Toward a machine-certified correctness proof of Wand's type reconstruction algorithm

Presented by Sunil Kothari
Joint work with Prof. James Caldwell

Department of Computer Science,
University of Wyoming, USA

Outline

Outline

1. **Overview**
   - Type Reconstruction Algorithms
2. **Introduction**
   - Wand's Algorithm
   - Substitution
3. **Correctness Proof**
   - Issues In Formalization
   - Soundness and Completeness Proofs
4. **Conclusions and Future Work**

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.

Substitution-based Algorithms

Examples

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

Substitution-based Algorithms

Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99, NN96].

Substitution-based Algorithms

Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99, NN96].
- Nominal verification of Algorithm W (in Isabelle/HOL) [UN09].

Substitution-based Algorithms

Machine-Certified Correctness Proof

- Algorithm W in Coq, Isabelle/HOL [DM99, NN99, NN96].
- Nominal verification of Algorithm W (in Isabelle/HOL) [UN09].
- The formalization in Coq is not available online.

Constraint-based Frameworks/Algorithms

Examples

- Wand's algorithm [Wan87].
- HM(X) [SOW97] by Sulzmann et al. 1999, Pottier and Rémy 2005 [PR05], Qualified types [Jon95].
- Top quality error messages [Hee05].

Constraint-based Algorithms/Frameworks

Machine-Certified Correctness Proof

- We know of no correctness proof of Wand's type reconstruction algorithm not verified in any theorem prover.

Constraint-based Algorithms/Frameworks

Machine-Certified Correctness Proof

- We know of no correctness proof of Wand's type reconstruction algorithm not verified in any theorem prover.
- We want to verify our extension of Wand's algorithm for polymorphic let.

Constraint-based Algorithms/Frameworks

Machine-Certified Correctness Proof

- We know of no correctness proof of Wand's type reconstruction algorithm not verified in any theorem prover.
- We want to verify our extension of Wand's algorithm for polymorphic let.
- POPLMark challenge also aims at mechanizing meta-theory.

Outline

1. Overview
   - Type Reconstruction Algorithms
2. Introduction
   - Wand's Algorithm
   - Substitution
3. Correctness Proof
   - Issues In Formalization
   - Soundness and Completeness Proofs
4. Conclusions and Future Work

Terms and Constraint Syntax

Terms

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

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 $\tau \stackrel{c}{=} \tau'$.

Substitution

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

Substitution

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

Unifier

- We write $\sigma \models (\tau_1 \stackrel{c}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.

Substitution

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

Unifier

- We write $\sigma \models (\tau_1 \stackrel{c}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.

Substitution

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

Unifier

- We write $\sigma \models (\tau_1 \stackrel{c}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.

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 $\sigma \circ \sigma' \approx \sigma''$.

Wand's Algorithm

Let G denote a set of goals. And E a set of equations.

- **Input.** A term M of Λ.
- **Initialization.** Set $E = \emptyset$ and $G = \{(\Gamma, M, \alpha_0)\}$.
- **Loop Step.** If $G = \emptyset$ then return E else choose a subgoal (Γ, M, τ) from G and add to E and G new verification conditions and subgoals by looking at the action table.

Wand's Algorithm

Action Table

Case (Γ, x, τ). Generate the equation $\tau \stackrel{c}{=} \Gamma(x)$.

Wand's Algorithm

Action Table

Case (Γ, x, τ). Generate the equation $\tau \overset{c}{=} \Gamma(x)$.

Case (Γ, MN, τ). Generate subgoals $(\Gamma, M, \tau' \rightarrow \tau)$ and (Γ, N, τ').

Wand's Algorithm

Action Table

Case (Γ, x, τ). Generate the equation $\tau \overset{c}{=} \Gamma(x)$.

Case (Γ, MN, τ). Generate subgoals $(\Gamma, M, \tau' \rightarrow \tau)$ and (Γ, N, τ').

Case ($\Gamma, \lambda x.M, \tau$). Generate equation $\tau \overset{c}{=} \tau' \rightarrow \tau''$ and subgoal $([x : \tau'] :: \Gamma, M, \tau'')$.

