

Towards a better-behaved unification algorithm for Coq

Beta Ziliani¹ and Matthieu Sozeau²

¹ MPI-SWS (beta@mpi-sws.org)

² Inria Paris (matthieu.sozeau@inria.fr)

1 Introduction

The unification algorithm is at the heart of a proof assistant like Coq. In particular, it is a key component in the *refiner* (the algorithm that has to infer implicit terms and missing type annotations) and in the application of lemmas. In the first case, unification is in charge of equating the type of function arguments with the type of the elements to which the function is applied. In the second case, for instance when using the `apply` tactic, it is in charge of unifying the current goal with the conclusion of the lemma.

Despite playing a central role in proof development, there is no good source of documentation to understand Coq's unification algorithm. Since unification is inherently undecidable in Coq, as it must deal with higher-order problems up to conversion, some form of heuristic is desirable in order to solve problems that are trivial to the human eye. Otherwise, the proof developer will get easily frustrated when she finds two apparently equal terms not being unified. For instance, a desirable heuristic will equate the terms $?x ++ ?y \approx [] ++ (1 :: [])$, assigning $?x$ to the empty list and $?y$ to the singleton list $(1 :: [])$, where $?x$ and $?y$ are meta-variables and `++` is the list concatenation function. There exist other possible (convertible) solutions, like for instance assigning $(1 :: [])$ to $?x$ and $[]$ to $?y$, but in most of the cases preserving the structures of terms gives reasonable solutions.

The current approach in the source code of Coq includes this heuristic but also some harmful ones, like *Constraint Postponement*. In certain cases, when an equation has multiple solutions, it is delayed until more information is gathered to solve the ambiguity. Constraint Postponement is commonly used (see *e.g.* [1]), and gives in practice reasonably good results. However, it also has its drawbacks. It may lead to extraneous error messages, since errors are reported at a later point in the unification process. More importantly, it does not mix well with other aspects of Coq's unification, like resolution of *Canonical Structures* [8]. Canonical Structures is an overloading mechanism similar to *type classes*, extensively used in the Mathematical Components library [4], on which the recent feat of proving the odd-order theorems [3] crucially relies. Supporting canonical structures resolution in unification makes the algorithm extremely sensitive to heuristics, since instance resolution depends heavily on the order in which unification problems are considered. Constraint Postponement also has an impact in performance, which became unpredictable as the unifier's complexity does not depend only on the size of the reduction paths of the two inputs.

In this talk we are going to present our work in progress on a new unification algorithm, built from scratch, which focuses on the following main properties:

Understandable: The algorithm can be described in full in a few pages, including canonical structures instance resolution.

Sound: The algorithm, when it succeeds, provides a well-typed substitution that equates both terms (up to conversion).

Predictable: The algorithm does not include heuristics that are hard to reason about.

In this paper, we will quickly introduce the language (§ 2) and delve into the delicate issues that come up in our setting, mainly due to unification up-to reduction, backtracking and dependencies (§ 3).

2 A primer on CIC with meta-variables

The *Calculus of Inductive Constructions* (CIC) is a dependently typed λ -calculus extended with inductive types. The terms of the language are defined as

$$\begin{aligned}
t, T \hat{=} & x \mid c \mid i \mid k \mid s & x \in \mathcal{V}, c \in \mathcal{C}, i \in \mathcal{I}, k \in \mathcal{K}, s \in \mathcal{S} \\
& \mid \forall x : T. T \mid \lambda x : T. t \mid t t \mid \text{let } x = t : T \text{ in } t \mid ?u[\sigma] & ?u \in \mathcal{M} \\
& \mid \text{case}_T t \text{ of } k_1 \bar{x} \Rightarrow t; \dots; k_n \bar{x} \Rightarrow t \text{ end} \\
& \mid \text{fix}_j \{x/n : T := t; \dots; x/n : T := t\}
\end{aligned}$$

where \mathcal{V} is an enumerable set of variables, \mathcal{M} of meta-variables, \mathcal{C} of constants, \mathcal{I} of inductive types, \mathcal{K} of inductive constructors, and \mathcal{S} is an infinite set of sorts defined as $\{\text{Prop}, \text{Type}(i) \mid i \in \mathbb{N}\}$.

Meta-variables are equipped with a substitution σ , which is nothing more than a list of terms.

In order to destruct an inductive type CIC provides a case constructor (match in vernacular) and a fixpoint. `case` is annotated with the return predicate T . In the term `fix`, the expression $x/n : T := t$ means that T is a type starting with at least n product types, and the n -th variable is the decreasing one in t . The subscript j of `fix` selects the j -th function as the main entry point.

The local context Γ , the meta-variables context Σ , and global environment E are defined as:

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

Meta-variables have types T with all free variables bounded within a local context Ψ . In this work we borrow the notation $T[\Psi]$ from Contextual Modal Type Theory [6], while in [7] this is noted $\Psi \vdash T$.

Each possibly mutually recursive inductive type is stored in the environment E with the shape

$$\begin{aligned}
I \hat{=} & \forall x_1 : T_1, \dots, \forall x_h : T_h, \\
& \{ \overline{i_m : A_m := \{k_1^m : C_1^m; \dots; k_{n_1}^m : C_{n_1}^m\}^m} \}
\end{aligned}$$

where every $i_m \in \mathcal{I}$, every $k_n^j \in \mathcal{K}$, every C_n^m has the shape $\forall x : \overline{T}, i_m t_1 \dots t_h$, and every A_m has the shape $\forall x : \overline{T}, s$. Inductive definitions are restricted to avoid circularity (each i_m can only appear strictly positive in every i_n depending on it). For the purpose of this work, we are not taking this restriction into consideration.

