A machine checked model of MGU axioms: applications of finite maps and functional induction

Sunil Kothari and James Caldwell *
Department of Computer Science
4093 Engineering Hall
University of Wyoming
Laramie, WY 82071-3315, USA

Abstract

The most general unifier (MGU) of a pair of terms can be specified by four axioms. In this paper we generalize the standard presentation of the axioms to specify the MGU of a list of equational constraints and we formally verify that the unification algorithm satisfies the axioms. Our constraints are equalities between terms in a language of simple types. We model substitutions as finite maps from the Coq library *Coq.FSets.FMapInterface*. Since the unification algorithm is general recursive we show termination using a lexicographic ordering on lists of constraints. Coq's method of functional induction is the main proof technique used in proving the axioms.

1 Introduction

As a step toward a comprehensive library of theorems about unification and substitution we verify the unification algorithm over a language of simple types. We take the axioms presented in [UN09] as our specification and show that the first-order unification algorithm is a model of the axioms. In the formalization we represent substitutions using Coq's finite map library. This verification is a step toward a formal verification of an extended version of Wand's constraint based type reconstruction algorithm [KC08]. The main idea behind our approach there is to have a multi-phase unification in the constraint solving phase. By formalizing the first-order unification, we will be able to extend the first-order unification to this multi-phase unification. We believe that the verification described here may be of interest in and of itself to researchers in the unification community.

In recent literature on machine certified proof of correctness of type inference algorithms (mostly on substitution-based type reconstruction algorithms), the most general unifier is axiomatized by a set of four axioms. In this paper, we follow Urban and Nipkow's [UN09] axioms.

$$\begin{array}{ll}
(i) & mgu\ \sigma\ (\tau_1 \stackrel{\mathrm{e}}{=} \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \\
(ii) & mgu\ \sigma\ (\tau_1 \stackrel{\mathrm{e}}{=} \tau_2) \land \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \delta . \sigma' \approx \sigma \circ \delta \\
(iii) & mgu\ \sigma\ (\tau_1 \stackrel{\mathrm{e}}{=} \tau_2) \Rightarrow \mathsf{FTV}\ (\sigma) \subseteq \mathsf{FVC}\ (\tau_1 \stackrel{\mathrm{e}}{=} \tau_2) \\
(iv) & \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'.\ mgu\ \sigma'(\tau_1 \stackrel{\mathrm{e}}{=} \tau_2)
\end{array}$$

We give an axiomatic presentation of substitutions and provide a model using substitutions formalized with the Coq's Finite map library. Using this presentation of substitutions, we prove the correctness of first order unification - by showing that the unification algorithm satisfies the four axioms. Since the unification algorithm is not structurally recursive, we have to also prove the termination of the first-order unification algorithm by giving a measure and showing that it reduces on each recursive call. The entire verification is done in Coq [Cdt07] - a theorem prover based on calculus of inductive constructions [CH88].

The rest of this paper is organized as follows: Section 2 introduces the concepts and terminologies needed for this paper and includes a description of substitutions as finite functions. Section 3 describes

*The work of the authors was partially supported by NSF 0613919.

the formalization of first-order unification algorithm in Coq. Section 4 describes the proof that the unification algorithm satisfies the four axioms and presents a number of supporting lemmas. Section 5 summarizes our current work and mentions further work.

2 Types and Substitutions

Unification is implemented here over a language of types for (untyped) lambda terms. The language of types is given by the following grammar:

$$\tau ::= \quad \mathsf{TyVar}\ x \mid \quad \tau_1 \to \tau_2$$

where $x \in \mathit{Var}$ is a variable and $\tau_1, \tau_2 \in \tau$ are type terms.

Thus, a type is either a type variable or a function type.

We have adopted the following conventions in this paper. Atomic types (of the form $\mathsf{TyVar}\ x$) are denoted by α, β, α' etc., compound types by τ, τ', τ_1 etc., substitutions by $\sigma, \sigma', \sigma_1$ etc. By convention, the type constructor $\to$ associates to the right. List append is denoted by $++$. We use $\{\_\}$ to denote small finite substitutions. For example, a substitution that binds x to τ and y to τ' is denoted as $\{x \mapsto \tau, y \mapsto \tau'\}$. When necessary we follow Coq's namespace conventions; every library function has a qualifier which denotes the library it belongs to. For example, $M.map$ is a function from the finite maps library whereas $List.map$ is a function from list library.

The work described here is being extended to the polymorphic case and so the language of types will be extended to include universally quantified type variables. Anticipating this, although all type variables occurring in types as defined here are free, we define the list of *free variables of a type* (FTV) as:

$$\begin{array}{rcl} \mathsf{FTV}\ (\mathsf{TyVar}\ x) & = & [x] \\ \mathsf{FTV}\ (\tau \to \tau') & = & \mathsf{FTV}\,(\tau)\ ++\ \mathsf{FTV}\,(\tau') \end{array}$$

We also have a notion of equational constraints of the form $\tau \stackrel{\mathrm{e}}{=} \tau'$.
The list of *free variables of a constraint list*, denoted by FVC, is given as:

$$\begin{array}{rcl} \mathsf{FVC}\ [\,] & = & [\,] \\ \mathsf{FVC}\ ((\tau_1 \stackrel{\mathrm{e}}{=} \tau_2) :: \mathbb{C}) & = & \mathsf{FTV}\,(\tau_1)\ ++\ \mathsf{FTV}\,(\tau_2)\ ++\ \mathsf{FVC}\,(\mathbb{C}) \end{array}$$