Wand's Algorithm - Example

$\{(\emptyset, \lambda x.\lambda y.\lambda z.xz(yz), \alpha_0)\}; \{\}$
$\{((x : \alpha_1), \lambda y.\lambda z.xz(yz), \alpha_2)\}; \{\alpha_0 \stackrel{c}{=} \alpha_1 \to \alpha_2\}$
$\{((x : \alpha_1, y : \alpha_3), \lambda z.xz(yz), \alpha_4)\}; \{\alpha_2 \stackrel{c}{=} \alpha_3 \to \alpha_4\}$
$\{((x : \alpha_1, y : \alpha_3, z : \alpha_5), xz(yz), \alpha_6)\}; \{\alpha_4 \stackrel{c}{=} \alpha_5 \to \alpha_6\}$
$\{(((x : \alpha_1, z : \alpha_5), xz, \alpha_7 \to \alpha_6), ((y : \alpha_3, z : \alpha_5), yz, \alpha_7))\}; \{\}$
$\{((x : \alpha_1), x, \alpha_8 \to (\alpha_7 \to \alpha_6)), ((z : \alpha_5), z, \alpha_8), ((y : \alpha_3, z : \alpha_5), yz, \alpha_7)\}; \{\}$
$\{(((z : \alpha_5), z, \alpha_8), ((y : \alpha_3, z : \alpha_5), yz, \alpha_7))\}; \{\alpha_1 \stackrel{c}{=} \alpha_8 \to \alpha_7 \to \alpha_6\}$
$\{((y : \alpha_3, z : \alpha_5), yz, \alpha_7)\}; \{\alpha_8 \stackrel{c}{=} \alpha_5\}$
$\{((y : \alpha_3), y, \alpha_9 \to \alpha_7), ((z : \alpha_5), z, \alpha_9)\}; \{\}$
$\{((z : \alpha_5), z, \alpha_9)\}; \{\alpha_9 \to \alpha_7 \stackrel{c}{=} \alpha_3\}$
$\emptyset; \{\alpha_9 \stackrel{c}{=} \alpha_5\}$

Wand's Algorithm Example - Alternate View

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\alpha_1\} \overset{\{\alpha_8\to\alpha_7\to\alpha_6 \stackrel{c}{=} \alpha_1\}}{\vdash} x:\alpha_8\to\alpha_7\to\alpha_6} \quad \dfrac{}{\{z:\alpha_5\} \overset{\{\alpha_8 \stackrel{c}{=} \alpha_5\}}{\vdash} z:\alpha_8}}{\{x:\alpha_1, z:\alpha_5\} \overset{\{\}}{\vdash} xz:\alpha_7\to\alpha_6} \quad \dfrac{\dfrac{}{\{y:\alpha_3\} \overset{\{\alpha_9\to\alpha_7 \stackrel{c}{=} \alpha_3\}}{\vdash} y:\alpha_9\to\alpha_7} \quad \dfrac{}{\{z:\alpha_5\} \overset{\{\alpha_9 \stackrel{c}{=} \alpha_5\}}{\vdash} z:\alpha_9}}{\{y:\alpha_3, z:\alpha_5\} \overset{\{\}}{\vdash} yz:\alpha_7}}{\{x:\alpha_1, y:\alpha_3, z:\alpha_5\} \overset{\{\}}{\vdash} xz(yz):\alpha_6}}{\{x:\alpha_1, y:\alpha_3\} \overset{\{\alpha_4 \stackrel{c}{=} \alpha_5\to\alpha_6\}}{\vdash} \lambda z.xz(yz):\alpha_4}}{\{x:\alpha_1\} \overset{\{\alpha_2 \stackrel{c}{=} \alpha_3\to\alpha_4\}}{\vdash} \lambda y.\lambda z.xz(yz):\alpha_2}}{\{\} \overset{\{\alpha_0 \stackrel{c}{=} \alpha_1\to\alpha_2\}}{\vdash} \lambda x.\lambda y.\lambda z.xz(yz):\alpha_0}
$$

Example - Solution

$\alpha_0 \stackrel{c}{=} \alpha_1 \rightarrow \alpha_2$
$\alpha_2 \stackrel{c}{=} \alpha_3 \rightarrow \alpha_4$
$\alpha_4 \stackrel{c}{=} \alpha_5 \rightarrow \alpha_9$
$\alpha_1 \stackrel{c}{=} \alpha_8 \rightarrow \alpha_7 \rightarrow \alpha_9$
$\alpha_8 \stackrel{c}{=} \alpha_5$
$\alpha_9 \rightarrow \alpha_7 \stackrel{c}{=} \alpha_3$
$\alpha_9 \stackrel{c}{=} \alpha_5$

After unifying the above constraints,

$\alpha_0 \mapsto (\alpha_5 \rightarrow \alpha_7 \rightarrow \alpha_6) \rightarrow (\alpha_5 \rightarrow \alpha_7) \rightarrow (\alpha_5 \rightarrow \alpha_6)$

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 a finite map

- Used the Coq's finite maps library *Coq.FSets.FMapInterface* (ver. 8.1pl3).
- Axiomatic presentation.
- Provides no induction principle.
- Forward reasoning is often required.