Reduction: Since the unification algorithm have to equate terms up to conversion, we need to present the reduction rules for CIC, listed in Figure 1. Besides the standard β rule, we have the ζ rule that expands let-definitions, three δ rules, that expand definitions from any of the contexts, and the ι rules for case destruction and fixpoint unfolding. The reduction rules depend on the contexts, which we assume as given. These rules rely on the standard multi-substitution of terms, noted $t\{\overline{t_n/x_n}\}$, which replaces each variable x_i with term t_i in term t .

3 Unification

The algorithm takes terms t_1 and t_2 , a well-formed meta-variable context Σ and a well-formed local context Γ . We have a well-formed global environment E that is omnipresent. As precondition, the two terms should be well typed with types A_1 and A_2 respectively. Note that we do not require the types

$$\begin{array}{l}
(\lambda x : T.t) t' \rightsquigarrow_{\beta} t\{t'/x\} \\
\text{let } x = t' : T \text{ in } t \rightsquigarrow_{\zeta} t\{t'/x\} \\
h \rightsquigarrow_{\delta} t \qquad \text{if } (h := t : T) \in E \text{ or } (h := t : T) \in \Gamma \\
?u[\sigma] \rightsquigarrow_{\delta} t\{\sigma/\Psi\} \qquad \text{if } (?u := t : T[\Psi]) \in \Sigma \\
\text{case}_T (k_j \bar{a}) \text{ of } \overline{k \bar{x} \Rightarrow t} \text{ end} \rightsquigarrow_t t_j\{\bar{a}/x_j\} \\
\text{fix}_j \{F\} \bar{a} \rightsquigarrow_t t_j\{\overline{\text{fix}_m \{F\}/x_m}\} \bar{a} \qquad F = \overline{x/n : T := t}
\end{array}$$

Figure 1: Reduction rules in CIC.

to be equal. Upon success, the algorithm returns a new meta-variable context Σ' with instantiations for the meta-variables appearing in the terms or in Γ . The algorithm ensures that the terms t_1 and t_2 are convertible under this new meta-context. The unification judgment is noted as $\Sigma; \Gamma \vdash t_1 \approx t_2 \triangleright \Sigma'$.

In this abstract we will not present the unification rules, which the interested reader is invited to read from the accompanying appendix. Instead, we will focus on three points that make the design of the algorithm delicate: (1) type dependencies, (2) conversion, and (3) canonical structures resolution. In the following sections we explain these and motivate the need for a change in the current algorithm.

3.1 Type dependencies

Sometimes the unification algorithm is faced with an equation that has not one but many solutions, in a context where there should only be one possible candidate. For instance, consider the following term witnessing an existential quantification:

$$\text{exist } _ 0 \text{ (le_n 0)} : \exists x.x \leq x$$

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 we want the whole term to be a proof that there exists a number that is lesser or equal to itself. In this case, we provide 0 as such number, and the proof `le_n 0` which has type $0 \leq 0$.

When typechecking the term, Coq first considers the term and then it checks that it is compatible to the typing constraint. More precisely, Coq will create a fresh meta-variable for the predicate P , let's call it $?P$, and unify $?P 0$ with $0 \leq 0$. Without any further information, Coq has four different (incomparable) options 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 delays the equation in the hope that further information will help disambiguate the problem. In this case, that information was given through the typing constraint, and Coq succeeds to typecheck the term. If there were no typing constraint, Coq would have picked an arbitrary solution. There are two direct consequences of these design decisions. On one hand, the algorithm is more “complete”, in the sense that less typing annotations are required (in this case, we do not need to specify P). On the other hand, the arbitrary solution selected by Coq may not be the one expected by the proof developer. In the example above, for instance, when we remove the typing constraint Coq will decide that the term has type $\exists x.0 \leq x$. If, at a different point in the proof script, this term is used to prove $\exists x.x \leq x$, the proof developer will have to debug the proof script to find out where the problem originated from.

Moreover, performance-wise, constraint postponement can be disastrous as it might postpone unsolvable constraints and make failure exponentially slower, e.g. due to first-order unification (see 3.2) being applied to other unification problems before finally failing.

For all these reasons we decided to remove postponement from the algorithm. Actually, in most cases it is possible to achieve the same level of “completeness” without constraint postponement, by using *bidirectional typechecking*, that is, to use the typing information available (e.g., from the typing constraint) to infer meta-variables in the term. Then, when typechecking exist_0_le_n_0 under the typing constraint $\exists x.0 \leq 0$, we can propagate a *unique* solution for P from the type to the term. Removing postponement also helps to get a simpler proof of type soundness for the unification algorithm, which we plan to mechanize.

3.2 First-order approximation

The algorithm includes a so-called first-order unification rule:

$$\frac{\Sigma_0; \Gamma \vdash u \approx u' \triangleright \Sigma_1}{\Sigma_0; \Gamma \vdash t u \approx t u' \triangleright \Sigma_1} \text{APP-FO}$$

This rule applies when two applications are unified (here in a simplified binary application version), even if the head t might be unfoldable (i.e., a definition in some context). This rule clearly precludes the generation of most general unifiers, as u and u' do not need to be unified if, for example, t is $\lambda x.0$. However, it is very natural to add it as it can shorten many unifications that would in the end result in the unification of the arguments of t . So we must bear with it if we are to be compatible with the existing algorithm and keep in sync with its performance. Of course, a drawback of this rule is that we must *backtrack* on its application if the premise cannot be derived, and try instead to perform a step of reduction, like unfolding the head constant.

