A Machine Checked Model of Idempotent MGU Axioms For Lists of Equational Constraints

Sunil Kothari, James Caldwell
Department of Computer Science,
University of Wyoming, USA

We present formalized proofs verifying that the first-order unification algorithm defined over lists of satisfiable constraints generates a most general unifier (MGU), which also happens to be idempotent. All of our proofs have been formalized in the Coq theorem prover. Our proofs show that finite maps produced by the unification algorithm provide a model of the axioms characterizing idempotent MGUs of lists of constraints. The axioms that serve as the basis for our verification are derived from a standard set by extending them to lists of constraints. For us, constraints are equalities between terms in the language of simple types. Substitutions are formally modeled as finite maps using the Coq library *Coq.FSets.FMapInterface*. Coq's method of functional induction is the main proof technique used in proving many of the axioms.

1 Introduction

We present formalized proofs verifying that the first-order unification algorithm defined over lists of satisfiable constraints generates a most general unifier (MGU), which also happens to be idempotent. All of our proofs have been formalized in the Coq theorem prover [6]. Our proofs show that substitutions produced by the unification algorithm provide a model of the axioms characterizing the idempotent MGUs of lists of constraints.

The formalization and verification presented here was motivated by our work on to verifying Wand's constraint based type inference algorithm [26] (and to verify our extension of Wand's algorithm to include polymorphic let [15]). In the recent literature on machine certified proof of correctness of type inference algorithms [13, 20, 25], most general unifiers are characterized by four axioms.

Recall that τ and τ' (in some language) are *unifiable* if there exists a substitution ρ mapping variables to terms in the language such that $\rho(\tau) = \rho(\tau')$. In such a case, ρ is called a *unifier*. A unifier ρ is a *most general unifier* if for any other unifier ρ'' there is a substitution ρ' such that $\rho \circ \rho' = \rho''$.

We consider the MGU axioms given by Nipkow and Urban [25]. Let ρ, ρ', ρ'' denote substitutions *i.e.* functions mapping type variables to terms, constraints are of the form $\tau \stackrel{e}{=} \tau'$ where τ and τ' are simple types and the symbol FTV is overloaded to denote the free type variables of substitutions, constraints and types and the notation. Composition of substitutions[1] is denoted $\rho \circ \rho'$. With these notational conventions in mind, the MGU axioms are presented as follows:

$$
\begin{array}{ll}
(i) & \mathit{mgu}\ \rho\ (\tau_1 \stackrel{e}{=} \tau_2) \Rightarrow \rho(\tau_1) = \rho(\tau_2) \\
(ii) & \mathit{mgu}\ \rho\ (\tau_1 \stackrel{e}{=} \tau_2)\ \wedge\ \rho'(\tau_1) = \rho'(\tau_2) \Rightarrow \exists \rho''.\rho' = \rho \circ \rho'' \\
(iii) & \mathit{mgu}\ \rho\ (\tau_1 \stackrel{e}{=} \tau_2) \Rightarrow \mathsf{FTV}(\rho) \subseteq \mathsf{FTV}(\tau_1 \stackrel{e}{=} \tau_2) \\
(iv) & \rho(\tau_1) = \rho(\tau_2) \Rightarrow \exists \rho'.\ \mathit{mgu}\ \rho'\ (\tau_1 \stackrel{e}{=} \tau_2)
\end{array}
$$

[1] The reader should note that in this paper, composition of functions is characterized by the equation $(\rho \circ \rho')(x) = \rho'(\rho(x))$.

Maribel Fernandez (Ed.): 24th International Workshop on
Unification (UNIF2010).
EPTCS 42, 2010, pp. 24–38, doi:10.4204/EPTCS.42.3

These axioms, modeling MGUs, have proved useful in verifying substitution-based type inference algorithms where the constraints are solved as they are generated, one at a time. In constraint-based type inference algorithms like Wand's, the constraints are generated before they are solved. Thus, for use in the constraint based setting, we lift the MGU axioms to lists of constraints. To do so, we restate the standard axioms to apply to constraint lists, add two new axioms which characterize MGUs of lists of constraints; one axiom for the empty list and another for lists constructed by appends. Also, reasoning about Wand's type inference algorithm requires the MGUs be idempotent, so we add another axiom for idempotency. Idempotent MGUs have the nice property that their domain and range elements are disjoint.

We proceed by characterizing idempotent MGUs for lists of equational constraints by presenting seven axioms. Then we show that the first order unification algorithm models those axioms. The theorems and supporting lemmas mentioned in this paper have been formalized and verified in Coq [23] - a theorem prover based on calculus of inductive constructions [12]. In the formalization, we represent substitutions using Coq's finite map library [1].

To start, we generalize the standard MGU axioms to constraint lists. In addition to the notations introduced above, if C is a list of constraints, $\rho \models C$ (read ρ *satisfies* C) means that ρ unifies all constrains in C. Let C denote a constraint list, then the MGU axioms (for a list of constraints) are:

$$
\begin{array}{ll}
(i) & \mathit{mgu}\ \rho\ C\ \Rightarrow \rho \models C \\
(ii) & \mathit{mgu}\ \rho\ C\ \wedge\ \rho' \models C \Rightarrow \exists \rho''.\ \rho' = \rho \circ \rho'' \\
(iii) & \mathit{mgu}\ \rho\ C\ \Rightarrow \mathsf{FTV}\ (\rho) \subseteq \mathsf{FTV}\ (C) \\
(iv) & \rho \models\ C\ \Rightarrow \exists \rho'.\ \mathit{mgu}\ \rho' C
\end{array}
$$

To the axioms just mentioned we add three more axioms that characterize idempotent MGUs for a list of equational constraints. List append is denoted by $++$.

$$
\begin{array}{ll}
(v) & \mathit{mgu}\ \rho\ C\ \Rightarrow\ \rho \circ \rho = \rho \\
(vi) & \mathit{mgu}\ \rho\ [\,] \Rightarrow \rho = \mathit{Id} \\
(vii) & \mathit{mgu}\ \rho'\ C'\ \wedge\ \mathit{mgu}\ \rho''\ (\rho'(C'')) \wedge \mathit{mgu}\ \rho\ (C' +\!\!+ C'') \Rightarrow \rho = \rho' \circ \rho''
\end{array}
$$

These additional axioms are mentioned elsewhere in the unification literature, namely [14, 17]. The statement of axiom *vii* is convenient in proofs where constraint lists are constructed by combining lists of constraints rather than adding them one at a time. A lemma characterizing lists constructed by conses is easily proved from this axiom.

Formalizing substitutions as finite maps in Coq, we show that first-order unification (unify) is a model of the MGU axioms. To distinguish the formal representation of substitutions as finite maps from mathematical functions, we denote finite maps by σ, σ', σ_1, etc. Mathematical functions enjoy extensional equality while finite maps do not (more about this later). We write $\rho \approx \rho'$ to denote extensional equality for finite maps; *i.e.* that under application they agree pointwise on all inputs. With these considerations in mind, we have proved the following in Coq:

$$
\begin{array}{ll}
(i) & \mathsf{unify}(C) = \sigma \;\Rightarrow \sigma \models C \\
(ii) & (\mathsf{unify}(C) = \sigma \;\wedge \sigma' \models C) \Rightarrow \exists \sigma''.\; \sigma' \approx \sigma \circ \sigma'' \\
(iii) & \mathsf{unify}(C) = \sigma \;\Rightarrow \mathsf{FTV}(\sigma) \subseteq \mathsf{FTV}(C) \\
(iv) & \sigma \models C \;\Rightarrow \exists \sigma'.\; \mathsf{unify}(C) = \sigma' \\
(v) & \mathsf{unify}(C) = \sigma \;\Rightarrow \sigma \circ \sigma \approx \sigma \\
(vi) & \mathsf{unify}([\,]) = \sigma \;\Rightarrow \sigma = \sigma_{\mathbb{E}} \\
(vii) & (\mathsf{unify}(C') = \sigma' \;\wedge \mathsf{unify}(\sigma'(C'')) = \sigma'' \wedge \mathsf{unify}(C' +\!\!+ C'') = \sigma) \Rightarrow \;\sigma \approx \sigma' \circ \sigma''
\end{array}
$$

The rest of this paper is organized as follows: Section 2 introduces a number of formal definitions and terminologies needed for this paper. It also includes more discussion about substitutions represented as finite functions. Section 3 describes the formalization of a first-order unification algorithm and the termination argument. Section 4 describes the functional induction tactic and the theorems and lemmas proved in the verification that unify models the idempotent MGU axioms. Finally, Section 5 mentions related work and also summarizes our current work.

