

Functional Programming

Polymorphic Types

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

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

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

Applied Mini-ML

Syntax

$$\text{Exp} \ni e, f ::= x \mid \lambda x. e \mid f e \mid \text{let } x = e \text{ in } f \mid n \mid \text{SUCC } e$$
$$\text{Val} \ni v ::= \lambda x. e \mid n$$

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

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

BETA-LET

$$\text{let } x = v \text{ in } e \rightarrow_v e[x \mapsto v]$$

SUCC L

$$\frac{e \rightarrow_v e'}{\text{SUCC } e \rightarrow_v \text{SUCC } e'}$$

DELTA

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

Types for Applied Mini-ML

Syntax of Types

τ	$::=$	$\alpha \mid \tau \rightarrow \tau \mid \text{Nat}$	Types
σ	$::=$	$\tau \mid \forall \alpha. \sigma$	Type Schemes
A	$::=$	$\cdot \mid A, x : \sigma$	Type Environments

A **type scheme** $\forall \alpha. \sigma \dots$

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

Examples

$$\begin{aligned}\forall \alpha. \alpha \rightarrow \alpha &\succeq \text{Nat} \rightarrow \text{Nat} \\ \forall \alpha. \alpha \rightarrow \beta \rightarrow \alpha &\succeq \beta \rightarrow \beta \rightarrow \beta\end{aligned}$$

$$\begin{aligned}\forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha &\succeq \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \\ \forall \alpha. \alpha \rightarrow \beta \rightarrow \alpha &\succeq \text{Nat} \rightarrow \beta \rightarrow \text{Nat}'\end{aligned}$$

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

Examples

$$\begin{aligned} \forall \alpha. \alpha \rightarrow \alpha &\succeq \text{Nat} \rightarrow \text{Nat} \\ \forall \alpha. \alpha \rightarrow \beta \rightarrow \alpha &\succeq \beta \rightarrow \beta \rightarrow \beta \end{aligned}$$

$$\begin{aligned} \forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha &\succeq \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \\ \forall \alpha. \alpha \rightarrow \beta \rightarrow \alpha &\succeq \text{Nat} \rightarrow \beta \rightarrow \text{Nat}' \end{aligned}$$

Generalization

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

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

Inference Rules for Mini-ML

syntax-directed

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

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

$$\text{APP} \quad \frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash f : \tau}{A \vdash e f : \tau'}$$

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

$$\text{NUM} \quad \frac{}{A \vdash n : \text{Nat}}$$

$$\text{SUCC} \quad \frac{A \vdash e : \text{Nat}}{A \vdash \text{SUCC } e : \text{Nat}}$$

Example Revisited

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

- $\cdot \vdash \lambda x.x : \alpha \rightarrow \alpha$
- $gen(\cdot, \alpha \rightarrow \alpha) = \forall \alpha. \alpha \rightarrow \alpha$
- Generalized binding: $i : \forall \alpha. \alpha \rightarrow \alpha$
- $i\ 42$ using instance $\forall \alpha. \alpha \rightarrow \alpha \succeq Nat \rightarrow Nat$
- $i\ (\lambda y.SUCC\ y)$ using instance $\forall \alpha. \alpha \rightarrow \alpha \succeq (Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$
- 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

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

Mini-ML Type Inference Algorithm, Part II

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

Properties of Type Inference for Mini-ML

Soundness

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

Completeness

If $SA \vdash e : \tau'$, then $\mathcal{W}(A; e) = \mathbf{return} (T, \tau)$ 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