ' and a set of variables X, we write $\tau =_X \tau'$ if $i[\tau] = i[\tau']$ for all $i \in X$.

Lemma 6.20 (Key lemma)**.** *Suppose $\mathcal{C}_1;\theta_1 \rightsquigarrow \mathcal{C}_2;\theta_2$. Then*

(1) $\tau \vDash \mathcal{C}_1;\theta_1$ *and* $\mathsf{dom}(\tau) \subseteq \mathsf{fv}(\mathcal{C}_1;\theta_1)$ *imply* $\tau' \vDash \mathcal{C}_2;\theta_2$ *for some* τ' *with* $\tau' =_{\mathsf{fv}(\mathcal{C}_1;\theta_1)} \tau$ *and* $\mathsf{dom}(\tau') \subseteq \mathsf{fv}(\mathcal{C}_2;\theta_2)$

(2) $\tau \vDash \mathcal{C}_2;\theta_2$ *implies* $\tau \vDash \mathcal{C}_1;\theta_1$

Proof. The only possible case is rule (Solve):

$$\{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C};\theta \rightsquigarrow \mathcal{C}[\sigma];\sigma \cup \theta[\sigma]$$

where σ is a m.g.u. of l_1 and l_2. We show each point separately.

[9]Note that this property is not true in Robinson's unification algorithm, because the step eliminating a trivial equation $\{x \stackrel{?}{=} x\} \cup \mathcal{C};\theta \rightsquigarrow \mathcal{C};\theta$ may decrease the set of free variables of the configuration.

(1) By hypothesis we have $l_1[\tau] \simeq l_2[\tau]$, so because σ is a m.g.u. for l_1 and l_2 it follows that for some θ' we have $i[\sigma][\theta'] \simeq i[\tau]$ for all $i \in \mathsf{fv}(l_1, l_2)$. In the following, we suppose wlog that $\mathsf{dom}(\theta') \subseteq \mathsf{vrange}(\sigma)$ — otherwise we just take the restriction of θ' to $\mathsf{vrange}(\sigma)$, and the equation $i[\sigma][\theta'] \simeq i[\tau]$ still holds.

We first claim that for some τ' we have $i[\sigma][\tau'] \simeq i[\tau]$ for all $i \in \mathsf{fv}(\{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C}; \theta)$. Because $\mathsf{dom}(\tau) \subseteq \mathsf{fv}(\{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C}; \theta)$ and $\mathsf{dom}(\theta') \subseteq \mathsf{vrange}(\sigma)$ and $\mathsf{vrange}(\sigma)$ only contains fresh variables, we have $\mathsf{dom}(\theta') \cap \mathsf{dom}(\tau) = \varnothing$, allowing us to define $\tau' := \tau \cup \theta'$. If $i \in \mathsf{fv}(l_1, l_2)$, then $i[\sigma]$ only contains fresh variables, and because τ' and θ' agree on fresh variables, we have $i[\sigma][\tau'] = i[\sigma][\theta'] \simeq i[\tau]$. Finally, if $i \in \mathsf{fv}(\{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C}; \theta) \backslash \mathsf{fv}(l_1, l_2)$ then because $\mathsf{fv}(l_1, l_2) = \mathsf{dom}(\sigma)$ we have $i[\sigma][\tau'] = i[\tau']$, and because τ and τ' agree on non-fresh variables we have $i[\tau'] = i[\tau]$.

By the above claim, for each $l'_1 \stackrel{?}{=} l'_2 \in \mathcal{C}; \theta$ we have $l'_p[\tau] \simeq l'_p[\sigma][\tau']$ for $p = 1, 2$, so $\tau \vDash \mathcal{C}; \theta$ implies $\sigma[\tau'] \vDash \mathcal{C}; \theta$ and thus $\tau' \vDash \mathcal{C}[\sigma]; \theta[\sigma]$. Finally, the claim also implies $i[\sigma][\tau'] \simeq i[\tau']$ for all $i \in \mathsf{dom}(\sigma)$, given that $i[\tau] = i[\tau']$ for i not fresh. Hence, we conclude $\tau' \vDash \mathcal{C}[\sigma]; \sigma \cup \theta[\sigma]$ as required.

