A Machine-Checked Model of MGU Axioms: Applications of Finite Maps and Functional Induction

Presented by Sunil Kothari
Joint work with Prof. James Caldwell

Department of Computer Science,
University of Wyoming, USA

23rd International Workshop on Unification
August 2, 2009
Outline

1. Overview
   - Type Reconstruction Algorithms

2. Introduction
   - Substitution
   - Coq

3. First-order unification algorithm
   - Specification in Coq
   - Termination

4. A model for MGU axioms
   - Axiom iii
   - Axiom iv

5. Conclusions and Future Work
Overview
- Type Reconstruction Algorithms

Introduction
- Substitution
- Coq

First-order unification algorithm
- Specification in Coq
- Termination

A model for MGU axioms
- Axiom iii
- Axiom iv

Conclusions and Future Work
Essential feature of many functional programming languages (ML, Haskell, OCaml, etc.).
Highlights

- Essential feature of many functional programming languages (ML, Haskell, OCaml, etc.).
- Automated type reconstruction is possible.
Highlights

- Essential feature of many functional programming languages (ML, Haskell, OCaml, etc.).
- Automated type reconstruction is possible.
  - Substitution-based algorithms.
    - Intermittent constraint generation and constraint solving.
Essential feature of many functional programming languages (ML, Haskell, OCaml, etc.).

Automated type reconstruction is possible.
  - Substitution-based algorithms.
    - Intermittent constraint generation and constraint solving.
  - Constraint-based algorithms.
    - Two distinct phases: constraint generation and constraint solving.
Substitution-based

Substitution-based

Constraint-based Frameworks/Algorithms
- Wand’s algorithm [Wan87].
- Qualified types [Jon95].
- HM(X) [SOW97] by Sulzmann et al. 1999, Pottier and Rémy 2005 [PR05].
- Top quality error messages [Hee05].
Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99a, NN99b, NN96].
Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99a, NN99b, NN96].
- Nominal verification of Algorithm W [UN09].
Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99a, NN99b, NN96].
- Nominal verification of Algorithm W [UN09].
- We want to formalize multi-phase unification algorithm needed to handle polymorphic let.
Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99a, NN99b, NN96].
- Nominal verification of Algorithm W [UN09].
- We want to formalize multi-phase unification algorithm needed to handle polymorphic let.
- POPLMark challenge also aims at mechanizing meta-theory.
Overview

Type Reconstruction Algorithms

Type Reconstruction Algorithms... Contd

Modeling MGU

- The *most general unifier* (MGU) is often a first-order unification algorithm over simple type terms.
Modeling MGU

The *most general unifier* (MGU) is often a first-order unification algorithm over simple type terms.

In machine checked correctness proofs, the MGU is modeled as a set of four axioms:

1. \( \text{mgu} \sigma (\tau_1 \overset{c}{=} \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \)
2. \( \text{mgu} \sigma (\tau_1 \overset{c}{=} \tau_2) \land \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \sigma''. \sigma' \approx \sigma \circ \sigma'' \)
3. \( \text{mgu} \sigma (\tau_1 \overset{c}{=} \tau_2) \Rightarrow \text{FTVS} (\sigma) \subseteq \text{FVC} (\tau_1 \overset{c}{=} \tau_2) \)
4. \( \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'. \text{mgu} \sigma'(\tau_1 \overset{c}{=} \tau_2) \)
Outline

1. Overview
   - Type Reconstruction Algorithms

2. Introduction
   - Substitution
   - Coq

3. First-order unification algorithm
   - Specification in Coq
   - Termination

4. A model for MGU axioms
   - Axiom iii
   - Axiom iv

5. Conclusions and Future Work
Terms and Constraint Syntax

Terms

\[ \tau ::= \text{TyVar}(x) \mid \tau' \rightarrow \tau'' \]
Terms and Constraint Syntax

Terms

- $\tau ::= \text{TyVar}(x) \mid \tau' \rightarrow \tau''$
- Atomic types (of the form TyVar $x$) are denoted by $\alpha, \beta, \alpha'$ etc.
Terms and Constraint Syntax

Terms

- $\tau ::= \text{TyVar}(x) \mid \tau' \to \tau''$
- Atomic types (of the form $\text{TyVar} \ x$) are denoted by $\alpha, \beta, \alpha'$ etc.

