Principles of Programming Languages

Lecture 05 Types

Albert-Ludwigs-Universität Freiburg

Peter Thiemann
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

28 May 2018

UNI FREIBURG

Plan

1. Type Inference for the Simply-Typed Lambda Calculus
2. ML-Style Polymorphic Types
3. Type Inference for Mini-ML

Type Inference for the Simply-Typed Lambda Calculus (STLC)

Typing Problems

Type checking: Given environment Γ, a term e and a type τ, is $\Gamma \vdash e : \tau$ derivable?

Type inference: Given a term e, are there Γ and τ such that $\Gamma \vdash e : \tau$ is derivable?

Type Inference for the Simply-Typed Lambda Calculus (STLC)

Typing Problems

Type checking: Given environment Γ, a term e and a type τ, is $\Gamma \vdash e : \tau$ derivable?

Type inference: Given a term e, are there Γ and τ such that $\Gamma \vdash e : \tau$ is derivable?

Typing Problems for STLC

- Type checking and type inference are decidable for STLC
- Moreover, for each typable e there is a *principal typing* $\Gamma \vdash e : \tau$ such that any other typing is a substitution instance of the principal typing

Substitution

- A *(type) substitution* is a finite map S from type variables to types such that $\mathit{dom}(S) \cap \mathit{Var}(S\alpha) = \emptyset$, for all type variables α
- A substitution extends to a type τ by applying it to all variables in τ

Substitution

- A *(type) substitution* is a finite map S from type variables to types such that $\mathit{dom}(S) \cap \mathit{Var}(S\alpha) = \emptyset$, for all type variables α
- A substitution extends to a type τ by applying it to all variables in τ

Substitution

- $S = \{\alpha \mapsto \mathtt{Int}, \beta \mapsto \mathtt{Int} \to \mathtt{Int}\}$
- $\tau = \alpha \to \alpha \Rightarrow S\tau = \mathtt{Int} \to \mathtt{Int} = S\beta$

Let $\mathcal{E} ::= \emptyset \mid \tau \doteq \tau', \mathcal{E}$ be a set of equations on types.

Unifiers and Most General Unifiers

- A substitution S is a *unifier of* $\mathcal{E}$ if, for each $\tau \doteq \tau' \in \mathcal{E}$, it holds that $S\tau = S\tau'$.
- A substitution S is a *most general unifier of* $\mathcal{E}$ if S is a unifier of $\mathcal{E}$ and for every other unifier S' of $\mathcal{E}$, there is a substitution T such that $S' = T \circ S$.

Let $\mathcal{E} ::= \emptyset \mid \tau \doteq \tau', \mathcal{E}$ be a set of equations on types.

Unifiers and Most General Unifiers

- A substitution S is a *unifier of* $\mathcal{E}$ if, for each $\tau \doteq \tau' \in \mathcal{E}$, it holds that $S\tau = S\tau'$.
- A substitution S is a *most general unifier of* $\mathcal{E}$ if S is a unifier of $\mathcal{E}$ and for every other unifier S' of $\mathcal{E}$, there is a substitution T such that $S' = T \circ S$.

Unification

There is an algorithm $\mathcal{U}$ that, on input of $\mathcal{E}$, either returns a most general unifier of $\mathcal{E}$ or fails if none exists.

Unification Algorithm

Original Algorithm [Robinson 1965]

Simple, but exponential complexity.

Unification Algorithm

Original Algorithm [Robinson 1965]

Simple, but exponential complexity.

