Classes

- Last time: introduced OO languages
  - See Abadi & Cardelli for nice informal intro

Class definition generates several types, values (first- and second-class)

```plaintext
class C extends D implements I {
  constructor C(xc: τc) = D(eD); …
lj = ej …
  public methods … m = λx[τi]. ei …
  protected fields … lj: τj …
}
```

ObjectT(C)
ObjProtT(C)
ClassT(C)
ClassProtT(C)
ClassObj(C)

Instances of C

Subtyping vs Inheritance

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

  ```plaintext
  C extends D     ObjectT(D)
  |                |
  |                |
  |                |
  ClassT(C)      ObjectT(D)
  |                |
  |                |
  |                |
  ClassProtT(C)  ObjProtT(D)
  |                |
  |                |
  |                |
  ClassObj(C)    constructor
  |
  Instances of C
  ```

Conformance

- Checking “C extends D” involves checking conformance between two classes: types must agree to have C ≤ D (C = 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!

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

Object Types

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

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

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

```plaintext
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:

```plaintext
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+{..li:ei..}:

```plaintext
{ a = 0 } + { b = 1 } = { a = 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 = x:d:int. new_point(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

Explicit recursion

Option 1: constructor receives reference to constructed object to build on

\[ \text{constructor } C(x_1 : \tau_1, \ldots, x_n : \tau_n) = \{ \text{l}_1 \ldots \text{l}_n \ldots \text{e}_j \ldots \} \]

\[ \text{new } C(\text{e}_j) \Rightarrow \text{rec self. \{ m = \lambda x_1 : \tau_1, \ldots, e \ldots \}, } \]

\[ C_{\text{con}}: \text{ObjProtT}(\text{C}) \Rightarrow \text{ObjProtT}(\text{C}) \Rightarrow \lambda \text{self}. \x : \text{D}_{\text{con}}(\text{e}_j) + \{ \text{l}_1 \ldots \text{e}_j \ldots \} \]

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

Typed object calculus

\[ e ::= \ldots | x | \text{e} \cdot \text{e} | \text{e} \text{ with } x.l = e' \]

\[ o ::= \{ x, l = e_j, i : [\text{L}; \tau_i \text{ is L}] \} \quad \text{object type} \]

\[ o, l \rightarrow e_o(o/x) \]

\[ o \text{ with } x.l = e \Rightarrow \{ x.l = e, x_1.l = e_j \} \quad (j \in 1..n) \]

\[ \Gamma, x_1 : \tau_1 \vdash e : \tau_i \]

\[ \Gamma \vdash e : \tau_i \quad \Gamma, x_1 : \tau_1 \vdash e_j : \tau_i \quad (o = [x_1.l = e_j, i : \text{L}]) \]

\[ \Gamma \vdash e, e_j : \tau_i \]

\[ \Gamma, x_1 : \tau_1 \vdash e : \tau_i \]

\[ \Gamma \vdash e : \tau_i \quad \Gamma, x_1, \text{with } x.l = e \]

Java Constructors

\[ \text{class } C \text{ extends D implements } I \{ \]

\[ \text{constructor } C(x_1 : \tau_1, \ldots, x_n : \tau_n) = \{ \text{l}_1 \ldots \text{l}_n \ldots \text{e}_j \ldots \} \]

\[ \text{new } C(\text{e}_j) \Rightarrow \text{C}_{\text{con}}(\ldots \text{this}, m_1 : \lambda x_1 : \tau_1, \ldots, \text{e}_j) \]

\[ C_{\text{con}} = \lambda \text{this}, x_1. \text{D}_{\text{con}}(\text{this}, \text{e}_j) \text{ with } \ldots \text{with this, l}_1 = e_j \]

\[ \text{Doesn't type check if } e_j \text{ uses this} \quad \text{(Can fix by initializing fields to nil)} \]
C++ constructors

```cpp
class C extends D implements I {
    constructor C (xc: \tau_c) = D(e_D'); ... l_j = e_j ... public methods ... \eta_i = \lambda x_i: \tau_i . e_i ... protected fields ... l_j: \tau_j ... this in scope in e_o
    ObjProtT(C) = \mu C. [ ... l_j: \tau_j ... ]
    ClassT(C) = { Ccon: \tau_c \rightarrow ObjProtT(C) }
    new C(ec) \Rightarrow Ccon(ec)
    Ccon = \lambda this, x_c. Let o = Dcon(e_D) in
    [ ... copy fields from o ... ] this.l_j = e_j ...
    ... this.m_i = \lambda x_i: \tau_i . e_i ... ]
    • Expressions e_o, e_i 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 = \lambda d. p with q.x = p.x+d, q.y=p.y]
Make_point = \lambda x,y. (point with p.x = x, p.y=y)
colored_point = point with cp.draw = ... cp.color...
Make_cp = \lambda x,y,c. Make_point(x,y) with cp.color = c
```

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 }]
    }
    Intersects(Box b, Triangle t) { triangle/box code }
    Intersects(Triangle t1, Triangle t2) { triangle/triangle code }
    Intersects(Circle c, Triangle t1) { Triangle/circle code }
    ... extensible!
    But... semantics difficult to define (what is scope of generic function? Ambiguities?), type-checking problematic
```