Encoding objects

- Current model: objects are recursive records. Ignoring encapsulation, existentials, etc:

```haskell
class point = { x, y: int;
    point(x1,y1) { x = x1; y = y1; }
    point movx(dx) = new point(x + dx, y) }
```

```
ObjectT(point) = \mu P. \{ x, y: int, movx: int \to P \},
```

```
ClassObj(point) = \{ constructor = rec f:int*int \to Tpoint.
    \lambda x1,y1.fold
        Tpoint
    rec self.{x = x1, y= y1,
        movx = \lambda dx. p(self.x + dx, self.y) }
```

```
ClassT(point) = \{ constructor : int*int \to ObjectT(point) 
```

Fixed point works if “this” in scope only in methods.

“Interfaces”

- Java interface is a recursive object type

```haskell
interface pt {
    boolean equals(pt p);
    pt movx(int dx);
}
```

```
ObjectT(pt) = \mu T. \{ equals: T \to bool,
    movvx: int \to \mu P. \}
```

Classes

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

```haskell
class C extends D implements I {
    constructor C(xc: \tau) = D(eD); …
    static methods … m' i = \lambda x_i: \tau_i. e_i …
    static fields … l' j: \tau_j …
    methods … m_i = \lambda x_i: \tau_i. e_i …
    fields … l_i j: \tau_j …
}
```

```
ClassT(C) \to ObjectT(C) \to ObjProtT(C) \to ClassObj(C) 
```

```
ClassObj(C) Instances of C
```

Subtyping vs Inheritance

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

```
C extends D ObjectT(C) \to ObjectT(D) \to ObjProtT(D) 
```

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

```
C inherits D ObjectT(C) \to ObjectT(D) \to ObjProtT(D) 
```

Conformance

- Checking “C extends D” involves checking conformance between two classes: types must agree to have \( C \leq D \) (\( C = ObjectT(C) \))

- What conformance is required for “C inherits D”?
  – Can introduce type variable \texttt{Self} representing subclass when inherited (\texttt{self: Self})
  – Value of type \texttt{C} will not be used at type \texttt{D}: can relax checking. Covariant argument types ok!

```haskell
class D { boolean equals(\texttt{Self} x) }
class C inherits D { boolean equals(\texttt{Self} x); }
```
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(dx); return new colored_point(x+dx, y, c);
    }
  }
  ```
  (in conflict, RHS wins; type of RHS field may be subtype)

- How to define new_colored_point constructor while using new_point?
- Assume record extension operator \( e \{ \ldots \} \):
  ```
  \{ \alpha=0 \} + \{ b=1 \} = \{ \alpha=0, b=1 \}
  e^{\{ \ldots \}} = \text{let } r:\{ x:\tau_1, \ldots, x_m:\tau_m, \ldots \} = \epsilon \text{ in }
  \{ x_1 = r.x_1, \ldots, x_m = r.x_m, \ldots \} \}
  ```

- Constructor Implementation

  - Java-like constructor:
    ```
    colored_point extends point
    {
      Color c;
      colored_point(int x, int y, Color cc) {
        point(x,y); cc = c;
        move_x(dx); return new colored_point(x+dx, y, c);
      }
    }
    ```
    (in conflict, RHS wins; type of RHS field may be subtype)

  - Very imperative... hard to describe cleanly
  - Possible to access an uninitialized field?

Failure

```java
new point(x1,y1) = rec self { x = x1, y = y1,
  movex = \lambda d: int. new point(self.x + d, self.y) }
```  
  ```java
ew 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

Model: constructor receives reference to final result to close recursion
```java
class C extends D implements I {
  constructor C(x, \tau) = \{ D(e_0); \ldots \} i = e_\ldots \}
  \text{ new } C(e) \text{ creates } O \text{ object with uninitialized fields, initialized methods, invokes } C
  \text{ constructor}
  - C constructor invokes D constructor ...
  - D constructor runs body to initialize fields \( I_j \)
  - C constructor runs body to initialize fields \( i_j \)

- Very imperative... hard to describe cleanly
- Possible to access an uninitialized field?

C++ constructors

```java
class C extends D implements I {
  constructor C(x, \tau) = \{ D(e_0); \ldots \} i = e_\ldots \}
  \text{ new } C(e) \text{ creates } O \text{ object with uninitialized fields, initialized methods, invokes } C
  \text{ constructor}
  - C constructor invokes D constructor ...
  - D constructor runs body to initialize fields \( I_j \)
  - C constructor runs body to initialize fields \( i_j \)

- Very imperative... hard to describe cleanly
- Possible to access an uninitialized field?
```

- Fixed point! Need bottom element at every type null/0