Deep-Embedded Unification[⋆]

Évelyne Contejean[1], Julien Forest[2], and Xavier Urbain[2]

[1] LRI, Université Paris-Sud, CNRS, INRIA Futurs, Orsay, France
[2] CÉDRIC – ENSIIE, Évry, France

We present some experiments with C*i*ME/COCCINELLE about standard first-order unification of deep-embedded terms in the Coq proof assistant. We have modelled parts of the usual inference rules for unification, namely Decompose, Merge and Coalesce as Coq functions and we proved their soundness and completeness. Regarding the Generalized Occur-Check, we use the C*i*ME rewriting toolbox as an oracle either to provide a total ordering compatible with Occur-Check, or to exhibit a cycle. Then, using these informations, it is possible to prove in Coq that there is no most general unifier (mgu), or to compute it and prove that it actually enjoys the mgu properties. All together, it means that it is possible to compute an mgu for deep-embedded terms in Coq. This is a first step towards formal proofs of confluence for TRS.

1 Introduction

Unification [9, 13] is widely used in computer science where it plays a fundamental role in many domains: in resolution-based programming [10] for instance, and more generally in many proof techniques. As a consequence, a development in the framework of a sceptical proof assistant (like Coq [15] or Isabelle/HOL [12] for instance) is likely to involve unification mechanisms at several levels.

A first level is the use of the built-in unification mechanism of the proof assistant. This is the easiest way to handle unification but the built-in mechanism might be restricted to simple cases (without equational theory, for instance), and moreover it applies on a *shallow* model only: where the relevant term algebra is the one of the proof assistant.

A second level is closely related to automation of proofs. Actually, sceptical proof assistants are seen as trustworthy: confidence may be enforced by a type-checking kernel or a rewriting kernel for example, but they often lack automation. Note that using a satellite automated prover is not straightforward as its results *must* be checked by the proof assistant (thus sceptical). This implies an intricate task of modelling the relevant notions and of providing enough information from the prover into the proof assistant. Rewriting techniques for instance are good candidates for plugging automation/decision procedures into proof assistants. However, modelling powerful rewriting techniques amounts to dealing with terms algebras that are different from the proof assistant's built-in one. Hence one has to model terms and properties at a *deep* level, and a new unification mechanism has to be defined in that model.

The A3PAT project[3] aims at bringing automation into sceptical proof assistants, in particular by providing trustworthy decision procedures based on term rewriting [6, 8];

[⋆] Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

[3] `http://a3pat.ensiie.fr/`

it uses the COCCINELLE library for rewriting [4]. We present hereafter the approach we developed in this project to model free[4] unification for deep terms into the Coq proof assistant.

One of the challenges in modelling unification is the use of graphs made by the Occur-Check rule. So as to handle this rule efficiently, we use an oracle, namely the C*i*ME rewriting-tool developed in the A3PAT framework. Thus, graph handling and computations on graphs are performed outside the Coq proof assistant, which just has to certify the result. Our approach provides an effective and certified unification algorithm for deep embedded terms in Coq, thus opening the way to proofs of confluence or to completion for relations modelled in deep embedding. Our solution is implemented in the COCCINELLE library for rewriting, developed in the project.

We give some prerequisites about first order terms and first-order unification, and about the Coq proof assistant in Section 2. In particular, we briefly present the Coq library COCCINELLE developed in the project, and how it models terms and properties on them. Then, in Section 3, we describe the architecture of our unification mechanism. We illustrate our approach with two examples respectively for two terms that unify and two terms that do not. Eventually, we conclude and give some perspectives.

2 Preliminaries

We assume the reader to be familiar with basic concepts of first order term algebras [1, 3]. We recall usual notions and give our notations.

