Impredicative polymorphism

- Predicative polymorphism supports abstraction over, instantiation on ordinary types \( \tau \):
  
  \[
  e ::= x \mid \lambda x. \, e \mid e_0 \, e_1 \mid \Lambda X. \, e \mid e[\tau]
  \]

  \[
  \tau ::= B \mid \tau_1 \to \tau_2
  \]

  \[
  \sigma ::= \tau \mid \forall X. \sigma \mid \sigma_1 \to \sigma_2
  \]

- Impredicative polymorphism unifies types and type schemas
  
  \[
  \sigma, \tau ::= B \mid X \mid \forall X. \sigma \mid \tau_1 \to \tau_2
  \]

- System F = polymorphic lambda calculus = 2\textsuperscript{nd} – order lambda calculus
**Operational semantics**

- Same as for predicative language
- Term application: \((\lambda x: \sigma. e_1) e_2 \rightarrow e_1[e_2/x]\)
- Type application: \((\Lambda X. e) [\sigma] \rightarrow e[\sigma/X]\)

**Type system**

- Same as before! (type recon. undecidable)
- Well-formed types:

\[
\begin{align*}
\Delta &\vdash B \\
\Delta, X &\vdash X \\
\Delta &\vdash \tau_1 \quad \Delta &\vdash \tau_2 \\
\Delta &\vdash \tau_1 \rightarrow \tau_2 \\
\Delta &\vdash X \vdash \sigma \\
\Delta &\vdash \forall X. \sigma
\end{align*}
\]

- Well-formed terms:

\[
\begin{align*}
\Delta, X; \Gamma &\vdash e : \sigma \\
\Delta, X; \Gamma &\vdash e : \forall X. \sigma \\
\Delta; \Gamma &\vdash e : \forall X. \sigma \\
\Delta; \Gamma &\vdash (\lambda x: \sigma. e) : \sigma \rightarrow \sigma' \\
\Delta; \Gamma, x: \sigma &\vdash e : \sigma' \\
\Delta; \Gamma &\vdash [\tau] : \sigma[\tau/X]
\end{align*}
\]
Some properties of System F

• Now can type self-application $\lambda x.x x$:
  $\lambda x:\forall X.X \to X. x[\forall X.X \to X] x : (\forall X.X \to X) \to (\forall X.X \to X)$

• Can compute primitive recursive functions (like `for i:=1 to n` language)
  – E.g., prime numbers but not Ackermann’s fn

• But still can’t type $\Omega$...strongly normalizing (logical relations again...)

Denotational model?

• Predicative polymorphism: $\forall X.\sigma$
  interpreted as all functions mapping domains $D \in U$ to corresponding interpretation of $\sigma$ (with $X$ bound to $D$)
  $\mathcal{J}[\forall X.\sigma] = \Pi_{D \in U} \mathcal{J}[\sigma]_{X = D}$

• Impredicative: $D$ needs to denote meaning of type schemas too.
  – Domain of functions includes domain of functions to support? E.g., $\lambda x:\forall X.\sigma . x[\forall X.\sigma]$

• Simple set-theoretic model doesn’t work.
Some applications

- **Simply-typed lambda calculus**: application
  \[ e_0 \, e_1 \quad \text{term} \times \text{term} \rightarrow \text{term} \]

- **Second-order lambda calculus**: polymorphism
  \[ e_0[\tau] \quad \text{term} \times \text{type} \rightarrow \text{term} \]
  - C++: `template class<T> sort(T[] arr);`

- **Dependent types** (similar: singleton types):
  \[ \tau_e \quad \text{type} \times \text{term} \rightarrow \text{type} \]
  - some Pascals: `sort(a: array[1..n] of int, n: int)`
  - Cyclone: pointer types marked by `region` variables
  - Careful about what `e`’s are allowed, else unsound!

Parameterized Types/
Higher-order polymorphism

- Have introduced some type constructors:
  \[ \rightarrow, *, + \]

- Can think of type constructors as functions from types to types:
  \[ \rightarrow, + :: \text{type} \times \text{type} \rightarrow \text{type} \]
  - `ref :: type -> type` &c.

