Under consideration for publication in J. Functional Programming

A Comprehensible Guide to a New Unifier for CIC Including Universe Polymorphism and Overloading

BETA ZILIANI
FAMAF, Universidad Nacional de Córdoba (Argentina), CONICET (Argentina)
bziliani@famaf.unc.edu.ar
and
MATTHIEU SOZEAU
Inria & PPS (France), Université Paris Diderot (France)
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 (CIC, the base logic of COQ), building it up from a basic calculus to the full CIC as it is implemented in 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.

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 (Gonthier *et al.*, 2013b) in the proof assistant COQ (The Coq Development Team, 2012). 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*

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* (Saïbi, 1999). 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* (Asperti *et al.*, 2009). It is so powerful, in fact, that it enables the development of dependently-typed logic meta-programs (Gonthier *et al.*, 2013a).

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 (Mahboubi & Tassi, 2013; Garillot *et al.*, 2009; Garillot, 2011; Gonthier *et al.*, 2013a), 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 main contributions are:

1. An original, full-fledged description of a unification algorithm for CIC, incorporating canonical structures and universe polymorphism (Sozeau & Tabareau, 2014).
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.
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 (Gonthier *et al.*, 2008) and CPDT (Chlipala, 2011).

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 (Norell, 2009), Matita (Asperti *et al.*, 2006), or Idris (Brady, 2013). Developers of such systems, or new systems to come, may find the discussions in this work about incorporating or removing certain heuristics inspiring.

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

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

Fig. 1. List membership axioms and lemmas.

This work is an extended version—and a total restructuring—of Ziliani & Sozeau (2015). In this new version we split different features of the algorithm in different sections, building from the basic Calculus of Constructions up to the full Calculus of Inductive Constructions implemented by COQ, making the presentation much more palatable. We have also incorporated several real or realistic examples and fixed some bugs and inconsistencies in notation.

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 Calculus of Constructions (CC) together with a simple minded unification algorithm for it (§3). We extend the algorithm to include β-reduction and η-expansion (§4), local and global definitions (§5), universes (§6), inductive types (§7), and overloading (§8). We also incorporate controlled backtracking in §9 and several heuristics for meta-variable instantiation in §10. The last addition to the algorithm is universe polymorphism (§11), and once every rule is given we specify the priority of the rules (§12). We discuss why we did not incorporate the technique known as Constraint Postponement in §13, and we show it is not that important in real developments (§14). We discuss what would be a correctness criterion for the algorithm in §15. We show in detail an example inspired by Gonthier *et al.* (2011) in §16. Related work is discussed in §17, and conclusion are drawn in §18.

2 COQ's Unification at a Glance

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

Term unification: The unification algorithm of COQ must deal with unification of *terms* and not only *types*. In fact, in the Calculus of Inductive Constructions (CIC), the base logic of COQ, there is no syntactical distinction between types and terms.

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:

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

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 $l \mathbin{++} r$ for the concatenation of lists l and r. We also denote $[a_1; \ldots; a_n]$ for 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 (note how the *type* here is a predicates over lists, *i.e.*, *terms*). 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.*, Miller, 1991; Reed, 2009; Abel & Pientka, 2011; Peyton Jones *et al.*, 2006) 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 (Norell, 2009), 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 (Sacerdoti Coen, 2004, 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,

[1] How elaboration works will not be discussed in this work. The interested reader is invited to read (Asperti *et al.*, 2012), which provides details on bi-directional elaboration in the Matita proof assistant, also based on CIC.

where the list on the left of the concatenation is **let**-bound:

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

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. To see this, let's the result of reducing both sides:

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

While the head of both lists is the same, the tail $[y_2]$ cannot be matched with the concatenation of two unknown lists. For this reason, the algorithm 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 (Gonthier *et al.*, 2013a).

Interactivity: Coq is an *interactive* theorem prover, meaning that the user writes her proof interactively, step by step. This has the consequence that the user will likely see the result of the algorithm, so it is not the same if terms are reduced or not—not only for *coverability*, as seen in the previous example, but also for *visibility*.

It is interesting to note that the interaction between Coq users and the prover differs greatly from that of Agda users, generating also different expectations for what the algorithm produces. In Agda, the application of a lemma is often written in full form, with all of its non-implicit arguments fleshed out, since it is seen more like a function application in a regular programming language.

Instead, Coq users more often than not apply lemmas using the apply *tactic*—an external program that inserts meta-variables for the arguments of the lemma, letting unification guess the actual values. Therefore, an example like ex0 is more common to see written as:

```
Definition ex0 : y₁ ∈ ([y₁] ++ [y₂]).
Proof
  apply inL. apply in_head.
Qed
```

If the unification algorithm were restricted to only output MGUs, proofs like the one above—that is, most of the existing proofs!—would currently break, as, even if the current algorithm does handle constraint postponment, those must be solved at each full stop (.) using heuristics. Coq 8.6 has an option to let those constraints float without applying heuristics, however, this mode of use can make tactic programming very inconvenient, as failure to solve the constraints (no progress) might mean either that the unifier was not capable enough and needed more information or that the constraints are effectively unsolvable but cannot be seen to be yet. Together with the backtracking behavior of tactics, this can result in very unpredictable tactic programs. We are currently exploring solutions to this problem by giving the user control over these constraints, however it is unlikely to be the kind of mode a user interacting with COQ would expect.

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 (Wadler & Blott, 1989), 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:

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

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:

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

where 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:

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

where

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

In order to use instances eq_bool and 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 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 eqType_pair, resulting in two new unification subproblems, for fresh

[2] This example is a significant simplification of one taken from (Gonthier *et al.*, 2013a, 2008).

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 during unification. Previous work by Harper & Pollack (1991) considered only *conversion* on a universe polymorphic version of CC with definitions and typical ambiguity (see §17 for a more detailed comparison). 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.

Immediate resolution: When considering that the unification algorithm is the central component of proof-search—either directly using Canonical Structures (*c.f.*, Section 16), or indirectly by calling a tactic that uses unification, such as apply—it becomes crucial for the algorithm to produce a definite answer, be it positive or negative.

As a consequence, the order of unification subproblems matter. For instance, consider the following example:

```
Definition pair_example (z:nat) :
  let x := _ in let y := _ in
  (x, x + y) = (0, z)
:= eq_refl.
```

At high-level, it produces two equations:

$$?x \approx 0 \tag{1}$$

$$?x + ?y \approx z \tag{2}$$

After solving the first one it can successfully solve the second one, since $0 + ?y$ reduces to $?y$, obtaining the equation $?y \approx z$. However, when we swap the pairs:

```
Definition pair_example_fail (z:nat) :
  let x := _ in let y := _ in
  (x + y, x) = (z, 0)
:= eq_refl.
```

The equations are now ordered so that Equation 2 above is considered first. Since it is not solvable as is—that is, without *delaying* it until the first equation is considered—the algorithm fails.

In Agda, for instance, equations that cannot be solved are delayed, meaning that the example above typechecks. In Coq, the existing algorithm delays equations only if one of the two sides of the equation is a meta-variable (and therefore, it fails to typecheck the example). Our algorithm goes a step beyond and forbids delaying of equations entirely, favoring a somewhat more predictable algorithm (*c.f.*, Section 13).

3 Structural Unification for CC

In this section we start developing an intuitive, simple-minded, algorithm for the Calculus of Constructions (CC), the basic theory behind the Calculus of Inductive Constructions (CIC). The presentation here is inspired by Pfenning (1991) and Sacerdoti Coen (2004).

CC is a λ-calculus with dependent types defined as:

$$\begin{array}{rll} s = & \mathsf{Prop} \mid \mathsf{Type} & \textit{sorts} \\ t,u,T,U = & x \mid c \mid s \mid ?x \mid \forall x:T.\, U \mid \lambda x:T.\, t \mid t\ \overline{u} & \textit{terms and types} \end{array}$$

Sorts, also called *kinds*, include the set of propositions Prop, and its *kind* Type. In CC, terms and types live in the same syntactic class, although we will differentiate a term from a type by writing the former in lowercase, as in t and u, and the latter in uppercase, as in T and U. Terms and types are constructed with variables $x \in \mathscr{V}$, constants $c \in \mathscr{C}$, sorts s, meta-variables $?x$, dependent products $\forall x : T.\ U$, function abstractions $\lambda x : T.\ t$, and applications. A meta-variable represents a *hole*, that is, a missing piece of the term (or proof). For applications we borrow the *spine* notation from Cervesato & Pfenning (2003), and note $t\ \overline{u}$ to represent the application of a list of terms $\overline{u}$ to term t. We call t the *head* of the term, which cannot be itself an application. If we need to specify the number n of arguments, we extend the notation as $\overline{u_n}$.

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

- Meta-context containing meta-variable declarations and definitions.
- Local context for bound variables.
- A global environment E, containing the types for constants.

Formally,

$$\begin{array}{rll} \Sigma = & \cdot \mid ?x:T,\Sigma \mid ?x := t:T,\Sigma & \textit{meta-contexts} \\ \Gamma = & \cdot \mid x:T,\Gamma & \textit{local contexts} \\ E = & \cdot \mid c:T,E & \textit{global environment} \end{array}$$

Meta-variables are declared (or defined) in the *set* Σ. We emphasize the word *set* because a meta-variable in Σ may contain other meta-variables, even newer ones, in its type T or its definition t (when defined). The only restriction is that the dependencies between meta-variables form an acyclic graph. This is in contrast with the other two contexts Γ and E. We write $?x := t : T$ to indicate that $?x$ is defined, that is, should be substituted by t. In this way, our meta-context also serves the purpose of representing a substitution. For the moment we will not consider meta-variables having free variables in its type T (or

$$(\lambda x:T.\,t)\;u \;\rightsquigarrow_{\beta}\; t\{u/x\} \qquad \frac{?x := t : T \in \Sigma}{?x \;\rightsquigarrow_{\delta\Sigma}\; t} \qquad \frac{t \rightsquigarrow_r t'}{t \overset{\mathrm{w}}{\rightsquigarrow}_r t'} \qquad \frac{t \overset{\mathrm{w}}{\rightsquigarrow}_r t'}{t\;u \overset{\mathrm{w}}{\rightsquigarrow}_r t'\;u}$$

Fig. 2. Reduction rules in CC.

defining term t). It is common to consider meta-variables with closed types, although we will later show why this is not good for our purposes and change to a richer definition of meta-variables in Section 10.

The local context is a *list* associating variables with types, where each type might include free variables previously declared, and meta-variables.

The global environment is another list associating constants c with a type T. No meta-variable nor variable is allowed to occur freely in T.

3.1 Reduction rules

Reduction of CC terms is performed through a set of rules listed in Figure 2. We have the standard β reduction rule, where we note $t\{u/x\}$ as the standard capture-avoiding substitution of x by u in t. More interestingly, the $\delta\Sigma$ reduction rule takes a meta-variable $?x$ defined in Σ, and replaces it by its definition t. The relation $t \overset{\mathrm{w}}{\rightsquigarrow}_r u$ states the *one-step weak-head reduction* of t into u using the relation stated in r ($r \in \{\beta, \delta\Sigma\}$ for the moment).

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

3.2 Structurally unifying CC terms

We show an algorithm to *structurally* unify two CC terms. That is, similarly to Sacerdoti Coen (2004, chp. 10), our algorithm will not reduce terms, so it will preserve the original structure of terms. Therefore, an equation like $(\lambda x.\ ?y)\ \mathsf{c} \ \approx\ \mathsf{d}$, where c and d are constants, will not have a solution, although a β-convertible solution exists; one that assigns d to $?y$. In following sections we will enrich our algorithm to allow such solutions.

Throughout this paper we will represent the algorithm using the following judgment:

$$\Sigma;\Gamma \vdash t_1 \approx t_2 \rhd \Sigma'$$

It unifies terms t_1 and t_2, given meta-context Σ and a local context Γ, and an implicit global environment E. For practical reasons, t_1 and t_2 are assumed to be well-typed under the contexts provided, although they might have different types. This is in contrasts to typed-directed algorithms to be found in the literature (*e.g.*, Abel & Pientka, 2011).

The algorithm returns a new meta-context Σ', which is an *extension* of Σ, perhaps with new meta-variables or definitions of existing meta-variables in Σ. If the algorithm succeeds, terms t_1 and t_2 are convertible under the returned context Σ'.

Figure 3 shows the rules of the algorithm. Rules TYPE-SAME, VAR-SAME, and RIGID-SAME apply when both terms are the same sort, variable, or constant, respectively. The reason to split them in three different rules is because we are going to change some of them in the coming sections. For products (PROD-SAME) and abstractions (LAM-SAME), we first unify

$$\textsc{Type-Same}\quad \frac{s \in \{\mathsf{Prop}, \mathsf{Type}\}}{\Sigma; \Gamma \vdash s \approx s \rhd \Sigma} \qquad \textsc{Var-Same}\quad \frac{x \in \mathscr{V}}{\Sigma; \Gamma \vdash x \approx x \rhd \Sigma} \qquad \textsc{Rigid-Same}\quad \frac{c \in \mathscr{C}}{\Sigma; \Gamma \vdash c \approx c \rhd \Sigma}$$

$$\textsc{Prod-Same}\quad \frac{\Sigma_0; \Gamma \vdash T_1 \approx U_1 \rhd \Sigma_1 \qquad \Sigma_1; \Gamma, x : T_1 \vdash T_2 \approx U_2 \rhd \Sigma_2}{\Sigma_0; \Gamma \vdash \forall x : T_1.\, T_2 \approx \forall x : U_1.\, U_2 \rhd \Sigma_2}$$

$$\textsc{Lam-Same}\quad \frac{\Sigma_0; \Gamma \vdash T \approx U \rhd \Sigma_1 \qquad \Sigma_1; \Gamma, x : T \vdash t \approx u \rhd \Sigma_2}{\Sigma_0; \Gamma \vdash \lambda x : T.\, t \approx \lambda x : U.\, u \rhd \Sigma_2}$$

$$\textsc{App-FO}\quad \frac{\Sigma_0; \Gamma \vdash t \approx u \rhd \Sigma_1 \qquad n > 0 \qquad \Sigma_1; \Gamma \vdash \overline{t_n} \approx \overline{u_n} \rhd \Sigma_2}{\Sigma_0; \Gamma \vdash t\, \overline{t_n} \approx u\, \overline{u_n} \rhd \Sigma_2}$$

$$\textsc{Meta-}\delta\textsc{R}\quad \frac{\begin{array}{c}\Sigma; \Gamma \vdash u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta\Sigma} u' \\ \Sigma; \Gamma \vdash t \approx u' \rhd \Sigma'\end{array}}{\Sigma; \Gamma \vdash t \approx u \rhd \Sigma'} \qquad \textsc{Meta-}\delta\textsc{L}\quad \frac{\begin{array}{c}\Sigma; \Gamma \vdash t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta\Sigma} t' \\ \Sigma; \Gamma \vdash t' \approx u \rhd \Sigma'\end{array}}{\Sigma; \Gamma \vdash t \approx u \rhd \Sigma'}$$

$$\textsc{Meta-Same}\quad \frac{\begin{array}{c} ?x : U \in \Sigma \qquad U \equiv \forall \Gamma_0.\, T \qquad \Gamma_0 \vdash \overline{y} \cap \overline{z} \rhd \Gamma_1 \\ \cdot \vdash \mathsf{sanitize}(\Gamma_1) \rhd \Gamma_2 \qquad \mathsf{FV}(T) \subseteq \mathsf{dom}(\Gamma_2) \qquad ?y \text{ fresh} \qquad \Sigma' = \Sigma \cup \{?y : \forall \Gamma_2.\, T, ?x := ?y\, \widehat{\Gamma_2}\}\end{array}}{\Sigma; \Gamma \vdash ?x\, \overline{y} \approx ?x\, \overline{z} \rhd \Sigma'}$$

$$\textsc{Meta-InstR}\quad \frac{\begin{array}{c} ?x : U \in \Sigma_0 \qquad U \equiv \forall \Gamma_0.\, T \\ t' = (\lambda \Gamma_0.\, t)\{\overline{y}/\hat{\Gamma_0}\}^{-1} \qquad \Sigma_0; \Gamma_0 \vdash t' : T' \qquad \Sigma_0; \Gamma_0 \vdash T' \approx T \rhd \Sigma_1 \qquad ?x \notin \mathsf{FMV}(t')\end{array}}{\Sigma_0; \Gamma \vdash t \approx ?x\, \overline{y} \rhd \Sigma_1 \cup \{?x := t'\}}$$

$$\textsc{Meta-InstL}\quad \frac{\begin{array}{c} ?x : U \in \Sigma_0 \qquad U \equiv \forall \Gamma_0.\, T \\ t' = (\lambda \Gamma_0.\, t)\{\overline{y}/\hat{\Gamma_0}\}^{-1} \qquad \Sigma_0; \Gamma_0 \vdash t' : T' \qquad \Sigma_0; \Gamma_0 \vdash T' \approx T \rhd \Sigma_1 \qquad ?x \notin \mathsf{FMV}(t')\end{array}}{\Sigma_0; \Gamma \vdash ?x\, \overline{y} \approx t \rhd \Sigma_1 \cup \{?x := t'\}}$$

Fig. 3. Unification algorithm for CC.

$$\textsc{Intersec-Nil}\quad \frac{}{\cdot \vdash \cdot \cap \cdot \rhd \cdot} \qquad \textsc{Intersec-Keep}\quad \frac{\Gamma \vdash \overline{y} \cap \overline{z} \rhd \Gamma'}{\Gamma, x : A \vdash \overline{y}, x' \cap \overline{z}, x' \rhd \Gamma', x : A} \qquad \textsc{Intersec-Remove}\quad \frac{\Gamma \vdash \overline{y} \cap \overline{z} \rhd \Gamma' \qquad y' \neq z'}{\Gamma, x : T \vdash \overline{y}, y' \cap \overline{z}, z' \rhd \Gamma'}$$

Fig. 4. Intersection judgment.

the types of the arguments, and then the body of the binder, with the local context extended with the bound variable.

We consider the application of a function to multiple arguments in the rule App-FO. The rule first compares functions t and u, and then proceeds to unify point-wise each of the arguments. The remaining rules consider meta-variables in the head position of a term. Rules Meta-δR and Meta-δL expand the definition of the meta-variable on the r.h.s. and

$$\textsc{Sanitize-Nil}\quad \frac{}{\xi \vdash \mathsf{sanitize}(\cdot) \rhd \cdot} \qquad\qquad \textsc{Sanitize-Keep}\quad \frac{\mathsf{FV}(T) \subseteq \overline{x} \qquad y,\overline{x} \vdash \mathsf{sanitize}(\Gamma) \rhd \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y : T, \Gamma) \rhd y : T, \Gamma'}$$

$$\textsc{Sanitize-Remove}\quad \frac{\mathsf{FV}(T) \not\subseteq \overline{x} \qquad \overline{x} \vdash \mathsf{sanitize}(\Gamma) \rhd \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y : T, \Gamma) \rhd \Gamma'}$$

Fig. 5. Sanitization of contexts.

l.h.s. respectively—a one-step $\delta\Sigma$ reduction. In the following we will write META-δ (without R or L) to mean both rules.

If the meta-variable has no definition we have to define (instantiate) the meta-variable. In Section 10 we will incorporate several useful heuristics to the algorithm for this particular case, but for the moment we restrict the algorithm to a subclass of equations known as *higher-order pattern unification* (HOPU) (Miller, 1991). Equations in this class, in which the meta-variable is applied to a spine of (distinct) variables, possess a Most General Unifier (MGU), that is, a unique solution that represents all possible solutions. We have two cases: either both sides of the equation have the same meta-variable at its head position (META-SAME), or we have a meta-variable in the head position on one side, and a term on the other (META-INSTR and META-INSTL). In the following paragraphs we explain each.

