Functional Programming

Polymorphic Types

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

ML-Style Polymorphic Types

Simple Types

restrictive, insufficient modularity

Example

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

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

ML-Style Polymorphic Types

Simple Types

restrictive, insufficient modularity

Example

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

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

Approach: Parametric polymorphism $\lambda x.x : \forall\alpha.\alpha \to \alpha$

Applied Mini-ML

Syntax

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

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

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

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

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
- restricts introduction of type variables to toplevel!

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 $\tau' = S\tau$.

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, $\beta_i \notin fv(\sigma)$ and there is a substitution S with $dom(S) \subseteq \{\alpha_1, \dots, \alpha_m\}$ such that $\tau' = S\tau$.

Examples

$$\forall\alpha.\alpha \to \alpha \succeq Nat \to Nat \qquad \forall\alpha\beta.\alpha \to \beta \to \alpha \succeq \forall\alpha.\alpha \to \alpha \to \alpha$$

$$\forall\alpha.\alpha \to \beta \to \alpha \succeq \beta \to \beta \to \beta \qquad \forall\alpha.\alpha \to \beta \to \alpha \succeq Nat \to \beta \to Nat'$$

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

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

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

Generalization

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

where $\{\alpha_1, \dots, \alpha_m\} = \mathit{fv}(\tau) \setminus \mathit{fv}(A)$. $\alpha_1, \dots, \alpha_m$ are **generic variables** in τ.

Inference Rules for Mini-ML

syntax-directed

VAR
$$\frac{\sigma \succeq \tau}{A, x : \sigma \vdash x : \tau}$$

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

APP
$$\frac{A \vdash e : \tau \to \tau' \qquad A \vdash f : \tau}{A \vdash e\,f : \tau'}$$

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

NUM
$$A \vdash n : Nat$$

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

Example Revisited

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

- $\cdot \vdash \lambda x.x : \alpha \to \alpha$
- $gen(\cdot, \alpha \to \alpha) = \forall\alpha.\alpha \to \alpha$
- 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

Hindley-Milner Type Inference 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).
See: Milner, Robin (1978). A Theory of Type Polymorphism in Programming. JCSS, 17: 348–375

Type Inference for Mini-ML

Hindley-Milner Type Inference 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).
See: Milner, Robin (1978). A Theory of Type Polymorphism in Programming. JCSS, 17: 348–375

Notation

- **fresh** creates one or more fresh type variables, which are not yet in use
- ID the identity substitution
- S and U range over type substitutions

Mini-ML Type Inference Algorithm, Part I

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

Mini-ML Type Inference Algorithm, Part II

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

Properties of Type Inference for Mini-ML

Soundness

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

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