Term, Substitution. A *signature* $\mathcal{F}$ is a finite set of *symbols* with arities $\in \mathbb{N}$. Let $\mathcal{X}$ be a set of *variables*; $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of finite *terms* on $\mathcal{F}$ and $\mathcal{X}$, and it is defined as follows: if $x \in \mathcal{X}$ then $x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, if $f \in \mathcal{F}$ with arity n and if $t_1, \ldots, t_n$ belong to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ then $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Let t be a term in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the *set of variables in* t, $\text{Var}(t)$ is inductively defined as: if $x \in \mathcal{X}$ then $\text{Var}(x) = \{x\}$, if $f \in \mathcal{F}$ with arity n and if $t_1, \ldots, t_n$ belong to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ then $\text{Var}(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \text{Var}(t_i)$.

Terms can be seen as trees: we denote Λ the root position, and $\Lambda(t)$ is the symbol at root position in term t. We write $t|_p$ for the subterm of t at position p and $t[u]_p$ for term t where $t|_p$ has been replaced by u.

A substitution σ of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is a map from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$ that is the identity except on a finite set of variables, namely the *domain* of σ, denoted $\text{Dom}(\sigma)$. If $\text{Dom}(\sigma) = \{x_1, \ldots, x_n\}$, then σ is completely defined whenever are known the values $t_1, \ldots, t_n$ to which $x_1, \ldots, x_n$ map. In that case, we denote $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. We write, as it is usual, $x\sigma$ for $\sigma(x)$.

A substitution σ extends naturally to a *unique* function $\mathscr{H}_\sigma$ from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the following way: $\mathscr{H}_\sigma(x) = x$ if $x \notin \text{Dom}(\sigma)$, $\mathscr{H}_\sigma(x_i) = x_i\sigma$ if $x_i \in \text{Dom}(\sigma)$, and $\mathscr{H}_\sigma(f(s_1, \ldots, s_m)) = f(\mathscr{H}_\sigma(s_1), \ldots, \mathscr{H}_\sigma(s_m))$ if $f(s_1, \ldots, s_m) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Hereafter, for the sake of clarity, we gather substitutions and their extensions under the same notion: substitutions.

[4] That is without any equational theory.

2.1 Unification

Unification Problem, Solved Form. let $\mathcal{T}(\mathcal{F},\mathcal{X})$ be a term algebra. A *unification problem* is either of the form: $\top$, $\bot$, or $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$. Every substitution solves $\top$ and no substitution solves $\bot$. A substitution σ is a *solution* of $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$ if for all $i = 1, \ldots, n$, $s_i\sigma = t_i\sigma$ holds. We denote $\mathscr{U}(P)$ the set of solutions for a unification problem P.

Two problems for $\mathcal{T}(\mathcal{F},\mathcal{X})$, P_1 and P_2, are said to be *equivalent* whenever $\mathscr{U}(P_1) = \mathscr{U}(P_2)$.

The main property of unification problems is the following:

Proposition 1 ([2,11]). *Let P be a unification problem for a term algebra $\mathcal{T}(\mathcal{F},\mathcal{X})$, then one of the following properties holds for $\mathscr{U}(P)$: either $\mathscr{U}(P) = \emptyset$, or there is a μ that is unique up to renaming of variables and such that $\mathscr{U}(P) = \{\mu\theta \mid \theta$ is a substitution$\}$.*

This property is actually shown using an algorithm that transforms unification problems into equivalent ones enjoying immediate solutions: problems in Solved Form.

Let P be a unification problem for $\mathcal{T}(\mathcal{F},\mathcal{X})$, P is *in Solved Form* if either $P \equiv \top$, or $P \equiv \bot$, or $P \equiv x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ where x_i are pairwise disjoint variables, and t_j are terms that contain no occurrence of x_i. In particular:

Proposition 2 ([2,11]). *Let $P \equiv x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ in Solved Form, and let θ be the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, then $\mathscr{U}(P) = \{\mu\theta \mid \theta$ is a substitution$\}$.*

For a problem P equivalent to the Solved Form $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$, the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is said to be the *most general unifier* for P, denoted $mgu(P)$.