Same Meta-Variable: The rule META-SAME is used when we have the same meta-variable $?x$ in the head position of both terms in an equation. To better understand this rule, let us look at an example.

Example 1. *Suppose meta-variable $?z$ with type $\forall x_1 : \mathsf{nat}.\ \forall x_2 : \mathsf{nat}.\ T$, and the following equation*

$$?z\ y_1\ y_2 \ \approx\ ?z\ y_1\ y_3$$

where y_1, y_2 and y_3 are all distinct variables.

From this equation we cannot know yet what value $?z$ will be substituted with, but at least we know it cannot be a function using its second argument, x_2. If, for instance, later on $?z$ is instantiated with a term like $(\lambda x_1.\ \lambda x_2.\ x_2)$, applying the substitution and β-reducing both terms of the equation we obtain terms y_2 and y_3 respectively, which are not convertible. So we need to restrict the set of possible solutions to replace $?z$ such that they do not refer to x_2. This is achieved by creating a fresh meta-variable $?z'$ as a function of x_1 and instantiate $?z$ with it. The resulting substitution is:

$$\Sigma = \{?z' : (\forall x_1 : \mathsf{nat}.\ T), ?z := (\lambda x_1.\ \lambda x_2.\ ?z'\ x_1) : (\forall x_1 : \mathsf{nat}.\ \forall x_2 : \mathsf{nat}.\ T)\}$$

Rule META-SAME allows for the construction of such solution. It equates the application of a meta-variable $?x$ to two spines of variables $\overline{y}$ and $\overline{z}$, on the l.h.s. and on the r.h.s. respectively. First, we have that $?x$ has type U, and that U is convertible to a product type $\forall\Gamma_0.\ T$, where we implicitly assume that the size of Γ_0 is equal to the size of the spines (note that, since terms are assumed to be well typed, U must be convertible to such product type). Then, we filter all variables in Γ_0 where $\overline{y}$ and $\overline{z}$ disagree, obtaining a new context Γ_1.

This is reflected in the hypothesis

$$\Gamma_0 \vdash \overline{y} \cap \overline{z} \rhd \Gamma_1$$

The intersection judgment is shown in Figure 4. This judgment performs an intersection of the spines, filtering out those positions from the context Γ_1 where the substitutions disagree, resulting in Γ_2. Continuing with Example 1, Γ_0 is $x_1 : \mathsf{nat}, x_2 : \mathsf{nat}$, $\overline{y}$ is y_1, y_2, and $\overline{z}$ is y_1, y_3. Since y_2 and y_3 are different variables, the resulting context is $\Gamma_1 = x_1 : \mathsf{nat}$.

Fast-forwarding a bit, the last two hypotheses of the rule create a fresh meta-variable $?y$ with a product type using the previously generated context (after being *sanitized*, as we are going to see next), and substitutes $?x$ with $?y$, applying the spine of variables taken from the new context using the type-eraser function $\widehat{\cdot}$, defined as:

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

.

This would be all for the equation of the same meta-variable if it were not for the fact that the types of products might weakly depend on previous variables, and those variables might be eliminated by the intersection judgment. Let us illustrate with an example, where we assume the existence of constants for the theory of natural numbers ($0, \geq$, etc), with standard arity.

Example 2. *Let* $\Sigma = \{?x : \forall z : \mathsf{nat}.\ (\lambda w.\ 0)\ z \geq 0\}$ *and* $\Gamma = y : \mathsf{nat}, v : \mathsf{nat}$ *and equation*

$$?x\ y\ \approx\ ?x\ v$$

Note that the type of $?x$ is not β-normal, and that z is not really used in the co-domain, which can be normalized to $0 \geq 0$. But since in the equation z is being replaced with distinct variables y and v, then the intersection judgment will remove z from the type of $?x$, obtaining the ill-typed type:

$$(\lambda w.\ 0)\ z \geq 0$$

where z is not bound anywhere.

One option to solve this kind of issues is to normalize terms in each context, but that can be rather expensive. Instead, we take a different approach and restrict the set of solutions to not include such cases. That is, instead of making an effort to find a solution (for instance, by β-reducing the type of $?x$) we fail to find a solution. Formally, we make sure every variable whose type depends on a removed variable is also removed (hypothesis $\cdot \vdash \mathsf{sanitize}(\Gamma_1) \rhd \Gamma_2$), and then we check that the type has all free variables in the new context (hypothesis $\mathsf{FV}(T) \subseteq \mathsf{dom}(\Gamma_2)$). The sanitization judgment is defined in Figure 5.

Before moving to the next rule, we note that META-SAME generates an MGU or fails. This is because the intersection judgment only cuts variables where substitutions differ *strictly*: any solution using those variables must do it *weakly* (*e.g.*, like z in Example 2). Therefore, the normalized solution, without those variables, is still in the set of MGUs.

Meta-Variable Instantiation: The META-INST rules instantiate a meta-variable $?x$ with a term t, if such instantiation can be performed. As required by HOPU, the meta-variable is applied to a spine of variables $\overline{y}$. As with the META-SAME rule, we can assume $?x$ has type (convertible to) $\forall \Gamma_0.\ T$, where Γ_0 has the same size as $\overline{y}$. This rule must find a term t' to

$$
\begin{aligned}
y_i\{\overline{y}_n/\overline{x}_n\}^{-1} &= x_i & y_i \notin \{y_1, \dots y_{i-1}, y_{i+1}, \dots y_n\} \\
h\{\overline{y}/\overline{x}\}^{-1} &= h & h \in \mathscr{C} \cup \{\mathsf{Prop}, \mathsf{Type}\} \\
?x\{\overline{y}/\overline{x}\}^{-1} &= ?x & \\
(\forall x : T.\, U)\{\overline{y}/\overline{x}\}^{-1} &= \forall x : T\{\overline{y}/\overline{x}\}^{-1}.\, U\{\overline{y}/\overline{x}\}^{-1} & \\
(\lambda x : T.\, t)\{\overline{y}/\overline{x}\}^{-1} &= \lambda x : T\{\overline{y}/\overline{x}\}^{-1}.\, t\{\overline{y}/\overline{x}\}^{-1} & \\
(t\ u)\{\overline{y}/\overline{x}\}^{-1} &= t\{\overline{y}/\overline{x}\}^{-1}\ u\{\overline{y}/\overline{x}\}^{-1} &
\end{aligned}
$$

Fig. 6. Inverse substitution

substitute $?x$ with such that

$$t \equiv t'\ \overline{y}$$

t' must be a closed term; a function abstracting every variable in Γ_0 that, when applied to $\overline{y}$, returns t. We construct such term by "inverting" the substitution mapping variables from Γ_0 to variables in $\overline{y}$. The inverse substitution is defined in Figure 6. The only interesting case is when the term is a variable, in which there are two possible scenarios:

1. If the variable y_i is in the image of the substitution $\overline{y}_n$, and it appears only once in the image, then it is substituted with the variable at the same location in the domain x_i.
2. If the variable is not in the image, or it appears more than once, the substitution gets undefined.

The type of t', which now only depends on the context Γ_0, is computed as T', and unified with the type of $?x$, obtaining a new meta-context Σ_1. Finally, an *occurs check* is performed to prevent illegal solutions, making sure $?x$ does not occur in t'. The algorithm outputs Σ_1 plus the instantiation of $?x$ with t'.

The last two hypotheses are shown in gray because they are not needed if the type of the terms are unified prior to unifying the terms.

The rules listed in Figure 3 are overlapping. APP-FO overlaps with all of the META- rules, and these overlap with themselves. An actual algorithm will break the overlap between APP-FO and the rest of the rules by forbidding t and u's heads to be a meta-variable. Similarly, the overlap between META-SAME and the other two META-INST rules is avoided by requiring that t's head is not the same meta-variable $?y$ in the META-INST rules.

But what about the overlap between META-INSTR and META-INSTL? When both terms are different meta-variables applied to spines of variables, one can choose which rule to use. But note that the inversion of the substitution will not always be defined. The following example illustrates this case:

Example 3. *Assume* $\Sigma = \{?x : \forall v : \mathsf{nat}.\ \forall w : \mathsf{nat}.\ \mathsf{nat}, ?y : \forall u : \mathsf{nat}.\ \mathsf{nat}\}$. *In the following equation* META-INSTL *finds the solution while* META-INSTR *does not, assuming variable z is defined in the local context with the right type.*

$$?y\ z \ \approx\ ?x\ z\ z$$

In the r.h.s. the duplication of z makes the inversion of the substitution undefined, but on the l.h.s. it only occurs once so the substitution is perfectly well defined. A unification algorithm should try both cases in order to ensure no solution is missed.

Before moving to the next section we show the derivation tree from the example in the introduction.

Example 4 (Unification in a problem about list membership)**.** *Consider the* Coq *definition*

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

Containing the main unification problem:

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

Each meta-variable is defined as a function from the context, in this case containing variables y_1 and y_2.

We have to be honest here and say upfront that Coq has a more sophisticated representation for meta-variables, using what is called “Contextual Types”. For the moment, we will stick to the current representation of meta-variables, until Section 10 in which we introduce heuristics requiring contextual types.

Figure 7 shows the derivation tree from the example. It is interesting to note that this is a slightly beautified version of the actual derivation tree our algorithm outputs when put in debug mode. For this reason there are a few differences in the notation shown above and the one in the figures. Functions are written using standard Coq notation: fun x => t instead of $\lambda x.\, t$. If necessary, the type of x is added using the traditional $x : T$ notation. Also, the $\in$-operator is noted In in the figure, and cons and app are the names for the consing and appending list operations, respectively. We will often switch from the mathematical notation we used so far to Coq’s own and vice versa, assuming they are equivalent to the reader. In both notations we take the convention of collapsing several abstractions into one, and write $\lambda x_1\ x_2.\, t$ for $\lambda x_1.\, \lambda x_2.\, t$ (similarly for $\forall$s).

The rule Reduce-Same is a little optimization that compares two meta-closed terms (*i.e.*, with no meta-variables) for convertibility:

$$\textsc{Reduce-Same}\quad \frac{\mathsf{FMV}(t) = \mathsf{FMV}(u) = \emptyset \qquad t \equiv u}{\Sigma; \Gamma \vdash t \approx u \triangleright \Sigma}$$

In the figure there is a meta-variable $?z_0$ that is not present in the equation shown above. This meta-variable is defined as $?z_1\ y_1\ y_2 :: ?z_2\ y_1\ y_2$. The derivation is self-explanatory: At the top level the App-FO rule is applied, comparing first the head on both sides of the equation (In in both cases), and then compares the arguments. For the first argument we have $?z_1\ y_1\ y_2 \approx y_1$, which by the Meta-Inst rule instantiates $?z_1$ with $\lambda x_1\ x_2.\ x_1$, after checking that the type of both sides of the equation coincides (nat and nat). For the second argument we have $?z_0\ y_1\ y_2 +\!\!+ ?z_3\ y_1\ y_2 \approx [y_1] +\!\!+ [y_2]$. After comparing the heads, it is left with equations $?z_0\ y_1\ y_2 \approx [y_1]$ and $?z_3\ y_1\ y_2 \approx [y_2]$. The first one, after an implicit Meta-$\delta\Sigma$ step, is $?z_1\ y_1\ y_2 :: ?z_2\ y_1\ y_2 \approx [y_1]$. In this case, $?z_1$ definition is expanded, leading to the convertible equation $(\lambda x_1\ x_2.\, x_1)\ y_1 \approx y_1$. $?z_2$ is instantiated with a function returning the empty list. Similarly, for the second equation, $?z_3$ is instantiated with a function $\lambda x_1\ x_2.\ [x_2]$.

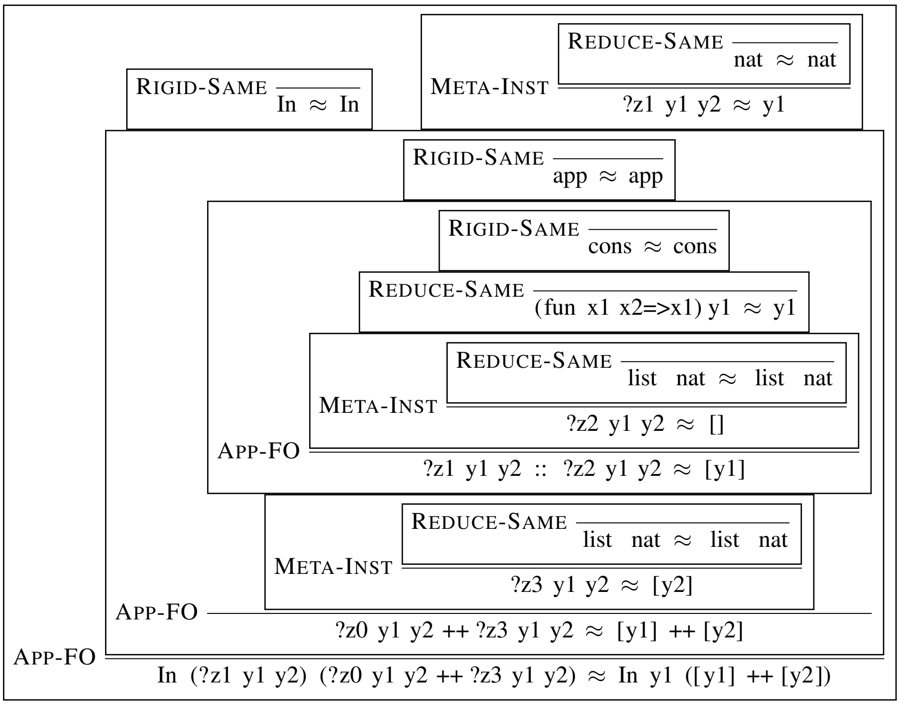

Fig. 7. Derivation tree of the unification problem.

$$\textsc{Lam-}\beta\textsc{R}\quad \frac{\Sigma;\Gamma\vdash u \overset{\text{w}}{\rightsquigarrow}_\beta u' \qquad \Sigma;\Gamma\vdash t\approx u' \rhd \Sigma'}{\Sigma;\Gamma\vdash t\approx u\rhd \Sigma'} \qquad\qquad \textsc{Lam-}\beta\textsc{L}\quad \frac{\Sigma;\Gamma\vdash t \overset{\text{w}}{\rightsquigarrow}_\beta t' \qquad \Sigma;\Gamma\vdash t'\approx u \rhd \Sigma'}{\Sigma;\Gamma\vdash t\approx u\rhd \Sigma'}$$

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

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

Fig. 8. β-reduction and η-expansion.

4 Unification modulo β-reduction and η-expansion

The first extension we do to the algorithm presented in Section 3 is to allow for β-reductions and η-expansions. We will use the exact same calculus as in Section 3, so we do not need to present it here.

The new rules are listed in Figure 8. The first two rules, LAM-βR and L, apply one-step β-reduction to each side of the equation. Following, we have η-expansion (LAM-η rules). These two rules unify 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 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 by calling the following function:

$$\begin{aligned}&\mathsf{ensure\_product}(\Sigma_0;\Gamma;T;U) = \Sigma_2\\ &\qquad \text{where } \Sigma_1 = \Sigma_0, ?v : \forall\Gamma.\ \forall y : T.\ \mathsf{Type} \qquad\qquad \text{for fresh } ?v\\ &\qquad \text{and } \Sigma_1;\Gamma \vdash U \approx \forall y : T.\ ?v\ \widehat{\Gamma}\ y \triangleright \Sigma_2\end{aligned}$$

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 function type $\forall\Gamma.\ \forall y : T.\ \mathsf{Type}$. That is, $?v$ has access to variables in the context Γ plus the new variable y.

The LAM-β rules introduce a new overlap with the rule APP-FO. The algorithm first tries APP-FO, and if it fails then it tries the LAM-β rules. There is also a new overlap among the LAM-β and the LAM-η rules, when having a β-redex on one side and an abstraction on the other. In this case the wise thing to do is to assign a priority to the rules. Our algorithm performs η-expansion last in the hope that β-reducing first will reveal the abstraction that will match that of the other side. But, ultimately, the set of solutions is the same if η-expansion is attempted first.

We show an example from the Ssreflect library (Gonthier *et al.*, 2008) featuring β-reductions.

Example 5 (Unification problems featuring β-reductions)**.** *The example comes from proving that subtracting n from m is odd iif XORing the oddity of m and the oddity of n is true:*

$$\mathsf{odd}\ (m - n) = \mathsf{odd}\ m \oplus \mathsf{odd}\ n \tag{3}$$

for any to natural numbers m and n.

We are interested only in the first step of the proof, in which the second argument of $\oplus$ is "moved" to the left:

$$\mathsf{odd}\ (m - n) \oplus \mathsf{odd}\ n = \mathsf{odd}\ m \tag{4}$$

This step is performed by the (partial) application of lemma canRL

$$\mathsf{canRL}\ \_\ \_\ (\mathsf{odd}\ (m - n))\ (\mathsf{odd}\ m)\ (\mathsf{addbK}\ \_)\ \_ \tag{5}$$

where

$$\begin{aligned}\mathsf{canRL} &: \forall (f\ g : \mathsf{bool} \to \mathsf{bool})\ (x\ y : \mathsf{bool}).\ \mathsf{cancel}\ f\ g \to f\ x = y \to x = g\ y\\ \mathsf{addbK} &: \forall b : \mathsf{bool}.\ \mathsf{cancel}\ (\lambda x : \mathsf{bool}.\ x \oplus b)\ (\lambda x : \mathsf{bool}.\ x \oplus b)\end{aligned}$$

With these ingredients we show two unification problems that arise from (5). In this work we will not explain in detail how the type inference algorithm of COQ works, and only provide the basics required to understand the examples. When COQ applies a term like (5) to the goal (3) it proceeds as follows:

1. It computes the type of the head element. In this case the head element is canRL and its type is

$$\forall (f\ g : \mathsf{bool} \to \mathsf{bool})\ (x\ y : \mathsf{bool}).\ \mathsf{cancel}\ f\ g \to f\ x = y \to x = g\ y$$

2. For each argument,
   - if it is a hole (_), it generates a fresh meta-variable with the right type, as a function of the local context. For instance, the first two arguments generate meta-variables $?f$ and $?g$ with type

     $$\forall m\ n : \mathsf{bool}.\ \mathsf{bool} \to \mathsf{bool}$$

     (Equivalent to $\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$.)
   - if it is a term, it unifies its type with the type corresponding to the argument's position. For instance, the type of $\mathsf{odd}\ (m - n)$ and $\mathsf{odd}\ m$ is unified with the type of x and y respectively (both of type bool).
3. Once every argument is processed, the type of the whole term is unified with the goal.

The two interesting bits are the unification of the type of addbK _ with the first unnamed argument and the unification of the whole term with the goal. The first one is:

$$\mathsf{cancel}\ (\lambda x : \mathsf{bool}.\ x \oplus (?b\ m\ n))\ (\lambda x : \mathsf{bool}.\ x \oplus (?b\ m\ n)) \approx \mathsf{cancel}\ (?f\ m\ n)\ (?g\ m\ n) \quad (6)$$

where $?b$ comes from the hole in $(\mathsf{addbK}\ \_)$, and has type $\forall m\ n : \mathsf{bool}.\ \mathsf{bool}$. The derivation tree resulting from solving this equation can be seen in Figure 9. In the figure non-dependent products $T \to U$ are written $\forall \_ : T.\ U$. Subtraction is noted subn, equality eq (parametrized over the type of the arguments), and the XOR operator is addb (for boolean addition). As can be seen in Figure 9, this problem is merely a structural matching between the two terms, in which $?f$ and $?g$ are instantiated with the same term

```
fun m n x => addb x (?b m n).
```

Now for the second equation, remember that we have to unify $x = g\ y$ with the goal, where x, y, and g are $\mathsf{odd}\ (m - n)$, $\mathsf{odd}\ m$, and $?g\ m\ n$, respectively. The unification problem becomes

$$\mathsf{odd}\ (m - n) = ?g\ m\ n\ (\mathsf{odd}\ m) \approx \mathsf{odd}\ (m - n) = \mathsf{odd}\ m \oplus \mathsf{odd}\ n$$

The derivation tree is shown in Figure 10. The left-hand sides of the equalities are exactly the same ($\mathsf{odd}\ (m - n)$). In the right-hand sides is where we see some action: the meta-variable $?g$ is (implicitly) $\delta\Sigma$-reduced in the l.h.s. to expose the function $(\lambda m\ n\ x.\ x \oplus ?b\ m\ n)$ applied to m, n, and $(\mathsf{odd}\ m)$. At this point the l.h.s. is β-reduced three times, until the constant addb occurs on both sides of the equation. Via APP-FO, we arrive at the point in which $?b$ is instantiated as $\lambda m\ n.\ \mathsf{odd}\ n$.

