Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 27, Num. 3 (2020) 13-24

RESEARCH ARTICLE

A Mechanized Proof of a Textbook Type Unification Algorithm

Uma prova de correção formalmente verificada de um algoritmo de unificação de tipos

Maycon Amaro[1], Rodrigo Ribeiro[1]*, André Rauber Du Bois[2]

Abstract: Unification is the core of type inference algorithms for modern functional programming languages, like Haskell and SML. 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. We also report on the use of such formalization to build a correct type inference algorithm for the simply typed λ-calculus.

Keywords: Unification — Type Inference — Coq proof assistant

Resumo: A unificação é um componente central de algoritmos de inferência de tipos presentes em linguagens funcionais modernas, como Haskell e SML. Esse trabalho relata os primeiros passos em direção a formalização, usando o assistente de provas Coq, de um algoritmo de inferência de tipos conforme este é apresentado em livros texto da área de linguagens de programação. A partir do algoritmo formalizado, descrevemos uma implementação de um algoritmo de inferência de tipos para o λ-cálculo simplesmente tipado.

Palavras-Chave: Unificação — Inferência de tipos — Assistente de provas Coq

[1] *Universidade Federal de Ouro Preto, Ouro Preto, Minas Gerais, Brazil*
[2] *Universidade Federal de Pelotas, Pelotas, Rio Grande do Sul, Brazil*
***Corresponding author**: rodrigo.ribeiro@ufop.edu.br

DOI: https://doi.org/10.22456/2175-2745.100968 • **Received:** 10/03/2020 • **Accepted:** 28/04/2020

1. Introduction

Modern functional programming languages like Haskell [1] and SML [2] allows programmers to write program texts without requiring programmers to write 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, 9, 10, 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 described in e.g. [12, 13]. These textbook proofs are referred to as “straightforward”. 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 either 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).
3. We use the developed formalization to certify a type inference algorithm for the simply typed λ-calculus (STLC) in Coq. The type inference algorithm is constructed by combining constraint generation and constraint solving (unification). We also use Coq extraction

feature to produce a Haskell implementation of the formalized type inference algorithm.

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] and a Java Card platform [16], as well as on the verification of mathematical theorems (cf. e.g. [17, 18]).

This paper extends previous work [19] mainly by (1) defining a correct constraint-based type inference algorithm for STLC by combining a constraint generator with the previously formalized unification algorithm, and (2) producing a Haskell implementation of type inference for STLC from our Coq formalization.

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 reports on our efforts on using the formalized algorithm in the implementation of a type inference algorithm. Section 7 presents related work and Section 8 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 uses Coq version 8.5pl2 and it is available online [20].

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 λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[1], where types represent logical formulas, λ-terms represent proofs [21] 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 make it easier to construct proof terms.

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

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

In Coq, such theorem can be expressed as:

```
Section EXAMPLE.
   Variables A B C : Prop.
   Theorem ex : (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[2] 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, instead of 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 $\mathscr{L}$tac. In this work, the developed proofs follow the style, advocated by e.g. Chlipala [22], where most proofs are built using $\mathscr{L}$tac scripts, to automate proof steps and make them more robust. Details about $\mathscr{L}$tac can be found in [22, 6].

Another feature of Coq is the so-called extraction [23], which generates data types and functions in Haskell, OCaml and Scheme, from a Coq formalization by erasing all values whose types are in Prop universe. As an example of this feature, consider the following Coq source code piece, which defines a certified predecessor function for natural numbers in Peano notation:

After defining inductive type nat, we declare function pred, which returns the predecessor of a given natural number together with a evidence that the returned value is indeed the predecessor of input value or a proof that such input is

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

[2] In Coq, sections can be used to delimit the scope of local variables.

```
Section PRED.

  Inductive nat : Set :=
   | O : nat
   | S : nat -> nat.