This behavior is problematic, especially in presence of fixpoint definitions which might generate repeated, almost identical applications of this rule which ultimately fail, when only the normal forms of the fixpoint applications can unify. We are experimenting with ideas to keep track of successes (and failures) of unifications to avoid an exponential blowup due to that behavior.

3.3 Canonical Structures resolution

A *structure* is a particular inductive type: it has only one constructor, and it generates one projector for each argument of the constructor. The syntax is

$$\text{structure } i \bar{a} : s := k \{ p_1 : A_1; \dots; p_n : A_n \}$$

where \bar{a} is a list of *arguments* of the type, of the form $x_1 : T_1, \dots, x_m : T_m$. Each p_j is a *projector* name. This language construct generates an inductive type

$$\{ i : \forall \bar{a}. s := \{ k : \forall \bar{a}. \overline{\forall p : A}. i \bar{a} \} \}$$

and for each projector *name* p_j it generates a projector *function*:

$$\lambda \bar{a}. \lambda z. \text{case } s \text{ with } k \ x_1 \ \dots \ x_j \ \dots \ x_n \Rightarrow x_j \ \text{end} : \forall \bar{a}. \forall z : i \bar{a}. A_j$$

An instance t of the structure is created with the constructor k :

$$t := \overline{\forall x : B}. k \ t_1 \ \dots \ t_{m+n}$$

where m is the number of arguments of the structure. Terms t_1 to t_m corresponds to the arguments of the structure, and t_{m+1} to t_{m+n} to each of the values p_j .

The important aspect of structures is that their instances can be deemed as “canonical”. A canonical instance instructs the unification algorithm to instantiate a structure meta-variable with the instance, if certain conditions holds. More precisely, a canonical instance populates the canonical instance database Δ_{db} with triples (p_j, h_j, ι) , where h_j is the head constant appearing in value t_{m+j} . (h_j can also be an implication (\rightarrow) or a sort.) Then, whenever the unification algorithm have to solve a problem of the form $p_j \bar{a} ?s \approx h_j \bar{b}$, it instantiates $?s$ with ι . There cannot be two triples with the same projector and head constant. Coq enforces this invariant by shadowing previous triples with new overlapping triples.

An immediate consequence of using the head constant to determine the instance is that δ -expanding a term may expose a different constant, and therefore a different instance. In [5, 2], for instance, this fact is used to resolve overlapping instances. Similarly, an earlier δ -expansion may prevent the use of an instance, so δ -expansion is delayed as much as possible.

Another relevant aspect, as we mentioned in the introduction, is that constraint postponement in the presence of Canonical Structures resolution may lead to unexpected results.

4 Conclusion

We have presented the main features and design choices of our algorithm. It has been implemented and is being successfully tested on the Canonical Structures resolution part of the mathematical components library, which relies on all the expressive power of the unifier. With our collaborators, we are also working on a proof of soundness, mechanized in Coq, to provide a solid ground on which to build complex tactics. As a mid-term goal, we also plan to aggressively optimize the algorithm, making sure its semantics remains the same. Altogether, our work will give Coq users a fast, completely verified and documented unification algorithm.

References

- [1] A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA*, pages 10–26, 2011.
- [2] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101, 2008.
- [3] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O’Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP 2013*, volume 7998 of *LNCS*. Springer, 2013.
- [4] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008.
- [5] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23:357–401, 7 2013.
- [6] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9:23:1–23:49, June 2008.
- [7] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pages 473–487. Springer-Verlag, 2003.
- [8] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proc of POPL’97*, pages 292–301, 1997.

5 Appendix: Unification rules

Given the complexity of the algorithm, we split the rules in different figures. Figures 2 and 3 show the subset of rules involving the CIC constructs without meta-variables; Figure 4 considers meta-variable instantiation; and Figure 8 considers canonical structures resolution.

The first three rules (PROP-SAME, TYPE-SAME and TYPE-SAME-LE) unifies two types, according to the restriction imposed on universe levels. For abstractions (LAM-SAME) and products (PROD-SAME) we first unify the types of the arguments and then the body of the binder, with the local context extended with the bound variable.

When unifying two lets the rule LET-SAME compares the definitions and then the body, augmenting the context with the definition. Note that we don't need to check the types of the definitions, since if the definitions are unifiable then their type is unifiable as well. If the rule fails to apply, then the rule LET-ZETA unfolds the definitions in both sides and tries again.

RIGID-SAME equates the same variable, constant, inductive type, or constructor.

The following two rules consider the rules for matching cases and fixpoints. In both cases we just unify pointwise every component of the constructors (case and fix respectively).

The last rule of Figure 2 considers two applications with the same number of arguments (n). It first compares the head element (t and t') and then proceeds to unify each of the arguments.

When the rules in Figure 2 fails to apply, then the algorithm tries to do one step reduction and try again, in the hope to find a solution. This process is described in Figure 3. The rules are pretty easy to read, and are labeled according to the reduction step they take.

One point should be made: when one of the terms is a case or a fix the algorithm tries weak-head reducing the term. We denote the weak head reduction of t under contexts Σ and Γ as $\Sigma; \Gamma \vdash t \downarrow_{\beta \xi \delta}^w t'$. As a sanity check, we make sure that progress was made, by comparing the result of the weak head reduction with the original term.

Meta-variable instantiation is considered in figure 4. We proceed to describe each rule according to the case.