5 Adding local and global definitions

In this section we will add an important feature to our language: local and global definitions. The terms of the language are extended with $\mathbf{let} - \mathbf{in}$s:

$$t, u, T, U = \ \dots \mid \mathbf{let}\ x := t : T\ \mathbf{in}\ u \qquad \textit{terms and types}$$

The definitions are "stored" in the local and global context:

$$\Gamma = \ \dots \mid x := t : T, \Gamma \qquad \textit{local contexts}$$
$$E = \ \dots \mid c := t : T, E \qquad \textit{global environment}$$

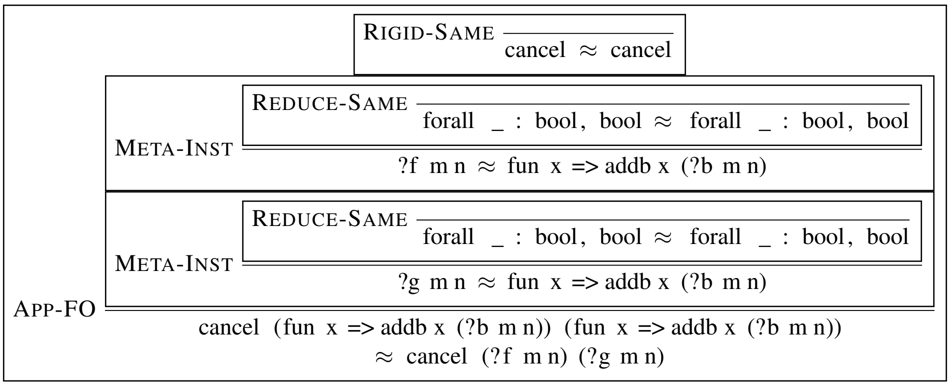
RIGID-SAME
cancel ≈ cancel
REDUCE-SAME
forall _ : bool, bool ≈ forall _ : bool, bool
META-INST
?f m n ≈ fun x => addb x (?b m n)
REDUCE-SAME
forall _ : bool, bool ≈ forall _ : bool, bool
META-INST
?g m n ≈ fun x => addb x (?b m n)
APP-FO
cancel (fun x => addb x (?b m n)) (fun x => addb x (?b m n))
≈ cancel (?f m n) (?g m n)

Fig. 9. Derivation of unifying the type addbK _ with the type of the argument in canRL.

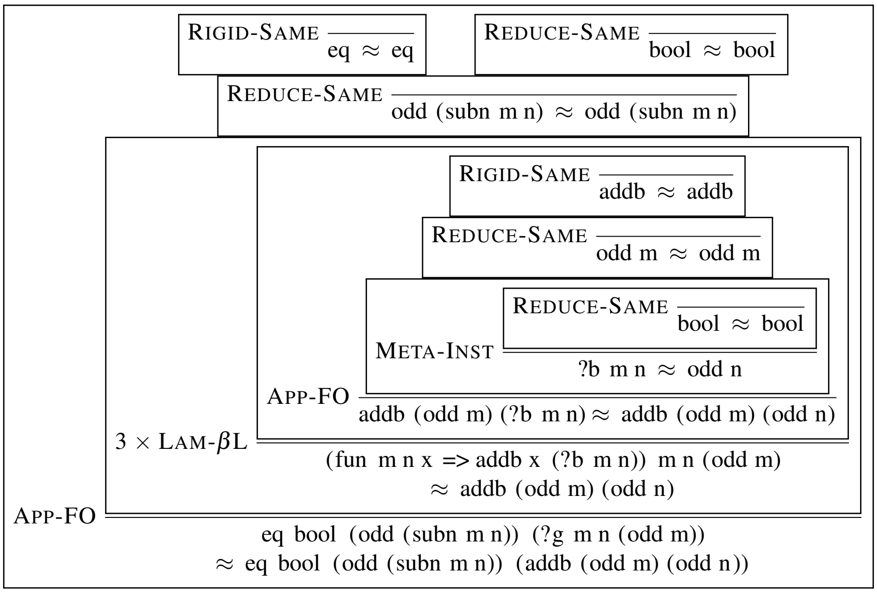
RIGID-SAME
eq ≈ eq
REDUCE-SAME
bool ≈ bool
REDUCE-SAME
odd (subn m n) ≈ odd (subn m n)
RIGID-SAME
addb ≈ addb
REDUCE-SAME
odd m ≈ odd m
REDUCE-SAME
bool ≈ bool
META-INST
?b m n ≈ odd n
APP-FO
addb (odd m) (?b m n) ≈ addb (odd m) (odd n)
3 × LAM-βL
(fun m n x => addb x (?b m n)) m n (odd m)
≈ addb (odd m) (odd n)
APP-FO
eq bool (odd (subn m n)) (?g m n (odd m))
≈ eq bool (odd (subn m n)) (addb (odd m) (odd n))

Fig. 10. Derivation of a simple unification problem featuring β-reduction.

We add three reduction rules, one to substitute the local definition in the body of the term (ζ-reduction), and two to expand local and global definitions:

$$\textbf{let}\ x := u : T\ \textbf{in}\ t \leadsto_{\zeta} t\{u/x\} \qquad \frac{(x := t : T) \in \Gamma}{x \leadsto_{\delta\Gamma} t} \qquad \frac{(c := t : T) \in E}{c \leadsto_{\delta E} t}$$

Figure 11 shows the new unification rules. They reduce terms according to the aforementioned reduction rules, with the sole exception of the LET-SAME rule: instead of ζ-reducing two **let**-**in**s, it tries to unify the definitions and then the bodies in a context augmented with one of the definitions (it takes the one from the right). This rule introduces several benefits: First, if the definition is used many times in the body, it will only be unified once. Second, if the variable of the definition occurs in the spine of a meta-variable, replacing it for the

$$\textsc{Let-Same}\quad \frac{\begin{array}{c}\Sigma_1;\Gamma \vdash t_2 \approx u_2 \triangleright \Sigma_2 \\ \Sigma_2;\Gamma, x := t_2 \vdash t_1 \approx u_1 \triangleright \Sigma_3\end{array}}{\Sigma_0;\Gamma \vdash \mathsf{\textbf{let}}\ x := t_2 : T\ \mathsf{\textbf{in}}\ t_1 \approx \mathsf{\textbf{let}}\ x := u_2 : U\ \mathsf{\textbf{in}}\ u_1 \triangleright \Sigma_3}$$

$$\textsc{Let-}\zeta\textsc{R}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash u \overset{w}{\rightsquigarrow}_{\zeta} u' \\ \Sigma;\Gamma \vdash t \approx u' \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'} \qquad \textsc{Let-}\zeta\textsc{L}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash t \overset{w}{\rightsquigarrow}_{\zeta} t' \\ \Sigma;\Gamma \vdash t' \approx u \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'}$$

$$\textsc{Var-}\delta\Gamma\textsc{R}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash u \overset{w}{\rightsquigarrow}_{\delta\Gamma} u' \\ \Sigma;\Gamma \vdash t \approx u' \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'} \qquad \textsc{Var-}\delta\Gamma\textsc{L}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash t \overset{w}{\rightsquigarrow}_{\delta\Gamma} t' \\ \Sigma;\Gamma \vdash t' \approx u \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'}$$

$$\textsc{Cons-}\delta E\textsc{R}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash u \overset{w}{\rightsquigarrow}_{\delta E} u' \\ \Sigma;\Gamma \vdash t \approx u' \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'} \qquad \textsc{Cons-}\delta E\textsc{L}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash t \overset{w}{\rightsquigarrow}_{\delta E} t' \\ \Sigma;\Gamma \vdash t' \approx u \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'}$$

Fig. 11. Unification rules for local and global definitions.

definition might make the equation fall outside HOPU (*c.f.*, Section 3.2). Third, it provides solutions structurally similar.

If the two **let-in**s fail to unify, the algorithm proceeds to reduce each side with the Let-ζ rules. Rules Var-$\delta\Gamma$ and Const-δE expand local and global definitions respectively. As we will see in Section 9, expanding definitions blindly is not a good idea, so we will present a heuristic that will improve the performance and coverability of the algorithm.

Additions to intersection and sanitization judgments: The intersection judgment should consider definitions in the local context:

$$\textsc{Intersec-Keep-Def}\quad \frac{\Gamma \vdash \overline{y} \cap \overline{z} \triangleright \Gamma'}{\Gamma, x := u : A \vdash \overline{y}, x' \cap \overline{z}, x' \triangleright \Gamma', x := u : A} \qquad \textsc{Intersec-Remove-Def}\quad \frac{\Gamma \vdash \overline{y} \cap \overline{z} \triangleright \Gamma' \qquad y' \neq z'}{\Gamma, x := u : T \vdash \overline{y}, y' \cap \overline{z}, z' \triangleright \Gamma'}$$

Similarly, we extend the sanitization judgment with the following rules:

$$\textsc{Sanitize-Keep-Def}\quad \frac{\mathsf{FV}(T) \subseteq \overline{x} \qquad \mathsf{FV}(u) \subseteq \overline{x} \qquad y, \overline{x} \vdash \mathsf{sanitize}(\Gamma) \triangleright \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y := u : T, \Gamma) \triangleright y := u : T, \Gamma'}$$

$$\textsc{Sanitize-Remove-Def}\quad \frac{\mathsf{FV}(T) \not\subseteq \overline{x} \vee \mathsf{FV}(u) \not\subseteq \overline{x} \qquad \overline{x} \vdash \mathsf{sanitize}(\Gamma) \triangleright \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y := u : T, \Gamma) \triangleright \Gamma'}$$

6 CC$^{\omega}$: the Type hierarchy

The Calculus of Constructions as presented so far only admits *impredicative* constructions, which is fine if one does not mind identifying different objects of the same type (the natural semantics of proof objects in this calculus). If we want to extend our calculus with *predicative* constructions, then we need to consider the Type hierarchy if we do not want to get caught by Girard's Paradox. The sorts of our language are replaced with:

$$\begin{array}{rlr} s = & \mathsf{Type}(\overline{K}^{+}) & \textit{sorts} \\ K = & \ell \mid K+1 & \\ \ell,\kappa,i,j \in \mathbb{N} & & \textit{universe levels} \end{array}$$

Sorts now include algebraic universes, which represent least upper bounds of a (non-empty) set of levels or successors of levels. They are used notably to sort products, e.g. $(\forall A : \mathsf{Type}(i), \mathsf{Type}(j)) : \mathsf{Type}(i{+}1, j{+}1)$, meaning that the type of $\forall A : \mathsf{Type}(i), \mathsf{Type}(j)$ is a Type with a level expected to be the greatest among $i+1$ and $j+1$. The special sort Prop is encoded as $\mathsf{Type}(0^{-})$, where the negative sign indicates its impredicative nature. For the purpose of unification, it is equivalent to $\mathsf{Type}(0)$.

The unification algorithm must check that universes are treated properly, so we need to extend it with a new context Φ to handle universe constraints.

$$\begin{array}{rllr} \Phi = & \overline{\ell} \vDash \mathscr{C} & & \textit{universe contexts} \\ \mathscr{C} = & \cdot \mid \mathscr{C} \wedge \ell \mathrel{\mathscr{O}} \ell' & \text{where } \mathscr{O} \in \{=,\leq,<\} & \textit{constraints} \end{array}$$

A universe context is basically a set of constraints $\mathscr{C}$ on universe levels $\overline{\ell}$. In Section 11 we will extend universe contexts to support polymorphic levels. Each constraint restricts a universe level to be equal, less than or equal, or less than another level.

The unification judgment is extended to receive and return universe contexts:

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

Where the relation

$$\mathscr{R} \;=\; \equiv \mid \leq$$

indicates if were are trying to derive conversion of the two terms or *cumulativity*, the subtyping relation on universes.

The new definition for the unification judgment forces us to rewrite the entirety of the rules we presented in previous sections. However, in order to ease the presentation, we will only focus on the main changes, and leave the rest of the rules untouched. The algorithm in full is shown in Appendix A.

$$\textsc{Type-Same}\quad \frac{\Phi' = \overline{\ell} \vDash \mathscr{C} \wedge \overline{u} \mathrel{\mathscr{R}} \kappa \qquad \Phi' \vDash}{\overline{\ell} \vDash \mathscr{C};\Sigma;\Gamma \vdash \mathsf{Type}(\overline{u}) \approx_{\mathscr{R}} \mathsf{Type}(\kappa) \rhd \Phi';\Sigma}$$

$$\textsc{App-FO}\quad \frac{\Phi_0;\Sigma_0;\Gamma \vdash t \approx_{\mathscr{R}} u \rhd \Phi_1;\Sigma_1 \qquad n > 0 \qquad \Phi_1;\Sigma_1;\Gamma \vdash \overline{t_n} \approx_{\equiv} \overline{u_n} \rhd \Phi_2;\Sigma_2}{\Phi_0;\Sigma_0;\Gamma \vdash t\ \overline{t_n} \approx_{\mathscr{R}} u\ \overline{u_n} \rhd \Phi_2;\Sigma_2}$$

$$\textsc{Lam-Same}\quad \frac{\Phi_0;\Sigma_0;\Gamma \vdash T \approx_{\equiv} U \rhd \Phi_1;\Sigma_1 \qquad \Phi_1;\Sigma_1;\Gamma, x : T \vdash t \approx_{\mathscr{R}} u \rhd \Phi_2;\Sigma_2}{\Phi_0;\Sigma_0;\Gamma \vdash \lambda x : T.\ t \approx_{\mathscr{R}} \lambda x : U.\ u \rhd \Phi_2;\Sigma_2}$$

The rule <span style="font-variant:small-caps">Type-Same</span> unifies two sorts, according to the relation $\mathscr{R}$. By an invariant on typing derivations, we know that the right-hand side universe can only be a single level

```
Inductive tree (A B : Set) : Set :=
  node : A → branch A B → tree A B
with branch (A B : Set) : Set :=
  | leaf : B → branch A B
  | cons : tree A B → branch A B → branch A B.

Fixpoint tree_size (t : tree) : nat :=
  match t with
  | node a f ⇒ S (branch_size f)
  end
with branch_size (b : branch) : nat :=
  match b with
  | leaf _ ⇒ 1
  | cons t b' ⇒ (tree_size t + branch_size b')
  end.
```

Fig. 12. A mutually inductive type: a tree.

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 Sozeau & Tabareau (2014) for details). The predicate $\Phi \models$ denotes satisfiability of the set of constraints in Φ.

For the rest of the rules, the universe context is just threaded along, as can be seen in the new version of rules App-FO and Lam-Same. More interestingly, the relation $\mathscr{R}$ is treated as follows: For App-FO, the head elements are unified respecting $\mathscr{R}$, while the arguments must respect strict conversion ($\equiv$). For Lam-Same the type of the arguments are unified using $\equiv$ while the body respects the given relation. The rest of the rules are modified accordingly.

7 CIC: Extending CC$^\omega$ with inductive types

In this section we arrive at the full calculus in which Coq is based on: the Calculus of Inductive Constructions (CIC) (The Coq Development Team, 2012, chap. 4). In essence it is CC$^\omega$ extended with inductive types. It also includes co-inductive types, but their formulation is not important for this work, so it will be omitted.

We show an example of a mutually inductive datatype in Figure 12. It is inspired by The Coq Development Team (2012, chap. 4). It consist of a tree, which is a node containing an element and finitely many branch es. Each branch consists of a leaf or the consing of a tree to a branch. Leafs are allowed to have objects of different type (B) than objects in trees (A). As an example of a mutually recursive fixpoint, we show in the same figure how to compute the size of the tree.

The terms (and types) of the language are extended with the following definitions:

$$\begin{array}{rll} t,u,T,U = & \ldots \mid i \mid k \mid \mathbf{match}_T\ t\ \mathbf{with}\ k_1\ \overline{x_1} \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x_n} \Rightarrow t_n\ \mathbf{end} & \textit{terms and types} \\ & \mid \mathbf{fix}_j\ \{f_1/n_1 : T_1 := t_1; \ldots; f_m/n_m : T_m := t_m\} & \end{array}$$

Terms include inductive type constructors $i \in \mathscr{I}$ and constructors $k \in \mathscr{K}$. In order to destruct an element of an inductive type, CIC provides regular pattern **match**ing and

$$
\begin{array}{l}
\{\ \mathsf{tree} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} := \{\mathsf{node} : \forall A\ B.\ \ A \to \mathsf{branch}\ A\ B \to \mathsf{tree}\ A\ B\}; \\
\quad \mathsf{branch} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} := \\
\qquad\quad \{\mathsf{leaf} : \forall A\ B.\ \ B \to \mathsf{branch}\ A\ B; \\
\qquad\quad\ \mathsf{cons} : \forall A\ B.\ \ \mathsf{tree}\ A\ B \to \mathsf{branch}\ A\ B \to \mathsf{branch}\ A\ B\}\ \} \\
\\
\quad \mathbf{fix}_0\ \{\ \mathsf{tree\_size}/0 : \mathsf{tree} \to \mathsf{nat} := \\
\qquad\quad \lambda t : \mathsf{tree}.\ \mathbf{match}_{\mathsf{nat}}\ t\ \mathbf{with} \\
\qquad\qquad\quad \mathsf{node}\ a\ f \Rightarrow \mathsf{S}\ (\mathsf{forest\_size}\ f) \\
\qquad\quad \mathbf{end}; \\
\qquad\quad \mathsf{branch\_size}/0 : \mathsf{branch} \to \mathsf{nat} := \\
\qquad\quad \lambda b : \mathsf{branch}.\ \mathbf{match}_{\mathsf{nat}}\ b\ \mathbf{with} \\
\qquad\qquad\quad \mathsf{leaf}\ l \Rightarrow 1 \\
\qquad\qquad\quad \mathsf{cons}\ t\ b' \Rightarrow (\mathsf{tree\_size}\ t + \mathsf{branch\_size}\ b') \\
\qquad\quad \mathbf{end}\ \}
\end{array}
$$

Fig. 13. Representation of the tree type in CIC.

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, $f/n : T := t$ means that f is a function of type T, with at least n arguments, and the n-th variable is the decreasing one in the body 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.

The global environment E is extended to allow inductive types:

$$
\begin{array}{lr}
E = \ \ldots\ \mid I, E & \textit{global environment} \\
I = \forall\Gamma.\ \{\ \overline{i : \forall \overline{y : T_h}.\ s := \{k_1 : U_1; \ldots; k_n : U_n\}}\ \} & \textit{inductive types}
\end{array}
$$

A set of mutually recursive inductive types I is prepended with 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 $\forall \overline{z : U'}.\ i\ t_1\ \ldots\ t_h$. The representation of the example in Figure 12 in our internal language is presented in Figure 13.

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 (The Coq Development Team, 2012, 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 Section 15.

