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

Outline

Highlights

- 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.

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.
  - Constraint-based algorithms.
    - Two distinct phases: constraint generation and constraint solving.

Type Reconstruction Algorithms...contd

Substitution-based

- Algorithm W, J by Milner, 1978.
- Algorithm M by Leroy, 1993.

Type Reconstruction Algorithms...contd

Substitution-based

- Algorithm W, J by Milner, 1978.
- Algorithm M by Leroy, 1993.

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].

Type Reconstruction Algorithms... Contd

Machine-Certified Correctness Proof

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

Type Reconstruction Algorithms... Contd

Machine-Certified Correctness Proof

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

Type Reconstruction Algorithms... Contd

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.

Type Reconstruction Algorithms... Contd

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.

Type Reconstruction Algorithms... Contd

Modeling MGU

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

Type Reconstruction Algorithms... Contd

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:

$$
\begin{array}{ll}
(i) & \mathit{mgu}\,\sigma\ (\tau_1 \overset{c}{=} \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \\
(ii) & \mathit{mgu}\,\sigma\ (\tau_1 \overset{c}{=} \tau_2) \wedge \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \sigma''.\sigma' \approx \sigma \circ \sigma'' \\
(iii) & \mathit{mgu}\,\sigma\ (\tau_1 \overset{c}{=} \tau_2) \Rightarrow \mathrm{FTVS}\ (\sigma) \subseteq \mathrm{FVC}\ (\tau_1 \overset{c}{=} \tau_2) \\
(iv) & \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'.\ \mathit{mgu}\,\sigma'(\tau_1 \overset{c}{=} \tau_2)
\end{array}
$$

Outline

Terms and Constraint Syntax

Terms

- $\tau ::= \mathsf{TyVar}(x) \mid \tau' \to \tau''$

Terms and Constraint Syntax

Terms

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

Terms and Constraint Syntax

Terms

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

Constraints

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

Terms and Constraint Syntax

Terms

- $\tau ::= \mathsf{TyVar}(x) \mid \tau' \rightarrow \tau''$
- Atomic types (of the form TyVar *x*) are denoted by α, β, α' etc.

Constraints

- Constraint are of the form $\tau \stackrel{c}{=} \tau'$.
- A list of constraint is given as:
  - $\mathbb{C} ::= [\,] \mid \tau \stackrel{c}{=} \tau' :: \mathbb{C}'$

FTV and FVC

Free type variable (FTV)

$$\mathrm{FTV}\ (\mathrm{TyVar}\ x) \stackrel{def}{=} [x]$$
$$\mathrm{FTV}\ (\tau \rightarrow \tau') \stackrel{def}{=} \mathrm{FTV}\,(\tau) +\!+ \ \mathrm{FTV}\,(\tau')$$

FTV and FVC

Free type variable (FTV)

$$\begin{aligned}
\text{FTV}\ (\text{TyVar}\ x) &\stackrel{\text{def}}{=} [x] \\
\text{FTV}\ (\tau \rightarrow \tau') &\stackrel{\text{def}}{=} \text{FTV}\,(\tau) +\!\!+\ \text{FTV}\,(\tau')
\end{aligned}$$

Free variables of a constraint list (FVC)

$$\begin{aligned}
\text{FVC}\ [\,] &\stackrel{\text{def}}{=} [\,] \\
\text{FVC}\ ((\tau_1 \stackrel{c}{=} \tau_2) :: \mathbb{C}) &\stackrel{\text{def}}{=} \text{FTV}\,(\tau_1) +\!\!+\ \text{FTV}\,(\tau_2) +\!\!+\ \text{FVC}\,(\mathbb{C})
\end{aligned}$$

Substitution

Related Concepts

- A *substitution* (denoted by ρ) maps type variables to types.