   Definition pred (n : nat)
        : {m | n = S m} + {n = O}.
     exact (match n as n'
         return {m | n' = S m} + {n' = O} with
             | O => inright eq_refl
             | S m => inleft _ (exist _ m eq_refl)
             end).
   Defined.
End PRED.
Extraction Language Haskell.
Extract Inductive sumor =>
    "Maybe" ["Just" "Nothing"].
Recursive Extraction pred.
```

Figure 1. Simple verified predecessor function and its extraction commands

equal to zero. This type is specified using type constructors `sig` and `sumor`. Type constructor `sig` can be understood as an existential quantifier[3], since its value constructor

```
exist : forall x : A, P x -> sig P
```

receives a witness `x` and a proof that this value satisfies a property (`P`). Notation `{x : A | P x}` represents type `sig (fun x : A => P)`. For a type `A : Type` and a proposition `B : Prop`, inductive type `sumor A B` represents values of type `A` or proofs of `B` and Coq's standard library defines it with the following notation: `A + {B}`. The type of `pred` uses `sig` to guarantee that the returned value is the predecessor of input `n`, i.e. `{m | n = S m}`, and `|`sumor— to combine it with the type of a proof that input `n` is equal to zero (i.e. `n = O`).

Function `pred` is declared using tactic `exact` which allows direct specification of a term. Definition of `pred` is as follows: If `n = O`, we return a proof that `n = 0`, denoted by term `inright eq_refl`; otherwise, we have that `n = S m` and we return `m` as `n`'s predecessor together with a proof of this fact. Both cases uses constructor `eq_refl` that represents *propositional equality* proofs and it has the type

```
eq_refl : forall {A : Type}{x : A}, x = x
```

By default, Coq extracts code to OCaml but, we can configure it to produce Haskell or Scheme code using command `Extraction Language`. Also we can customize extraction to modify how functions and data types are extracted. As an example of this feature, in Figure 1, we use command:

```
Extract Inductive sumor =>
   "Maybe" ["Just" "Nothing"].
```

to extract `sumor` type to Haskell's `Maybe`. Using this command, Coq does not create a Haskell type for `sumor` and maps its first constructor to `Just` and the second to `Nothing`. Such customizations must be done with care, since they could introduce unexpected behaviour, if we don't pay attention to constructor order [24].

Haskell code extracted from Figure 1 is shown in Figure 2.

