CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 41: Objects and parameterized types
7 Dec 01

Prototype-based languages

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

Object calculus

- Can explain semantics of OO languages more simply with more powerful construct than recursive records: *object calculus*
  - *Abadi & Cardelli, Ch. 7-8*
- New primitive object expression for object creation: $\{x_1.l_1=e_1,...,x_n.l_n=e_n\}$
  - Idea: x_i stands for name of object (receiver/self) in expression e_i (implicit recursion)
  - Can extend object expression, automatically rebind recursion:

not xx!

```
new_point(xx,yy) = { s.x = xx, s.y = yy,
                     s.movex = λd:int . s + {r.x=s.x+d }}
```


Prototype example

In untyped object calculus:

```
point = {p.movex = λd. p + {q.x = p.x+d, q.y=p.y}
constr_point = λp,x,y. p + {p.x = x, p.y=y}
new_point = λx,y. constr_point(point)
colored_point = point + {cp.draw = … cp.color…}
constr_cp = λp, x,y,c. constr_point(p) + {cp.color = c}
new_cp = λx,y,c. constr_cp(colored_point,x,y,c)
a_cp = Make_cp(10,10,red) = { p.movex = …, p.x = 10,
            p.y = 10, cp.draw = …, cp.color = red }
```

Inheritance without classes!
Methodology: *template, traits* superobjects

Typed object calculus

$e ::= ... \mid x \mid e.l \mid o \mid e + \{x.l = e'\}$

$v, o ::= \{x_i.l_i = e_i \ ^{i\in 1..n}\}$ $(n\geq 0)$

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

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

$$\frac{}{o + \{x.l_j = e\} \mapsto \{x.l_j = e, x_i.l_i = e_i \ ^{i\in(1..n)-\{j\}}\}} \quad (j\in 1..n)$$

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

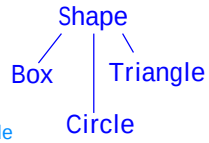
$$\frac{\Gamma\vdash e:\tau_o}{\Gamma\vdash e.l_i:\tau_i} \qquad \frac{\Gamma\vdash e_o:\tau_o \quad \Gamma\vdash e:\tau_j}{\Gamma\vdash e_o + \{x.l_j = e\}}$$

Multimethods

- Object provide possible extensibility at each method invocation *o.m(a,b,c)*
  - Different class for "o" permits different code to be substituted (via subtyping)
  - *Object dispatch* selects correct code to run
  - Run-time types of 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

Intersecting shapes

```
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

7

Parameterized Types

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

$\rightarrow$ +:: **type** $\rightarrow$ **type** $\rightarrow$ **type**, ref :: **type** $\rightarrow$ **type** &c.

- Can we allow the programmer to define their *own* type constructors?
- Data structures:

Hashtable[Key, Value] Hashtable:: **type** $\rightarrow$ **type** $\rightarrow$ **type**

Set[Element] Set:: **type** $\rightarrow$ **type**

datatype α list = nil|some of α*(α list) list :: **type** $\rightarrow$ **type**

8

PolyJ

- Java + parametric polymorphism, parameterized types (also: GJ):

```
interface Collection[T] {
  public boolean add(T x);
  public boolean contains(T x);
  public Iterator[T] iterator();
  public boolean remove(T x);
...}
```

Collection: type $\rightarrow$ type

Collection[int] : type

HashSet[int] $\leq$Set[int] $\leq$Collection[int]

9

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) {...}
...}
```

10

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}:

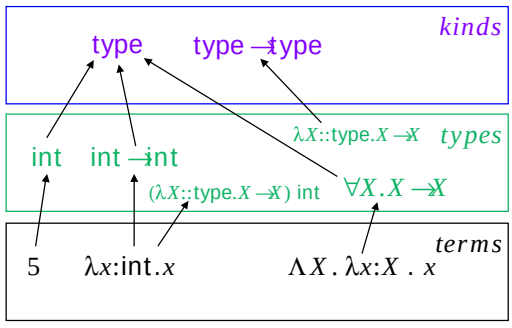
$K \in \textbf{Kind} ::= \text{type} \mid K \rightarrow K$

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

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

11

Types, Terms, Kinds



12

F^{ω}: Higher-order polymorphism

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- 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) =
$\lambda K \leq \text{Hashable}{::}\text{type}.\lambda V{::}\text{type}.\mu S.\{\text{add: } K{*}V \rightarrow \text{bool}, \ldots\}$

Bounded polymorphism

```
class HashMap[K ≤ Comparable[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?

$\Lambda K \leq \text{Comparable}[K]{::}\text{type}.\Lambda V{::}\text{type}.\{\ldots\text{static methods}\ldots\}$
$: \forall K \leq \text{Comparable}[K]{::}\text{type}.\forall V{::}\text{type}.\{\ldots\text{static methods}\ldots\}$

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

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