Substitution

Related Concepts

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

Substitution

Related Concepts

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

$$
\begin{aligned}
\sigma\ (\mathrm{TyVar}(x)) &\stackrel{\mathit{def}}{=} \mathit{if}\ \langle x, \tau\rangle \in \sigma\ \mathit{then}\ \tau\ \mathit{else}\ \mathrm{TyVar}(x) \\
\sigma\ (\tau_1 \rightarrow \tau_2) &\stackrel{\mathit{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2)
\end{aligned}
$$

Substitution

Related Concepts

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

$$\begin{array}{rcl} \sigma\ (\mathsf{TyVar}(x)) & \stackrel{def}{=} & \textit{if}\ \langle x, \tau\rangle \in \sigma\ \textit{then}\ \tau\ \textit{else}\ \mathsf{TyVar}(x) \\ \sigma\ (\tau_1 \to \tau_2) & \stackrel{def}{=} & \sigma(\tau_1) \to \sigma(\tau_2) \end{array}$$

- Application of a substitution to a constraint is defined similarly:

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

Substitution

Substitution Composition

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

Theorem 1 (Composition apply)

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

Substitution

Substitution Composition

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

Theorem 1 (Composition apply)

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

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.

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

Unifiers and MGUs

Unifier

- We write $\sigma \models (\tau_1 \stackrel{c}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.
- $\sigma \models \mathbb{C} \stackrel{def}{=} \forall c \in \mathbb{C}, \sigma \models c$.

Unifiers and MGUs

Unifier

- We write $\sigma \models (\tau_1 \stackrel{c}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.
- $\sigma \models \mathbb{C} \stackrel{def}{=} \forall c \in \mathbb{C}, \sigma \models c$.

Most General Unifier

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

Coq

Overview

- Based on the Calculus of Constructions.

Coq

Overview

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

Coq

Overview

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

Coq

Overview

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

Coq

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.

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.

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.

Substitution Related Concepts in Coq

Domain

$$\mathrm{dom\_subst}(\sigma) \stackrel{def}{=} \mathrm{List.map}\ (\lambda t.\mathrm{fst}\ (t))\ (\mathrm{M.elements}(\ \sigma))$$

Substitution Related Concepts in Coq

Domain

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

Range

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

Substitution Related Concepts in Coq

Domain

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

Range

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

FTVS

$$\text{FTVS}(\sigma) \stackrel{def}{=} \text{dom\_subst}(\sigma) ++ \text{range\_subst}(\sigma)$$

Outline

Unification

The Algorithm

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

Unification

The Algorithm

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

Specification in Coq

```
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)
```

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 ....

Termination

Lexicographic Ordering

- The lexicographic ordering ($\prec_3$) on the two triples $\langle n_1, n_2, n_3 \rangle$ and $\langle m_1, m_2, m_3 \rangle$ is defined as
$\langle n_1, n_2, n_3 \rangle \prec_3 \langle m_1, m_2, m_3 \rangle \stackrel{def}{=}$
$$(n_1 < m_1) \vee (n_1 = m_1 \wedge n_2 < m_2) \vee (n_1 = m_1 \wedge n_2 = m_2 \wedge n_3 < m_3),$$
where $<$ and $=$ are the ordinary less-than inequality and equality on natural numbers.

The Triple

- The triple is $<|\, C_{FVC} \,|, |\, C_{\rightarrow} \,|, |\, C \,|>$, 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.

Termination...contd

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

Table: Variation of termination measure components on the recursive call

Outline

Functional Induction in Coq

- Requires an induction principle generated before.

Functional Induction in Coq

- Requires an induction principle generated before.
- `functional induction (f x1 x2 x3 .. xn)` is a short form for `induction x1 x2 x3 ...xn f(x1 ... xn) using` *id*, where *id* is the induction principle for *f*.

Functional Induction in Coq

- Requires an induction principle generated before.
- `functional induction (f x1 x2 x3 .. xn)` is a short form for `induction x1 x2 x3 ...xn f(x1 ... xn) using` *id*, where *id* is the induction principle for *f*.
  - `functional induction (unify c)` $\leadsto$ `induction c (unify c) using unif_ind`.
- Important first step in proof of the axioms.

