Functional Programming

Polymorphic Types

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

ML-Style Polymorphic Types

- Simple types are restrictive and offer insufficient modularity
- Example:

$$(\lambda i.(i\,(\lambda y.SUCC\,y))\,(i\,42))\,(\lambda x.x)$$

  - $\lambda x.x : \alpha \to \alpha$
  - $i\,42$ requires $i : Nat \to \beta$
  - $i\,(\lambda y.SUCC\,y)$ requires $i : (Nat \to Nat) \to \gamma$
  - Unification of the assumption on i fails: term has no simple type
  - However, term evaluates without error

Applied Mini-ML

Syntax

$$\begin{array}{rcl} \text{Exp} \ni \quad e, f & ::= & x \mid \lambda x.e \mid f\ e \mid let\, x = e\ in\, f \mid n \mid SUCC\, e \\ \text{Val} \ni \quad v & ::= & \lambda x.e \mid n \end{array}$$

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

Beta-Let
$$let\, x = v\ in\, e \rightarrow_v e[x \mapsto v]$$

SuccL
$$\frac{e \rightarrow_v e'}{SUCC\, e \rightarrow_v SUCC\, e'}$$

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

Types for Applied Mini-ML

Syntax of Types

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

A 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

Generic Instance

$\sigma = \forall \alpha_1 \dots \alpha_m.\tau$ has a **generic instance** $\sigma' = \forall \beta_1 \dots \beta_n.\tau'$, written as $\sigma \succeq \sigma'$, if for all i, $\beta_i \notin \mathit{fv}(\sigma)$ and there is a substitution S with $\mathit{dom}(S) \subseteq \{\alpha_1, \dots, \alpha_m\}$ such that $\tau' = S\tau$.

Examples

$$\forall \alpha.\alpha \to \alpha \succeq \mathit{Nat} \to \mathit{Nat} \qquad \forall \alpha\beta.\alpha \to \beta \to \alpha \succeq \forall \alpha.\alpha \to \alpha \to \alpha$$

Operations on Type Schemes

Generic Instance

$\sigma = \forall \alpha_1 \dots \alpha_m . \tau$ has a **generic instance** $\sigma' = \forall \beta_1 \dots \beta_n . \tau'$, written as $\sigma \succeq \sigma'$, if for all i, $\beta_i \notin \mathit{fv}(\sigma)$ and there is a substitution S with $\mathit{dom}(S) \subseteq \{\alpha_1, \dots, \alpha_m\}$ such that $\tau' = S\tau$.

Examples

$$\forall \alpha . \alpha \to \alpha \succeq \mathit{Nat} \to \mathit{Nat} \qquad \forall \alpha\beta . \alpha \to \beta \to \alpha \succeq \forall \alpha . \alpha \to \alpha \to \alpha$$

Generalization

$$\mathit{gen}(A, \tau) = \forall \alpha_1 \dots \alpha_m . \tau$$

where $\{\alpha_1, \dots, \alpha_m\} = \mathit{fv}(\tau) \setminus \mathit{fv}(A)$.

Inference Rules for Mini-ML

$$\textsc{Var}\quad \frac{\sigma \succeq \tau}{A, x : \sigma \vdash x : \tau}$$

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

$$\textsc{App}\quad \frac{A \vdash e : \tau \to \tau' \qquad A \vdash f : \tau}{A \vdash e\, f : \tau'}$$

$$\textsc{Let}\quad \frac{A \vdash e : \tau \qquad A, x : gen(A, \tau) \vdash f : \tau'}{A \vdash let\, x = e\, in\, f : \tau'}$$

$$\textsc{Num}\quad A \vdash n : Nat$$

$$\textsc{Succ}\quad \frac{A \vdash e : Nat}{A \vdash SUCC\, e : Nat}$$

Example Revisited

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

- $\lambda x.x : \alpha \to \alpha$
- Generalized binding: $i : \forall\alpha.\alpha \to \alpha$
- $i\, 42$ using instance $\forall\alpha.\alpha \to \alpha \succeq Nat \to Nat$
- $i\, (\lambda y.SUCC\, y)$ using instance $\forall\alpha.\alpha \to \alpha \succeq (Nat \to Nat) \to (Nat \to Nat)$
- Type checking succeeds
- Type checking the uses of i is better decoupled from i's definition $\Rightarrow$ modularity improved

Properties

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

Type Inference for Mini-ML

Type Inference for Mini-ML

The algorithm $\mathcal{W}(A; e)$ transforms a type environment A 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.
See: Milner, Robin (1978). A Theory of Type Polymorphism in Programming. JCSS, 17: 348–375

Mini-ML Type Inference Algorithm, Part I

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

Mini-ML Type Inference Algorithm, Part II

$$
\begin{array}{lcl}
\mathcal{W}(A; n) & = & \textbf{return}\ (ID, Nat) \\
\mathcal{W}(A; SUCC\, e) & = & (S, \tau) \leftarrow \mathcal{W}(A; e) \\
 & & \textbf{let}\ \ T \leftarrow \mathcal{U}(\tau \doteq Nat)\ \textbf{in} \\
 & & \textbf{return}\ (T \circ S, Nat)
\end{array}
$$

Properties of Type Inference for Mini-ML

Soundness

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

Completeness

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

Principal types

Completeness implies that $\mathcal{W}$ computes **principal types** because all other types of the same term are instances of the computed type.

Wrapup

- ML polymorphism is based on type schemes
- Type checking and inference is decidable
- Type inference yields a principal type