CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 40
Objects and Higher-Order Types
1 Dec 00

Typed object calculus

$e ::= \ldots \mid x \mid e.l \mid o \mid e \text{ with } x.l = e'$

$o ::= [x_i.l_i = e_i \ ^{i\in 1..n}]$

$\tau ::= \ldots \mid [l_i{:}\tau_i \ ^{i\in 1..n}]$ ← *object type*

$$\frac{}{o.l_i \rightarrow e_i\{o/x_i\}}$$

$$\frac{}{o \text{ with } x.l_j = e \rightarrow [x.l_j = e,\ x_i.l_i = e_i \ ^{i\in(1..n)-\{j\}}]} \quad j\in 1..n$$

$$\frac{\Gamma, x_i{:}\tau_o \vdash e_i{:}\tau_i}{\Gamma \vdash o : \tau_o} \quad (o=[x_i.l_i = e_i \ ^{i\in 1..n}]) \quad (\tau_o = [l_i{:}\tau_i \ ^{i\in 1..n}])$$

$$\frac{\Gamma\vdash e : \tau_o}{\Gamma \vdash e.l_i : \tau_i} \qquad \frac{\Gamma\vdash e_o : \tau_o \quad \Gamma\vdash e : \tau_j}{\Gamma \vdash e_o \text{ with } x.l_j = e}$$

Java Constructors

class C extends D implements I {
  constructor $C(x_c{:}\tau_c) = D(e_D)$; ... $l_j = e_j$ ...
  public methods ... $m_i = \lambda x_i{:}\tau_i.e_i$ ...
  protected fields ... l_j: τ_j...
}

ObjProtT(C) = μC.[... l_j: τ_j... ... $m_i{:}\tau_j \rightarrow \tau'_j$ ...]
ClassT(C) = { C_{con}: MethodsT(C)*τ_c→ObjProtT(C) }
MethodsT(C) = μC.[... $m_i{:}\tau_j \rightarrow \tau'_j$ ...]

new $C(e_c) \Rightarrow C_{con}$([... this.$m_i = \lambda x_i{:}\tau_j.e_i$ ...], e_c)
$C_{con} = \lambda$this, x_c. D_{con}(this, e_D) with ... with this.$l_i = e_i$ ...

Doesn't type check if e_D uses this
(Can fix by initializing fields to nil)

C++ constructors

class C extends D implements I {
  constructor $C(x_c{:}\tau_c) = D(e_D)$; ... $l_j = e_j$ ...
  public methods ... $m_i = \lambda x_i{:}\tau_i.e_i$ ...
  protected fields ... l_j: τ_j...
}

this not in scope in e_D

ObjProtT(C) = μC.[... l_j: τ_j... ... $m_i{:}\tau_j \rightarrow \tau'_j$ ...]
ClassT(C) = { C_{con}: τ_c→ObjProtT(C) }
new $C(e_c) \Rightarrow C_{con}(e_c)$
$C_{con} = \lambda$this, x_c. Let $o = D_{con}(e_D)$ in
[... *copy fields from o* ..., ... this.$l_i = e_i$ ...
... this.$m_i = \lambda x_i{:}\tau_j.e_i$ ...]

- Expressions e_i, e_j evaluated in context of complete object so far—cannot see uninitialized fields or methods
- But: methods overwritten at every level of construction
- Other options: *makers* initialize fields first (Theta, Moby), or don't have constructors at all (Modula-3)

Prototype-based languages

- So far: *class-based* languages
  - Classes are second-class values, objects are first-class
  - Objects only produced via classes
- Another option: *object-* or *prototype-based* languages (ala object calculus!)
  - No classes (can be simulated, as shown)
  - Can clone other objects, overriding fields
  - Examples: SELF, Cecil, object calculus

Prototype example

In *untyped* object calculus:

point = [p.movex = λd. p with q.x = p.x+d, q.y=p.y]
Inh_point = λP,x,y. (P with p.x = x, p.y=y)
Make_point = λx,y. Inh_point(point, x, y)
colored_point = point with cp.draw = ... cp.color...
Inh_cp = λP,x,y,c.(Inh_point(P,x,y) with p.color = c)
Make_cp = λx,y,c. Inh_cp(colored_point, x, y)

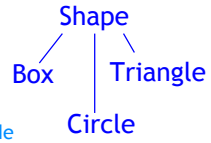
Inheritance without classes!

