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

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
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
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 Algorithms

Examples
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].
Machine-Certified Correctness Proof

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

Type Reconstruction Algorithms

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.
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

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

Terms

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

Constraints

- Constraint are of the form \( \tau^c \vdash \tau' \).
Substitution

- A *substitution* (denoted by $\sigma$) maps type variables to types.
Substitution

- A substitution (denoted by $\sigma$) maps type variables to types.

Unifier

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

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

**Unifier**

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

- A substitution (denoted by $\sigma$) maps type variables to types.

Unifier

- We write $\sigma \models (\tau_1 \equiv \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$.

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

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

- **Input.** A term $M$ of $\Lambda$.
- **Initialization.** Set $E = \emptyset$ and $G = \{(\Gamma, M, \alpha_0)\}$.
- **Loop Step.** If $G = \emptyset$ then return $E$ else choose a subgoal $(\Gamma, M, \tau)$ 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 $(\Gamma, x, \tau)$. Generate the equation $\tau \cata{c} \Gamma(x)$. 

Wand’s Algorithm

Action Table

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

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

Kothari Caldwell (U. of Wyoming)  
Machine-certified proof of Wand’s type reconstrunction  
WMM’09 11 / 31
Wand’s Algorithm

Action Table

Case \((\Gamma, x, \tau)\). Generate the equation \(\tau \triangleq \Gamma(x)\).

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

Case \((\Gamma, \lambda x . M, \tau)\). Generate equation \(\tau \triangleq \tau' \rightarrow \tau''\) and subgoal 
\([x : \tau'] :: \Gamma, M, \tau''\).
Wand’s Algorithm - Example

\[
\begin{align*}
\{ & (\emptyset, \lambda x.\lambda y.\lambda z.xz(yz), \alpha_0) \}; \{ \} \\
\{ & ((x : \alpha_1), \lambda y.\lambda z.xz(yz), \alpha_2) \}; \{ \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \} \\
\{ & ((x : \alpha_1, y : \alpha_3), \lambda z.xz(yz), \alpha_4) \}; \{ \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \} \\
\{ & ((x : \alpha_1, y : \alpha_3, z : \alpha_5), xz(yz), \alpha_6) \}; \{ \alpha_4 \rightarrow \alpha_5 \rightarrow \alpha_6 \} \\
\{ & (((x : \alpha_1, z : \alpha_5), xz, \alpha_7 \rightarrow \alpha_6), ((y : \alpha_3, z : \alpha_5), yz, \alpha_7)) \}; \{ \} \\
\{ & ((x : \alpha_1), x, \alpha_8 \rightarrow (\alpha_7 \rightarrow \alpha_6)), ((z : \alpha_5), z, \alpha_8), ((y : \alpha_3, z : \alpha_5), yz, \alpha_7)) \}; \{ \} \\
\{ & (((z : \alpha_5), z, \alpha_8), ((y : \alpha_3, z : \alpha_5), yz, \alpha_7)) \}; \{ \alpha_1 \rightarrow \alpha_8 \rightarrow \alpha_7 \rightarrow \alpha_6 \} \\
\{ & ((y : \alpha_3, z : \alpha_5), yz, \alpha_7)) \}; \{ \alpha_8 \rightarrow \alpha_5 \} \\
\{ & ((y : \alpha_3), y, \alpha_9 \rightarrow \alpha_7), ((z : \alpha_5), z, \alpha_9)) \}; \{ \} \\
\{ & ((z : \alpha_5), z, \alpha_9)) \}; \{ \alpha_9 \rightarrow \alpha_7 \rightarrow \alpha_3 \} \\
\emptyset; \{ \alpha_9 \rightarrow \alpha_5 \}
\end{align*}
\]
Wand’s Algorithm Example - Alternate View

\[
\begin{align*}
\{ & \alpha_8 \rightarrow \alpha_7 \rightarrow \alpha_6 \equiv \alpha_1 \} \\
\{ & x : \alpha_1 \} \vdash x : \alpha_8 \rightarrow \alpha_7 \rightarrow \alpha_6 \\
\{ & z : \alpha_5 \} \vdash z : \alpha_8 \\
\{ & \alpha_8 \equiv \alpha_5 \} \\
\{ & y : \alpha_3 \} \vdash y : \alpha_9 \rightarrow \alpha_7 \\
\{ & \alpha_9 \equiv \alpha_3 \} \\
\{ & z : \alpha_5 \} \vdash z : \alpha_9 \\
\{ & \alpha_9 \equiv \alpha_5 \}
\end{align*}
\]