2.1 Substitutions

Substitutions are finite functions mapping type variables to types. Application of a substitution to a type is defined as:

$$\begin{array}{rcl} \sigma\ (\mathsf{TyVar}(x)) & \stackrel{def}{=} & \mathit{if}\ \langle x, \tau\rangle\ \in\ \sigma\ \mathit{then}\ \tau\ \mathit{else}\ \mathsf{TyVar}(x) \\ \sigma\ (\tau_1 \to \tau_2) & \stackrel{def}{=} & \sigma(\tau_1) \to \sigma(\tau_2) \end{array}$$

Thus, if a variable x is not in the domain of the substitution, it lifts that variable to $\mathsf{TyVar}(x)$. Application of a substitution to a constraint is defined similarly:

$$\sigma(\tau_1 \stackrel{\mathrm{e}}{=} \tau_2) \stackrel{def}{=} \sigma(\tau_1) \stackrel{\mathrm{e}}{=} \sigma(\tau_2)$$

Since substitutions are functions their equality is extensional; they are equal if they behave the same on all type variables.

$$\sigma \approx \sigma' \stackrel{def}{=} \forall \alpha.\ \sigma(\alpha) = \sigma'(\alpha)$$

Two type terms τ_1 and τ_2 are *unifiable* if there exists a substitution σ such that $\sigma(\tau_1) = \sigma(\tau_2)$. In such a case, σ is called a *unifier*. More formally, we denote solvability of a constraint by $\models$ (read solves'). We write $\sigma \models (\tau_1 \stackrel{\mathrm{e}}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$. We extend the solvability notion to a list of constraints and we write $\sigma \models \mathbb{C}$ if and only if for every $c \in \mathbb{C}$, $\sigma \models c$. A unifier σ is the *most general unifier* if there is a substitution σ' such that for any other unifier σ'', $\sigma \circ \sigma' \approx \sigma''$.

We define composition of substitutions as follows:

$$\sigma \circ \sigma' = \lambda\tau.\sigma'(\sigma(\tau))$$

Composition of substitutions is associative but not commutative.

2.2 Implementing Substitutions as Finite Maps

The representation of substitutions plays an important role in the formalization exercise. In the verification literature substitutions have been represented as functions, lists of pairs, and as sets of pairs. The literature on representing substitutions as finite maps is sparse.

We use the Coq finite map library (*Coq.FSets.FMapInterface*) which provides an axiomatic presentation of finite maps and a number of supporting implementations. It does not provide an induction principle and forward reasoning is the predominate style of proof required to use the library. The fact that we were able to reason about substitution composition without using induction principle explains the power and expressiveness of the existing library. We found we did not need induction to reason on finite maps, though there are natural induction principles we might have proved [CS95, MW85]. The most recent release of the library (v. 8.2) supports one.

To consider the domain and range of a finite function (and this is the key feature of the function being finite) we use the finite map library function $\mathsf{M.elements}$. $\mathsf{M.elements}(\sigma)$ returns the list of pairs corresponding to the finite map σ. The domain and the range of a substitution are defined as:

Definition 1. [Domain subst]
$\mathsf{dom}(\sigma) \stackrel{def}{=} \mathsf{List.map}\ (\lambda t.\mathsf{fst}\ (t))\ (\mathsf{M.elements}(\ \sigma))$

Definition 2. [Range subst]
$\mathsf{range}(\sigma) \stackrel{def}{=} \mathsf{List.flat\_map}\ (\lambda t.\mathsf{FTV}\ (\mathsf{snd}\ (t)))\ (\mathsf{M.elements}(\ \sigma))$

The function $\mathsf{List.flat\_map}$ is defined in the Coq library *Coq.List.List* as:
$\mathsf{flat\_map}\ f\ [\,] = [\,]$
$\mathsf{flat\_map}\ f\ h :: t = (f\ h) +\!+\ \mathsf{flat\_map}\ f\ t$

The free type variables of a substitution, denoted by $\mathsf{FTV\_subst}$, is defined in terms of domain and range of a substitution as:

Definition 3. [Free type variables of a substitution]
$\mathsf{FTV\_subst}(\sigma) \stackrel{def}{=} \mathsf{dom}(\sigma) +\!+\ \mathsf{range}(\sigma)$

Applying a substitution σ' to a substitution σ means applying the σ' to the range elements of σ.

Definition 4. [Apply subst subst]
$\sigma'(\sigma) \stackrel{def}{=} \mathsf{M.map}\ (\lambda t.\sigma'(t))\ \sigma$

The function $\mathsf{choose\_subst}$ chooses a binding from the two different bindings. The binding in the first argument is preferred over the binding in the second argument.

Definition 5. [choose subst]
$$\begin{array}{ll}\mathsf{choose\_subst\ T1\ T2} \stackrel{def}{=} & \mathsf{match}\ (T1, T2)\ \mathsf{with} \\ & \mid \mathsf{Some\ T3, SomeT4} \Rightarrow \mathsf{Some\ T3} \\ & \mid \mathsf{Some\ T3, None} \Rightarrow \mathsf{Some\ T3} \\ & \mid \mathsf{None, Some\ T4} \Rightarrow \mathsf{Some\ T4} \\ & \mid \mathsf{None, None} \Rightarrow \mathsf{None}\end{array}$$

Definition 6. [Subst diff]
$\mathsf{subst\_diff}\ \sigma\ \sigma' \stackrel{def}{=} \mathsf{M.map2\ choose\_subst}\ \sigma\ \sigma'$

The function $\mathsf{M.map2}$ is defined in Coq library as the function that takes two maps σ, σ', and a function ($\mathsf{choose\_subst}$) and creates a map whose binding belongs to either one of σ or σ' depending upon the function.