2 Types and Substitutions

Unification is implemented here over a language of simple types given by the following grammar:

$$\tau ::= \quad \alpha \mid \quad \tau_1 \to \tau_2$$

where α is a type variable, and $\tau_1, \tau_2 \in \tau$ are type terms.

Thus, a type is either a type variable or a function type. We define the list of *free*[2] *variables of a type* (FTV) as:

$$
\begin{array}{lcl}
\mathsf{FTV}(\alpha) & \stackrel{def}{=} & [\alpha] \\
\mathsf{FTV}(\tau \to \tau') & \stackrel{def}{=} & \mathsf{FTV}(\tau) +\!\!+ \mathsf{FTV}(\tau')
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathsf{FTV}([\,]) & \stackrel{def}{=} & [\,] \\
\mathsf{FTV}((\tau_1 \stackrel{e}{=} \tau_2) :: C) & \stackrel{def}{=} & \mathsf{FTV}(\tau_1) +\!\!+ \mathsf{FTV}(\tau_2) +\!\!+ \mathsf{FTV}(C)
\end{array}
$$

Substitutions are formally represented as finite maps where the domain of the map is the collection of type variables and the codomain is the simple types. Application of a finite map to a type is defined as:

$$
\begin{array}{lcl}
\sigma(\alpha) & \stackrel{def}{=} & \begin{cases} \tau & \textit{if } \langle \alpha, \tau \rangle \in \sigma \\ \alpha & \textit{otherwise} \end{cases} \\
\sigma(\tau_1 \to \tau_2) & \stackrel{def}{=} & \sigma(\tau_1) \to \sigma(\tau_2)
\end{array}
$$

Application of a finite map to a constraint is defined similarly as:

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

Since Coq's finite maps are not extensional, we define extensionality ($\approx$) as a relation on finite maps as follows:

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

Moreover, the equality can be extended to all types as given by the following lemma:

Lemma 1. $\forall \alpha.\, \sigma(\alpha) = \sigma'(\alpha) \Leftrightarrow \forall \tau.\, \sigma(\tau) = \sigma'(\tau)$

[2]Strictly speaking, since we have no binding operators in the language of simple types the modifier "free" is unnecessary, we include it here anticipating a more complex language of types in future developments.

2.1 Implementing Substitutions as Finite Maps

The representation of substitutions and the libraries available to a user plays a very important role in the formalization. In the verification literature, substitutions have been represented as functions [25], as lists of pairs [13], and as sets of pairs [21]. We represent substitutions as finite functions (a.k.a finite maps in Coq). We use the Coq finite map library *Coq.FSets.FMapInterface* [1], which provides an axiomatic presentation of finite maps and a number of supporting implementations. However, it does not provide an induction principle for finite maps, and forward reasoning is often needed to use the library. We found we did not need induction to reason on finite maps, though there are natural induction principles we might have proved [11, 18]. The fact that the library does not provide for extensional equality of finite maps means that, for example, the following simple lemma does not hold:

Lemma 2. $\sigma_{\mathbb{E}} \circ \sigma_{\mathbb{E}} = \sigma_{\mathbb{E}}$

But the following is easily proved:

Lemma 3. $\forall \tau.(\sigma_{\mathbb{E}} \circ \sigma_{\mathbb{E}})(\tau) = \sigma_{\mathbb{E}}(\tau)$

To give a feel of the Coq's finite map library, we define free type variables of a substitution, and the substitution composition operator using the finite map library functions. In the definitions below, 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 (*M*) which maps a function over the range elements of a finite map, whereas *List.map* is a function from the list library.
First, we define the list of free type variables of a substitution:

$$\mathsf{FTV}(\sigma) \stackrel{def}{=} \mathsf{dom\_subst}(\sigma) +\!\!+ \mathsf{range\_subst}(\sigma)$$

To consider the *domain* and *range* elements of a finite function (and this is the key feature of the function being finite), we use the finite map library function M.elements. M.elements(σ) returns a list of pairs (key-value pairs) corresponding to the finite map σ. The domain and range elements of a substitution are defined as:

$$\mathsf{dom\_subst}(\sigma) \stackrel{def}{=} \mathsf{List.map}\ (\lambda t.\mathsf{fst}\ (t))\ (\mathsf{M.elements}(\sigma))$$
$$\mathsf{range\_subst}(\sigma) \stackrel{def}{=} \mathsf{List.flat\_map}\ (\lambda t.\mathsf{FTV}\ (\mathsf{snd}\ (t)))\ (\mathsf{M.elements}(\sigma))$$

The function List.flat_map is also known as mapcan in LISP and concatMap in Haskell. Next, we define a few utility functions to help us define the composition operator $\circ$. Applying a substitution σ' to a substitution σ means applying σ' to the range elements of σ.

$$\sigma'(\sigma) \stackrel{def}{=} \mathsf{M.map}\ (\lambda \tau.\sigma'(\tau))\ \sigma$$

The function subst_diff is used to define composition of finite maps, and is defined as:

$$\mathsf{subst\_diff}\ \sigma\ \sigma' \stackrel{def}{=} \mathsf{M.map2\ choose\_subst}\ \sigma\ \sigma'$$

In this definition, M.map2 is defined in Coq library as the function that takes two maps σ and σ', and creates a map whose binding belongs to either σ or σ' based on the function choose_subst, which determines the presence and value for a key (absence of a value is denoted by None). The values in the first map are preferred over the values in the second map for a particular key. The function choose_subst is defined as:

$$\mathsf{choose\_subst}\ (\mathsf{Some}\,\tau_1)\ (\mathsf{Some}\,\tau_2) \stackrel{def}{=} \mathsf{Some}\ \tau_1$$
$$\mathsf{choose\_subst}\ (\mathsf{Some}\,\tau_1)\ \mathsf{None} \stackrel{def}{=} \mathsf{Some}\ \tau_1$$
$$\mathsf{choose\_subst\ None}\ (\mathsf{Some}\,\tau_2) \stackrel{def}{=} \mathsf{Some}\,\tau_2$$
$$\mathsf{choose\_subst\ None\ None} \stackrel{def}{=} \mathsf{None}$$

Finally, the composition of finite maps ($\circ$) is defined as:

$$\sigma \circ \sigma' \stackrel{def}{=} \mathsf{subst\_diff}\ \sigma'(\sigma)\ \sigma'$$

Substitution composition application to a type has the following property:

Theorem 1. $\forall\sigma.\,\forall\sigma'.\,\forall\tau.(\sigma\circ\sigma')(\tau) = \sigma'(\sigma(\tau))$

Proof. By induction on the type τ followed by case analysis on the binding's occurrence in the composed substitution and in the individual substitutions. □

Interestingly, the base case (when τ is a type variable) is more difficult than the inductive case (when τ is a compound type). Incidentally, the same theorem has been formalized in Coq [13], where substitutions are represented as lists of pairs, but the proof there required 600 proof steps. We proved Theorem 1 in about 100 proof steps.