(2) By hypothesis we have $i[\tau] \simeq i[\sigma][\tau]$ for all $i \in \mathsf{dom}(\sigma)$, and the equation trivially holds for $i \notin \mathsf{dom}(\sigma)$, given that in this case $i[\sigma] = i$. Therefore, from $\tau \vDash \mathcal{C}[\sigma]; \theta[\sigma]$ we get $\tau \vDash \mathcal{C}; \theta$. Finally, because σ unifies l_1 and l_2, we have $l_1[\sigma][\tau] \simeq l_2[\sigma][\tau]$, and because $l_p[\tau] \simeq l_p[\sigma][\tau]$ for $p = 1, 2$, we get $l_1[\tau] \simeq l_2[\tau]$. We conclude $\tau \vDash \{l_1 \stackrel{?}{=} l_2\} \cup \mathcal{C}; \theta$. □

The key lemma then leads to the correctness of the unification algorithm.

Theorem 6.21 (Correctness of unification)**.** *If $\mathcal{C}; \varnothing \rightsquigarrow^* \varnothing; \theta$ then θ is a most general unifier for $\mathcal{C}$, and if $\mathcal{C}; \varnothing \rightsquigarrow^* \bot$ then $\mathcal{C}$ has no unifier.*

Proof. Suppose that $\mathcal{C}; \varnothing \rightsquigarrow^* \varnothing; \theta$. Because θ is idempotent, we have $\theta \vDash \varnothing; \theta$, so by iterating Lemma 6.20 we get $\theta \vDash \mathcal{C}; \varnothing$, showing that θ is a unifier for $\mathcal{C}$. To see it is a most general one, consider any other unifier τ, and let τ' be its restriction to variables occurring in $\mathcal{C}$. Then by iterating Lemma 6.20 we get $\tau'' \vDash \varnothing; \theta$ for some $\tau'' =_{\mathsf{fv}(\mathcal{C})} \tau'$, hence $i[\tau''] \simeq i[\theta][\tau'']$ for all $i \in \mathsf{dom}(\theta)$. But because this equation also holds trivially for $i \notin \mathsf{dom}(\theta)$, we get $i[\tau''] \simeq i[\theta][\tau'']$ for all i. Finally, because τ and τ' and τ'' all agree on $\mathsf{fv}(\mathcal{C})$, then we get $i[\tau] \simeq i[\theta][\tau'']$ for all $i \in \mathsf{fv}(\mathcal{C})$.

Now suppose that $\mathcal{C}; \varnothing \rightsquigarrow^* \bot$. Then we have $\mathcal{C}; \varnothing \rightsquigarrow^* \mathcal{C}'; \theta' \rightsquigarrow \bot$. If τ is a unifier for $\mathcal{C}$, then by iterating Lemma 6.20 with the restriction of τ to $\mathsf{fv}(\mathcal{C})$, we get a unifier for $\mathcal{C}'$. But if $\mathcal{C}'; \theta' \rightsquigarrow \bot$, then $\mathcal{C}'$ must contain an unsolvable equation, a contradiction. □

Remark 6.22. We note that the correctness proofs do not rely on any specificity of the equational theory of universe levels, and therefore the algorithm of Figure 6 can be used with any equational theory in which one can compute a m.g.u. for two terms when it exists.

Because our algorithm uses Theorem 6.11, which gives a complete characterization of the equations that admit a m.g.u., it follows that our algorithm is complete for solving equations, in the sense that it can always find a m.g.u. for an equation that admits one. We can then wonder whether if it is also complete for problems that contain more than one equation. The following example shows that this is not the case.

Example 6.23. Consider the problem $\mathcal{C} := \{1 + i_0 \stackrel{?}{=} i_2 \sqcup 1 + i_1,\ 1 + i_0 \stackrel{?}{=} i_1 \sqcup 1 + i_2\}$. We can check that, according to Theorem 6.11, both equations are solvable but admit no most general unifiers, so neither the step (Solve) nor (Fail) apply. Nevertheless, by combining

both equations we get $i_2 \sqcup 1 + i_1 \stackrel{?}{=} i_1 \sqcup 1 + i_2$, whose canonical form is $0 \sqcup i_1 \stackrel{?}{=} 0 \sqcup i_2$. Therefore, $\mathcal{C}$ is equivalent to $\mathcal{C} \cup \{0 \sqcup i_1 \stackrel{?}{=} 0 \sqcup i_2\}$, a problem that can be solved by our algorithm, yielding the m.g.u. $\theta = i_1 \mapsto 0 \sqcup i_2,\ i_0 \mapsto 0 \sqcup i_2$. It follows that $\mathcal{C}$ also admits θ as a m.g.u., yet our algorithm does not return any m.g.u., showing it is not complete for problems with more than one equation.