MGU axioms

Old Axioms

$$
\begin{array}{ll}
(i) & \textit{mgu}\ \sigma\ (\tau_1 \stackrel{c}{=} \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \\
(ii) & \textit{mgu}\ \sigma\ (\tau_1 \stackrel{c}{=} \tau_2) \wedge \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \delta . \sigma' \approx \sigma \circ \delta \\
(iii) & \textit{mgu}\ \sigma\ (\tau_1 \stackrel{c}{=} \tau_2) \Rightarrow \mathsf{FTVS}\ (\sigma) \subseteq \mathsf{FVC}\ (\tau_1 \stackrel{c}{=} \tau_2) \\
(iv) & \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma' .\ \textit{mgu}\ \sigma'(\tau_1 \stackrel{c}{=} \tau_2)
\end{array}
$$

MGU axioms

Old Axioms

$$
\begin{array}{ll}
(i) & \textit{mgu}\ \sigma\ (\tau_1 \stackrel{c}{=} \tau_2) \Rightarrow \sigma(\tau_1) = \sigma(\tau_2) \\
(ii) & \textit{mgu}\ \sigma\ (\tau_1 \stackrel{c}{=} \tau_2) \wedge \sigma'(\tau_1) = \sigma'(\tau_2) \Rightarrow \exists \delta . \sigma' \approx \sigma \circ \delta \\
(iii) & \textit{mgu}\ \sigma\ (\tau_1 \stackrel{c}{=} \tau_2) \Rightarrow \text{FTVS}\ (\sigma) \subseteq \text{FVC}\ (\tau_1 \stackrel{c}{=} \tau_2) \\
(iv) & \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma' .\ \textit{mgu}\ \sigma' (\tau_1 \stackrel{c}{=} \tau_2)
\end{array}
$$

New Generalized Axioms

$$
\begin{array}{ll}
(i) & \text{unify}\ \mathbb{C} = \text{Some}\ \sigma\ \Rightarrow \sigma \models \mathbb{C} \\
(ii) & (\text{unify}\ \mathbb{C} = \text{Some}\ \sigma\ \wedge \sigma' \models \mathbb{C}) \Rightarrow \exists \sigma''.\ \sigma' \approx \sigma \circ \sigma'' \\
(iii) & \text{unify}\ \mathbb{C} = \text{Some}\ \sigma\ \Rightarrow \text{FTVS}(\sigma) \subseteq \text{FVC}\ (\mathbb{C}) \\
(iv) & \sigma \models \mathbb{C}\ \Rightarrow \exists \sigma'.\ \text{unify}\ \mathbb{C} = \text{Some}\ \sigma'
\end{array}
$$

Axiom iii

Lemma 2 (Compose and domain membership)

$$\forall x, y.\ \forall\tau.\forall\sigma.\ y \in \text{dom\_subst}\ (\{x \mapsto \tau\} \circ \sigma)) \\ \quad \Rightarrow y \in \text{dom\_subst}\ \{x \mapsto \tau\} \lor y \in \text{dom\_subst}\ \sigma$$

Lemma 3 (Compose and range membership)

$$\forall x, y.\ \forall\tau.\ \forall\sigma.\ (x \notin (\text{FTV}\ \tau) \land y \in \text{range\_subst}\ (\{x \mapsto \tau\} \circ \sigma)) \\ \quad \Rightarrow y \in \text{range\_subst}\ \{x \mapsto \tau\} \lor y \in \text{range\_subst}\ \sigma$$

Axiom iii ...contd

Lemma 4 (Subst range abstraction)

$\forall x.\, \forall \sigma.\, x \in \text{range\_subst}\ (\sigma) \Leftrightarrow \exists y. y \in \text{dom\_subst}\ (\sigma) \wedge x \in FTV(\sigma(\text{TyVar}\ y))$

Theorem 5

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

Proof.

