A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading

Beta Ziliani
MPI-SWS
beta@mpi-sws.org

Matthieu Sozeau
Inria & PPS, Université Paris Diderot
matthieu.sozeau@inria.fr

Abstract

Unification is a core component of every proof assistant or programming language featuring dependent types. In many cases, it must deal with higher-order problems up to conversion. Since unification in such conditions is undecidable, unification algorithms may include several heuristics to solve common problems. However, when the stack of heuristics grows large, the result and complexity of the algorithm can become unpredictable.

Our contributions are twofold: (1) We present a full description of a new unification algorithm for the Calculus of Inductive Constructions (the base logic of Coq), including universe polymorphism, canonical structures (the overloading mechanism baked into Coq's unification), and a small set of useful heuristics. (2) We implemented our algorithm, and tested it on several libraries, providing evidence that the selected set of heuristics suffices for large developments.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.4.1 [*Mathematical Logic And Formal Languages*]: Mathematical Logic—Mechanical theorem proving

Keywords Interactive theorem proving; unification; Coq; universe polymorphism; overloading.

1. Introduction

In the last decade proof assistants have become more sophisticated and, as a consequence, increasingly adopted by computer scientists and mathematicians. In particular, they are being adopted to help dealing with very complex proofs, proofs that are hard to grasp—and more importantly, to *trust*—for a human. For example, in the area of algebra, the Feit-Thompson Theorem was recently formalized [10] in the proof assistant Coq [22]. To provide a sense of the accomplishment of Gonthier and his team, the original proof of this theorem was published in two volumes, totaling an astounding 250 pages. The team formalized it entirely in Coq, together with several books of algebra required as background material.

In order to make proofs manageable, this project relies heavily on the ability of Coq's unification algorithm to infer implicit arguments and expand heavily overloaded functions. This goes to the point that it is not rare to find in the source files a short definition that is expanded, by the unification algorithm, into several lines of code in the *Calculus of (co-)Inductive Constructions* (CIC), the base logic of Coq. This expansion is possible thanks to the use of the overloading mechanism in Coq called *canonical structures* [20]. This mechanism, similar in spirit to Haskell's *type classes*, is baked into the unification algorithm. By being part of unification, this mechanism has a unique opportunity to drive unification to solve particular unification problems in a similar fashion to Matita's *hints* [3]. It is so powerful, in fact, that it enables the development of dependently-typed logic meta-programs [12].

Another important aspect of the algorithm is that it must deal with higher-order problems, which are inherently undecidable, up-to a subtyping relation on universes. For this reason, the current implementation of the unification algorithm has grown with several heuristics, yielding acceptable solutions to common problems in practice. Unfortunately, the algorithm is unpredictable and hard to reason about: given a unification problem, it is hard to predict the substitution the algorithm will return, and the time complexity for the task. This unpredictability of the current implemented algorithm has two main reasons: (*i*) it lacks a specification, and (*ii*) it incorporates a number of heuristics that obfuscate the order in which unification subproblems are considered.

While the algorithm being unpredictable is bad on its own, the problem gets exacerbated when combined with canonical structures, since their resolution may depend on the solutions obtained in previous unification problems. To somehow accomodate for this unfortunate situation, several works in the literature explain canonical structures by example [8, 9, 12, 13], providing some intuition on how canonical structures work, in some cases even detailing certain necessary aspects of the unification process. However, they fall short of explaining the complex process of unification as a whole.

This paper presents our remedy to the current situation. More precisely, our four main contributions are:

1. An original, full-fledged description of a unification algorithm for CIC, incorporating canonical structures and universe polymorphism [21].
2. The first formal description, to the best of our knowledge, of an extremely useful heuristic implemented in the unification algorithm of Coq, *controlled backtracking*.
3. A corresponding pluggable implementation, incorporating only a restricted set of heuristics, such as controlled backtracking. Most notably, we purposely left out a technique known as *constraint postponement*, present in many systems and in the current implementation in Coq, which may reorder unification subproblems. This reordering prevents us from knowing exactly when equations are being solved.

$\mathsf{in\_head} : \forall\ (x : A)\ (l : \mathsf{list}\ A),\ x \in (x :: l)$
$\mathsf{in\_tail} : \forall\ (x : A)\ (y : A)\ (l : \mathsf{list}\ A),\ x \in l \to x \in (y :: l)$

Lemma $\mathsf{inL} : \forall\ (x : A)\ (l\ r : \mathsf{list}\ A),\ x \in l \to x \in (l +\!\!+ r)$
Lemma $\mathsf{inR} : \forall\ (x : A)\ (l\ r : \mathsf{list}\ A),\ x \in r \to x \in (l +\!\!+ r)$

Figure 1. List membership axioms and lemmas.

4. Evidence that such principled heuristics suffice to solve 99.9% of the unification problems that arise in libraries such as the Mathematical Components library [11] and CPDT [7].

It is interesting to note that during this work we found two bugs in the logic of the original unification algorithm of Coq. While this work focuses on the Coq proof assistant, the problems and solutions presented may be of interest to other type theory based assistants and programming languages, such as Agda [16], Matita [2], or Idris [5].

In the rest of the paper, we start introducing with examples some features and heuristics included in Coq's unification algorithm (§2). Then, we present the language used in the paper (§3), necessary to understand the core contribution of this work, a new unification algorithm (§4). We evaluate the algorithm (§5) and conclude (§6).

2. Coq's Unification at a Glance

We start by showing little examples highlighting some of the particularities of Coq's unification algorithm.

First-order approximation: In many cases, a unification problem may have several incomparable solutions. Consider for instance the following definition in a context where y_1 and y_2 are defined:

$$\textbf{Definition}\ \mathsf{ex0} : y_1 \in ([y_1] +\!\!+ [y_2]) := \mathsf{inL}\ \_\ \_\ \_\ (\mathsf{in\_head}\ \_\ \_)$$

We assume the definitions and lemmas for list membership listed in Figure 1, and note $(x :: s)$ for the *consing* of x to list s, [] for the empty list, and $l +\!\!+ r$ for the concatenation of lists l and r. We also denote $[a_1; \ldots; a_n]$ a list with elements a_1 to a_n.

This definition is a proof that the element y_1 is in the list resulting from concatenating the singleton lists $[y_1]$ and $[y_2]$. The proof in itself provides evidence that the element is in the head (in_head) of the list on the left (inL). As customary in Coq code, the type annotation shows what the definition is proving, and the proof omits the information that can be inferred, replacing each argument to inL and in_head with *holes* (_). The elaboration mechanism of Coq, that is, the algorithm in charge of filling up these holes, calls the unification algorithm with the following unification problem, where the left-hand side corresponds to what the body of the definition proves, and the right-hand side to what it is expected to prove:[1]

$$?z_1 \in ((?z_1 :: ?z_2) +\!\!+ ?z_3) \ \approx\ y_1 \in ([y_1] +\!\!+ [y_2])$$

where $?z_1, ?z_2$ and $?z_3$ are fresh meta-variables. In turn, after assigning y_1 to $?z_1$, the unification algorithm has to solve the following problem:

$$(y_1 :: ?z_2) +\!\!+ ?z_3 \ \approx\ [y_1] +\!\!+ [y_2]$$

One possible solution to this equation is to assign [] to $?z_2$, and $[y_2]$ to $?z_3$, which corresponds to equate each argument of the concatenation, similar to what we did before with the $\in$ *predicate*. However, since concatenation is a *function*, *i.e.*, it computes the concatenation of the two lists, there are other possible solutions that makes both terms *convertible* (*i.e.*, having the same normal form). One such solution, for instance, is to assign $[y_2]$ to $?z_2$, and [] to $?z_3$.

Many works in the literature [*e.g.*, 1, 14, 17, 18] are devoted to the creation of unification algorithms returning a *Most General Unifier* (MGU), that is, a *unique* solution that serves as a representative for all *convertible* solutions. Agda [16], for instance, which incorporates such type of unification algorithm, fails to compile Example ex0 above, since no such MGU exists. This forces the proof developer to manually fill-in the holes.

Despite the equation having multiple solutions, however, not every solution is equally "good". For ex0, the first solution is the most *natural* one, meaning the one expected by the proof developer. For this reason, instead of failing, Coq favors syntactic equality by trying first-order unification. Formally, when faced with a problem of the form

$$t\ t_1\ \ldots\ t_n\ \approx\ u\ u_1\ \ldots\ u_n$$

the algorithm decomposes the problem into $n + 1$ subproblems, first equating $t \approx u$, and then $t_i \approx u_i$, for $0 < i \leq n$.

Controlled backtracking: In [19, chp. 10], a unification algorithm for CIC is presented, performing *only* first-order unification. In Coq, instead, when first-order approximation fails, in an effort to find a solution to the equation, the algorithm reduces the terms *carefully*. For instance, consider the following variation of the previous example, where the list on the left of the concatenation is **let**-bound:

$$\begin{aligned}\textbf{Definition}\ \mathsf{ex1} : y_1 \in (\textbf{let}\ l := [y_1]\ \textbf{in}\ (l +\!\!+ [y_2])) \\ := \mathsf{inL}\ \_\ \_\ \_\ (\mathsf{in\_head}\ \_\ \_)\end{aligned}$$

The main equation to solve now is

$$(y_1 :: ?z_2) +\!\!+ ?z_3 \ \approx\ \textbf{let}\ l := [y_1]\ \textbf{in}\ (l +\!\!+ [y_2])$$

Since both terms do not share the same *head* (the concatenation operator on the left and the **let**-binding on the right), the algorithm reduces the **let**-binding, obtaining the same problem as in ex0. Note that it has to be careful: it should not reduce the concatenation operator, otherwise the problem will become unsolvable. For this reason, it delays the unfolding of constants, such as $+\!\!+$, and, in the case of having constants on both sides of the equation, it takes special care of which one to unfold. This heuristic enables fine control over the instance resolution mechanism of canonical structures [12].

Canonical structures: *Canonical structures* (CS) is a powerful overloading mechanism, baked into the unification algorithm. We demonstrate this mechanism with a typical example from overloading: the equality operator. Similar to how type classes are used in Haskell [23], we define a *class* or, in CS terminology, a *structure*:[2]

$$\begin{aligned}\textbf{Structure}\ \mathsf{eqType} := \mathsf{EqType}\ \{\ &\mathsf{sort} : \mathsf{Type}; \\ &\mathsf{equal} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{bool}\ \}\end{aligned}$$

eqType is a record type with two fields: a type sort, and a boolean binary operation equal on sort. These fields can be accessed using projectors:

$$\begin{aligned}\mathsf{sort} &: \mathsf{eqType} \to \mathsf{Type} \\ \mathsf{equal} &: \forall e{:}\mathsf{eqType}.\ \mathsf{sort}\ e \to \mathsf{sort}\ e \to \mathsf{bool}\end{aligned}$$

To construct an element of the type, the constructor EqType is provided, which takes the values for the two fields as arguments. For example, one possible eqType *instance* for bool is:

$$\textbf{Definition}\ \mathsf{eqType\_bool} := \mathsf{EqType}\ \mathsf{bool}\ \mathsf{eq\_bool}$$

[1] How elaboration works will not be discussed in this work. The interested reader is invited to read [4], which provides details on bi-directional elaboration in the Matita proof assistant, also based on CIC.

[2] This example is a significant simplification of one taken from [11, 12].

where $\mathsf{eq\_bool}\ x\ y := (x\ \&\&\ y)\ ||\ (!x\ \&\&\ !y)$. (We denote boolean conjunction, disjunction and negation as &&, || and !.)