Substitution

Related Concepts

- 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

- Substitution application to a type τ is defined as:

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

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

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

- Assumption: Idempotent substitution.

Substitution

Substitution Composition

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

Theorem 1 (Composition apply)

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

- Substitution representation determines the reasoning.
  - A list of pairs: 600 proof steps [DM99].
  - Finite maps: 100 proof steps.

Outline

Wand's Algorithm

Issues in formalization

- Raise exceptions, but that's not possible.
  - We choose an `option` type.
- Freshness is now explicit.
- The W-App rule now generates a constraint.

Wand's Algorithm

Issues in formalization

- Raise exceptions, but that's not possible.
  - We choose an `option` type.

$$\frac{\text{search\_type\_env}(x, \Gamma) = \text{Some } \tau}{\text{Wand}(\Gamma, x, n_0) = (\text{Some } \{\text{Tvar}(n_0) \overset{c}{=} \tau\}, n_0 + 1)} \text{ (W-Var)}$$

$$\frac{\text{Wand}(((x : \text{Tvar}(n_0 + 1)) :: \Gamma), M, n_0 + 2) = (\text{Some } \mathbb{C}, n_1)}{\text{Wand}(\Gamma, \lambda x.M, n_0) = (\text{Some } \{\text{Tvar}(n_0) \overset{c}{=} \text{Tvar}(n_0 + 1) \to \text{Tvar}(n_0 + 2)\} \cup \mathbb{C}, n_1)} \text{ (W-Abs)}$$

$$\frac{\text{Wand}(\Gamma, M, n_0 + 1) = (\text{Some } \mathbb{C}', n_1) \qquad \text{Wand}(\Gamma, N, n_1) = (\text{Some } \mathbb{C}'', n_2)}{\text{Wand}(\Gamma, MN, n_0) = (\text{Some } \{\text{Tvar}(n_0 + 1) \overset{c}{=} \text{Tvar}(n_1) \to \text{Tvar}(n_0)\} \cup \mathbb{C}' \cup \mathbb{C}'', n_2)} \text{ (W-App)}$$

Wand's Algorithm

Issues in formalization

- Freshness is now explicit.

$$\frac{\text{search\_type\_env}(x, \Gamma) = \text{Some } \tau}{\text{Wand}(\Gamma, x, n_0) = (\text{Some } \{\text{Tvar}(n_0) \stackrel{c}{=} \tau\}, n_0 + 1)} \text{ (W-Var)}$$

$$\frac{\text{Wand}(((x : \text{Tvar}(n_0 + 1)) :: \Gamma), M, n_0 + 2) = (\text{Some } \mathbb{C}, n_1)}{\text{Wand}(\Gamma, \lambda x.M, n_0) = (\text{Some } \{\text{Tvar}(n_0) \stackrel{c}{=} \text{Tvar}(n_0 + 1) \rightarrow \text{Tvar}(n_0 + 2)\} \cup \mathbb{C}, n_1)} \text{ (W-Abs)}$$

$$\frac{\text{Wand}(\Gamma, M, n_0 + 1) = (\text{Some } \mathbb{C}', n_1) \qquad \text{Wand}(\Gamma, N, n_1) = (\text{Some } \mathbb{C}'', n_2)}{\text{Wand}(\Gamma, MN, n_0) = (\text{Some } \{\text{Tvar}(n_0 + 1) \stackrel{c}{=} \text{Tvar}(n_1) \rightarrow \text{Tvar}(n_0)\} \cup \mathbb{C}' \cup \mathbb{C}'', n_2)} \text{ (W-App)}$$

Wand's Algorithm

Issues in formalization

- The W-App rule now generates a constraint.

$$\frac{\text{search\_type\_env}(x,\Gamma) = \text{Some } \tau}{\text{Wand}(\Gamma, x, n_0) = (\text{Some } \{\text{Tvar}(n_0) \stackrel{c}{=} \tau\}, n_0 + 1)} \text{ (W-Var)}$$

$$\frac{\text{Wand}(((x : \text{Tvar}(n_0+1)) :: \Gamma), M, n_0 + 2) = (\text{Some } \mathbb{C}, n_1)}{\text{Wand}(\Gamma, \lambda x.M, n_0) = (\text{Some } \{\text{Tvar}(n_0) \stackrel{c}{=} \text{Tvar}(n_0+1) \rightarrow \text{Tvar}(n_0+2)\} \cup \mathbb{C}, n_1)} \text{ (W-Abs)}$$

$$\frac{\text{Wand}(\Gamma, M, n_0 + 1) = (\text{Some } \mathbb{C}', n_1) \qquad \text{Wand}(\Gamma, N, n_1) = (\text{Some } \mathbb{C}'', n_2)}{\text{Wand}(\Gamma, MN, n_0) = (\text{Some } \{\text{Tvar}(n_0+1) \stackrel{c}{=} \text{Tvar}(n_1) \rightarrow \text{Tvar}(n_0)\} \cup \mathbb{C}' \cup \mathbb{C}'', n_2)} \text{ (W-App)}$$