Definition 7. [compose subst]
$\sigma \circ \sigma' \stackrel{def}{=} \mathsf{subst\_diff}\ \sigma'(\sigma)\ \ \sigma'$

Theorem 1. [Composition Apply]
$\forall \tau.(\sigma \circ \sigma')(\tau) = \sigma'(\sigma(\tau))$

Proof. The proof is by induction on the type τ followed by case analysis on the binding's occurrence in the composed substitution and the individual substitutions. $\square$

Interestingly, the base case (when τ is a type variable) is harder compared to the inductive case (τ is a compound type). Incidentally, the same theorem has been formalized in Coq [DM99], where substitutions are represented as a list of pairs, but required 600 proof steps. We proved in about 100 proof steps.

3 Unification

3.1 The algorithm

We use the following standard presentation of first-order unification algorithm.

$$
\begin{array}{lcl}
\mathsf{unify}\ (\alpha \overset{e}{=} \alpha) \cup \mathbb{C} & = & \mathsf{unify}\ \mathbb{C} \\
\mathsf{unify}\ (\alpha \overset{e}{=} \tau) \cup \mathbb{C} & = & \mathsf{if}\ \alpha\ \mathsf{occurs\ in}\ \tau\ \mathsf{then\ Fail\ else}\ \{\alpha \mapsto\ \tau\} \circ \mathsf{unify}\ (\{\alpha \mapsto \tau\}\mathbb{C}) \\
\mathsf{unify}\ (\tau \overset{e}{=} \alpha) \cup \mathbb{C} & = & \mathsf{if}\ \alpha\ \mathsf{occurs\ in}\ \tau\ \mathsf{then\ Fail\ else}\ \{\alpha \mapsto\ \tau\} \circ \mathsf{unify}\ (\{\alpha \mapsto \tau\}\mathbb{C}) \\
\mathsf{unify}\ (\tau_1 \rightarrow \tau_2 \overset{e}{=} \tau_3 \rightarrow \tau_4) \cup \mathbb{C} & = & \mathsf{unify}\ (\tau_1 \overset{e}{=} \tau_3, \tau_2 \overset{e}{=} \tau_4 \cup \mathbb{C}) \\
\mathsf{unify}\ \emptyset & = & Id
\end{array}
$$

The algorithm presented above is still not quite ready for formalization since we have not represented failure. Coq provides a *option* type (also available in OCaml as a standard data type) to allow for failure.

$$\mathsf{Inductive\ option\ (A : Set)\ :\ Set\ :=\ Some\ (\_ : A)\ |\ None.}$$

We use the option None to indicate failure and in the result Some(σ), σ is the resulting substitution. The unification algorithm is fully formalized as shown in Appendix 7.1.

The above presentation of the unification algorithm is general recursive, *i.e.* the recursive call is not necessarily on structurally smaller argument. Various papers have described the non-structural recursion aspect of first-order unification [Bov01, McB03]. To allow Coq to accept our definition of unification, we have to either give a measure that shows that recursive argument is smaller or give a well-founded ordering relation. We chose the latter. The {`wf meaPairMLt`} annotation in the specification is precisely that. The advantage of specifying the unification algorithm as shown above is that we get an induction principle for free. This induction principle will be used later in a Coq tactic named as functional induction for the axiom proofs. We will have more to say about the induction principle and the tactic later in Section 4.1. satisfy

3.2 Termination

Since the unification algorithm is general-recursive, we need to give an ordering that is well-founded. We use the lexicographic ordering ($\prec_3$) on the triple (see below). The lexicographic ordering on the two triples $\langle n_1, n_2, n_3\rangle$ and $\langle m_1, m_2, m_3\rangle$ is defined as

$\langle n_1, n_2, n_3\rangle \prec_3 \langle m_1, m_2, m_3\rangle \overset{def}{=} (n_1 < m_1) \lor (n_1 = m_1 \land n_2 < m_2) \lor (n_1 = m_1 \land n_2 = m_2 \land n_3 < m_3)$,

where $<, =$ are the ordinary less-than inequality and equality on naturals.

Our triple is similar to the triple proposed by others [Bov01, BS01, Apt03], but a little simpler. The triple is $\langle\!| \,|C_{FVC}|, |C_{\rightarrow}|, |C|\, |\!\rangle$, where

- $|C_{FVC}|$ - number of *unique* free variables in a constraint list;
- $|C_{\rightarrow}|$ - total number of arrows in the constraint list;
- $|C|$ - the length of the constraint list.

