CS 611: Lecture 33

Parametric Polymorphism
November 15, 1999

Cornell University Computer Science Department
Andrew Myers

ML type reconstruction

- Last time: introduced idea of type reconstruction, started on Algorithm $\mathcal{W}$
- ML-like language:
  e ::= n | #t | #f | x | $e_0\ e_1$ | fn $x\ e$ | let $(x\ e_0)\ e_1$ | letrec $((x_1\ e_1)...(x_n\ e_n))\ e_b$ | if $e_0\ e_1\ e_2$
- Types:
  t ::= int | bool | $\tau_0 \to \tau_1$ | α (monomorphic)
  $\sigma ::= \forall\alpha_1,...,\alpha_n.\tau$ | τ (polymorphic)
- Typing context extended to allow polymorphic types:
  $\Gamma : \text{Var} \to \sigma$
- But expressions have monomorphic type:
  $\Gamma \vdash e : \tau$

Typing rules

- Correctness of algorithm W is judged with respect to typing rules
  $\mathcal{W}(e, \Gamma, S) = \langle \tau, S' \rangle$
  Reconstructing type of e in typing context Γ with respect to substitution S yields type τ, stronger substitution S'
  Correctness: $S'\Gamma \vdash e : S'\tau$
- Rules of interest:

$$\frac{\Gamma x = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma x = \forall\alpha_1,...,\alpha_n.\tau}{\Gamma \vdash x : \tau\ [\ \tau_1/\alpha_1,...,\tau_n/\alpha_n]}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma[x \mapsto \mathit{Generic}(\tau_0, \Gamma)] \vdash e_1 : \tau_1}{\Gamma \vdash (\text{let } (x\ e_0)\ e_1)}$$

Making types generic

- $\mathit{Generic}(\tau, \Gamma)$ converts a type into a polymorphic form
- $\mathit{Generic}(\tau, \Gamma) = \forall\alpha_1,...,\alpha_n.\tau$
  where $\{\alpha_1,...,\alpha_n\} = \mathit{FTV}(\tau) - \mathit{FTE}(\Gamma)$
- $\mathit{FTE}(\Gamma)$ is removed to avoiding capturing type variables: $\mathit{FTE}(\Gamma) = \cup_{\tau \in \text{range}(\Gamma)} \mathit{FTV}(\tau)$

$\mathit{FTV}(\alpha) = \alpha$ $\qquad$ $\mathit{FTV}(\text{int}) = \varnothing$
$\mathit{FTV}(\tau_0 \to \tau_1) = \mathit{FTV}(\tau_0) \cup \mathit{FTV}(\tau_1)$

Algorithm $\mathcal{W}$

$\mathcal{W}(e, \Gamma, S) = \langle \tau, S' \rangle$ $\qquad$ $S'\Gamma \vdash e : S'\tau$
$\mathit{Unify}(\tau_1 = \tau_2, S) = S'$

$\mathcal{W}(n, \Gamma, S) = \langle \text{int}, S' \rangle$
$\mathcal{W}(\#t, \Gamma, S) = \langle \text{bool}, S' \rangle$
$\mathcal{W}(x, \Gamma, S) = \text{case } \Gamma x \text{ of } \tau \Rightarrow \langle \tau, S \rangle$
$\quad | \ \forall\alpha_1,...,\alpha_n.\tau \Rightarrow \langle \tau[\alpha^1_f/\alpha_1,...], S \rangle$
$\mathcal{W}(e_0\ e_1, \Gamma, S) = \text{let } \langle \tau_0, S_0 \rangle = \mathcal{W}(e_0, \Gamma, S) \text{ in}$
$\quad \text{let } \langle \tau_1, S_1 \rangle = \mathcal{W}(e_0, \Gamma, S_0) \text{ in}$
$\quad \langle \alpha_f, \mathit{Unify}(\tau_1 \to \alpha_f = \tau_0, S_1) \rangle$
$\mathcal{W}(\text{ fn } x\ e, \Gamma, S) = \text{let } \langle \tau, S' \rangle = \mathcal{W}(e, \Gamma[x \mapsto \alpha_f], S) \text{ in}$
$\quad \langle \alpha_f \to \tau, S' \rangle$

