# A Mechanized Textbook Proof of a Type Unification Algorithm

Rodrigo Ribeiro [1] Carlos Camarão [2]

[1]Departamento de Computação e Sistemas - UFOP

[2]Departamento de Ciência da Computação - UFMG

XVIII Brazilian Symposium on Formal Methods, 2015

# Introduction

- Type inference is an important mechanism of modern functional languages, like Haskell and ML
- Type inference algorithms divided in
  - Constraint generation
  - Constraint solving
- Constraint solving for parametric polymorphism: **First order unification**

# Introduction

- Soundness: Computed substiution is a unifier.
- Completeness: Every unifier can be obtained as $S \circ S_c$, for some $S$, where $S_c$ is the computed substitution.
- Simple algorithms contained in textbooks, e.g:
  - Types and Programming Languages, Benjamin Pierce, The MIT Press, 2002.
  - Foundations for Programming Languages, John Mitchell, The MIT Press, 1996.

# Motivation

- Build a sound, complete and "axiom-free" formalization of unification, following textbooks presentations.
- First step toward a complete formalization of type inference algorithm for Haskell.

# Coq Proof Assistant

- Formalization developed using Coq version 8.4.
- Why Coq?
  - Mature tool used in several large scale formalizations: e.g. C compiler, Java Card plataform and mathematical theorems.
- Code avaliable at:

  `https://github.com/rodrigogribeiro/unification`

# Coq Proof Assistant

- Proof checking consists of type checking
- Provides **tactics** to ease proof construction.
- Has built-in DSL for building tactics: $\mathcal{L}$tac

# Coq Proof Assistant

- Sample theorem — tactic based version

```
Variables A B C : Prop.

Theorem example : (A → B) → (B → C) → A → C.
Proof.
    intros H H' HA.
    apply H'.
    apply H.
    assumption.
Qed.
```

# Coq Proof Assistant

- Sample theorem — term based version

```
Definition example' : (A → B) → (B → C) → A → C :=
    fun (H : A → B) (H' : B → C) (HA : A) ⇒ H' (H HA).
```

- We'll use a more familiar notation (not Coq) for definitions of types and functions

# Definitions

- We consider that terms are types, formed by type variables ($\alpha$), type constructors ($c$) and arrows $\rightarrow$

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

- Kinding information needed to model Haskell types, but:
  - The use of kinds is orthogonal to unification
  - Kinds are omitted for clarity
  - Handling kinds is straightforward

# Definitions

- $FV(\tau)$: free type variables from $\tau$
- $\tau_1 \stackrel{e}{=} \tau_2$: equality constraint
- Meta-variable $\mathbb{C}$ denotes a list of (equality) constraints
- Size of a type.

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

# Definitions

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

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

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

# Definitions

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

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

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

# Substitutions

- Finite functions from type variables to types.
- Metavariable $S$ denotes substitutions and *id* denotes the identity substitution.
- Represented as finite mappings: $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$

# Applying a Mapping

$$
\begin{array}{lcl}
[\alpha \mapsto \tau'] \, \tau_1 \to \tau_2 & = & \tau_1' \to \tau_2' \\
 & & \text{where: } \left\{ \begin{array}{l} \tau_1' = [\alpha \mapsto \tau']\tau_1 \\ \tau_2' = [\alpha \mapsto \tau']\tau_2 \end{array} \right. \\
[\alpha \mapsto \tau'] \, \alpha & = & \tau' \\
[\alpha \mapsto \tau'] \, \tau & = & \tau
\end{array}
$$

# Substitution Application

- Defined in a variable-by-variable way by recursion on the applied substitution.

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

# Extensionality Lemma

- Used to state completeness of unification.
- Not necessary if we allow ourselves to postulate function extensionality.

**Lemma:** 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.

# Well-Formedness Conditions

- Conditions imposed on types, constraints and substitutions to give simple proofs of termination, soundness and completeness.
- During the execution of *unify* the variable context (a set of variables) is used to hold the complement of the unifier domain.

# Well-Formedness Conditions

- Type $\tau$ is well-formed in a variable context $\mathcal{V}$, written as $wf(\mathcal{V}, \tau)$, if all type variables that occur in $\tau$ are in $\mathcal{V}$.
- A constraint $\tau_1 \stackrel{e}{=} \tau_2$ is well-formed, written as $wf(\mathcal{V}, \tau_1 \stackrel{e}{=} \tau_2)$, if both $\tau_1$ and $\tau_2$ are well-formed in $\mathcal{V}$.

# Well-Formedness Conditions

- A list of constraints $\mathbb{C}$ is well-formed in $\mathcal{V}$, written as $wf(\mathcal{V}, \mathbb{C})$, if all of its equality constraints are well-formed in $\mathcal{V}$.