Constraints

- Constraint are of the form $\tau \stackrel{c}{=} \tau'$.
Terms and Constraint Syntax

Terms

- \( \tau ::= \text{TyVar}(x) \mid \tau' \rightarrow \tau'' \)
- Atomic types (of the form TyVar \( x \)) are denoted by \( \alpha, \beta, \alpha' \) etc.

Constraints

- Constraint are of the form \( \tau^c = \tau' \).
- A list of constraint is given as:
  - \( \mathbb{C} ::= [] \mid \tau^c = \tau' :: \mathbb{C}' \)
Free type variable (FTV)

\[
\begin{align*}
\text{FTV (TyVar } x) & \overset{def}{=} [x] \\
\text{FTV } (\tau \to \tau') & \overset{def}{=} \text{FTV } (\tau) ++ \text{FTV } (\tau')
\end{align*}
\]
FTV and FVC

Free type variable (FTV)

\[
\begin{align*}
\text{FTV (TyVar } x \text{)} & \overset{\text{def}}{=} [x] \\
\text{FTV (} \tau \rightarrow \tau' \text{)} & \overset{\text{def}}{=} \text{FTV (} \tau \text{) ++ FTV (} \tau' \text{)}
\end{align*}
\]

Free variables of a constraint list (FVC)

\[
\begin{align*}
\text{FVC []} & \overset{\text{def}}{=} [] \\
\text{FVC (}(\tau_1 \overset{c}{=} \tau_2) :: C) & \overset{\text{def}}{=} \text{FTV (} \tau_1 \text{) ++ FTV (} \tau_2 \text{) ++ FVC (} C \text{)}
\end{align*}
\]
Related Concepts

- A *substitution* (denoted by $\rho$) maps type variables to types.
Related Concepts

- A substitution (denoted by $\rho$) maps type variables to types.
- Denoted by $\sigma, \sigma', \sigma_1$ etc.
Related Concepts

- A substitution (denoted by $\rho$) maps type variables to types.
- Denoted by $\sigma, \sigma', \sigma_1$ etc.
- Substitution application to a type $\tau$ is defined as:

$$\sigma\ (\text{TyVar}(x)) \overset{\text{def}}{=} \text{if } \langle x, \tau \rangle \in \sigma \text{ then } \tau \text{ else } \text{TyVar}(x)$$

$$\sigma\ (\tau_1 \rightarrow \tau_2) \overset{\text{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2)$$
A substitution (denoted by \( \rho \)) maps type variables to types.

- Denoted by \( \sigma, \sigma', \sigma_1 \) etc.
- Substitution application to a type \( \tau \) is defined as:

\[
\sigma \left( \text{TyVar}(x) \right) \overset{\text{def}}{=} \text{if } \langle x, \tau \rangle \in \sigma \text{ then } \tau \text{ else } \text{TyVar}(x)
\]

\[
\sigma \left( \tau_1 \rightarrow \tau_2 \right) \overset{\text{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2)
\]

Application of a substitution to a constraint is defined similarly:

\[
\sigma(\tau_1 \overset{c}{=} \tau_2) \overset{\text{def}}{=} \sigma(\tau_1) \overset{c}{=} \sigma(\tau_2)
\]
Substitution Composition

- Substitution composition definition using Coq’s finite maps is complicated.
- But the following theorem holds

**Theorem 1 (Composition apply)**

\[ \forall \sigma, \sigma'. \forall \tau. (\sigma \circ \sigma') \tau = \sigma' (\sigma (\tau)) \]
Substitution Composition

- Substitution composition definition using Coq’s finite maps is complicated.
- But the following theorem holds

**Theorem 1 (Composition apply)**

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

Substitution Composition

- Substitution composition definition using Coq’s finite maps is complicated.
- But the following theorem holds

Theorem 1 (Composition apply)

\[ \forall \sigma, \sigma'. \forall \tau. (\sigma \circ \sigma') \tau = \sigma' (\sigma (\tau)) \]

Extensional equality

- Substitutions are equal if they behave the same on all type variables.

\[ \sigma \approx \sigma' \overset{\text{def}}{=} \forall \alpha. \sigma (\alpha) = \sigma' (\alpha) \]
Unifiers and MGUs

Unifier

- We write $\sigma \models (\tau_1 \overset{c}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.
- $\sigma \models C \overset{def}{=} \forall c \in C, \sigma \models c$. 
Unifiers and MGUs

Unifier

- We write $\sigma \models (\tau_1 =_c \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.
- $\sigma \models C \triangleq \forall c \in C, \sigma \models c$.

