CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 36: Beyond classes
24 Nov 04

Prototype-based languages

- So far, have discussed *class-based* languages
  - Classes are second-class values, objects are first-class
  - Objects only produced by class constructors
- Another option: *object-based/prototype-based* languages
  - No classes (can be simulated via *template* objects)
  - Inheritance by *cloning* other objects, overriding fields & methods
  - Examples: SELF, Cecil, object calculus

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: $\{x_1.l_1=e_1, \ldots, x_n.l_n=e_n\}$
  - 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 }}

Untyped object calculus

Syntax

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

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

Reductions

$$o.l_i \longrightarrow e_i\{o/x_i\}$$

$$o + \{x.l= e\} \longrightarrow \{x.l = e, x_i.l_i = e_i \ ^{\forall l_i\in\{l_1,\ldots,l_n\} \setminus \{l\}}\})$$

- Can encode untyped lambda calculus
- Can encode classes as objects

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, x, y)

colored_point = point + {cp.draw = … cp.color…}

constr_cp = λp,x,y,c. constr_point(p, x, y) + {cp.color = c}

new_cp = λx,y,c. constr_cp(colored_point,x,y,c)

a_cp = new_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

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

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

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

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

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

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

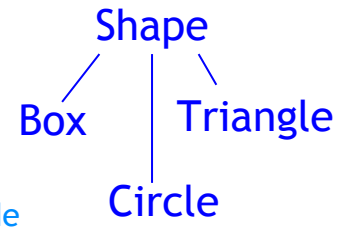
$$\frac{\Gamma \vdash e : \tau_o}{\Gamma \vdash e.l_i : \tau_i} \qquad \frac{\Gamma \vdash e_o : \tau_o \quad \Gamma \vdash e : \tau_j}{\Gamma \vdash e_o + \{x.l_j = e\}}$$

Multimethods

- Object provide possible extensibility at each method invocation o.m(a,b,c)
  - Different class for "o" permits different code to be substituted 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, MultiJava) : 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

Java 1.5

- Java + parametric polymorphism, parameterized types (see also: PolyJ, NextGen):

```
interface Collection⟨T⟩ {
  public boolean add(T x);
  public boolean contains(T x);
  public Iterator⟨T⟩ iterator();
  public boolean remove(T x);
…}
Collection = λT. { add: T → boolean, … }

Collection: type→type
Collection⟨Foo⟩ : type
HashSet⟨Foo⟩ ≤ Set⟨Foo⟩ ≤ Collection⟨Foo⟩
```


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) {…}
…}
```

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:
HashMap =
$\lambda K \leq$Hashable::type.λV::type.μS.{add: $K^*V \rightarrow$bool,…}

Bounded polymorphism

```
class HashMap⟨K≤Comparable⟨K⟩,V⟩ implements Map⟨K,V⟩{
   static Hashmap() {…}
   bool add(K key, V value) { int i = key.hashCode(); … }
}
interface Comparable⟨K⟩ { int compareTo(K other); }
```

- In general need *parameterized* bounding interfaces
- What is value of *class* object?
  $\Lambda K \leq$(Comparable K)::type.ΛV::type.{…static methods…}
  :$\forall$K$\leq$(Comparable$K$)::type.$\forall$V::type.{…static methods…}

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

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

Functors

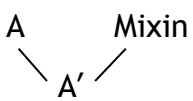
•SML: modules can be parameterized wrt other modules, bounded by signatures

```
signature ORDERED_SET_PARAMS = sig
   type key
   type elem
   val keyOf: elem -> key
   val compare: key * key -> order
end
functor RedBlackTree
   (structure Params : ORDERED_SET_PARAMS) =
struct
   type key = Params.key
   ...Params.compare(key1, key2)...
end

structure StringParams = struct type key=string, ... end
structure StringTree = RedBlackTree(StringParams)
```


Mixins

- Code is expensive and slow to produce...
- Inheritance, polymorphism, functors are abstraction mechanisms, supporting:
  - modular programming
  - code reuse
  - extensibility
- Mixin: mechanism that allows functionality to be "mixed in" to existing class or code base
  - Multimethods: some support
  - Multiple inheritance:
    class A′ extends A, Mixin



Parametric mixins

class Mixin⟨T⟩ extends T { *new functionality* }

- Applying mixin to a class produces a new subclass! (not supported by Java 1.5)
- Problem with parametric reuse (e.g. functors): parameters tend to proliferate

Nested inheritance

- Ordinary inheritance inherits fields, methods
- Inherits some behavior, representation and allows tinkering
- Sometimes want to inherit a whole body of code while preserving class relationships
- Virtual (super-)classes, nested inheritance two mechanisms for this (gBeta, Jx):

```
class A {
    class B {
        void g() { f(); }
        void f();
    }
    class C extends B {
    }
}
```

```
class A' extends A {
    class B {
        int x;
    }
    class C {
        void f() { this.x = 0; }
    }
}
```