```
module Main where

import qualified Prelude

type Sig a = a
-- singleton inductive,
-- whose constructor was exist

data Nat = O | S Nat

pred :: Nat -> Maybe Nat
pred n =
   case n of {
      O -> Nothing;
      S m -> Just m}
```

Figure 2. Extracted Haskell code from Figure 1.

3. Definitions

3.1 Terms

Following common practice, we consider the following syntax for λ-terms, where meta-variable x denotes (term) variables:

$$e \quad ::= \quad x \mid \lambda x.e \mid e\,e \mid c$$

Our syntax only consider just one constant value, c, of type C. We use a term representation that is based on names to represent variable binding. Usage of binding representations (like De Bruijn indexes, locally nameless representation or parametric high-order abstract syntax [25, 26]) is orthogonal to our work, since our focus is on type inference and not on semantics, where capture free (term) substitutions do not play an important role.

3.2 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:

$$\tau \quad ::= \quad \alpha \mid C \mid \tau \rightarrow \tau$$

where α stands for a type variable and C is a type constructor. All meta-variables (τ, α) can appear primed or subscripted and, as usual, we consider that $\rightarrow$ associates to the right and represent type variable names as natural numbers.

[3] The main difference between an inductive type constructed with `sig` and the one constructed with an existential quantifier (`ex`) is that the former survives program extraction, since extraction only remove values whose type is a propostion (universe `Prop`).

The list of type variables within type τ is denoted by $\mathrm{FV}(\tau)$ and its definition is straightforward.

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

$$\begin{array}{lcl} size(\tau_1 \rightarrow \tau_2) & = & 1 + size(\tau_1) + size(\tau_2) \\ size(\alpha) & = & 1 \\ size(C) & = & 1 \end{array}$$

We let $\tau_1 \overset{e}{=} \tau_2$ denote the equality constraint between two types τ_1 and τ_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. Also, we follow Haskell notation and represent singleton list containing x by $[x]$.

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:*

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

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

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

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

Proof. Induction over τ and case analysis over τ', using the definition of *size*. □

3.3 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 of the fact that a variable never appears twice in a substitution.

There are other possible representations for substitutions (e.g. finite maps implemented as red-black trees). We chose to represent them as lists to ease the task automating proofs by scripting custom tactics. Coq standard library provides several containers (finite-maps, finite-sets, etc) implemented as balanced-tree structures. However, since the Coq library relies on modules, the finite map structure is hidden from the client users which would add unnecessary complexity on proof automation by inspecting substitution's structure.

The domain of a sub stitution, denoted by $dom(S)$, is defined as:

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

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

$$\begin{array}{lcl} [\alpha \mapsto \tau'](\tau_1 \rightarrow \tau_2) & = & ([\alpha \mapsto \tau']\,\tau_1) \rightarrow ([\alpha \mapsto \tau']\,\tau_2) \\ [\alpha \mapsto \tau']\,\alpha & = & \tau' \\ [\alpha \mapsto \alpha']\,\alpha & = & \alpha', if\ \alpha' \neq \alpha \\ [\alpha \mapsto \alpha']C & = & \mathrm{C} \end{array}$$

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 straightfoward 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 α, then $S(\tau) = S'(\tau)$ for all types τ.*

Proof. Induction over τ, using the definition of substitution application. □

3.4 Type System

In order to formalize a type inference algorithm for STLC, we need to define its type system. Following standard practice, we let meta-variable Γ denote typing contexts. We also define the following operations on typing contexts:

$$\begin{array}{l} \Gamma, x : \tau = (\Gamma - \{x : \tau' \in \Gamma\}) \cup \{x : \tau\} \\ \Gamma(x) = \tau, \text{ if } x : \tau \in \Gamma \end{array}$$

We let symbol $\emptyset$ denote the empty typing context. Application of substitutions on typing contexts is straightforwardly defined by recursion using application of substitution on types. We represent substitution application on types and typing contexts using the same notation.

The type system is presented as a syntax directed proof system for judgements $\Gamma \vdash e : \tau$ which means that term e has type τ using information in typing context Γ.

$$\frac{}{\Gamma \vdash c : C}\ (Const) \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (Var)$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}\ (Abs)$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\ (App)$$

In order to ensure that type inference is sound w.r.t. STLC type system, we need to show that derivability is preserved by type substitutions. This is ensured by the next theorem.

Theorem 1. *Let Γ be a typing context, e a term and τ a type such that $\Gamma \vdash e : \tau$. Then, for every substitution S, we have that $S(\Gamma) \vdash e : S(\tau)$.*

Proof. By induction over the derivation of $\Gamma \vdash e : \tau$. □

3.5 Well-Formedness Conditions

Now, we consider notions of well-formedness with regard to types, typing contexts, 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, $\mathscr{V}$, 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 τ, we have that $FV(S(\tau)) \cap dom(S) = \emptyset$".

We consider that:

- A type τ is well-formed in $\mathscr{V}$, written as $wf(\mathscr{V}, \tau)$, if all type variables that occur in τ are in $\mathscr{V}$.
- A constraint $\tau_1 \overset{e}{=} \tau_2$ is well-formed in $\mathscr{V}$, written as $wf(\mathscr{V}, \tau_1 \overset{e}{=} \tau_2)$, if both τ_1 and τ_2 are well-formed in $\mathscr{V}$.
- A list of constraints $\mathbb{C}$ is well-formed in $\mathscr{V}$, written as $wf(\mathscr{V}, \mathbb{C})$, if all of its equality constraints are well-formed in $\mathscr{V}$.
- A typing context Γ is well-formed in $\mathscr{V}$, written as $wf(\mathscr{V}, \Gamma)$, if for all $x : \tau \in \Gamma$, we have $wf(\mathscr{V}, \tau)$.
- A substitution S is well-formed in $\mathscr{V}$, written as $wf(\mathscr{V}, S)$, if either $S = []$ or, if $S = \{[\alpha \mapsto \tau]\} :: S'$, the following conditions apply:
  - $\alpha \in \mathscr{V}$
  - $wf(\mathscr{V} - \{\alpha\}, \tau)$
  - $wf(\mathscr{V} - \{\alpha\}, S')$

The requirement that type τ is well-formed in $\mathscr{V} - \{\alpha\}$ 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 $wf(\mathscr{V}, S_1)$ and $wf(\mathscr{V} - dom(S_1), S_2)$. The composition $S_2 \circ S_1$ can be defined simply as the append operation of these substitutions:

$$S_2 \circ S_1 = S_1 ++ S_2$$

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