Reduction of **fix**points and **match**es is performed with the ι-reduction:

$$
\mathbf{match}_T\ k_j\ \bar{t}\ \mathbf{with}\ \overline{k\ \bar{x} \Rightarrow u}\ \mathbf{end}\ \leadsto_\iota\ u_j\{\overline{t/x_j}\}
\qquad
\frac{F = \overline{f/n : T := t} \qquad a_n = k_j\ \bar{t}}{\mathbf{fix}_j\ \{F\}\ \bar{a}\ \leadsto_\iota\ t_j\{\overline{\mathbf{fix}_m\ \{F\}/f_m}\}\ \bar{a}}
$$

$$\textsc{Rigid-Same}\quad \frac{h \in \mathscr{C} \cup \mathscr{I} \cup \mathscr{K}}{\Sigma;\Gamma \vdash h \approx h \triangleright \Sigma}$$

$$\textsc{Case-Same}\quad \frac{\Sigma_0;\Gamma \vdash T \approx U \triangleright \Sigma_1 \qquad \Sigma_1;\Gamma \vdash t \approx u \triangleright \Sigma_2 \qquad \Sigma_2;\Gamma \vdash \overline{b} \approx \overline{b'} \triangleright \Sigma_3}{\Sigma_0;\Gamma \vdash \mathbf{match}_T\ t\ \mathbf{with}\ \overline{b}\ \mathbf{end} \approx \mathbf{match}_U\ u\ \mathbf{with}\ \overline{b'}\ \mathbf{end} \triangleright \Sigma_3}$$

$$\textsc{Fix-Same}\quad \frac{\Sigma_0;\Gamma \vdash \overline{T} \approx \overline{U} \triangleright \Sigma_1 \qquad \Sigma_1;\Gamma \vdash \overline{t} \approx \overline{u} \triangleright \Sigma_2}{\Sigma_0;\Gamma \vdash \mathbf{fix}_j\ \{\overline{x/n : T := t}\} \approx \mathbf{fix}_j\ \{\overline{x/n : U := u}\} \triangleright \Sigma_2}$$

$$\textsc{Case-}\iota\textsc{R}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash u \overset{\text{w}}{\rightsquigarrow}_{\iota} u' \\ \Sigma;\Gamma \vdash t \approx u' \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'} \qquad \textsc{Case-}\iota\textsc{L}\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash t \overset{\text{w}}{\rightsquigarrow}_{\iota} t' \\ \Sigma;\Gamma \vdash t' \approx u \triangleright \Sigma'\end{array}}{\Sigma;\Gamma \vdash t \approx u \triangleright \Sigma'}$$

Fig. 14. Unifying terms sharing the same head constructor.

When the scrutinee of the **match** is constructor k_j applied to terms $\bar{t}$, the corresponding branch u_j is returned, replacing every variable in the pattern with $\bar{t}$. For **fix**points, the body t_j of the j-th function defined in F is returned, substituting each occurrence of recursive calls f_m with the fixpoint definition for that m.

As for unification, we extend the rule Rigid-Same to also consider inductive types and constructors. Additionally, we have new rules Case-Same and Fix-Same which unify **match**es and **fix**points, respectively, by unifying pointwise every component of the term constructors. Finally, we have two rules to perform ι-reductions, one on each side of the equation (Case-ι).

8 Canonical Structures Resolution

We mentioned in the introduction the overloading mechanism known as *Canonical Structures* (CS). The example, which we revisit here, was the typical overloading of the (decidable) equality operator.

Example 6 (Overloading of equality operator)**.**

We define the eqType structure, similar to the Eq typeclass in Haskell:

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

A **Structure** in Coq, also known as a *record type* in systems like Agda, is just syntactic sugar for an inductive type with only one constructor, and with projections generated for each argument of the constructor. If we print the generated projector equal, for instance, we obtain:

$$\mathsf{equal} = \lambda e : \mathsf{eqType}.\ \mathbf{match}\ e\ \mathbf{with}\ |\ \mathsf{EqType}\ \_\ eq \Rightarrow eq\ \mathbf{end}$$

$$\textsc{Lookup-CS}\quad \frac{(p_j, h, c_\iota) \in \Delta_{\mathrm{db}} \qquad c_\iota \leadsto_{\delta E} \lambda \overline{x:T}.\, k\ \overline{p'}\ \overline{v} \qquad \Sigma_1 = \Sigma_0, \overline{?y:T} \qquad \Sigma_1;\Gamma \vdash \overline{p} \approx \overline{p'\{\overline{?y}/\overline{x}\}} \rhd \Sigma_2}{\Sigma_0 \vdash (p_j; \overline{p}; h) \in ?\ \Delta_{\mathrm{db}} \rhd \Sigma_2; c_\iota\ \overline{?y}, v_j\{\overline{?y}/\overline{x}\}}$$

$$\textsc{CS-ConstR}\quad \frac{\Sigma_0 \vdash (p_j; \overline{p}; c) \in ?\ \Delta_{\mathrm{db}} \rhd \Sigma_1; \iota; c\ \overline{u'} \qquad \Sigma_1;\Gamma \vdash \overline{u} \approx \overline{u'} \rhd \Sigma_2 \qquad \Sigma_2;\Gamma \vdash i \approx \iota \rhd \Sigma_3 \qquad \Sigma_4;\Gamma \vdash \overline{t'} \approx \overline{t} \rhd \Sigma_4}{\Sigma_0;\Gamma \vdash c\ \overline{u}\ \overline{t'} \approx p_j\ \overline{p}\ i\ \overline{t} \rhd \Sigma_4}$$

$$\textsc{CS-ProdR}\quad \frac{\Sigma_0 \vdash (p_j; \overline{p}; \rightarrow) \in ?\ \Delta_{\mathrm{db}} \rhd \Sigma_1; \iota; u \rightarrow u' \qquad \Sigma_1;\Gamma \vdash t \approx u \rhd \Sigma_2 \qquad \Sigma_2;\Gamma \vdash t' \approx u' \rhd \Sigma_3 \qquad \Sigma_3;\Gamma \vdash i \approx \iota \rhd \Sigma_4}{\Sigma_0;\Gamma \vdash t \rightarrow t' \approx p_j\ \overline{p}\ i \rhd \Sigma_4}$$

$$\textsc{CS-SortR}\quad \frac{\Sigma_0 \vdash (p_j; \overline{p}; s) \in ?\ \Delta_{\mathrm{db}} \rhd \Sigma_1; \iota; v_j \qquad \Sigma_1;\Gamma \vdash s \approx v_j \rhd \Sigma_2 \qquad \Sigma_2;\Gamma \vdash i \approx \iota \rhd \Sigma_3}{\Sigma_0;\Gamma \vdash s \approx p_j\ \overline{p}\ i \rhd \Sigma_3}$$

$$\textsc{CS-DefaultR}\quad \frac{\Sigma_0 \vdash (p_j; \overline{p}; \_) \in ?\ \Delta_{\mathrm{db}} \rhd \Sigma_1; \iota; v_j \qquad \Sigma_2;\Gamma \vdash t \approx v_j \rhd \Sigma_3 \qquad \Sigma_3;\Gamma \vdash i \approx \iota \rhd \Sigma_4}{\Sigma_0;\Gamma \vdash t \approx p_j\ \overline{p}\ i \rhd \Sigma_4}$$

Fig. 15. Canonical Structures resolution.

We instantiate the structure with equality for booleans and pairs, and made them *canonical*:

```
Definition eqType_bool := EqType bool eq_bool
Canonical eqType_bool

Definition eqType_pair (A B : eqType) :=
    EqType (sort A × sort B) (eq_pair A B)
Canonical eqType_pair
```

The expected behavior of declaring a definition as canonical is to let the unification algorithm know that, when the sort of an unknown instance is being matched with a constant, it should instantiate the unknown with the canonical instance declared with that same constant. For instance, the declarations above defines instances eq_bool and eq_pair as the canonical instances for bool and the × operator, respectively. With these definitions, the unification algorithm is able to find the missing bit in the expression equal _ (b_1, b_2) (c_1, c_2), where each variable is of type bool.

Technically, 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 is a triple (p, h, i), and 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. Besides constants, Coq allows three other types of keys: sorts, non-dependent products, and variables (which turn into *default* instances matching anything).

The process is formally described in Figure 15. We always start from an equation of the form:

$$t'' \approx p_j\,\overline{p}\,i\,\overline{t}$$

where p_j is a projector of a structure, $\overline{p}$ are the parameters of the structure, i is the instance (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''. Its body is a function taking arguments $\overline{x : T}$ and returning the term $k\,\overline{p'}\,\overline{v}$, with k the constructor of the structure, $\overline{p'}$ 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 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.
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.

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.

Figure 16 shows the derivation tree for the example posed at the beginning of the section. For readability we left out the spine of variables applied to each meta-variable. They play no role in this derivation.

9 Controlled backtracking

In Section 5, and Section 7 we incorporated several reduction strategies ($\delta\Gamma, \delta\Sigma, \iota$), without any consideration of the performance penalty that this process may incur. However, if at every unfolding of a constant, fixpoint evaluation, or case analysis, we consider again the whole set of rules, backtracking at every mismatch, it is quite easy to trash the performance of the algorithm. Therefore, a heuristic to reduce terms after a $\delta\Gamma, \delta\Sigma, \iota$ step is needed, taking into account that we might miss solutions if we reduce them too much, as already pointed out when introducing controlled backtracking in Section 1.

$$\boxed{\text{\textsc{CS}}\ \dfrac{\begin{array}{c}\boxed{\text{\textsc{CS}}\ \dfrac{\boxed{\text{\textsc{Meta-Inst}}\ \dfrac{\boxed{\text{\textsc{Reduce-Same}}\ \dfrac{}{\mathrm{eqType} \approx \mathrm{eqType}}}}{?\mathrm{X70} \approx \mathrm{eqType\_bool}}}}{\mathrm{sort}\ ?\mathrm{X70} \approx \mathrm{bool}}} \\ \boxed{\text{\textsc{CS}}\ \dfrac{\boxed{\text{\textsc{Meta-Inst}}\ \dfrac{\boxed{\text{\textsc{Reduce-Same}}\ \dfrac{}{\mathrm{eqType} \approx \mathrm{eqType}}}}{?\mathrm{X71} \approx \mathrm{eqType\_bool}}}}{\mathrm{sort}\ ?\mathrm{X71} \approx \mathrm{bool}}} \\ \boxed{\text{\textsc{Meta-Inst}}\ \dfrac{\boxed{\text{\textsc{Reduce-Same}}\ \dfrac{}{\mathrm{eqType} \approx \mathrm{eqType}}}}{?\mathrm{X67} \approx \mathrm{eqType\_pair}\ ?\mathrm{X70}\ ?\mathrm{X71}}}\end{array}}{\mathrm{sort}\ ?\mathrm{X67} \approx (?\mathrm{X68} * ?\mathrm{X69})}}$$

Fig. 16. Example of CS resolution for equality.

In this section we introduce changes to the rules Case-ι and Cons-δ. The high level idea will be to stop reducing when the algorithm finds a constant (or defined variable). More precisely, for Case-ι we want to be able to reduce the scrutinee of a case, or the argument of a fixpoint, 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:

$$\textbf{match}\ \mathsf{c}\ \textbf{with}\ 0 \Rightarrow \mathsf{d} \mid \_ \Rightarrow 1\ \textbf{end} \approx \mathsf{proj}\ ?f \tag{7}$$

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 new Case-ι rules shown in Figure 18 by weak-head normalizing the l.h.s. using the standard $\beta\zeta\delta\Sigma\iota$ reduction rules plus a new reduction rule, θ, which weak-head normalizes scrutinees (Figure 17). 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 7 not unifiable).

In Equation 7 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

$$\frac{t \downarrow^{\mathrm{w}}_{\beta\zeta\delta\iota} k_j\ \overline{a}}{\textsf{match}_T\ t\ \textsf{with}\ \overline{k\ \overline{x} \Rightarrow t'}\ \textsf{end} \leadsto_\theta \textsf{match}_T\ k_j\ \overline{a}\ \textsf{with}\ \overline{k\ \overline{x} \Rightarrow t'}\ \textsf{end}}$$

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

Fig. 17. The θ-reduction strategy.

both sides of the equation. For instance, consider the following equation:

$$\mathsf{c} \approx \mathsf{proj}\ ?f \tag{8}$$

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 $?f$:

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

(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 8 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

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

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 Gonthier *et al.* (2013a). In Section 16 we show in detail an example of this type of meta-programming.

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

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

$$\textsc{Cons-}\delta\textsc{NotStuckR}\quad \frac{\text{not } \Sigma;\Gamma \vdash \text{is\_stuck } u \qquad u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \qquad \Sigma;\Gamma \vdash u' \downarrow^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta} u'' \qquad \Sigma;\Gamma \vdash t \approx u'' \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\textsc{StuckL}\quad \frac{\Sigma;\Gamma \vdash \text{is\_stuck } u \qquad t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \qquad \Sigma;\Gamma \vdash t' \downarrow^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta} t'' \qquad \Sigma;\Gamma \vdash t'' \approx u \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\text{R}\quad \frac{\text{not } t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \qquad u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \qquad \Sigma;\Gamma \vdash u' \downarrow^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta} u'' \qquad \Sigma;\Gamma \vdash t \approx u'' \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx u \rhd \Sigma'}$$

$$\textsc{Cons-}\delta\text{L}\quad \frac{\text{not } u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \qquad t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \qquad \Sigma;\Gamma \vdash t' \downarrow^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta} t'' \qquad \Sigma;\Gamma \vdash t'' \approx u \rhd \Sigma'}{\Sigma;\Gamma \vdash t \approx u \rhd \Sigma'}$$

Fig. 18. New unification rules for reduction.

10 Heuristics for Meta-variable instantiation

We mentioned in Section 3 that meta-variables can only be instantiated with closed terms, which forces the creation of meta-variables having product type abstracting every variable from the local context. This treatment of meta-variables is easy to understand and implement, but very inefficient and leading to unnecessarily large terms. For instance, remember Example 4. The equation to solve there was:

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

And the solution generated by the algorithm was:

$$\begin{aligned} ?z_1 &:= \lambda x_1\ x_2.\ x_1 \\ ?z_2 &:= \lambda x_1\ x_2.\ [] \\ ?z_3 &:= \lambda x_1\ x_2.\ [x_2] \end{aligned}$$

Substituting these meta-variables in the term on the left of the original equation, we obtain the fairly unreadable term (remember, the user might stumble across this term!):

$$(\lambda x_1\ x_2.\ x_1)\ y_1\ y_2 \in (((\lambda x_1\ x_2.\ x_1)\ y_1\ y_2 :: (\lambda x_1\ x_2.\ [])\ y_1\ y_2) +\!\!+ (\lambda x_1\ x_2.\ [x_2])\ y_1\ y_2)$$

Instead, we would like our term to look like the original one:

$$y_1 \in ((y_1 :: []) +\!\!+ ([y_2])$$

Not only because it is cleaner to read; also because it avoids unnecessary β-reduction, like two of the three reductions in Figure 10.

One option is to force β-reduction when $\delta\Sigma$-expanding a meta-variable. But that does not solve the performance problem, and might reduce a function where it should not. After all, which abstractions should be considered part of the “local context” of the meta-variable, and be reduced to obtain a more “natural-looking” term? Coq solves this issue by encoding meta-variables with *contextual types*.

The definition of the meta-context changes to:

$$\begin{aligned}\Sigma &= \cdot \mid ?x : T[\Psi], \Sigma \mid ?x := t : T[\Psi], \Sigma \\ \Psi &= \Gamma\end{aligned}$$

Where type T and term t must have all of their free variables bound within the local context Ψ. In this work we borrow the notation $T[\Psi]$ from Contextual Modal Type Theory (Nanevski *et al.*, 2008).

The definition of terms also has to change to accommodate for the new definition. Since meta-variables are no longer defined as abstractions of the local context, we have to somehow specify what is the relation between the local context and the meta-variable context. This is done with what is called a *suspended substitution*, which is nothing more than a list of terms.

$$\begin{aligned}t, u, T, U &= \ldots \mid ?x[\sigma] && \textit{terms and types} \\ \sigma &= \bar{t}\end{aligned}$$

The expansion of the definition of a meta-variable in a term changes to:

$$?x[\sigma] \rightsquigarrow_{\delta\Sigma} t\{\sigma/\widehat{\Psi}\} \qquad\qquad \text{if } ?x := t : T[\Psi] \in \Sigma$$

That is, every variable in the domain of Ψ is replaced with the terms from σ.

In the simple examples shown so far, the local context of meta-variables 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 Ziliani *et al.* (2015). Suppose the function f is defined locally as follows:

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

The accessory typing annotation provides the expected type for the meta-variable. Assuming no other variables occur in scope, after elaboration f becomes:

$$f := \lambda w : \mathsf{nat}.\ ?v[w] \tag{9}$$

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 (9), $[w]$ 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:

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

After unfolding the definition of g (CONS-δL) it compares the two lambda abstractions (LAM-SAME twice), pushing 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 (CONS-δL and LAM-βL), we are left with 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), and were allowed to be defined with an open term, 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 $?v$ should be instantiated with w, not x. Under that instantiation, g will normalize to $\lambda x\, y : \mathtt{nat}.\ x$, and h will normalize to $\lambda z : \mathtt{nat}.\ z$.

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 $?v : \mathtt{nat}[w : \mathtt{nat}]$ with w is due to the problem falling in the pattern unification subset (*c.f.*, Section 3). 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.

In the following subsections we introduce different modifications and additions to the algorithm in order to deal with contextual types, at the same time enhancing the algorithm to solve a broader set of equations. For brevity we will only provide rules where the meta-variable is on the r.h.s. (rules ending with R).

Metavariables and goals Metavariables with contextual types are actually used in COQ to represent not only unification variables but also goals in interactive proof mode. One can hence expect metavariables to have a long lifespan, be seen from the user-level interface (they can be named in COQ for example), be dependent on each other and persist across tactic invocations. Historically, COQ had another unifier using untyped, context-free metavariables for the specific purpose of tactics (e.g. `apply` does not use the same algorithm as type inference). These light metavariables were expected to have a short lifespan and be used in a restricted context with few dependencies and a common scope, but metavariables with contextual types were alreay needed to handle more complex scenarios. Our work aims at making those untyped metavariables disappear by having a common algorithm for tactics and type inference/elaboration using contextual metavariables.

$$\textsc{Meta-Same-Same}\quad \frac{\Sigma;\Gamma \vdash \bar{t} \approx_{\equiv} \bar{u} \triangleright \Sigma'}{\Sigma;\Gamma \vdash ?x[\sigma]\,\bar{t} \approx_{\mathscr{R}} ?x[\sigma]\,\bar{u} \triangleright \Sigma'}$$

$$\textsc{Meta-Same}\quad \frac{\begin{array}{lll} ?x : T[\Psi_1] \in \Sigma & \Psi_1 \vdash \sigma \cap \sigma' \triangleright \Psi_2 & \cdot \vdash \mathsf{sanitize}(\Psi_2) \triangleright \Psi_3 \\ \mathsf{FV}(T) \subseteq \Psi_3 & \multicolumn{2}{l}{\Sigma \cup \{?y : T[\Psi_3], ?x := ?y[\widehat{\Psi_3}]\};\Gamma \vdash \bar{t} \approx_{\equiv} \bar{u} \triangleright \Sigma'} \end{array}}{\Sigma;\Gamma \vdash ?x[\sigma]\,\bar{t} \approx_{\mathscr{R}} ?x[\sigma']\,\bar{u} \triangleright \Sigma'} \qquad \textsc{Intersec-Nil}\quad \frac{}{\cdot \vdash \cdot \cap \cdot \triangleright \cdot}$$

$$\textsc{Intersec-Keep}\quad \frac{\Gamma \vdash \sigma \cap \sigma' \triangleright \Gamma'}{\Gamma, x : A \vdash \sigma, t \cap \sigma', t \triangleright \Gamma', x : A} \qquad \textsc{Intersec-Remove}\quad \frac{\Gamma \vdash \sigma \cap \sigma' \triangleright \Gamma' \qquad y \neq z}{\Gamma, x : T \vdash \sigma, y \cap \sigma', z \triangleright \Gamma'}$$

$$\textsc{Intersec-Keep-Def}\quad \frac{\Gamma \vdash \sigma \cap \sigma' \triangleright \Gamma'}{\Gamma, x := u : A \vdash \sigma, t \cap \sigma', t \triangleright \Gamma', x := u : A} \qquad \textsc{Intersec-Remove-Def}\quad \frac{\Gamma \vdash \sigma \cap \sigma' \triangleright \Gamma' \qquad y \neq z}{\Gamma, x := u : T \vdash \sigma, y \cap \sigma', z \triangleright \Gamma'}$$

Fig. 19. Unification of the same meta-variable.

$$\boxed{\textsc{App-FO}\ \frac{\begin{array}{c}\boxed{\textsc{Rigid-Same}\ \texttt{seq} \approx_{\leq} \texttt{seq}} \\ \boxed{\textsc{Meta-Same}\ \texttt{?X[R, e, t, sub, t', m]} \approx_{=} \texttt{?X[R, e, t, sub, t', m.+1]}}\end{array}}{\texttt{seq ?X[R, e, t, sub, t', m]} \approx_{\leq} \texttt{seq ?X[R, e, t, sub, t', m.+1]}}}$$

Fig. 20. Example of use of META-SAME rule in Ssreflect.