Same meta-variable: If both terms are the same meta-variable $?u$, we have two distinct cases: if their substitution is the exact same list of variables ξ , the rule META-SAME-SAME applies, in which the arguments of the meta-variable are compared point-wise. If, instead, their substitution is different, then the rule META-SAME is attempted. To better understand this rule, let's look at an example. Say $?u$ has type $T[x_1 : \text{nat}, x_2 : \text{nat}]$ and we have to solve the equation

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

where y_1, y_2 and y_3 are defined in the local context. From this equation we cannot know yet what value $?u$ will hold, but at least we know it cannot refer to the second parameter, x_2 , since that will render the equation above false. This reasoning is reflected in the rule META-SAME in the hypothesis

$$\Psi_1 \vdash \xi \cap \xi' \triangleright \Psi_2$$

This judgment performs an intersection of both substitutions, filtering out those positions from the context of the meta-variable Ψ_1 where the substitution disagree, resulting in Ψ_2 . This judgment is defined in Figure 5.

Once we filter out the disagreeing positions of the substitution we need to create a new meta-variable $?v$ with same type of $?u$, but in the shorter context Ψ_2 . We further instantiate $?u$ with $?v$. Both the creation of $?v$ and the instantiation of $?u$ in the context Σ is expressed in the fragment $\Sigma \cup \{?v : T[\Psi_2], ?u := ?v[\text{id}_{\Psi_2}]\}$ of the last hypothesis. We use this new context to compare point-wise the arguments of the meta-variable.

$$\begin{array}{c}
\frac{}{\Sigma; \Gamma \vdash \text{Prop} \approx \text{Prop} \triangleright \Sigma} \text{PROP-SAME} \\
\\
\frac{i = j}{\Sigma; \Gamma \vdash \text{Type}(i) \approx \text{Type}(j) \triangleright \Sigma} \text{TYPE-SAME} \\
\\
\frac{i \leq j}{\Sigma; \Gamma \vdash \text{Type}(i) \lesssim \text{Type}(j) \triangleright \Sigma} \text{TYPE-SAME-LE} \\
\\
\frac{\Sigma; \Gamma \vdash A_1 \approx A_2 \triangleright \Sigma' \quad \Sigma'; \Gamma, x : A_1 \vdash t_1 \approx t_2 \triangleright \Sigma''}{\Sigma; \Gamma \vdash \lambda x : A_1. t_1 \approx \lambda x : A_2. t_2 \triangleright \Sigma''} \text{LAM-SAME} \\
\\
\frac{\Sigma; \Gamma \vdash A_1 \approx A_2 \triangleright \Sigma' \quad \Sigma'; \Gamma, x : A_1 \vdash B_1 \approx B_2 \triangleright \Sigma''}{\Sigma; \Gamma \vdash \forall x : A_1. B_1 \approx \forall x : A_2. B_2 \triangleright \Sigma''} \text{PROD-SAME} \\
\\
\frac{\Sigma; \Gamma \vdash t_2 \approx t'_2 \triangleright \Sigma' \quad \Sigma'; \Gamma, x := t_2 \vdash t_1 \approx t'_1 \triangleright \Sigma''}{\Sigma; \Gamma \vdash \text{let } x = t_2 : T \text{ in } t_1 \approx \text{let } x = t'_2 : T' \text{ in } t'_1 \triangleright \Sigma''} \text{LET-SAME} \\
\\
\frac{\Sigma; \Gamma \vdash t_1 \{t_2/x\} \approx t'_1 \{t'_2/x\} \triangleright \Sigma'}{\Sigma; \Gamma \vdash \text{let } x = t_2 : T \text{ in } t_1 \approx \text{let } x = t'_2 : T' \text{ in } t'_1 \triangleright \Sigma'} \text{LET-ZETA} \\
\\
\frac{h \in \mathcal{V} \cup \mathcal{C} \cup \mathcal{S} \cup \mathcal{K}}{\Sigma; \Gamma \vdash h \approx h \triangleright \Sigma} \text{RIGID-SAME} \\
\\
\frac{\Sigma_0; \Gamma \vdash T \approx T' \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash t \approx t' \triangleright \Sigma_2 \quad \Sigma_2; \Gamma \vdash \bar{b} \approx \bar{b}' \triangleright \Sigma_3}{\Sigma_0; \Gamma \vdash \text{case}_T t \text{ of } \bar{b} \text{ end} \approx \text{case}_{T'} t' \text{ of } \bar{b}' \text{ end} \triangleright \Sigma_3} \text{CASE-SAME} \\
\\
\frac{\Sigma_0; \Gamma \vdash \bar{T} \approx \bar{T}' \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash \bar{t} \approx \bar{t}' \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash \text{fix}_j \{x/n : T := t\} \approx \text{fix}_j \{x'/n' : T' := t'\} \triangleright \Sigma_2} \text{FIX-SAME} \\
\\
\frac{\Sigma_0; \Gamma \vdash t \approx t' \triangleright \Sigma_1 \quad \Sigma_1; \Gamma \vdash \bar{t}_n \approx \bar{t}'_n \triangleright \Sigma_2}{\Sigma_0; \Gamma \vdash t \bar{t}_n \approx t' \bar{t}'_n \triangleright \Sigma_2} \text{APP-FO}
\end{array}$$

Figure 2: Unification algorithm: pure CIC constructs (part 1).