# Well-Formedness Conditions

- A substitution $S = \{[\alpha \mapsto \tau]\} :: S'$ is well-formed in $\mathcal{V}$, written as $wf(\mathcal{V}, S)$, if the following conditions apply:
  - $\alpha \in \mathcal{V}$
  - $wf(\mathcal{V} - \{\alpha\}, \tau)$
  - $wf(\mathcal{V} - \{\alpha\}, S')$

# Substitution Composition

- Let $S_1$ be a substitution such that $wf(\mathcal{V}, S_1)$;
- Let $S_2$ a substitution such that $wf(\mathcal{V} - dom(S_1), S_2)$.
- We can define composition as:

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

# Substitution Composition

**Theorem:** For all types $\tau$ and all substitutions $S_1$, $S_2$ such that $wf(\mathcal{V}, S_1)$ and $wf(\mathcal{V} - dom(S_1), S_2)$ we have that $(S_2 \circ S_1)(\tau) = S_2(S_1(\tau))$.
**Proof:** By induction over the structure of $S_2$.

# Occurs Check

- Avoids the generation of cyclic mappings like $[\alpha \mapsto \alpha \to \alpha]$.
- $occurs(\alpha, \tau)$ is inhabited iff $\alpha \in \mathrm{FV}(\tau)$:

$$
\begin{array}{lcl}
occurs(\alpha, \tau_1 \to \tau_2) & = & occurs(\alpha, \tau_1) \vee occurs(\alpha, \tau_2) \\
occurs(\alpha, \alpha) & = & \texttt{True} \\
occurs(\alpha, \tau) & = & \texttt{False} \text{ otherwise}
\end{array}
$$

# Occurs Check

- Occurs check is crucial to prove termination of unification.
- Next lemma is important to establish a relation between application of substitution and the occurs check.

**Lemma:** Let $\tau$ be s.t. $wf(\mathcal{V}, \tau)$ and $\neg occurs(\alpha, \tau)$.
Then $wf(\mathcal{V} - \{\alpha\}, \tau)$.
**Proof:** Induction over the structure of $\tau$.

# Unification Algorithm

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

Coq's termination checker rejects calls in red.

# Termination

- Termination argument based on the notion of *degree* $(n, m)$ of $\mathbb{C}$.
  - $n$: number of type variables in $\mathbb{C}$
  - $m$: total size of types in $\mathbb{C}$.
- Termination argument based on lexicographic ordering of pairs.

# Termination

- The next lemma is used to convince Coq that the following call decreases input $\mathbb{C}$:

$$\mathit{unify}([\alpha \mapsto \tau]\mathbb{C})$$

**Lemma:** For all $\alpha \in \mathcal{V}$, all well-formed types $\tau$ and well-formed lists of constraints $\mathbb{C}$, it holds that

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

# Termination

- The next lemma is used to convince Coq that the following call decreases input $\mathbb{C}$:

$$unify((\tau_1 \stackrel{e}{=} \tau) :: (\tau_2 \stackrel{e}{=} \tau') :: \mathbb{C})$$

**Lemma:** For all well-formed $\tau_1, \tau_2, \tau'_1, \tau'_2$ and all well-formed $\mathbb{C}$,

$$degree((\tau_1 \stackrel{e}{=} \tau'_1, \tau_2 \stackrel{e}{=} \tau'_2) :: \mathbb{C}) \prec degree((\tau_1 \rightarrow \tau_2 \stackrel{e}{=} \tau'_1 \rightarrow \tau'_2) :: \mathbb{C})$$

# Soundness and Completeness

- Unification either fails or returns a substitution that is the *least unifier* for a constraint $\mathbb{C}$.
- A substitution $S$ is a unifier iff $unifier(\mathbb{C}, S)$ is provable

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

# Soundness and Completeness

- Substitution ordering

$$S \leq S' \stackrel{def}{=} \exists S_1.\forall\alpha.\, S'(\alpha) = S_1 \circ S(\alpha)$$

- Least unifier definition

$$least(S, \mathbb{C}) \;=\; \forall S'.\, unifier(\mathbb{C}, S') \rightarrow S \leq S'$$

# Soundness and Completeness

- Type of unification algorithm:

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

- $\mathit{UnifyFailure}(\mathbb{C})$: type that explain the reason of failure of unification of $\mathbb{C}$.

# Soundness and Completeness

- Proofs of soundness and completenes tied with algorithm definition.
  - “Holes” mark positions where proof terms are expected.
  - Proof obligations generated by holes filled by custom $\mathcal{L}$tac scripts

# Automating Proofs

- Proof automation is crucial to scale Coq formalizations.
- $\mathcal{L}$tac scripts fill all proof obligations for termination, soundness and completeness.
- Main tools used for automating proofs:
  - Custom $\mathcal{L}$tac scripts for proof state simplification.
  - Use of auto tactic with hint databases.

# Conclusion

- Complete formalization of unification in Coq.
- Development statistics:
  - 31 lemmas and theorems
  - 34 type and function definitions
  - Total: 610 lines (94 lines of comments)
- Implemention effort on termination: 293 lines (21 theorems).