Table 1 shows how these components vary depending on constraint at the head of the constraint list. The table closely follows the reasoning we did to satisfy the proof obligations (shown in the Appendix 7.3) generated by the above specification. We use -, $\uparrow$, $\downarrow$ to denote the component is unchanged, increased or decreased, respectively. We could have used the finite sets here (for counting the unique free variables of a constraint list). But we went ahead with the unique lists (in Coq they are referred as NoDup). We found the existing Coq list library offering plenty of support for lists in general, and unique lists in

Original call	Recursive call	Conditions, if any	$\lvert C_{FVC}\rvert$	$\lvert C_{\to}\rvert$	$\lvert C\rvert$
$(\alpha \stackrel{e}{=} \alpha) :: \mathbb{C}$	$\mathbb{C}$	$\alpha \in (\text{FVC } \mathbb{C})$	-	-	$\downarrow$
$(\alpha \stackrel{e}{=} \alpha) :: \mathbb{C}$	$\mathbb{C}$	$\alpha \notin (\text{FVC } \mathbb{C})$	$\downarrow$	-	$\downarrow$
$(\alpha \stackrel{e}{=} \beta) :: \mathbb{C}$	$\{\alpha \mapsto \beta\}\mathbb{C}$	$\alpha \neq \beta$	$\downarrow$	-	$\downarrow$
$(\alpha \stackrel{e}{=} \tau) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \notin (\text{FVC } \mathbb{C})$	$\downarrow$	$\downarrow$	$\downarrow$
$(\alpha \stackrel{e}{=} \tau) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \in (\text{FVC } \mathbb{C})$	$\downarrow$	$\uparrow$	$\downarrow$
$(\tau \stackrel{e}{=} \alpha) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \notin (\text{FVC } \mathbb{C})$	$\downarrow$	$\downarrow$	$\downarrow$
$(\tau \stackrel{e}{=} \alpha) :: \mathbb{C}$	$\{\alpha \mapsto \tau\}\mathbb{C}$	$\alpha \notin (\text{FTV } \tau) \wedge \alpha \in (\text{FVC } \mathbb{C})$	$\downarrow$	$\uparrow$	$\downarrow$
$(\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: \mathbb{C}$	$(\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: \mathbb{C}$	-	-	$\downarrow$	$\uparrow$

Table 1: Variation of termination measure components on the recursive call

particular. We also had to use the following lemma mentioned in the formalization of Sudoku puzzle by Laurent Théry [The06].

Lemma 1. [list subset membership and unique list length]
$\forall l1, l2 : listD, \mathsf{NoDup}\ l1 \Rightarrow \mathsf{NoDup}\ l2 \Rightarrow \mathsf{List.incl}\ l1\ l2 \Rightarrow \neg\mathsf{List.incl}\ l2\ l1 \Rightarrow (\mathsf{List.length}\ l1) < (\mathsf{List.length}\ l2)$

The lemma is essential for our termination proofs. Coq also provide a library to reason about lists modulo permutation. Together we were able to reason with the lists as finite sets.

4 MGU axioms

Note that each of the axioms, introduced earlier in Section 1, is characterizing the mgu behavior on a single constraint (a pair of terms). In our verification, we will lift these axioms to a constraint list. This is necessary since constraint-based type reconstruction algorithms solve all the constraints in one go. The new axioms are:

$$\begin{array}{rl}
(i) & \mathsf{unify}\,\mathbb{C} = \mathsf{Some}\,\sigma \Rightarrow \sigma \models \mathbb{C} \\
(ii) & (\mathsf{unify}\,\mathbb{C} = \mathsf{Some}\,\sigma \wedge \sigma' \models \mathbb{C}) \Rightarrow \exists \sigma''.\ \sigma' \approx \sigma \circ \sigma'' \\
(iii) & \mathsf{unify}\,\mathbb{C} = \mathsf{Some}\,\sigma \Rightarrow \mathsf{FTV}(\sigma) \subseteq \mathsf{FVC}\,(\mathbb{C}) \\
(iv) & \sigma \models \mathbb{C} \Rightarrow \exists \sigma'.\ \mathsf{unify}\,\mathbb{C} = \mathsf{Some}\,\sigma'
\end{array}$$

We can now go into the proof of the above axioms. The underlying theme in all of the proofs below is the use of functional induction tactic in Coq. The tactic ensures that we have the right induction hypothesis, when we want to prove a property for the inductive case. We mention this general technique next.

4.1 Functional Induction in Coq

In Coq, the functional induction technique generates uses the induction principle which is generated for the definitions defined using the Function keyword. The induction principle is shown in the Appendix 7.2. The induction principle is rather long because of the following two reasons. Firstly, as a result of rewriting step above the specification has become verbose and, secondly, because of the cases involved; there are 3 cases with 3 outcomes each.

In the next few sections, we mention only the important lemmas involved in the proofs of each of the axioms. For many of these lemmas, we give the main technique involved in the proofs.

4.2 Axiom i

Lemma 2. [satisfy and compose subst]
$\forall x.\ \forall \mathbb{C}.\ \forall \sigma.\ \forall \tau.\ \sigma \models \{x \mapsto \tau\}(\mathbb{C}) \Rightarrow (\{x \mapsto \tau\} \circ \sigma) \models \mathbb{C}$

Proof. By induction on $\mathbb{C}$. □

Lemma 3. **[membership in a constraint list is an invariant under substitution]**
$\forall x.\, \forall \mathbb{C}.\, \forall \tau, \tau_1, \tau_2.\, (\tau_1 \stackrel{e}{=} \tau_2) \in \mathbb{C} \Rightarrow \{x \mapsto \tau\}(\tau_1 \stackrel{e}{=} \tau_2) \in \{x \mapsto \tau\}(\mathbb{C})$