Similarly, it is possible to declare *recursive* instances. For example, consider the instance for the pair type $A \times B$, where A and B are themselves instances of eqType:

$$\textbf{Definition}\ \mathsf{eqType\_pair}\ (A\ B : \mathsf{eqType}) :=$$
$$\mathsf{EqType}\ (\mathsf{sort}\ A \times \mathsf{sort}\ B)\ (\mathsf{eq\_pair}\ A\ B)$$

where

$$\mathsf{eq\_pair}\ (A\ B : \mathsf{eqType})\ (u\ v : \mathsf{sort}\ A \times \mathsf{sort}\ B) :=$$
$$(\mathsf{equal}\ A\ (\pi_1\ u)\ (\pi_1\ v))\ \&\&\ (\mathsf{equal}\ B\ (\pi_2\ u)\ (\pi_2\ v))$$

In order to use instances $\mathsf{eq\_bool}$ and $\mathsf{eq\_pair}$ for overloading, we need to declare them as **Canonical**. After they have been declared canonical, whenever the elaboration mechanism is asked to elaborate a term like $\mathsf{equal}\ \_\ (b_1, b_2)\ (c_1, c_2)$, for booleans b_1, b_2, c_1 and c_2, it will generate a unification problem matching the expected and inferred type of the second argument of equal, that is,

$$\mathsf{sort}\ ?e\ \approx\ \mathsf{bool} \times \mathsf{bool}$$

for some meta-variable $?e$ elaborated from the *hole* ($\_$).

To solve the equation above, Coq's unification will try instantiating $?e$ using the canonical instance $\mathsf{eqType\_pair}$, resulting in two new unification subproblems, for fresh meta-variables $?A$ and $?B$:

$$\mathsf{sort}\ ?A\ \approx\ \mathsf{bool} \qquad \mathsf{sort}\ ?B\ \approx\ \mathsf{bool}$$

Next, it will choose $?A := \mathsf{eqType\_bool}$ and $?B := \mathsf{eqType\_bool}$, resulting in that $\mathsf{equal}\ ?e\ (b_1, b_2)\ (c_1, c_2)$ reduces, as expected, to $\mathsf{eq\_bool}\ b_1\ c_1\ \&\&\ \mathsf{eq\_bool}\ b_2\ c_2$.

We can declare a number of canonical eqType instances for our types equipped with decidable equality. Then, we can uniformly write $\mathsf{equal}\ \_\ t\ u$, and let unification compute the corresponding instance for the hole, according to the type of t and u.

Polymorphic universes and subtyping: Unification in CIC is not a simple equational theory, in the sense that it must deal with the subtyping relation generated by the cumulative universe hierarchy ($\mathsf{Type}(i) \leq \mathsf{Type}(j) \iff i \leq j$). To our knowledge, we present the first algorithm dealing with this relation properly. In Coq, previous algorithms relied on the kernel to check the proper use of universes, resulting in particular in non-local error reporting and the inability to backtrack on these errors, which becomes crucial in presence of universe polymorphism and first-order approximation.

3. The Language: CIC with Open Terms

Before presenting the algorithm, we need to present the base language of Coq, the Calculus of Inductive Constructions (CIC) [22, chap. 4]. It is a dependently typed λ-calculus extended with inductive types. It also includes co-inductive types, but their formulation is not important for this work, so it will be omitted.

The terms (and types) of the language are defined as

$$\begin{aligned}
t, u, T, U =\ & x \mid c[\overline{\ell}] \mid i[\overline{\ell}] \mid k[\overline{\ell}] \mid s \mid ?x[\sigma] \\
& \mid \forall x : T.\ U \mid \lambda x : T.\ t \mid t\ u \mid \textbf{let}\ x := t : T\ \textbf{in}\ u \\
& \mid \textbf{match}_T\ t\ \textbf{with}\ k_1\ \overline{x_1} \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x_n} \Rightarrow t_n\ \textbf{end} \\
& \mid \textbf{fix}_j\ \{x_1/n_1 : T_1 := t_1; \ldots; x_m/n_m : T_m := t_m\} \\
\sigma =\ & \overline{t} \\
\ell, \kappa \in\ & \mathcal{L} \cup 0^- \\
K =\ & \kappa \mid K + 1 \\
s =\ & \mathsf{Type}(\overline{K^+})
\end{aligned}$$

Terms include variables $x \in \mathcal{V}$, constants $c \in \mathcal{C}$, inductive type constructors $i \in \mathcal{I}$ and constructors $k \in \mathcal{K}$, these last three being applied to universe instances $\overline{\ell}$ built from universe levels $\ell \in \mathcal{L} \cup 0^-$. Terms also includes sorts s, representing algebraic universes. Algebraic universes represent least upper bounds of a (non-empty) set of levels or successors of levels. They are used notably to sort products, e.g. $(\forall A : \mathsf{Type}(i), \mathsf{Type}(j)) : \mathsf{Type}(i{+}1, j)$. The impredicative sort Prop, the type of propositions, is represented as $\mathsf{Type}(0^-)$. Terms may contain a *hole*, representing a missing piece of the term (or proof). Holes are represented with meta-variables, a variable prepended with a question mark, as in $?x$. For reasons that will become apparent soon, meta-variables are applied to a *suspended substitution* $[\sigma]$, which is nothing more than a list of terms.

In order to destruct an element of an inductive type, CIC provides regular pattern **match**ing and mutually recursive **fix**points. Their notation is slightly different from, but easily related to, the actual notation from Coq. **match** is annotated with the return predicate T, meaning that the type of the whole **match** expression may depend on the element being pattern matched (**as** ... **in** ... in standard Coq notation). In the **fix** expression, $x/n : T := t$ means that T is a type starting with at least n product types, and the n-th variable is the decreasing one in t (**struct** in Coq notation). The subscript j of **fix** selects the j-th function as the main entry point of the mutually recursive fixpoints.

In order to typecheck and reduce terms, Coq uses several contexts, each handling different types of knowledge:

1. Universe contexts Φ declaring universe level names and associated constraints ((in-)equalities on levels);
2. Local contexts Γ, including bound variables and **let**-bound expressions;
3. Meta-contexts Σ, containing meta-variable declarations and definitions; and
4. A global environment E, containing the global knowledge; that is, axioms, theorems, and inductive definitions, along with a *global* universe context that can be incrementally enriched.

Formally, they are defined as follows:

$$\begin{aligned}
\Phi &= \overline{\ell} \vDash \mathcal{C} \qquad \mathcal{C} = \cdot \mid \mathcal{C} \wedge \ell\ \mathcal{O}\ \ell' \qquad \text{where } \mathcal{O} \in \{=, \leq, <\} \\
\Gamma, \Psi &= \cdot \mid x : T, \Gamma \mid x := t : T, \Gamma \\
\Sigma &= \cdot \mid ?x : T[\Psi], \Sigma \mid ?x := t : T[\Psi], \Sigma \\
E &= \cdot \mid c[\Phi] : T, E \mid c[\Phi] := t : T, E \mid I, E \mid \Phi, E \\
I &= \forall \Phi, \Gamma.\ \{\ \overline{i : \forall \overline{y : T_h}.\ s := \{k_1 : U_1; \ldots; k_n : U_n\}}\ \}
\end{aligned}$$

A universe context consists of a list of levels $\overline{\ell}$ and a set of constraints $\mathcal{C}$ on those levels. The local context is standard, and requires no further explanation. Meta-variables have *contextual types*, meaning that the type T of a meta-variable must have all of its free variables bound within the local context Ψ. In this work we borrow the notation $T[\Psi]$ from Contextual Modal Type Theory [15]. A meta-variable can be instantiated with a term t, noted $?x := t : T[\Psi]$. In this case, t should also contain only free variables occurring in Ψ.

The global environment associates a constant c with a local universe context (usually omitted), a type and, optionally, a definition. In the first case, c is an axiom, while in the second c is a theorem proved by term t. Additionally, this environment may also contain (mutually recursive) inductive types and global universe and constraint declarations.

A set of mutually recursive inductive types I is prepended with a universe context Φ and a list of parameters Γ. Every inductive type i defined in the set has sort s, with parameters $\overline{y : T_h}$. It has a possibly empty list of constructors $k_1, \ldots, k_n$. For every j, each type U_j of constructor k_j has shape $\forall \overline{z : U'}.\ i\ t_1\ \ldots\ t_h$.

Inductive definitions are restricted to avoid circularity, meaning that every type constructor i can only appear in a strictly positive position in the type of every constructor. For the purpose of this work, understanding this restriction is not crucial, and we refer the interested reader to [22, chap. 4]. Additionally, fixpoints on inductive types must pass the guard condition (ibid., §4.5.5) to be accepted by the kernel, a syntactic criterion ensuring termination. We will come back to this point in §4.7. It is interesting to note that a **Structure**, like the one shown in the previous section, is syntactic sugar for an inductive type with only one constructor, and with projections generated for each argument of the constructor.

3.1 Meta-Variables and Contextual Types

At a high-level, meta-variables are holes in a term, which are expected to be filled out at a later point in time. For instance, when a lemma is applied to solve some goal, Coq internally creates fresh meta-variables for all the formal parameters of the lemma, and proceeds to unify the goal with the conclusion of the lemma. During unification, meta-variables are instantiated so that both terms (the goal and the conclusion of the lemma) become *convertible* (equal modulo reduction rules, see §3.2).

In the simple examples shown so far, contextual types played no role but, as we are going to see in the next example, they prevent illegal instantiations of meta-variables. For instance, such illegal instantiations could potentially happen if the same meta-variable occurs at different locations in a term, with different variables in the scope of each occurrence. We illustrate this point with an example taken from [24]. Suppose function f defined locally as follows:

$$f := \lambda w : \mathsf{nat}.\ (\_ : \mathsf{nat})$$

where the hole $(\_)$ is an indication to Coq's elaboration mechanism to "fill in this hole with a meta-variable". The accessory typing annotation provides the expected type for the meta-variable. Assuming no other variables occur in scope, after elaboration f becomes:

$$f := \lambda w : \mathsf{nat}.\ ?v[w] \tag{1}$$

for some fresh meta-variable $?v$. Since any instantiation of $?v$ may only refer to w, its type becomes $\mathsf{nat}[w : \mathsf{nat}]$. This contextual type specifies precisely that $?v$ may only be instantiated with a term of type nat containing at most a single free variable w of type nat. In the elaborated term (1), $[w]$ stands for the *suspended substitution* specifying how to transform such instantiation into one that is well-typed under the current context. In this case, this substitution is the identity, because the current context and the context under which $?v$ was created are identical (in fact, the latter is a copy of the former).

Now suppose that we define functions g and h referring to f:

$$g := \lambda x\ y : \mathsf{nat}.\ f\ x \qquad h := \lambda z : \mathsf{nat}.\ f\ z$$

and proceed to unify g with a function projecting the first argument:

$$g \ \approx\ \lambda x\ y : \mathsf{nat}.\ x$$

In order to solve this equation, Coq proceeds to unfold the definition of g and to push x and y in the local context. The new equation to solve becomes:

$$f\ x \ \approx\ x$$

After unfolding f and β-reducing the left-hand side, it amounts to solving the following equation:

$$?v[x] \ \approx\ x$$

At this point is where the contextual type of $?v$ comes into play. If meta-variables were created with a normal type, that is, not having contextual type (and suspended substitution), it would seem that the only solution for $?v$ is x. However, that solution would break the definition of h since x is not in scope there. Given the contextual information, however, Coq will correctly realize that

$$(\lambda x : T.\ t)\ u \ \rightsquigarrow_\beta\ t\{u/x\} \qquad \mathbf{let}\ x := u : T\ \mathbf{in}\ t \ \rightsquigarrow_\zeta\ t\{u/x\}$$

$$\frac{(x := t : T) \in \Gamma}{x \ \rightsquigarrow_{\delta\Gamma}\ t} \qquad \frac{?x := t : T[\Psi] \in \Sigma}{?x[\sigma] \ \rightsquigarrow_{\delta\Sigma}\ t\{\sigma/\widehat{\Psi}\}}$$

$$\frac{(c[\overline{\ell} \vDash \mathcal{C}] := t : T) \in E}{c[\overline{\kappa}] \ \rightsquigarrow_{\delta E}\ t[\overline{\kappa/\ell}]}$$

$$\mathbf{match}_T\ k_j[\overline{\kappa}]\ \overline{t}\ \mathbf{with}\ \overline{k\ \overline{x} \Rightarrow u}\ \mathbf{end} \ \rightsquigarrow_\iota\ u_j\{\overline{t/x_j}\}$$

$$\frac{F = \overline{x/n : T := t} \qquad a_n = k_j[\overline{\kappa}]\ \overline{t}}{\mathbf{fix}_j\ \{F\}\ \overline{a} \ \rightsquigarrow_\iota\ t_j\{\overline{\mathbf{fix}_m\ \{F\}/x_m}\}\ \overline{a}}$$

Figure 2. Reduction rules in CIC.