Most General Unifier

- A unifier $\sigma$ is the most general unifier (MGU) if for any other unifier $\sigma''$ there is a substitution $\sigma'$ such that $\sigma \circ \sigma' \approx \sigma''$. 
Overview

Based on the Calculus of Constructions.
Overview

- Based on the Calculus of Constructions.
- System F extended with dependent types.
Overview

- Based on the Calculus of Constructions.
- System F extended with dependent types.
- Support for inductive datatypes.
Overview

- Based on the Calculus of Constructions.
- System F extended with dependent types.
- Support for inductive datatypes.
- Programs can be extracted from proofs.
Overview

- Based on the Calculus of Constructions.
- System F extended with dependent types.
- Support for inductive datatypes.
- Programs can be extracted from proofs.
- Lots of libraries.
Representing substitutions

- Substitution represented as a list of pairs, set of pairs, and normal function.
- We represent a substitution as a finite function.
Finite maps in Coq

Representing substitutions
- Substitution represented as a list of pairs, set of pairs, and normal function.
- We represent a substitution as a finite function.

Substitution as finite map
- Used the Coq’s finite maps library `Coq.FSets.FMapInterface`.
- Axiomatic presentation.
- Provides no induction principle.
- Forward reasoning is often required.
Domain

\[ \text{dom\_subst}(\sigma) \overset{\text{def}}{=} \text{List.map}(\lambda t. \text{fst}(t)) (\text{M.elements}(\sigma)) \]
Substitution Related Concepts in Coq

**Domain**

```
\text{dom\_subst}(\sigma) \overset{\text{def}}{=} \text{List.map}(\lambda t. \text{fst}(t)) (\text{M.elements}(\sigma))
```

**Range**

```
\text{range\_subst}(\sigma) \overset{\text{def}}{=} \text{List.flat\_map}(\lambda t. \text{FTV}(\text{snd}(t))) (\text{M.elements}(\sigma))
```
Substitution Related Concepts in Coq

Domain

\[ \text{dom\_subst}(\sigma) \overset{\text{def}}{=} \text{List.map} (\lambda t. \text{fst}(t)) (M.\text{elements}(\sigma)) \]

Range

\[ \text{range\_subst}(\sigma) \overset{\text{def}}{=} \text{List.flat\_map} (\lambda t. \text{FTV}(\text{snd}(t))) (M.\text{elements}(\sigma)) \]

FTVS

\[ \text{FTVS}(\sigma) \overset{\text{def}}{=} \text{dom\_subst}(\sigma) ++ \text{range\_subst}(\sigma) \]
Outline

1 Overview
   - Type Reconstruction Algorithms

2 Introduction
   - Substitution
   - Coq

3 First-order unification algorithm
   - Specification in Coq
   - Termination

4 A model for MGU axioms
   - Axiom iii
   - Axiom iv

5 Conclusions and Future Work
Unification

The Algorithm

\[
\begin{align*}
\text{unify } (\alpha \equiv \alpha) &:: C \quad \overset{\text{def}}{=} \text{unify } C \\
\text{unify } (\alpha \equiv \beta) &:: C \quad \overset{\text{def}}{=} \{\alpha \mapsto \beta\} \circ \text{unify } (\{\alpha \mapsto \beta\} C) \\
\text{unify } (\alpha \equiv \tau) &:: C \quad \overset{\text{def}}{=} \begin{cases} 
\text{Fail} & \text{if } \alpha \text{ occurs in } \tau \\
\{\alpha \mapsto \tau\} \circ \text{unify } (\{\alpha \mapsto \tau\} C) & \text{else}
\end{cases} \\
\text{unify } (\tau \equiv \alpha) &:: C \quad \overset{\text{def}}{=} \text{unify } (\alpha \equiv \tau) :: C \\
\text{unify } (\tau_1 \rightarrow \tau_2) &:: C \\
\overset{\text{def}}{=} \text{unify } (\tau_1 \equiv \tau_3 :: \tau_2 \equiv \tau_4 :: C) \\
\text{unify } [] &:: C \\
\overset{\text{def}}{=} \text{Id}
\end{align*}
\]
Unification