Overview

- Correctness is given w.r.t the Hindley-Milner type system:

$$\frac{\langle x, \tau \rangle \in \Gamma \text{ is the leftmost binding of x in `}}{\Gamma \ \triangleright x : \tau} \qquad \text{(HM-Var)}$$

$$\frac{(x, \tau) :: \Gamma \ \triangleright \ M : \tau'}{\Gamma \ \triangleright \ \lambda x.M \ : \ \tau \rightarrow \tau'} \qquad \text{(HM-Abs)}$$

$$\frac{\Gamma \ \triangleright \ M : \tau' \rightarrow \tau \qquad \Gamma \ \triangleright \ N : \tau'}{\Gamma \ \triangleright \ MN : \tau} \qquad \text{(HM-App)}$$

Soundness Proof

Informally

If Wand's algorithm returns a unifiable constraint set, then there is a Hindley-Milner proof.

Our Statement

$\forall \Gamma, \forall M, \forall \sigma, \forall n, \forall n', \forall \mathbb{C}.$
$\text{Wand}(\Gamma, M, n) = (\text{Some } \mathbb{C},\ n') \ \wedge\ \text{unify } \mathbb{C} = \text{Some } \sigma$
$\quad \Rightarrow \vdash \sigma(\Gamma) \rhd_{HM} M : \sigma(\tau)$

Wand's Statement

$$\forall \sigma . \sigma \models (E, G) \Rightarrow \vdash \sigma(\Gamma_0) \rhd_{HM} M_0 : \sigma(\tau_0)$$

Completeness Proof

Informally

If there is a Hindley-Milner proof (that a term has some type), then Wand's algorithm returns a solvable constraint set that will return the given type.

Our Statement

$$
\begin{array}{l}
\forall \Gamma', \forall M, \forall \tau. \\
\;\vdash \Gamma' \rhd_{HM} M : \tau \\
\qquad \Rightarrow \forall \Gamma, \forall n. (\exists \sigma.\ \sigma(\Gamma) = \Gamma') \;\wedge\; \text{fresh\_env } n\ \Gamma \\
\qquad\qquad \Rightarrow \forall \mathbb{C}, \forall n'. \text{Wand}(\Gamma, M, n) = (\text{Some } \mathbb{C}, n') \wedge \\
\qquad\qquad\quad \exists \sigma'. \text{unify } \mathbb{C} = \text{Some } \sigma' \\
\qquad\qquad\qquad \Rightarrow \exists \sigma''. (\sigma' \circ \sigma'')(\text{Tvar}(n)) = \tau \wedge \\
\qquad\qquad\qquad\qquad (\sigma' \circ \sigma'')(\Gamma) = \Gamma'
\end{array}
$$

Wand's Statement

$$\vdash \Gamma \rhd_{HM} M_0 : \tau \Rightarrow (\exists \rho.\ \rho \models (E, G) \wedge \Gamma = \rho\Gamma_0 \wedge \tau = \rho\tau_0)$$

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:

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

MGU Axioms

Old Axioms

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

MGU Axioms

Old Axioms

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

New Generalized Axioms

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

Functional Induction in Coq

- Axioms proved in Coq [KC09].
- Important first step in proof of the axioms.
- Requires an induction principle generated before.

Functional Induction in Coq

- Axioms proved in Coq [KC09].
- Important first step in proof of the axioms.
- 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

- Axioms proved in Coq [KC09].
- Important first step in proof of the axioms.
- 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)` $\rightsquigarrow$ `induction c (unify c) using unif_ind`.

Outline

Conclusions and Future Work

- Used Coq's finite maps library to represent substitution.
- MGU is not axiomatized in our verification.
- Completeness is work in progress, but so far 8000 lines of Coq tactics and specification.
- The final goal is to have a machine certified correctness proof of our extension of Wand's algorithm to polymorphic let.

- 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.

- J. Roger Hindley and Jonathan P. Seldin.
  Introduction to Combinators and λ-Calculus.
  Cambridge University Press, 1986.

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

- Sunil Kothari and James Caldwell.

A machine checked model of MGU axioms: applications of finite maps and functional induction.
In *Proceedings of the 23rd International Workshop on Unification*, pages 17–31, 2009.

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.

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.

- Andrew K. Wright and Matthias Felleisen.
  A syntactic approach to type soundness.
  Information and Computation, 115(1):38–94, 1994.