$?v$ should be instantiated with w, not x. Under that instantiation, g will normalize to $\lambda x\ y : \mathsf{nat}.\ x$, and h will normalize to $\lambda z : \mathsf{nat}.\ z$.

The suspended substitution and the contextual type are the tools that the unification algorithm uses to know how to instantiate the meta-variable. The decision to solve $?v[x] \ \approx\ x$ by instantiating $?v : \mathsf{nat}[w : \mathsf{nat}]$ with w is due to the problem falling in the *pattern* unification subset [14]. When Coq faces a problem of the form

$$?u[y_1, \ldots, y_n] \ \approx\ e$$

where the $y_1, \ldots, y_n$ are all distinct variables, then the *most general* solution to the problem is to *invert* the substitution and apply it on the right-hand side of the equation, in other words instantiating $?u$ with $e\{x_1/y_1, \ldots, x_n/y_n\}$, where $x_1, \ldots, x_n$ are the variables in the local context of $?u$ (and assuming the free variables of e are in $\{y_1, \ldots, y_n\}$).

In the example above, at the point where Coq tries to unify $?u[x] \ \approx\ x$, the solution (through inversion) is to instantiate $?u$ with $x\{w/x\}$, that is, w.

3.2 Semantics

Reduction of CIC terms is performed through a set of rules listed in Figure 2. Besides the standard β rule, CIC provides six more rules to destruct the different term constructions: the ζ rule, which expands let-definitions, three δ rules, which expand definitions from each of the contexts, and two ι rules, which evaluate pattern **match**ings and **fix**points.

Most of the rules are self explanatory, with the sole exception of the $\delta\Sigma$ rule. It takes a meta-variable $?x$, applied to suspended substitution σ, and replaces it by its definition t, replacing each variable from its local context Ψ by the corresponding term from substitution σ. For this we use the multi-substitution of terms, mapping the variables coming from the domain of Ψ with terms in σ. To obtain the domain of Ψ, we use the type-eraser function $\widehat{\cdot}$, defined as:

$$\widehat{x_1 : T_1, \ldots, x_n : T_n} = x_1, \ldots, x_n$$

The *unfolding* rules ($\delta\Gamma$, $\delta\Sigma$, δE), of course, depend on the contexts. As customary, we will always consider the environment E implicit. We will also omit Γ and Σ when there is no room for ambiguity.

Conversion ($\equiv$) is defined as the congruent closure of these reduction rules, plus η-conversion: $u \equiv \lambda x : T.u\ x$ iff $x \notin \mathsf{FV}(u)$.

4. The Algorithm

We proceed to describe our proposal in the following pages, underlining every non-standard design decision. We emphasize that this

is not a faithful description of the current unification algorithm in Coq, but a new one that is, however, close enough. In particular, we purposely left out a technique known as *constraint postponement* (§4.6), as well as other heuristics hard to grasp. At the same time, we introduced our own set of heuristics, based on practical examples (§5) and on previous work by Abel and Pientka (§4.3).

The unification judgment is of the form:

$$\Phi; \Sigma; \Gamma \vdash t_1 \approx_{\mathcal{R}} t_2 \rhd \Phi', \Sigma'$$

It unifies terms t_1 and t_2, given a universe context Φ, meta-context Σ and a local context Γ. There is an implicit global environment E. The universe context carries additional information for each universe variable introduced: they are either flexible ($\ell_{\mathtt{f}}$) or rigid ($\ell_{\mathtt{r}}$). This information is used when unifying two instances of the same constant to avoid forcing universe constraints that would not appear if the bodies of the instantiations were unified instead, respecting transparency of the constants. Flexible variables are generated when taking a fresh instance of a polymorphic constant, inductive or constructor during elaboration, while rigid ones correspond to user-specified levels or Type annotations. The relation $\mathcal{R}$ ($\equiv$ or $\leq$) indicates if were are trying to derive conversion of the two terms or *cumulativity*, the subtyping relation on universes. The rules decomposing constructions switch to conversion in their premises except for sorts and dependent function spaces, otherwise the relation is preserved when reducing one side or the other. The algorithm returns a new universe context Φ' and meta-context Σ', which are extensions of Φ and Σ, respectively, perhaps with new universes constraints in Φ', and new meta-variables or instantiations of existing meta-variables in Σ. The algorithm ensures that terms t_1 and t_2 are convertible (or in the cumulativity relation) in the returned contexts.

In the presentation of the algorithm we will often omit the universe context Φ, since it is, in most of the cases, simply threaded along. The unification algorithm is quite involved, so to help readability we split the rules across four different subsections. Roughly, in §4.1 we consider the case when the two terms being unified have no arguments, and share the same head constructor; in §4.2 we consider terms having arguments; in §4.3 we consider meta-variable unification; and in §4.4 we consider canonical structures resolution. The algorithm's strategy, which backtracks in some particular cases, cannot be understood by the ordering of the rules, so we devote §4.5 to explain in detail the algorithm's strategy. In §4.6 we explain the technique of constraint postponement, and the reason for its omission in our algorithm. Finally, in §4.7 we comment on the correctness of the algorithm.

4.1 Same Constructor

Figure 3 shows the rules that apply when both terms share the same head constructor. We need to distinguish this set of rules from the other rules in the algorithm, so we annotate them with a 0 as subscript of the turnstile ($\vdash_0$). The reasons will become evident when we look at the rules in the next subsection.

The rule Type-Same unifies two sorts, according to the relation $\mathcal{R}$. By invariant, we know that the right-hand side universe can only be a single level while the l.h.s. can be the least upper bound of a set of universe levels or successors iff the relation is cumulativity, and any such $\leq$ constraints can be translated to a set of atomic $\leq$ or $<$ constraints (see [21] for details). The predicate $\mathcal{C} \models$ denotes satisfiability of set of constraints $\mathcal{C}$ and naturally extends to consistency of universe contexts ($\phi \models$).

For abstractions (Lam-Same) and products (Prod-Same), we first unify the types of the arguments, and then the body of the binder, with the local context extended with the bound variable. (The universe context is omitted, as in some of the following rules, since it is just threaded along.) When unifying two **let**s, the rule

Type-Same
$$\frac{\mathcal{C}' = \mathcal{C} \wedge \overline{u}\ \mathcal{R}\ \kappa \qquad \mathcal{C}' \models}{\ell \models \mathcal{C}; \Sigma; \Gamma \vdash_0 \mathsf{Type}(\overline{u}) \approx_{\mathcal{R}} \mathsf{Type}(\kappa) \rhd \ell \models \mathcal{C}'; \Sigma}$$

Prod-Same, Lam-Same
$$\frac{\Pi \in \{\lambda, \forall\} \qquad \Sigma_0; \Gamma \vdash T_1 \approx_{\equiv} U_1 \rhd \Sigma_1 \qquad \Sigma_1; \Gamma, x : T_1 \vdash T_2 \approx_{\mathcal{R}} U_2 \rhd \Sigma_2}{\Sigma_0; \Gamma \vdash_0 \Pi x : T_1.\ T_2 \approx_{\mathcal{R}} \Pi x : U_1.\ U_2 \rhd \Sigma_2}$$

Let-Same
$$\frac{\Sigma_0; \Gamma \vdash T \approx_{\equiv} U \rhd \Sigma_1 \qquad \Sigma_1; \Gamma \vdash t_2 \approx_{\equiv} u_2 \rhd \Sigma_2 \qquad \Sigma_2; \Gamma, x := t_2 \vdash t_1 \approx_{\mathcal{R}} u_1 \rhd \Sigma_3}{\Sigma_0; \Gamma \vdash_0 \mathbf{let}\ x := t_2 : T\ \mathbf{in}\ t_1 \approx_{\mathcal{R}} \mathbf{let}\ x := u_2 : U\ \mathbf{in}\ u_1 \rhd \Sigma_3}$$

Rigid-Same
$$\frac{h \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{K} \qquad \mathcal{C}_1 = \mathcal{C}_0 \wedge \overline{\kappa = \kappa'} \qquad \mathcal{C}_1 \models}{(\overline{\ell} \models \mathcal{C}_0); \Sigma; \Gamma \vdash_0 h[\overline{\kappa}] \approx_{\mathcal{R}} h[\overline{\kappa'}] \rhd (\overline{\ell} \models \mathcal{C}_1), \Sigma}$$

Flexible-Same
$$\frac{h \in \mathcal{C} \qquad \Phi_0 \models \overline{\ell} = \overline{\kappa} \rhd \Phi_1}{\Phi_0; \Sigma; \Gamma \vdash_0 h[\overline{\ell}] \approx_{\mathcal{R}} h[\overline{\kappa}] \rhd \Phi_1, \Sigma}$$

Univ-Eq
$$\frac{\Phi \models i = j}{\Phi \models i = j \rhd \Phi}$$

Univ-Flexible
$$\frac{i_{\mathtt{f}} \vee j_{\mathtt{f}} \in \overline{\ell} \qquad \mathcal{C} \wedge i = j \models}{(\overline{\ell} \models \mathcal{C}) \models i = j \rhd (\ell \models \mathcal{C} \wedge i = j)}$$

Case-Same
$$\frac{\Sigma_0; \Gamma \vdash T \approx_{\equiv} U \rhd \Sigma_1 \qquad \Sigma_1; \Gamma \vdash t \approx_{\equiv} u \rhd \Sigma_2 \qquad \Sigma_2; \Gamma \vdash \overline{b} \approx_{\equiv} \overline{b'} \rhd \Sigma_3}{\Sigma_0; \Gamma \vdash_0 \mathbf{match}_T\ t\ \mathbf{with}\ \overline{b}\ \mathbf{end} \approx_{\mathcal{R}} \mathbf{match}_U\ u\ \mathbf{with}\ \overline{b'}\ \mathbf{end} \rhd \Sigma_3}$$

Fix-Same
$$\frac{\Sigma_0; \Gamma \vdash \overline{T} \approx_{\equiv} \overline{U} \rhd \Sigma_1 \qquad \Sigma_1; \Gamma \vdash \overline{t} \approx_{\equiv} \overline{u} \rhd \Sigma_2}{\Sigma_0; \Gamma \vdash_0 \mathbf{fix}_j\ \{\overline{x/n : T := t}\} \approx_{\mathcal{R}} \mathbf{fix}_j\ \{\overline{x/n : U := u}\} \rhd \Sigma_2}$$

Figure 3. Unifying terms sharing the same head constructor.

Let-Same compares first the type of the definitions, then the definitions themselves, and finally the body. In the last case, it augments the local context with the definition on the left (choosing the one on the left is an arbitrary choice, but after unification both definitions are convertible, *i.e.*, indistinguishable).