Proof. By induction on τ. ☐

Lemma 4. $\forall \mathbb{C}.\, \forall \sigma.\, \forall \tau, \tau'.\, (\tau \stackrel{e}{=} \tau') \in \mathbb{C} \wedge \mathsf{unify}\, \mathbb{C} = \mathsf{Some}\, \sigma \;\Rightarrow \sigma \models (\tau \stackrel{e}{=} \tau')$

Proof. By functional induction on $\mathsf{unify}\, \mathbb{C}$ and theorem 1. ☐

Theorem 2. $\forall \sigma.\, \forall \mathbb{C}.\, \mathsf{unify}\, \mathbb{C} = \mathsf{Some}\, \sigma \;\Rightarrow \sigma \models \mathbb{C}$

Proof. By functional induction on $\mathsf{unify}\, \mathbb{C}$ and the theorem 1 and lemma 3. ☐

4.3 Axiom ii

Lemma 5. **[Equal substitution instance for singleton subst]**
$\forall \sigma.\, \forall \alpha.\, \forall \tau, \tau'.\, \alpha \notin (\mathsf{FTV}\, \tau) \wedge \sigma(\alpha) = \sigma(\tau) \;\Rightarrow\; \sigma(\tau') = \sigma(\{\alpha \mapsto \tau\}(\tau'))$

Proof. By induction on τ'. ☐

Lemma 6. **[Constraint satisfaction extended to a substitution instance of a constraint]**
$\forall \mathbb{C}.\, \forall \sigma.\, \forall \alpha.\, \forall \tau.\, \sigma \models \mathbb{C} \wedge \alpha \notin (\mathsf{FTV}\, \tau) \wedge \sigma(\alpha) = \sigma(\tau) \;\Rightarrow\; \sigma \models \{\alpha \mapsto \tau\}(\mathbb{C})$

Proof. By induction on $\mathbb{C}$ and lemma 5. ☐

The following lemma lifts the extensional equality on type variables to any type.

Lemma 7. **[Squiggle extensionality extended to any type]**
$\forall \sigma, \sigma'.\, \forall \alpha.\, \sigma(\alpha) = \sigma'(\alpha) \;\Leftrightarrow\; \forall \tau.\, \sigma(\tau) = \sigma'(\tau)$

Proof. ($\Rightarrow$) By induction on τ.
($\Leftarrow$) Trivial. ☐

Theorem 3. $\forall \sigma.\, \forall \mathbb{C}.\, (\mathsf{unify}\, \mathbb{C} = \mathsf{Some}\, \sigma \;\wedge \sigma' \models \mathbb{C}) \Rightarrow \exists \sigma''.\, \sigma' \approx \sigma \circ \sigma''$

Proof. By functional induction on $\mathsf{unify}\, \mathbb{C}$ and the theorem 1, and the lemma 6, 7. ☐

4.4 Axiom iii

Lemma 8. **[Compose and domain membership]**
$\forall \alpha, \alpha'.\, \forall \tau. \forall \sigma.\, \alpha' \in \mathsf{dom\_subst}\ (\{\alpha \mapsto \tau\} \circ \sigma))$
$\Rightarrow \alpha' \in \mathsf{dom\_subst}\, \{\alpha \mapsto \tau\} \vee \alpha' \in \mathsf{dom\_subst}\, \sigma$

Lemma 9. **[Compose and range membership]**
$\forall \alpha, \alpha'.\, \forall \tau.\, \forall \sigma.\, (\alpha \notin (\mathsf{FTV}\, \tau) \wedge \alpha' \in \mathsf{range\_subst}\ (\{\alpha \mapsto \tau\} \circ \sigma))$
$\Rightarrow \alpha' \in \mathsf{range\_subst}\, \{\alpha \mapsto \tau\} \vee \alpha' \in \mathsf{range\_subst}\, \sigma$

Without going into details, the following lemma helps us in proving Lemma 9. Note that the definition of range_subst contains references to higher order functions M.map2 and this lemma helps in not having to reason about M.map2.

Lemma 10. **[Subst range abstraction]**
$\forall \alpha.\, \forall \sigma.\, \alpha \in \mathsf{range\_subst}\ (\sigma) \Leftrightarrow \exists \alpha'.\alpha' \in \mathsf{dom\_subst}\ (\sigma) \wedge \alpha \in \sigma(\alpha')$

Theorem 4. $\forall \sigma, \sigma'.\, \forall \mathbb{C}.\, \mathsf{unify}\, \mathbb{C} = \mathsf{Some}\, \sigma \;\Rightarrow \mathsf{FTV\_subst}(\sigma) \subseteq \mathsf{FVC}(\mathbb{C})$

Proof. By functional induction on $\mathsf{unify}\, \mathbb{C}$ and the lemmas 8 and 9. ☐

4.5 Axiom iv

This axiom requires the notion of subterms, which we define below:

$$\mathsf{subterms}\,\alpha = [\,]$$
$$\mathsf{subterms}\,(\tau_1 \to \tau_2) = \tau_1 :: \tau_2 :: (\mathsf{subterms}\,\tau_1) +\!+ (\mathsf{subterms}\,\tau_2)$$

Then we can define what it means to for a term to be contained in another term.

Lemma 11. [Containment]
$\forall \tau, \tau'.\ \tau \in (\mathsf{subterms}\ \tau') \ \Rightarrow\ \forall \tau''.\ \tau'' \in (\mathsf{subterms}\ \tau) \Rightarrow \tau'' \in (\mathsf{subterms}\ \tau')$

Proof. By induction on the τ'. □

A somewhat related lemma is used to show well foundedness of types.