10.1 Improving META-SAME

In Section 3 the rule META-SAME intersected the arguments of the (same) meta-variable, using the intersection judgment from Abel & Pientka (2011). In this section we consider a different rule. When the algorithm is faced with the following equation:

$$?x[\sigma]\,\bar{t} \;\approx\; ?x[\sigma']\,\bar{u}$$

only the suspended substitutions are intersected. In order to compensate for not intersecting the arguments, we allow for solvable differences (that is, arguments that are convertible or unifiable). Additionally, we broaden the intersection judgment to consider general terms and not only variables, as it is a problem that arises frequently. For instance, Figure 20 shows another example taken from the Ssreflect library, in which one of the terms in the substitution is the successor of m (written $m.+1$).

The new definitions are given in Figure 19. We include a little optimization: if both substitutions have the same terms, no intersection (and therefore no new meta-variable) is generated (rule META-SAME-SAME).

10.2 Improving the META-INST *rules*

We make several enhancements to the instantiation rule, trying to maintain the "spirit" of the rule: only find a solution when there is not (much) place for ambiguities.

META-INSTR

$$\frac{\begin{array}{c} ?x : T[\Psi] \in \Sigma_0 \\ t', \overline{z}' = \mathsf{remove\_tail}(t; \overline{z}) \qquad t' \downarrow_{\beta}^{\mathrm{w}} t'' \qquad \Sigma_0 \vdash \mathsf{prune}(?x; \overline{y}, \overline{z}'; t'') \rhd \Sigma_1 \qquad \Sigma_1; \Gamma \vdash \overline{z}' : \overline{U} \\ t''' = (\lambda \overline{w : U}.\, \Sigma_1(t''))\{\overline{y}, \overline{z}' / \hat{\Psi}, \overline{w}\}^{-1} \qquad \Sigma_1; \Psi \vdash t''' : T' \qquad \Sigma_1; \Psi \vdash T' \approx_{\leq} T \rhd \Sigma_2 \end{array}}{\Sigma_0; \Gamma \vdash t \approx_{\mathscr{R}} ?x[\overline{y}]\ \overline{z} \rhd \Sigma_2 \cup \{?x := t'''\}}$$

Remember from Section 3 that the META-INST rules instantiate a meta-variable applying a variation of higher-order pattern unification (HOPU). They unify a meta-variable $?x$ with some term t, obtaining a most general unifier (MGU). (A caveat: in this section, we will get *almost* a MGU.) As required by HOPU, the meta-variable is applied to a suspended substitution mapping variables to variables, $\overline{y}$, and a spine of arguments $\overline{z}$, of variables only. Assuming $?x$ has (contextual) type $T[\Psi]$, this rule must find a term t''' to instantiate $?x$ such that

$$t \ \approx \ ?x[\overline{y}]\ \overline{z}$$

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

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

1. To avoid unnecessarily η-expanded solutions, the term t and arguments $\overline{z}$ are decomposed using the function $\mathsf{remove\_tail}(\cdot;\cdot)$:

$$\begin{array}{rll} \mathsf{remove\_tail}(t\ x; \overline{z}, x) = \mathsf{remove\_tail}(t; \overline{z}) & & \text{if } x \notin \mathsf{FV}(t) \wedge x \notin \overline{z} \\ \mathsf{remove\_tail}(t; \overline{z}) = (t, \overline{z}) & & \text{in any other case} \end{array}$$

   This function, applied to t and $\overline{z}$, returns a new term t' and a list of variables $\overline{z}'$, where there exists $\overline{z}''$ such that $t = t'\ \overline{z}''$ and $\overline{z} = \overline{z}', \overline{z}''$, and $\overline{z}''$ 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, $\overline{z}'$ is the empty list, $\overline{z}''$ is $[x, y]$, and t' is addn.
   The check that $x \notin \mathsf{FV}(t)$ and $x \notin \overline{z}$ 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 $t' \downarrow_{\beta}^{\mathrm{w}} t''$. 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 Abel & Pientka (2011). The formal description will be discussed below.
   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 $\overline{y}, \overline{z}'$. It does

so by removing elements from the suspended substitutions occurring in t'', containing variables outside of $\overline{y}, \overline{z}'$. 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

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

First, note that t''' has to be a function taking n arguments $\overline{w}$, where $n = |\overline{z}'|$. The list of types of $\overline{w}$ comes from the types of variables $\overline{z}'$ (noted $\Sigma_1; \Gamma \vdash \overline{z}' : \overline{U}$). The body of this function is the term obtained from the second step, t'', after its defined meta-variables are normalized with respect to the meta-context obtained in the previous step, noted $\Sigma_1(t'')$, in order to replace the meta-variables with the pruned ones. This step effectively removes false dependencies on variables not occurring in $\overline{y}, \overline{z}'$. The final term is obtained applying the inverse substitution (defined in Section 3), in which each variable in $\overline{y}, \overline{z}'$ are replaced with variables in the local context of the meta-variable $\widehat{\Psi}$ and the (freshly introduced) variables $\overline{w}$.

5. Finally, 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 $k \leq i, j$.

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

Pruning: Figure 21 shows the actual process of pruning. The pruning judgment is noted

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

It takes a meta-context Σ, a meta-variable $?x$, a list of variables $\overline{y}$, 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, *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 $?z$ applied to the suspended substitution σ. We have two possibilities: either every variable from every term in σ is included in $\overline{y}$, 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

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

PRUNE-RIGID

$$\frac{h \in s \cup \mathscr{C}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};h) \rhd \Sigma}$$

PRUNE-VAR

$$\frac{x \in \overline{y}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};x) \rhd \Sigma}$$

PRUNE-LAM, PRUNE-PROD

$$\frac{\Pi \in \{\lambda, \forall\} \qquad \Sigma \vdash \mathsf{prune}(?x;\overline{y},z;t) \rhd \Sigma'}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};\Pi z.\ t) \rhd \Sigma'}$$

PRUNE-LET

$$\frac{\begin{array}{c}\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};t_2) \rhd \Sigma_1 \\ \Sigma_1 \vdash \mathsf{prune}(?x;\overline{y},z;t_1) \rhd \Sigma_2\end{array}}{\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};\textbf{let}\ z := t_2\ \textbf{in}\ t_1) \rhd \Sigma_2}$$

PRUNE-APP

$$\frac{\begin{array}{c}\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};t) \rhd \Sigma_1 \\ \Sigma_i \vdash \mathsf{prune}(?x;\overline{y};t_i) \rhd \Sigma_{i+1} \qquad i \in [1,n]\end{array}}{\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};t\ \overline{t_n}) \rhd \Sigma_{n+1}}$$

PRUNE-META-NOPRUNE

$$\frac{\begin{array}{c}?z : T[\Psi_0] \in \Sigma \qquad ?x \neq ?z \\ \Psi_0 \vdash \mathsf{prune\_ctx}(?x;\overline{y};\sigma) \rhd \Psi_0\end{array}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};?z[\sigma]) \rhd \Sigma}$$

PRUNE-META

$$\frac{\begin{array}{c}?u : T[\Psi_0] \in \Sigma \qquad ?x \neq ?z \qquad \Psi_0 \vdash \mathsf{prune\_ctx}(?x;\overline{y};\sigma) \rhd \Psi_1 \\ \cdot \vdash \mathsf{sanitize}(\Psi_1) \rhd \Psi_2 \qquad \Sigma \vdash \mathsf{prune}(?x;\widehat{\Psi_2};T) \rhd \Sigma'\end{array}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};?z[\sigma]) \rhd \Sigma', ?u : \Sigma'(T)[\Psi_2] \cup \{?z := ?u[\widehat{\Psi_2}]\}}$$

PRUNECTX-NIL

$$\frac{}{\cdot \vdash \mathsf{prune\_ctx}(?x;\overline{y};\cdot) \rhd \cdot}$$

PRUNECTX-NOPRUNE

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

PRUNECTX-PRUNE

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

Fig. 21. Pruning of meta-variables.

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

Coming back to the rules in Figure 21, 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 $?z$. 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 $?z$ with a fresh meta-variable $?u$, having contextual type $T[\Psi_2]$.

It is important to note that, due to conversion, the process of pruning may lose 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 $?y$.

10.3 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*:

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

It unifies 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 APP-FO requires both terms being equated to have the same number of arguments.

10.4 Meta-Variable Dependencies Erasure

If META-INST and META-FO do not apply, the algorithm makes a somewhat brutal attempt. The rule META-DELDEPSR shown below 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. Like META-FO, this rule fixes an arbitrary solution where many solutions may exist, which might not be the one expected by the user. But, as we are going to see in Section 14, the solution selected works more often than not.

$$\textsc{Meta-DelDepsR}\quad \frac{\begin{array}{c} ?x : T[\Psi] \in \Sigma_0 \qquad l = [i \mid \sigma_i \text{ is variable and } \nexists j > i.\ \sigma_i = (\sigma,\overline{u})_j] \\ \cdot \vdash \mathsf{sanitize}(\Psi_{|\,l}) \rhd \Psi' \qquad \Sigma_0 \vdash \mathsf{prune}(?x;\hat{\Psi}';T) \rhd \Sigma_1 \\ \Sigma_1 \cup \{?y : \Sigma_1(T)[\Psi'], ?x := ?y[\widehat{\Psi'}]\};\Gamma \vdash t \approx ?y[\sigma_{|\,l}]\ \overline{u} \rhd \Sigma_2 \end{array}}{\Sigma_0;\Gamma \vdash t \approx ?x[\sigma]\ \overline{u} \rhd \Sigma_2}$$

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, Ψ, noting it as $\Psi_{|\,l}$. After sanitizing this context we obtain context Ψ'. We prune offending variables in T not in Ψ', and create a fresh meta-variable $?y$ in this restricted local context. $?x$ 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 $t \ \approx\ ?y[\sigma_{|\,l}]\ \overline{u}$, where $\sigma_{|\,l}$ is the restriction of σ to positions in l.

Following we analyze, for the Mathematical Components library (version 1.4), different cases where this rule is effectively used (totaling +300 lines of the library), and study alternatives to avoid it if one wishes for a more "principled" algorithm.

Non-dependent if−then−elses: Most notably, two thirds of the cases in which rules META-DELDEPS are required 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 and the rule is switched off. 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, our algorithm (without the META-DELDEPS rules) fails.

False dependency in the in **modifier:** A 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, our algorithm 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: If the rule is switched off, 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, it needs META-DELDEPS to succeed.

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. It would be interesting to study if such dependencies can be avoided with little changes to elaboration and the tactics, in order to avoid relying on META-DELDEPS to do the "dirty job".

10.5 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.

META-REDUCER

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

11 Universe Polymorphism

In the previous sections we explained a new unification algorithm that can be used for COQ version 8.4. But we aim at more; we want to tackle the recently released 8.5 version, and for that we need to take into account one if its major improvements: universe polymorphism. We use the example in Chapter 29 of COQ's Reference Manual[3] to understand the limitations of the monomorphic universes presented in Section 6, and the idea behind

[3] Available online at `http://coq.inria.fr/distrib/V8.5rc1/refman/Reference-Manual032.html`

universe polymorphism. The polymorphic identity function, in its traditional, non-universe polymorphic form, is defined as:

$$\textbf{Definition}\ \mathsf{id} := \lambda T\ (x : T).\ x$$

Implicitly, T has type $\mathsf{Type}(i)$ for some universally quantified level i. If we apply this definition to a kind, say Prop:

$$\textbf{Definition}\ \mathsf{idProp} := \mathsf{id}\ \_\ \mathsf{Prop}$$

COQ creates a new level j for the implicit Type, with the following universe constraints:

$$\mathsf{Prop} < j \land j < i$$

That is, the level of the implicit Type must be greater than Prop (since we have $\mathsf{Prop} : \mathsf{Type}(j)$), but since it is being the argument of id, it has to be lower than i, the level coming from id.

But what if we try to apply id to itself? The following definition, although perfectly valid from a theoretical point of view, is ill-typed:

$$\textbf{Definition}\ \mathsf{idid} := \mathsf{id}\ \_\ \mathsf{id}$$

The reason should be self-evident now: we are asking the implicit type to be greater than the type of id, which should be at the same time smaller than the type of id! If we call the implicit type level j', COQ is faced with the following, unsolvable, constraints:

$$i < j' \land j' < i$$

The problem comes from using the same level i in the two occurrences of id in the definition.

In COQ, universe polymorphism allows us to instantiate each occurrence of a polymorphic universe level with a universally quantified one (implicitly, *i.e.*, without user interaction). So, for instance, the universe polymorphic identity is declared as:

$$\textbf{Polymorphic Definition}\ \mathsf{pid} := \lambda T\ (x : T).\ x$$

(Note that the only difference with id is the declaration **Polymorphic**.) Now COQ allows the application of pid to itself:

$$\textbf{Definition}\ \mathsf{pidpid} := \mathsf{pid}\ \_\ \mathsf{pid}$$

Behind the scenes, the universe level in the definition of pid is what we call a *flexible* universe ℓ, and is instantiated with different levels in each occurrence of pid. The unsugared form of pidpid is:

$$\mathsf{pidpid} = \mathsf{pid}[\ell]\ \_\ \mathsf{pid}[\kappa]$$

with the universe context containing the following restriction:

$$\kappa < \ell$$

That is, without knowing what ℓ and κ will be, we know the former has to be larger than the latter.

COQ also allows inductive types to be universe polymorphic. We have to perform two changes in the language: extend constants, type constructors, and constructors with a substitution for universe levels (like $[\ell]$ above), and to extend the universe context to

$$\textsc{Type-Same}\quad \frac{\mathscr{C}' = \mathscr{C} \wedge \overline{u}\,\mathscr{R}\,\kappa \qquad \mathscr{C}' \vDash}{(\overline{\ell} \vDash \mathscr{C});\Sigma;\Gamma \vdash \mathsf{Type}(\overline{u}) \approx_{\mathscr{R}} \mathsf{Type}(\kappa) \triangleright \overline{\ell} \vDash \mathscr{C}';\Sigma}$$

$$\textsc{Rigid-Same}\quad \frac{h \in \mathscr{I} \cup \mathscr{K} \qquad \mathscr{C}_1 = \mathscr{C}_0 \wedge \overline{\kappa = \kappa'} \qquad \mathscr{C}_1 \models}{(\overline{\ell} \vDash \mathscr{C}_0);\Sigma;\Gamma \vdash h[\overline{\kappa}] \approx_{\mathscr{R}} h[\overline{\kappa'}] \triangleright (\overline{\ell} \vDash \mathscr{C}_1);\Sigma}$$

$$\textsc{Flexible-Same}\quad \frac{h \in \mathscr{C} \qquad \Phi_0 \models \overline{\ell} = \overline{\kappa} \triangleright \Phi_1}{\Phi_0;\Sigma;\Gamma \vdash h[\overline{\ell}] \approx_{\mathscr{R}} h[\overline{\kappa}] \triangleright \Phi_1;\Sigma}$$

$$\textsc{Univ-Eq}\quad \frac{\Phi \models i = j}{\Phi \models i = j \triangleright \Phi}$$

$$\textsc{Univ-Flexible}\quad \frac{i_{\mathtt{f}} \vee j_{\mathtt{f}} \in \overline{\ell} \qquad \mathscr{C} \wedge i = j \models}{(\overline{\ell} \vDash \mathscr{C}) \models i = j \triangleright (\overline{\ell} \vDash \mathscr{C} \wedge i = j)}$$

Fig. 22. Unification of universe polymorphic terms.

distinguish flexible levels from rigid ones. Each universe polymorphic constant and inductive type in the global environment is universally quantified with its universe context.

$$\begin{aligned}
t,u,T,U &= \ \ldots \mid c[\overline{\ell}] \mid i[\overline{\ell}] \mid k[\overline{\ell}] && \textit{term and types}\\
\Phi &= \ \overline{\ell} \vDash \mathscr{C} && \textit{universe context}\\
E &= \ \cdot \mid c : \forall\Phi.\, T, E \mid c := t : \forall\Phi.\, T, E \mid I, E \mid \Phi, E && \textit{global environment}\\
I &= \forall\Phi,\Gamma.\ \{\ \overline{i : \forall\overline{y : T_h}.\ s := \{k_1 : U_1; \ldots; k_n : U_n\}}\ \} && \textit{inductive types}
\end{aligned}$$

As before, a universe context consists of a list of levels $\overline{\ell}$ and a set of constraints $\mathscr{C}$ on levels. But now levels in $\overline{\ell}$ are annotated as 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, like ℓ and κ in pidpid above, while rigid ones correspond to user-specified levels or Type annotations.

The δE expansion rule must take into account universe levels, replacing those levels defined in the environment with the ones applied to the constant:

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

Reduction of pattern-matching and fixpoint constructs is easily extended:

$$\mathbf{match}_T\ k_j[\overline{\kappa}]\ \overline{t}\ \mathbf{with}\ \overline{k\,\overline{x} \Rightarrow u}\ \mathbf{end}\ \rightsquigarrow_\iota\ u_j\{\overline{t/x_j}\} \qquad \frac{F = \overline{x/n : T := t} \qquad a_n = k_j[\overline{\kappa}]\ \overline{t}}{\mathbf{fix}_j\ \{F\}\ \overline{a}\ \rightsquigarrow_\iota\ t_j\{\overline{\mathbf{fix}_m\ \{F\}/x_m}\}\ \overline{a}}$$

As binding of universes happens only at the global level (constants or inductives), local reduction rules do not need to substitute universes.

We extend the algorithm to consider universe polymorphic terms. Figure 22 shows the new and updated rules. Rule Type-Same is equal to the one in Section 6, but considering the new form of universe contexts. Rigid-Same equates the same 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 (Section 5), which will generaly result in weaker, more general constraints to be enforced.

Following is a simple example showing how backtracking ensures weaker constraints when universes cannot be matched:

Example 7 (Unfolding ensures weaker constraints)**.**

$$\textbf{Definition}\ \mathsf{weaker} : \mathsf{id}\ \_\ T := (\mathsf{nat} : \mathsf{id}\ \_\ \mathsf{Set}).$$

where T : $\mathsf{Type}(i)$ *for* $i > 0$*, and* Set *is in* Coq *the name for (predicative)* $\mathsf{Type}(0)$.