Moreover, Theorem 6.2 shows that, even if our algorithm were complete, it would still get stuck in problems which are solvable but admit no m.g.u. 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 computed substitution is still a unifier, but might not be a most general one. This means that the term which generated the unification problem can still be translated to a valid term in $\mathbb{UPP}$, but the resulting term might not be a most general universe-polymorphic instance.

7. Predicativize, the implementation

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

Our tool is implemented on top of DkCheck [Sai15], a type-checker for Dedukti, and thus does not rely neither on the codebase of Agda, nor on the codebase of any other proof assistant. Like in the case of Universo [Thi20], we instrument DkCheck's conversion checker in order to implement the computation of level constraints.

Because the currently available type-checkers for Dedukti do not implement rewriting modulo for equational theories other than AC (associativity-commutativity), we used Genestier's encoding of the equational theory of universe levels [Gen20] in order to define the theory $\mathbb{UPP}$ in a DkCheck file.

We also note that for the moment the implementation lags behind the theory in various places, in particular by still using the older unification algorithm and the previous version of $\mathbb{UPP}$ proposed in our previous work [FBB23].

To see how everything works in practice, one can run `make running-example` which translates our running example and produces 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 [Thi18] to $\mathbb{UPP}$.

In the following, let us give a high-level description of some of the practical differences with the theory presented until now.

User added constraints. As we have seen, our transformation 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 7.1. Consider the local signature

$$\Phi = \mathsf{Nat} : \mathsf{Tm}\ \mathsf{U}_\square,\ \mathsf{zero} : \mathsf{Tm}\ \mathsf{Nat},\ \mathsf{succ} : \mathsf{Tm}\ \mathsf{Nat} \to \mathsf{Tm}\ \mathsf{Nat}$$

defining the natural numbers in $\mathbb{I}$. The translation of this signature by our algorithm is

$$\Phi' = \mathsf{Nat} : (i : \mathsf{Lvl}) \to \mathsf{Tm}\ \mathsf{U}_i,\ \mathsf{zero} : (i : \mathsf{Lvl}) \to \mathsf{Tm}\ (\mathsf{Nat}\ i),$$
$$\mathsf{succ} : (i\ j : \mathsf{Lvl}) \to \mathsf{Tm}\ (\mathsf{Nat}\ i) \to \mathsf{Tm}\ (\mathsf{Nat}\ j)$$

However, we normally would like to impose i to be equal to j in the type of succ, or even to impose Nat not to 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 the bottom universe, or $i = j$ in the type of the successor. Adding constraints can also help the unification algorithm, which can be particularly useful for simplifying unification problems when translating definitions that do not need to be universe-polymorphic.

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 [Ass15, Fer21, Thi20]. If we simply ignore these rules when performing the translation, we would run into problems as the entries that appear after may need them to typecheck.

Therefore, our implementation extends the presented algorithm to 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 experimental, this step requires user intervention in most cases. This is done by adding new constraints over the symbols appearing in the rules, in order for their translations to be less universe-polymorphic, which helps the algorithm. This part of the translation is yet to be formally defined, and its correctness is still to be proven. Nevertheless, it has been successfully used on the translation of Matita's arithmetic library to Agda.

Agda output. Predicativize produces proofs in the theory $\mathbb{UPP}$, which is a subtheory of the one implemented by the Agda proof assistant. In order to produce proofs that can be used by Agda, we also integrated in Predicativize a translator that performs a simple syntactical translation from a Dedukti file in the theory $\mathbb{UPP}$ to an Agda file. 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

We now discuss how we used Predicativize to translate Matita's arithmetic library to Agda. The translation is summarized in Figure 7, where $\mathbb{CIC}$ stands for a Dedukti theory defining the Calculus of Inductive Constructions, the underlying type theory of the Matita proof assistant.

Matita's arithmetic library [mat] was already available in Dedukti thanks to Krajono [Ass15, Dedb], a translator from Matita to the theory $\mathbb{CIC}$ in Dedukti. Therefore, the first step of the translation was already done for us.