Lemma 12. [Well founded types]
$\forall \tau.\ \neg\ \tau \in (\mathsf{subterms}\ \tau)$

Proof. By induction on the τ and by lemma 11. □

Lemma 13. [Member subterms unequal]
$\forall \tau, \tau'.\ \tau \in (\mathsf{subterms}\ \tau') \Rightarrow \tau \neq \tau'$

Proof. By case analysis on $\tau = \tau'$ and by lemma 12. □

The following obvious but powerful lemma helps in proving the axiom:

Lemma 14. [member subterms and apply subst]
$\forall \sigma.\ \forall \alpha.\ \forall \tau.\ \alpha \in (\mathsf{subterms}\ \tau) \Rightarrow \sigma(\alpha) \neq \sigma(\tau)$

Proof. By induction on τ and by lemma 13. □

Lemma 15. [Member arrow and subterms]
$\forall \sigma.\ \forall \alpha.\ \forall \tau_1, \tau_2.\ \mathsf{member}\ \alpha\ (\mathsf{FTV}\ \tau_1) = \mathit{true} \vee \mathsf{member}\ \alpha\ \ (\mathsf{FTV}\ \tau_2) = \mathit{true}$
$\Rightarrow \alpha \in \mathsf{subterms}(\tau_1 \to \tau_2)$

Proof. By induction on τ_1, followed by induction on τ_2. □

A corollary from the above two gives us the required lemma:

Corollary 1. [Member apply subst unequal]
$\forall \sigma.\ \forall \alpha.\ \forall \tau_1, \tau_2.\ \mathsf{member}\ \alpha\ (\mathsf{FTV}\ \tau_1) = \mathit{true} \vee \mathsf{member}\ \alpha\ \ (\mathsf{FTV}\ \tau_2) = \mathit{true}$
$\Rightarrow \sigma(\alpha) \neq \sigma(\tau_1 \to \tau_2)$

Proof. By lemma 14 and 15. □

Theorem 5. $\forall \sigma.\ \forall \mathbb{C}.\ \sigma \models \mathbb{C} \ \Rightarrow \exists \sigma'.\ \mathsf{unify}\ \mathbb{C} = \mathsf{Some}\ \sigma'$

Proof. By functional induction on $\mathsf{unify}\ \mathbb{C}$ and the Lemma 6 and Corollary 1. □

5 Related Work and Conclusions

5.1 Related Work

There are formalizations of the unification algorithm in a number of different theorem provers [Bla08, Pau85, Rou94]. Unification is fundamentally used in type inference. Many of the existing verifications of type inference algorithms [DM99, NN99, NN96, UN09] axiomatize the behavior of the MGU rather than provide an implementation as we do here.

We comment on the implementation in the CoLoR library [BDCG+06]. CoLoR is an extensive and very successful library supporting reasoning about termination and rewriting. This Coq implementation of the unification algorithm was recently released [Bla08]. Our implementation differs from theirs in a number of ways. Perhaps the most significant difference is that we represent substitutions as finite maps whereas in the implementation in CoLoR they are represented by functions from variables to a generalized term structure. The axioms verified here are not explicitly verified in CoLoR, however the library there could serve as a basis for doing so. We believe that the lemmas supporting our verification could be translated into their more general framework but that the proofs would be significantly different because we use function induction which follows the structure of our algorithm. The algorithm in CoLoR is a specified in a significantly different style, as an iterated step function.

5.2 Future Work

The current work serves as a first step in verification of various constraint-based type reconstruction algorithms. The entire formalization is done in Coq 8.1.pl3 version in about 4400 lines of specifications and tactics, and is available online at `http://www.cs.uwyo.edu/~skothari`. The choice of representing substitutions as finite functions was crucial, but it still leaves some question unanswered. An induction principle for finite maps would have been useful for some of the proofs and indeed there is a new version of the library in Coq 8.2 which provides this. We believe that this entire work should lead to a better understanding and appreciation of the finite maps library in Coq. These proofs are part of a larger effort to verify our extended version of Wand's algorithm which handles the polymorphic let construct [Kot07, KC08].

6 Acknowledgments

We would like to thank Santiago Zanella (INRIA - Sophia Antipolis) for showing us how to encode lexicographic ordering for 3-tuples in Coq. Thanks also to Frederic Blanqui for answering our queries regarding the new release of CoLoR library. We are also thankful to Laurent Théry for making his Coq formulation of Sudoku available on the web, to Stéphane Lescuyer and other Coq-club members for answering our queries on the Coq-club mailing list.

References

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[BDCG+06] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *8th International Workshop on Termination (WST '06)*, pages 69–73, 2006.

[Bla08] Frederic Blanqui. CoLor, a Coq library on rewriting and termination., January 2008. `http://color.inria.fr/doc/CoLoR.Term.WithArity.AUnif.html`.

[Bov01] Ana Bove. Simple General Recursion in Type Theory. *Nordic J. of Computing*, 8(1):22–42, 2001.

[BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[Cdt07] The Coq development team. *The Coq proof assistant reference manual.* INRIA, LogiCal Project, 2007. Version 8.1.3.

[CH88] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[CS95] Graham Collins and Don Syme. A Theory of Finite Maps. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 122–137. Springer-Verlag, 1995.