This definition requires the solution to the following equation:

$$\mathsf{id}\ \mathsf{Type}(\ell)\ \mathsf{Set} \approx_{\leq} \mathsf{id}\ \mathsf{Type}(\kappa)\ T$$

where ℓ and κ are universe levels introduced at elaboration for the two _ in the definition.

The rule App-FO (*c.f.*, Section 6) compares the heads using cumulativity, and the arguments using conversion:

$$\begin{aligned} \mathsf{id} &\approx_{\leq} \mathsf{id} && (1)\\ \mathsf{Type}(\ell) &\approx_{\equiv} \mathsf{Type}(\kappa) && (2)\\ \mathsf{Set} &\approx_{\equiv} T && (3) \end{aligned}$$

The first one is solved immediately. The second one forces ℓ to be equal to κ. But in the third one the algorithm fails, because it cannot ensure that Set and T are equal. Backtracking and unfolding both sides of the equation we obtain the weaker equation:

$$\mathsf{Set} \approx_{\leq} T$$

which is now solvable.

To conclude this section, note that this example was about the monomorphic id function. What would be different if instead we use the polymorphic pid function? In essence, the algorithm does the same unfoldings as with id to solve the problem, although it does so without comparing the arguments. When comparing the head constants, which are now applied to different flexible variables j and j', rule Flexible-Same cannot enforce the equality of j and j' because that requires Set and T being equal. Therefore, it immediately backtracks and unfolds, as before.

12 Rule Priorities and Backtracking

The rules shown across the different sections 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 one is attempted). 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 (Reduce-Same).

Except where noted, the algorithm tries always to perform a step in the right-hand side prior to the left-hand side. The reason is merely practical: the r.h.s. is often the term from the goal, while the l.h.s. is the one provided by the user. The algorithm tries to keep the term given by the user as close as possible to what she wrote.

For the ordering of the rules involving meta-variables, we refer the reader to Section 10.

1. If a term has a *defined* meta-variable in its head position, its definition is exposed:
   - (a) Meta-δR, Meta-δL
2. If the heads of both terms are *the same* (undefined) meta-variable:
   - (a) Meta-Same-Same, Meta-Same
3. If the heads of both terms are *different* (undefined) meta-variables:
   - (a) If the suspended substitution of the meta-variable on the left is larger than the one on the right:
     Meta-InstL | Meta-InstR | Meta-FOL | Meta-FOR |
     Meta-DelDepsL | Meta-DelDepsR
   - (b) Otherwise:
     Meta-InstR | Meta-InstL | Meta-FOR | Meta-FOL |
     Meta-DelDepsR | Meta-DelDepsL
4. If one term has an *undefined* meta-variable, and the other term does not have a meta-variable in its head position:
   Meta-InstR | Meta-FOR | Meta-DelDepsR | Meta-ReduceR |
   Lam-ηR | Meta-InstL | Meta-FOL | Meta-ReduceL |
   Meta-DelDepsL | Lam-ηL
5. Else:
   - (a) If the two terms are not the same constant, it searches for a canonical instance:
     - 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 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

     Constants are unfolded after any other reduction rule (except η-expansion) for performance reasons, and to avoid missing canonical instances (*c.f.*, Section 9). Similarly, the reason for delaying η-expansion as much as possible is two-fold: it is a costly operation (since it must ensure the term η-expanded is a product, *c.f.*, Section 4), and it might prevent the use of a canonical instance, for instance *hiding* a constant under a λ-abstraction.

13 A Deliberate Omission: Constraint Postponement

The technique of *constraint postponement* (Dowek *et al.*, 1996; Reed, 2009) 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 (Gonthier *et al.*, 2013a; Ziliani *et al.*, 2015), 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 (Section 14) show that this technique is not crucial.

14 Evaluation of the Algorithm

Since, as we saw in Section 13, 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[4], which, when requested, changes the current unification algorithm 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) (Chlipala, 2011). 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.*, Asperti *et al.*, 2012) enabled by COQ's **Program** keyword. For instance, some lines construct witnesses for existential quantification, similar to the example shown in Section 13.

The third one is the Mathematical Components library (Gonthier *et al.*, 2008), version 1.6. This library presents several challenges, making it appealing for our purpose: (1) It is a huge development, with a total of 87 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 almost 85,000 lines in the library, less than 30 lines required changes.[5]

The last set of files also focuses in different canonical structures idioms: the files from Lemma Overloading (Gonthier *et al.*, 2013a). It compiles almost as-is, with only one line requiring an extra annotation.

The little extra annotations required in these libraries allow us to conclude that our set of heuristics is a reasonable one.

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

[5] The modified files of the library can be downloaded from `https://github.com/unicoq/math-comp/tree/unicoq`

15 Correctness of the algorithm

In the literature there are usually two things to say about the correctness of a unification algorithm. The first one is to characterize the set of solutions, which usually involves proving that the algorithm generates Most General Unifiers (MGUs). However, as we mentioned throughout this work, in Coq we do not care about MGUs, since that will render the algorithm pretty much useless. Several useful heuristics presented here pick arbitrary yet sensible solutions.

The second thing one might want to prove is the following correctness criterion:

Conjecture 1 (Correctness criterion for unification)**.** *Let Φ,Σ, and Γ be a universe context, a meta-context, and a local context, and let t_1 and t_2 be two well-typed terms and T_1 and T_2 its types, i.e.,*

$$\Phi;\Sigma;\Gamma \vdash t_i : T_i \qquad \textit{for } i \in [1,2]$$

and such that they unify under relation $\mathscr{R}$:

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

then t_1 and t_2 are well-typed in the new contexts

$$\Phi';\Sigma';\Gamma \vdash t_i : T_i \qquad \textit{for } i \in [1,2]$$

and are convertible under relation $\mathscr{R}$.

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, resulting in an ill-typed term correctly rejected by the kernel typechecker.

The following example illustrates this point. It is real Coq code annotated with the names of the meta-variables used in the derivation tree shown in Figure 23.

Example 8 (Proof of False—rejected by the kernel)**.**

```
Definition False_proof : False :=
  let h : (nat → False) → nat → False := _ (*?X1*) in
  let T := fix f (x:nat):False := h f x in
  let _ : h = @id (nat → False) := eq_refl _ (*?X4*) in
  T 0.
```

It creates a fixpoint f with a meta-variable h ($?X_1$) applied to f. Later on h is instantiated with the identity function, therefore tying the knot. In the code the “@” symbol is notation in Coq to explicitly provide every implicit argument, in this case the polymorphic type of the identity function. eq_refl is the proof of reflexivity.

Hence, we must weaken this conjecture to use a weaker notion of typing, as in Coen’s thesis (Sacerdoti Coen, 2004), or restrict unification so that any fixpoint term given to it is closed w.r.t. meta variables and guarded.

For the moment we lack a correctness proof, which we are attempting directly in Coq. This work sets the first stone presenting a specification faithful to an implementation that

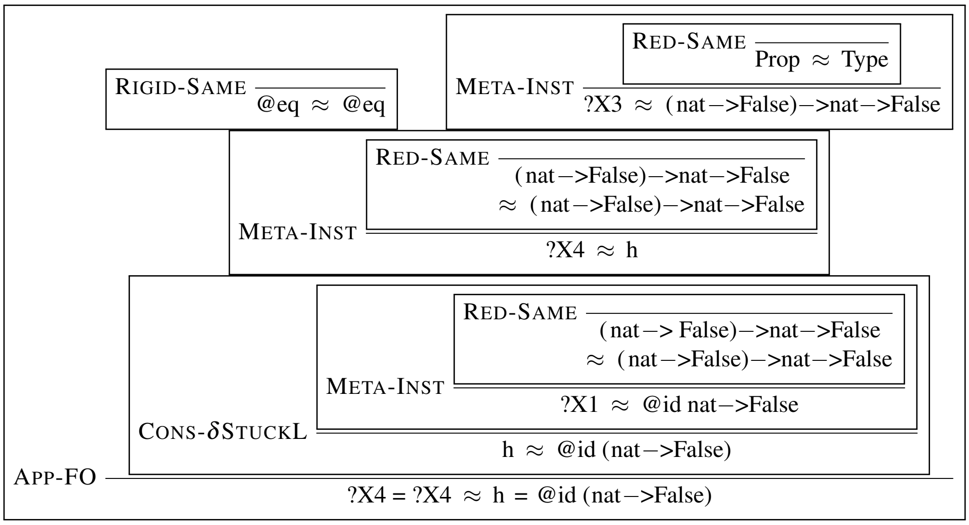

Fig. 23. Instantiation of the body of a fixpoint with a non-structurally-recursive term.

performs well on a variety of large examples (Section 14). We anticipate that, once the basic theory is set, the proof will be simpler than for existing algorithms, notably due to the lack of postponement which usually complicates the argument of type preservation.

16 Lemma Overloading: proof search during unification

In different examples we saw how unification was in charge of "filling in" missing bits of (proof) terms. For instance, in Example 4 the algorithm completed the missing lists in the proof of list membership. We also saw in Example 6 how the algorithm is capable of finding the missing *code* for the equality *function*, based on the *type* of its operands. It was just a matter of time to realize that it is also possible to find a *proof* for a *lemma* based on the *terms* (or types) of its arguments, thanks to the proofs-as-programs concept in which CIC is based, together with the lack of distinction between the syntactical classes of terms and types.

Gonthier *et al.* (2011, 2013a) developed this concept, which they called *Lemma Overloading*. In particular, they showed how to tackle certain limitations of Canonical Structures to transform the unification algorithm into an ad-hoc proof search engine. In this section we show some of the main ideas in Gonthier *et al.* (2013a), focusing on the aspects of unification that makes Lemma Overloading possible.

We develop an example, again from list membership. Although not a very interesting problem on its own, it already allows us to present Lemma Overloading without needing new concepts. For more engaging and realistic problems, we invite the reader to read Gonthier *et al.* (2013a).

Let us look again at Example 4. There we were proving that

$$y_1 \in ([y_1] +\!\!+ [y_2])$$

```
Structure listTag := ListTag { luntag : list nat }.
Definition tailTag := ListTag.
Definition foundTag := tailTag.
Definition leftTag := foundTag.
Canonical rightTag l := leftTag l.
```

Fig. 24. A tagging structure for lists.

```
Structure search x := Search {
  list_of : listTag;
  proof : In x (luntag list_of)
}.

Canonical tail_proof x y (f : search x) :=
  Search x (tailTag (y :: luntag (list_of f))) (in_tail _ _ _ (proof f)).
Canonical found_proof x s :=
  Search x (foundTag (x :: s)) (in_head _ _).
Canonical left_proof x r (f : search x) :=
  Search x (leftTag (luntag (list_of f) ++ r)) (inL _ _ _ (proof f)).
Canonical right_proof x l (f : search x) :=
  Search x (rightTag (l ++ luntag (list_of f))) (inR _ _ _ (proof f)).
```

Fig. 25. Structure to create the overloaded lemma for list membership.

by providing the proof term

inL _ _ _ (in_head _ _)

We relied on unification to instantiate all the meta-variables produced by the different holes (_) in the term. Now, would it be too much to ask for to also get the whole proof? This is what Lemma Overloading is about.

We will proceed to explain this technique writing the necessary Coq code to solve this problem. At high level, we will create structures and canonical instances to build a search procedure that will look for an element x in a list s, on the way computing the proof of $x \in s$. (As it turns out, it will be a *dependently-typed logic* program.) This program will do casing on the list: if it is a concatenation of two lists l and r, it will first search for x on r, and if it is not there, in l. If the list is the consing of element y and list l, it will first check if x is equal to y and, if not, look for x in l. Note that this corresponds precisely to each of the list axioms presented in Figure 1.

The code will look confusing at first, but it should become clear once we tight these ideas with the heuristics shown in previous sections. We start introducing a *tagging* structure (Figure 24), crucial to distinguish the different cases of the algorithm. It consists of a structure listTag with only one field, in this case a list, named luntag. It is accompanied with a chain of definitions: rightTag is defined as leftTag, which is itself defined as foundTag, and so on, until we arrive at the constructor ListTag of the structure. The top-most definition, rightTag is made **Canonical**, adding the triple (luntag, _, rightTag) to Δ_{db}, the canonical structures database.

Then, we encode the search procedure in the structure search displayed in Figure 25. This structure is parametrized over the element we are looking for, x, and contains two fields: list_of, a listTag, and the proof of x being in the *untagged* list. This field is our *overloaded lemma*.

This structure contains one canonical instance for each case of the procedure. Each instance is constructed using one of the *tags* defined in Figure 24: the instance tail_proof, which constructs a proof using axiom in_tail, is constructed using tag tailTag; the instance found_proof, which constructs a proof using axiom in_head, uses foundTag; and so on. This has the effect of inserting in the Δ_{db} the triples (list_of, tailTag, tail_proof), (list_of, foundTag, found_proof), etc. And this is the key to understand the idea behind *tagging* the list: the database cannot be populated with the same key twice, and our example requires two instances for consing and two for appending. With the tags, we managed to create different keys, one for each of the instances.

With these definitions, we are now able to prove Example 4 simply writing:

Example 9 (Proving list membership using an overloaded lemma)**.**

$$\textbf{Definition}\ \mathsf{excs} : y_1 \in ([y_1] +\!\!+ [y_2]) := \mathsf{proof}\ \_.$$

We can provide an accurate description of what is going on under the hood to make this proof possible, thanks to the rules provided in previous sections (most notably, sections 8, 9, and 12). We will omit the suspended substitutions in meta-variables, as they play no role in the example (every meta-variable instantiation will follow the higher-order pattern restriction).

The proof search starts when the type of proof _ gets equated with the type of the example:

$$?x \in (\mathsf{luntag}\ (\mathsf{list\_of}\ ?f)) \approx y_1 \in ([y_1] +\!\!+ [y_2]) \tag{10}$$

where $?f$ is the implicit structure (_) that the Canonical Structures mechanism must instantiate. After Equation 10, rule App-FO is triggered, obtaining two sub-equations:

1. $?x \approx y_1$
2. $\mathsf{luntag}\ (\mathsf{list\_of}\ ?f) \approx [y_1] +\!\!+ [y_2]$

The first one is solved immediately with Meta-Inst, instantiating $?x$ with y_1. For the second one the algorithm must find a canonical instance to solve it. The algorithm first tries CS-Const but fails: there is no key pairing luntag with ++. Before giving up, it tries CS-Default, which now finds that there is a default key (luntag, _, rightTag). This rule equates the argument of luntag with the instance rightTag applied to the term on the r.h.s. $([y_1] +\!\!+ [y_2])$:

$$\mathsf{list\_of}\ ?f \approx \mathsf{rightTag}\ ([y_1] +\!\!+ [y_2]) \tag{11}$$

Now we have again a projector of a structure on the l.h.s. and a constant on the r.h.s., triggering rule CS-Const. It finds key (list_of, rightTag, right_proof), and proceeds to perform the following actions, in order:

1. Generates a fresh meta-variable for each argument of right_proof: $?x_1, ?l, ?f_1$.
2. Equates the parameters of right_proof to the parameters of $?f$:

$$?x_1 \approx y_1$$

3. Equates the arguments of rightTag in right_proof with those in Equation 11:

$$?l +\!\!+ (\mathsf{luntag}\ (\mathsf{list\_of}\ ?f_1)) \approx [y_1] +\!\!+ [y_2]$$

4. Equate the the argument of list_of in Equation 11 with the new instance:

$$?f \approx \mathsf{right\_proof}\ ?x_1\ ?l\ ?f_1$$

Step 2 is solved immediately with META-INST. Step 3, thanks to APP-FO, will first assign $[y_1]$ to $?l$, and then equate:

$$\mathsf{luntag}\ (\mathsf{list\_of}\ ?f_1) \approx [y_2]$$

Again, because of CS-DEFAULT we obtain the equation

$$\mathsf{list\_of}\ ?f_1 \approx \mathsf{rightTag}\ [y_2] \tag{12}$$

Which, again using CS-CONST, generates equation

$$?l_1 +\!\!+ \mathsf{luntag}\ (\mathsf{list\_of}\ ?f_2) \approx [y_2]$$

for fresh $?l_1$ and $?f_2$ (Step 3 above). Now the algorithm tries APP-FO, but it fails when comparing the heads (:: and $+\!\!+$). Before giving up, it unfolds the definition of $+\!\!+$ (CONS-δL[6]), only to find a pattern matching matching on the meta-variable $?l_1$.

Backtracking a bit, it considers again Equation 12 and notices it can δ-reduce the head constants at either side of the equation. But which one? Here is where the hypothesis of being stuck comes in handy: on the l.h.s. there is a projection of meta-variable $?f_1$, which is therefore *stuck*. If it unfolds that definition, the algorithm will miss opportunities for finding more plausible canonical instances. Instead, it proceeds to unfold the r.h.s. (CONS-δNOTSTUCKR), discovering a new constant:

$$\mathsf{list\_of}\ ?f_1 \approx \mathsf{leftTag}\ [y_2] \tag{13}$$

At this point the algorithm tries again CS-CONST, finds triple (list_of, leftTag, left_proof), repeating the steps mentioned above for right_proof. But it soon finds out that the head constant is not a concatenation. Fast-forwarding a bit, it considers again Equation 13, unfolds the r.h.s., obtaining equation

$$\mathsf{list\_of}\ ?f_1 \approx \mathsf{foundTag}\ [y_2] \tag{14}$$

This time it will try to use instance found_proof, but since y_2 is not the element we are looking for (y_1, the parameter of $?f_1$), it fails again. After unfolding the r.h.s. once more, we get tailTag, and the process is repeated only to find out the element was not there after all.

Backtracking again, we arrive at Equation 11. Using CONS-δNOTSTUCKR it unfolds rightTag to get leftTag. The whole process is repeated, this time finding the element on the list on the left. The whole successful derivation tree is shown in Figure 26 where, for the sake of space, we removed the unification of types in the rule META-INST, trivial in this example, and we renamed the rules: MI for META-INST, R-SAME for REDUCE-SAME, FO for APP-FO, δNS for CONS-δNOTSTUCKR, and CS for any of the rules for Canonical Structures.

[6] It is interesting to note that both sides of the equation are *stuck*.

The final result can be traced following the names of the meta-variables:

$$\begin{aligned}
\text{In}\ \ ?\text{X1}\ (\text{luntag}\ \ (\ \text{list\_of}\ \ \ ?\text{X2})) &\approx \text{In}\ \ \text{y1}\ \ ([\text{y1}]\ \ \text{++}\ [\text{y2}]) \\
?\text{X1} &\approx \text{y1} \\
?\text{X2} &\approx \ \text{left\_proof}\ \ \ ?\text{X13}\ ?\text{X14}\ ?\text{X15} \\
?\text{X13} &\approx \text{y1} \\
?\text{X14} &\approx [\text{y2}] \\
?\text{X15} &\approx \text{found\_proof}\ \ ?\text{X32}\ ?\text{X33} \\
?\text{X32} &\approx \text{y1} \\
?\text{X33} &\approx []
\end{aligned}$$

The l.h.s. $\delta\Sigma$-normalizes to

$$\text{In}\ \ \text{y1}\ \ (\text{luntag}\ \ (\ \text{list\_of}\ \ \ (\ \text{left\_proof}\ \ \ \text{y1}\ \ [\text{y2}]\ \ (\text{found\_proof}\ \ \text{y1}\ \ \ [])))))$$

The proof is therefore

$$\text{proof}\ \ (\ \text{left\_proof}\ \ \ \text{y1}\ \ [\text{y2}]\ \ (\text{found\_proof}\ \ \text{y1}\ \ \ []))$$

which effectively normalizes to the proof the user wrote.