The Algorithm

\[
\begin{align*}
\text{unify } (\alpha \equiv \alpha) &:: \text{C} \quad \overset{\text{def}}{=} \text{unify } \text{C} \\
\text{unify } (\alpha \equiv \beta) &:: \text{C} \quad \overset{\text{def}}{=} \{\alpha \mapsto \beta\} \circ \text{unify } (\{\alpha \mapsto \beta\}\text{C}) \\
\text{unify } (\alpha \equiv \tau) &:: \text{C} \quad \overset{\text{def}}{=} \begin{cases} 
\text{if } \alpha \text{ occurs in } \tau \\
\text{then Fail} \\
\text{else } \{\alpha \mapsto \tau\} \circ \text{unify } (\{\alpha \mapsto \tau\}\text{C}) 
\end{cases} \\
\text{unify } (\tau \equiv \alpha) &:: \text{C} \quad \overset{\text{def}}{=} \begin{cases} 
\text{if } \alpha \text{ occurs in } \tau \\
\text{then Fail} \\
\text{else } \{\alpha \mapsto \tau\} \circ \text{unify } (\{\alpha \mapsto \tau\}\text{C}) 
\end{cases} \\
\text{unify } (\tau_1 \rightarrow \tau_2) &:: \text{C} \quad \overset{\text{def}}{=} \text{unify } (\tau_1 \equiv \tau_3 :: \tau_2 \equiv \tau_4 :: \text{C}) \\
\text{unify } [] &:: \text{C} \quad \overset{\text{def}}{=} \text{Id}
\end{align*}
\]
Function unify (c:list constr)\{wf meaPairMLt\} : (option (M.t type)) :=
match c with
  | nil => Some (M.empty type)
  | h::t => (match h with
    | EqCons (TyVar x) (TyVar y) =>
      if eq_dec_stamp x y
      then unify t
      else (match unify (apply_subst_to_constr_list
          (M.add x (TyVar y)
           (M.empty type)) t) with
         Some p => Some (compose_subst
           (M.add x (TyVar y)
            (M.empty type)) p)
         | None => None
      end)
    | EqCons (TyVar x) (Arrow ty3 ty4) =>
      if (member x (FTV ty3)) || (member x (FTV ty4))
      then None
      else (match (unify (apply_subst_to_constr_list
          (M.add x (Arrow ty3 ty4)
           (M.empty type)) t)) with
        Some p => Some (compose_subst
           (M.add x (Arrow ty3 ty4)
            (M.empty type)) p)
        | None => None
      end)
    | EqCons (Arrow ty3 ty4)(TyVar x) =>
      if (member x (FTV ty3)) || (member x (FTV ty4))
      then None
      else (match (unify (apply_subst_to_constr_list
          (M.add x (Arrow ty3 ty4)
           (M.empty type)) t)) with
        Some p => Some (compose_subst
           (M.add x (Arrow ty3 ty4)
            (M.empty type)) p)
        | None => None
      end)
    | EqCons (Arrow ty3 ty4)(Arrow ty5 ty6) =>
      unify ((EqCons ty3 ty5)::((EqCons ty4 ty6)::t))
  end)
First-order unification algorithm
Specification in Coq

First-order unification in Coq

Issues in formalization

- Raise exceptions, but that’s not possible.
- We choose an option type defined as:

  \[
  \text{Inductive option } (A : \text{Set}) : \text{Set} := \text{Some } (_ : A) \mid \text{None}.
  \]

- Our specification of unification is general recursive – so Coq will require a termination criteria.
  - Give a measure that reduces on each recursive call.
  - Give a well-founded ordering, and ...
    - Show that recursive call is lower in order w.r.t the above order (bunched together as proof obligations).
    - Show that the ordering is well-founded.
  - Others ....
Lexicographic Ordering

- The lexicographic ordering ($≺_3$) on the two triples $\langle n_1, n_2, n_3 \rangle$ and $\langle m_1, m_2, m_3 \rangle$ is defined as

$\langle n_1, n_2, n_3 \rangle ≺_3 \langle m_1, m_2, m_3 \rangle \overset{\text{def}}{=} (n_1 < m_1) \lor (n_1 = m_1 \land n_2 < m_2) \lor (n_1 = m_1 \land n_2 = m_2 \land n_3 < m_3)$,

where $<$ and $=$ are the ordinary less-than inequality and equality on natural numbers.