[DM99] C. Dubois and V. M. Morain. Certification of a Type Inference Tool for ML: Damas–Milner within Coq. *J. Autom. Reason.*, 23(3):319–346, 1999.

[KC08] Sunil Kothari and James Caldwell. On Extending Wand's Type Reconstruction Algorithm to Handle Polymorphic Let. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, pages 254–263. University of Athens, 2008.

[Kot07] Sunil Kothari. Wand's Algorithm Extended For The Polymorphic ML-Let. Technical report, University of Wyoming, 2007.

[McB03] Conor McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003.

[MW85] Zohar Manna and Richard Waldinger. *The logical basis for computer programming. Volume 1: deductive reasoning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[NN96] Dieter Nazareth and Tobias Nipkow. *Theorem Proving in Higher Order Logics*, volume 1125, chapter Formal Verification of Alg. W: The Monomorphic Case, pages 331–345. Springer Berlin / Heidelberg, 1996.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, 23(3):299–318, 1999.

[Pau85] L. C. Paulson. Verifying the Unification Algorithm in LCF. *Sci. of Computer Programming*, 5:143–169, 1985.

[Rou94] J. Rouyer. *Developpement d'Algorithmes dans le Calcul des Constructions.* PhD thesis, Institut National Polytechnique de Lorraine, Nancy, France, 1994.

[The06] Laurent Thery. Sudoku in Coq. 2006.

[UN09] Christian Urban and Tobias Nipkow. *From Semantics to Computer Science*, chapter Nominal verification of algorithm W. Cambridge University Press, Not yet published 2009.

7 Appendix

7.1 First-order unification specification in Coq

```
Function unify (c:list constr){wf meaPairMLt} :(option (M.t type)) :=
match c with
   nil => Some (M.empty type)
| h::t => (match h with
              EqCons (TyVar x) (TyVar y)  =>
                  if eq_dec_stamp x y
                  then unify t
                  else (match unify (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t) with
                           Some p => Some (compose_subst (M.add x (TyVar y)(M.empty type)) p)
                         | None => None
                       end)
             | EqCons (TyVar x) (Arrow ty3 ty4) =>
                  if (member x (FTV ty3)) || (member x (FTV ty4))
                  then None
                  else (match (unify (apply_subst_to_constr_list
                                        (M.add x (Arrow ty3 ty4) (M.empty type))
                                         t) with
                          Some p => Some (compose_subst (M.add x (Arrow ty3 ty4) (M.empty type)) p)
                        | None => None
                       end)
            | EqCons (Arrow ty3 ty4)(TyVar x) =>
                  if (member x (FTV ty3)) || (member x  (FTV ty4))
                  then None
                  else (match (unify (apply_subst_to_constr_list
                                         (M.add x (Arrow ty3 ty4) (M.empty type))
                                          t)) with
                           Some p => Some (compose_subst (M.add x (Arrow ty3 ty4) (M.empty type)) p)
                         | None => None
                  end )
           | EqCons (Arrow ty3 ty4)(Arrow ty5 ty6)=> unify ((EqCons ty3 ty5)::((EqCons ty4 ty6)::t))
          end)
end.
```