\[
\begin{align*}
\{ & x : \alpha_1, z : \alpha_5 \} \vdash xz : \alpha_7 \rightarrow \alpha_6 \\
\{ & y : \alpha_3, z : \alpha_5 \} \vdash yz : \alpha_7 \\
\{ & \alpha_4 \equiv \alpha_5 \rightarrow \alpha_6 \} \\
\{ & x : \alpha_1, y : \alpha_3 \} \vdash \lambda z. xz(yz) : \alpha_4 \\
\{ & \alpha_4 \equiv \alpha_5 \rightarrow \alpha_6 \} \\
\{ & x : \alpha_1 \} \vdash \lambda y. \lambda z. xz(yz) : \alpha_2 \\
\{ & \alpha_2 \equiv \alpha_3 \rightarrow \alpha_4 \} \\
\{ & \alpha_0 \equiv \alpha_1 \rightarrow \alpha_2 \} \\
\{ & \} \vdash \lambda x. \lambda y. \lambda z. xz(yz) : \alpha_0
\end{align*}
\]
Example - Solution

\[
\begin{align*}
\alpha_0 & \overset{c}{=} \alpha_1 \rightarrow \alpha_2 \\
\alpha_2 & \overset{c}{=} \alpha_3 \rightarrow \alpha_4 \\
\alpha_4 & \overset{c}{=} \alpha_5 \rightarrow \alpha_9 \\
\alpha_1 & \overset{c}{=} \alpha_8 \rightarrow \alpha_7 \rightarrow \alpha_9 \\
\alpha_8 & \overset{c}{=} \alpha_5 \\
\alpha_9 & \rightarrow \alpha_7 \overset{c}{=} \alpha_3 \\
\alpha_9 & \overset{c}{=} \alpha_5 \\
\end{align*}
\]

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.
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 $\tau$ is defined as:

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

$$\sigma (\tau_1 \rightarrow \tau_2) \overset{\text{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2)$$
Substitution

Related Concepts

- Substitution application to a type $\tau$ is defined as:

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

  $\sigma (\tau_1 \rightarrow \tau_2) \overset{\text{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2)$

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

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

- Assumption: Idempotent 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

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
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.

\[
\text{search}_\text{type}_\text{env}(x, \Gamma) = \text{Some } \tau \\
Wand(\Gamma, x, n_0) = (\text{Some } \{\text{Tvar}(n_0) \triangleq \tau\}, n_0 + 1) \quad (\text{W-Var})
\]

\[
Wand((x : \text{Tvar}(n_0 + 1)) :: \Gamma), M, n_0 + 2) = (\text{Some } C, n_1) \\
Wand(\Gamma, \lambda x. M, n_0) = (\text{Some } \{\text{Tvar}(n_0) \triangleright \text{Tvar}(n_0 + 1) \rightarrow \text{Tvar}(n_0 + 2)\} \cup C, n_1) \quad (\text{W-Abs})
\]

\[
Wand(\Gamma, M, n_0 + 1) = (\text{Some } C', n_1) \\
Wand(\Gamma, N, n_1) = (\text{Some } C'', n_2) \\
Wand(\Gamma, MN, n_0) = (\text{Some } \{\text{Tvar}(n_0 + 1) \triangleright \text{Tvar}(n_1) \rightarrow \text{Tvar}(n_0)\} \cup C' \cup C'', n_2) \quad (\text{W-App})
\]
Wand’s Algorithm

Issues in formalization

- Freshness is now explicit.

\[
\text{search}_\text{type}_\text{env}(x, \Gamma) = \text{Some } \tau
\]

\[
\text{Wand}(\Gamma, x, n_0) = (\text{Some } \{\text{Tvar}(n_0) \triangleq \tau\}, n_0 + 1) \quad \text{(W-Var)}
\]

\[
\text{Wand}(((x : \text{Tvar}(n_0 + 1)) :: \Gamma), M, n_0 + 2) = (\text{Some } \mathbb{C}, n_1)
\]

\[
\text{Wand}(\Gamma, \lambda x.M, n_0) = (\text{Some } \{\text{Tvar}(n_0) \triangleq \text{Tvar}(n_0 + 1) \to \text{Tvar}(n_0 + 2)\} \cup \mathbb{C}, n_1) \quad \text{(W-Abs)}
\]

\[
\text{Wand}(\Gamma, M, n_0 + 1) = (\text{Some } \mathbb{C}', n_1) \quad \text{Wand}(\Gamma, N, n_1) = (\text{Some } \mathbb{C}'', n_2)
\]

\[
\text{Wand}(\Gamma, MN, n_0) = (\text{Some } \{\text{Tvar}(n_0 + 1) \triangleq \text{Tvar}(n_1) \to \text{Tvar}(n_0)\} \cup \mathbb{C}' \cup \mathbb{C}'', n_2) \quad \text{(W-App)}
\]
Wand’s Algorithm

Issues in formalization

- The W-App rule now generates a constraint.

