CS 611
Advanced Programming Languages

Andrew Myers

Cornell University

Lecture 40

⟦Classes, Inheritance, Constructors⟧

5 Dec 01

Encoding objects

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

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

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

ClassObj(point) =

{ constructor = rec f:int*int $\to T_{point}$.$\lambda x_1,y_1$.fold$_{T_{point}}$ rec self {x = x_1, y= y_1,

movx = λdx. 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

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

ObjectT(pt) = μT. {equals: T $\to$ bool,

movx: int $\to T$ },

Classes

- 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 ...

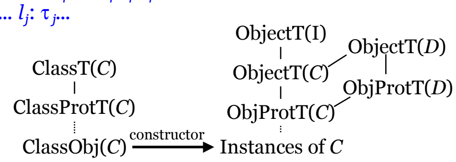
static methods ... $m'_i = \lambda x_i{:}\tau_i.e_i$ ...

static fields ... l'_j: τ_j...

methods ... $m_i = \lambda x_i{:}\tau_i.e_i$ ...

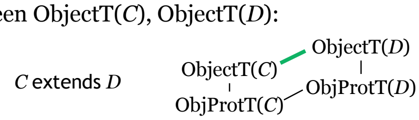
fields ... l_j: τ_j...

}

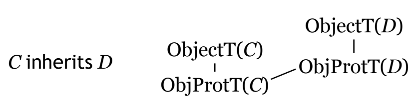


Subtyping vs Inheritance

- 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 $C \leq D$ ($C \equiv$ ObjectT(C))
- What conformance is required for "C inherits D"?
  - 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); }
```

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 $e+\{\ldots l_i=e_i\ldots\}$:
  $\{ a=0 \} + \{b = 1\} = \{a=0, b=1\}$
  $e+\{..l_i=e_i..\}$ = let $r{:}\{x_1{:}\tau_1,\ldots,x_m{:}\tau_m\} = e$ in
  $\{ x_1 = r.x_1,\ldots, x_m=r.x_m, \ldots l_i = e_i\ldots\}$

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

Failure

```
new point(x1,y1) = rec self {x = x1, y = y1,
        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 $C(x_c{:}\tau_c) = \{\ D(e_D);\ \ldots\ l_j = e_j\ \ldots\ \}$

  - 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 fields l_j
- Very imperative… hard to describe cleanly
  - Possible to access an uninitialized field?

Explicit recursion

Model: constructor receives reference to final result to close recursion

class C extends D implements I {
  constructor $C(x_c{:}\tau_c) = \{\ D(e_D);\ \ldots\ l_j = e_j\ \ldots\ \}$
  methods … $m_i = \lambda x_i{:}\tau_i.e_i$ …
  fields … $l_j{:}\ \tau_j$…
}

Java constructors:

Constr(C) : ObjectProtT(C)$*\tau_c \to$ ObjectProtT(C)
  $= \lambda x_c{:}\tau_c, \mathit{self}$: ObjectProtT(C).
    Constr(D)$(e_D, \mathit{self}) + \{\ldots\ m_i = \lambda x_i{:}\tau_j.e_i\ \ldots, \ldots l_j = e_j\ldots\}$

new $C(e_c)$= rec self: ObjectProtT(C). Constr(C) (e_c, self)

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

C++ constructors

class C extends D implements I {
  constructor $C(x_c{:}\tau_c) = D(e_D);\ \ldots\ l_j = e_j\ \ldots$
  public methods … $m_i = \lambda x_i{:}\tau_i.e_i$ …
  protected fields … $l_j{:}\ \tau_j$… }

MethodT(C) = { …$\tau_i \to \tau_i$… } — this not in scope in e_D

Constr(C): MethodT(C) $* \tau_c \to$ ObjectProtT(C) =
  $\lambda \mathit{self}, x_c$. let app = Constr(D)(e_D) in $\mathit{app}(\mathit{self})$ +
    $\{\ldots\ m_i = \lambda x_i{:}\tau_j.e_i\ \ldots\}$ +
    (let self = (rec s.app(s) + $\{\ldots\ m_i = \lambda x_i{:}\tau_j.e_i\ \ldots\}$) in
      $\{\ \ldots l_j = e_j \ldots\}$)

- Expressions e_i, e_j evaluated in context of completed object so far—cannot see uninitialized fields or methods
- Object constructed in series of *observable* approximations: methods overwritten at every level!
- Other options: *makers* initialize fields first (Theta, Moby), or don't have constructors at all (Modula-3)