Rigid-Same equates the same variable, inductive type or constructor, enforcing that their universe instances are equal (note that the application of the rule will fail if these new constraints are inconsistent). The Flexible-Same rule unifies two instances of the same constant using a stronger condition on universe instances: they must unify according to the current constraints and by equating rigid universe variables with flexible variables only ($\Phi \models i = j$ *checks* if the constraint is already derivable). Otherwise we will backtrack on this rule to unfold the constant and unify the bodies (§4.2), which will generaly result in weaker, more general constraints to be enforced. The last two rules (Case-Same and Fix-Same) unify **match**es and **fix**points, respectively. In both cases we just unify pointwise every component of the term constructors.

4.2 Reduction

The previous subsection considered only the cases when both terms have no arguments and share the same constructor. If that is not the case, the algorithm first tries first-order approximation (rule App-FO in Figure 4). This rule, when considering two applications with the same number of arguments (n), compares the head element

$$\textsc{App-FO}\quad \frac{\Sigma_0;\Gamma \vdash_0 t \approx_{\mathcal{R}} u \rhd \Sigma_1 \qquad n \geq 0 \qquad \Sigma_1;\Gamma \vdash \overline{t_n} \approx_{\equiv} \overline{u_n} \rhd \Sigma_2}{\Sigma_0;\Gamma \vdash t\ \overline{t_n} \approx_{\mathcal{R}} u\ \overline{u_n} \rhd \Sigma_2}$$

$$\textsc{Meta-}\delta\textsc{R, Lam-}\beta\textsc{R, Let-}\zeta\textsc{R}\quad \frac{\Sigma;\Gamma \vdash u \overset{\text{w}}{\rightsquigarrow}_{\delta\Sigma,\beta,\zeta} u' \qquad \Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u' \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Meta-}\delta\textsc{L, Lam-}\beta\textsc{L, Let-}\zeta\textsc{L}\quad \frac{\Sigma;\Gamma \vdash t \overset{\text{w}}{\rightsquigarrow}_{\delta\Sigma,\beta,\zeta} t' \qquad \Sigma;\Gamma \vdash t' \approx_{\mathcal{R}} u \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Case-}\iota\textsc{R}\quad \frac{u \text{ is } \mathbf{fix} \text{ or } \mathbf{match} \qquad \Sigma;\Gamma \vdash u \downarrow^{\text{w}}_{\beta\zeta\delta\Sigma\iota\theta} u' \qquad u \neq u' \qquad \Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u' \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Case-}\iota\textsc{L}\quad \frac{t \text{ is } \mathbf{fix} \text{ or } \mathbf{match} \qquad \Sigma;\Gamma \vdash t \downarrow^{\text{w}}_{\beta\zeta\delta\Sigma\iota\theta} t' \qquad t \neq t' \qquad \Sigma;\Gamma \vdash t' \approx_{\mathcal{R}} u \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\textsc{NotStuckR}\quad \frac{\text{not } \Sigma;\Gamma \vdash \text{is\_stuck } u \qquad u \overset{\text{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \qquad \Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u' \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\textsc{StuckL}\quad \frac{\Sigma;\Gamma \vdash \text{is\_stuck } u \qquad t \overset{\text{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \qquad \Sigma;\Gamma \vdash t' \approx_{\mathcal{R}} u \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\textsc{R}\quad \frac{\Sigma;\Gamma \vdash u \overset{\text{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \qquad \Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u' \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\textsc{L}\quad \frac{\Sigma;\Gamma \vdash t \overset{\text{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \qquad \Sigma;\Gamma \vdash t' \approx_{\mathcal{R}} u \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} u \rhd \Sigma'}$$

$$\textsc{Lam-}\eta\textsc{R}\quad \frac{\begin{array}{c}u\text{'s head is not an abstraction} \\ \Sigma_0;\Gamma \vdash u : U \qquad \mathsf{ensure\_product}(\phi_0;\Sigma_0;\Gamma;T;U) = (\phi_1;\Sigma_1) \\ \phi_1;\Sigma_1;\Gamma,x:T \vdash u\ x \approx_{\equiv} t \rhd \phi_2;\Sigma_2\end{array}}{\phi_0;\Sigma_0;\Gamma \vdash u \approx_{\mathcal{R}} \lambda x:T.\ t \rhd \phi_2;\Sigma_2}$$

$$\textsc{Lam-}\eta\textsc{L}\quad \frac{\begin{array}{c}u\text{'s head is not an abstraction} \\ \Sigma_0;\Gamma \vdash u : U \qquad \mathsf{ensure\_product}(\phi_0;\Sigma_0;\Gamma;T;U) = (\phi_1;\Sigma_1) \\ \phi_1;\Sigma_1;\Gamma,x:T \vdash t \approx_{\equiv} u\ x \rhd \phi_2;\Sigma_2\end{array}}{\phi_0;\Sigma_0;\Gamma \vdash \lambda x:T.\ t \approx_{\mathcal{R}} u \rhd \phi_2;\Sigma_2}$$

Figure 4. Reduction steps attempted during unification.

(t and t', using only the rules in Figure 3), and then proceeds to unify each of the arguments. As customary, we denote multiple applications as a *spine* [6], using the form $t\ \overline{u_n}$ to represent the term $(\ldots(t\ u_1)\ \ldots\ u_n)$. We call t the *head* of the term.

If the rules in Figure 3 plus App-FO fail to apply, then the algorithm tries different reduction strategies. Except in some particular cases, the algorithm first tries reducing the right-hand side (rules ending with R) and then the left-hand side (rules ending with L). Except where noted, every L rule is just the mirror of the corresponding R rule, swapping the terms being unified in the conclusion and applications of unification in the premises. We will often omit the last letter (R or L), and simply write *e.g.*, Meta-δ when referring to both rules.

The algorithm first tries one step of either weak-head expansion of meta-variables ($\delta\Sigma$), weak-head β reduction, or weak-head **let**-expansion (ζ). These steps are described in rules Meta-δ, Lam-β,

$$\frac{t \downarrow^{\text{w}}_{\beta\zeta\delta\iota} k_j\ \overline{a}}{\mathbf{match}_T\ t\ \mathbf{with}\ \overline{k\ \overline{x} \Rightarrow t'}\ \mathbf{end} \rightsquigarrow_\theta\ \mathbf{match}_T\ k_j[\overline{\kappa}]\ \overline{a}\ \mathbf{with}\ \overline{k\ \overline{x} \Rightarrow t'}\ \mathbf{end}}$$

$$\frac{a_{n_j} \downarrow^{\text{w}}_{\beta\zeta\delta\iota} k\ \overline{b}}{\mathbf{fix}_j\ \{F\}\ a_1\ \ldots\ a_{n_j} \rightsquigarrow_\theta\ \mathbf{fix}_j\ \{F\}\ a_1\ \ldots\ a_{n_j-1}\ (k\ \overline{b})}$$

Figure 5. The θ-reduction strategy.

and Let-ζ. (Actually, as we are going to see in §4.5, the order of the rules is slightly different, although for the moment the implicit ordering obtained from the figure suffices.)

More interesting are the cases for δE, $\delta\Gamma$ and ι reductions. The high level idea is that special care should be taken when unfolding defined constants and variables. One reason is efficiency: we hope that, before performing the unfolding of a constant or variable, we will find the same constant or variable on the other side of the equation. The second reason is to avoid missing potential solutions, as already mentioned when introducing controlled backtracking in §2.

In the case of a **match** or a **fix** (rules Case-ι), we want to be able to reduce the scrutinee using all reduction rules, including δE and $\delta\Gamma$, and then (if applicable), continue reducing the corresponding branch of the **match** or the body of the **fix**, but avoiding δE and $\delta\Gamma$. We illustrate this desired behavior with a simple example using canonical structures. Consider the environment $E = \mathsf{d} := 0; \mathsf{c} := \mathsf{d}$, where there is also a structure with projector proj. Suppose further that there is a canonical instance i registered for proj and d. Then, the algorithm should succeed finding a solution for the following equation:

$$\mathbf{match}\ \mathsf{c}\ \mathbf{with}\ 0 \Rightarrow \mathsf{d} \mid \_ \Rightarrow 1\ \mathbf{end}\ \approx\ \mathsf{proj}\ ?f \tag{2}$$

where $?f$ is an unknown instance of the structure. More precisely, we expect the left-hand side to be reduced as

$$\mathsf{d}\ \approx\ \mathsf{proj}\ ?f$$

therefore enabling the use of the canonical instance i to solve $?f$.

This is done in the rule Case-ιL by weak-head normalizing the left-hand side using the standard $\beta\zeta\delta\Sigma\iota$ reduction rules plus a new reduction rule, θ, which weak-head normalizes scrutinees (Figure 5). Note that we really need this new reduction rule: we cannot consider weak-head reducing the term using δE, as it will destroy the constant d in the example above, nor restrict reduction of the scrutinee to not include δE, as it will be too restrictive (disallowing δE in the reduction on the l.h.s. makes Equation 2 not unifiable).

In Equation 2 we have a **match** on the l.h.s., and a constant on the r.h.s. (the projector). By giving priority to the ι reduction strategy over the δE one we can be sure that the projector will not get unfolded beforehand, and therefore the canonical instance resolution mechanism will work as expected. Different is the situation when we have constants on both sides of the equation. For instance, consider the following equation:

$$\mathsf{c}\ \approx\ \mathsf{proj}\ ?f \tag{3}$$

in the same context as before. Since there is no instance defined for c, we expect the algorithm to unfold it, uncovering the constant d. Then, it should solve the equation, as before, by instantiating $?f$ with i. If the projector is unfolded first instead, then the algorithm will not find the solution. The reason is that the projector unfolds to a case on the unknown $?f$:

$$\mathsf{c}\ \approx\ \mathbf{match}\ ?f\ \mathbf{with}\ \mathsf{Constr}\ a_1\ \ldots\ a_n \Rightarrow a_j\ \mathbf{end}$$

(Assuming the projector proj corresponds to the j-th field in the structure, and Constr is the constructor of the structure.) Now the canonical instance resolution will fail to see that the right-hand side is (was) a projector, so after unfolding c and d on the left, the algorithm will give up and fail.

In this case we cannot just simply rely on the ordering of rules, since that would make the algorithm sensitive to the position of the terms. In order to solve Equation 3 above, for instance, we need to prioritize reduction on the l.h.s. over the r.h.s., but this prioritization will have a negative impact on equations having the projector on the left instead of the right. The solution is to unfold a constant on the r.h.s. *only if the term does not "get stuck"*, that is, does not evaluate to certain values, like an irreducible **match**. More precisely, we define the concept of "being stuck" as

$$\text{is\_stuck } t = \quad \exists t'\ t''.\ t \rightsquigarrow^{0..1}_{\delta E,\delta\Gamma} t' \wedge t' \downarrow^{\mathrm{w}}_{\beta\zeta\iota\theta} t'' \text{ and the head of } t'' \text{ is a variable, case, fix, or abstraction}$$

That is, after performing an (optional) δE or $\delta\Gamma$ step and $\beta\zeta\iota\theta$-weak-head reducing the definition, the head element of the result is tested to be a **match**, **fix**, variable, or a λ-abstraction. Note that the reduction will effectively stop at the first head constant, without unfolding it further. This is important, for instance, when having a definition that reduces to a projector of a structure. If the projector is not exposed, and is instead reduced, then some canonical solution may be lost.

The rule Cons-δNotStuckR unfolds the right-hand side constant only if it will not get stuck. If it is stuck, then the rule Cons-δStuckL triggers and unfolds the left-hand side, which is precisely what happened in the example above. The rules Cons-δ are triggered as a last resort. This controlled unfolding of constants, together with canonical structures resolution, is what allows the encoding of sophisticated meta-programming idioms in [12].