Proof. By induction over the structure of S_2. □

3.6 Occurs Check

Type unification algorithms use a well-known occurs check in order to avoid the generation of cyclic mappings in a substitution, like $[\alpha \mapsto \alpha \rightarrow \alpha]$. 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, $occurs(\alpha, \tau)$, that is inhabited[4] only if $\alpha \in \mathrm{FV}(\tau)$:

$$\begin{array}{lcl} occurs(\alpha, \tau_1 \rightarrow \tau_2) & = & occurs(\alpha, \tau_1) \vee occurs(\alpha, \tau_2) \\ occurs(\alpha, \alpha') & = & \alpha' \neq \alpha \\ occurs(\alpha, C) & = & \texttt{False} \end{array}$$

Coq type `False` is the and empty type[5], 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 α and all types τ, we have that either $occurs(\alpha, \tau)$ or $\neg occurs(\alpha, \tau)$ holds.*

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

[5] 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-correspondence, they correspond to a true and false propositions, respectively [21].

Proof. Induction over the structure of τ. □

If a variable α does not occur in a well-formed type, this type is well-formed in a variable context where α 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 α_1, α_2 and all variable contexts $\mathscr{V}$, if $\alpha_1 \in \mathscr{V}$ and $\alpha_2 \neq \alpha_1$ then $\alpha_1 \in (\mathscr{V} - \{\alpha_2\})$.*

Proof. Induction over $\mathscr{V}$. □

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

Proof. Induction on the structure of τ, using Lemma 5 in the variable case. □

3.7 Constraint Generation

The type inference algorithm for STLC is due to Hindley [27]. Essentially, the algorithm uses the fact that the type system for STLC is syntax-directed, i.e. the syntactic structure of a well-typed term uniquely determines which type rule should be applied to produce a valid typing derivation. Hindley's idea is that each term induces a *candidate type derivation*, where all missing types consist of distinct type variables. In order for the candidate derivation to be considered as a valid type derivation, all of its type variables must be instantiated to a ground type[6], subject to equality constraints.

Constraint generation is a simple recursive procedure that produces a list of constraints, from a given candidate typing derivation. An algorithm for generating constraints for STLC is presented next in Figure 3.

$$
\begin{array}{lcl}
\langle\langle \Gamma \vdash c : \tau \rangle\rangle & = & [\tau \overset{e}{=} C] \\
\langle\langle \Gamma \vdash x : \tau \rangle\rangle & = & [\Gamma(x) \overset{e}{=} \tau] \\
\langle\langle \Gamma \vdash \lambda x.e : \tau \rangle\rangle & = & \tau \overset{e}{=} \alpha_1 \rightarrow \alpha_2 :: \langle\langle \Gamma, x : \alpha_1 \vdash e : \alpha_2 \rangle\rangle \\
 & & \text{where } \alpha_1, \alpha_2 \text{ are fresh} \\
\langle\langle \Gamma \vdash e_1\, e_2 : \tau \rangle\rangle & = & \langle\langle \Gamma \vdash e_1 : \alpha \rightarrow \tau \rangle\rangle {+\!+} \langle\langle \Gamma \vdash e_2 : \alpha \rangle\rangle \\
 & & \text{where } \alpha \text{ is fresh}
\end{array}
$$

Figure 3. Constraint generation algorithm.

We implement constraint generation using a simple state monad to store a list of used type variables names (i.e. it stores the variable context $\mathscr{V}$) and a natural number that denotes the next fresh variable name to be used by the algorithm.

The next lemma shows that if we call constraint generation on a well-formed typing context for a given term, it produces a well-formed constraint. It it is crucial to establish type inference soundness.

Lemma 7. *Let e be an term, $\mathscr{V}$ a variable context and Γ a typing context such that $wf(\mathscr{V}, \Gamma)$ holds. If $\langle\langle \Gamma \vdash e : \alpha \rangle\rangle$ succeds then $wf(\mathscr{V}, \langle\langle \Gamma \vdash e : \alpha \rangle\rangle)$ holds.*

Proof. By structural induction on e. □

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 τ and τ':

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

Figure 4. Unification algorithm.

Our formalization differs from the presented algorithm (Figure 4) in two aspects:

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

2. 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 aspects 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 the Coq reference manual [28], 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.

[6] We say that a type τ is ground if $\mathrm{FV}(\tau) = \emptyset$.

4.1 Termination Proof

For any list of equalities the unification algorithm always terminates, 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 $|\mathbb{C}|$, 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 $(m,n) \prec (m',n')$ denote the usual lexicographic ordering of degrees.

Textbooks usually consider it straightforward 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 4, 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 8 (Substitution application decreases degree)**.** *For all variables $\alpha \in \mathscr{V}$, all well-formed types τ and well-formed lists of constraints $\mathbb{C}$, it holds that*

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

Proof. Induction over $\mathbb{C}$. □

On line 5 of Figure 4, 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 9 (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*

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

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

$$|\mathbb{C}| \prec |\tau \overset{e}{=} \tau' :: \mathbb{C}|$$

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 3.3.

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

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

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:

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

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

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

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"[7] 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 presented below. They specify properties of unification and application of substitutions.

Lemma 11. *For all type variables α, types τ, τ' and substitutions S, if $S(\alpha) = S(\tau')$ then $S(\tau) = S([\alpha \mapsto \tau']\,\tau)$.*

Proof. Induction over the structure of τ. □

Lemma 12. *For all type variables α, types τ, variable contexts $\mathscr{V}$ 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 11. □

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

[7] 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.

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 $\mathscr{L}$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 $\mathscr{L}$tac custom tactic used in our development is a proof state simplifier that performs several manipulations on the hypotheses and on the conclusion. 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 5, performs pattern matching on a proof state using $\mathscr{L}$tac `match goal` construct. Patterns have the form:

```
[ h1:t1, h2:t2 ... |- 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 $\mathscr{L}$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 5, 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
    (** some branches omitted for brevity *)
    | [ H : _ /\ _ |- _] => destruct H
    | [ H : _ \/ _ |- _] => destruct H
    | [ |- 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 : None = Some _ |- _] => congruence
    | [ |- _ /\ _] => split
  end.

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

Figure 5. Main proof state simplifier tactic.

Tactic `inverts* H` generates necessary conditions used to prove `H` and afterwards executes tactic `auto`.[8] Tactic `split` divides a conjunction goal in its constituent parts.

Besides $\mathscr{L}$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.[9] 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 5, `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 8, 9 e 10 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. Extracting a Type Inference Algorithm

6.1 Defining the Type Inference Algorithm

Our main goal in the formalization of a type unification algorithm is to build a certified type inference tool. The inference algorithm receives a term and returns its inferred type (using

[8] This tactic is defined on a tactic library developed by Arthur Charguéraud [29].

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

constructor |Some— of Coq standard library |option— type) or signals a failure, if no such type exists. The definition of the algorithm is presented below.

$$\begin{array}{lcl} \text{infer}\,\text{e} & = & t,\ \text{where:} \\ & & \begin{array}{lcl} \alpha & \text{is} & \text{fresh.} \\ \mathbb{C} & = & \langle\langle \emptyset \vdash e : \alpha \rangle\rangle \\ R & = & \mathit{unify}(\mathbb{C}) \\ t & = & \text{if}\, R = \texttt{Some}\ S\ \texttt{then}\ S(\alpha) \\ & & \quad \texttt{else}\ \text{fail} \end{array} \end{array}$$

Basically, the algorithm calls the constraint generator for input term using a fresh type variable α as its type and passing the produced constraint to unification algorithm. If unification succeeds, we apply the resulting substitution to α or signal a failure, otherwise.

Soundness and completeness of type inference is ensured by the following theorems.

Theorem 3 (Type inference soundness)**.** *For all terms e and types τ, if infer $e = \tau$ then $\emptyset \vdash e : \tau$.*

Proof. By structural induction on e using lemma 3.7 and theorem 3.4. □

Theorem 4 (Type inference completeness)**.** *For all terms e and types τ, if $\emptyset \vdash e : \tau$ then exists a substitution S' such that $S'(\tau') = \tau$, where $\tau' = infer(e)$.*

Proof. By structural induction on e using the completeness property of unification. □

6.2 Extracting a Certified Type Inference Algorithm

Now that we have defined a verified type inference algorithm for STLC, how can we use it? Our strategy is to replace code of a Haskell type checker for STLC tested using QuickCheck [30], by extracted Haskell code from our formalization. This idea of replacing code tested with QuickCheck by Coq extracted code isn't new and was already used to formalize core functionality of a window manager implemented in Haskell [24].

In order to test extracted code, we have implemented a type inference algorithm for STLC in Haskell and built a testsuite for it. We have implemented a generator for well-typed λ-terms and use QuickCheck to test the following property on both algorithms:

$$\forall e. \forall \tau. \emptyset \vdash e : \tau \rightarrow \text{infer(e)} = \tau$$

i.e. given a well-typed closed term e, the type inference algorithm returns it's correct type.

Since we have formalized type inference for STLC in Coq, such extracted code should be obviously correct. In principle, code Coq extraction preserves semantics if no extraction customization command is used [23]. Commands for customizing extraction behavior could introduce bugs and invalidate any extraction guarantees. Also, Coq extraction mechanism doesn't make use of any standard Haskell library type. Motivated by this, we've made some customizations trying to produce code that use as much of Haskell's library as possible. In order to validate the correctness of our customizations, we have submitted extracted code to our QuickCheck based testsuite and it passed in all tests for random generated terms.