The Triple

- The triple is $\langle |C_{FVC}|, |C_{\rightarrow}|, |C| \rangle$, where
  - $|C_{FVC}|$ - the number of unique free variables in a constraint list;
  - $|C_{\rightarrow}|$ - the total number of arrows in the constraint list;
  - $|C|$ - the length of the constraint list.
### Table: Variation of termination measure components on the recursive call

<table>
<thead>
<tr>
<th>Original call</th>
<th>Recursive call</th>
<th>Conditions, if any</th>
<th>$C_{FVC}$</th>
<th>$C_{→}$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\alpha ≡ α) :: C$</td>
<td>$C$</td>
<td>$\alpha \in (FVC \ C)$</td>
<td>-</td>
<td>-</td>
<td>↓</td>
</tr>
<tr>
<td>$(\alpha ≡ α) :: C$</td>
<td>$C$</td>
<td>$\alpha \notin (FVC \ C)$</td>
<td>$\downarrow$</td>
<td>-</td>
<td>↓</td>
</tr>
<tr>
<td>$(\alpha ≡ \beta) :: C$</td>
<td>${\alpha \mapsto \beta}C$</td>
<td>$\alpha \neq \beta$</td>
<td>$\downarrow$</td>
<td>-</td>
<td>↓</td>
</tr>
<tr>
<td>$(\alpha ≡ \tau) :: C$</td>
<td>${\alpha \mapsto \tau}C$</td>
<td>$\alpha \notin (FTV \ \tau) \land \alpha \notin (FVC \ C)$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>↓</td>
</tr>
<tr>
<td>$(\alpha ≡ \tau) :: C$</td>
<td>${\alpha \mapsto \tau}C$</td>
<td>$\alpha \notin (FTV \ \tau) \land \alpha \in (FVC \ C)$</td>
<td>$\downarrow$</td>
<td>$\uparrow$</td>
<td>↓</td>
</tr>
<tr>
<td>$(\tau \equiv \alpha) :: C$</td>
<td>${\alpha \mapsto \tau}C$</td>
<td>$\alpha \notin (FTV \ \tau) \land \alpha \notin (FVC \ C)$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>↓</td>
</tr>
<tr>
<td>$(\tau \equiv \alpha) :: C$</td>
<td>${\alpha \mapsto \tau}C$</td>
<td>$\alpha \notin (FTV \ \tau) \land \alpha \in (FVC \ C)$</td>
<td>$\downarrow$</td>
<td>$\uparrow$</td>
<td>↓</td>
</tr>
<tr>
<td>$(\tau_1 \rightarrow \tau_2$</td>
<td>$\equiv \tau_3 \rightarrow \tau_4) :: C$ : $(\tau_1 \equiv \tau_3) :: C$</td>
<td>$\tau_1 \equiv \tau_3$</td>
<td>None</td>
<td>-</td>
<td>↓</td>
</tr>
</tbody>
</table>
Outline

1 Overview
   - Type Reconstruction Algorithms

2 Introduction
   - Substitution
   - Coq

3 First-order unification algorithm
   - Specification in Coq
   - Termination

4 A model for MGU axioms
   - Axiom iii
   - Axiom iv

5 Conclusions and Future Work
Functional Induction in Coq

- Requires an induction principle generated before.
Functional Induction in Coq

- Requires an induction principle generated before.
- \text{functional induction} \ (f \ x_1 \ x_2 \ x_3 \ldots \ x_n) \ \text{is a short form for} \ \text{induction} \ x_1 \ x_2 \ x_3 \ldots x_n \ f(x_1 \ldots x_n) \ \text{using} \ \text{id}, \ \text{where} \ \text{id} \ \text{is the induction principle for} \ f.
Functional Induction in Coq

- Requires an induction principle generated before.
- \[\text{functional induction } (f \ x_1 \ x_2 \ x_3 \ldots \ x_n) \text{ is a short form for } \text{induction } x_1 \ x_2 \ x_3 \ldots x_n \ f(x_1 \ldots x_n) \text{ using } \text{id}, \text{ where } \text{id} \text{ is the induction principle for } f.\]
  - \[\text{functional induction } (\text{unify } c) \rightarrow \text{induction } c \ (\text{unify } c) \text{ using unif_ind.}\]
- Important first step in proof of the axioms.
Old Axioms

