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

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 \(\Gamma\)
- 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 \(T\)
- \(S \leq T = \text{“}S\text{ is a subtype of }T\text{”}\)
- \(S \leq T \Rightarrow \square[S] \leq \square[T]\)
- 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 later

First-class polymorphic values

- ML: polymorphic values only bound by “let”... instantiated immediately on use
- Polymorphic values as first-class values:
  \[
  \begin{align*}
  \tau &::= B \mid X \mid \tau_1 \rightarrow \tau_2 \\
  \sigma &::= \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2 \\
  e &::= \lambda x:\sigma.e \mid e_1.e_2 \mid \Lambda X.e \mid e[\tau]
  \end{align*}
  \]
- Idea: polymorphic value \(\Lambda X.e\) can be explicitly instantiated on type \(\tau\) using \(e[\tau]\)
- Can have \(\forall X.\sigma\) inside function type: \((\forall X.X)\rightarrow\text{bool}\)
- Predicative polymorphism: \(\sigma, \tau\) separated, can only instantiate on \(\tau\)
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  type  abstraction  abstraction

\[ (\lambda x.\sigma . e) e_2 \rightarrow e_1(e_2/x ) \]
\[ (\Lambda X.e)[\tau] \rightarrow e(\forall X.\tau) \]

* Type abstraction and application are purely compile-time phenomena

Static Semantics

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

\[ \Delta ; \Gamma : e ; \sigma \]
\[ \Delta \rightarrow \sigma \]

Judgements:

\[ \Delta ; \Gamma : e ; \sigma \]
\[ \Delta ; \Gamma : e ; \sigma ' \]
\[ \Delta ; \Gamma : e ; \sigma \rightarrow \sigma ' \]
\[ \Delta ; \Gamma : e ; \sigma ' \]

\[ \forall X.\sigma \equiv \forall Y.\sigma (Y/X) \]
\[ \forall X.\sigma \]

\[ \Delta ; \Gamma : e ; \forall X.\sigma \]
\[ \Delta ; \Gamma : e[\tau] ; \sigma(\tau/X) \]

Interpreting terms

* Given type variable environment \( \chi \), ordinary variable environment \( \rho \), such that \( \chi ; \Delta \) and \( \rho \vdash \Gamma \),

\[ c[\Delta ; \Gamma : e ; \sigma](\rho) \in \mathcal{C}[\sigma] \]
\[ c[\Delta ; \Gamma : e ; \sigma ; \rho] = \mathcal{C}[\Delta ; \Gamma : e ; \sigma](\rho) \]
\[ c[\Delta ; \Gamma : \Lambda X.e ; \sigma \rightarrow \sigma'](\rho) = \mathcal{C}[\Delta ; \Gamma : e ; \sigma](\rho) \]
\[ c[\Delta ; \Gamma : e[\tau] ; \sigma](\rho) = \mathcal{C}[\Delta ; \Gamma : e ; \sigma](\rho) \]

Substitution lemma:

\[ \mathcal{C}[\tau](\rho) \in \mathcal{C}[\tau] \]

System F

* a.k.a “The polymorphic lambda calculus”

Merges types schemes and types:

\[ \tau ::= B \mid X \mid \tau_1 \rightarrow \tau_2 \]
\[ \sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2 \]

* Type schemes are first-class: can instantiate

Impredicative polymorphism

* Type inference: undecidable

No model for types based on sets

Examples

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

\[ IF \equiv \Lambda V.\lambda f : \forall X.X \rightarrow X. \]
\[ \lambda x : V.\lambda y : V. f [V] x y \]

TRUE \( \equiv \Lambda V.\lambda x : V. x : \forall X.X \rightarrow X \rightarrow X \)

FALSE \( \equiv \Lambda V.\lambda x : V. y : V. x : \forall X.X \rightarrow X \rightarrow X \)

if \( e_2 \rightarrow e_1 : \tau \rightarrow \tau_1 \rightarrow \tau_2 \rightarrow \forall X.\tau \)
Self-application

- In System F, can write a type for term \((\lambda x \ (x \ x))\) without recursion:
- \[ SELF-APP \equiv (\forall T. T \rightarrow T. (x [\forall T. T \rightarrow T] x)) : (\forall T. T \rightarrow T) \rightarrow (\forall T. T \rightarrow T) \]
- All standard \(\lambda\)-calculus encodings can be given types too (Church numerals, etc.)

No set model

- Predicative model:
  \[ \mathcal{J}[\forall X. \sigma]_X = \Pi D \in \mathcal{U} \mathcal{J}[\sigma]_{X \rightarrow D} \]
- \(\forall X. \sigma\) is the set of all functions from type interpretations \(D\) (in \(\mathcal{U}\)) to corresponding sets \(\mathcal{J}[\sigma]_{X \rightarrow D}\)
- Need to extend \(\mathcal{U}\) to include \(\sigma\)'s
- \(\forall X. X\) is function mapping all \(D \in \mathcal{U}\) to \(D\)
- Extension of \(\mathcal{J}[\forall X. X]\) is \(\{ D, D \mid D \in \mathcal{U} \}\).
  But \(\mathcal{J}[\forall X. X] \in \mathcal{U}\)... can’t be a set!

More polymorphism

- Ordinary function application \((e_1, e_2)\):
  \[ \text{term} \times \text{term} \rightarrow \text{term} \]
- Type abstraction application \((e[\tau])\):
  \[ \text{term} \times \text{type} \rightarrow \text{term} \]
- More options?
  \[ \text{type} \times \text{term} \rightarrow \text{type} : \text{dependent typing} \]
  \[ \text{type} \times \text{type} \rightarrow \text{type} : \text{polymorphic types} \]
  ... 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:
  
  \[ \forall n : \text{int}, \alpha : \text{type}. \text{array}[n, \alpha] \rightarrow \text{unit} \]
  
  • Have to be careful about defining type equivalence if “n” can change...

Polymorphic Types

- Most languages include at least one built-in polymorphic type:
  \[ \text{array} : \text{type} \rightarrow \text{type} \]
  Useful to be able to declare own types:

  ML:
  
  \[ \text{datatype } 'a \text{ tree } = \text{ leaf of 'a }| \text{ branch of 'a }\text{ a tree }\text{ a tree} \]

  PolyJ:
  
  \[ \text{class Tree[T] } \{ T e; T element(); \} \]
  \[ \text{class Branch[T] extends Leaf[T] } \{ \text{ Tree[T] left, right; } \} \]