17 Related work

The first formal introduction of the problem of unification is due to Robinson (1965), 50 years ago, making the task of listing related work on the area a rather dull and daunting task. Instead, we focus our attention on a set of works that inspired our work, in the narrower area of higher-order unification, and refer the reader to different books and surveys (Knight, 1989; Baader & Siekmann, 1994; Baader & Nipkow, 1998; Huet, 2002).

Most of the work in the literature focuses on obtaining Most General Unifiers, something we purposely avoid for the sake of usability. That makes our work quite unique. Nevertheless, we will list several works that are somehow related to ours.

We mentioned already Pfenning (1991). It presents a unification algorithm for the Calculus of Constructions, but without introducing definitions (as in Section 5), and only unifying β-normal terms. The unification of meta-variables presented in Section 3 is similar to the one presented in this work.

Definitions were added to the aforementioned work in Pfenning & Schürmann (1998), taking particular care of when to δ-unfold constants. More precisely, they consider a class of definitions which are *strict*; a semantic subclass of terms in which *injectivity* is guaranteed, that is, it is valid that

$$c\,t\ \approx\ c\,u \implies t\ \approx\ u$$

and therefore if $t \not\approx u$, then the algorithm fails without unfolding c. Our algorithm always unfolds constants, therefore potentially considering again the unification of t and u, which can be a major performance bottleneck. But it is not so easy to port the ideas from Pfenning & Schürmann (1998) to our setting, most notably because of Canonical Structures resolution (see *e.g.*, Section 16). Ultimately, it might be just a case of narrowing the notion of *strict* terms, although if it ends up being *too* narrow it might end up pretty much useless.

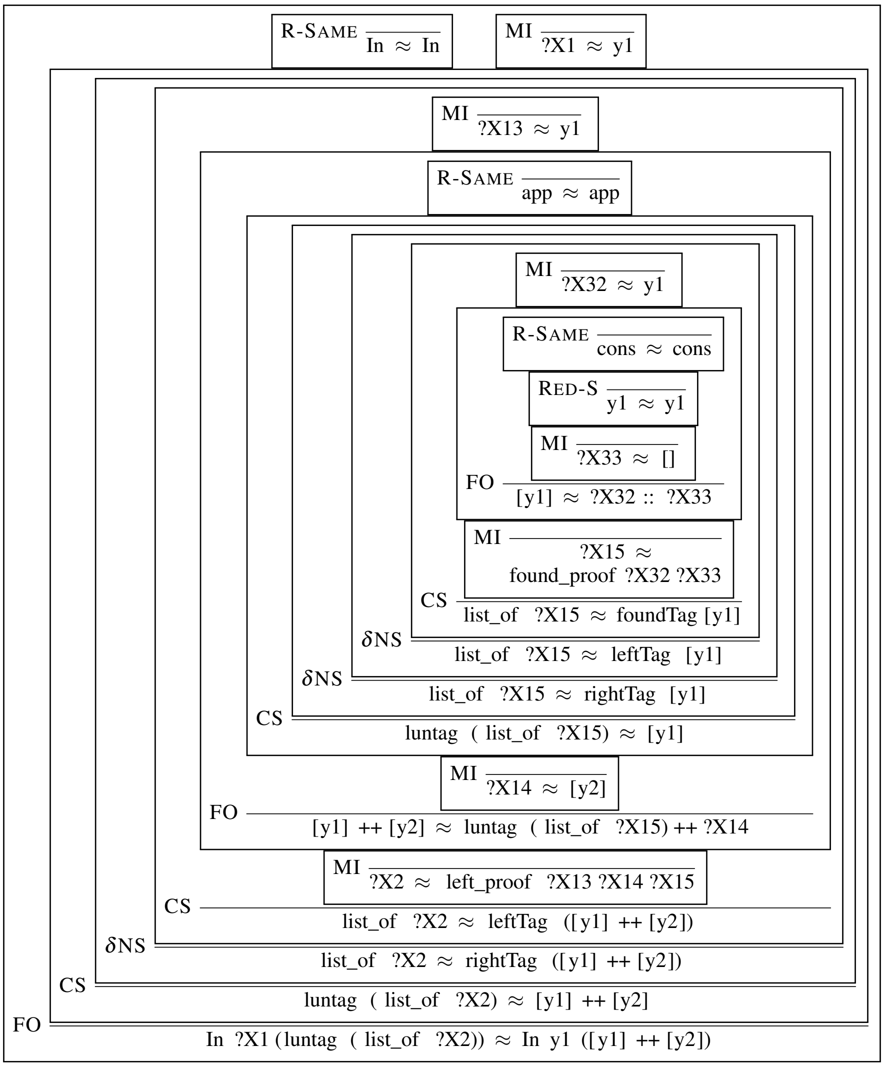
R-SAME In ≈ In
MI ?X1 ≈ y1
MI ?X13 ≈ y1
R-SAME app ≈ app
MI ?X32 ≈ y1
R-SAME cons ≈ cons
RED-S y1 ≈ y1
MI ?X33 ≈ []
FO [y1] ≈ ?X32 :: ?X33
MI ?X15 ≈ found_proof ?X32 ?X33
CS list_of ?X15 ≈ foundTag [y1]
δNS list_of ?X15 ≈ leftTag [y1]
δNS list_of ?X15 ≈ rightTag [y1]
CS luntag (list_of ?X15) ≈ [y1]
MI ?X14 ≈ [y2]
FO [y1] ++ [y2] ≈ luntag (list_of ?X15) ++ ?X14
MI ?X2 ≈ left_proof ?X13 ?X14 ?X15
CS list_of ?X2 ≈ leftTag ([y1] ++ [y2])
δNS list_of ?X2 ≈ rightTag ([y1] ++ [y2])
CS luntag (list_of ?X2) ≈ [y1] ++ [y2]
FO In ?X1 (luntag (list_of ?X2)) ≈ In y1 ([y1] ++ [y2])

Fig. 26. Derivation tree of an overloaded lemma in action.

Some of the problems adapting Huet's algorithm to a richer language with dependent types were discussed in Elliott (1989).

Dowek *et al.* (1996) introduced constraint postponement, although with a subtle error making the algorithm non-terminating. This work was fixed by Reed (2009). In a sense, our work goes in the opposite direction, forbidding constraint postponement (Section 13) and fixating solutions where multiple solutions exists (rule META-DELDEPS, Section 10). We must note that our algorithm might non-terminate on certain inputs. Firstly, because the language allows for fixpoints, which are hard to check for termination in the presence of meta-variables (Section 15), and secondly because Canonical Structures incorporates a Turing complete machine to the unification algorithm (as a matter of fact, the first interpreter of the language Mtac (Ziliani *et al.*, 2013, 2015) was created using Canonical Structures!).

The pruning judgment, the intersection judgment, and the inversion of substitution are modified versions of those in Abel & Pientka (2011). Abel & Pientka (2011) presents an algorithm for unification for $\lambda^{\Pi\Sigma}$, with the novelty of performing η-expansion for Σ-types. We have not considered yet the inclusion of such rule.

Canonical Structures were introduced in Saïbi (1999, chap. 4), although at a much higher level and with a different order in which subproblems are considered. Matita's *hints* are a similar concept developed by Asperti *et al.* (2009).

The elaboration mechanism for the Lean theorem prover is presented in de Moura *et al.* (2015), putting special emphasis on the different mechanisms, such as overloading and the unification algorithm. With respect to unification, they restrict themselves to a somewhat naive, Huet-style algorithm. They claim they do not require the several heuristics presented in our work, in particular stressing that the simple representation of meta-variables like the one presented in Section 3 suffices for their needs. We think that the richer approach of using contextual types for meta-variables, used in several of the aforementioned works and in ours, allows for useful heuristics like META-DELDEPS (Section 10), but ultimately more study on the trade-offs of each representation should be performed.

For λProlog Dunchev *et al.* (2015) created recently a fast interpreter, which includes a fast HO-unification algorithm. The key insight of this work is to note that there is a large fraction of λProlog programs that admits linear time unification. An important design decision when building the interpreter was to realize that de Bruijn *levels* (in opposition to the commonly used de Bruijn *indices*) has better properties for a fast unification algorithm. We based our work in the current implementation of COQ, and therefore we did not explore different representations of terms (COQ's internal representation of terms is using de Bruijn indices). It might be worth the effort to study optimizations like the ones proposed in this work.

Universe polymorphism was first introduced by Harper & Pollack (1991) where they study a variant of the calculus of constructions with universe polymorphism, definitions and typical ambiguity. The version implemented in COQ Sozeau & Tabareau (2014) removes some of the restrictions in that work (e.g. variables couldn't be polymorphic), and changes the philosophy of the system to allow polymorphism everywhere, in particular not unfolding polymorphic definitions in types, which would be very expensive in practice. As we saw, this requires a subtle approach during unification (§11).

When it comes to the verification and the correct construction of a unification algorithm, Paulson (1985) provides a formalization of the algorithm in LCF's at the time. More recently, Vezzosi[7] formalized in Agda a simple algorithm featuring Higher-Order pattern unification. In Cockx *et al.* (2016) the algorithm of Agda is replaced by an algorithm that produces evidence showing that the two terms being unified are indeed equal. Every new rule added to the algorithm is required to produce such evidence, effectively raising the level of confidence in the algorithm. It must be noted, though, that this evidence is not enough to fully trust the output of the algorithm, as they say nothing about how the solution was constructed.

[7] `https://github.com/Saizan/miller`

18 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* (Section 9), which, together with overloading (Section 8), allow us to explain the patterns introduced in Gonthier *et al.* (2013a) (Section 16). 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 (Section 13). This omission, made in favor of predictability, has shown not to be problematic in practice (Section 14).

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 (Section 10.4).

The ideas presented in this work were built from the ground up, starting from the basic Calculus of Constructions (Section 3) up to the full Calculus of Inductive Constructions implemented by COQ (sections 7, 8, 9, 10, and 11).

In the future we plan to prove soundness of the algorithm (see Section 15), 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 for their important feedback on earlier versions of this work. We are also thankful to the anonymous reviewers of the ICFP'15 paper for their input. This research was partially supported by EU 7FP grant agreement 295261 (MEALS).

Bibliography

Abel, Andreas, & Pientka, Brigitte. (2011). Higher-order dynamic pattern unification for dependent types and records. *International conference on typed lambda calculi and applications (TLCA)*. Springer.

Asperti, Andrea, Coen, Claudio Sacerdoti, Tassi, Enrico, & Zacchiroli, Stefano. (2006). Crafting a proof assistant. *TYPES*. Springer-Verlag.

Asperti, Andrea, Ricciotti, Wilmer, Coen, Claudio Sacerdoti, & Tassi, Enrico. (2009). Hints in unification. *TPHOLs*. LNCS, vol. 5674. Springer.

Asperti, Andrea, Ricciotti, Wilmer, Coen, Claudio Sacerdoti, & Tassi, Enrico. (2012). A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical methods in computer science (LMCS)*, **8**(1).

Baader, Franz, & Nipkow, Tobias. (1998). *Term rewriting and all that*. New York, NY, USA: Cambridge University Press.

Baader, Franz, & Siekmann, Jörg H. (1994). Handbook of logic in artificial intelligence and logic programming. New York, NY, USA: Oxford University Press, Inc.

Brady, Edwin. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming (JFP)*, **23**.

Cervesato, Iliano, & Pfenning, Frank. (2003). A linear spine calculus. *Journal of logic and computation*, **13**(5), 639–688.

Chlipala, Adam. (2011). *Certified programming with dependent types*. MIT Press. `http://adam.chlipala.net/cpdt/`.

Cockx, Jesper, Devriese, Dominique, & Piessens, Frank. (2016). Unifiers as equivalences: Proof-relevant unification of dependently typed data. *Pages 270–283 of: Proceedings of the 21st ACM sigplan international conference on functional programming*. ICFP 2016. New York, NY, USA: ACM.

de Moura, L., Avigad, J., Kong, S., & Roux, C. (2015). Elaboration in Dependent Type Theory. *Arxiv e-prints*, May.

Dowek, Gilles, Hardin, Therese, Kirchner, Claude, & Pfenning, Frank. (1996). Unification via explicit substitutions: The case of higher-order patterns. *Pages 36637–4 of: Proceedings of lics'95*. IEEE Computer Society Press.

Dunchev, Cvetan, Guidi, Ferruccio, Sacerdoti Coen, Claudio, & Tassi, Enrico. (2015). Elpi: Fast, embeddable, λprolog interpreter. *Pages 460–468 of:* Davis, Martin, Fehnker, Ansgar, McIver, Annabelle, & Voronkov, Andrei (eds), *Logic for programming, artificial intelligence, and reasoning*. Lecture Notes in Computer Science, vol. 9450. Springer Berlin Heidelberg.

Elliott, Conal M. (1989). Higher-order unification with dependent function types. *Pages 121–136 of: 3rd int. conf. rewriting techniques and applications, lncs 355*. Springer-Verlag.

Garillot, François. 2011 (Dec.). *Generic Proof Tools and Finite Group Theory*. Ph.D. thesis, Ecole Polytechnique X.

Garillot, François, Gonthier, Georges, Mahboubi, Assia, & Rideau, Laurence. (2009). Packaging Mathematical Structures. *TPHOL*. Springer.

Gonthier, Georges, Mahboubi, Assia, & Tassi, Enrico. (2008). *A small scale reflection extension for the Coq system*. Tech. rept. INRIA.

Gonthier, Georges, Ziliani, Beta, Nanevski, Aleksandar, & Dreyer, Derek. (2011). How to make ad hoc proof automation less ad hoc. *Inernational conference of functional programming (ICFP)*.

Gonthier, Georges, Ziliani, Beta, Nanevski, Aleksandar, & Dreyer, Derek. (2013a). How to make ad hoc proof automation less ad hoc. *Journal of functional programming (JFP)*, **23**(04), 357–401.

Gonthier, Georges, Asperti, Andrea, Avigad, Jeremy, Bertot, Yves, Cohen, Cyril, Garillot, François, Le Roux, Stéphane, Mahboubi, Assia, O'Connor, Russell, Ould Biha, Sidi, Pasca, Ioana, Rideau, Laurence, Solovyev, Alexey, Tassi, Enrico, & Théry, Laurent. (2013b). A machine-checked proof of the odd order theorem. *ITP*. Springer.

Harper, Robert, & Pollack, Robert. (1991). Type checking with universes. *Theor. comput. sci.*, **89**(1), 107–136.

Huet, Gérard P. (2002). Higher order unification 30 years later. *Pages 3–12 of: Proceedings of the 15th international conference on theorem proving in higher order logics*. TPHOLs '02. London, UK, UK: Springer-Verlag.

Knight, Kevin. (1989). Unification: A multidisciplinary survey. *Acm comput. surv.*, **21**(1), 93–124.

Mahboubi, Assia, & Tassi, Enrico. (2013). Canonical Structures for the working Coq user. *ITP*. Springer.

Miller, Dale. (1991). Unification of simply typed lamda-terms as logic programming. *ICLP*. MIT Press.

Nanevski, Aleksandar, Pfenning, Frank, & Pientka, Brigitte. (2008). Contextual modal type theory. *ACM Trans. Comput. Logic*, **9**(3).

Norell, Ulf. (2009). Dependently Typed Programming in Agda. *Types in language design and implementation (TLDI)*. ACM.

Paulson, Lawrence C. (1985). Verifying the unification algorithm in lcf. *Sci. comput. program.*, **5**(2), 143–169.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Washburn, Geoffrey. (2006). Simple unification-based type inference for GADTs. *Inernational conference of functional programming (ICFP)*. ACM.

Pfenning, Frank. (1991). Unification and anti-unification in the calculus of constructions. *Pages 74–85 of: Sixth annual ieee symposium on logic in computer science*.

Pfenning, Frank, & Schürmann, Carsten. (1998). Algorithms for equality and unification in the presence of notational definitions. *Page 1657 of: Types for proofs and programs*. Springer-Verlag LNCS.

Reed, Jason. (2009). Higher-order constraint simplification in dependent type theory. *Logical frameworks and meta languages: Theory and practice (LFMTP)*.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, **12**(1), 23–41.

Sacerdoti Coen, Claudio. (2004). *Mathematical knowledge management and interactive theorem proving*. Ph.D. thesis, University of Bologna.

Saïbi, Amokrane. (1999). *Outils generiques de modelisation et de demonstration pour la formalisation des mathematiques en theorie des types. application a la theorie des categories*. Ph.D. thesis, University Paris 6.

Sozeau, Matthieu, & Tabareau, Nicolas. (2014). Universe Polymorphism in Coq. *International conference on interactive theorem proving (ITP)*. Springer.

The Coq Development Team. (2012). *The Coq Proof Assistant Reference Manual – Version V8.4*. See `http://coq.inria.fr/V8.4/CREDITS`.

Wadler, Philip, & Blott, Stephen. (1989). How to make ad-hoc polymorphism less ad hoc. *Pages 60–76 of: ACM symposium on principles of programming languages (POPL)*.

Ziliani, Beta, & Sozeau, Matthieu. (2015). A unification algorithm for Coq featuring universe polymorphism and overloading. *Pages 179–191 of: Inernational conference of functional programming (ICFP)*. New York, NY, USA: ACM.

Ziliani, Beta, Dreyer, Derek, Krishnaswami, Neelakantan R., Nanevski, Aleksandar, & Vafeiadis, Viktor. (2013). Mtac: A monad for typed tactic programming in Coq. *Inernational conference of functional programming (ICFP)*.

Ziliani, Beta, Dreyer, Derek, Krishnaswami, Neel, Nanevski, Aleksandar, & Vafeiadis, Viktor. (2015). Mtac: A monad for typed tactic programming in Coq. *Journal of functional programming (JFP)*, **25**.