When none of the rules above applies, the algorithm tries η-expansion (Lam-η rules). These rules unifies a function $\lambda x : T.\ t$ with a term u. The first premise ensures that u's head is not an abstraction to avoid overlapping with the Lam-Same rules, otherwise it is possible to build an infinite loop together with the rules Lam-β. The following two hypotheses ensure that u has product type with T as domain. First, the type of u is computed as U, and then we ensure U is a product with domain T calling the following function:

$$\begin{aligned}&\mathsf{ensure\_product}(\overline{\ell} \vDash \mathcal{C}; \Sigma_0; \Gamma; T; U) = (\phi_2; \Sigma_2)\\ &\quad \text{where } \phi_1 = \overline{\ell}, i \vDash \mathcal{C} \text{ for fresh universe level } i\\ &\quad \text{and } \Sigma_1 = \Sigma_0, ?v : \mathsf{Type}(i)[\Gamma, y : T] \text{ for fresh } ?v\\ &\quad \text{and } \phi_1; \Sigma_1; \Gamma \vdash U \approx_{\equiv} \forall y : T.\ ?v[\widehat{\Gamma}, y] \rhd \phi_2; \Sigma_2\end{aligned}$$

This function returns the result of unifying U with a product type with domain T and unknown range $?v$. For this, the meta-context Σ_0 is extended with $?v$ having type $\mathsf{Type}(i)$, for fresh universe level i, and context Γ extended with $y : T$.

We consider η-expansion as a last resort in the hope of obtaining a function beforehand, after expanding some definition.

4.3 Meta-Variable Instantiation

The rules for meta-variable instantiation are considered in Figure 6, most of which are inspired by Abel & Pientka [1]. There are, however, several differences between their work and ours, since we have a different base logic (CIC instead of LF), and a different assumption on the types of the terms: they require the terms being unified to have the same (known) type, while we do not (types play no directing role in our unification judgment).

For presentation purposes, we only present the rules having a meta-variable on the right-hand side of the equation, but the algorithm also includes the rules with the terms swapped.

Same Meta-Variable: If both terms are the same meta-variable $?x$, we have two distinct cases: if their substitution is exactly the same, the rule Meta-Same-Same applies, in which the arguments of the meta-variable are compared point-wise. Note that we require the elements in the substitution to be the same, and not just convertible. If, instead, their substitutions are different, the rule Meta-Same is attempted. To better understand this rule, let's look at an example. Suppose $?z$ has type $T[x_1 : \mathsf{nat}, x_2 : \mathsf{nat}]$ and we have to solve the equation

$$?z[y_1, y_2] \ \approx\ ?z[y_1, y_3]$$

where y_1, y_2 and y_3 are distinct variables. From this equation we cannot know yet what value $?z$ will hold, but at least we know it cannot refer (up to conversion) to the second parameter, x_2, since then the above equation would have no solution. This reasoning is reflected in the rule Meta-Same in the hypothesis

$$\Psi_1 \vdash \sigma \cap \sigma' \rhd \Psi_2$$

This judgment performs an intersection of both substitutions, filtering out those positions from the context of the meta-variable Ψ_1 where the substitutions disagree, resulting in Ψ_2.

The intersection judgment is defined in the Intersec-* rules in the same figure. The text inside grey boxes defines a different rule: it allows us to collapse a rule for declarations and a rule for definitions into one only rule. The judgment is conservative: it only filters out different *variables*. The judgment is undefined if the two substitutions have different *terms* (not variables) in some position. Of course, a more aggressive approach is possible, checking for convertibility of the terms instead of just syntactic equality, but it is not clear whether the few more cases covered compensates for the potential loss in performance.

Coming back to the rule Meta-Same, by filtering out the disagreeing positions of the substitution, we obtain a new context Ψ_2, which is a subset of Ψ_1. Since this smaller context may be ill-formed, we *sanitize* it. The sanitization judgment is defined at the bottom of the figure, and it simply removes every (possibly defined) variable in the context whose free variables are not included in the context. This process results in a new (possibly smaller) context Ψ_3. After making sure that the type T of $?x$ is still well-formed in this context, we restrict $?x$ to only refer to the variables in Ψ_3. We do this by creating a new meta-variable $?y$ with the type of $?x$, but in the context Ψ_3. We further instantiate $?x$ with $?y$. Both the creation of $?y$ and the instantiation of $?x$ in the meta-context Σ is expressed in the fragment $\Sigma \cup \{?y : T[\Psi_3], ?x := ?y[\widehat{\Psi_3}]\}$ of the last hypothesis. We use this new meta-context to compare point-wise the arguments of the meta-variable.

Meta-Variable Instantiation: The Meta-Inst rules instantiate a meta-variable applying a variation of *higher-order pattern unification* (HOPU) [14]. They unify a meta-variable $?x$ with some term t, obtaining a MGU (actually, as we will see in §4.3.1, *almost* a MGU). As required by HOPU, the meta-variable is applied to a suspended substitution mapping variables to variables, ξ, and a spine of arguments ξ', of variables only. Assuming $?x$ has (contextual) type $T[\Psi]$, this rule must find a term t''' to instantiate $?x$ such that

$$t \ \approx\ ?x[\xi]\ \xi'$$

that is, after performing the suspended substitution ξ and applying arguments ξ' (formally, $t'''\{\xi/\hat{\Psi}\}\ \xi'$), results in a term convertible to t.

Having contexts Σ_0 and Γ, the new term t''' is crafted from t following these steps:

$$\textsc{Meta-Same-Same}\ \frac{\Sigma;\Gamma \vdash \overline{t} \approx_{\equiv} \overline{u} \rhd \Sigma'}{\Sigma;\Gamma \vdash ?x[\sigma]\ \overline{t} \approx_{\mathcal{R}} ?x[\sigma]\ \overline{u} \rhd \Sigma'}$$

$$\textsc{Meta-Same}\ \frac{?x : T[\Psi_1] \in \Sigma \qquad \Psi_1 \vdash \sigma \cap \sigma' \rhd \Psi_2 \qquad \cdot \vdash \mathsf{sanitize}(\Psi_2) \rhd \Psi_3 \qquad \mathsf{FV}(T) \subseteq \Psi_3 \qquad \Sigma \cup \{?y : T[\Psi_3], ?x := ?y[\widehat{\Psi_3}]\};\Gamma \vdash \overline{t} \approx_{\equiv} \overline{u} \rhd \Sigma'}{\Sigma;\Gamma \vdash ?x[\sigma]\ \overline{t} \approx_{\mathcal{R}} ?x[\sigma']\ \overline{u} \rhd \Sigma'}$$

$$\textsc{Meta-InstR}\ \frac{?x : T[\Psi] \in \Sigma_0 \qquad t', \xi_1 = \mathsf{remove\_tail}(t;\xi') \qquad t' \downarrow^{\mathrm{w}}_{\beta} t'' \qquad \Sigma_0 \vdash \mathsf{prune}(?x;\xi,\xi_1;t'') \rhd \Sigma_1 \qquad \Sigma_1;\Gamma \vdash \xi_1 : \overline{U} \qquad t''' = \lambda y : U\{\xi,\xi_1/\hat{\Psi},\overline{y}\}^{-1}.\ \Sigma_1(t'')\{\xi,\xi_1/\hat{\Psi},\overline{y}\}^{-1} \qquad \Sigma_1;\Psi \vdash t''' : T' \qquad \Sigma_1;\Psi \vdash T' \approx_{\leq} T \rhd \Sigma_2 \qquad ?x \notin \mathsf{FMV}(t''')}{\Sigma_0;\Gamma \vdash t \approx_{\mathcal{R}} ?x[\xi]\ \xi' \rhd \Sigma_2 \cup \{?x := t'''\}}$$

$$\textsc{Meta-For}\ \frac{?x : T[\Psi] \in \Sigma_0 \qquad 0 < n \qquad \Sigma_0;\Gamma \vdash u\ \overline{u'_m} \approx_{\equiv} ?x[\sigma] \rhd \Sigma_1 \qquad \Sigma_1;\Gamma \vdash \overline{u''_n} \approx_{\equiv} \overline{t_n} \rhd \Sigma_2}{\Sigma_0;\Gamma \vdash u\ \overline{u'_m u''_n} \approx_{\mathcal{R}} ?x[\sigma]\ \overline{t_n} \rhd \Sigma_2}$$

$$\textsc{Meta-DelDepsR}\ \frac{?x : T[\Psi] \in \Sigma \qquad l = [i | \sigma_i \text{ is variable and } \nexists j > i.\ \sigma_i = (\sigma,\overline{u})_j] \qquad \cdot \vdash \mathsf{sanitize}(\Psi_l) \rhd \Psi' \qquad \mathsf{FV}(T) \subseteq \Psi' \qquad \Sigma \cup \{?y : T[\Psi'], ?x := ?y[\widehat{\Psi'}]\};\Gamma \vdash t \approx_{\mathcal{R}} ?y[\sigma_l]\ \overline{u} \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx_{\mathcal{R}} ?x[\sigma]\ \overline{u} \rhd \Sigma'}$$

$$\textsc{Meta-ReduceR}\ \frac{?u : T[\Psi] \in \Sigma_0 \qquad t \overset{\mathrm{w}\ 0..1}{\rightsquigarrow_{\delta}} t' \qquad t' \downarrow^{\mathrm{w}}_{\beta\zeta\iota\theta} t'' \qquad \Sigma_0;\Gamma \vdash t'' \approx_{\mathcal{R}} ?u[\sigma]\ \overline{t_n} \rhd \Sigma_1}{\Sigma_0;\Gamma \vdash t \approx_{\mathcal{R}} ?u[\sigma]\ \overline{t_n} \rhd \Sigma_1}$$

$$\textsc{Intersec-Nil}\ \frac{}{\cdot \vdash \cdot \cap \cdot \rhd \cdot}$$

$$\textsc{Intersec-Keep}\ \frac{\Psi \vdash \sigma \cap \sigma' \rhd \Psi'}{\Psi, x := u : A \vdash \sigma, t \cap \sigma', t \rhd \Psi', x : A}$$

$$\textsc{Intersec-Remove}\ \frac{\Psi \vdash \sigma \cap \sigma' \rhd \Psi' \qquad y \neq z}{\Psi, x := u : T \vdash \sigma, y \cap \sigma', z \rhd \Psi'}$$

$$\textsc{Sanitize-Nil}\ \frac{}{\xi \vdash \mathsf{sanitize}(\cdot) \rhd \cdot}$$

$$\textsc{Sanitize-Keep}\ \frac{\mathsf{FV}(T) \subseteq \xi \qquad \mathsf{FV}(u) \subseteq \xi \qquad x, \xi \vdash \mathsf{sanitize}(\Psi) \rhd \Psi'}{\xi \vdash \mathsf{sanitize}(x := u : T, \Psi) \rhd x : T, \Psi'}$$

$$\textsc{Sanitize-Remove}\ \frac{\mathsf{FV}(T) \not\subseteq \xi \vee \mathsf{FV}(u) \not\subseteq \xi \qquad \xi \vdash \mathsf{sanitize}(\Psi) \rhd \Psi'}{\xi \vdash \mathsf{sanitize}(x := u : T, \Psi) \rhd \Psi'}$$

Figure 6. Meta-variable instantiation.

1. To avoid unnecessarily η-expanded solutions, the term t and arguments ξ' are decomposed using the function $\mathsf{remove\_tail}(\cdot;\cdot)$:

$$\begin{aligned} \mathsf{remove\_tail}(t\ x;\xi,x) &= \mathsf{remove\_tail}(t;\xi) && \text{if } x \notin \mathsf{FV}(t) \\ \mathsf{remove\_tail}(t;\xi) &= (t,\xi) && \text{in any other case} \end{aligned}$$

   This function, applied to t and ξ', returns a new term t' and a list of variables ξ_1, where there exists ξ_2 such that $t = t'\ \xi_2$ and $\xi' = \xi_1, \xi_2$, and ξ_2 is the longest such list. For instance, in the following example

$$?f[]\ x\ y\ \approx\ \mathsf{addn}\ x\ y$$

   where addn is the addition operation on natural numbers, we want to remove "the tail" on both sides of the equation, leading to the natural solution $?f[] := \mathsf{addn}$. In this example, ξ_1 is the empty list, ξ_2 is $[x, y]$, and t' is addn.