Then, using Predicativize we translated the library from $\mathbb{CIC}$ to $\mathbb{UPP}$. As the encoding of Matita's recursive functions uses rewrite rules, their translation required some user

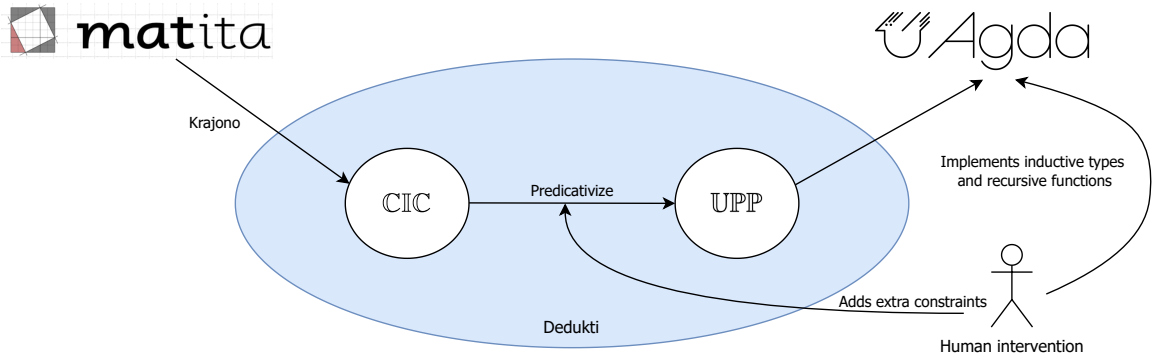

Figure 7: Diagram representing the translation of Matita's arithmetic library into Agda

intervention to add constraints over certain constants, as mentioned in the previous section. Moreover, in order to help the unification algorithm, we also added constraints for fixing the levels of many definitions which were only required to be at one universe. The list of all added constraints can be found in the file `extra_cstrs/matita.dk` in the implementation. These were obtained, for each of the concerned definitions, by looking at its (unconstrained) output and then adding equations involving some of its level variables — similarly to how i and j can be equated in Example 7.1. Once this step is done, the library is known to be predicative, as it typechecks in $\mathbb{UPP}$.

We then used Predicativize to translate these files to Agda files. However, because the rewrite rules in the Dedukti files 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. To do this we first assembled the type formers and constructors, which had been translated simply as postulates, into inductive type declarations. This required us to add further constraints for some constants, for instance between i and j in the type of the successor (Example 7.1), in order to implement them as constructors of an inductive type.

With the inductive types defined, we could then define the recursive functions (like addition), which had been translated as postulates with no computational content. Thankfully, even if we cannot translate the rewrite rules from Dedukti to Agda in a way that is accepted by Agda, we could still translate them as comments in the Agda files. Then, instead of writing such functions from scratch, we could just adapt these comments into valid Agda function declarations. We believe that this step, also needed in previous work [Thi18], could be automated by better studying the translation between different representations of recursive functions. Nevertheless, because most of Matita's arithmetic library is made of proofs, whose translation we do not need to change, automating it was not crucial in our case, so we decided to leave this study for future work.

The result of the translation is available at

`https://doi.org/10.5281/zenodo.10686897`

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 the `--safe` flag, attesting that it does not use any of Agda's more exotic and unsafe features.

	Matita	Dedukti ($\mathbb{CIC}$)	Dedukti ($\mathbb{UPP}$)	Agda
File size (in Kb)	67	640	570	190

Table 1: Comparison of (compressed) file sizes

We conclude by discussing some statistics about the translation. The total translation time, from Dedukti ($\mathbb{CIC}$) to Dedukti ($\mathbb{UPP}$) and then to Agda, is about 32 minutes on a machine with an i7 processor. We also provide in Table 1 a comparison of the file sizes in Matita, Dedukti (in both theories $\mathbb{CIC}$ and $\mathbb{UPP}$) and Agda. Here we chose to analyze their compressed sizes (using `.tar.xz`) to avoid discrepancies arising from administrative differences in the files and formats. As we see, the translation from Matita to Dedukti ($\mathbb{CIC}$) increases a lot the file sizes, which are multiplied by almost 10. This is not surprising, as the original proofs are done using tactics, that are compiled to proof terms when going to Dedukti. Moreover, the representation of terms in Dedukti is much more annotated and low-level than in commonly-used proof assistants, which also explains why some of this extra size is eliminated when going from Dedukti ($\mathbb{UPP}$) to Agda. Yet, the proofs in Agda are still much more low-level than their Matita counterparts given that they still use proof terms instead of tactics. Finally, we see that going from Dedukti ($\mathbb{CIC}$) to Dedukti ($\mathbb{UPP}$) only mildly alters the file sizes, which is not surprising since our translation does not drastically change the terms.