Remainder of algorithm

$\mathcal{W}(\text{if } e_0\ e_1\ e_2, \Gamma, S) = \text{let } \langle \tau_0, S_0 \rangle = \mathcal{W}(e_0, \Gamma, S) \text{ in}$
$\quad \text{let } \langle \tau_1, S_1 \rangle = \mathcal{W}(e_1, \Gamma, \mathit{Unify}(\tau_0 = \text{bool}, S_0)) \text{ in}$
$\quad \text{let } \langle \tau_2, S_2 \rangle = \mathcal{W}(e_2, \Gamma, S_1) \text{ in}$
$\quad \langle \tau_1, \mathit{Unify}(\tau_1 = \tau_2, S_2) \rangle$
$\mathcal{W}(\text{let } (x\ e_0)\ e_1, \Gamma, S) = \text{let } \langle \tau_0, S_0 \rangle = \mathcal{W}(e_0, \Gamma, S) \text{ in}$
$\quad \mathcal{W}(e_1, \Gamma[x \mapsto \mathit{Generic}(S_0\tau, S_0\Gamma)], S_0)$
$\mathcal{W}(\text{letrec } ((x_1\ e_1)...(x_n\ e_n))\ e_b, \Gamma, S_0) =$
$\quad \text{let } \Gamma' = \Gamma[x_1 \mapsto \alpha^1_f,...] \text{ in}$
$\quad \text{let } \langle \tau_1, S_1 \rangle = \mathcal{W}(e_1, \Gamma', S) \text{ in } ...\text{let } \langle \tau_n, S_n \rangle = \mathcal{W}(e_n, \Gamma', S_{n-1}) \text{ in}$
$\quad \text{let } S_1' = \mathit{Unify}(\alpha^1_f = \tau_1, S_n) \text{ in} ...\text{let } S_n' = \mathit{Unify}(\alpha^n_f = \tau_n, S'_{n-1}) \text{ in}$
$\quad \text{let } \Gamma'' = \Gamma[x_1 \mapsto \mathit{Generic}(S_n'\tau_1, S_n\Gamma),...] \text{ in}$
$\quad \mathcal{W}(e_b, \Gamma'', S_n')$

Parametric polymorphism

- Polymorphism: expression has multiple types
- Parametric polymorphism ($\forall\alpha_1,\ldots,\alpha_n.\tau$)
  - types appear as parameters
  - expression is written the same way for all types
  - polymorphic value is *instantiated* on some types
- Predicative polymorphism: polymorphic values are not first class (can only be instantiated)
  - ML (implicit), CLU, PolyJ (explicit)
- Bounded parametric polymorphism ($\forall\alpha$:P.τ) -- can only be instantiated on types satisfying P. P may be a *kind* or a fancier bound (CLU, PolyJ)
- Templates (C++, Modula-3) : polymorphic expression is not type checked until instantiated: a macro

Ad-hoc polymorphism

- Same name can be used to denote values with different types
- e.g. "+" refers to one operator on int, another on float, yet another on String
- Examples: C++, Java overloading
- Can be modeled by allowing overloading in the type context Γ
- Not true polymorphism: no polymorphic values

Dependent types

- Types polymorphic with respect to a *value*
- Example: CLU, some varieties of Pascal

```
procedure quicksort(a: array[1..n] of integer,
                    n: int)
```

- A generalization of bounded parametric polymorphism:

$$\forall n : \text{int}, \alpha\text{: type. } \tau$$

- Have to be careful about defining type equivalence if "n" can change...

Subtype polymorphism

- One type S is a *subtype* of another type T if all the values of S are also values of T
- A value is polymorphic because it is a member of all types that are *supertypes* of its type
- Object-oriented languages support subtype polymorphism -- we will discuss in more detail later

First-class polymorphism

- Polymorphic values to be used as first-class values: *impredicative parametric polymorphism*
- 2nd-order polymorphic lambda calculus
- Examples: GJ, Haskell