$$\begin{array}{c}
\frac{\Sigma; \Gamma \vdash t' \approx t\{t_1/x\} t_2 \dots t_n \triangleright \Sigma'}{\Sigma; \Gamma \vdash t' \approx (\lambda x : A.t) t_1 \dots t_n \triangleright \Sigma'} \text{LAM-BETAR} \\
\\
\frac{\Sigma; \Gamma \vdash t\{t_1/x\} t_2 \dots t_n \approx t' \triangleright \Sigma'}{\Sigma; \Gamma \vdash (\lambda x : A.t) t_1 \dots t_n \approx t' \triangleright \Sigma'} \text{LAM-BETAL} \\
\\
\frac{\Sigma; \Gamma \vdash t' \approx t_1\{t_2/x\} \bar{t}_n \triangleright \Sigma'}{\Sigma; \Gamma \vdash t' \approx (\text{let } x = t_2 : T \text{ in } t_1) \bar{t}_n \triangleright \Sigma'} \text{LET-ZETAR} \\
\\
\frac{\Sigma; \Gamma \vdash t_1\{t_2/x\} \bar{t}_n \approx t' \triangleright \Sigma'}{\Sigma; \Gamma \vdash (\text{let } x = t_2 : T \text{ in } t_1) \bar{t}_n \approx t' \triangleright \Sigma'} \text{LET-ZETAL} \\
\\
\frac{(x := t : A) \in \Gamma \quad \Sigma; \Gamma \vdash t' \approx t \bar{t}_n \triangleright \Sigma'}{\Sigma; \Gamma \vdash t' \approx x \bar{t}_n \triangleright \Sigma'} \text{RIGID-DELTA-VARR} \\
\\
\frac{(x := t : A) \in \Gamma \quad \Sigma; \Gamma \vdash t \bar{t}_n \approx t' \triangleright \Sigma'}{\Sigma; \Gamma \vdash x \bar{t}_n \approx t' \triangleright \Sigma'} \text{RIGID-DELTA-VARL} \\
\\
\frac{t' \text{ is fix or case} \quad \Sigma; \Gamma \vdash t' \downarrow_{\beta\zeta\delta\iota}^w t'' \quad t' \neq t'' \quad \Sigma; \Gamma \vdash t \approx t'' \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx t' \triangleright \Sigma'} \text{WHDR} \\
\\
\frac{t \text{ is fix or case} \quad \Sigma; \Gamma \vdash t \downarrow_{\beta\zeta\delta\iota}^w t'' \quad t \neq t'' \quad \Sigma; \Gamma \vdash t'' \approx t' \triangleright \Sigma'}{\Sigma; \Gamma \vdash t \approx t' \triangleright \Sigma'} \text{WHDL} \\
\\
\frac{(c := t : A) \in E \quad \Sigma; \Gamma \vdash t' \approx t \bar{t}_n \triangleright \Sigma'}{\Sigma; \Gamma \vdash t' \approx c \bar{t}_n \triangleright \Sigma'} \text{RIGID-DELTA-CONSR} \\
\\
\frac{(c := t : A) \in E \quad \Sigma; \Gamma \vdash t \bar{t}_n \approx t' \triangleright \Sigma'}{\Sigma; \Gamma \vdash c \bar{t}_n \approx t' \triangleright \Sigma'} \text{RIGID-DELTA-CONSL}
\end{array}$$

Figure 3: Unification algorithm: pure CIC constructs (part 2).

There are two other hypotheses that ensure that nothing goes wrong. Again, we explain them by means of example. The hypothesis

$$\text{FV}(T) \subseteq \Psi_2$$

ensures that the type T is well formed in the new (shorter) context Ψ_2 . This condition might sound redundant at first sight, and, in fact, it is not present in [1]. However, it is necessary since, unlike in [1], we don't have as premise that the type of both terms are the same. As example, consider the contexts

$$\Sigma = \{?u : x[x : \text{Type}]\} \quad \Gamma = \{y : \text{Type}, z : \text{Type}\}$$

and the equation

$$u[y] \approx u[z]$$

The intersection of both substitutions will return an empty context. But we cannot create a new meta-variable $?v$ with type x in the empty context! The problem comes from the fact that both terms have different types (y and z respectively). By ensuring that every free variable in the type of the meta-variable is in the context Ψ_2 we prevent this issue.

More subtle is the inclusion of the premise

$$\Sigma \vdash \Psi_2$$

Because of convertibility, it may happen that the two substitutions agree on a value whose type depends on a previous value not equal in both substitutions. As example, consider contexts

$$\Sigma = \{?v : \text{Prop}[x : \text{Type}, p : \text{fst}(\text{Prop}, x)]\} \quad \Gamma = \{y : \text{Type}, z : \text{Type}, w : \text{Prop}\}$$

and the equation

$$?v[y, w] \approx ?v[z, w]$$

After performing the intersection, we get the ill-formed context $[p : \text{fst}(\text{Prop}, x)]$.