For the complete type inference algorithm, our Haskell implementation is written in 142 lines of code, while Coq extracted code is written in around 300 lines. Such difference is justified as follows:

1. Coq extraction doesn't make use of any predefined Haskell type or type class. A possible fix is pointed by Swierstra [24], make use of axioms and sections to postulate the existence of an equality test and define some wrapper Haskell functions that calls extracted code with a proper |Eq—[10] dictionary. Such situation is even worse in our context, since our Haskell implementation makes use of monads and Haskell's |do—-notation for monadic programming. Since extraction isn't aware of Haskell's |Monad— type class and its associated syntactic sugar, code produced expose, in several places, a pattern-maching structure that could be hidden by the monadic "bind" function.

2. Most of the code used in the formalization to produce the termination argument for unification wasn't erased by program extraction. Coq code for unification takes an additional argument to ensure that all recursive calls are made on constraints with a smaller degree. We have defined several functions to manipulate variable contexts to represent that whenever we solve a type variable in unification, it should be removed from variable context, $\mathcal{V}$. Since, such arguments use functions that manipulate such non-propositional values (i.e. values whose type isn't in sort |Prop—), they aren't removed by extraction and it resulted in functions with unnecessary parameters in Haskell code for unification.

7. Related Work

Formalizations of unification algorithms: Formalization of unification algorithms has been the subject of several research works [8, 9, 10, 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

[10] Type class |Eq— denotes the set of all Haskell types that supports equality tests [1].

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 a specific implementation of the technique for general recursion based on well founded relations [31], 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 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 [32]. 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 [33]. 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 [34] 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. [35] 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.

Formalization of type inference algorithms: Most works on formalizing type inference algorithms focus on (or some variant of) Damas-Milner type system (c.f. [36, 37, 38, 39, 40, 41]).

The first works on formalizing type inference are by Nazareth and Narascewski in Isabelle/HOL [37, 38]. Both works focus on formalizing the well-known algorithm W [3], but unlike our work, they don't provide a verified implementation of unification. They assume all the necessary unification properties to finish their certification of type inference. The work of Dubois [36] also postulates unification and prove properties of type inference for ML using the Coq proof assistant system. Nominal techniques were used by Urban [39] to certify algorithm W in Isabelle/HOL using the Nominal package. As in other works, Urban just assumes properties that the unification algorithm should have without formalizing it.

Full formalizations of type inference for ML with structural polymorphism was reported by Jacques Garrigue [40, 41]. He fully formalizes interpreters for fragments of the OCaml programming language. Since the type system of OCaml is far more elaborate than STLC, his work involves a more substantial formalization effort than the one reported in this work. Garrigue's formalization of unification avoids the construction of a well-founded relation for constraints by defining the algorithm by using a "bound" on the number of allowed recursive calls made. Also, he uses libraries for dealing with bindings using the so-called locally nameless approach [42].

Applications of proof assistants. Ribeiro and Du Bois [43] described the formalization of a RE (regular expression) parsing algorithm that produces a bit representation of its parse tree in the dependently typed language Agda. The algorithm computes bit-codes using Brzozowski derivatives and they proved that the produced codes are equivalent to parse trees ensuring soundness and completeness with respect to an inductive RE semantics. They included the certified algorithm in a tool developed by themselves, named verigrep, for RE-based search in the style of GNU grep. While the authors provided formal proofs, their tool show a bad performance when compared to other approaches to RE parsing.

A formal constructive theory of RLs (regular language) was presented by Doczkal et. al. in [44]. They formalized some fundamental results about RLs. For their formalization, they used the Ssreflect extension to Coq, which features an extensive library with support for reasoning about finite structures such as finite types and finite graphs. They established all of their results in about 1400 lines of Coq, half of which are specifications. Most of their formalization deals with translations between different representations of RLs, including REs, DFAs (deterministic finite automata), minimal DFAs and NFAs (non-deterministic finite automata). They formalized all these (and other) representations and constructed computable conversions between them. Besides other interesting aspects of their work, they proved the decidability of language equivalence for all representations. Unlike our work, Doczkal et. al.'s only concerns about formalizing classical results of RL theory in Coq, without using the formalized automata in practical applications, like matching or parsing.