(i) \( \text{mgu } \sigma (\tau_1 \equiv \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \)

(ii) \( \text{mgu } \sigma (\tau_1 \equiv \tau_2) \land \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \delta. \sigma' \approx \sigma \circ \delta \)

(iii) \( \text{mgu } \sigma (\tau_1 \equiv \tau_2) \Rightarrow \text{FTVS } (\sigma) \subseteq \text{FVC } (\tau_1 \equiv \tau_2) \)

(iv) \( \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'. \text{mgu } \sigma'(\tau_1 \equiv \tau_2) \)
MGU axioms

Old Axioms

(i) \( \text{mgu } \sigma (\tau_1 \equiv \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \)
(ii) \( \text{mgu } \sigma (\tau_1 \equiv \tau_2) \land \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \delta. \sigma' \approx \sigma \circ \delta \)
(iii) \( \text{mgu } \sigma (\tau_1 \equiv \tau_2) \Rightarrow \text{FTVS}(\sigma) \subseteq \text{FVC}(\tau_1 \equiv \tau_2) \)
(iv) \( \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'. \text{mgu } \sigma'(\tau_1 \equiv \tau_2) \)

New Generalized Axioms

(i) \( \text{unify } C = \text{Some } \sigma \Rightarrow \sigma \models C \)
(ii) \( (\text{unify } C = \text{Some } \sigma \land \sigma' \models C) \Rightarrow \exists \sigma''. \sigma' \approx \sigma \circ \sigma'' \)
(iii) \( \text{unify } C = \text{Some } \sigma \Rightarrow \text{FTVS}(\sigma) \subseteq \text{FVC}(C) \)
(iv) \( \sigma \models C \Rightarrow \exists \sigma'. \text{unify } C = \text{Some } \sigma' \)
A model for MGU axioms

Axiom iii

Lemma 2 (Compose and domain membership)

\[ \forall x, y. \forall \tau. \forall \sigma. \quad y \in \text{dom}_\text{subst} (\{x \mapsto \tau\} \circ \sigma) \quad \Rightarrow \quad y \in \text{dom}_\text{subst} \{x \mapsto \tau\} \lor y \in \text{dom}_\text{subst} \sigma \]

Lemma 3 (Compose and range membership)

\[ \forall x, y. \forall \tau. \forall \sigma. \quad (x \notin \text{FTV} \tau) \land y \in \text{range}_\text{subst} (\{x \mapsto \tau\} \circ \sigma) \quad \Rightarrow \quad y \in \text{range}_\text{subst} \{x \mapsto \tau\} \lor y \in \text{range}_\text{subst} \sigma \]
A model for MGU axioms

Axiom iii ...

Lemma 4 (Subst range abstraction)

\[ \forall x. \forall \sigma. \ x \in \text{range\_subst} (\sigma) \iff \exists y. y \in \text{dom\_subst} (\sigma) \land x \in FTV(\sigma(TyVar y)) \]

Theorem 5

\[ \forall \sigma, \sigma'. \forall C. \text{unify } C = \text{Some } \sigma \Rightarrow \text{FTVS}(\sigma) \subseteq \text{FVC}(C) \]

Proof.

By functional induction on \text{unify } C and lemmas 2, 3.
A model for MGU axioms

**Axiom iv**

**Proper Subterms Definition**

\[
\begin{align*}
\text{subterms } & \alpha \quad \text{def} \quad [ ] \\
\text{subterms } (\tau_1 \rightarrow \tau_2) \quad \text{def} \quad \tau_1 :: \tau_2 :: (\text{subterms } \tau_1) + (\text{subterms } \tau_2)
\end{align*}
\]

**Lemma 6 (Containment)**

\[\forall \tau, \tau'. \tau \in (\text{subterms } \tau') \Rightarrow \forall \tau''. \tau'' \in (\text{subterms } \tau) \Rightarrow \tau'' \in (\text{subterms } \tau')\]

**Proof.**

By induction on \(\tau'\).