   The check that $x \notin \mathsf{FV}(t)$ in the first case above ensures that no solutions are erroneously discarded. Consider the following equation:

$$?f[]\ x\ \approx\ \mathsf{addn0}\ x\ x$$

   If we remove the argument of the meta-variable, we will end up with the unsolvable equation $?f[]\ \approx\ \mathsf{addn0}\ x$.

2. The term obtained in the previous step is weak head β normalized, noted $t' \downarrow^{\mathrm{w}}_{\beta} t''$. This is performed in order to remove false dependencies, like variable x in $(\lambda y.\ 0)\ x$.

3. The meta-variables in t'' are *pruned*. This process is quite involved, and detailed examples can be found in [1]. The formal description will be discussed below in §4.3.1.

   At high level, the pruning judgment ensures that the term t'' has no "offending variables", that is, free variables outside of those occurring in the substitution ξ, ξ_1. It does so by removing elements from the suspended substitutions occurring in t'', containing variables outside of ξ, ξ_1. For instance, in the example $?f[]\ x\ \approx\ \mathsf{addn0}\ ?u[x, y]$, the variable y has to be removed from the substitution on the r.h.s. since it does not occur in the l.h.s.. Similarly, if the meta-variable being instantiated occurs inside a suspended substitution, it has to be removed from the substitution to avoid a circularity in the instantiation. The output of this judgment is a new meta-context Σ_1.

4. The final term t''' is constructed as

$$\overline{\lambda y : U\{\xi,\xi_1/\hat{\Psi},\overline{y}\}^{-1}.\ \Sigma_1(t'')\{\xi,\xi_1/\hat{\Psi},\overline{y}\}^{-1}}$$

   First, note that t''' has to be a function taking n arguments $\overline{y}$, where $n = |\xi_1|$. For the moment, let's forget about the types of each y_j.

   The body of this function is the term obtained from the second step, t'', after performing a few changes. First, all of its defined meta-variables are normalized with respect to the meta-context obtained in the previous step, Σ_1, in order to replace the meta-variables with the pruned ones. This step effectively removes false dependencies on variables not occurring in ξ, ξ_1.

Then, the *inversion* of substitution $\xi, \xi_1/\hat{\Psi}, \overline{y}$ is performed. This inversion ensures that all free variables in $\Sigma_1(t'')$ are replaced by variables in Ψ and $\overline{y}$. More precisely, it replaces every variable in $\Sigma_1(t'')$ appearing *only once* in the image of the substitution (ξ, ξ_1) by the corresponding variable in the domain of the substitution $(\hat{\Psi}, \overline{y})$. If a variable appears multiple times in the image and occur in term t'', then inversion fails.

The type of each argument y_j of the function is the type U_j, obtained from the j-th element in ξ_1, after performing the inversion substitution (with the caveat that the substitution includes only the $j-1$ elements in $\overline{y}$).

5. The type of t''', which now only depends on the context Ψ, is computed as T', and unified with the type of $?x$, obtaining a new meta-context Σ_2.

   In the special case where t''' is itself a meta-variable of type an arity (an n-ary dependent product whose codomain is a sort), we do not directly force the type of the instance T' to be smaller than T, which would unnecessarily restrict the universe graph. Instead, we downcast T and T' to a smaller type according to the cumulativity relation before converting them. The idea is that, if we are unifying meta-variables $?x$ and $?y$, with $?x : \mathsf{Type}(i)[\Gamma]$ and $?y : \mathsf{Type}(j)[\Gamma']$, the body of $?x$ and $?y$ just has to be of type $\mathsf{Type}(k)$ for some $k \leq i, j$.

6. Finally, an *occurs check* is performed to prevent illegal solutions, making sure $?x$ does not occur in t'''.

The algorithm outputs Σ_2 plus the instantiation of $?x$ with t'''.

First-Order Approximation: The rules META-INST only applies if the spine of arguments of the meta-variable only have variables. This can be quite restrictive. Consider for instance the following equation that tries to unify an unknown function, applied to an unknown argument, with the term 1 (expanded to $\mathsf{S}\ 0$):

$$\mathsf{S}\ 0 \ \approx\ ?f[]\ ?y[]$$

As usual, such equations have multiple solutions, but there is one that is "more natural": assign S to $?f$ and 0 to $?y$. However, since the argument to the meta-variable is not a variable, it does not comply with HOPU, and therefore is not considered by the META-INST rules. In an scenario like this, the META-FO rules perform a *first order approximation*, unifying the meta-variable ($?f$ in the equation above) with the term on the l.h.s. without the last n arguments (S), which are in turn unified pointwise with the n arguments in the spine of the meta-variable (0 and $?y$, respectively). Note that the rule APP-FO does not subsume this rule, as it requires both terms being equated to have the same number of arguments.

Meta-Variable Dependencies Erasure: If none of the rules above work, the algorithm makes a somewhat brutal attempt: the rule META-DELDEPSR chops off every element in the substitution that is not a variable, or that is a duplicated variable. Therefore, problems not complying with HOPU can be reconsidered. One issue with this rule is that it fixes a solution where many solutions may exist. Although the selected solution works most of the time, as we are going to see in §5, it might not be the one expected by the user.

Formally, this rule first takes each position i in σ such that σ_i is a variable with no duplicated occurrence in $\sigma, \overline{u}$. The resulting list l containing those positions is used to filter out the local context of the meta-variable, obtaining the new context Ψ'. After sanitizing this context, a fresh meta-variable $?y$ is created in this restricted local context, and $?x$ is instantiated with this meta-variable. The new meta-context obtained after this instantiation is used to recursively call the unification algorithm to solve the problem $?y[\sigma_l]\ \overline{u} \ \approx\ t$.

Eliminating Dependencies via Reduction: Sometimes the term being assigned to the meta-variable has variables not occurring in the substitution, but that can be eliminated via reduction. For instance, take the following equation

$$\pi_1(0, x) \ \approx\ ?g[]$$

It has a solution, after reducing the term on the l.h.s., obtaining the easily solvable equation $0 \ \approx\ ?g[]$. This is precisely what rules META-REDUCE do, as a last attempt to make progress.

4.3.1 Pruning

Figure 7 shows the actual process of pruning. The pruning judgment is noted

$$\Sigma \vdash \mathsf{prune}(?x; \xi; t) \rhd \Sigma'$$

It takes a meta-context Σ, a meta-variable $?x$, a list of variables ξ, the term to be pruned t, and returns a new meta-context Σ', which is an extension of Σ where all the meta-variables with offending variables in their suspended substitution are instantiated with pruned ones.

For brevity, we only show rules for the Calculus of Constructions fragment of CIC, *i.e.*, without considering pattern matching and fixpoints. The missing rules are easy to extrapolate from the given ones. The only interesting case is when the term t is a meta-variable $?y$ applied to the suspended substitution σ. We have two possibilities: either every variable from every term in σ is included in ξ, in which case we do not need to prune (PRUNE-META-NOPRUNE), or there exists some terms which have to be removed (pruned) from σ (PRUNE-META).

These two rules use an auxiliary judgment to prune the local context of the meta-variable Ψ_0. This judgment has the form

$$\Psi \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma) \rhd \Psi'$$

Basically, it filters out every variable in Ψ where σ has an *offending term*, that is, a term with a free variable not in ξ, or having $?x$ in the set of free meta-variables. Ψ' is the result of this process.

Coming back to the rules in Figure 7, in PRUNE-META-NOPRUNE we have the condition that the pruning of context Ψ_0 resulted in the same context (no need for a change). More interestingly, when the pruning of Ψ_0 results in a new context Ψ_1, PRUNE-META does the actual pruning of $?y$. Similarly to the rule META-SAME, it first sanitizes the new context Ψ_1, obtaining a new context Ψ_2, then it ensures that the type T is valid in Ψ_2, by pruning variables outside Ψ_2, and finally instantiates the meta-variable $?y$ with a fresh meta-variable $?z$, having contextual type $T[\Psi_2]$.

It is important to note that, due to conversion, the process of pruning may loose solutions. For instance, consider the following equation:

$$\pi_1(0, ?x[n]) \ \approx\ ?y[]$$

The pruning algorithm will remove n from $?x$, although another solution exists by reducing the l.h.s., assigning 0 to $?y$.

4.4 Canonical Structures Resolution

When an instance i of a structure is declared **Canonical**, COQ will add, for each projector, a record in the *canonical structures database* (Δ_{db}). Each record registers a key consisting of the projector p and the head constructor h of the value for that projector in the instance, and a value, the instance i itself. Then, at high level, when the algorithm has to solve an equation of the form $h\ t \ \approx\ p\ ?x$, it searches for the key (p, h) in the database, finding that $?x$ should be instantiated with i.

The process is formally described in Figure 8. We always start from an equation of the form:

$$t'' \ \approx\ p_j[\overline{\kappa}]\ \overline{p}\ i\ \overline{t}$$

where p_j is a projector of a structure applied to some universe instance $\overline{\kappa}$, $\overline{p}$ are the parameters of the structure, i is the instance

$$\textsc{Prune-Constant}\ \frac{h \in \mathcal{S} \cup \mathcal{C}}{\Sigma \vdash \mathsf{prune}(?x; \xi; h) \rhd \Sigma} \qquad \textsc{Prune-Var}\ \frac{x \in \xi}{\Sigma \vdash \mathsf{prune}(?x; \xi; x) \rhd \Sigma}$$

$$\textsc{Prune-Lam, Prune-Prod}\ \frac{\Pi \in \{\lambda, \forall\} \qquad \Sigma \vdash \mathsf{prune}(?x; \xi, x; t) \rhd \Sigma'}{\Sigma \vdash \mathsf{prune}(?x; \xi; \Pi x.\ t) \rhd \Sigma'}$$

$$\textsc{Prune-Let}\ \frac{\Sigma_0 \vdash \mathsf{prune}(?x; \xi; t_2) \rhd \Sigma_1 \quad \Sigma_1 \vdash \mathsf{prune}(?x; \xi, x; t_1) \rhd \Sigma_2}{\Sigma_0 \vdash \mathsf{prune}(?x; \xi; \mathsf{let}\ x := t_2\ \mathsf{in}\ t_1) \rhd \Sigma_2}$$

$$\textsc{Prune-App}\ \frac{\Sigma_0 \vdash \mathsf{prune}(?x; \xi; t) \rhd \Sigma_1 \quad \Sigma_i \vdash \mathsf{prune}(?x; \xi; t_i) \rhd \Sigma_{i+1} \qquad i \in [1, n]}{\Sigma_0 \vdash \mathsf{prune}(?x; \xi; t\ \overline{t_n}) \rhd \Sigma_{n+1}}$$

$$\textsc{Prune-Meta-NoPrune}\ \frac{?y : T[\Psi_0] \in \Sigma \qquad ?x \neq ?y \quad \Psi_0 \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma) \rhd \Psi_0}{\Sigma \vdash \mathsf{prune}(?x; \xi; ?y[\sigma]) \rhd \Sigma}$$

$$\textsc{Prune-Meta}\ \frac{?y : T[\Psi_0] \in \Sigma \qquad ?x \neq ?y \qquad \Psi_0 \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma) \rhd \Psi_1 \quad \cdot \vdash \mathsf{sanitize}(\Psi_1) \rhd \Psi_2 \qquad \Sigma \vdash \mathsf{prune}(?x; \widehat{\Psi_2}; T) \rhd \Sigma'}{\Sigma \vdash \mathsf{prune}(?x; \xi; ?y[\sigma]) \rhd \Sigma', ?z : T[\Psi_2] \cup \{?y := ?z[\widehat{\Psi_2}]\}}$$

$$\textsc{PruneCtx-Nil}\ \frac{}{\cdot \vdash \mathsf{prune\_ctx}(?x; \xi; \cdot) \rhd \cdot}$$

$$\textsc{PruneCtx-NoPrune}\ \frac{\mathsf{FV}(t) \subseteq \xi \qquad ?x \notin \mathsf{FMV}(t) \qquad \Psi \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma) \rhd \Psi'}{\Psi, z : A \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma, t) \rhd \Psi', z : A}$$

$$\textsc{PruneCtx-Prune}\ \frac{\mathsf{FV}(t) \not\subseteq \xi \vee ?x \in \mathsf{FMV}(t) \qquad \Psi \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma) \rhd \Psi'}{\Psi, x : A \vdash \mathsf{prune\_ctx}(?x; \xi; \sigma, t) \rhd \Psi'}$$