By functional induction on unify $\mathbb{C}$ and lemmas 2, 3. □

Axiom iv

Proper Subterms Definition

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

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 τ'. □

Lemma 7 (Well founded types)

$$\forall \tau.\ \neg\, \tau \in (\text{subterms } \tau)$$

Proof.

By induction on τ and by lemma 6. □

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 τ and by lemma 8. □

Axiom iv...contd

Lemma 10 (Member arrow and subterms)

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

Proof.

By induction on τ_1, followed by induction on τ_2. □

Corollary 11 (Member apply subst unequal)

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

Proof.

By lemma 9 and 10. □

Axiom iv ... contd

Theorem 12

$\forall \sigma.\ \forall \mathbb{C}.\ \sigma \models \mathbb{C} \ \Rightarrow \exists \sigma'.$ unify $\mathbb{C} =$ Some σ'

Proof.

By functional induction on unify $\mathbb{C}$ and lemma **??** and corollary 11. □

Outline

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.

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`.

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`.
- Many of the lemmas and theorems will be useful in our machine certified correctness proof of Wand's algorithm.

Merci!!!!!

Induction Principle

```
unify_ind
     : forall P : list constr -> option (M.t type) -> Prop,
       (forall c : list constr, c = nil -> P nil (Some (M.empty type))) ->
       (forall (c : list constr) (h : constr) (t : list constr),
        c = h :: t ->
        forall x y : nat,
        h = EqCons (TyVar x) (TyVar y) ->
        forall _x : x = y,
        eq_dec_stamp x y = left (x <> y) _x ->
        P t (unify t) -> P (EqCons (TyVar x) (TyVar y) :: t) (unify t)) ->
       (forall (c : list constr) (h : constr) (t0 : list constr),
        c = h :: t0 ->
        forall x y : nat,
        h = EqCons (TyVar x) (TyVar y) ->
        forall _x : x <> y,
        eq_dec_stamp x y = right (x = y) _x ->
        P (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0)
          (unify
             (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type))
                t0)) ->
        forall p : M.t type,
        unify
          (apply_subst_to_constr_list (M.add x (TyVar y) (M.empty type)) t0) =
        Some p ->
        P (EqCons (TyVar x) (TyVar y) :: t0)
          (Some (compose_subst (M.add x (TyVar y) (M.empty type)) p))) ->
       (forall (c : list constr) (h : constr) (t0 : list constr),
        c = h :: t0 ->
        forall x y : nat,
        h = EqCons (TyVar x) (TyVar y) ->
        forall _x : x <> y,
```

Catherine Dubois and Valerie M. Morain.
Certification of a type inference tool for ml: Damas–milner within coq.
J. Autom. Reason., 23(3):319–346, 1999.

Bastiaan Heeren.
Top Quality Type Error Messages.
PhD thesis, Universitiet Utrecht, 2005.

Mark P. Jones.
Qualified Types: Theory and Practice.
Distinguished Dissertations in Computer Science. Cambridge University Press, 1995.

Dieter Nazareth and Tobias Nipkow.
Theorem Proving in Higher Order Logics, volume 1125, chapter Formal Verification of Alg. W: The Monomorphic Case, pages 331–345.
Springer Berlin / Heidelberg, 1996.

Wolfgang Naraschewski and Tobias Nipkow.
Type inference verified: Algorithm w in isabelle/hol.
J. Autom. Reason., 23(3):299–318, 1999.

Wolfgang Naraschewski and Tobias Nipkow.
Type Inference Verified: Algorithm W in Isabelle/HOL.
Journal of Automated Reasoning, 23(3-4):299–318, 1999.

F. Pottier and D. Rémy.
The essence of ML type inference.
In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

Martin Sulzmann, Martin Odersky, and Martin Wehr.
Type inference with constrained types.
In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.

Christian Urban and Tobias Nipkow.

From Semantics to Computer Science, chapter Nominal verification of algorithm W.
Cambridge University Press, 2009.

Mitchell Wand.
A simple algorithm and proof for type inference.
Fundamenta Informaticae, 10:115–122, 1987.