Meta-variable instantiation: The rules META-INSTL (R) are in charge of instantiating a meta-variable. On the left (right) hand side it has meta-variable $?u$ applied to the (variable to variable) substitution ξ and with (only variables) arguments ξ' . On the right (left) hand side it has some term t . Assuming $?u$ has (contextual) type $T[\Psi]$, this rule must find a term t' such that $t' \{ \xi / \Psi \} \xi'$ is convertible to t , where $\hat{\cdot}$ is defined as

$$x_1 : T_1, \dots, x_n : T_n \hat{=} x_1, \dots, x_n$$

In order to obtain t' the following steps are followed:

1. The meta-variables in t are *pruned*, as we are going to explain in the next section.
2. t' is constructed as a function taking arguments \bar{x} , one for each variable in ξ . The body of this function is the *inversion* of substitution $\xi, \xi' / \hat{\Psi}, \bar{x}$. More precisely, every variable in t appearing only once in the image of the substitution (ξ, ξ') is replaced by the corresponding variable in the domain of the substitution $(\hat{\Psi}, \bar{x})$. If a variable appears multiple times in the image and occur in term t , then inversion fails.
3. The type of t' is unified with the type of $?u$. We do this in order to ensure soundness of unification. Since we do not contemplate the types of the terms being unified, we need to obtain the type of t' in order to compare it with T . This introduces a penalty in the performance of the algorithm, but since we know t' is well typed (the unification algorithm requires both terms to be well typed, and the inversion process preserves the type), then we can perform a fast *retyping* of t' .
4. The term t' is *occur checked* to not contain meta-variable $?u$.

5.1 Pruning

The idea behind pruning can be understood with an example. Say we want to unify terms

$$?w[x, y] \approx c ?u[z, ?v[y]] \tag{1}$$

A solution exists, although z is a free variable in the rhs not appearing in the image of the substitution of the lhs. The solution has to restrict $?u$ so it does not depends on the first substitution. This can be done by meta-substituting $?u$ with a new $?u'$ with a smaller context. That is, if $?u : T[a : T_1, b : T_2]$, then

$$\begin{array}{c}
\frac{\Sigma; \Gamma \vdash \bar{t} \approx \bar{t}' \triangleright \Sigma'}{\Sigma; \Gamma \vdash ?u[\xi] \bar{t} \approx ?u[\xi] \bar{t}' \triangleright \Sigma'} \text{META-SAME-SAME} \\
\\
\frac{\Psi_1 \vdash \xi \cap \xi' \triangleright \Psi_2 \quad \Sigma \vdash \Psi_2 \quad \begin{array}{c} ?u : T[\Psi_1] \in \Sigma \\ \text{FV}(T) \subseteq \Psi_2 \quad \Sigma \cup \{?v : T[\Psi_2], ?u := ?v[\text{id}_{\Psi_2}]\}; \Gamma \vdash \bar{t} \approx \bar{t}' \triangleright \Sigma' \end{array}}{\Sigma; \Gamma \vdash ?u[\xi] \bar{t} \approx ?u[\xi'] \bar{t}' \triangleright \Sigma'} \text{META-SAME} \\
\\
\frac{\begin{array}{c} ?u : T[\Psi] \in \Sigma_0 \quad \Sigma_0 \vdash \text{prune}(\xi, \xi'; t) \triangleright \Sigma_1 \\ t' = \lambda \bar{x}. t \{ \xi, \xi' / \hat{\Psi}, \bar{x} \}^{-1} \quad \Sigma_1; \Psi \vdash t' : T' \quad \Sigma_1; \Psi \vdash T' \lesssim T \triangleright \Sigma_2 \quad ?u \notin t' \end{array}}{\Sigma_0; \Gamma \vdash ?u[\xi] \xi' \approx t \triangleright \Sigma_2 \cup \{?u := t'\}} \text{META-INSTL} \\
\\
\frac{\begin{array}{c} ?u : T[\Psi] \in \Sigma_0 \\ n \leq m \quad n > 0 \quad \Sigma_1; \Gamma \vdash ?u[\sigma] \approx t' \bar{t}'_{m-n} \triangleright \Sigma_2 \quad \Sigma_0; \Gamma \vdash \bar{t}_n \approx \overline{t'_{m-n+1..m}} \triangleright \Sigma_1 \end{array}}{\Sigma_0; \Gamma \vdash ?u[\sigma] \bar{t}_n \approx t' \bar{t}'_m \triangleright \Sigma_2} \text{META-FOL}
\end{array}$$

Figure 4: Meta-variable instantiation.

$$\begin{array}{c}
\overline{\cdot \vdash \cdot \cap \cdot \triangleright \cdot} \\
\\
\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi'}{\Psi, x : A \vdash \xi, y \cap \xi', y \triangleright \Psi', x : A} \\
\\
\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi'}{\Psi, x := t : A \vdash \xi, y \cap \xi', y \triangleright \Psi', x := t : A} \\
\\
\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi' \quad z \neq y}{\Psi, x : A \vdash \xi, y \cap \xi', z \triangleright \Psi'} \\
\\
\frac{\Psi \vdash \xi \cap \xi' \triangleright \Psi' \quad z \neq y}{\Psi, x := t : A \vdash \xi, y \cap \xi', z \triangleright \Psi'}
\end{array}$$

Figure 5: Intersection of substitutions

a fresh unification variable $?u'$ is created with type $T[b : T_2]$, and $?u := ?u'[b]$. The result of this process in Equation 1 is

$$?w[x, y] \approx c ?u'[?v[y]]$$

which can now be easily solved. Instead, if z occurs inside the substitution of $?v$,

$$?w[x, y] \approx c ?u[x, ?v[z]]$$

then it is not clear anymore, since a solution may exist by pruning z from $?v$, or by pruning $?v[z]$ from $?u$.