Figure 7. Pruning of meta-variables.

$$\textsc{Lookup-CS}\ \frac{\begin{array}{c}(p_j, h, c_\iota) \in \Delta_{\mathrm{db}} \\ \Phi_1, \iota = \mathsf{fresh}(\Phi_0, c_\iota) \qquad \iota \rightsquigarrow_{\delta E} \lambda \overline{x : T}.\ k[\overline{\kappa'}]\ \overline{p'}\ \overline{v} \\ \Sigma_1 = \Sigma_0, \overline{?y : T} \qquad \Phi_1 \models \overline{\kappa} = \overline{\kappa'} \rhd \Phi_2 \\ \Phi_2; \Sigma_1; \Gamma \vdash \overline{p} \approx_{\equiv} \overline{p'\{?y/x\}} \rhd \Phi_3; \Sigma_2\end{array}}{\Phi_0; \Sigma_0 \vdash (p_j, \overline{\kappa}, \overline{p}, h) \in ?\ \Delta_{\mathrm{db}} \rhd \Phi_3, \Sigma_2, \iota\ \overline{?y}, v_j\{\overline{?y}/\overline{x}\}}$$

$$\textsc{CS-ConstR}\ \frac{\begin{array}{c}\Phi_0; \Sigma_0 \vdash (p_j, \overline{\kappa}, \overline{p}, c) \in ?\ \Delta_{\mathrm{db}} \rhd \Phi_1, \Sigma_1, \iota, c[\overline{\ell'}]\ \overline{u'} \\ \Phi_1 \models \overline{\ell} = \overline{\ell'} \rhd \Phi_2 \qquad \Phi_2; \Sigma_1; \Gamma \vdash \overline{u} \approx_{\equiv} \overline{u'} \rhd \Phi_3; \Sigma_2 \\ \Phi_3; \Sigma_2; \Gamma \vdash i \approx_{\equiv} \iota \rhd \Phi_4; \Sigma_3 \\ \Phi_4; \Sigma_4; \Gamma \vdash \overline{t'} \approx_{\equiv} \overline{t} \rhd \Phi_5; \Sigma_4\end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash c[\overline{\ell}]\ \overline{u}\ \overline{t'} \approx_{\mathcal{R}} p_j[\overline{\kappa}]\ \overline{p}\ i\ \overline{t} \rhd \Phi_5; \Sigma_4}$$

$$\textsc{CS-ProdR}\ \frac{\begin{array}{c}\Phi_0; \Sigma_0 \vdash (p_j, \overline{\kappa}, \overline{p}, \to) \in ?\ \Delta_{\mathrm{db}} \rhd \Phi_1, \Sigma_1, \iota, u \to u' \\ \Phi_1; \Sigma_1; \Gamma \vdash t \approx_{\equiv} u \rhd \Phi_2; \Sigma_2 \\ \Phi_2; \Sigma_2; \Gamma \vdash t' \approx_{\mathcal{R}} u' \rhd \Phi_3; \Sigma_3 \\ \Phi_3; \Sigma_3; \Gamma \vdash i \approx_{\equiv} \iota \rhd \Phi_4; \Sigma_4\end{array}}{\Phi_0, \Sigma_0; \Gamma \vdash t \to t' \approx_{\mathcal{R}} p_j[\overline{\kappa}]\ \overline{p}\ i \rhd \Phi_4; \Sigma_4}$$

$$\textsc{CS-SortR}\ \frac{\begin{array}{c}\Phi_0; \Sigma_0 \vdash (p_j, \overline{\kappa}, \overline{p}, s) \in ?\ \Delta_{\mathrm{db}} \rhd \Phi_1, \Sigma_1, \iota, v_j \\ \Phi_1; \Sigma_1; \Gamma \vdash s \approx_{\mathcal{R}} v_j \rhd \Phi_2; \Sigma_2 \\ \Phi_2; \Sigma_2; \Gamma \vdash i \approx_{\equiv} \iota \rhd \Phi_3; \Sigma_3\end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash s \approx_{\mathcal{R}} p_j[\overline{\kappa}]\ \overline{p}\ i \rhd \Phi_3; \Sigma_3}$$

$$\textsc{CS-DefaultR}\ \frac{\begin{array}{c}\Phi_0; \Sigma_0 \vdash (p_j, \overline{\kappa}, \overline{p}, \_) \in ?\ \Delta_{\mathrm{db}} \rhd \Phi_1, \Sigma_1, \iota, v_j \\ \Phi_3; \Sigma_2; \Gamma \vdash t \approx_{\mathcal{R}} v_j \rhd \Phi_4; \Sigma_3 \\ \Phi_4; \Sigma_3; \Gamma \vdash i \approx_{\equiv} \iota \rhd \Phi_5; \Sigma_4\end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash t \approx_{\mathcal{R}} p_j[\overline{\kappa}]\ \overline{p}\ i \rhd \Phi_5; \Sigma_4}$$

Figure 8. Canonical structures resolution.

(usually a meta-variable), and $\overline{t}$ are the arguments of the projected value, in the case when it has product type. In order to solve this equation the algorithm proceeds as follows:

1. First, a constant c_ι is selected from Δ_{db}, keying on the projector p_j and the head element h of t''. The constant c_ι stored in the database is a potentially polymorphic constant, so we build a fresh instance of it (ι) and add the fresh universe levels and constraints to the universe context. Its body is a function taking arguments $\overline{x : T}$ and returning the term $k[\overline{\kappa'}]\ \overline{p'}\ \overline{v}$, with k the constructor of the structure applied to a universe instance $\overline{\kappa'}$, $\overline{p'}$ the parameters of the structure, and $\overline{v}$ the values for each of the fields of the structure.
2. Then, the expected and inferred universe instances and parameters of the instance are unified, after replacing every argument x with a corresponding fresh meta-variable $?y$.
3. According to the class of h, the algorithm considers different rules:
   (a) CS-Const if h is a constant c.
   (b) CS-Prod if h is a non-dependent product $t \to t'$.
   (c) CS-Sort if h is a sort s.

   If these do not apply, then it tries CS-Default.
4. Next, the term t'' is unified with the corresponding projected term in the value of the instance for the j-th field. If t'' is a constant c applied to arguments $\overline{u}$ and the value v_j of the j-th field of ι is c applied to $\overline{u'}$, then the universe instances and arguments $\overline{u}$ and $\overline{u'}$ are unified. If t'' is a product with premise t and conclusion t', they are unified with the corresponding terms (u and u') in v_j. Note that cumulativity is preserved in the codomain of products and in the CS-Sort rule.
5. The instance of the structure i is unified with the instance found in the database, ι, applied to the meta-variables $\overline{y}$. Typically, i is a meta-variable, and this step results in instantiating the meta-variable with the constructed instance.
6. Finally, for CS-Const only, if the j-th field of the structure has product type, and is applied to $\overline{t'}$ arguments, then these arguments are unified with the arguments $\overline{t}$ of the projector.

As with the rules for meta-variable instantiation, we only show the rules in one direction, with the projector on the right-hand side, but the algorithm also includes the rules in the opposite direction.

4.5 Rule Priorities and Backtracking

The figures shown above does not precisely nail the priority of the rules, nor when the algorithm backtracks. Below we show the precise order of application of the rules, where the rules in the same line are tried in the given order *without* backtracking (the first one matching the conclusion and whose side-conditions are satisfied is used). Rules in different lines or in the same line separated by | are tried *with* backtracking (if one fails to apply, the next is tried). Note that if at any point the environment and the two terms to be unified are ground (they do not contain meta-variables), unification is skipped entirely and a call to Coq's efficient conversion algorithm is made instead.

1. If a term has a *defined* meta-variable in its head position:
   (a) Meta-δR, Meta-δL
2. If both terms heads are *the same undefined* meta-variable:
   (a) Meta-Same-Same, Meta-Same
3. If one term has an *undefined* meta-variable, and the other term does not have the same meta-variable in its head position:
   Meta-InstR | Meta-FOR | Meta-ReduceR | Meta-DelDepsR | Lam-ηR | Meta-InstL | Meta-FOL | Meta-ReduceL | Meta-DelDepsL | Lam-ηL
4. Else:
   (a) If the two terms have different head constants:
      i. (CS-ConstR, CS-ProdR, CS-SortR) | CS-DefaultR
      ii. (CS-ConstL, CS-ProdL, CS-SortL) | CS-DefaultL
   (b) App-FO
   (c) The remaining rules from Figure 4 in the following order, backtracking only if the hypotheses that are not recursive calls to the algorithm fail to apply:
   Lam-βR | Let-ζR | Case-ιR | Lam-βL | Let-ζL | Case-ιL | Cons-δNotStuckR | Cons-δStuckL | Cons-δR | Cons-δL | Lam-ηR | Lam-ηL

4.6 A Deliberate Omission: Constraint Postponement

The technique of *constraint postponement* [18] is widely adopted in unification algorithms, including the current algorithm of Coq. It has however some negative impact in Coq, and, as it turns out, it is not as crucial as generally believed.

First, let us show why this technique is incorporated into proof assistants. Sometimes the unification algorithm is faced with an equation that has multiple solutions, in a context where there should only be one possible candidate. For instance, consider the following term witnessing an existential quantification:

$$\mathsf{exist}\ \_\ 0\ (\mathsf{le\_n}\ 0) : \exists x.\ x \leq x$$

where exist is the constructor of the type $\exists x.\ P\,x$, with P a predicate over the (implicit) type of x. More precisely, exist takes a predicate P, an element x, and a proof that P holds for x, that is, $P\,x$. In the example above we are providing an underscore in place of P, since we want Coq to find out the predicate, and we annotate the term with a typing constraint (after the colon) to specify that the whole term is a proof of existence of a number lesser or equal to itself. In this case, we provide 0 as such number, and the proof le_n 0, which has type $0 \leq 0$.

During typechecking, Coq first infers the type of the term on the left of the colon, and only then it verifies that this type is compatible (*i.e.*, unifiable) with the typing constraint. When inferring the type for the term on the left, Coq will create a fresh meta-variable for the predicate P, let's call it $?P$, and unify $?P\ 0$ with $0 \leq 0$, the type of le_n 0. Without any further information, Coq has four different (incomparable) solutions for P: $\lambda x.\ 0 \leq 0, \lambda x.\ x \leq 0, \lambda x.\ 0 \leq x, \lambda x.\ x \leq x$.

When faced with such an ambiguity, Coq postpones the equation in the hope that further information will help disambiguate the problem. In this case, the necessary information is given later on through the typing constraint, which narrows the set of solutions to a unique solution.