7.2 Induction principle used in the functional induction

```
unify_ind
     : forall P : list constr -> option (M.t type) -> Prop,
       (forall c : list constr, c = nil -> P nil (Some (M.empty type))) ->
       (forall (c : list constr) (h : constr) (t : list constr),
        c = h :: t ->
        forall x y : nat,
        h = EqCons (TyVar x) (TyVar y) ->
        forall _x : x = y,
        eq_dec_stamp x y = left (x <> y) _x ->
        P t (unify t) -> P (EqCons (TyVar x) (TyVar y) :: t) (unify t)) ->
       (forall (c : list constr) (h : constr) (t0 : list constr),
        c = h :: t0 ->
        forall x y : nat,
        h = EqCons (TyVar x) (TyVar y) ->
        forall _x : x <> y,
        eq_dec_stamp x y = right (x = y) _x ->
        P (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0)
          (unify
             (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type))
                t0)) ->
        forall p : M.t type,
```

```
 unify
   (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0) =
 Some p ->
 P (EqCons (TyVar x) (TyVar y) :: t0)
   (Some (compose_subst (M.add x (TyVar y) (M.empty type)) p))) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall x y : nat,
 h = EqCons (TyVar x) (TyVar y) ->
 forall _x : x <> y,
 eq_dec_stamp x y = right (x = y) _x ->
 P (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0)
   (unify
      (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type))
         t0)) ->
 unify
   (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0) =
 None -> P (EqCons (TyVar x) (TyVar y) :: t0) None) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall (x : nat) (ty3 ty4 : type),
 h = EqCons (TyVar x) (Arrow ty3 ty4) ->
 member x (FTV ty3) || member x (FTV ty4) = true ->
 P (EqCons (TyVar x) (Arrow ty3 ty4) :: t0) None) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall (x : nat) (ty3 ty4 : type),
 h = EqCons (TyVar x) (Arrow ty3 ty4) ->
 member x (FTV ty3) || member x (FTV ty4) = false ->
 P
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
   (unify
      (apply_subst_to_constr_list
         (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
 forall p : M.t type,
 unify
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = Some p ->
 P (EqCons (TyVar x) (Arrow ty3 ty4) :: t0)
   (Some (compose_subst (M.add x (Arrow ty3 ty4) (M.empty type)) p))) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall (x : nat) (ty3 ty4 : type),
 h = EqCons (TyVar x) (Arrow ty3 ty4) ->
 member x (FTV ty3) || member x (FTV ty4) = false ->
 P
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
   (unify
      (apply_subst_to_constr_list
         (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
 unify
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = None ->
 P (EqCons (TyVar x) (Arrow ty3 ty4) :: t0) None) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall (ty3 ty4 : type) (x : nat),
 h = EqCons (Arrow ty3 ty4) (TyVar x) ->
 member x (FTV ty3) || member x (FTV ty4) = true ->
```

```
 P (EqCons (Arrow ty3 ty4) (TyVar x) :: t0) None) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall (ty3 ty4 : type) (x : nat),
 h = EqCons (Arrow ty3 ty4) (TyVar x) ->
 member x (FTV ty3) || member x (FTV ty4) = false ->
 P
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
   (unify
      (apply_subst_to_constr_list
         (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
 forall p : M.t type,
 unify
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = Some p ->
 P (EqCons (Arrow ty3 ty4) (TyVar x) :: t0)
   (Some (compose_subst (M.add x (Arrow ty3 ty4) (M.empty type)) p))) ->
(forall (c : list constr) (h : constr) (t0 : list constr),
 c = h :: t0 ->
 forall (ty3 ty4 : type) (x : nat),
 h = EqCons (Arrow ty3 ty4) (TyVar x) ->
 member x (FTV ty3) || member x (FTV ty4) = false ->
 P
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0)
   (unify
      (apply_subst_to_constr_list
         (M.add x (Arrow ty3 ty4) (M.empty type)) t0)) ->
 unify
   (apply_subst_to_constr_list
      (M.add x (Arrow ty3 ty4) (M.empty type)) t0) = None ->
 P (EqCons (Arrow ty3 ty4) (TyVar x) :: t0) None) ->
(forall (c : list constr) (h : constr) (t : list constr),
 c = h :: t ->
 forall ty3 ty4 ty5 ty6 : type,
 h = EqCons (Arrow ty3 ty4) (Arrow ty5 ty6) ->
 P (EqCons ty3 ty5 :: EqCons ty4 ty6 :: t)
   (unify (EqCons ty3 ty5 :: EqCons ty4 ty6 :: t)) ->
 P (EqCons (Arrow ty3 ty4) (Arrow ty5 ty6) :: t)
   (unify (EqCons ty3 ty5 :: EqCons ty4 ty6 :: t))) ->
forall c : list constr, P c (unify c)
```

7.3 Proof Obligations

There are 5 proof obligations related to the 5 recursive call sites in the specification. The sixth proof obligation is to show that the ordering relation is well founded.

```
forall (c : list constr) (h : constr) (t : list constr),
c = h :: t ->
forall t0 t1 : type,
h = EqCons t0 t1 ->
forall x : nat,
t0 = TyVar x ->
forall y : nat,
t1 = TyVar y ->
forall anonymous : x = y,
eq_dec_stamp x y = left (x <> y) anonymous ->
meaPairMLt t (EqCons (TyVar x) (TyVar y) :: t)
```

```
______________________________________(2/6)
forall (c : list constr) (h : constr) (t : list constr),
c = h :: t ->
forall t0 t1 : type,
h = EqCons t0 t1 ->
forall x : nat,
t0 = TyVar x ->
forall y : nat,
t1 = TyVar y ->
forall anonymous : x <> y,
eq_dec_stamp x y = right (x = y) anonymous ->
meaPairMLt (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t)
  (EqCons (TyVar x) (TyVar y) :: t)

______________________________________(3/6)
forall (c : list constr) (h : constr) (t : list constr),
c = h :: t ->
forall t0 t1 : type,
h = EqCons t0 t1 ->
forall x : nat,
t0 = TyVar x ->
forall ty3 ty4 : type,
t1 = Arrow ty3 ty4 ->
member x (FTV ty3) || member x (FTV ty4) = false ->
meaPairMLt
  (apply_subst_to_constr_list (M.add x (Arrow ty3 ty4) (M.empty type)) t)
  (EqCons (TyVar x) (Arrow ty3 ty4) :: t)

______________________________________(4/6)
forall (c : list constr) (h : constr) (t : list constr),
c = h :: t ->
forall t0 t1 : type,
h = EqCons t0 t1 ->
forall ty3 ty4 : type,
t0 = Arrow ty3 ty4 ->
forall x : nat,
t1 = TyVar x ->
member x (FTV ty3) || member x (FTV ty4) = false ->
meaPairMLt
  (apply_subst_to_constr_list (M.add x (Arrow ty3 ty4) (M.empty type)) t)
  (EqCons (Arrow ty3 ty4) (TyVar x) :: t)

______________________________________(5/6)
forall (c : list constr) (h : constr) (t : list constr),
c = h :: t ->
forall t0 t1 : type,
h = EqCons t0 t1 ->
forall ty3 ty4 : type,
t0 = Arrow ty3 ty4 ->
forall ty5 ty6 : type,
t1 = Arrow ty5 ty6 ->
meaPairMLt (EqCons ty3 ty5 :: EqCons ty4 ty6 :: t)
  (EqCons (Arrow ty3 ty4) (Arrow ty5 ty6) :: t)

______________________________________(6/6)
well_founded meaPairMLt
```