3 First-Order Unification

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

$$
\begin{array}{lcl}
\mathsf{unify}\,[\,] & \stackrel{def}{=} & \mathit{Id} \\
\mathsf{unify}\,((\alpha \stackrel{\mathrm{e}}{=} \beta) :: C) & \stackrel{def}{=} & \mathsf{if}\ \alpha = \beta\ \mathsf{then}\ \mathsf{unify}(C)\ \mathsf{else}\ \{\alpha \mapsto\ \beta\} \circ \mathsf{unify}\,(\{\alpha \mapsto \beta\}(C)) \\
\mathsf{unify}\,((\alpha \stackrel{\mathrm{e}}{=} \tau) :: C) & \stackrel{def}{=} & \mathsf{if}\ \alpha\ \mathsf{occurs\ in}\ \tau\ \mathsf{then\ Fail\ else}\ \{\alpha \mapsto\ \tau\} \circ \mathsf{unify}\,(\{\alpha \mapsto \tau\}(C)) \\
\mathsf{unify}((\tau \stackrel{\mathrm{e}}{=} \alpha) :: C) & \stackrel{def}{=} & \mathsf{if}\ \alpha\ \mathsf{occurs\ in}\ \tau\ \mathsf{then\ Fail\ else}\ \{\alpha \mapsto\ \tau\} \circ \mathsf{unify}\,(\{\alpha \mapsto \tau\}(C)) \\
\mathsf{unify}\,((\tau_1 \to \tau_2 \stackrel{\mathrm{e}}{=} \tau_3 \to \tau_4) :: C) & \stackrel{def}{=} & \mathsf{unify}((\tau_1 \stackrel{\mathrm{e}}{=} \tau_3) :: (\tau_2 \stackrel{\mathrm{e}}{=} \tau_4) :: C)
\end{array}
$$

This specification is written in a functional style. It would also have been possible to formalize unify in a relational style. A discussion of the trade-offs between these two styles of formalization Coq can be found in [5]. Since Coq's type theory requires functions to be total, the functional style carries an overhead; we need a value to represent failure. We used Coq's `option` type to make first-order unification total. The *option* type (*maybe* in Haskell) is defined in Coq as follows:

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

The constructor None indicates failure and the term $\mathsf{Some}(\sigma)$ indicates success (with σ as the result). In the presentation here, we omit the None and Some constructors. In virtually all theorems proved here, the None case is trivial.

The presentation of the unification algorithm given here is general recursive, *i.e.*, the recursive call is not necessarily on a structurally smaller argument. Various papers have discussed the non-structural recursion used in the standard first-order unification algorithm. McBride has given a structurally recursive unification algorithm [19]. Bove [9] gives an algorithm similar to ours and proves termination in Alf [2]. We believe our presentation of the algorithm is more perspicuous than Bove's although a similar termination argument works here. 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. We use the standard lexicographic ordering on the triple: $<|C_{FVC}|, |C_{\to}|, |C|>$, where

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

Our triple is similar to the triple proposed by others [9, 4, 3], but a little simpler.

Table 1 shows how these components vary depending on the constraint at the head of the constraint list. The table closely follows the reasoning we used to satisfy the proof obligations generated by the above specification [16]. We use -, ↑, ↓ to denote whether the component is unchanged, increased or

Original call	Recursive call	Conditions, if any	$\lvert C_{FVC}\rvert$	$\lvert C_{\rightarrow}\rvert$	$\lvert C\rvert$
$(\alpha \stackrel{e}{=} \alpha) :: C$	C	$\alpha \in \mathsf{FVC}(C)$	-	-	$\downarrow$
$(\alpha \stackrel{e}{=} \alpha) :: C$	C	$\alpha \notin \mathsf{FVC}(C)$	$\downarrow$	-	$\downarrow$
$(\alpha \stackrel{e}{=} \beta) :: C$	$\{\alpha \mapsto \beta\}(C)$	$\alpha \neq \beta$	$\downarrow$	-	$\downarrow$
$(\alpha \stackrel{e}{=} \tau) :: C$	$\{\alpha \mapsto \tau\}(C)$	$\alpha \notin \mathsf{FTV}(\tau) \wedge \alpha \notin \mathsf{FVC}(C)$	$\downarrow$	$\downarrow$	$\downarrow$
$(\alpha \stackrel{e}{=} \tau) :: C$	$\{\alpha \mapsto \tau\}(C)$	$\alpha \notin (\mathsf{FTV}(\tau) \wedge \alpha \in \mathsf{FVC}(C)$	$\downarrow$	$\uparrow$	$\downarrow$
$(\tau \stackrel{e}{=} \alpha) :: C$	$\{\alpha \mapsto \tau\}(C)$	$\alpha \notin \mathsf{FTV}(\tau) \wedge \alpha \notin \mathsf{FVC}(C)$	$\downarrow$	$\downarrow$	$\downarrow$
$(\tau \stackrel{e}{=} \alpha) :: C$	$\{\alpha \mapsto \tau\}(C)$	$\alpha \notin \mathsf{FTV}(\tau) \wedge \alpha \in \mathsf{FVC}(C)$	$\downarrow$	$\uparrow$	$\downarrow$
$((\tau_1 \rightarrow \tau_2) \stackrel{e}{=} (\tau_3 \rightarrow \tau_4)) :: C$	$((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C)$	None	-	$\downarrow$	$\uparrow$

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

decreased, respectively. We might have used finite sets here (for counting the unique free variables of a constraint list), but we used lists because of our familiarity with the list library. We found the existing Coq list library offers excellent support for reasoning about lists in general, and unique lists in particular. Coq also provides a library to reason about sets as lists modulo permutation.

We found the following lemma mentioned in the formalization of Sudoku puzzles by Laurent Théry [24] very useful in our termination proofs.

Lemma 4.

$$\forall l, l' : \mathit{list}\ D,\ \mathsf{NoDup}\ l \Rightarrow \mathsf{NoDup}\ l' \Rightarrow \mathsf{List.incl}\ l\ l' \Rightarrow \neg\mathsf{List.incl}\ l'\ l \Rightarrow (\mathsf{List.length}\ l) < (\mathsf{List.length}\ l')$$

This lemma nicely relates list inclusion to length.

4 Verification of the Model

Now we present the proofs of the theorems verifying our model of the idempotent MGU axioms. The underlying theme in almost all of the proofs presented below is the use of the functional induction tactic [5] in Coq. This tactic is available to us because we have specified first-order unification in a functional style rather than the relational style. The functional induction technique generates an induction principle for definitions defined using the Function keyword. Given a general recursive algorithm known to terminate (termination requires a separate proof), the induction principle generated for that particular algorithm allows a symbolic unfolding of the computation with induction hypotheses for all recursive calls. This technique is featured in other theorem provers and was pioneered in Nqthm by Boyer and Moore [10].

Functional induction is obviously stronger than the normal list induction, it closely follows the syntax of the definition and tends to generate induction hypotheses of exactly the right form needed. The actual induction principle is available in [16]. The induction principle for the unification algorithm itself is rather long because of the number of cases involved; there are five cases - three of which have three sub-cases each.

In the next few sections, we present the formal statements of the most important lemmas involved in the proofs of each of the axioms. For many of these lemmas, we describe the main technique involved in the proofs. Due to limitations on space, lemmas stated without comment on their proofs should be assumed to follow by structural induction on a constraint list or type.

4.1 Axiom i

Lemma 5. $\forall\alpha.\,\forall C.\,\forall\sigma.\,\forall\tau.\,\sigma \models \{\alpha \mapsto \tau\}(C) \Rightarrow (\{\alpha \mapsto \tau\} \circ \sigma) \models C$

Theorem 2. $\forall C.\,\forall\sigma.\,\mathsf{unify}(C) = \sigma \Rightarrow \sigma \models C$

Proof. Choose an arbitrary C. By functional induction on unify C, there are two main cases:

Case $C = [\,]$. Follows trivially since any substitution satisfies an empty constraint list.

Case $C \neq [\,]$. We consider the various cases based on the constraint at the head of the constraint list.

1. Case $(\alpha \stackrel{e}{=} \alpha) :: C'$. This case follows from the induction hypothesis.
2. Case $(\alpha \stackrel{e}{=} \beta) :: C'$ and $\alpha \neq \beta$. The reasoning is similar to case 3 below.
3. Case $(\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. We know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma'$ and the induction hypothesis is
   $\forall\sigma.\,\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma \Rightarrow \sigma \models \{\alpha \mapsto \tau_1 \to \tau_2\}(C')$.
   We have to show
   $\forall\sigma.\sigma = (\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma') \Rightarrow \sigma \models (\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$. Pick an arbitrary σ. Assume $\sigma = \{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma'$. We must show $(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma') \models (\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$. Since we know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma'$, so by the induction hypothesis we know $\sigma' \models \{\alpha \mapsto \tau_1 \to \tau_2\}(C')$. We must show $(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma') \models (\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$. By the definition of satisfiability, we must show:
   (a) $(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma') \models (\alpha \stackrel{e}{=} \tau_1 \to \tau_2)$.
   By Theorem 1 and the definition of satisfiability, we must show $\sigma'(\{\alpha \mapsto \tau_1 \to \tau_2\}(\alpha))$ = $\sigma'(\{\alpha \mapsto \tau_1 \to \tau_2\}(\tau_1 \to \tau_2))$. Since we know $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$, so $\{\alpha \mapsto \tau_1 \to \tau_2\}(\tau_1 \to \tau_2) = \tau_1 \to \tau_2$ and the proof follows.
   (b) $(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma') \models C'$.
   Since we know $\sigma' \models \{\alpha \mapsto \tau_1 \to \tau_2\}(C')$, so by Lemma 5 we know $(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma') \models C'$ as was to be shown.
4. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \alpha) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. Same as case 3 above.
5. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C'$. The induction hypothesis is
   $\forall\sigma.\,\mathsf{unify}((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C') = \sigma \Rightarrow \sigma \models ((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C')$.
   We have to show
   $\forall\sigma'.\,\mathsf{unify}((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C') = \sigma' \Rightarrow \sigma' \models ((\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C')$.
   Pick an arbitrary σ' and assume $\mathsf{unify}\ ((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C') = \sigma'$. Since we know $\mathsf{unify}\ ((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C') = \sigma'$, so by the induction hypothesis we know
   $\sigma' \models ((\tau_1 \stackrel{e}{=} \tau_3) :: (\tau_2 \stackrel{e}{=} \tau_4) :: C')$. But by the definition of satisfiability, we know
   $\sigma'(\tau_1) = \sigma'(\tau_3)$, $\sigma'(\tau_2) = \sigma'(\tau_4)$ and $\sigma' \models C'$.
   To show $\sigma' \models ((\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C')$, we must show:
   (a) $\sigma' \models \tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4$. By the definition of satisfiability, we must show $\sigma'(\tau_1 \to \tau_2) = \sigma'(\tau_3 \to \tau_4)$. But we assumed $\sigma'(\tau_1) = \sigma'(\tau_3)$ and $\sigma'(\tau_2) = \sigma'(\tau_4)$, so this case holds.
   (b) $\sigma' \models C'$. But that we already know.

□

4.2 Axiom ii

Lemma 6. $\forall C.\, \forall \sigma.\, \forall \alpha.\, \forall \tau.\, (\sigma \models C \wedge \alpha \notin \mathsf{FTV}(\tau) \wedge \sigma(\alpha) = \sigma(\tau)) \Rightarrow \sigma \models \{\alpha \mapsto \tau\}(C)$

Proof. By induction on the constraint list C, followed by induction on the structure of the type τ. □

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

Proof. Choose an arbitrary constraint list C. By the definition of extensional equality on finite maps, we must show $\forall \sigma. \forall \sigma'.\, (\mathsf{unify}(C) = \sigma \wedge \sigma' \models C) \Rightarrow \exists \sigma''.\, \forall \alpha.\, \sigma'(\alpha) = (\sigma \circ \sigma'')(\alpha)$.
By functional induction on $\mathsf{unify}(C)$, there are two main cases:

Case $C = [\,]$. Choose an arbitrary σ and σ'. Assume $\mathsf{unify}([\,]) = \sigma$ and $\sigma' \models [\,]$. By the definition of unify, we know $\sigma = \sigma_{\mathbb{E}}$. So we must show $\exists \sigma''. \forall \alpha. \sigma'(\alpha) = (\sigma_{\mathbb{E}} \circ \sigma'')(\alpha)$. Let σ' be the witness for σ'' in $\exists \sigma''. \forall \alpha. \sigma'(\alpha) = (\sigma_{\mathbb{E}} \circ \sigma'')(\alpha)$. Choose an arbitrary α. Then we must show $\sigma'(\alpha) = (\sigma_{\mathbb{E}} \circ \sigma')(\alpha)$. But by Theorem 1, we have $(\sigma_{\mathbb{E}} \circ \sigma')(\alpha) = \sigma'(\sigma_{\mathbb{E}}(\alpha))$. So we must show $\sigma'(\sigma_{\mathbb{E}}(\alpha)) = \sigma'(\alpha)$. But that follows since $\sigma_{\mathbb{E}}(\alpha) = \alpha$.

Case $C \neq [\,]$. We consider the various cases based on the constraint at the head of the constraint list:

1. Case $(\alpha \stackrel{e}{=} \alpha) :: C'$. Apply the induction hypothesis and then this case is trivial.
2. Case $(\alpha \stackrel{e}{=} \beta) :: C'$ and $\alpha \neq \beta$. Reasoning is similar to case 3 below.
3. Case $(\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. We know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma_1$ and the induction hypothesis is
   $\forall \sigma. \forall \sigma'. (\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma \wedge \sigma' \models (\{\alpha \mapsto \tau_1 \to \tau_2\}(C')))$
   $\Rightarrow \exists \sigma''. \forall \alpha'. \sigma'(\alpha') = (\sigma \circ \sigma'')(\alpha')$.
   We must show
   $\forall \sigma_p. \forall \sigma_2. \sigma_p = (\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1) \wedge \sigma_2 \models ((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C')$
   $\Rightarrow \exists \sigma_3. \forall \alpha''. \sigma_2(\alpha'') = (\sigma_p \circ \sigma_3)(\alpha'')$.
   Pick an arbitrary σ_p and σ_2.
   Assume $\sigma_p = \{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1$ and $\sigma_2 \models ((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C')$. We must show $\exists \sigma_3. \forall \alpha''. \sigma_2(\alpha'') = ((\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1) \circ \sigma_3)(\alpha'')$. Since $\sigma_2 \models ((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C')$ so, by the definition of constraint satisfiability, we know $\sigma_2(\alpha) = \sigma_2(\tau_1 \to \tau_2)$ and $\sigma_2 \models C'$. Then, by Lemma 6 and by our assumptions, we know $\sigma_2 \models (\{\alpha \mapsto \tau_1 \to \tau_2\}(C'))$. Since we also know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma_1$, so, by the induction hypothesis, we know $\exists \sigma''. \forall \alpha'. \sigma_2(\alpha') = (\sigma_1 \circ \sigma'')(\alpha')$. We assume $\forall \alpha'. \sigma_2(\alpha') = (\sigma_1 \circ \sigma_4)(\alpha')$, where σ_4 is fresh. Then, to show $\exists \sigma_3. \forall \alpha''. \sigma_2(\alpha'') = ((\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1) \circ \sigma_3)(\alpha'')$, we choose the witness σ_4 and show $\forall \alpha''. \sigma_2(\alpha'') = ((\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1) \circ \sigma_4)(\alpha'')$. Pick an arbitrary α'' and show $\sigma_2(\alpha'') = ((\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1) \circ \sigma_4)(\alpha'')$. By Theorem 1, we must show $\sigma_2(\alpha'') = \sigma_4(\sigma_1(\{\alpha \mapsto \tau_1 \to \tau_2\}(\alpha''))$. There are two cases to consider:
   (a) Case $\alpha \neq \alpha''$. Then we must show $\sigma_2(\alpha'') = \sigma_4(\sigma_1(\alpha''))$. But that follows our assumptions and Theorem 1.
   (b) Case $\alpha = \alpha''$. Then we must show $\sigma_2(\alpha) = \sigma_4(\sigma_1(\tau_1 \to \tau_2))$. Since we know $\sigma_2(\alpha) = \sigma_2(\tau_1 \to \tau_2)$, so we must show $\sigma_2(\tau_1 \to \tau_2) = \sigma_4(\sigma_1(\tau_1 \to \tau_2))$. But that follows from our assumptions and Lemma 1 and Theorem 1.
4. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \alpha) :: C$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. Same as case 3 above.
5. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C$. Apply the induction hypothesis and then this case is trivial.

□

4.3 Axiom iii

Lemma 7. $\forall \alpha, \alpha'. \forall \tau. \forall \sigma.\, \alpha' \in \mathsf{dom\_subst}(\{\alpha \mapsto \tau\} \circ \sigma) \Rightarrow$
$\alpha' \in \mathsf{dom\_subst}(\{\alpha \mapsto \tau\}) \vee \alpha' \in \mathsf{dom\_subst}(\sigma)$

Lemma 8. $\forall \alpha, \alpha'. \forall \tau. \forall \sigma.\, (\alpha \notin \mathsf{FTV}(\tau) \wedge \alpha' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau\} \circ \sigma)) \Rightarrow$
$\alpha' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau\}) \vee \alpha' \in \mathsf{range\_subst}(\sigma)$

Without going into the details, the following lemma helps us in proving Lemma 8. Note that the definition of $\circ$ contains references to higher order functions M.map2 and this lemma helps in not having to reason about M.map2 function but instead we use Theorem 1 to reason about substitution composition.

Lemma 9. $\forall \alpha. \forall \sigma.\, \alpha \in \mathsf{range\_subst}(\sigma) \Leftrightarrow \exists \alpha'. \alpha' \in \mathsf{dom\_subst}(\sigma) \wedge \alpha \in \mathsf{FTV}(\sigma(\alpha'))$

Lemma 10. $\forall \alpha, \alpha'. \forall \tau.\, \forall C.\, (\alpha' \notin \mathsf{FTV}(\tau) \wedge \alpha' \in \mathsf{FTV}(\{\alpha \mapsto \tau\}(C))) \Rightarrow \alpha' \in \mathsf{FTV}(C)$.

Lemma 11. $\forall C.\, \forall \sigma.\, \mathsf{unify}(C) = \sigma \Rightarrow \mathsf{dom\_subst}(\sigma) \subseteq \mathsf{FTV}(C)$

Proof. By functional induction on $\mathsf{unify}(C)$ and Lemma 7. □

We focus on the proof of the most involved lemma.

Lemma 12. $\forall C.\, \forall \sigma.\, \mathsf{unify}(C) = \sigma \Rightarrow \mathsf{range\_subst}(\sigma) \subseteq \mathsf{FTV}(C)$

Proof. Choose an arbitrary C. Unfolding the definition of $\subseteq$, we must show
$\forall \sigma.\, \mathsf{unify}(C) = \sigma \Rightarrow \forall \alpha'.\, \alpha' \in \mathsf{range\_subst}(\sigma) \Rightarrow \alpha' \in \mathsf{FTV}(C)$. By functional induction on $\mathsf{unify}(C)$, there are two main cases:

Case $C = [\,]$. Then, by the definition of unify, we know $\sigma = \sigma_{\mathbb{E}}$. So we must show $\mathsf{range\_subst}(\sigma_{\mathbb{E}}) \subseteq \mathsf{FTV}([\,])$. The proof follows from the definition of range_subst and the definition of FTV.

Case $C \neq [\,]$. We consider the various cases based on the constraint at the head of the constraint list:

1. Case $(\alpha \stackrel{e}{=} \alpha) :: C'$. The induction hypothesis is:
   $\forall \sigma.\, \mathsf{unify}(C') = \sigma \Rightarrow \forall \alpha''. \alpha'' \in \mathsf{range\_subst}(\sigma) \Rightarrow \alpha'' \in \mathsf{FTV}(C')$
   and we must show
   $\forall \sigma.\, \mathsf{unify}(C') = \sigma \Rightarrow \forall \alpha'. \alpha' \in \mathsf{range\_subst}(\sigma) \Rightarrow \alpha' \in \mathsf{FTV}((\alpha \stackrel{e}{=} \alpha) :: C')$.
   Pick an arbitrary σ and assume $\mathsf{unify}(C') = \sigma$. Pick an arbitrary α'.
   Assume $\alpha' \in \mathsf{range\_subst}(\sigma)$ and show $\alpha' \in \mathsf{FTV}((\alpha \stackrel{e}{=} \alpha) :: C')$.
   Since we know $\mathsf{unify}(C') = \sigma$, so, by the induction hypothesis, we know
   $\forall \alpha''.\, \alpha'' \in \mathsf{range\_subst}(\sigma) \Rightarrow \alpha'' \in \mathsf{FTV}(C')$. Since we also know $\alpha' \in \mathsf{range\_subst}(\sigma)$, so we know $\alpha' \in \mathsf{FTV}(C')$. That also means $\alpha' \in \mathsf{FTV}((\alpha \stackrel{e}{=} \alpha) :: C')$ as was to be shown.
2. Case $(\alpha \stackrel{e}{=} \beta) :: C'$ and $\alpha \neq \beta$. Reasoning is similar to case 3 below.
3. Case $(\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. We know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma_1$, and the induction hypothesis is
   $\forall \sigma'.\, \mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma' \Rightarrow$
   $\quad \forall \alpha'. \alpha' \in \mathsf{range\_subst}(\sigma') \Rightarrow \alpha' \in \mathsf{FTV}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C'))$.
   We must show
   $\forall \alpha''. \alpha'' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1) \Rightarrow \alpha'' \in \mathsf{FTV}((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C')$.
   Pick an arbitrary α'' and assume $\alpha'' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1)$. We must show $\alpha'' \in \mathsf{FTV}(\{\alpha \stackrel{e}{=} \tau_1 \to \tau_2\} :: C')$. There are two cases:

(a) Case $\alpha'' = \alpha$. Then clearly $\alpha'' \in \mathsf{FTV}(\{\alpha \stackrel{e}{=} \tau_1 \to \tau_2\}(C'))$ as was to be shown.
(b) Case $\alpha'' \neq \alpha$. Then we have two cases:
   i. $\alpha'' \in \mathsf{FTV}(\tau_1 \to \tau_2)$. Then clearly $\alpha'' \in \mathsf{FTV}(\{\alpha \stackrel{e}{=} \tau_1 \to \tau_2\} :: C')$ as was to be shown.
   ii. $\alpha'' \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. Then we must show $\alpha'' \in \mathsf{FTV}(C')$. Since we know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma_1$, so by the induction hypothesis we know $\forall \alpha'.\alpha' \in \mathsf{range\_subst}(\sigma_1) \Rightarrow \alpha' \in \mathsf{FTV}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C'))$.
   Since $\alpha'' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_1)$ so, by Lemma 8, we know either $\alpha'' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau_1 \to \tau_2\})$ or $\alpha'' \in \mathsf{range\_subst}(\sigma_1)$. Again, there are two cases:
   A. Case $\alpha'' \in \mathsf{range\_subst}(\{\alpha \mapsto \tau_1 \to \tau_2\})$. Then $\alpha'' \in \mathsf{FTV}(\tau_1 \to \tau_2)$ - a contradiction.
   B. Case $\alpha'' \in \mathsf{range\_subst}(\sigma_1)$. Then from the induction hypothesis we know $\alpha'' \in \mathsf{FTV}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C'))$. Then by Lemma 10, $\alpha'' \in \mathsf{FTV}(C')$ as was to be shown.

4. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \alpha) :: C$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. Same as case 3 above.
5. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C$. Apply the induction hypothesis and then this case is trivial.

□

Theorem 4. $\forall C.\, \forall \sigma.\, \mathsf{unify}\, C = \sigma \Rightarrow \mathsf{FTV}(\sigma) \subseteq \mathsf{FTV}(C)$

Proof. By the definition of FTV and by Lemma 11 and Lemma 12. □

4.4 Axiom iv

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

$$\begin{array}{lcl} \mathsf{subterms}(\alpha) & \stackrel{def}{=} & [\,] \\ \mathsf{subterms}(\tau_1 \to \tau_2) & \stackrel{def}{=} & \tau_1 :: \tau_2 :: (\mathsf{subterms}\ \tau_1) +\!\!+ (\mathsf{subterms}\ \tau_2) \end{array}$$

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

Lemma 13. $\forall \tau, \tau'.\, \tau \in \mathsf{subterms}(\tau') \Rightarrow \forall \tau''.\, \tau'' \in \mathsf{subterms}(\tau) \Rightarrow \tau'' \in \mathsf{subterms}(\tau')$

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

Lemma 14. $\forall \tau.\, \neg\, \tau \in \mathsf{subterms}(\tau)$

Proof. By induction on the structure of the type τ and by Lemma 13. □

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

Lemma 15. $\forall \sigma.\, \forall \alpha.\, \forall \tau.\, \alpha \in \mathsf{subterms}(\tau) \Rightarrow \sigma(\alpha) \neq \sigma(\tau)$

Proof. By induction on the structure of the type τ and by Lemma 14. □

Lemma 16. $\forall \sigma.\, \forall \alpha.\, \forall \tau_1, \tau_2.\, \alpha \in \mathsf{FTV}(\tau_1) \vee \alpha \in \mathsf{FTV}(\tau_2) \Rightarrow \alpha \in \mathsf{subterms}(\tau_1 \to \tau_2)$

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. $\forall\sigma.\forall\alpha.\forall\tau_1,\tau_2.\,\alpha\in\mathsf{FTV}(\tau_1)\vee\alpha\in\mathsf{FTV}(\tau_2)\Rightarrow\sigma(\alpha)\neq\sigma(\tau_1\to\tau_2)$

Proof. By Lemma 15 and 16. □

This is the only theorem where the failure cases are interesting. So in the following theorem we carry along the constructor that shows success or failure of unify function call.

Theorem 5. $\forall C.\,\forall\sigma.\,\sigma\models C\Rightarrow\exists\sigma'.\,\mathsf{unify}(C)=\mathsf{Some}\;\sigma'$

Proof. Choose an arbitrary C and σ. By functional induction on $\mathsf{unify}(C)$, there are two main cases:

Case $C=[\,]$. Assume $\sigma\models[\,]$. Then we must show $\exists\sigma'.\,\mathsf{unify}([\,])=\mathsf{Some}\;\sigma'$. Let $\sigma_{\mathbb{E}}$ be the witness for σ' in $\exists\sigma'.\,\mathsf{unify}([\,])=\mathsf{Some}\;\sigma'$. So we must show $\mathsf{unify}\;[\,]=\mathsf{Some}\;\sigma_{\mathbb{E}}$ but that follows from the definition of unify.

Case $C\neq[\,]$. We consider the various cases based on the constraint at the head of the constraint list:

1. Case $(\alpha\stackrel{e}{=}\alpha)::C'$. Apply the induction hypothesis and then this case is trivial.
2. Case $(\alpha\stackrel{e}{=}\beta)::C'$ and $\alpha\neq\beta$. Reasoning is similar to case 3 below.
3. Case $(\alpha\stackrel{e}{=}\tau_1\to\tau_2)::C'$ and $\alpha\notin\mathsf{FTV}(\tau_1\to\tau_2)$. We know $\mathsf{unify}(\{\alpha\mapsto\tau_1\to\tau_2\}(C'))=\mathsf{None}$ and the induction hypothesis is: $\sigma'\models(\{\alpha\mapsto\tau_1\to\tau_2\}(C'))\Rightarrow\exists\sigma''.\,\mathsf{unify}(\{\alpha\mapsto\tau_1\to\tau_2\}(C'))=\mathsf{Some}\;\sigma''$. We must show $\sigma'\models((\alpha\stackrel{e}{=}\tau_1\to\tau_2)::C')\Rightarrow\exists\sigma_3.\,\mathsf{None}=\mathsf{Some}\;\sigma_3$. Assume $\sigma'\models((\alpha\stackrel{e}{=}\tau_1\to\tau_2)::C')$, *i.e.*, $\sigma'(\alpha)=\sigma'(\tau_1\to\tau_2)$ and $\sigma'\models C'$. We must show $\exists\sigma_3.\,\mathsf{None}=\mathsf{Some}\;\sigma_3$. By Lemma 6 and by our assumptions, we know $\sigma'\models(\{\alpha\mapsto\tau_1\to\tau_2\}(C'))$. So, by the induction hypothesis, we know $\exists\sigma''.\,\mathsf{None}=\mathsf{Some}\;\sigma''$. Since we know $\exists\sigma''.\,\mathsf{None}=\mathsf{Some}\;\sigma''$, so assume $\mathsf{None}=\mathsf{Some}\;\sigma'''$, where σ''' is fresh, but that is a contradiction and so this case holds.
4. Case $(\alpha\stackrel{e}{=}\tau_1\to\tau_2)::C'$ and $\alpha\in\mathsf{FTV}(\tau_1\to\tau_2)$. Then, we must show $\sigma'\models((\alpha\stackrel{e}{=}\tau_1\to\tau_2)::C')\Rightarrow\exists\sigma_3.\mathsf{None}=\mathsf{Some}\;\sigma_3$. Assume $\sigma'\models((\alpha\stackrel{e}{=}\tau_1\to\tau_2)::C')$, *i.e.*, $\sigma'(\alpha)=\sigma'(\tau_1\to\tau_2)$ and $\sigma'\models C'$. Since we know $\alpha\in\mathsf{FTV}(\tau_1\to\tau_2)$, *i.e.*, either $\alpha\in\mathsf{FTV}(\tau_1)$ or $\alpha\in\mathsf{FTV}(\tau_2)$, so by Corollary 1 $\sigma'(\alpha)\neq\sigma'(\tau_1\to\tau_2)$, which is a contradiction. Thus the proof follows trivially.
5. Case $(\tau_1\to\tau_2\stackrel{e}{=}\alpha)::C$ and $\alpha\notin\mathsf{FTV}(\tau_1\to\tau_2)$. Similar to case 3.
6. Case $(\tau_1\to\tau_2\stackrel{e}{=}\alpha)::C$ and $\alpha\in\mathsf{FTV}(\tau_1\to\tau_2)$. Similar to case 4.
7. Case $(\tau_1\to\tau_2\stackrel{e}{=}\tau_3\to\tau_4)::C$. Apply the induction hypothesis and then this case is trivial.

□

4.5 Axiom v

The following lemmas are needed for the main proof, the first two follow by induction on the structure of the type τ and the third by induction on C.

Lemma 17. $\forall\sigma.\,\forall\alpha.\,\forall\tau.\,\alpha\notin\mathsf{FTV}(\tau)\wedge\alpha\notin\mathsf{FTV}(\sigma)\Rightarrow\alpha\notin\mathsf{FTV}\;(\sigma(\tau))$

Lemma 18. $\forall\alpha.\,\forall\tau,\tau'.\,\alpha\notin\mathsf{FTV}(\tau)\Rightarrow\{\alpha\mapsto\tau'\}(\tau)=\tau$

Lemma 19. $\forall\alpha.\,\forall\tau.\,\forall C.\,\alpha\notin\mathsf{FTV}(\tau)\Rightarrow\alpha\notin\mathsf{FTV}(\{\alpha\mapsto\tau\}(C))$

The theorem we must prove is:

Theorem 6. $\forall C.\ \forall \sigma.\ \mathsf{unify}(C) = \sigma \Rightarrow (\sigma \circ \sigma) \approx \sigma$.

Proof. Pick an arbitrary C. Unfolding the definition of $\approx$, and by Theorem 1, we must show:
$\forall \sigma.\ \mathsf{unify}(C) = \sigma \Rightarrow \forall \alpha.\ \sigma(\sigma(\alpha)) = \sigma(\alpha)$.
By functional induction on unify C, there are two main cases:

Case $C = [\,]$. This case follows since $\forall \alpha.\ \sigma_{\mathbb{E}}(\alpha) = \alpha$.

Case $C \neq [\,]$. We consider the various cases based on the constraint at the head of the constraint list:

1. Case $(\alpha \stackrel{e}{=} \alpha) :: C'$. Apply the induction hypothesis and then this case is trivial.
2. Case $(\alpha \stackrel{e}{=} \beta) :: C'$. Reasoning is similar to case 3 below.
3. Case $(\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. We know unify $(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma$ and the induction hypothesis is:
   $\forall \sigma'.$unify $\{\alpha \mapsto \tau_1 \to \tau_2\}(C') = \sigma' \Rightarrow \forall \alpha'.\sigma'(\alpha') = \sigma'(\sigma'(\alpha'))$
   And we must show:
   $\sigma(\{\alpha \mapsto \tau_1 \to \tau_2\}(\alpha'')) = (\sigma(\{\alpha \mapsto \tau_1 \to \tau_2\}(\sigma(\{\alpha \mapsto \tau_1 \to \tau_2\}(\alpha'')))))$.
   There are two cases:
   (a) Case $\alpha = \alpha''$. Then we must show $\sigma(\tau_1 \to \tau_2) = \sigma(\{\alpha \mapsto \tau_1 \to \tau_2\}(\sigma(\tau_1 \to \tau_2)))$. From Lemma 19 and Theorem 4, we know that $\alpha \notin \mathsf{FTV}(\sigma)$. Since $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$ and $\alpha \notin \mathsf{FTV}(\sigma)$, so by Lemma 17, $\alpha \notin \mathsf{FTV}(\sigma(\tau_1 \to \tau_2))$. By Lemma 18 (choosing τ' to be $\tau_1 \to \tau_2$), we get $\sigma(\tau_1 \to \tau_2) = \{\alpha \mapsto \tau_1 \to \tau_2\}(\sigma(\tau_1 \to \tau_2))$. So now we must show $\sigma(\tau_1 \to \tau_2) = \sigma(\sigma(\tau_1 \to \tau_2))$. Then, by Lemma 1, we must show $\forall \beta.\ \sigma(\beta) = \sigma(\sigma(\beta))$. Choose an arbitrary β and show $\sigma(\beta) = \sigma(\sigma(\beta))$, but that follows from the induction hypothesis (by choosing σ' to be σ and α' to be β) and our assumptions.
   (b) Case $\alpha \neq \alpha''$. Then we must show $\sigma(\alpha'') = \sigma(\{\alpha \mapsto \tau_1 \to \tau_2\}(\sigma(\alpha'')))$. From Lemma 19 and Theorem 4, we know that $\alpha \notin \mathsf{FTV}(\sigma)$. Since $\alpha \notin \mathsf{FTV}(\alpha'')$ and $\alpha \notin \mathsf{FTV}(\sigma)$, so by Lemma 17, $\alpha \notin \mathsf{FTV}(\sigma(\alpha''))$. By Lemma 18 and using τ' to be $\tau_1 \to \tau_2$ we get $\sigma(\alpha'') = (\{\alpha \mapsto \tau_1 \to \tau_2\}(\sigma(\alpha'')))$. So now we must show $\sigma(\alpha'') = \sigma(\sigma(\alpha''))$, but that follows from the induction hypothesis (by choosing σ' to be σ and α' to be α'') and our assumptions.
4. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \alpha) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. Same as Case 3.
5. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C'$. Apply the induction hypothesis and then this case is trivial.

□

4.6 Axiom vi

The theorem we must prove is:

Theorem 7. $\forall \sigma.$unify $[\,] = \sigma \Rightarrow\ \sigma = \sigma_{\mathbb{E}}$

Proof. Choose an arbitrary σ. Assume unify $[\,] = \sigma$. Unfold the definition of unify. Then we know $\sigma = \sigma_{\mathbb{E}}$ as was to be shown. □

4.7 Axiom vii

The main proof requires a lemma, which we mention next.

Lemma 20. $\forall C, C'.\ \forall \alpha.\ \forall \tau.\ \{\alpha \mapsto \tau\}(C) +\!\!+ \{\alpha \mapsto \tau\}(\mathbb{C}') = \{\alpha \mapsto \tau\}(C +\!\!+ C')$

The theorem we must prove is:

Theorem 8. $\forall C, C_2.\, \forall \sigma', \sigma'', \sigma'''.\, (\mathsf{unify}(C) = \sigma' \wedge \mathsf{unify}(\sigma'(C_2)) = \sigma'' \wedge \mathsf{unify}(C +\!\!+ C_2) = \sigma''') \Rightarrow \sigma''' \approx (\sigma' \circ \sigma'')$

Proof. Pick an arbitrary C. By Theorem 1 and unfolding the definition of $\approx$, we must show:
$\forall C_2.\, \forall \sigma', \sigma'', \sigma'''.\, (\mathsf{unify}(C) = \sigma' \wedge \mathsf{unify}(\sigma'(C_2)) = \sigma'' \wedge \mathsf{unify}(C +\!\!+ C_2) = \sigma''') \Rightarrow \forall \alpha'.\, \sigma'''(\alpha') = \sigma''(\sigma'(\alpha'))$.
By functional induction on $\mathsf{unify}(C)$, there are two main cases:

Case $C = [\,]$. Follows from Theorem 7 and the assumptions.

Case $C \neq [\,]$. Consider the various cases based on the constraint at the head of the constraint list.

1. Case $(\alpha \stackrel{e}{=} \alpha) :: C'$. This case follows from the induction hypothesis and the definition of append.
2. Case $(\alpha \stackrel{e}{=} \beta) :: C'$ and $\alpha \neq \beta$. Similar to case 3 below.
3. Case $(\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C'$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. We know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma$. The induction hypothesis is:
   $\forall C_1.\, \forall \sigma_1, \sigma_2, \sigma_3.$
   $(\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) = \sigma_1 \wedge$
   $\mathsf{unify}(\sigma_1(C_1)) = \sigma_2 \wedge$
   $\mathsf{unify}((\{\alpha \mapsto \tau_1 \to \tau_2\}(C')) +\!\!+ C_1) = \sigma_3)$
   $\Rightarrow \forall \alpha''.\, \sigma_3(\alpha'') = \sigma_2(\sigma_1(\alpha''))$.
   We must show:
   $\forall C_2.\, \forall \sigma', \sigma'', \sigma'''.$
   $(\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma = \sigma' \wedge$
   $\mathsf{unify}(\sigma'(C_2)) = \sigma'' \wedge$
   $\mathsf{unify}(((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C') +\!\!+ C_2) = \sigma''')$
   $\Rightarrow \forall \alpha'.\, \sigma'''(\alpha') = \sigma''(\sigma'(\alpha'))$.
   Pick an arbitrary C_2, σ', σ'' and σ'''. Assume $\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma = \sigma'$ and $\mathsf{unify}(\sigma'(C_2)) = \sigma''$ and $\mathsf{unify}(((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: C') +\!\!+ C_2) = \sigma'''$. By the definition of append, the last assumption is $\mathsf{unify}((\alpha \stackrel{e}{=} \tau_1 \to \tau_2) :: (C' +\!\!+ C_2)) = \sigma'''$.
   Unfolding the unify definition once, we know $\mathsf{unify}(\{\alpha \mapsto \tau_1 \to \tau_2\}(C' +\!\!+ C_2)) = \sigma_T$, where $\sigma''' = \{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_T$. Also, since $\sigma' = \{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma$, so we know $\mathsf{unify}((\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma)(C_2)) = \sigma''$. Since we know $\sigma''' = \{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_T$, so we must show $\forall \alpha'.\, (\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma_T)(\alpha') = \sigma''((\{\alpha \mapsto \tau_1 \to \tau_2\} \circ \sigma)(\alpha'))$. Pick an arbitrary α'. By Theorem 1, we must show
   $\sigma_T(\{\alpha \mapsto \tau_1 \to \tau_2\}(\alpha')) = \sigma''(\sigma(\{\alpha \mapsto \tau_1 \to \tau_2\}(\alpha')))$. There are two cases:
   (a) Case $\alpha = \alpha'$. Then we must show $\sigma_T(\tau_1 \to \tau_2) = \sigma''(\sigma(\tau_1 \to \tau_2))$. But by Lemma 1, we must show $\forall \alpha'''.\, \sigma_T(\alpha''') = \sigma''(\sigma(\alpha'''))$. Pick an arbitrary α''' and so we must show $\sigma_T(\alpha''') = \sigma''(\sigma(\alpha'''))$. But that follows from the induction hypothesis (by choosing C_1 to be $\{\alpha \mapsto \tau_1 \to \tau_2\}(C_2)$, σ_1 to be σ, σ_2 to be σ'' and σ_3 to be σ_T) and the definition of substitution composition and Lemma 20 and the assumptions.
   (b) Case $\alpha \neq \alpha'$. Then we must show $\sigma_T(\alpha') = \sigma''(\sigma(\alpha'))$. But that follows from the induction hypothesis (by choosing C_1 to be $\{\alpha \mapsto \tau_1 \to \tau_2\}(C_2)$, σ_1 to be σ, σ_2 to be σ'' and σ_3 to be σ_T) and the definition of substitution composition and Lemma 20 and the assumptions.

4. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \alpha) :: C$ and $\alpha \notin \mathsf{FTV}(\tau_1 \to \tau_2)$. Same as the above case.
5. Case $(\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: C$. Apply the induction hypothesis.

□

5 Related Work and Conclusions

Unification is fundamentally used in type inference. There are formalizations of the unification algorithm in a number of different theorem provers [8, 21, 22]. We comment on the implementation in the CoLoR library [7]. CoLoR is an extensive and very successful library supporting reasoning about termination and rewriting. A Coq implementation of the unification algorithm was recently released [8]. 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 CoLoR the substitutions are represented by functions from type variables to a generalized term structure. The axioms verified here are not explicitly verified in CoLoR, however their library 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 functional induction which follows the structure of our algorithm. The unification algorithm in CoLoR is specified in a significantly different style (as an iterated step function).

Though many lemmas were simple, many others required generalization in order for the proof to go through. Our choice of finite maps library to represent substitutions helped us significantly. Coq's finite maps library is expressive enough to specify complicated definitions (substitution composition, range elements) yet the reasoning with them is simple if we abstract away from the actual definition and look at the extensional behavior instead. Since we used an interface, we could not really argue about the normal substitution equality. Our specification of unification was in a functional style but the definition was general recursive. This meant that we had to show the termination using a well-founded ordering. Once termination was established, the `functional induction` tactic helped us immensely in reasoning about the first-order unification algorithm.

The entire formalization (all seven axioms) is done in Coq 8.1.pl3 version in around 5000 lines of specifications and tactics, and is available online at `http://www.cs.uwyo.edu/~skothari`.

We would like to thank Santiago Zanella (INRIA - Sophia Antipolis) for showing us how to encode lexicographic ordering for 3-tuples in Coq. We thank Frederic Blanqui for answering our queries regarding the new release of CoLoR library, Laurent Théry for making his Coq formulation of Sudoku [24] available on the web, Stéphane Lescuyer and other Coq-club members for answering our queries on the Coq-club mailing list, and Christian Urban (TU Munich) for discussing at length the MGU axioms used in their verification of Algorithm W [25]. Finally, we want to thank anonymous referees for their detailed comments and suggestions (on an earlier draft of this paper), which greatly improved the presentation of this paper.

References

[1] The Coq proof assistant reference manual version 8.1.3: *Finite Map Interface*. `http://coq.inria.fr/V8.1/stdlib/Coq.FSets.FMapInterface.html`.

[2] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström & Björn von Sydow (1994). *A user's guide to ALF*.

[3] Krzysztof R. Apt (2003): *Principles of Constraint Programming*. Cambridge University Press.

[4] F. Baader & W. Snyder (2001): *Unification Theory*. In: A. Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, I, chapter 8, Elsevier Science, pp. 445–532.

[5] Gilles Barthe & Pierre Courtieu (2002): *Efficient Reasoning about Executable Specifications in Coq*. In: *TPHOLs '02:15th International Conference on Theorem Proving in Higher Order Logics*, pp. 31–46.

[6] Yves Bertot & Pierre Castran (2004): *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Springer.

[7] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer & A. Koprowski (2006): *CoLoR, a Coq Library on Rewriting and termination*. In: *8th International Workshop on Termination (WST '06)*, pp. 69–73.

[8] Frederic Blanqui (2008). *CoLoR, a Coq library on rewriting and termination.* `http://color.inria.fr/doc/CoLoR.Term.WithArity.AUnif.html`.

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

[10] Robert S. Boyer & J. Strother Moore (1988): *A Computational Logic Handbook*. Academic Press Professional, Inc.

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

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

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

[14] Elmar Eder (1985): *Properties of substitutions and unifications*. *J. Symb. Comput.* 1(1), pp. 31–46.

[15] Sunil Kothari & James Caldwell (2008): *On Extending Wand's Type Reconstruction Algorithm to Handle Polymorphic Let*. In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, University of Athens, pp. 254–263.

[16] Sunil Kothari & James L. Caldwell (2009): *A Machine Checked Model of MGU Axioms: Applications of Finite Maps and Functional Induction*. In: *UNIF'09*, pp. 17–31.

[17] J. L. Lassez, M. J. Maher & K. Marriott (1988): *Unification revisited*. *Foundations of deductive databases and logic programming* , pp. 587–625.

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

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

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

[21] L. C. Paulson (1985): *Verifying the Unification Algorithm in LCF*. *Sci. of Comp. Prog.* 5, pp. 143–169.

[22] J. Rouyer (1994): *Developpement d'Algorithmes dans le Calcul des Constructions*. Ph.D. thesis, Institut National Polytechnique de Lorraine, Nancy, France.

[23] The Coq development team (2007): *The Coq proof assistant reference manual*. INRIA, LogiCal Project. Version 8.1.3.

[24] Laurent Théry (2006). *Sudoku in Coq*.

[25] Christian Urban & Tobias Nipkow (2009): *From Semantics to Computer Science*, chapter Nominal verification of algorithm W. Cambridge University Press.

[26] M. Wand (1987): *A Simple Algorithm and Proof for Type Inference*. *Fundamenta Informaticae* 10, pp. 115–122.