CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 28: More type parameterization

5 Nov 04

Impredicative polymorphism

- Predicative polymorphism supports abstraction over, instantiation on ordinary types τ:

$e ::= x \mid \lambda x.\, e \mid e_0\, e_1 \mid \Lambda X.e \mid e[\tau]$

$\tau ::= B \mid \tau_1 \rightarrow \tau_2$

$\sigma ::= \tau \mid \forall X.\sigma \mid \sigma_1 \rightarrow \sigma_2$

- Impredicative polymorphism unifies types and type schemas

$\sigma, \tau ::= B \mid X \mid \forall X.\sigma \mid \tau_1 \rightarrow \tau_2$

- System F = polymorphic lambda calculus = 2nd – 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:

$$\frac{}{\Delta \vdash B} \qquad \frac{}{\Delta, X \vdash X} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \rightarrow \tau_2} \qquad \frac{\Delta, X \vdash \sigma}{\Delta \vdash \forall X.\sigma}$$

- Well-formed terms:

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

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

Some properties of System F

- Now can type self-application λx.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 Ω…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 σ (with X bound to D)

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

- 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
  $e_0\ e_1$ term * term → term
- **Second-order lambda calculus**: polymorphism
  $e_0[\tau]$ term * type → term
  - C++: `template class<T> sort(T[] arr);`
- **Dependent types** (similar: singleton types):
  τ_e type * term → 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:
  →, *, +
- Can think of type constructors as functions from types to types:

→, + :: **type∗type→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:

```
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

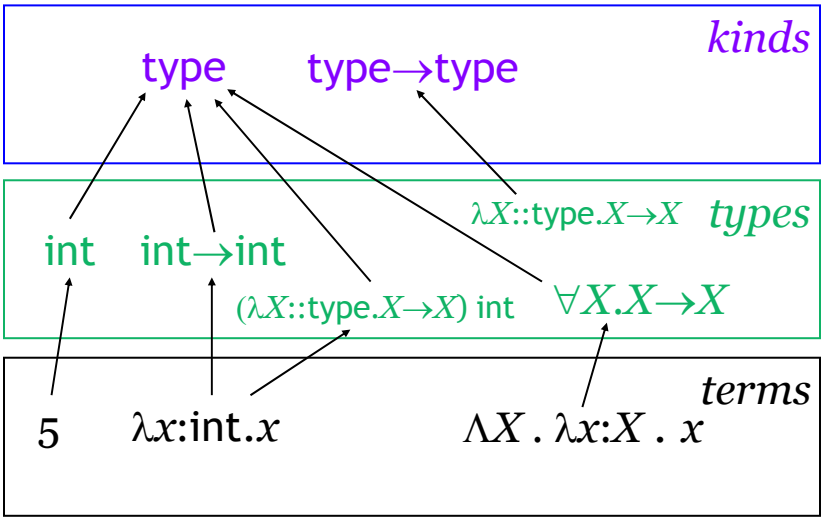
- How to prevent ill-formed types like Collection[Collection]?
- Need to keep track of identifiers like Collection, Hashtable, etc. and keep track of their *kind*
- F^{ω} :

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

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

—A copy of the lambda calculus "one level up" with type as the *base kind*

Types, Terms, Kinds



F^ω : Higher-order polymorphism

$K ::= \text{type} \mid K \to K$

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

$e ::= x \mid \lambda x{:}\tau.e \mid e_1\ e_2$

$\Delta ::= \varnothing \mid \Delta, X::K$

$\Gamma ::= \varnothing \mid \Gamma, x{:}\tau$

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

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

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \to \tau' \quad \Delta;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{\Delta\,;\,\Gamma, x{:}\tau \vdash e : \tau' \quad \Delta \vdash \tau :: \text{type}}{\Delta\,;\,\Gamma \vdash (\lambda x{:}\tau.e) : \tau \to \tau'}$$

Kinding rules ($\Delta \vdash \tau :: K$)

- Just the $\lambda^\to$ rules...

$$\frac{}{\Delta, X::K \vdash X::K}$$

$$\frac{\Delta, X::K \vdash \tau :: K'}{\Delta \vdash (\lambda X::K.\tau) :: K \to K'}$$

$$\frac{\Delta \vdash \tau_1 :: K \to K' \quad \Delta \vdash \tau_2 :: K}{\Delta \vdash \tau_1\ \tau_2 :: K'}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_1 \cong \tau_2 :: \text{type}}{\Delta;\Gamma \vdash e : \tau_2}$$

$K ::= \text{type} \mid K \to K$
$\tau ::= X \mid B \mid \tau_1 \to \tau_2 \mid \tau_1\ \tau_2 \mid \lambda X::K.\tau$
$e ::= x \mid \lambda x{:}\tau.e \mid e_1\ e_2$
$\Delta ::= \varnothing \mid \Delta, X::K$
$\Gamma ::= \varnothing \mid \Gamma, x{:}\tau$

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