**Lemma 7 (Well founded types)**

\[\forall \tau. \neg \tau \in (\text{subterms } \tau)\]

**Proof.**

By induction on \(\tau\) and by lemma 6.
A model for MGU axioms

Axiom iv ... contd

**Lemma 8 (Member subterms unequal)**

\[ \forall \tau, \tau'. \; \tau \in (\text{subterms } \tau') \Rightarrow \tau \neq \tau' \]

**Proof.**

By case analysis on \( \tau = \tau' \) and by lemma 7.

**Lemma 9 (Member subterms and apply subst)**

\[ \forall \sigma. \; \forall \alpha. \; \forall \tau. \; \alpha \in (\text{subterms } \tau) \Rightarrow \sigma(\alpha) \neq \sigma(\tau) \]

**Proof.**

By induction on \( \tau \) and by lemma 8.
Lemma 10 (Member arrow and subterms)

$$\forall \sigma. \forall x. \forall \tau_1, \tau_2. \text{member } x \ (\text{FTV } \tau_1) = \text{true } \lor \text{member } x \ (\text{FTV } \tau_2) = \text{true} \Rightarrow \text{TyVar } (x) \in \text{subterms}(\tau_1 \rightarrow \tau_2)$$

Proof.
By induction on $\tau_1$, followed by induction on $\tau_2$.

Corollary 11 (Member apply subst unequal)

$$\forall \sigma. \forall x. \forall \tau_1, \tau_2. \text{member } x \ (\text{FTV } \tau_1) = \text{true } \lor \text{member } x \ (\text{FTV } \tau_2) = \text{true} \Rightarrow \sigma(\text{TyVar } (x)) \neq \sigma(\tau_1 \rightarrow \tau_2)$$

Proof.
By lemma 9 and 10.
Theorem 12
\[ \forall \sigma. \forall C. \sigma \models C \Rightarrow \exists \sigma'. \text{unify } C = \text{Some } \sigma' \]

Proof.
By functional induction on unify \( C \) and lemma ?? and corollary 11.
Outline

1 Overview
   - Type Reconstruction Algorithms

2 Introduction
   - Substitution
   - Coq

3 First-order unification algorithm
   - Specification in Coq
   - Termination

4 A model for MGU axioms
   - Axiom iii
   - Axiom iv

5 Conclusions and Future Work
Some of the lemmas are more generalized version of the lemmas actually needed.
Conclusions and Future Work

- Some of the lemmas are more generalized version of the lemmas actually needed.
- In many proofs we abstracted away from the actual construct or looked at its behavior.

The entire verification took almost 4400 lines of specification and tactics and is available online at [http://www.cs.uwyo.edu/~skothari](http://www.cs.uwyo.edu/~skothari).

Many of the lemmas and theorems will be useful in our machine certified correctness proof of Wand's algorithm.
Some of the lemmas are more generalized version of the lemmas actually needed.

In many proofs we abstracted away from the actual construct or looked at its behavior.

The entire verification took almost 4400 lines of specification and tactics and is available online at http://www.cs.uwyo.edu/~skothari.
Some of the lemmas are more generalized version of the lemmas actually needed.

In many proofs we abstracted away from the actual construct or looked at its behavior.

The entire verification took almost 4400 lines of specification and tactics and is available online at http://www.cs.uwyo.edu/~skothari.

Many of the lemmas and theorems will be useful in our machine certified correctness proof of Wand’s algorithm.
Merci!!!!!
Conclusions and Future Work