Algorithm by Martelli and Montanari

Based on rewriting $\mathcal{E}$

- **delete** $\mathcal{E} \cup \{\tau \doteq \tau\} \Rightarrow \mathcal{E}$
- **decompose** $\mathcal{E} \cup \{\tau_1 \to \tau_2 \doteq \tau_1' \to \tau_2'\} \Rightarrow \mathcal{E} \cup \{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\}$
- **conflict** $\mathcal{E} \cup \{\tau \doteq \tau'\} \Rightarrow \bot$ if τ and τ' are both not variables and start with a different type constructor
- **swap** $\mathcal{E} \cup \{\tau \doteq \alpha\} \Rightarrow \mathcal{E} \cup \{\alpha \doteq \tau\}$ if τ is not a variable
- **eliminate** $\mathcal{E} \cup \{\alpha \doteq \tau\} \Rightarrow \mathcal{E}[\alpha \mapsto \tau] \cup \{\alpha \doteq \tau\}$ if $\alpha \notin \mathit{Var}(\tau)$ and $\alpha \in \mathit{Var}(\mathcal{E})$
- **check** $\mathcal{E} \cup \{\alpha \doteq \tau\} \Rightarrow \bot$ if $\alpha \in \mathit{Var}(\tau)$

Unification Example

$$
\begin{array}{lll}
\texttt{Int} \to \alpha \doteq \beta & \gamma \to (\texttt{Int} \to \texttt{Int}) \doteq \beta & \\
\quad \beta \doteq \texttt{Int} \to \alpha & \gamma \to (\texttt{Int} \to \texttt{Int}) \doteq \beta & \\
\quad \beta \doteq \texttt{Int} \to \alpha & \gamma \to (\texttt{Int} \to \texttt{Int}) \doteq \texttt{Int} \to \alpha & \\
\quad \beta \doteq \texttt{Int} \to \alpha & \gamma \doteq \texttt{Int} & \texttt{Int} \to \texttt{Int} \doteq \alpha \\
\quad \beta \doteq \texttt{Int} \to \alpha & \gamma \doteq \texttt{Int} & \alpha \doteq \texttt{Int} \to \texttt{Int} \\
\quad \beta \doteq \texttt{Int} \to (\texttt{Int} \to \texttt{Int}) & \gamma \doteq \texttt{Int} & \alpha \doteq \texttt{Int} \to \texttt{Int}
\end{array}
$$

Principal Type Inference for STLC

The algorithm (due to John Mitchell) transforms a term into a principal typing judgment for the term or fails if no typing exists.

$$
\begin{array}{lcl}
\mathcal{P}(x) & = & \textbf{return } x : \alpha \vdash x : \alpha \\
\mathcal{P}(\lambda x\,.\,e) & = & \textbf{let } \Gamma \vdash e : \tau \leftarrow \mathcal{P}(e) \textbf{ in} \\
 & & \textbf{if } x : \tau_x \in \Gamma \textbf{ then return } \Gamma_x \vdash \lambda x\,.\,e : \tau_x \to \tau \\
 & & \textbf{else choose } \alpha \notin \mathit{Var}(\Gamma, \tau) \textbf{ in} \\
 & & \quad\quad \textbf{return } \Gamma \vdash \lambda x\,.\,e : \alpha \to \tau \\
\mathcal{P}(e_0\ e_1) & = & \textbf{let } \Gamma_0 \vdash e_0 : \tau_0 \leftarrow \mathcal{P}(e_0) \textbf{ in} \\
 & & \textbf{let } \Gamma_1 \vdash e_1 : \tau_1 \leftarrow \mathcal{P}(e_1) \textbf{ in} \\
 & & \textbf{with disjoint type variables in } (\Gamma_0, \tau_0) \textbf{ and } (\Gamma_1, \tau_1) \\
 & & \textbf{choose } \alpha \notin \mathit{Var}(\Gamma_0, \Gamma_1, \tau_0, \tau_1) \textbf{ in} \\
 & & \textbf{let } S \leftarrow \mathcal{U}(\Gamma_0 \doteq \Gamma_1, \tau_0 \doteq \tau_1 \to \alpha) \textbf{ in} \\
 & & \textbf{return } S\Gamma_0 \cup S\Gamma_1 \vdash e_0\ e_1 : S\alpha \\
\mathcal{P}(\lceil n \rceil) & = & \textbf{return } \cdot \vdash \lceil n \rceil : \mathbf{N} \\
\mathcal{P}(\mathit{succ}\ e) & = & \textbf{let } \Gamma \vdash e : \tau \leftarrow \mathcal{P}(e) \textbf{ in} \\
 & & \textbf{let } S \leftarrow \mathcal{U}(\tau \doteq \mathbf{N}) \textbf{ in} \\
 & & \textbf{return } S\Gamma \vdash \mathit{succ}\ e : \mathbf{N}
\end{array}
$$