Multimethods

- Object provide possible extensibility at each method invocation o.m(a,b,c)
  - Different class for "o" permits different code to be substituted after the fact
  - *Object dispatch* selects correct code to run
  - Different classes for a, b, c have no effect on choice of code: not the *method receiver*
- Multimethods/generic functions (CLOS, Dylan, Cecil) : can dispatch on any argument

Shape example

```
class Shape {
   boolean intersects(Shape s);
}
Class Triangle extends Shape {
   boolean intersects(Shape s) {
      typecase (s) {
         Box b => … triangle/box code
         Triangle t => triangle/triangle code
         Circle c => triangle/circle code }}
```



Generic functions:

```
intersects(Box b, Triangle t) { triangle/box code }
intersects(Triangle t1, Triangle t2) { triangle/triangle }
intersects(Circle c, Triangle t) { Triangle/circle }
```

… extensible!

But… semantics difficult to define (what is scope of generic function? Ambiguities!), type-checking problematic

Polymorphic Types

- Have introduced a number of type constructors: →, *, +, {}, [], ref, array, …
- Can think of type constructors as functions from types to types:

→ :: **type** → **type**, + :: **type**→**type**→**type**, &c.

- Can we allow the programmer to define their *own* type constructors? Is this useful?
- Yes: data structures

Hashtable[Key, Value] Hashtable:: **type**→**type**→**type**

Set[Element] Set:: **type**→**type**

datatype α list = nil|some of α*(α list) list :: **type**→**type**

PolyJ

- Java + parametric polymorphism, parameterized types:

```
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

HashSet[int] ≤ Set[int] ≤ Collection[int]

Implementation

```
class HashSet[T] implements Collection[T] {
   private HashMap[T,T] m;
   public boolean add(T x) {…}
   public boolean contains(T x)
        { return m.containsKey(x); }
   public Iterator[T] iterator() {…}
   public boolean remove(T x) {…}
…}
```


Kinds

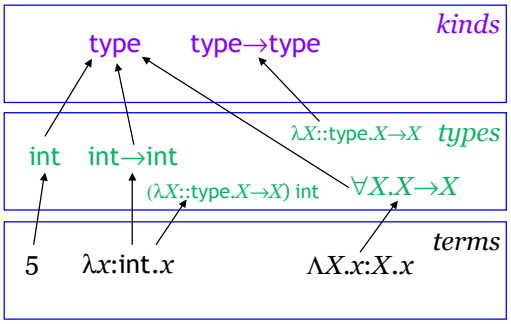
- How to prevent ill-formed types like Collection[Collection]?
- Need to keep track of identifiers like Collection, Hashtable, etc. and keep track of their *kind*
- F^{ω}:

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

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

—A copy of the lambda calculus "one level up"

Types, Terms, Kinds



Fω

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

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

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

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

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

Type judgment: $\Delta;\Gamma \vdash e : \tau$
Kind judgment: $\Delta \vdash \tau :: K$
Type equivalence: $\Delta \vdash \tau_1 \cong \tau_2 :: K$

Typing rules

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

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

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

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

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

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

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

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

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

Bounded type parameters

```
class HashMap[K,V] implements Map[K,V] {
   bool add(K key, V value) { int i = key.hashCode(); ... }
}
```

- Hash table code must be able to compute hash value for values of type K: can't apply HashMap to every type!
- Key type K okay if subtype of

```
interface Hashable { int hashCode(); }
```

K is a *bounded* parameter:
ObjectT(HashMap) =
λK≤Hashable::type.λV::type.{add: K*V→bool}

Bounded polymorphism

```
class HashMap[K,V] implements Map[K,V] {
   static Hashmap() {...}
   bool add(K key, V value) { int i = key.hashCode(); ... }
}
```

- Defines parameterized type ObjectT(HashMap): type of objects
- What is value of *class* object?

ΛK≤Hashable::type.ΛV::type.{...static methods...}
:∀K≤Hashable::type.∀V::type.{...static methods...}

$\tau ::= X \mid B \mid \tau_1 \to \tau_2 \mid \tau_1\ \tau_2 \mid \lambda X{\le}\tau'::K.\tau \mid \forall X{\le}\tau'::K.\tau$

$e ::= x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda X{\le}\tau'::K.e \mid e\,[\tau]$