We shall also use a more technical definition: let P be a unification problem for $\mathcal{T}(\mathcal{F},\mathcal{X})$, P is *in Sequential Solved Form* if either $P \equiv \top$, or $P \equiv \bot$, or $P \equiv x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ where x_i are pairwise disjoint variables.

Transformation Algorithm and Properties. The steps that transform a unification problem into a problem in Solved Form are usually expressed as the set of rules in Figure 1 [2,11]. The following proposition is well-known.

Proposition 3 ([2,11]). *The transformation algorithm given by rules in Figure 1 terminates, is sound, and is complete.*

However, so as to use this unification algorithm in a sceptical proof assistant, we have to provide a mechanical proof of Proposition 3.

2.2 The Coq proof assistant and the COCCINELLE library

The Coq proof assistant. Coq is a proof assistant based on *type theory*. It features: 1) A *formal language* dealing with objects, properties and proofs in a unified way; all these are represented as terms of an expressive λ-calculus: the *Calculus of Inductive Constructions* (CIC) [7]. λ-abstraction is denoted `fun x:T ⇒ t`, and application is denoted `t u`.

Trivial $\dfrac{s = s}{\top}$

Decompose $\dfrac{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}{s_1 = t_1 \wedge \ldots \wedge s_n = t_n}$

Clash $\dfrac{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)}{\perp}$ if $f \neq g$

Coalesce $\dfrac{x = y \wedge P}{x = y \wedge P\{x \mapsto y\}}$ if $x, y \in \mathrm{Var}(P)$

Replace $\dfrac{x = s \wedge P}{x = s \wedge P\{x \mapsto s\}}$ if $x \in \mathrm{Var}(P) \setminus \mathrm{Var}(s)$ and $s \notin \mathcal{X}$

Merge $\dfrac{x = s \wedge x = t}{x = s \wedge s = t}$ if $x \in \mathcal{X}$, $s, t \notin \mathcal{X}$ and $|s| \leq |t|$

Occur-Check $\dfrac{x_1 = t_1[x_2]_{p_1} \wedge \ldots x_n = t_n[x_1]_{p_n}}{\perp}$ if $p_1 \cdot \ldots \cdot p_n \neq \Lambda$

Fig. 1. Inference rules for transformation towards Solved Form

2) A *proof checker* which checks the validity of proofs written as CIC terms. In this framework, a term is a *proof* of its type, and checking a proof consists in typing a term. The correctness of the tool relies on this type checker, which is a small kernel of 5 000 lines of Ocaml code.

For example the following simple terms are proofs of the following (tautological) types (the implication arrow → is right associative): the identity function `fun x:A ⇒ x` is a proof of `A →A`, and `fun (x:A) (f:A→B) ⇒ f x` is a proof of `A →(A →B) →B`.

Coq enjoys the ability to define powerful *inductive types* to express inductive data types and inductive properties. For instance, the data type `nat` of natural numbers may be defined using the following inductive types, where `O` and `S` (successor) are the two constructors, and `even` is the property of being an even natural number.

```
Inductive nat : Set := | O : nat | S : nat →nat.
Inductive even : nat →Prop := | even_O : even O
 | even_S : ∀n : nat, even n →even (S (S n)).
```

Hence the term `even_S (S (S O)) (even_S O (even_O))` is of type `even (S (S (S (S O))))` so it is a proof that 4 is even.

We propose in this work a way to deal with free unification on terms algebras that one defines in the proof assistant, that is in deep embedding.

The COCCINELLE library. A deep modelling of terms is formalized in the public Coq library called COCCINELLE [4]. To start with, it contains a modelling of the mathematical notions needed for rewriting, such as term algebras, generic rewriting, generic and AC equational theories and RPO with status. It contains also proofs of properties of these notions, for example that RPO is well-founded whenever the underlying precedence is.