Constraint postponment has its consequences, though: On one hand, the algorithm can solve more unification problems and hence fewer typing annotations are required (*e.g.*, we do not need to specify P). On the other hand, since constraints are delayed, the algorithm becomes hard to debug and, at times, slow. The reason for these assertions comes from the realisation that the algorithm will continue to (try to) unify the terms, piling up constraints on the way, perhaps to later on find out that, after all, the terms are not unifiable (or are unifiable only if some decision is taken on the delayed equations).

When combined with canonical structures resolution, or any other form of proof automation, this technique is particularly bad, as it may break the assumption that certain value has been previously assigned. The motivation to omit this technique came from experience in projects on proof automation by the first author [12, 25], and on bi-directional elaboration by the second author (in the above example, a bi-directional elaboration algorithm will unify the type returned by exist with the expected type, and only then unify the type of its arguments, thereby posing the unification problems in the right order).

Our results (§5) show that this technique is not crucial.

4.7 Correctness

Our algorithm should satisfy the following correctness criterion: if two well-typed terms t_1 and t_2 unify under universe context Φ and meta-context Σ, resulting in a new universe Φ' and meta-context Σ', both terms should also be well typed under Σ'. Moreover, both terms should be convertible (or in the cumulativity relation) under Φ', Σ'.

However, this is false—for both the current algorithm implemented in Coq, and the one described here. The culprit is the syntactic check required at typechecking to ensure termination of fixpoints, the guard condition. Indeed, it is easy to make unification instantiate a meta-variable with a term containing a non-structurally-recursive call to a recursive function in its context, resulting in an ill-typed term. Hence, we must weaken this conjecture to use a weaker notion of typing, as in Coen's thesis [19].

For the moment we lack a correctness proof. This work sets the first stone presenting a specification faithful to an implementation that performs well on a variety of large examples (§5). We anticipate that the proof will be simpler than for existing algorithms, notably due to the lack of postponment which usually complicates the argument of type preservation.

5. Evaluation of the Algorithm

Since, as we saw in §4.6, our algorithm does not incorporate certain heuristics, it is reasonable to expect that it will fail to solve several unification problems appearing in existing libraries. To test our algorithm "in the wild" we developed a plugin called UniCoq[3], which, when requested, changes the current unification algorithm

[3] Sources can be downloaded from http://github.com/unicoq.

of Coq with ours. With this plugin, we compiled four different libraries, and evaluated the number of lines that required changes. These changes may be necessary either because UniCoq found a different solution from the expected one, or because it found no solution at all. As it turns out, UniCoq solved most of the problems it encountered.

The first set of files we considered is the standard library of Coq. With UniCoq, it compiles almost out of the box, with only a few lines requiring extra typing annotations. We believe the reason for such success is that most of the files in the library are several years old, and were conceived in older versions of Coq, when it had a much simpler unification algorithm.

The second set of files come from Adam Chlipala's book "Certified Programming with Dependent Types" (CPDT) [7]. This book provides several examples of functional programming with dependent types, including several non-trivial unification patterns coming from dependent matches. As a result, from a total of 6,200 lines, only 14 required extra typing annotations. It is interesting to note that 8 of those lines are solved with the use of a bi-directional elaboration algorithm [*e.g.*, 4] enabled by Coq's **Program** keyword. For instance, some lines construct witnesses for existential quantification, similar to the example shown in §4.6.

The third one is the Mathematical Components library [11], version 1.5beta1. This library presents several challenges, making it appealing for our purpose: (1) It is a huge development, with a total of 78 theory files. (2) It uses canonical structures heavily, providing us with several examples of canonical structures idioms that UniCoq should support. (3) It uses its own set of tactics uniformly calling the same unification algorithm used for elaboration. This last point is extremely important, although a bit technical. Truth be told, Coq has actually two different unification algorithms. One of these algorithms is mainly used by elaboration, and it outputs a sound substitution (up to bugs). This is the one mentioned in this paper as "the original unification algorithm of Coq". The other algorithm is used by most of Coq's original tactics (like apply or rewrite), but it is unsound (in Coq 8.4, it may return ill-typed solutions). Ssreflect's tactics use the former algorithm which is the one being replaced by our plugin. From the 82,000 lines in the library, only 40 lines required changes.

The last set of files also focuses in different canonical structures idioms: the files from Lemma Overloading [12]. It compiles almost as-is, with only one line requiring an extra annotation.

5.1 A Word on the Meta-DelDeps Rules

In a sense, the rules Meta-DelDeps are a bit brutal: they fix an arbitrary solution from the set of possible ones, which might not be the one expected. However, as the numbers above suggest, it works most of the time. In this section we analyse, for the Mathematical Components library, the origins of the unification problems that fail when this rule is turned off (totaling +300 lines).

Non-dependent* if−then−else*s: Most notably, the culprit for about two thirds of the failures are Ssreflect's **if**−**then**−**else**s. In Ssreflect, the type of the branches of an **if** are assumed to depend on the conditional. For instance, the example **if** b **then** 0 **else** 1 fails to compile if the Ssreflect library is imported. With Ssreflect, a fresh meta-variable $?T$ is created for the type of the branches, with contextual type $\mathsf{Type}[b : \mathsf{true}]$. When unifying it with the actual type of each branch, b is substituted by the corresponding boolean constructor. This results in the following equations:

$$?T[\mathsf{true}] \approx \mathsf{nat} \qquad ?T[\mathsf{false}] \approx \mathsf{nat}$$

Since they are not of the form required by HOPU, UniCoq (without the Meta-DelDeps rules) fails.

False dependency in the* in *modifier: Another less common issue comes from the in modifier in Ssreflect's rewrite tactic. This modifier allows the selection of a portion of the goal to perform the rewrite. For instance, if the goal is $1 + x = x + 1$ and we want to apply commutativity of addition on the term on the right, we can perform the following rewrite:

$$\mathsf{rewrite}\ [\mathsf{in}\ X\ \mathsf{in}\ \_ = X]\mathsf{addnC}$$

With the rule, UniCoq instantiates X with the r.h.s. of the equation, and rewrite applies commutativity only to that portion of the goal. Without it, however, rewrite fails. In this case, the hole (_) is replaced by a meta-variable $?y$, which is assumed to depend on X. But X is also replaced by a meta-variable, $?z$, therefore the unification problem becomes

$$?y[x, ?z[x]] = ?z[x] \approx 1 + x = x + 1$$

that, in turn, poses the equation $?y[x, ?z[x]] \approx 1 + x$, which does not have an MGU.

Non-dependent products: About 30 lines required a simple typing annotation to remove dependencies in products. Consider the following Coq term:

$$\forall P\ x.\ (P\ (\mathsf{S}\ x) = \mathsf{True})$$

When Coq elaborates this term, it first assigns P and x two unknown types, $?T$ and $?U$ respectively, the latter depending on P. Then, it elaborates the term on the left of the equal sign, obtaining further information about the type $?T$ of P: it has to be a (possibly dependent) function $\forall y : \mathsf{nat}.\ ?T'[y]$. The type of the term on the left is the type of P applied to $\mathsf{S}\ x$, that is, $?T'[\mathsf{S}\ x]$. After elaborating the term on the right and finding out it is a Prop, it unifies the types of the two terms, obtaining the equation

$$?T'[\mathsf{S}\ x] \approx \mathsf{Prop}$$

Since, again, this equation does not comply with HOPU, UniCoq fails without Meta-DelDeps.

Explicit duplicated dependencies: There are 15 occurrences where the proof developer wrote explicitly a dependency that duplicates an existing one. Consider for instance the following rewrite statement:

$$\mathsf{rewrite}\ [\_ + \_\ w]\mathsf{addnC}$$

Here, the proof developer intends to rewrite using commutativity on a fragment of the goal matching the pattern $\_ + \_\ w$. Let's assume that in the goal there is one occurrence of addition having w occurring in the right, say $t + (w + u)$, for some terms t and u. Since the holes (_) are elaborated as a meta-variable depending on the entire local context, in this case it will include w. Therefore, the pattern will be elaborated as $?y[w] + ?z[w]\ w$ (assuming no other variables appear in the local context). When unifying the pattern with the desired occurrence we obtain the problem:

$$?z[w]\ w \approx w + u$$

This equation does not have a MGU, since either w on the l.h.s. can be used as a representative for the w on the r.h.s.. The rules Meta-DelDeps remove the inner w.

Looking closely into these issues, it seems as if the dependencies were incorrectly introduced in the first place. We plan to study modifications to elaboration and tactics to avoid these dependencies, and study the impact of such changes.

6. Closing remarks

We presented the first formalization of a realistic unification algorithm for Coq, featuring overloading and universe polymorphism.

Moreover, we give a precise characterization of *controlled backtracking* (rules APP-FO plus CONS-*), which, together with the rules for overloading (Figure 8), allow us to explain the patterns introduced in [12]. The algorithm presented in this work is predictable, in the sense that the order in which subproblems are evaluated can be deduced directly from the rules. In particular, we have not introduced the technique of constraint postponement, which reorders unification subproblems. This omission, made in favor of predictability, has shown not to be problematic in practice (§5).

The algorithm includes a heuristic, incarnated in the rules META-DELDEPS that forces a non-dependent solution where multiple solutions might exist. We have studied various scenarios where it is being used, and shown that this heuristic can be replaced in most cases by smarter tactics and elaboration algorithms (§5.1).

In the future we plan to prove soundness of the algorithm (see §4.7), and to improve its performance to make it significantly faster than the current algorithm of COQ.

Acknowledgments

We are deeply grateful to Georges Gonthier for his suggestion on adding the META-DELDEPS rules, Enrico Tassi for carefully explaining the θ reduction strategy and its use, and Andreas Abel, Derek Dreyer, Hugo Herbelin, Aleksandar Nanevski, Scott Kilpatrick, Viktor Vafeiadis, and the anonymous reviewers for their important feedback on earlier versions of this work. This research was partially supported by EU 7FP grant agreement 295261 (MEALS).

References

[1] A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA*. Springer, 2011.

[2] A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli. Crafting a proof assistant. In *TYPES*. Springer-Verlag, 2006.

[3] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. Hints in unification. In *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009.

[4] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *LMCS*, 8(1), 2012.

[5] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *JFP*, 23, 2013.

[6] I. Cervesato and F. Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. http://adam.chlipala.net/cpdt/.

[8] F. Garillot. *Generic Proof Tools and Finite Group Theory*. PhD thesis, Ecole Polytechnique X, Dec. 2011.

[9] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging Mathematical Structures. In *TPHOL*. Springer, 2009.

[10] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP*. Springer, 2013.

[11] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008.

[12] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *JFP*, 23(04):357–401, 2013.

[13] A. Mahboubi and E. Tassi. Canonical Structures for the working Coq user. In *ITP*. Springer, 2013.

[14] D. Miller. Unification of simply typed lamda-terms as logic programming. In *ICLP*. MIT Press, 1991.

[15] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3), June 2008.

[16] U. Norell. Dependently Typed Programming in Agda. In *TLDI*. ACM, 2009.

[17] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *ICFP*. ACM, 2006.

[18] J. Reed. Higher-order constraint simplification in dependent type theory. In *LFMTP*, 2009.

[19] C. Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004.

[20] A. Saïbi. *Outils Genriques de Modelisation et de Demonstration pour la Formalisation des Mathematiques en Theorie des Types. Application a la Theorie des Categories.* PhD thesis, University Paris 6, 1999.

[21] M. Sozeau and N. Tabareau. Universe Polymorphism in Coq. In *ITP*. Springer, 2014.

[22] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012.

[23] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, pages 60–76, 1989.

[24] B. Ziliani, D. Dreyer, N. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in coq. *To appear in JFP*, ??(?):??–??, 2015.

[25] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *ICFP*, 2013.