It is important to note that we can only prune offending variables that appear in the head of the term. Consider the following example:

$$?u[x] \approx c ?v[(x, y)] \quad (2)$$

One is tempted to prune the argument of $?v$, however this will prevent the unification algorithm from picking the following solution

$$?v[p] := \text{fst } p$$

instantiating further $?u$ with the (convertible) term x . As example, considering the following problem:

$$\text{let } p := (x, y) \text{ in } (?u[x], ?v[p]) \approx \text{let } p := (x, y) \text{ in } (c ?v[(x, y)], \text{fst } p)$$

After unifying the definition of the let, it introduces the definition $p := (x, y)$ in the local context and proceeds to pairwise unify the components of the pair. By unifying the first component we obtain Equation 2. If we (incorrectly) prune the argument from $?v$, this step succeeds instantiating $?u$ with $c ?v[]$. The second component will try to unify (after expanding the new definition for $?v$)

$$?v[] \approx \text{fst } p \quad (3)$$

failing to unify. In this example it is easy to see where things went wrong, but in general it's a bad idea to fail at the wrong place, as the developer has to trace the algorithm to find that, actually, the problem was in another place.

Figure 6 shows the rules for pruning. Given a meta-context Σ , a list of variables ξ and a term t , the pruning of meta-variables in t is denoted

$$\Sigma \vdash \text{prune}(\xi; t) \triangleright \Sigma'$$

where Σ' is a new meta-context extending Σ by instantiating the pruned meta-variables with new meta-variables, as we saw in the example above.

5.2 Canonical structures resolution

The scariest rules of this work are clearly the ones about canonical structures resolution, listed in Figure 8. But looking at them closely we can see they are not as scary as they look. The first rule CS-CONSTL shows the most common case of CS resolution. In this rule, on the left hand side we have projector p_j applied to the structure c , with structure parameters \bar{a} and arguments \bar{i} . On the right hand side we have constant h applied to arguments \bar{u} and \bar{i}' . That is, the j -th component of c should be a function taking arguments \bar{i} . In order to solve the equation we need an instance ι in the database relating p_j and h . This instance should be a function taking some arguments $x : B$ and returning the application of the constructor of the structure k to parameters \bar{a}' , and with field values \bar{v} . The j -th value should have head constant h , applied to arguments \bar{u}' . The algorithm should find the right instantiation for the arguments of the instance. For this, it creates new meta-variables $?y$, one for each argument of ι , and

$$\begin{array}{c}
\frac{h \in \{\text{Prop, Set, Type}\} \cup \mathcal{C}}{\Sigma \vdash \text{prune}(\xi; h) \triangleright \Sigma} \text{ PRUNE-CONSTANT} \\
\\
\frac{x \in \xi}{\Sigma \vdash \text{prune}(\xi; x) \triangleright \Sigma} \text{ PRUNE-VAR} \\
\\
\frac{\Sigma \vdash \text{prune}(\xi, x; t) \triangleright \Sigma'}{\Sigma \vdash \text{prune}(\xi; \lambda x. t) \triangleright \Sigma'} \text{ PRUNE-LAM} \\
\\
\frac{\Sigma \vdash \text{prune}(\xi, x; t) \triangleright \Sigma'}{\Sigma \vdash \text{prune}(\xi; \forall x. t) \triangleright \Sigma'} \text{ PRUNE-PROD} \\
\\
\frac{\Sigma_0 \vdash \text{prune}(\xi; t) \triangleright \Sigma_1 \quad \Sigma_i \vdash \text{prune}(\xi; t_i) \triangleright \Sigma_{i+1} \quad i \in [1, n]}{\Sigma_0 \vdash \text{prune}(\xi; t \bar{t}_n) \triangleright \Sigma_{i+1}} \text{ PRUNE-APP} \\
\\
\frac{\Sigma_1 \vdash \text{prune}(\xi; t_2) \triangleright \Sigma_2 \quad \Sigma_2 \vdash \text{prune}(\xi, x; t_1) \triangleright \Sigma_3}{\Sigma_1 \vdash \text{prune}(\xi; \text{let } x = t_2 \text{ in } t_1) \triangleright \Sigma_3} \text{ PRUNE-LET} \\
\\
\frac{\Psi \vdash \text{prune_ctx}(\xi; \sigma) \triangleright \Psi}{\Sigma, u : A[\Psi], \Sigma' \vdash \text{prune}(\xi; ?u[\sigma]) \triangleright \Sigma, u : A[\Psi], \Sigma'} \text{ PRUNE-META-NOPRUNE} \\
\\
\frac{u : A[\Psi] \in \Sigma \quad \Psi \vdash \text{prune_ctx}(\xi; \sigma) \triangleright \Psi' \quad \text{FV}(A) \subseteq \Psi'}{\Sigma \vdash \text{prune}(\xi; ?u[\sigma]) \triangleright \Sigma, ?v : A[\Psi'] \cup \{u := v[\text{id}_{\Psi'}]\}} \text{ PRUNE-META}
\end{array}$$

Figure 6: Pruning of meta-variables.

$$\begin{array}{c}
\frac{}{\cdot \vdash \text{prune_ctx}(\xi; \cdot) \triangleright \cdot} \text{ PRUNECTX-NIL} \\
\\
\frac{\text{FV}(t) \in \xi \quad \Psi \vdash \text{prune_ctx}(\xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \text{prune_ctx}(\xi; \sigma, t) \triangleright \Psi', x : A} \text{ PRUNECTX-NOPRUNE} \\
\\
\frac{y \notin \xi \quad \Psi \vdash \text{prune_ctx}(\xi; \sigma) \triangleright \Psi'}{\Psi, x : A \vdash \text{prune_ctx}(\xi; \sigma, y \bar{t}_n) \triangleright \Psi'} \text{ PRUNECTX-PRUNE}
\end{array}$$

Figure 7: Pruning of contexts.