8. 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 formalized unification algorithm is used to produce a correct constraint-based type inference algorithm for STLC in Coq. We use such formalization to produce a Haskell implementation from it. Since Coq extraction doesn't make use of any Haskell's library types, we use some customization commands to produce more idiomatic Haskell code. To validate such customizations, we have used property based tests to ensure that the produced Haskell code behaves as expected.

The developed formalization has 962 lines of code and around 100 lines of comments. The formalization is composed by 47 lemas and theorems, 49 type and function definitions and 5 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 [20].

Author contributions

All authors have contributed equally to the development of this work.

References

[1] JONES, S. P. *Haskell 98 Language and Libraries: the Revised Report*. [S.l.: s.n.], 2003.

[2] MILNER, R.; TOFTE, M.; HARPER, R. *The Definition of Standard ML*. [S.l.]: MIT Press, 1990. I-XI, 1-101 p.

[3] MILNER, R. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, v. 17, n. 3, p. 348–375, 1978.

[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. cap. 10, p. 389–489. Disponível em: ⟨http://cristal.inria.fr/attapl/⟩.

[5] ROBINSON, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, v. 12, n. 1, p. 23–41, 1965.

[6] BERTOT, Y.; CASTÉRAN, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. [S.l.]: Springer Verlag, 2004. (Texts in Theoretical Computer Science).

[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*. Berlin, Heidelberg: Springer-Verlag, 2009. (TPHOLs '09), p. 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*. 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.*, v. 13, n. 6, p. 1061–1075, 2003.

[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*. [S.l.: s.n.], 2010. (EPTCS, v. 42), p. 24–38.

[12] MITCHELL, J. C. *Foundations of Programming Languages*. Cambridge, MA, USA: MIT Press, 1996.

[13] PIERCE, B. C. *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002.

[14] MCBRIDE, C.; MCKINNA, J. The view from the left. *J. Funct. Program.*, v. 14, n. 1, p. 69–111, 2004.

[15] LEROY, X. Formal verification of a realistic compiler. *Commun. ACM*, v. 52, n. 7, p. 107–115, 2009.

[16] BARTHE, G. et al. A Formal Correspondence between Offensive and Defensive JavaCard Virtual Machines. In: CORTESI, A. (Ed.). *VMCAI*. [S.l.]: Springer, 2002. (Lecture Notes in Computer Science, v. 2294), p. 32–45.

[17] GONTHIER, G. The Four Colour Theorem: Engineering of a Formal Proof. In: KAPUR, D. (Ed.). *ASCM*. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 5081), p. 333.

[18] GONTHIER, G. Engineering mathematics: the odd order theorem proof. In: GIACOBAZZI, R.; COUSOT, R. (Ed.). *POPL*. [S.l.]: ACM, 2013. p. 1–2.

[19] RIBEIRO, R.; CAMARÃO, C. A mechanized textbook proof of a type unification algorithm. In: *SBMF*. [S.l.]: Springer, 2015. p. 127–141.

[20] RIBEIRO, R. et al. *A Mechanized Textbook Proof of a Type Unification Algorithm — On-line repository*. 2015. Https://github.com/rodrigogribeiro/unification.

[21] SØRENSEN, M.; URZYCZYN, P. *Lectures on the Curry-Howard Isomorphism*. [S.l.]: Elsevier, 2006. (Studies in Logic and the Foundations of Mathematics, v. 10).

[22] CHLIPALA, A. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. Disponível em: ⟨http://mitpress.mit.edu/books/certified-programming-dependent-types⟩.

[23] LETOUZEY, P. A New Extraction for Coq. In: *Proceedings of the 2002 International Conference on Types for Proofs and Programs*. Berlin, Heidelberg: Springer-Verlag, 2003. (TYPES'02), p. 200–219. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=1762639.1762651⟩.

[24] SWIERSTRA, W. Xmonad in Coq (Experience Report): Programming a Window Manager in a Proof Assistant. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 47, n. 12, p. 131–136, set. 2012. Disponível em: ⟨http://doi.acm.org/10.1145/2430532.2364523⟩.

[25] AYDEMIR, B. E. et al. Engineering formal metatheory. In: NECULA, G. C.; WADLER, P. (Ed.). *POPL*. [S.l.]: ACM, 2008. p. 3–15.

[26] CHLIPALA, A. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 43, n. 9, p. 143–156, set. 2008. Disponível em: ⟨http://doi.acm.org/10.1145/1411203.1411226⟩.