A The full unification algorithm

A.1 The language

$$
\begin{array}{rll}
t, u, T, U = & x \mid c[\overline{\ell}] \mid i[\overline{\ell}] \mid k[\overline{\ell}] \mid s \mid ?x[\sigma] & \textit{terms and types} \\
& \mid \forall x : T.\, U \mid \lambda x : T.\, t \mid t\ u \mid \textsf{\textbf{let}}\ x := t : T\ \textsf{\textbf{in}}\ u & \\
& \mid \textsf{\textbf{match}}_T\ t\ \textsf{\textbf{with}}\ k_1\ \overline{x_1} \Rightarrow t_1 \mid \ldots \mid k_n\ \overline{x_n} \Rightarrow t_n\ \textsf{\textbf{end}} & \\
& \mid \textsf{\textbf{fix}}_j\ \{x_1/n_1 : T_1 := t_1; \ldots; x_m/n_m : T_m := t_m\} & \\
\sigma = & \bar{t} & \textit{suspended substitutions} \\
s = & \mathsf{Type}(\overline{K}^{+}) & \textit{sorts} \\
K = & \kappa \mid K + 1 & \\
\ell, \kappa \in & \mathbb{N} \cup 0^{-} & \textit{universe levels}
\end{array}
$$

$$
\begin{array}{rll}
\Phi = & \overline{\ell} \vDash \mathscr{C} & \textit{universe contexts} \\
\mathscr{C} = & \cdot \mid \mathscr{C} \wedge \ell\ \mathscr{O}\ \ell' \qquad \text{where } \mathscr{O} \in \{=, \leq, <\} & \textit{universe constraints} \\
\Gamma, \Psi = & \cdot \mid x : T, \Gamma \mid x := t : T, \Gamma & \textit{local contexts} \\
\Sigma = & \cdot \mid ?x : T[\Psi], \Sigma \mid ?x := t : T[\Psi], \Sigma & \textit{meta-contexts} \\
E = & \cdot \mid c : \forall \Phi.\, T, E \mid c := t : \forall \Phi.\, T, E \mid I, E \mid \Phi, E & \textit{global environment} \\
I = & \forall \Phi, \Gamma.\ \{\ \overline{i : \forall \overline{y : T_h}.\ s := \{k_1 : U_1; \ldots; k_n : U_n\}}\ \} & \textit{inductive types}
\end{array}
$$

A.2 Reduction rules

$$
(\lambda x : T.\, t)\ u \ \rightsquigarrow_\beta\ t\{u/x\} \qquad \textsf{\textbf{let}}\ x := u : T\ \textsf{\textbf{in}}\ t \ \rightsquigarrow_\zeta\ t\{u/x\} \qquad \frac{(x := t : T) \in \Gamma}{x \ \rightsquigarrow_{\delta\Gamma}\ t}
$$

$$
\frac{?x := t : T[\Psi] \in \Sigma}{?x[\sigma] \ \rightsquigarrow_{\delta\Sigma}\ t\{\sigma/\widehat{\Psi}\}} \qquad \frac{(c := t : \forall \overline{\ell} \vDash \mathscr{C}.\, T) \in E}{c[\overline{\kappa}] \ \rightsquigarrow_{\delta E}\ t[\overline{\kappa/\ell}]}
$$

$$
\textsf{\textbf{match}}_T\ k_j[\overline{\kappa}]\ \bar{t}\ \textsf{\textbf{with}}\ \overline{k\,\bar{x} \Rightarrow u}\ \textsf{\textbf{end}} \ \rightsquigarrow_\iota\ u_j\{\overline{t/x_j}\} \qquad \frac{F = \overline{x/n : T := t} \qquad a_n = k_j[\overline{\kappa}]\ \bar{t}}{\textsf{\textbf{fix}}_j\ \{F\}\ \bar{a} \ \rightsquigarrow_\iota\ t_j\{\overline{\textsf{\textbf{fix}}_m\ \{F\}/x_m}\}\ \bar{a}}
$$

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

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

A.3 Unification algorithm

Type-Same

$$\frac{\mathscr{C}' = \mathscr{C} \wedge \overline{u}\, \mathscr{R}\, \kappa \qquad \mathscr{C}' \vDash}{\ell \vDash \mathscr{C}; \Sigma; \Gamma \vdash \mathsf{Type}(\overline{u}) \approx_{\mathscr{R}} \mathsf{Type}(\kappa) \rhd \ell \vDash \mathscr{C}'; \Sigma}$$

Var-Same

$$\frac{}{\Phi; \Sigma; \Gamma \vdash x \approx_{\mathscr{R}} x \rhd \Phi; \Sigma}$$

Rigid-Same

$$\frac{h \in \mathscr{I} \cup \mathscr{K} \qquad \mathscr{C}_1 = \mathscr{C}_0 \wedge \overline{\kappa = \kappa'} \qquad \mathscr{C}_1 \models}{(\overline{\ell} \vDash \mathscr{C}_0); \Sigma; \Gamma \vdash h[\overline{\kappa}] \approx_{\mathscr{R}} h[\overline{\kappa'}] \rhd (\overline{\ell} \vDash \mathscr{C}_1); \Sigma}$$

Flexible-Same

$$\frac{h \in \mathscr{C} \qquad \Phi_0 \models \overline{\ell} = \overline{\kappa} \rhd \Phi_1}{\Phi_0; \Sigma; \Gamma \vdash h[\overline{\ell}] \approx_{\mathscr{R}} h[\overline{\kappa}] \rhd \Phi_1; \Sigma}$$

Univ-Eq

$$\frac{\Phi \models i = j}{\Phi \models i = j \rhd \Phi}$$

Univ-Flexible

$$\frac{i_{\mathfrak{f}} \vee j_{\mathfrak{f}} \in \overline{\ell} \qquad \mathscr{C} \wedge i = j \models}{(\overline{\ell} \vDash \mathscr{C}) \models i = j \rhd (\overline{\ell} \vDash \mathscr{C} \wedge i = j)}$$

Prod-Same, Lam-Same

$$\frac{\begin{array}{c}\Pi \in \{\lambda, \forall\} \\ \Phi_0; \Sigma_0; \Gamma \vdash T_1 \approx_{\equiv} U_1 \rhd \Phi_1; \Sigma_1 \qquad \Phi_1; \Sigma_1; \Gamma, x : T_1 \vdash T_2 \approx_{\mathscr{R}} U_2 \rhd \Phi_2; \Sigma_2\end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash \Pi x : T_1.\, T_2 \approx_{\mathscr{R}} \Pi x : U_1.\, U_2 \rhd \Phi_2; \Sigma_2}$$

Let-Same

$$\frac{\begin{array}{c}\Phi_0; \Sigma_0; \Gamma \vdash T \approx_{\equiv} U \rhd \Phi_1; \Sigma_1 \qquad \Phi_1; \Sigma_1; \Gamma \vdash t_2 \approx_{\equiv} u_2 \rhd \Phi_2; \Sigma_2 \\ \Phi_2; \Sigma_2; \Gamma, x := t_2 \vdash t_1 \approx_{\mathscr{R}} u_1 \rhd \Phi_3; \Sigma_3\end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash \mathbf{let}\ x := t_2 : T\ \mathbf{in}\ t_1 \approx_{\mathscr{R}} \mathbf{let}\ x := u_2 : U\ \mathbf{in}\ u_1 \rhd \Phi_3; \Sigma_3}$$

Case-Same

$$\frac{\begin{array}{c}\Phi_0; \Sigma_0; \Gamma \vdash T \approx_{\equiv} U \rhd \Phi_1; \Sigma_1 \\ \Phi_1; \Sigma_1; \Gamma \vdash t \approx_{\equiv} u \rhd \Phi_2; \Sigma_2 \qquad \Phi_2; \Sigma_2; \Gamma \vdash \overline{b} \approx_{\equiv} \overline{b'} \rhd \Phi_3; \Sigma_3\end{array}}{\Phi_0; \Sigma_0; \Gamma \vdash \mathbf{match}_T\ t\ \mathbf{with}\ \overline{b}\ \mathbf{end} \approx_{\mathscr{R}} \mathbf{match}_U\ u\ \mathbf{with}\ \overline{b'}\ \mathbf{end} \rhd \Phi_3; \Sigma_3}$$

Fix-Same

$$\frac{\Phi_0; \Sigma_0; \Gamma \vdash \overline{T} \approx_{\equiv} \overline{U} \rhd \Phi_1; \Sigma_1 \qquad \Phi_0; \Sigma_1; \Gamma \vdash \overline{t} \approx_{\equiv} \overline{u} \rhd \Phi_2; \Sigma_2}{\Phi_0; \Sigma_0; \Gamma \vdash \mathbf{fix}_j\ \{\overline{x/n : T := t}\} \approx_{\mathscr{R}} \mathbf{fix}_j\ \{\overline{x/n : U := u}\} \rhd \Phi_2; \Sigma_2}$$

App-FO

$$\frac{\Phi_0;\Sigma_0;\Gamma\vdash t\approx_{\mathscr{R}} u \rhd \Phi_1;\Sigma_1 \qquad n\geq 0 \qquad \Phi_1;\Sigma_1;\Gamma\vdash \overline{t_n}\approx_{\equiv}\overline{u_n}\rhd \Phi_2;\Sigma_2}{\Phi_0;\Sigma_0;\Gamma\vdash t\ \overline{t_n}\approx_{\mathscr{R}} u\ \overline{u_n}\rhd \Phi_2;\Sigma_2}$$

Meta-δR, Lam-βR, Let-ζR

$$\frac{\begin{array}{c}\Sigma;\Gamma\vdash u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta\Sigma,\beta,\zeta} u' \\ \Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u' \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Meta-δL, Lam-βL, Let-ζL

$$\frac{\begin{array}{c}\Sigma;\Gamma\vdash t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta\Sigma,\beta,\zeta} t' \\ \Phi;\Sigma;\Gamma\vdash t'\approx_{\mathscr{R}} u \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Case-ιR

$$\frac{\begin{array}{c}u \text{ is } \mathbf{fix} \text{ or } \mathbf{match} \qquad \Sigma;\Gamma\vdash u{\downarrow}^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta}u' \\ u\neq u' \qquad \Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u' \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Case-ιL

$$\frac{\begin{array}{c}t \text{ is } \mathbf{fix} \text{ or } \mathbf{match} \qquad \Sigma;\Gamma\vdash t{\downarrow}^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta}t' \\ t\neq t' \qquad \Phi;\Sigma;\Gamma\vdash t'\approx_{\mathscr{R}} u \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Cons-δNotStuckR

$$\frac{\begin{array}{c}\text{not } \Sigma;\Gamma\vdash \text{is\_stuck } u \qquad u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \\ \Sigma;\Gamma\vdash u'{\downarrow}^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta}u'' \qquad \Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u'' \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Cons-δStuckL

$$\frac{\begin{array}{c}\Sigma;\Gamma\vdash \text{is\_stuck } u \qquad t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \\ \Sigma;\Gamma\vdash t'{\downarrow}^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta}t'' \qquad \Phi;\Sigma;\Gamma\vdash t''\approx_{\mathscr{R}} u \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Cons-δR

$$\frac{\begin{array}{c}\text{not } t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \qquad u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \\ \Sigma;\Gamma\vdash u'{\downarrow}^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta}u'' \qquad \Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u'' \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Cons-δL

$$\frac{\begin{array}{c}\text{not } u \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} u' \qquad t \overset{\mathrm{w}}{\rightsquigarrow}_{\delta E,\delta\Gamma} t' \\ \Sigma;\Gamma\vdash t'{\downarrow}^{\mathrm{w}}_{\beta\zeta\delta\Sigma\iota\theta}t'' \qquad \Phi;\Sigma;\Gamma\vdash t''\approx_{\mathscr{R}} u \rhd \Phi';\Sigma'\end{array}}{\Phi;\Sigma;\Gamma\vdash t\approx_{\mathscr{R}} u\rhd \Phi';\Sigma'}$$

Lam-ηR

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

Lam-ηL

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

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

Meta-Same-Same

$$\frac{\Phi;\Sigma;\Gamma \vdash \overline{t} \approx_{\equiv} \overline{u} \rhd \Phi';\Sigma'}{\Phi;\Sigma;\Gamma \vdash ?x[\sigma]\ \overline{t} \approx_{\mathscr{R}} ?x[\sigma]\ \overline{u} \rhd \Phi';\Sigma'}$$

Meta-Same

$$\frac{\begin{array}{c} ?x : T[\Psi_1] \in \Sigma \qquad \Psi_1 \vdash \sigma \cap \sigma' \rhd \Psi_2 \qquad \cdot \vdash \mathsf{sanitize}(\Psi_2) \rhd \Psi_3 \\ \mathsf{FV}(T) \subseteq \Psi_3 \qquad \Phi;\Sigma \cup \{?y : T[\Psi_3], ?x := ?y[\widehat{\Psi_3}]\};\Gamma \vdash \overline{t} \approx_{\equiv} \overline{u} \rhd \Phi';\Sigma' \end{array}}{\Phi;\Sigma;\Gamma \vdash ?x[\sigma]\ \overline{t} \approx_{\mathscr{R}} ?x[\sigma']\ \overline{u} \rhd \Phi';\Sigma'}$$

Intersec-Nil

$$\frac{}{\cdot \vdash \cdot \cap \cdot \rhd \cdot}$$

Intersec-Keep

$$\frac{\Gamma \vdash \sigma \cap \sigma' \rhd \Gamma'}{\Gamma, x : A \vdash \sigma, t \cap \sigma', t \rhd \Gamma', x : A}$$

Intersec-Remove

$$\frac{\Gamma \vdash \sigma \cap \sigma' \rhd \Gamma' \qquad y \neq z}{\Gamma, x : T \vdash \sigma, y \cap \sigma', z \rhd \Gamma'}$$

Intersec-Keep-Def

$$\frac{\Gamma \vdash \sigma \cap \sigma' \rhd \Gamma'}{\Gamma, x := u : A \vdash \sigma, t \cap \sigma', t \rhd \Gamma', x := u : A}$$

Intersec-Remove-Def

$$\frac{\Gamma \vdash \sigma \cap \sigma' \rhd \Gamma' \qquad y \neq z}{\Gamma, x := u : T \vdash \sigma, y \cap \sigma', z \rhd \Gamma'}$$

Meta-InstR

$$\frac{\begin{array}{c} ?x : T[\Psi] \in \Sigma_0 \qquad t', \overline{z}' = \mathsf{remove\_tail}(t;\overline{z}) \qquad t' \downarrow_{\beta}^{\mathrm{w}} t'' \\ \Sigma_0 \vdash \mathsf{prune}(?x;\overline{y},\overline{z}';t'') \rhd \Sigma_1 \qquad \Sigma_1;\Gamma \vdash \overline{z}' : \overline{U} \qquad t''' = (\lambda \overline{w : U}.\ \Sigma_1(t''))\{\overline{y},\overline{z}'/\hat{\Psi},\overline{w}\}^{-1} \\ \Sigma_1;\Psi \vdash t''' : T' \qquad \Phi;\Sigma_1;\Psi \vdash T' \approx_{\leq} T \rhd \Phi';\Sigma_2 \end{array}}{\Phi;\Sigma_0;\Gamma \vdash t \approx_{\mathscr{R}} ?x[\overline{y}]\ \overline{z} \rhd \Phi'\Sigma_2 \cup \{?x := t'''\}}$$

Meta-FOR

$$\frac{\begin{array}{c} ?x : T[\Psi] \in \Sigma_0 \\ 0 < n \qquad \Phi_0;\Sigma_0;\Gamma \vdash u\ \overline{u'_m} \approx_{\equiv} ?x[\sigma] \rhd \Phi_1;\Sigma_1 \qquad \Phi_1;\Sigma_1;\Gamma \vdash \overline{u''_n} \approx_{\equiv} \overline{t_n} \rhd \Phi_2;\Sigma_2 \end{array}}{\Phi_0;\Sigma_0;\Gamma \vdash u\ \overline{u'_m u''_n} \approx_{\mathscr{R}} ?x[\sigma]\ \overline{t_n} \rhd \Phi_2;\Sigma_2}$$

Meta-DelDepsR

$$\frac{\begin{array}{c} ?x : T[\Psi] \in \Sigma_0 \qquad l = [i \mid \sigma_i \text{ is variable and } \nexists j > i.\ \sigma_i = (\sigma,\overline{u})_j] \\ \cdot \vdash \mathsf{sanitize}(\Psi_{|\,l}) \rhd \Psi' \qquad \Sigma_0 \vdash \mathsf{prune}(?x;\hat{\Psi}';T) \rhd \Sigma_1 \\ \Sigma_1 \cup \{?y : \Sigma_1(T)[\Psi'], ?x := ?y[\widehat{\Psi'}]\};\Gamma \vdash t \approx ?y[\sigma_{|\,l}]\ \overline{u} \rhd \Sigma_2 \end{array}}{\Sigma_0;\Gamma \vdash t \approx ?x[\sigma]\ \overline{u} \rhd \Sigma_2}$$

Meta-ReducerR

$$\frac{?u : T[\Psi] \in \Sigma_0 \qquad t \overset{\mathrm{w}\ 0..1}{\rightsquigarrow_{\delta}} t' \qquad t' \downarrow_{\beta\zeta\iota\theta}^{\mathrm{w}} t'' \qquad \Phi_0;\Sigma_0;\Gamma \vdash t'' \approx_{\mathscr{R}} ?u[\sigma]\ \overline{t_n} \rhd \Phi_1;\Sigma_1}{\Phi_0;\Sigma_0;\Gamma \vdash t \approx_{\mathscr{R}} ?u[\sigma]\ \overline{t_n} \rhd \Phi_1;\Sigma_1}$$

SANITIZE-NIL

$$\frac{}{\xi \vdash \mathsf{sanitize}(\cdot) \rhd \cdot}$$

SANITIZE-KEEP

$$\frac{\mathsf{FV}(T) \subseteq \overline{x} \qquad y,\overline{x} \vdash \mathsf{sanitize}(\Gamma) \rhd \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y : T, \Gamma) \rhd y : T, \Gamma'}$$

SANITIZE-REMOVE

$$\frac{\mathsf{FV}(T) \not\subseteq \overline{x} \qquad \overline{x} \vdash \mathsf{sanitize}(\Gamma) \rhd \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y : T, \Gamma) \rhd \Gamma'}$$

SANITIZE-KEEP-DEF

$$\frac{\mathsf{FV}(T) \subseteq \overline{x} \qquad \mathsf{FV}(u) \subseteq \overline{x} \qquad y,\overline{x} \vdash \mathsf{sanitize}(\Gamma) \rhd \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y := u : T, \Gamma) \rhd y := u : T, \Gamma'}$$

SANITIZE-REMOVE-DEF

$$\frac{\mathsf{FV}(T) \not\subseteq \overline{x} \ \vee \ \mathsf{FV}(u) \not\subseteq \overline{x} \qquad \overline{x} \vdash \mathsf{sanitize}(\Gamma) \rhd \Gamma'}{\overline{x} \vdash \mathsf{sanitize}(y := u : T, \Gamma) \rhd \Gamma'}$$

PRUNE-RIGID

$$\frac{h \in s \cup \mathscr{C}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};h) \rhd \Sigma}$$

PRUNE-VAR

$$\frac{x \in \overline{y}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};x) \rhd \Sigma}$$

PRUNE-LAM, PRUNE-PROD

$$\frac{\Pi \in \{\lambda, \forall\} \qquad \Sigma \vdash \mathsf{prune}(?x;\overline{y},z;t) \rhd \Sigma'}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};\Pi z.\ t) \rhd \Sigma'}$$

PRUNE-LET

$$\frac{\begin{array}{c}\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};t_2) \rhd \Sigma_1 \\ \Sigma_1 \vdash \mathsf{prune}(?x;\overline{y},z;t_1) \rhd \Sigma_2\end{array}}{\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};\textsf{\textbf{let}}\ z := t_2\ \textsf{\textbf{in}}\ t_1) \rhd \Sigma_2}$$

PRUNE-APP

$$\frac{\begin{array}{c}\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};t) \rhd \Sigma_1 \\ \Sigma_i \vdash \mathsf{prune}(?x;\overline{y};t_i) \rhd \Sigma_{i+1} \qquad i \in [1,n]\end{array}}{\Sigma_0 \vdash \mathsf{prune}(?x;\overline{y};t\ \overline{t_n}) \rhd \Sigma_{n+1}}$$

PRUNE-META-NOPRUNE

$$\frac{\begin{array}{c}?z : T[\Psi_0] \in \Sigma \qquad ?x \neq ?z \\ \Psi_0 \vdash \mathsf{prune\_ctx}(?x;\overline{y};\sigma) \rhd \Psi_0\end{array}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};?z[\sigma]) \rhd \Sigma}$$

PRUNE-META

$$\frac{\begin{array}{c}?u : T[\Psi_0] \in \Sigma \qquad ?x \neq ?z \qquad \Psi_0 \vdash \mathsf{prune\_ctx}(?x;\overline{y};\sigma) \rhd \Psi_1 \\ \cdot \vdash \mathsf{sanitize}(\Psi_1) \rhd \Psi_2 \qquad \Sigma \vdash \mathsf{prune}(?x;\widehat{\Psi_2};T) \rhd \Sigma'\end{array}}{\Sigma \vdash \mathsf{prune}(?x;\overline{y};?z[\sigma]) \rhd \Sigma', ?u : \Sigma'(T)[\Psi_2] \cup \{?z := ?u[\widehat{\Psi_2}]\}}$$

PRUNECTX-NIL

$$\frac{}{\cdot \vdash \mathsf{prune\_ctx}(?x;\overline{y};\cdot) \rhd \cdot}$$

PRUNECTX-NOPRUNE

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

PRUNECTX-PRUNE

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

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

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

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

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

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