9. Conclusion

We have proposed a transformation for sharing proofs with predicative systems. Our implementation allowed to translate many non-trivial proofs from Matita's arithmetic library to Agda, showing that our proposal works well in practice.

Our solution is based on the use of universe-polymorphic elaboration. Even if elaboration algorithms are already well-studied in the literature, our proposal differs from most on the use of universe level unification, which is needed in our setting for handling universe polymorphism. Other proposals for universe-polymorphic elaboration such as [HP91] and [ST14] avoid the use of universe level unification by allowing in their target languages for entries in the signature to come with associated sets of constraints, which are then verified locally at each use. This feature is however unfortunately not supported by Agda, the main target of our translation.

Our proposal thus required us to study the problem of universe level unification. In order to provide an algorithm for this problem, we first contributed with a complete characterization of which equations admit a m.g.u., along with an explicit description of a m.g.u. when it exists. We then employed this characterization in the design of a unification algorithm, which is an improvement over our preliminary work [FBB23]. It is in particular able to solve all equations that admit a m.g.u., whereas the algorithm of [FBB23] was not — for instance, it was not capable of solving the first equation of Example 6.12. However some problems admitting a m.g.u. cannot be solved by our algorithm because they combine multiple equations, none of them admitting a m.g.u. (see Example 6.23). Even if our practical results show that our algorithm is already sufficiently powerful for our needs, one can wonder if a complete unification algorithm exists. We leave this interesting but difficult problem for future work.

Agda also features an algorithm for solving level metavariables, but it does not seem to have been formally specified or proven correct in the literature, making it hard to provide a detailed comparison with our work. However, practical tests seem to suggest that our algorithm is an improvement. As an example, typechecking in Agda the entry

$$\mathsf{test} : (A : \mathsf{Set}\ _) \to (B : \mathsf{Set}\ _) \to (C : \mathsf{Set}\ _) \to (R : (D : \mathsf{Set}\ _) \to D \to D \to \mathsf{Set}\ _) \to \mathsf{Set}\ _$$
$$\mathsf{test} = \lambda A\ B\ C\ R \to R\ (\mathsf{Set}\ _)\ (A \to C)\ (A \to B)$$

gives the error `Failed to solve the following constraints: _0 ⊔ _1 = _0 ⊔ _2`, however this constraint is solvable by our algorithm (see Example 6.12). Therefore, our work could also be used to improve Agda's unification algorithm.

For future work, we would also like to look 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. We thus expect improvements in this direction to be needed in order to translate larger proof libraries.

Acknowledgment

The authors would like to thank Ashish Kumar Barnawal for the very helpful discussions that led to this article, François Thiré for the help while developing Predicativize, Gilles Dowek for remarks about previous versions of this paper, Jesper Cockx and Vincent Moreau for discussions about universe levels and the anonymous reviewers of both CSL and LMCS for their very helpful comments and remarks.

References

[ABC+16] 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. Unpublished, 2016.

[ABKT19] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory—a syntactic translation. In *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings 13*, pages 155–196. Springer, 2019.

[ADJL17] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Untyped Confluence In Dependent Type Theories. working paper or preprint, April 2017. URL: `https://hal.inria.fr/hal-01515505`.

[AR12] Andrea Asperti and Wilmer Ricciotti. A proof of bertrand's postulate. *Journal of Formalized Reasoning*, 5(1):37–57, 2012.

[Ass15] Ali Assaf. *A framework for defining computational higher-order logics*. These, École polytechnique, September 2015. URL: `https://pastel.archives-ouvertes.fr/tel-01235303`.

[Bar93] H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993.

[BB88] Franz Baader and Wolfram Büttner. Unification in commutative idempotent monoids. *Theoretical Computer Science*, 56(3):345–353, 1988. `doi:10.1016/0304-3975(88)90140-5`.

[BCDE23] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. Type Theory with Explicit Universe Polymorphism. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.TYPES.2022.13`.