\[
\text{search	extunderscore type	extunderscore env}(x, \Gamma) = \text{Some } \tau \\
\text{Wand}(\Gamma, x, n_0) = (\text{Some } \{ \text{Tvar}(n_0) \triangleq \tau \}, n_0 + 1) \quad \text{(W-Var)}
\]

\[
\text{Wand}(((x : \text{Tvar}(n_0 + 1)) :: \Gamma), M, n_0 + 2) = (\text{Some } C, n_1) \\
\text{Wand}(\Gamma, \lambda x. M, n_0) = (\text{Some } \{ \text{Tvar}(n_0) \triangleq \text{Tvar}(n_0 + 1) \to \text{Tvar}(n_0 + 2) \} \cup C, n_1) \quad \text{(W-Abs)}
\]

\[
\text{Wand}(\Gamma, M, n_0 + 1) = (\text{Some } C', n_1) \quad \text{Wand}(\Gamma, N, n_1) = (\text{Some } C'', n_2) \\
\text{Wand}(\Gamma, MN, n_0) = (\text{Some } \{ \text{Tvar}(n_0 + 1) \triangleq \text{Tvar}(n_1) \to \text{Tvar}(n_0) \} \cup C' \cup C'', n_2) \quad \text{(W-App)}
\]
Overview

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

\[ \langle x, \tau \rangle \in \Gamma \text{ is the leftmost binding of } x \text{ in } \Gamma \]

\[ \Gamma \triangleright x : \tau \quad \text{(HM-Var)} \]

\[ (x, \tau) :: \Gamma \triangleright M : \tau' \]

\[ \Gamma \triangleright \lambda x. M : \tau \rightarrow \tau' \quad \text{(HM-Abs)} \]

\[ \Gamma \triangleright M : \tau' \rightarrow \tau \quad \Gamma \triangleright N : \tau' \]

\[ \Gamma \triangleright MN : \tau \quad \text{(HM-App)} \]
Informally
If Wand’s algorithm returns a unifiable constraint set, then there is a Hindley-Milner proof.

Our Statement
\[ \forall \Gamma, \forall M, \forall \sigma, \forall n, \forall n', \forall C. \]
\[ \text{Wand}(\Gamma, M, n) = (\text{Some } C, \ n') \land \text{unify } C = \text{Some } \sigma \]
\[ \Rightarrow \vdash \sigma(\Gamma) \triangleright_{HM} M : \sigma(\tau) \]

Wand’s Statement
\[ \forall \sigma. \sigma \models (E, G) \Rightarrow \vdash \sigma(\Gamma_0) \triangleright_{HM} M_0 : \sigma(\tau_0) \]
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

∀Γ′, ∀M, ∀τ.
⊢ Γ′ ⊨_{HM} M : τ
⇒ ∀Γ, ∀n.(∃σ. σ(Γ) = Γ′) ∧ fresh_env n Γ
⇒ ∀C, ∀n′. Wand(Γ, M, n) = (Some C, n′) ∧
∃σ′. unify C = Some σ′
⇒ ∃σ''. ((σ′ ◦ σ'')(Tvar(n))) = τ ∧
(σ′ ◦ σ'')(Γ) = Γ′

Wand’s Statement

⊢ Γ ⊨_{HM} M_0 : τ ⇒ (∃ρ. ρ |= (E, G) ∧ Γ = ρΓ_0 ∧ τ = ρτ_0)
The *most general unifier* (MGU) is often a first-order unification algorithm over simple type terms.
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:

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

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

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

(iv) \( \sigma(\tau_1) = \sigma(\tau_2) \Rightarrow \exists \sigma'. \text{mgu} \, \sigma'(\tau_1 \equiv \tau_2) \)
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) \[ \sigma \models C \Rightarrow \exists \sigma'. \text{unify } C = \text{Some } \sigma' \]
Axioms proved in Coq [KC09].
Important first step in proof of the axioms.
Requires an induction principle generated before.
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**.
Axioms proved in Coq [KC09].

Important first step in proof of the axioms.

Requires an induction principle generated before.

**functional induction** \((f \ x_1 \ x_2 \ x_3 \ldots \ x_n)\) is a short form for \(\text{induction } x_1 \ x_2 \ x_3 \ldots \ x_n \ f(x_1 \ldots \ x_n)\) using \textit{id}, where \textit{id} is the induction principle for \(f\).

**functional induction** \((\text{unify } c) \leadsto \text{induction } c\) \((\text{unify } c)\) using \textit{unif_ind}.
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. 

Bastiaan Heeren. 
*Top Quality Type Error Messages.* 

J. Roger Hindley and Jonathan P. Seldin. 
*Introduction to Combinators and λ-Calculus.* 

Mark P. Jones. 
*Qualified Types: Theory and Practice.* 

Sunil Kothari and James Caldwell.


F. Pottier and D. Rémy. The essence of ML type inference.

Conclusions and Future Work


