CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 33
Parametric Polymorphism
13 Nov 00

Last time

- Introduced type inference
  - can write type-safe programs without writing down types explicitly
  - useful for higher-order functions because types are large, obscure code
- Showed type inference algorithm for polymorphic code
  - limited form of *parametric polymorphism*
  - polymorphism is *orthogonal* to type inference (& more important!)

Parametric polymorphism

- Polymorphism: expression has multiple types
- Parametric polymorphism ($\forall X_1,\ldots,X_n.\tau$)
  - types appear as parameters
  - expression is written the same way for all types
  - polymorphic value is *instantiated* on some types
- Java, C: no parametric polymorphism
  - Can't write generic code that doesn't care what are the types of values it manipulates

    `void sort(T[ ] arr)` —must say what T is!
    `Map m;` —can't define key, value type of m
    `v = m.get(k);` — no useful type checking
- C++, Modula-3: generic code through *templates*
  - (nearly) textual substitution

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

Subtype polymorphism

- One type S is a *subtype* of another type T if all the values of S are also values of T
- S ≤ T = "S is a subtype of T"

$$S \le T \Rightarrow \mathcal{T}[\![S]\!] \subseteq \mathcal{T}[\![T]\!]$$
- A value is polymorphic because it is a member of all types that are *supertypes* of its type

  class C extends D { } $\Rightarrow$ C ≤ D
- Object-oriented languages support subtype polymorphism – we will discuss later

First-class polymorphic values

- ML: polymorphic values only bound by "let"... instantiated immediately on use
- Polymorphic values as first-class values:

$\tau ::= B \mid X \mid \tau_1 \to \tau_2$ Turbak & Gifford
$\sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \to \sigma_2$ "plambda" "proj"
$e ::= \lambda x{:}\sigma.e \mid e_1\, e_2 \mid \Lambda X.e \mid e[\tau]$
- Idea: polymorphic value $\Lambda X.e$ can be *explicitly* instantiated on type τ using $e[\tau]$
- Can have $\forall X.\sigma$ inside function type: $(\forall X.X) \to$ bool
- Predicative polymorphism: σ, τ separated, can only instantiate on τ

Operational Semantics

$\tau ::= B \mid X \mid \tau_1 \rightarrow \tau_2$
$\sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2$
$e ::= \lambda x{:}\sigma.e \mid e_1\, e_2 \mid \Lambda X.e \mid e[\tau]$

value abstraction — *type abstraction*

$(\lambda x{:}\sigma.e_1)\; e_2 \rightarrow e_1\{e_2/x\}$
$(\Lambda X.e)\; [\tau] \rightarrow e\{\tau/X\}$

- Type abstraction and application are purely compile-time phenomena

Examples

$\tau ::= B \mid X \mid \tau_1 \rightarrow \tau_2$
$\sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2$
$e ::= \lambda x{:}\sigma.e \mid e_1\, e_2 \mid \Lambda X.e \mid e[\tau]$

$IF \equiv \Lambda V.\lambda f{:}\forall X.X \rightarrow X \rightarrow X.$
$\qquad \lambda x{:}V.\lambda y{:}V.(f\,[V]\; x\; y)$
$TRUE \equiv \Lambda V.\lambda x,y{:}V.x : \forall X.X \rightarrow X \rightarrow X$
$FALSE \equiv \Lambda V.\lambda x,y{:}V.y : \forall X.X \rightarrow X \rightarrow X$

if $e_0\; e_1\; e_2 : \tau \quad \Rightarrow \quad IF[\tau]\; e_0\; e_1\; e_2 : \tau$

Static Semantics

$\tau ::= B \mid X \mid \tau_1 \rightarrow \tau_2$
$\sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2$
$e ::= \lambda x{:}\sigma.e \mid e_1\, e_2 \mid \Lambda X.e \mid e[\tau]$
$\Gamma ::= \emptyset \mid \Gamma, x{:}\sigma$
$\Delta ::= \emptyset \mid \Delta, X$

Judgements:
$\Delta ; \Gamma \vdash e : \sigma$
$\Delta \vdash \sigma$

$$\frac{}{\Delta ; \Gamma, x{:}\sigma \vdash x : \sigma}$$

$$\frac{\Delta ; \Gamma, x{:}\sigma \vdash e : \sigma'}{\Delta ; \Gamma \vdash (\lambda x{:}\sigma.e) : \sigma \rightarrow \sigma'}$$

$$\frac{\Delta ; \Gamma \vdash e_1 : \sigma \rightarrow \sigma' \quad \Delta ; \Gamma \vdash e_2 : \sigma}{\Delta ; \Gamma \vdash e_1\, e_2 : \sigma'}$$

$\forall X.\sigma \cong \forall Y.\sigma\{Y/X\}$

$$\frac{\Delta, X ; \Gamma \vdash e : \sigma}{\Delta ; \Gamma \vdash \Lambda X.e : \forall X.\sigma} \qquad \frac{\Delta ; \Gamma \vdash e : \forall X.\sigma}{\Delta ; \Gamma \vdash e[\tau] : \sigma\{\tau/X\}}$$

Modeling predicative polymorphism

- Need universe $\mathcal{U}$ of all ordinary type interpretations:
  - $\mathfrak{D}_0 = \{\mathbb{Z}, \mathbb{U}\}$
  - $\mathfrak{D}_n = \{X \rightarrow Y \mid X, Y \in \mathfrak{D}_{n-1}\} \cup \mathfrak{D}_{n-1}$
  - $\mathcal{U} = \bigcup_{n\in\omega} \mathcal{U}_n$
  - – Each element of $\mathcal{U}$ is meaning of some type

$\mathcal{T}[\![\text{int}]\!]\chi = Z,\ \mathcal{T}[\![\text{unit}]\!]\chi = U$
$\mathcal{T}[\![\tau_1 \rightarrow \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \rightarrow \mathcal{T}[\![\tau_2]\!]\chi$
$\mathcal{T}[\![\sigma_1 \rightarrow \sigma_2]\!]\chi = \mathcal{T}[\![\sigma_1]\!]\chi \rightarrow \mathcal{T}[\![\sigma_2]\!]\chi$
$\mathcal{T}[\![X]\!]\chi = \chi(X)$
$\mathcal{T}[\![\forall X.\sigma]\!]\chi = \Pi_{D\in\mathcal{U}}\, \mathcal{T}[\![\sigma]\!]\chi[X \mapsto D]$

dependent product: set of all functions mapping $D\in\mathcal{U}$ to $\mathcal{T}[\![\sigma]\!]\chi[X \mapsto D]$

Interpreting terms

- Given type variable environment χ, ordinary variable environment ρ, such that $\chi \vDash \Delta$ and $\rho \vDash \Gamma$,

$$\mathcal{C}[\![\Delta;\Gamma \vdash e : \sigma]\!]\chi\rho \in \mathcal{T}[\![\sigma]\!]\chi$$

$\mathcal{C}[\![\Delta;\Gamma \vdash x : \sigma]\!]\chi\rho = \rho(x)$
$\mathcal{C}[\![\Delta;\Gamma \vdash e_1\, e_2 : \sigma']\!]\chi\rho = (\mathcal{C}[\![\Delta;\Gamma \vdash e_1 : \sigma \rightarrow \sigma']\!]\chi\rho)$
$\qquad (\mathcal{C}[\![\Delta;\Gamma \vdash e_2 : \sigma]\!]\chi\rho)$
$\mathcal{C}[\![\Delta;\Gamma \vdash \lambda x{:}\sigma.e : \sigma \rightarrow \sigma']\!]\chi\rho = \lambda v\in\mathcal{T}[\![\sigma]\!]\chi.$
$\qquad \mathcal{C}[\![\Delta;\Gamma, x{:}\sigma \vdash e : \sigma']\!]\chi\rho[x \mapsto v])$
$\mathcal{C}[\![\Delta;\Gamma \vdash \Lambda X.e : \forall T.\sigma]\!]\chi\rho = \lambda D\in\mathcal{U}.\ \mathcal{C}[\![\Delta, X ; \Gamma \vdash e : \sigma]\!]\chi[T \mapsto D]\rho$
$\mathcal{C}[\![\Delta;\Gamma \vdash e[\tau] : \sigma\{\tau/T\}]\!]\chi\rho = (\mathcal{C}[\![\Delta;\Gamma \vdash e : \forall T.\sigma]\!]\chi\rho)(\mathcal{T}[\![\tau]\!]\chi)$

Substitution lemma: $\mathcal{T}[\![\sigma\{\tau/T\}]\!]\chi = \mathcal{T}[\![\sigma]\!]\chi[T \mapsto \tau]$

System F

- a.k.a "The polymorphic lambda calculus"
- Merges types schemes and types:

$\boxed{\begin{array}{l}\tau ::= B \mid X \mid \tau_1 \rightarrow \tau_2 \\ \sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2\end{array}} \Rightarrow \boxed{\sigma,\tau ::= B \mid X \mid \tau_1 \rightarrow \tau_2 \mid \forall X.\tau}$

- Type schemes are first-class: can instantiate
- *Impredicative polymorphism*
- Type inference: undecidable
- No model for types based on sets

Self-application

- In System F, can write a type for term (λx (x x)) without recursion:
- SELF-APP≡(λx: ∀T.T→T. (x[∀T.T→T] x)) : (∀T.T→T)→(∀T.T→T)
- All standard λ-calculus encodings can be given types too (Church numerals, etc.)

No set model

- Predicative model:

$$\mathcal{T}[\![\forall X.\sigma]\!]\chi = \Pi_{D\in\mathcal{U}}\, \mathcal{T}[\![\sigma]\!]\chi[X\mapsto D]$$

- $\forall X.\sigma$ is the set of all functions from type interpretations D (in $\mathcal{U}$) to corresponding sets $\mathcal{T}[\![\sigma]\!]\chi[X\mapsto D]$
- Need to extend $\mathcal{U}$ to include σ's
- $\forall X.X$ is function mapping all $D\in\mathcal{U}$ to D
- Extension of $\mathcal{T}[\![\forall X.X]\!]$ is $\{\langle D,D\rangle \mid D\in\mathcal{U}\}$. But $\mathcal{T}[\![\forall X.X]\!]\in\mathcal{U}$... can't be a set!

More polymorphism

- Ordinary function application ($e_1\, e_2$):
  term × *term* → *term*
- Type abstraction application ($e\,[\tau]$):
  term × *type* → *term*
- More options?
  type × *term* → *type : dependent typing*
  type × *type* → *type : polymorphic types*
  … *and more…*

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)

  – Allows more compile-time reasoning
  – A generalization of parametric polymorphism:
  $\forall n$: int, α: type. array[n, α]→unit
- Have to be careful about defining type equivalence if "n" can change...

Polymorphic Types

- Most languages include at least one built-in polymorphic type:
  array: *type* → *type*
  Useful to be able to declare own types:

  ML:
```
datatype 'a tree = leaf of 'a
         | branch of 'a*'a tree*'a tree
```
  PolyJ:
```
class Tree[T] { T e; T element(); }
class Branch[T] extends Leaf[T] { Tree[T] left, right; }
```