Moreover COCCINELLE is intended to be a mirror of the C*i*ME tool in Coq; this means that some of the types of COCCINELLE (terms, etc.) are translated from C*i*ME (in Ocaml) to Coq, as well as some functions (AC matching)[5]. Translating functions and proving their full correctness obviously provide a certification of the underlying algorithm. Note that some proofs may require that *all* objects satisfying a certain property have been built: for instance in order to prove local confluence of a TRS, one need to get all critical pairs, hence a *unification algorithm* which is complete.

Since module systems in Ocaml and Coq are similar, both C*i*ME and COCCINELLE have the same structure, except that C*i*ME contains only types and functions whereas COCCINELLE also contains properties over these types and functions.

2.3 Deep Embedded Terms in Coq using COCCINELLE

A signature is defined by a set of symbols with decidable equality, and a function `arity` mapping each symbol to its arity.

[5] It should be noticed that COCCINELLE is not a *full* mirror of C*i*ME: some parts of C*i*ME are actually search algorithms for proving for instance equality of terms modulo a theory or termination of TRSs. These search algorithms are much more efficient when written in Ocaml than in Coq, they just need to provide a *trace* for COCCINELLE.

The arity is not simply an integer, it mentions also whether a symbol is free of arity n, AC or C (of implicit arity 2) since there is a special treatment in the AC/C case.

```
Inductive arity_type : Set :=
  | Free : nat →arity_type | AC : arity_type | C : arity_type.

Module Type Signature.
  Declare Module Export Symb : decidable_set.S.
  Parameter arity : Symb.A →arity_type.
End Signature.
```

Up to now, our automatic proof generator does not deal with AC nor C symbols, hence in this work all symbols have an arity `Free n`. However, AC/C symbols are used in other parts of COCCINELLE, in particular the formalization of *AC matching* [5].

A term algebra is a module defined from its signature `F` and the set of variables `X`.

```
Module Type Term.
  Declare Module Import F : Signature.
  Declare Module Import X : decidable_set.S.
```

Terms are defined as variables or symbols applied to lists of terms. Lists are built from two constructors `nil` and `::`, and enjoy the usual `[ x ; y; ...]` notation.

```
Inductive term : Set :=
 | Var : variable →term | Term : symbol →list term →term.
```

This type allows to share terms in a standard representation as well as in a canonical form; but this also implies that terms may be ill-formed w.r.t. the signature. The module contains decidable definitions of well-formedness.

The term module type contains other useful definitions and properties that we omit here for the sake of clarity. The COCCINELLE library contains also a *functor* `term.Make` which, given a signature and a set of variables, returns a module of type `Term`. We will not show its definition here.

```
Module Make (F1 : Signature) (X1 : decidable_set.S) : Term.
```

Now we want to solve unification problem on those terms.

3 Certified Unification

In the following we propose an encoding of the rules presented in Figure 1 in the Coq proof assistant. We describe our approach and give mechanical proofs that the algorithm thus described terminates and that it is sound and complete. Eventually, we provide a solution for unification of deep embedded terms in the Coq proof assistant: we may compute a (certified) most general unifier, or a proof that there is none, with full automation.

One of the main difficulties lies in the Occur-Check rule. Applying this rule amounts to finding cycles in a graph of variables, or to providing an ordering compatible with Occur-Check. In order to avoid the cost of such computation in the proof assistant, we use the C*i*ME rewriting tool as an oracle. Called by Coq on a particular problem, C*i*ME performs the cycle analysis required by Occur-Check and returns a proof of its result.

Our general approach is schematized in Figure 2. It is based on three components:

1. A deep modelling of all rules except Occur-Check in the COCCINELLE library, with relevant properties and proofs;
2. A mechanism which generates traces for each call to the oracle regarding Occur-Check;
3. A Coq tactic which computes an mgu or prove that there is none, by using the previous two components.