[BDG+23] Frédéric Blanqui, Gilles Dowek, Emilie Grienenberger, Gabriel Hondet, and François Thiré. A modular construction of type theories. *Logical Methods in Computer Science*, Volume 19, Issue 1, February 2023. `doi:10.46298/lmcs-19(1:12)2023`.

[BKdVT03] Marc Bezem, Jan Willem Klop, Roel de Vrijer, and Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[Bla01] 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`.

[Bla03] Frédéric Blanqui. Rewriting modulo in deduction modulo. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 2706, 2003. 15 pages.

[Bla05] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[Bla22] Frédéric 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.

[BNW19] Michael Beeson, Julien Narboux, and Freek Wiedijk. Proof-checking Euclid. *Annals of Mathematics and Artificial Intelligence*, page 53, January 2019. `doi:10.1007/s10472-018-9606-x`.

[BS94] Franz Baader and Jörg H Siekmann. Unification theory., 1994.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Coq13] Thierry Coquand. Presheaf model of type theory. Unpublished note available at `http://www.cse.chalmers.se/~coquand/presheaf.pdf`, 2013.

[Deda] Deducteam. Lambdapi. URL: `https://github.com/Deducteam/lambdapi` [cited 2023].

[Dedb] Deducteam. Matita's arithmetic library in Dedukti. URL: `https://github.com/Deducteam/Deducteam.github.io/blob/master/data/libraries/matita.tar.gz`.

[Del20] 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`.

[DHKP96] Gilles Dowek, Thérese Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *JICSLP*, pages 259–273, 1996.

[Dow93] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, page 139–145, Berlin, Heidelberg, 1993. Springer-Verlag.

[FBB23] Thiago Felicissimo, Frédéric Blanqui, and Ashish Kumar Barnawal. Translating proofs from an impredicative type system to a predicative one. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland*, volume 252 of *LIPIcs*, pages 19:1–19:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.CSL.2023.19`.

[Fel22] Thiago Felicissimo. Adequate and Computational Encodings in the Logical Framework Dedukti. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.FSCD.2022.25`.

[Fel24] Thiago Felicissimo. Generic bidirectional typing for dependent type theories. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 143–170, Cham, 2024. Springer Nature Switzerland.

[Fer21] 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`.

[Gen20] 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`.

[Geu93] Herman Geuvers. *Logics and type systems*. PhD thesis, University of Nijmegen, 1993.

[Gé] Yoan Géran. Euclid's elements book 1 in dedukti. URL: `https://github.com/Karnaj/sttfa_geocoq_euclid` [cited 2022].

[HP91] 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.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM (JACM)*, 27(4):797–821, 1980.

[KHS19] Ambrus Kaposi, Simon Huber, and Christian Sattler. Gluing for type theory. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[Klo63] Jan Willem Klop. *Combinatory reduction systems*. PhD thesis, Rijksuniversiteit Utrecht, 1963.

[Kvv93] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1):279–308, 1993. doi:10.1016/0304-3975(93)90091-7.

[mat] Matita's arithmetic library. URL: https://github.com/LPCIC/matita/tree/master/matita/matita/lib/arithmetics.

[MN98] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 192(1):3–29, 1998.

[MW96] Paul-André Mellies and Benjamin Werner. A generic normalisation proof for pure type systems. In *International Workshop on Types for Proofs and Programs*, pages 254–276. Springer, 1996.

[Ree09] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 49–56, 2009.

[Sai15] Ronan Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In *International Conference on Interactive Theorem Proving*, pages 499–514. Springer, 2014.

[Ste19] Jonathan Sterling. Algebraic type theory and universe hierarchies. *arXiv preprint arXiv:1902.08848*, 2019.

[Tea] Agda Development Team. Agda 2.6.2.1 documentation. URL: https://agda.readthedocs.io/en/v2.6.2.1/index.html [cited 2022].

[Thi18] 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.

[Thi20] François Thiré. *Interoperability between proof systems using the logical framework Dedukti*. PhD thesis, ENS Paris-Saclay, 2020.

[Voe14] Vladimir Voevodsky. A universe polymorphic type system, October 22, 2014. An unfinished unreleased manuscript. URL: http://www.math.ias.edu/Voevodsky/voevodsky-publications_abstracts.html#UPTS.

[ZS17] Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for cic including universe polymorphism and overloading. *Journal of Functional Programming*, 27:e10, 2017.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany