# A Mechanized Textbook Proof of a Type Unification Algorithm

Rodrigo Ribeiro[1] and Carlos Camarão[2]

[1] Universidade Federal de Ouro Preto, João Monlevade, Minas Gerais, Brazil
`rodrigo@decsi.ufop.br`,
[2] Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brazil
`camarao@dcc.ufmg.br`

**Abstract.** Unification is the core of type inference algorithms for modern functional programming languages, like Haskell. As a first step towards a formalization of a type inference algorithm for such programming languages, we present a formalization in Coq of a type unification algorithm that follows classic algorithms presented in programming language textbooks.

## 1 Introduction

Modern functional programming languages like Haskell [1] and ML [2] provide type inference to free the programmer from having to write (almost all) type annotations in programs. Compilers for these languages can discover missing type information through a process called type inference [3].

Type inference algorithms are usually divided into two components: constraint generation and constraint solving [4]. For languages that use ML-style (or parametric) polymorphism, constraint solving reduces to first order unification.

A sound and complete algorithm for first order unification is due to Robinson [5]. The soundness and completeness proofs have a constructive nature, and can thus be formalized in proof assistant systems based on type theory, like Coq [6] and Agda [7]. Formalizations of unification have been reported before in the literature [8–11] using different proof assistants, but none of them follows the style of textbook proofs (cf. e.g. [12, 13]).

As a first step towards a full formalization of a type inference algorithm for Haskell, in this article, we describe an axiom-free formalization of type unification in the Coq proof assistant, that follows classic algorithms on type systems for programming languages [12, 13]. The formalization is "axiom-free" because it does not depend on axioms like function extensionality, proof irrelevance or the law of the excluded middle, i.e. our results are integrally proven in Coq.

More specifically, our contributions are:

1. A mechanization of a termination proof as it can be found in e.g. [12, 13]. In these books, the proof is described as "easy to check". In our formalization, it was necessary to decompose the proof in several lemmas in order to convince Coq's termination checker.

2. A correct by construction formalization of unification. In our formalization the unification function has a dependent type that specifies that unification produces the most general unifier of a given set of equality constraints, or a proof that explains why this set of equalities does not have a unifier (i.e. our unification definition is a view [14] on lists of equality constraints).

We chose Coq to develop this formalization because it is an industrial strength proof assistant that has been used in several large scale projects such as a Certified C compiler [15], a Java Card platform [16] and on verification of mathematical theorems (cf. e.g. [17, 18]).

The rest of this paper is organized as follows. Section 2 presents a brief introduction to the Coq proof assistant. Section 3 presents some definitions used in the formalization. Section 4 presents the unification algorithm. Termination, soundness and completeness proofs are described in Sections 4.1 and 4.2, respectively. Section 5 presents details about proof automation techniques used in our formalization. Section 6 presents related work and Section 7 concludes.

While all the code on which this paper is based has been developed in Coq, we adopt a "lighter" syntax in the presentation of its code fragments. In the introductory Section 2, however, we present small Coq source code pieces. We chose this presentation style in order to improve readability, because functions that use dependently typed pattern matching require a high number of type annotations, that would deviate from our objective of providing a formalization that is easy to understand. For theorems and lemmas, we sketch the proof strategy but omit tactic scripts. The developed formalization was verified using Coq version 8.4 and it is available online [19].

## 2 A Taste of Coq Proof Assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC) [6], a higher order typed $\lambda$-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[3], where types represent logical formulas, $\lambda$-terms represent proofs [20] and the task of checking if a piece of text is a proof of a given formula corresponds to checking if the term that represents the proof has the type corresponding to the given formula.

However, writing a proof term whose type is that of a logical formula can be a hard task, even for very simple propositions. In order to make the writing of complex proofs easier, Coq provides *tactics*, which are commands that can be used to construct proof terms in a more user friendly way.

As a tiny example, consider the task of proving the following simple formula of propositional logic:

$$(A \to B) \to (B \to C) \to A \to C$$

[3] Abbreviation of Brouwer, Heyting, Kolmogorov, de Bruijn and Martin-Löf Correspondence. This is also known as the Curry-Howard "isomorphism".

In Coq, such theorem can be expressed as:

```
Section EXAMPLE.
   Variables A B C : Prop.
   Theorem example : (A -> B) -> (B -> C) -> A -> C.
   Proof.
       intros H H' HA. apply H'. apply H. assumption.
   Qed.
End EXAMPLE.
```

In the previous source code piece, we have defined a Coq section named `EXAMPLE`[4] which declares variables `A`, `B` and `C` as being propositions (i.e. with type `Prop`). Tactic `intros` introduces variables `H`, `H'` and `HA` into the (typing) context, respectively with types `A -> B`, `B -> C` and `A` and leaves goal `C` to be proved. Tactic `apply`, used with a term `t`, generates goal `P` when there exists `t: P -> Q` in the typing context and the current goal is `Q`. Thus, `apply H'` changes the goal from `C` to `B` and `apply H` changes the goal to `A`. Tactic `assumption` traverses the typing context to find a hypothesis that matches with the goal.

We define next a proof of the previous propositional logical formula that, in contrast to the previous proof, that was built using tactics (`intros`, `apply` and `assumption`), is coded directly as a function:

```
   Definition example' : (A -> B) -> (B -> C) -> A -> C :=
       fun (H : A -> B) (H' : B -> C) (HA : A) => H' (H HA).
```

However, even for very simple theorems, coding a definition directly as a Coq term can be a hard task. Because of this, the use of tactics has become the standard way of proving theorems in Coq. Furthermore, the Coq proof assistant provides not only a great number of tactics but also a domain specific language for scripted proof automation, called $\mathcal{L}$tac. In this work, the developed proofs follow the style advocated by Chlipala [21], where most proofs are built using $\mathcal{L}$tac scripts, to automate proof steps and make them more robust. Details about $\mathcal{L}$tac can be found in [21, 6].

## 3 Definitions

### 3.1 Types

We consider a language of simple types formed by type variables, type constants (also called type constructors) and functional types given by the following grammar:

$$\tau ::= \alpha \mid c \mid \tau \to \tau$$

where $\alpha$ stands for a type variable and $c$ a type constructor. All meta-variables ($\tau$, $\alpha$ and $c$) can appear primed or subscripted and as usual we consider that $\to$ associates to the right.

Identifiers for variables and constructors are represented as natural numbers, following standard practice in formalized meta-theory [22, 23]. We are aware that

[4] In Coq, we can use sections to delimit the scope of local variables.

choosing this representation of types is not adequate to represent Haskell's types, since it does not allow the occurrence of $n$-ary type constructors. Using $n$-ary type constructors will only clutter definitions due to the need of using kinds[5]. Since the presence of kind information is orthogonal to unification, we prefer to omit it in order to clarify definitions and proofs.

The list of type variables of type $\tau$ is denoted by $\mathrm{FV}(\tau)$.

The *size* of a given type $\tau$, given by the number of arrows, type variables and constructors in $\tau$, is denoted by $size(\tau)$. Formally:

$$\begin{array}{ll} size(\tau_1 \to \tau_2) & = 1 + size(\tau_1) + size(\tau_2) \\ size(\tau) & = 1 \text{ otherwise } (\tau = \alpha \text{ or } \tau = c, \text{ for some } \alpha, c) \end{array}$$

We let $\tau_1 \overset{e}{=} \tau_2$ denote the equality constraint between two types $\tau_1$ and $\tau_2$.

Lists of equality constraints are represented by meta-variable $\mathbb{C}$. We use the left-associative operator :: for constructing lists: $a :: x$ denotes the list formed by head $a$ and tail $x$.

The definition of free type variables for constraints and their lists are defined in a standard way and the size of constraints and constraint lists are defined as the sum of their constituent types. The following simple lemmas will be later used to establish termination of the unification algorithm, defined in Section 4.

**Lemma 1** *For all types $\tau_1, \tau_1', \tau_2, \tau_2'$ and all lists of constraints $\mathbb{C}$ we have that:*

$$size((\tau_1 \overset{e}{=} \tau_1') :: (\tau_2 \overset{e}{=} \tau_2') :: \mathbb{C}) < size((\tau_1 \to \tau_2 \overset{e}{=} \tau_1' \to \tau_2') :: \mathbb{C})$$

*Proof.* Induction over $\mathbb{C}$ using the definition of *size*.

**Lemma 2** *For all types $\tau, \tau'$ and all lists of constraints $\mathbb{C}$ we have that*

$$size(\mathbb{C}) < size((\tau \overset{e}{=} \tau') :: \mathbb{C})$$

*Proof.* Induction over $\tau$ and case analysis over $\tau'$, using the definition of *size*.

### 3.2 Substitutions

Substitutions are functions mapping type variables to types. For convenience, a substitution is considered as a finite mapping $[\alpha_1 \mapsto \tau_1, ..., \alpha_n \mapsto \tau_n]$, for $i = 1, \ldots, n$, which is also abbreviated as $[\overline{\alpha} \mapsto \overline{\tau}]$ ($\overline{\alpha}$ and $\overline{\tau}$ denoting sequences built from sets $\{\alpha_1, ..., \alpha_n\}$ and $\{\tau_1, ..., \tau_n\}$, respectively). Meta-variable $S$ is used to denote substitutions.

In our formalization, a mapping $[\alpha \mapsto \tau]$ is represented as a pair of a variable and a type. Substitutions are represented as lists of mappings, taking advantage

[5] Kinds classify type expressions in the same way as types classify terms. More details about the use of kinds and high-order operators can be found in [13].

of the fact that a variable never appears twice in a substitution. The domain of a substitution, denoted by $dom(S)$, is defined as:

$$dom(S) = \{\alpha \mid S(\alpha) = \tau, \alpha \neq \tau\}$$

Following [10], we define *substitution application* in a variable-by-variable way; first, let the application of a mapping $[\alpha \mapsto \tau']$ to $\tau$ be defined by recursion over the structure of $\tau$:

$$\begin{array}{lll} [\alpha \mapsto \tau']\,(\tau_1 \to \tau_2) & = ([\alpha \mapsto \tau']\,\tau_1) \to ([\alpha \mapsto \tau']\,\tau_2) & \\ [\alpha \mapsto \tau']\,\alpha & = \tau' & \\ [\alpha \mapsto \tau']\,\tau & = \tau \ \text{ otherwise } (\tau = \alpha' \text{ for some } \alpha' \neq \alpha, \text{ or} \\ & \qquad\qquad\qquad\qquad\quad \tau = c \text{ for some } c) \end{array}$$

Next, substitution application follows by recursion on the number of mappings of the substitution, using the above defined application of a single mapping:

$$S(\tau) = \begin{cases} \tau & \text{if } S = [\,] \\ S'([\alpha \mapsto \tau']\,\tau) & \text{if } S = [\alpha \mapsto \tau'] :: S' \end{cases}$$

Application of a substitution to an equality constraint is defined in a straight-foward way:

$$S\,(\tau \overset{e}{=} \tau') = S(\tau) \overset{e}{=} S(\tau')$$

In order to maintain our development on a fully constructive ground, we use the following lemma, to cater for proofs of equality of substitutions. This lemma is used to prove that the result of the unification algorithm yields the most general unifier of a given set of types.

**Lemma 3** *For all substitutions $S$ and $S'$, if $S(\alpha) = S'(\alpha)$ for all variables $\alpha$, then $S(\tau) = S'(\tau)$ for all types $\tau$.*

*Proof.* Induction over $\tau$, using the definition of substitution application.

Substitutions and types are subject to well-formedness conditions, described in the next section.

## 3.3 Well-Formedness Conditions

Now, we consider notions of well-formedness with regard to types, substitutions and constraints. These notions are crucial to give simple proofs for termination, soundness and completeness of the unification algorithm.

Well-formed conditions are expressed in terms of a type variable context, $\mathcal{V}$, that contains, in each step of the execution of the unification algorithm, the *complement* of the set of type variables that are in the domain of the unifier. This context is used to formalize some notions that are assumed as immediate facts in textbooks, like: “at each recursive call of the unification algorithm, the number of distinct type variables occurring in constraints decreases” or “after applying a substitution $S$ to a given type $\tau$, we have that $FV(S(\tau)) \cap dom(S) = \emptyset$”.

We consider that:

- A type $\tau$ is well-formed in $\mathcal{V}$, written as $\mathit{wf}(\mathcal{V}, \tau)$, if all type variables that occur in $\tau$ are in $\mathcal{V}$.
- A constraint $\tau_1 \overset{e}{=} \tau_2$ is well-formed, written as $\mathit{wf}(\mathcal{V}, \tau_1 \overset{e}{=} \tau_2)$, if both $\tau_1$ and $\tau_2$ are well-formed in $\mathcal{V}$.
- A list of constraints $\mathbb{C}$ is well-formed in $\mathcal{V}$, written as $\mathit{wf}(\mathcal{V}, \mathbb{C})$, if all of its equality constraints are well-formed in $\mathcal{V}$.
- A substitution $S = \{[\alpha \mapsto \tau]\} :: S'$ is well-formed in $\mathcal{V}$, written as $\mathit{wf}(\mathcal{V}, S)$, if the following conditions apply:
  - $\alpha \in \mathcal{V}$
  - $\mathit{wf}(\mathcal{V} - \{\alpha\}, \tau)$
  - $\mathit{wf}(\mathcal{V} - \{\alpha\}, S')$

The requirement that type $\tau$ is well-formed in $\mathcal{V} - \{\alpha\}$ is necessary in order for $[\alpha \mapsto \tau]$ to be a well-formed substitution. This avoids cyclic equalities that would introduce infinite type expressions.

The well-formedness conditions are defined as recursive Coq functions that compute dependent types from a given variable context and a type, constraint or substitution.

A first application of these well-formedness conditions is to enable a simple definition of composition of substitutions. Let $S_1$ and $S_2$ be substitutions such that $\mathit{wf}(\mathcal{V}, S_1)$ and $\mathit{wf}(\mathcal{V} - \mathit{dom}(S_1), S_2)$. The composition $S_2 \circ S_1$ can be defined simply as the append operation of these substitutions:

$$S_2 \circ S_1 = S_1 \mathbin{{+}{+}} S_2$$

The idea of indexing substitutions by type variables that can appear in its domain and its use to give a simple definition of composition was proposed in [10].

We say that a substitution $S$ is more general than $S'$, written as $S \leq S'$, if there exists a substitution $S_1$ such that $S' = S_1 \circ S$.

The definition of composition of substitutions satisfies the following theorem:

**Theorem 1 (Substitution Composition and Application)** *For all types $\tau$ and all substitutions $S_1$, $S_2$ such that $\mathit{wf}(\mathcal{V}, S_1)$ and $\mathit{wf}(\mathcal{V} - \mathit{dom}(S_1), S_2)$ we have that $(S_2 \circ S_1)(\tau) = S_2(S_1(\tau))$.*

*Proof.* By induction over the structure of $S_2$.

### 3.4 Occurs Check

Type unification algorithms use a well-known occurs check in order to avoid the generation of cyclic mappings in a substitution, like $[\alpha \mapsto \alpha \to \alpha]$. In the context of finite type expressions, cyclic mappings do not make sense. In order to define the occurs check, we first define a dependent type, $\mathit{occurs}(\alpha, \tau)$, that is inhabited[6] only if $\alpha \in \mathrm{FV}(\tau)$:

[6] According to the BHK-interpretation, a type is inhabited only if it represents a logic proposition that is provable.

$$\begin{array}{lll} occurs(\alpha, \tau_1 \rightarrow \tau_2) & = occurs(\alpha, \tau_1) \vee occurs(\alpha, \tau_2) & \\ occurs(\alpha, \alpha) & = \texttt{True} & \\ occurs(\alpha, \tau) & = \texttt{False} \text{ otherwise} & \\ & \quad \text{i.e. if } \tau = \alpha' \text{ for some } \alpha' \neq \alpha \text{ or } \tau = c \text{ for some } c & \end{array}$$

Coq types `True` and `False` are the unit and empty type[7], respectively. Note that $occurs(\alpha, \tau)$ is provable if and only if $\alpha \in \mathrm{FV}(\tau)$.

Using type *occurs*, decidability of the occurs check can be established, by using the following theorem:

**Lemma 4 (Decidability of occurs check)** *For all variables $\alpha$ and all types $\tau$, we have that either $occurs(\alpha, \tau)$ or $\neg\, occurs(\alpha, \tau)$ holds.*

*Proof.* Induction over the structure of $\tau$.

If a variable $\alpha$ does not occur in a well-formed type, this type is well-formed in a variable context where $\alpha$ does not occur. This simple fact is an important step used to prove termination of unification. The next lemmas formalize this notion.

**Lemma 5** *For all variables $\alpha_1, \alpha_2$ and all variable contexts $\mathcal{V}$, if $\alpha_1 \in \mathcal{V}$ and $\alpha_2 \neq \alpha_1$ then $\alpha_1 \in (\mathcal{V} - \{\alpha_2\})$.*

*Proof.* Induction over $\mathcal{V}$.

**Lemma 6** *Let $\tau$ be a well-formed type in a variable context $\mathcal{V}$ and let $\alpha$ be a variable such that $\neg occurs(\alpha, \tau)$. Then $\tau$ is well-formed in $\mathcal{V} - \{\alpha\}$.*

*Proof.* Induction on the structure of $\tau$, using Lemma 5 in the variable case.

## 4 The Unification Algorithm

We use the following standard presentation of the first-order unification algorithm, where $\tau \equiv \tau'$ denotes a decidable equality test between $\tau$ and $\tau'$:

Our formalization differs from the presented algorithm (Figure 1) in two aspects:

– Since this presentation of the unification algorithm is general recursive, i.e., the recursive calls aren't necessarily made on structurally smaller arguments, we need to define it using recursion on proofs that *unify*'s arguments form a well-founded relation [6].

[7] In type theory terminology, the unit type is a type that has a unique inhabitant and the empty type is a type that does not have inhabitants. Under BHK-interpretation, they correspond to a true and false propositions, respectively [20].

(1) $\mathit{unify}([\,]) = [\,]$
(2) $\mathit{unify}((\alpha \overset{e}{=} \alpha) :: \mathbb{C}) = \mathit{unify}(\mathbb{C})$
(3) $\mathit{unify}((\alpha \overset{e}{=} \tau) :: \mathbb{C}) = \text{if } \mathit{occurs}(\alpha, \tau) \text{ then fail else } \mathit{unify}([\alpha \mapsto \tau]\mathbb{C}) \circ [\alpha \mapsto \tau]$
(4) $\mathit{unify}((\tau \overset{e}{=} \alpha) :: \mathbb{C}) = \text{if } \mathit{occurs}(\alpha, \tau) \text{ then fail else } \mathit{unify}([\alpha \mapsto \tau]\mathbb{C}) \circ [\alpha \mapsto \tau]$
(5) $\mathit{unify}((\tau_1 \to \tau_2 \overset{e}{=} \tau \to \tau') :: \mathbb{C}) = \mathit{unify}((\tau_1 \overset{e}{=} \tau) :: (\tau_2 \overset{e}{=} \tau') :: \mathbb{C})$
(6) $\mathit{unify}((\tau \overset{e}{=} \tau') :: \mathbb{C}) = \text{if } \tau \equiv \tau' \text{ then } \mathit{unify}(\mathbb{C}) \text{ else fail}$

**Fig. 1.** Unification algorithm.

– Instead of returning just a substitution that represents the argument constraint unifier, we return a proof that such substitution is indeed its most general unifier or a proof explaining that such unifier does not exist, when *unify* fails.

These two aspect are discussed in Sections 4.1 and 4.2, respectively.

It is worth mentioning that there are some Coq extensions that make the definitions of general recursive functions and functions defined by pattern matching on dependent types easier, namely commands `Function` and `Program`, respectively. However, according to [24], these are experimental extensions. Thus, we prefer to use well established approaches to overcome these problems: 1) use of a recursion principle derived from the definition of a well-founded relation [6] and 2) annotate every pattern matching construct in order to make explicit the relation between function argument and return types.

### 4.1 Termination Proof

The unification algorithm always terminates for any list of equalities, either by returning their most general unifier or by establishing that there is no unifier. The termination argument uses a notion of *degree* of a list of constraints $\mathbb{C}$, written as $\mathit{degree}(\mathbb{C})$, defined as a pair $(m, n)$, where $m$ is the number of distinct type variables in $\mathbb{C}$ and $n$ is the total size of the types in $\mathbb{C}$. We let $(n, m) \prec (n', m')$ denote the usual lexicographic ordering of degrees.

Textbooks usually consider it "easy to check" that each clause of the unification algorithm either terminates (with success or failure) or else make a recursive call with a list of constraints that has a lexicographically smaller degree. Since the implemented unification function is defined by recursion over proofs of lexicographic ordering of degrees, we must ensure that all recursive calls are made on smaller lists of constraints. In lines 3 and 4 of Figure 1, the recursive calls are made on a list of constraints of smaller degree, because the list of constraints $[\alpha \mapsto \tau]\mathbb{C}$ will decrease by one the number of type variables occurring in it. This is formalized in the following lemma:

**Lemma 7 (Substitution application decreases degree)** *For all variables $\alpha \in \mathcal{V}$, all well-formed types $\tau$ and well-formed lists of constraints $\mathbb{C}$, it holds* *that*

$$degree([\alpha \mapsto \tau]\, \mathbb{C}) \prec degree((\alpha \overset{e}{=} \tau) :: \mathbb{C})$$

*Proof.* Induction over $\mathbb{C}$.

On line 5 of Figure 1, we have that the recursive call is made on a constraint that has more equalities than the original but has a smaller degree, as shown by the following lemma.

**Lemma 8 (Fewer Arrows implies lower degree)** *For all well-formed types $\tau_1, \tau_2, \tau_1', \tau_2'$ and all well-formed lists of constraints $\mathbb{C}$, it holds that*

$$degree((\tau_1 \overset{e}{=} \tau_1', \tau_2 \overset{e}{=} \tau_2') :: \mathbb{C}) \prec degree((\tau_1 \to \tau_2 \overset{e}{=} \tau_1' \to \tau_2') :: \mathbb{C})$$

*Proof.* Immediate from Lemma 1.

Finally, the recursive calls in lines 2 and 6 also decrease the degree of the input list of constraints, according to the following:

**Lemma 9 (Less constraints implies lower degree)** *For all well-formed types $\tau$, $\tau'$ and all well-formed list of constraints $\mathbb{C}$, it holds that*

$$degree(\mathbb{C}) \prec degree(\{\tau \overset{e}{=} \tau'\} :: \mathbb{C})$$

*Proof.* Immediate from Lemma 2.

## 4.2 Soundness and Completeness Proof

Given an arbitrary list of constraints, the unification algorithm either fails or returns its most general unifier. We have the following properties:

- Soundness: the substitution produced is a unifier of the constraints.
- Completeness: the returned substitution is the least unifier, according to the substitution ordering defined in Section 3.2.

A substitution $S$ is called a unifier of a list of constraints $\mathbb{C}$ according to whether $unifier(\mathbb{C}, S)$ is provable, where $unifier(\mathbb{C}, S)$ is defined by induction on $\mathbb{C}$ as follows:

$$\begin{aligned} &unifier([], S) &&= \texttt{True} \\ &unifier((\tau \overset{e}{=} \tau') :: \mathbb{C}', S) &&= S(\tau) = S(\tau') \wedge unifier(\mathbb{C}', S) \end{aligned}$$

A substitution $S$ is a most general unifier of a list of constraints $\mathbb{C}$ if, for any other unifier $S'$ of $\mathbb{C}$, there exists $S_1$ such that $S' = S_1 \circ S$; formally:

$$least(S, \mathbb{C}) = \forall S'.\, unifier(\mathbb{C}, S') \to \exists S_1.\, \forall \alpha.\, (S_1 \circ S)(\alpha) = S'(\alpha)$$

The type of the unification function is a dependent type that ensures the following property of the returned substitution $S$:

$$\big(\mathit{unifier}(\mathbb{C}, S) \wedge \mathit{least}(S, \mathbb{C})\big) \vee \text{UnifyFailure}(\mathbb{C})$$

where UnifyFailure($\mathbb{C}$) is a type that encodes the reason why unification of $\mathbb{C}$ fails. There are two possible causes of failure: 1) an occurs check error, 2) an error caused by trying to unify distinct type constructors.

In the formalization source code, the definition of the *unify* function contains "holes"[8] to mark positions where proof terms are expected. Instead of writing such proof terms, we left them unspecified and use tactics to fill them with appropriate proofs. In the companion source code, the unification function is full of such holes and they mark the position of proof obligations for soundness, completeness and termination for each equation of the definition of *unify*.

In order to prove soundness obligations we define several small lemmas that are direct consequences of the definition of the application of substitutions, which are omitted for brevity. Other lemmas necessary to ensure soundness are sketched below. They specify properties of unification and application of substitutions.

**Lemma 10** *For all type variables $\alpha$, types $\tau, \tau'$ and substitutions $S$, if $S(\alpha) = S(\tau')$ then $S(\tau) = S([\alpha \mapsto \tau']\,\tau)$.*

*Proof.* Induction over the structure of $\tau$.

**Lemma 11** *For all type variables $\alpha$, types $\tau$, variable contexts $\mathcal{V}$ and constraint sets $\mathbb{C}$, if $S(\alpha) = S(\tau)$ and $\mathit{unifier}(\mathbb{C}, S)$ then $\mathit{unifier}([\alpha \mapsto \tau]\,\mathbb{C}, S)$.*

*Proof.* Induction over $\mathbb{C}$ using Lemma 10.

Completeness proof obligations are filled by scripted automatic proof tactics using Lemma 3.

## 5 Automating Proofs

Most parts of most proofs used to prove properties of programming languages and of algorithms are exercises that consist of a lot of somewhat tedious steps, with just a few cases representing the core insights. It is not unusual for mechanized proofs to take significant amounts of code on uninteresting cases and quite significant effort on writing that code. In order to deal with this problem in our development, we use $\mathcal{L}$tac, Coq's domain specific language for writing custom tactics, and Coq built-in automatic tactic `auto`, which implements a Prolog-like resolution proof construction procedure using hint databases within a depth limit.

The main $\mathcal{L}$tac custom tactic used in our development is a proof state simplifier that performs several manipulations on the hypotheses and on the conclusion.

[8] A hole in a function definition is a subterm that is left unspecified. In Coq, holes are represented by underscores and such unspecified parts of a definition are usually filled by tactic generated terms.

It is defined by means of two tactics, called `mysimp` and `s`. Tactic `mysimp` tries to reduce the goal and repeatedly applies tactic `s` to the proof state until all goals are solved or a failure occurs.

Tactic `s`, shown in Figure 2, performs pattern matching on a proof state using $\mathcal{L}$tac `match goal` construct. Patterns have the form:

$$\texttt{[h}_1\texttt{ : t}_1\texttt{,h}_2\texttt{ : t}_2\texttt{ ... |- C ] => tac}$$

where each of $\texttt{t}_i$ and `C` are expressions, which represents hypotheses and conclusion, respectively, and `tac` is the tactic that is executed when a successful match occurs. Variables with question marks can occur in $\mathcal{L}$tac patterns, and can appear in `tac` without the question mark. Names $\texttt{h}_i$ are binding occurrences that can be used in `tac` to refer to a specific hypothesis. Another aspect worth mentioning is keyword `context`. Pattern matching with `context[e]` is successful if `e` occurs as a subexpression of some hypothesis or in the conclusion. In Figure 2, we use `context` to automate case analysis on equality tests on identifiers and natural numbers, as shown below

```
[ |- context[eq_id_dec ?a ?b] ] =>
        destruct (eq_id_dec a b) ; subst ; try congruence
```

Tactic `destruct` performs case analysis on a term, `subst` searchs the context for a hypothesis of the form `x = e` or `e = x`, where `x` is a variable and `e` is an expression, and replaces all occurrences of `x` by `e`. Tactic `congruence` is a decision procedure for equalities with uninterpreted functions and data type constructors [6].

```
Ltac s :=
  match goal with
    | [ H : _ /\ _ |- _] => destruct H
    | [ H : _ \/ _ |- _] => destruct H
    | [ |- context[eq_id_dec ?a ?b] ] =>
            destruct (eq_id_dec a b) ; subst ; try congruence
    | [ |- context[eq_nat_dec ?a ?b] ] =>
            destruct (eq_nat_dec a b) ; subst ; try congruence
    | [ x : (id * ty)%type |- _ ] =>
            let t := fresh "t" in destruct x as [x t]
    | [ H : (_,_) = (_,_) |- _] => inverts* H
    | [ H : Some _ = Some _ |- _] => inverts* H
    | [ H : Some _ = None |- _] => congruence
    | [ H : None = Some _ |- _] => congruence
    | [ |- _ /\ _] => split
    | [ H : ex _ |- _] => destruct H
  end.

Ltac mysimp := repeat (simpl; s) ; simpl; auto with arith.
```

**Fig. 2.** Main proof state simplifier tactic.

Tactic `inverts* H` generates necessary conditions used to prove `H` and afterwards executes tactic `auto`.[9] Tactic `split` divides a conjunction goal in its constituent parts.

Besides $\mathcal{L}$tac scripts, the main tool used to automate proofs in our development is tactic `auto`. This tactic uses a relatively simple principle: a database of tactics is repeatedly applied to the initial goal, and then to all generated subgoals, until all goals are solved or a depth limit is reached.[10] Databases to be used — called *hint databases* — can be specified by command `Hint`, which allows declaration of which theorems are part of a certain hint database. The general form of this command is:

```
Hint Resolve thm1 thm2 ... thmn : db.
```

where $\texttt{thm}_i$ are defined lemmas or theorems and `db` is the database name to be used. When calling `auto` a hint database can be specified, using keyword `with`. In Figure 2, `auto` is used with database `arith` of basic Peano arithmetic properties. If no database name is specified, theorems are declared to be part of hint database `core`. Proof obligations for termination are filled using lemmas 7, 8 e 9 that are included in hint databases. Failures of unification, for a given list of constraints $\mathbb{C}$, is represented by `UnifyFailure` and proof obligations related to failures are also handled by `auto`, thanks to the inclusion of `UnifyFailure` constructors as `auto` hints using command

```
Hint Constructors UnifyFailure.
```

## 6 Related Work

Formalization of unification algorithms has been the subject of several research works [8–11].

In Paulson's work [8] the representation of terms, built by using a binary operator, uses equivalence classes of finite lists where order and multiplicity of elements is considered irrelevant, deviating from simple textbook unification algorithms ([13, 12]).

Bove's formalization of unification [9] starts from a Haskell implementation and describes how to convert it into a term that can be executed in type theory by acquiring an extra termination argument (a proof of termination for the actual input) and a proof obligation (that all possible inputs satisfy this termination argument). This extra termination argument is an inductive type whose constructors and indices represent the call graph of the defined unification function. Bove's technique can be seen as an specific implementation of the technique for general recursion based on well founded relations [26], which is the one implemented on Coq's standard library, used in our implementation. Also, Bove presents soundness and completeness proofs for its implementation together with

[9] This tactic is defined on a tactic library developed by Arthur Charguraud [25].

[10] The default depth limit used by `auto` is 5.

the function definition (as occurs with our implementation) as well as by providing theorems separated from the actual definitions. She argues that the first formalization avoids code duplication since soundness and completeness proofs follow the same recursive structure of the unification function. Bove's implementation is given in Alf, a dependently typed programming language developed at Chalmers that is currently unsupported.

McBride [10] develops a unification function that is structurally recursive on the number of non-unified variables on terms being unified. The idea of its termination argument is that at each step the unification algorithm gets rid of one unresolved variable from terms, a property that is carefully represented with dependent types. Soundness and completeness proofs are given as separate theorems in a technical report [27]. McBride's implementation is done on `OLEG`, a dependently typed programming language that is nowadays also unsupported.

Kothari [11] describes an implementation of a unification function in Coq and proves some properties of most general unifiers. Such properties are used to postulate that unification function does produce most general unifiers on some formalizations of type inference algorithms in type theory [28]. Kothari's implementation does not use any kind of scripted proof automation and it uses the experimental command `Function` in order to generate an induction principle from its unification function structure. He uses this induction principle to prove properties of the defined unification function.

Avelar et al.'s proof of completeness [29] is not focused on the proof that the unifier $S$ of types $\overline{\tau}$, returned by the unification algorithm, is the least of all existing unifiers of $\overline{\tau}$. It involves instead properties that specify: i) $dom(S) \subseteq \mathrm{FV}(\overline{\tau})$, ii) the contra-domain of $S$ is a subset of $\mathrm{FV}(\overline{\tau}) - dom(S)$, and iii) if the unification algorithm fails then there is no unifier. The proofs involve a quite large piece of code, and the program does not follow simple textbook unification algorithms. The proofs are based instead on concepts like the first position of conflict between terms (types) and on resolution of conflicts. More recent work of Avelar et al. [30] extends the previous formalization by the description of a more elaborate and efficient first-order unification algorithm. The described algorithm navigates the tree structure of the two terms being unified in such a way that, if the two terms are not unifiable then, after the difference at the first position of conflict between the terms is eliminated through a substitution, the search of a possible next position of conflict is computed through application of auxiliary functions starting from the previous position.

## 7 Conclusion

We have given a complete formalization of termination, soundness and completeness of a type unification algorithm in the Coq proof assistant. To the best of our knowledge, the proposed formalization is the first to follow the structure of termination proofs presented in classical textbooks on type systems [13, 12]. Soundness and completeness proofs of unification are coupled with the algorithm definition and are filled by scripted proof tactics using previously proved lemmas.

The developed formalization has 610 lines of code and around 94 lines of comments. The formalization is composed of 31 lemmas and theorems, 34 type and function definitions and 2 inductive types. Most of the implementation effort has been done on proving termination, which takes 293 lines of our code, expressed in 21 theorems. Compared with Kothari's implementation, that is written in more than 1000 lines, our code is more compact.

We intend to use this formalization to develop a complete type inference algorithm for Haskell in the Coq proof assistant. The developed work is available online [19].

## References

1. Peyton Jones, S.: Haskell 98 Language and Libraries: the Revised Report. (2003)
2. Milner, R., Tofte, M., Harper, R.: Definition of standard ML. MIT Press (1990)
3. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3) (1978) 348–375
4. Pottier, F., Rémy, D.: The essence of ML type inference. In Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2005) 389–489
5. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1) (1965) 23–41
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
7. Bove, A., Dybjer, P., Norell, U.: A brief overview of agda — a functional language with dependent types. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs '09, Berlin, Heidelberg, Springer-Verlag (2009) 73–78
8. Paulson, L.C.: Verifying the unification algorithm in lcf. CoRR **cs.LO/9301101** (1993)
9. Bove, A.: Programming in Martin-Löf type theory: Unification - A non-trivial example (November 1999) Licentiate Thesis of the Department of Computer Science, Chalmers University of Technology.
10. McBride, C.: First-order unification by structural recursion. J. Funct. Program. **13**(6) (2003) 1061–1075
11. Kothari, S., Caldwell, J.: A machine checked model of idempotent mgu axioms for lists of equational constraints. In Fernandez, M., ed.: Proceedings 24th International Workshop on Unification. Volume 42 of EPTCS. (2010) 24–38
12. Mitchell, J.C.: Foundations of Programming Languages. MIT Press, Cambridge, MA, USA (1996)
13. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
14. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. **14**(1) (2004) 69–111
15. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009) 107–115
16. Barthe, G., Dufay, G., Jakubiec, L., de Sousa, S.M.: A formal correspondence between offensive and defensive javacard virtual machines. In Cortesi, A., ed.: VMCAI. Volume 2294 of Lecture Notes in Computer Science., Springer (2002) 32–45

17. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In Kapur, D., ed.: ASCM. Volume 5081 of Lecture Notes in Computer Science., Springer (2007) 333
18. Gonthier, G.: Engineering mathematics: the odd order theorem proof. In Giacobazzi, R., Cousot, R., eds.: POPL, ACM (2013) 1–2
19. Ribeiro, R., et al.: A mechanized textbook proof of a type unification algorithm — on-line repository. https://github.com/rodrigogribeiro/unification (2015)
20. Sørensen, M., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism. Number v. 10 in Studies in Logic and the Foundations of Mathematics. Elsevier (2006)
21. Chlipala, A.: Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
22. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5) (1972) 381 – 392
23. Charguéraud, A.: The locally nameless representation. J. Autom. Reasoning **49**(3) (2012) 363–408
24. Coq Developement Team: Coq Proof Assistant — Reference Manual. `http://coq.inria.fr/distrib/current/refman/ /` (2014)
25. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjoberg, V., Yorgey, B.: Software Foundations. Electronic textbook (2015)
26. Nordström, B.: Terminating general recursion. BIT Numerical Mathematics **28**(3) (1988) 605–619
27. McBride, C.: First-order unification by structural recursion — correctness proof
28. Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm w in isabelle/hol. J. Autom. Reason. **23**(3) (November 1999) 299–318
29. Avelar, A.B., de Moura, F.L.C., Galdino, A.L., Ayala-Rincón, M.: Verification of the completeness of unification algorithms à la robinson. In Dawar, A., de Queiroz, R.J.G.B., eds.: Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010, Brasilia, Brazil, July 6-9, 2010. Proceedings. Volume 6188 of Lecture Notes in Computer Science., Springer (2010) 110–124
30. Avelar, A.B., Galdino, A.L., de Moura, F.L.C., Ayala-Rincón, M.: First-order unification in the PVS proof assistant. Logic Journal of the IGPL **22**(5) (2014) 758–789