An example run of $\mathcal{P}$

Plan

ML-Style Polymorphic Types

- Simple types are restrictive
- Example:

$$(\lambda i\,.\,(i\ (\lambda y\,.\,\mathit{succ}\ y))\ (i\ 42))\ (\lambda x\,.\,x)$$

  - $\lambda x\,.\,x : \alpha \to \alpha$
  - $i\ 42$ requires $i : \mathbf{N} \to \beta$
  - $i\ (\lambda y\,.\,\mathit{succ}\ y)$ requires $i : (\mathbf{N} \to \mathbf{N}) \to \gamma$
  - Unification of the assumption on i fails: term has no simple type
  - However, term evaluates without error
- Insufficient modularity

Applied Mini-ML

Syntax

$$\begin{array}{rcl} \mathrm{Exp} \ni e & ::= & x \mid \lambda x\,.\,e \mid e\ e \mid \mathit{let}\ x = e\ \mathit{in}\ e \mid \lceil n \rceil \mid \mathit{succ}\ e \\ \mathrm{Val} \ni v & ::= & \lambda x\,.\,e \mid \lceil n \rceil \end{array}$$

Evaluation (Call-by-Value)

Beta-V
$$(\lambda x\,.\,e)\ v \rightarrow_v e[x \mapsto v]$$

AppL
$$\frac{f \rightarrow_v f'}{f\ e \rightarrow_v f'\ e}$$

VAppR
$$\frac{e \rightarrow_v e'}{v\ e \rightarrow_v v\ e'}$$

LetL
$$\frac{e \rightarrow_v e'}{\mathit{let}\ x = e\ \mathit{in}\ f \rightarrow_v \mathit{let}\ x = e'\ \mathit{in}\ f}$$

Beta-Let
$$\mathit{let}\ x = v\ \mathit{in}\ e \rightarrow_v e[x \mapsto v]$$

SuccL
$$\frac{e \rightarrow_v e'}{\mathit{succ}\ e \rightarrow_v \mathit{succ}\ e'}$$

Delta
$$\frac{e \rightarrow_\delta e'}{e \rightarrow_v e'}$$

Types for Applied Mini-ML

Syntax of Types

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \tau \to \tau \mid \mathtt{Int} & \text{Types} \\
\sigma & ::= & \tau \mid \forall\alpha.\sigma & \text{Type Schemes} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \sigma & \text{Type Environments}
\end{array}
$$

The type scheme $\forall\alpha.\sigma$ ...

- *binds* type variable α
- can be *instantiated* by substituting a type for α in σ
- only appears in the type environment

Operations on Type Schemes

Instance

$\sigma = \forall\alpha_1 \dots \alpha_m.\tau$ has an *instance* τ', written as $\sigma \succeq \tau'$, if there is a substitution S with $dom(S) \subseteq \{\alpha_1, \dots, \alpha_m\}$ such that $\tau' = S\tau$.

Generalization

$$GEN(\Gamma, \tau) = \forall\alpha_1 \dots \alpha_m.\tau$$

where $\{\alpha_1, \dots, \alpha_m\} = FV(\tau) \setminus FV(\Gamma)$.

Inference Rules for Mini-ML

Var
$$\frac{\sigma \succeq \tau}{\Gamma, x : \sigma \vdash x : \tau}$$

Lam
$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x \,.\, e : \tau \to \tau'}$$

App
$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'}$$

Let
$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathit{GEN}(\Gamma, \tau) \vdash e_1 : \tau'}{\Gamma \vdash \mathit{let}\ x = e_0\ \mathit{in}\ e_1 : \tau'}$$

Num
$$\Gamma \vdash \lceil n \rceil : \mathtt{Int}$$

Succ
$$\frac{\Gamma \vdash e : \mathtt{Int}}{\Gamma \vdash \mathit{succ}\ e : \mathtt{Int}}$$