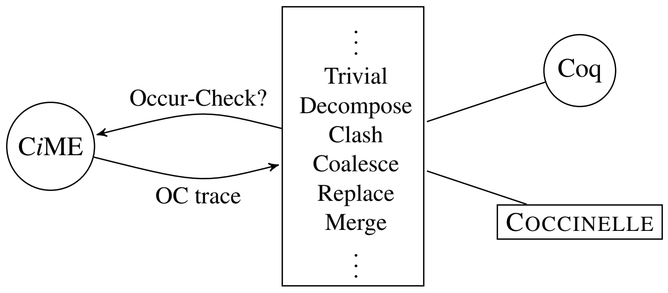

Fig. 2. Global scheme of our approach

3.1 Modelling of rules

The inference rules (except the Occur-Check and Replace rules) are gathered in a single Coq definition (see Figure 3) which is applied to a specific data structure for unification problems. We define a unification problem as a pair made of: a *solved* part, which is a substitution, and an *unsolved* part, which is a list of equations.

When the list of equations is empty, the problem is in *Sequential* Solved Form, and ready for the call to the oracle for occur-check.

Otherwise, according to the shape of the first equation in the unsolved part, `decomposition_step` selects which inference rule to apply. If the first equation `(s,t)` is a trivial one, it is simply removed as in Trivial (`(*** TRIVIAL ***)` in Figure 3). If the terms `s` and `t` are variables, the rule Coalesce applies (`(*** COALESCE ***)` in Figure 3). If `s` is a variable `x` and `t` a composite term, if x has no value in then $x = t$ is moved from the unsolved part into the solved part (this does not correspond to any inference rule), if x has a value in the solved part, the rule Merge (`(*** MERGE ***)` in Figure 3) applies. If s and t are both composite terms, either Decompose or Clash apply (`(*** DECOMPOSE ***)` or `(*** CLASH ***)` in Figure 3).

3.2 The oracle

As mentioned, the only part of the unification process which have not been modeled in COCCINELLE is the occur-check. We use C*i*ME as an oracle for Coq. The communica-

```
Definition decomposition_step (pb : unification_problem) : exc unification_problem :=
 match pb.(unsolved_part) with
 | nil ⇒@Not_applicable _ pb
 | (s,t) :: l ⇒
   if (eq_term_dec s t : sumbool _ _)
   then (*** TRIVIAL ***) Normal _ (mk_pb pb.(solved_part) l)
   else
   match s, t with
   | Var x, Var y ⇒(*** COALESCE ***)
    let x_maps_to_y := (x, t) :: nil in
    let solved_part' :=
       map_subst (fun _ v_sigma ⇒apply_subst ((x,t) :: nil) v_sigma)
                    pb.(solved_part) in
    let l' := map (fun uv ⇒
                    match uv with
                    | (u, v) ⇒(apply_subst ((x,t) :: nil) u,
                                    apply_subst ((x,t) :: nil) v)
                   end) l in
     match find eq_variable_dec x solved_part' with
     | Some x_val ⇒Normal _ (mk_pb ((x, t) :: solved_part') ((t, x_val) :: l'))
     | None ⇒Normal _ (mk_pb ((x, t) :: solved_part') l')
     end
   | Var x, (Term _ _ as u) ⇒
    match find eq_variable_dec x pb.(solved_part) with
     | None ⇒Normal _ (mk_pb ((x,u) :: pb.(solved_part)) l)
     | Some x_val ⇒(*** MERGE ***)
      if lt_ge_dec (T.size u) (T.size x_val)
      then (* u < x_val *) Normal _ (mk_pb ((x,u) :: pb.(solved_part)) ((x_val,u) :: l)
      else (* x_val <= u *) Normal _ (mk_pb pb.(solved_part) ((x_val,u) :: l))
    end
   | (Term _ _ as u), Var x ⇒
    match find eq_variable_dec x pb.(solved_part) with
     | None ⇒Normal _ (mk_pb ((x,u) :: pb.(solved_part)) l)
     | Some x_val ⇒(*** MERGE ***)
      if lt_ge_dec (T.size u) (T.size x_val)
      then Normal _ (mk_pb ((x,u) :: pb.(solved_part)) ((x_val,u) :: l))
      else Normal _ (mk_pb pb.(solved_part) ((x_val,u) :: l))
    end
   | Term f l1, Term g l2 ⇒
     if eq_symbol_dec f g
     then match eq_nat_dec (length l1) (length l2) with
         | left L ⇒(*** DECOMPOSE ***)
          Normal _ (mk_pb pb.(solved_part) (combine _ l l1 l2 L))
         | right _ ⇒(*** CLASH ***) @No_solution _
        end
    else (*** CLASH ***) @No_solution _
   end
 end.
```