Induction Principle

\[
\text{unify\_ind} \\
: \forall P : \text{list constr} \to \text{option (M.t type)} \to \text{Prop}, \\
\quad (\forall c : \text{list constr}, c = \text{nil} \to P \text{ nil (Some (M.empty type))}) \to \\
\quad (\forall (c : \text{list constr}) (h : \text{constr}) (t : \text{list constr}), \\
\quad \quad c = h :: t \to \\
\quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \forall _x : x = y, \\
\quad \quad \quad \quad \text{eq\_dec\_stamp} x y = \text{left (x <> y)} \_x \to \\
\quad \quad \quad \text{P} t (\text{unify} t) \to \text{P} (\text{EqCons (TyVar x) (TyVar y) :: t}) (\text{unify} t) \to \\
\quad \quad (\forall (c : \text{list constr}) (h : \text{constr}) (t0 : \text{list constr}), \\
\quad \quad \quad c = h :: t0 \to \\
\quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \forall _x : x <> y, \\
\quad \quad \quad \quad \text{eq\_dec\_stamp} x y = \text{right (x = y)} \_x \to \\
\quad \quad \quad \text{P} (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0}) \\
\quad \quad \quad \quad (\text{unify}) \\
\quad \quad \quad \quad \quad (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0})) \to \\
\quad \quad \forall p : \text{M.t type}, \\
\quad \quad \text{unify} \\
\quad \quad \quad (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0}) = \\
\quad \quad \quad \text{Some p \to} \\
\quad \quad \quad \text{P} (\text{EqCons (TyVar x) (TyVar y) :: t0}) \\
\quad \quad \quad \quad (\text{Some (compose\_subst (M.add x (TyVar y) (M.empty type)) p))) \to \\
\quad \quad \quad (\forall (c : \text{list constr}) (h : \text{constr}) (t0 : \text{list constr}), \\
\quad \quad \quad \quad c = h :: t0 \to \\
\quad \quad \quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \quad \forall _x : x <> y, \\
\quad \quad \quad \quad \text{forall p : M.t type, \\
\quad \quad \quad \quad \text{unify} \\
\quad \quad \quad \quad \quad (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0}) = \\
\quad \quad \quad \quad \text{Some p \to} \\
\quad \quad \quad \quad \text{P} (\text{EqCons (TyVar x) (TyVar y) :: t0}) \\
\quad \quad \quad \quad \quad (\text{Some (compose\_subst (M.add x (TyVar y) (M.empty type)) p))) \to \\
\quad \quad \quad \quad (\forall (c : \text{list constr}) (h : \text{constr}) (t0 : \text{list constr}), \\
\quad \quad \quad \quad \quad c = h :: t0 \to \\
\quad \quad \quad \quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \quad \quad \forall _x : x <> y, \\
\quad \quad \quad \quad \quad \text{forall p : M.t type, \\
\quad \quad \quad \quad \quad \text{unify} \\
\quad \quad \quad \quad \quad \quad (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0}) = \\
\quad \quad \quad \quad \quad \text{Some p \to} \\
\quad \quad \quad \quad \quad \text{P} (\text{EqCons (TyVar x) (TyVar y) :: t0}) \\
\quad \quad \quad \quad \quad \quad (\text{Some (compose\_subst (M.add x (TyVar y) (M.empty type)) p))) \to \\
\quad \quad \quad \quad \quad \quad (\forall (c : \text{list constr}) (h : \text{constr}) (t0 : \text{list constr}), \\
\quad \quad \quad \quad \quad \quad \quad c = h :: t0 \to \\
\quad \quad \quad \quad \quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad \quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \quad \quad \quad \quad \forall _x : x <> y, \\
\quad \quad \quad \quad \quad \quad \quad \text{forall p : M.t type, \\
\quad \quad \quad \quad \quad \quad \quad \text{unify} \\
\quad \quad \quad \quad \quad \quad \quad \quad (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0}) = \\
\quad \quad \quad \quad \quad \quad \quad \text{Some p \to} \\
\quad \quad \quad \quad \quad \quad \quad \text{P} (\text{EqCons (TyVar x) (TyVar y) :: t0}) \\
\quad \quad \quad \quad \quad \quad \quad \quad (\text{Some (compose\_subst (M.add x (TyVar y) (M.empty type)) p))) \to \\
\quad \quad \quad \quad \quad \quad \quad \quad (\forall (c : \text{list constr}) (h : \text{constr}) (t0 : \text{list constr}), \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad c = h :: t0 \to \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \forall _x : x <> y, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{forall p : M.t type, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{unify} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{apply\_subst\_to\_constr\_list (M.add x (TyVar y) (M.empty type)) t0}) = \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{Some p \to} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{P} (\text{EqCons (TyVar x) (TyVar y) :: t0}) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{Some (compose\_subst (M.add x (TyVar y) (M.empty type)) p))) \to \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\forall (c : \text{list constr}) (h : \text{constr}) (t0 : \text{list constr}), \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad c = h :: t0 \to \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \forall x y : \text{nat}, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad h = \text{EqCons (TyVar x) (TyVar y)} \to \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \forall _x : x <> y, \
\]
Conclusions and Future Work


Christian Urban and Tobias Nipkow.

Mitchell Wand.