$$\begin{array}{c}
\frac{(p_j, h, \iota) \in \Delta_{\text{db}} \quad \iota := \lambda \bar{x} : \overline{B.k \bar{a}' \bar{v}} \quad v_j = h \bar{u}' \quad \Sigma_1 = \Sigma_0, \overline{?y : B} \quad \Sigma_1; \Gamma \vdash \bar{a} \approx \overline{a' \{ \overline{?y/\bar{x}} \}} \triangleright \Sigma_2}{\Sigma_2; \Gamma \vdash \bar{u} \approx \overline{u' \{ \overline{?y/\bar{x}} \}} \triangleright \Sigma_3 \quad \Sigma_3; \Gamma \vdash c \approx \iota \overline{?y} \triangleright \Sigma_4 \quad \Sigma_4; \Gamma \vdash \bar{t} \approx \overline{t'} \triangleright \Sigma_5} \text{CS-CONSTL} \\
\Sigma_0; \Gamma \vdash p_j \bar{a} c \bar{t} \approx h \bar{u} \bar{t}' \triangleright \Sigma_5 \\
\\
\frac{\begin{array}{c} (p_j, \rightarrow, \iota) \in \Delta_{\text{db}} \\ \iota := \lambda \bar{x} : \overline{B.k \bar{a}' \bar{v}} \quad v_j = u \rightarrow u' \quad \Sigma_1 = \Sigma_0, \overline{?y : B} \quad \Sigma_1; \Gamma \vdash \bar{a} \approx \overline{a' \{ \overline{?y/\bar{x}} \}} \triangleright \Sigma_2 \\ \Sigma_2; \Gamma \vdash t \approx u \{ \overline{?y/\bar{x}} \} \triangleright \Sigma_3 \quad \Sigma_3; \Gamma \vdash t' \approx u' \{ \overline{?y/\bar{x}} \} \triangleright \Sigma_4 \quad \Sigma_4; \Gamma \vdash c \approx \iota \overline{?y} \triangleright \Sigma_5 \end{array}}{\Sigma_0; \Gamma \vdash p_j \bar{a} c \approx t \rightarrow t' \triangleright \Sigma_5} \text{CS-PRODL} \\
\\
\frac{\begin{array}{c} (p_j, s, \iota) \in \Delta_{\text{db}} \\ \iota := \lambda \bar{x} : \overline{B.k \bar{a}' \bar{v}} \quad \Sigma_1 = \Sigma_0, \overline{?y : B} \quad \Sigma_1; \Gamma \vdash \bar{a} \approx \overline{a' \{ \overline{?y/\bar{x}} \}} \triangleright \Sigma_2 \quad \Sigma_2; \Gamma \vdash c \approx \iota \overline{?y} \triangleright \Sigma_3 \end{array}}{\Sigma_0; \Gamma \vdash p_j \bar{a} c \approx s \triangleright \Sigma_3} \text{CS-SORTL} \\
\\
\frac{\begin{array}{c} (p_j, \rightarrow, \iota) \in \Delta_{\text{db}} \quad \iota := \lambda \bar{x} : \overline{B.k \bar{a}' \bar{v}} \quad v_j = x_j \quad \Sigma_1 = \Sigma_0, \overline{?y : B} \\ \Sigma_1; \Gamma \vdash \bar{a} \approx \overline{a' \{ \overline{?y/\bar{x}} \}} \triangleright \Sigma_2 \quad \Sigma_2; \Gamma \vdash x_j \{ \overline{?y/\bar{x}} \} \approx t \triangleright \Sigma_3 \quad \Sigma_3; \Gamma \vdash c \approx \iota \overline{?y} \triangleright \Sigma_4 \end{array}}{\Sigma_0; \Gamma \vdash p_j \bar{a} c \approx t \triangleright \Sigma_3} \text{CS-DEFAULTL}
\end{array}$$

Figure 8: Canonical structures resolution.

proceeds to unify the parameters of the projector with the parameters of the instance. Then, it unifies the arguments of the constant h encountered in the rhs with the ones in the field value. Is it after this point that it equates the structures with the instance. Finally, it unifies the arguments of the function defined by h on both sides of the equation.

The rule CS-PRODL considers the case when the value is a function type. It is similar to the previous one, except that the projector cannot have arguments. The same situation we have in rule CS-SORTL, where the right hand side is a sort (Prop or Type). The last rule CS-DEFAULTL considers the *default* instance, when the value of the j -th field of the instance is a variable.

For conciseness we have omitted the rules for when the projector is in the right hand side.

5.3 Algorithm

The rules shown does not precisely nail the way backtracking is handled, nor the priority of the rules.

First, the algorithm distinguishes three cases:

1. Any of the terms has a meta-variable in the head position. We have three subcases, where every attempt to use rules META-INSTL or META-INSTR is followed by an attempt to use rules META-FOL and META-FOR, respectively.
 - (a) Both terms have the same meta-variable in the head position. Try rules META-SAME and META-SAME-SAME.
 - (b) Both terms have different meta-variables. Try first META-INSTL and then META-INSTR if the variable on the left is the oldest one, or viceversa if it's the newest one.
 - (c) Any other case: try META-INSTL and META-INSTR.

2. If both terms have no arguments, try the rules in Figure 2, except of course APP-FO. Special case if both are lets: first try LET-SAME and if that fails LET-ZETA.
3. In any other case the algorithm tries the following sequence:
 - (a) If any of the sides is a projector of a structure it tries the rules in Figure 8. Except when both sides are the same projector.
 - (b) If both sides have the same number of arguments, try APP-FO.
 - (c) If any of the above failed, try rules in 3 in the order shown in the figure.