CS 611
Advanced Programming Languages

Andrew Myers

Cornell University

Lecture 39

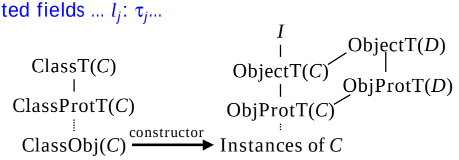
More Objects

29 Nov 00

Classes

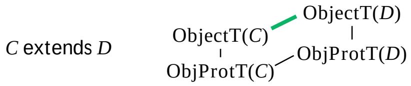
- Last time: introduced OO languages
  - See Abadi & Cardelli for nice informal intro
- Class definition generates several types, values (first- and second-class)

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…
}

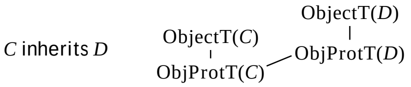


Subtyping vs Inheritance

- Subclassing in Java creates subtype relation between ObjectT(C), ObjectT(D):



- Separate subtyping, inheritance: allows more code reuse. C++: "private" inheritance, Modula-3: subtype relations encapsulated in module



Conformance

- Checking "C extends D" involves checking *conformance* between two classes: types must agree to have $C \le D$ ($C \equiv$ ObjectT(C))
- What conformance is required for inheritance *without* subtyping?
  - Can introduce type variable Self representing subclass when inherited (self: Self)
  - Value of type C will not be used at type D: can relax checking. Covariant argument types ok!

```
class D { boolean equals(Self x)}
class C inherits D { boolean equals(Self x); }
```

Object Types

- What is an object?
- First approximation: recursive record

```
class Point {
    int x, y;
    Point movex(int d) {…}}
```

ObjectT(Point) = $\mu P.\{x\text{: int}, y\text{: int}, movex\text{: int} \to P\}$

- Gives satisfactory account of field, method selection, object construction (w/o inheritance)

new_point(xx,yy) = rec self {x = xx, y = yy,
  movex = λd:int. new_point(self.x + d, self.y) }

- Can find fixed point in CBV language if o only in scope in function-typed exprs (methods)

Inheritance

- Consider colored_point subclass:

```
Class colored_point extends point
  { Color c;
   colored_point(int x, int y, Color cc)
      { point(x,y); cc = c; }
   move_x(int dx)
      { return new colored_point(x+dx, y, c); }}
```

- How to define new_colored_point constructor while using new_point?
- Assume record extension operator e+{..l_i=e_i..}:

  $\{\ a{=}0\ \} + \{b = 1\} = \{a{=}0,\ b{=}1\}$

  $\{\ a{=}0\ \} + \{a = 1\} = \{a{=}1\}$

(in conflict, RHS wins; type of RHS field may be subtype)

Failure

new_point(xx,yy) = rec self {x = xx, y = yy,
movex = λd:int. new_point(self.x + d, self.y) }

new_colored_point(xx,yy,cc) = new_point(xx,yy) +
{ c = cc, movex = ? }

- No way to bind “self” in movex to result of record extension
- No way to rebind “self” in inherited methods from new_point to result of record extension
  - Simple recursive record model is broken
  - Have to open up, rebind recursion of self reference in superclass

Constructor Implementation

- Java-like constructor:

constructor $C(x_c:\tau_c) = D(e_D)$; ... $l_j = e_j$ ...

  - new $C(e_C)$ creates C object with uninitialized fields, initialized methods, invokes C constructor
  - C constructor invokes D constructor ...
  - D constructor runs body to initialize fields $l_{j'}$
  - C constructor runs body to initialize its fields l_j
- Very imperative... hard to describe cleanly
  - Possible to access an uninitialized field?

Explicit recursion

Option 1: constructor receives reference to final result to close recursion, partially-constructed object to build on

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...
}

$$\text{new } C(e_c) \Rightarrow \text{rec } self.\ C_{con}(self, \overbrace{\{\ldots\ m_i = \lambda x_i{:}\tau_j.e_i\ \ldots\ \}}^{\textit{incl. superclass methods}}, e_c)$$

C_{con}: ObjProtT(C)*ObjProtT(C)*$\tau_c \rightarrow$ObjProtT(C) =
$\lambda\, self, o, x_c\,.\, D_{con}(e_D) + \{\ldots\ l_j = e_j\ \ldots\}$

- Need a fancy notion of fixed point to pull this off...

Option 2: object exprs

- 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) in expression e_i (implicit recursion)
  - Can extend object expression, automatically rebind recursion:

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

Typed object calculus

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

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

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

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

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

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

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

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)$*\tau_c\rightarrow$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}: $\tau_c \rightarrow$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, 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, 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]
Make_point = λx,y. (point with p.x = x, p.y=y)
colored_point = point with cp.draw = … cp.color…
Make_cp = λx,y,c. Make_point(x,y) with cp.color = c

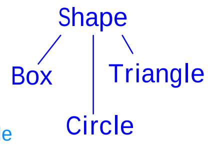
Inheritance without classes!

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