Fig. 3. The Coq definition gathering inference rules

tion between C*i*ME and Coq is ensured via the `external` tactic of Coq. It allows the user to communicate with an external tool using XML via the CIC dtd[6].

The communication process can be sketched as follows:

1. Firstly, Coq encodes the terms to give to the external oracle using the CIC dtd and gives them together with a "command name";
2. Then, the oracle parses the request and answers it. In our case, we use a standard occur-check algorithm.
3. Finally, the oracle must print its answer (and only this) onto its standard output buffer. This answer is usually a CIC term encoded using the CIC dtd. In our case, the answer is a pair consisting of a boolean flag (which encodes the presence of a cycle) and a list of variables which describe depending on the case either the cycle, or a total ordering compatible with Occur-Check.

3.3 The tactic

We define a tactic `unify` which allows the user to prove a goal having the following type (where `a` and `b` are COCCINELLE terms):

```
{sigma : substitution |
 (*** Either we can find a substitution which is an mgu
      of a and b ***)
 (∃ c : list variable,
   ∀tau : substitution,
   is_a_solution (mk_pb nil ((a, b) :: nil)) tau <->
   (∃ theta : substitution,
     ∀x : variable,
     apply_subst tau (Var x) =
     apply_subst theta (apply_subst (acc_inst c sigma) (Var x))))} +
(*** Either we can prove that a and b do not unify
    that is ∀tau, tau is NOT a unifier of a and b ***)
 {(∀ tau : substitution,
  is_a_solution (mk_pb nil ((a, b) :: nil)) tau ->False)}
```

The tactic `unify` uses three different steps.

Firstly, we decompose the unification problem `mk_pb nil ((a,b) :: nil)` in order to obtain a normal form (that is a Sequential Solved Form). This step is done by unfolding the `decomposition_step` function as far as possible.

Then, we use the aforementioned Coq feature `external` in order to call C*i*ME and to get its answer back.

Finally, depending on the answer, we call the generic tactic `oc_cycle` (which proves that we cannot find a unifier) or `oc_ok` (which allows us to define the mgu of `a` and `b` and to prove its properties).

4 Examples

We illustrate our approach with two examples from the TPTP database.

[6] available within the Coq distribution.

The following shows that `multiply (V, X, X)` and `multiply (Y, Y, W)` unify.

```
Goal {sigma : substitution |
   is_a_solution
     (mk_pb nil ((multiply (V, X, X), multiply
(Y, Y, W)) :: nil)) sigma ∧
   (∃ c : list variable,
     ∀ tau : substitution,
     is_a_solution
       (mk_pb nil ((multiply (V, X, X), multiply
(Y, Y, W)) :: nil)) tau <→
      (∃ theta : substitution,
        ∀ x : variable,
        apply_subst tau (Var x) =
        apply_subst theta (apply_subst (acc_inst c sigma) (Var x))))} +
   {(∀ tau : substitution,
    is_a_solution
      (mk_pb nil ((multiply (V, X, X), multiply
(Y, Y, W)) :: nil)) tau →
    False)}.
```