[27] HINDLEY, J. R.; SELDIN, J. P. *Lambda-Calculus and Combinators: An Introduction*. 2. ed. New York, NY, USA: Cambridge University Press, 2008.

[28] Coq Developement Team. *Coq Proof Assistant — Reference Manual*. 2014. Disponível em: ⟨http://coq.inria.fr/distrib/current/refman/⟩.

[29] PIERCE, B. C. et al. *Software Foundations*. [S.l.]: Electronic textbook, 2015.

[30] CLAESSEN, K.; HUGHES, J. Quickcheck: a lightweight tool for random testing of haskell programs. In: *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2000. p. 268–279. Disponível em: ⟨http://dx.doi.org/10.1145/351240.351266⟩.

[31] NORDSTRÖM, B. Terminating General Recursion. *BIT Numerical Mathematics*, v. 28, n. 3, p. 605–619, 1988.

[32] MCBRIDE, C. *First-order unification by structural recursion — Correctness proof.* Disponível em: ⟨http://strictlypositive.org/foubsr-website/proof.ps⟩.

[33] NARASCHEWSKI, W.; NIPKOW, T. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 23, n. 3, p. 299–318, nov. 1999. Disponível em: ⟨http://dx.doi.org/10.1023/A:1006277616879⟩.

[34] AVELAR, A. B. et al. Verification of the Completeness of Unification Algorithms à la Robinson. In: DAWAR, A.; QUEIROZ, R. J. G. B. de (Ed.). *Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010, Brasilia, Brazil, July 6-9, 2010. Proceedings*. Springer, 2010. (Lecture Notes in Computer Science, v. 6188), p. 110–124. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-13824-9_10⟩.

[35] AVELAR, A. B. et al. First-order unification in the PVS proof assistant. *Logic Journal of the IGPL*, v. 22, n. 5, p. 758–789, 2014. Disponível em: ⟨http://dx.doi.org/10.1093/jigpal/jzu012⟩.

[36] DUBOIS, C.; MÉNISSIER-MORAIN, V. Certification of a Type Inference Tool for ML: Damas-Milner within Coq. *J. Autom. Reasoning*, v. 23, n. 3-4, p. 319–346, 1999. Disponível em: ⟨http://dx.doi.org/10.1023/A:1006285817788⟩.

[37] NARASCHEWSKI, W.; NIPKOW, T. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, v. 23, p. 299–318, 1999.

[38] NAZARETH, D.; NIPKOW, T. Formal Verification of Algorithm W: The Monomorphic Case. In: WRIGHT, J. von; GRUNDY, J.; HARRISON, J. (Ed.). *Theorem Proving in Higher Order Logics (TPHOLs'96)*. Springer, 1996. (LNCS, v. 1125), p. 331–346. Disponível em: ⟨http://www4.informatik.tu-muenchen.de/~nipkow/pubs/tphol96.dvi.gz⟩.

[39] URBAN, C.; NIPKOW, T. Nominal verification of algorithm W. In: HUET, G.; LéVY, J.-J.; PLOTKIN, G. (Ed.). *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*. [S.l.]: Cambridge University Press, 2009. p. 363–382.

[40] GARRIGUE, J. A certified implementation of ML with structural polymorphism. In: UEDA, K. (Ed.). *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*. Springer, 2010. (Lecture Notes in Computer Science, v. 6461), p. 360–375. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-17164-2_25⟩.

[41] GARRIGUE, J. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, v. 25, n. 4, p. 867–891, 2015. Disponível em: ⟨http://dx.doi.org/10.1017/S0960129513000066⟩.

[42] CHARGUéRAUD, A. The Locally Nameless Representation. *J. Autom. Reasoning*, v. 49, n. 3, p. 363–408, 2012. Disponível em: ⟨http://dblp.uni-trier.de/db/journals/jar/jar49.html#Chargueraud12⟩.

[43] RIBEIRO, R.; BOIS, A. D. Certified Bit-Coded Regular Expression Parsing. *Proceedings of the 21st Brazilian Symposium on Programming Languages - SBLP 2017*, p. 1–8, 2017. Disponível em: ⟨http://dl.acm.org/citation.cfm?doid=3125374.3125381⟩.

[44] DOCZKAL, C.; KAISER, J. O.; SMOLKA, G. A constructive theory of regular languages in Coq. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 8307 LNCS, p. 82–97, 2013.