Example Revisited

$$let\ i = \lambda x\,.\,x\ in\ (i\ (\lambda y\,.\,succ\ y))\ (i\ 42)$$

- $\lambda x\,.\,x : \alpha \rightarrow \alpha$
- Generalized binding: $i : \forall\alpha.\alpha \rightarrow \alpha$
- i 42 using instance $\texttt{Int} \rightarrow \texttt{Int}$
- $i\ (\lambda y\,.\,succ\ y)$ using instance $(\texttt{Int} \rightarrow \texttt{Int}) \rightarrow (\texttt{Int} \rightarrow \texttt{Int})$
- Type checking succeeds
- Type checking the uses of i is better decoupled from i's definition $\Rightarrow$ improved modularity

Properties

- Type soundness
- Decidable type checking and type inference (upcoming)
- Basis for type system of ML, Haskell, and other languages
- Numerous extensions

Plan

Type Inference for Mini-ML

The algorithm $\mathcal{W}(\Gamma; e)$ transforms a type environment Γ and a term e into a pair (S, τ) of a substitution and a type (or fails if no typing exists).
This algorithm is the traditional Hindley-Milner type inference algorithm.

Mini-ML Type Inference Algorithm, Part I

$$
\begin{array}{lcl}
\mathcal{W}(\Gamma; x) & = & \textbf{let } \forall\alpha_1 \dots \alpha_m.\tau = \Gamma(x) \\
& & \beta_1 \dots \beta_m \leftarrow \textbf{fresh} \\
& & \textbf{return } (ID, \tau[\alpha_i \mapsto \beta_i]) \\
\mathcal{W}(\Gamma; \lambda x\,.\,e) & = & \beta \leftarrow \textbf{fresh} \\
& & (S, \tau) \leftarrow \mathcal{W}(\Gamma, x : \beta; e) \\
& & \textbf{return } (S, S\beta \rightarrow \tau) \\
\mathcal{W}(\Gamma; e_0\ e_1) & = & (S_0, \tau_0) \leftarrow \mathcal{W}(\Gamma; e_0) \\
& & (S_1, \tau_1) \leftarrow \mathcal{W}(S_0\Gamma; e_1) \\
& & \beta \leftarrow \textbf{fresh} \\
& & T \leftarrow \mathcal{U}(S_1\tau_0 \doteq \tau_1 \rightarrow \beta) \\
& & \textbf{return } (T \circ S_1 \circ S_0, T\beta) \\
\mathcal{W}(\Gamma; let\ x = e_0\ in\ e_1) & = & (S_0, \tau_0) \leftarrow \mathcal{W}(\Gamma; e_0) \\
& & \textbf{let } \sigma = GEN(S_0\Gamma, \tau_0) \\
& & (S_1, \tau_1) \leftarrow \mathcal{W}(S_0\Gamma, x : \sigma; e_1) \\
& & \textbf{return } (S_1 \circ S_0, \tau_1)
\end{array}
$$

Mini-ML Type Inference Algorithm, Part II

$$
\begin{array}{rcl}
\mathcal{W}(\Gamma; \lceil n \rceil) & = & \textbf{return } (\mathit{ID}, \texttt{Int}) \\
\mathcal{W}(\Gamma; \mathit{succ}\ e) & = & (S, \tau) \leftarrow \mathcal{W}(\Gamma; e) \\
 & & \textbf{let } T \leftarrow \mathcal{U}(\tau \doteq \texttt{Int}) \textbf{ in} \\
 & & \textbf{return } (T \circ S, \texttt{Int})
\end{array}
$$

Properties of Type Inference for Mini-ML

Soundness

If $\mathcal{W}(\Gamma; e) = \textbf{return } (S, \tau)$, then $S\Gamma \vdash e : \tau$.

Completeness

If $S\Gamma \vdash e : \tau'$, then $\mathcal{W}(\Gamma; e) = \textbf{return } (T, \tau)$ such that $S = S' \circ T$ and $\tau' = S'\tau$.