Using our approach, the proof is just a call to our tactic:

```
Proof.
 unify (multiply(V,X,X)) (multiply(Y,Y,W)).
Defined.
```

The next example shows that `multiply(X,V,inverse(V))` and `multiply(W,Y,Y)` do NOT unify.

```
Goal {sigma : substitution |
   is_a_solution
     (mk_pb nil
        ((multiply (X, V, inverse (V)), multiply
(W, Y, Y)) :: nil)) sigma ∧
   (∃ c : list variable,
     ∀ tau : substitution,
     is_a_solution
       (mk_pb nil
          ((multiply (X, V, inverse (V)), multiply
(W, Y, Y)) :: nil))
      tau <→
      (∃ theta : substitution,
        ∀ x : variable,
        apply_subst tau (Var x) =
        apply_subst theta (apply_subst (acc_inst c sigma) (Var x))))} +
   {(∀ tau : substitution,
    is_a_solution
      (mk_pb nil
         ((multiply (X, V, inverse (V)), multiply
(W, Y, Y)) :: nil)) tau →
```

```
    False)}.
```

Again the proof consists only in a call to our tactic:

```
Proof.
 unify (multiply(X,V,inverse(V))) (multiply(W,Y,Y)).
Defined.
```

Notice that those two goals are solved using exactly the same tactic (namely `unify`).

5 Conclusion

We presented an approach to perform unification on deep-embedded terms in the Coq proof assistant. All rules but Occur-Check are defined and proved to be sound and complete in our library COCCINELLE. One original side of our method is the call to an oracle for the Occur-Check rule. Regarding the oracle, we use the C*i*ME rewriting toolbox, which returns a trace of Occur-Check. Eventually, we provide a Coq tactic which computes an mgu for a given unification problem or returns a proof that there is none.

In 1992, Joseph Rouyer [14] developped a first certified unification algorithm: due to the early state of the Coq assistant (version 4.10 and then 5.6) its terms were very involved, and in contrast to our the algorithm was not intended to actually run on terms, but only to be extracted.

An interesting perspective is the certification of confluence for rewriting relations on deep-embedded terms, for which a complete unification algorithm is compulsory. It is also a first step towards more intricate unification algorithms like AC-unification (i.e. modulo Associativity and Commutativity) for instance, which are not built-in in all proof assistants, and thus require deep-embedding.

References

1. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
2. Franz Baader and Wayne Snyder. *Handbook of Automated Reasoning*, chapter Unification Theory, pages 445–533. North-Holland, 2001.
3. Paul Moritz Cohn. *Universal Algebra*. D. Reidel, Dordrecht, Holland, second edition, 1981. ISBN : 90-277-1213-1.
4. Évelyne Contejean. The Coccinelle library for rewriting. `http://www.lri.fr/~contejea/Coccinelle/coccinelle.html`.
5. Évelyne Contejean. A certified AC matching algorithm. In Vincent van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84, Aachen, Germany, June 2004. Springer-Verlag.
6. Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, September 2007. Springer-Verlag.

7. Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
8. Pierre Courtieu, Julien Forest, and Xavier Urbain. Certifying a Termination Criterion Based on Graphs, Without Graphs. In César Muñoz and Otmane Ait Mohamed, editors, *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, Lecture Notes in Computer Science, Montréal, Canada, August 2008. Springer-Verlag. To appear.
9. Jacques Herbrand. Recherches sur la théorie de la démonstration. Thèse d'État, Univ. Paris, 1930. Also in: Écrits logiques de Jacques Herbrand, PUF, Paris, 1968.
10. Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *International Federation for Information Processing (IFIP) Congress*, pages 569–574, Stockholm, Sweden, August 1974. North-Holland. ISBN : 0-7204-2803-3.
11. Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
12. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
13. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
14. Joseph Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Technical Report RR-1795, INRIA, 1992.
15. The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.1*, February 2007. `http://coq.inria.fr`.