- Can we allow the programmer to define their own type constructors?

- Data structures:
  
  ```
  Hashmap<Key, Value> Hashmap :: type*type->type
  Set<Element> Set :: type->type
  datatype α list = nil | some of α*(α list) list :: type->type
  ```
Java 1.5

• Adds parametric polymorphism to Java:

```java
interface Collection<T> {
    public boolean add(T x);
    public boolean contains(T x);
    public Iterator<T> iterator();
    public boolean remove(T x);
    ...
}
```

Collection: type→type
Collection<int> : type

Implementation

class HashSet<T> implements Collection<T> {
    private HashMap<T,T> m;
    public HashSet() { ... }
    public boolean add(T x) {...}
    public boolean contains(T x){
        return m.containsKey(x); }
    public Iterator<T> iterator(){...}
    public boolean remove(T x) {...}
    ...
}
Kinds

- How to prevent ill-formed types like Collection[Collection]?
- Need to keep track of identifiers like Collection, Hashtable, etc. and keep track of their kind
- $F^0$:

$$
K \in \text{Kind ::= type} \mid K \rightarrow K
$$

$$
\tau ::= X \mid B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \tau_2 \mid \lambda X::K.\tau
$$

—A copy of the lambda calculus “one level up” with type as the base kind

Types, Terms, Kinds

$$
\begin{array}{ccc}
\text{type} & \text{type} \rightarrow \text{type} & \text{kinds} \\
\text{int} & \text{int} \rightarrow \text{int} & \lambda X::\text{type}.X \rightarrow X \text{ types} \\
\text{(}\lambda X::\text{type}.X \rightarrow X\text{)} \text{ int} & \forall X.X \rightarrow X & \\
5 & \lambda x:\text{int}.x & \Lambda X.\lambda x:X. x \\
\end{array}
$$
**F°**: Higher-order polymorphism

\[
K ::= \text{type} \mid K \rightarrow K
\]

\[
\tau ::= X \mid B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \tau_2 \mid \lambda X::K.\tau
\]

\[
e ::= x \mid \lambda x::\tau.e \mid e_1 e_2
\]

\[
\Delta ::= \emptyset \mid \Delta, X::K
\]

\[
\Gamma ::= \emptyset \mid \Gamma, x::\tau
\]

**Typing judgment**: \(\Delta;\Gamma \vdash e : \tau\)

**Kinding judgment**: \(\Delta \vdash \tau :: K\)

**Type equivalence**: \(\Delta \vdash \tau_1 \cong \tau_2 :: K\)

**Typing rules**

\[
\frac{\Delta; \Gamma, x::\tau \vdash x : \tau}{\Delta; \Gamma \vdash x : \tau}
\]

\[
\frac{\Delta; \Gamma; e_1 : \tau' \vdash e_2 : \tau'}{\Delta; \Gamma; e_1 e_2 : \tau'}
\]

\[
\frac{\Delta; \Gamma \vdash e : \tau \quad \Delta \vdash \tau :: K}{\Delta; \Gamma \vdash (\lambda x::\tau.e) : \tau \rightarrow \tau'}
\]

**Kinding rules**

\(\Delta \vdash \tau :: K\)

- Just the \(\lambda \rightarrow\) rules...

\[
\frac{\Delta, X::K \vdash X::K}{\Delta, X::K \vdash \tau :: K'}
\]

\[
\frac{\Delta \vdash (\lambda X::K.\tau) :: K \rightarrow K'}{\Delta \vdash \tau :: K'}
\]

\[
\frac{\Delta \vdash \tau_1 :: K \rightarrow K' \quad \Delta \vdash \tau_2 :: K}{\Delta \vdash \tau_1 \tau_2 :: K'}
\]

\[
\frac{\Delta; \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_1 \cong \tau_2 :: K}{\Delta; \Gamma \vdash e : \tau_2}
\]

- Many ways to produce same type... how to decide type equivalence?
- Strong normalization: expansion will terminate!