CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 35
Object-oriented languages
23 Nov 04

Object-oriented languages

- Dominant programming paradigm for foreseeable future. Why?
  - Inheritance
  - Encapsulation/information hiding (∃)
  - Subtype polymorphism (≤)
  - Static typing
  - New! Parametric polymorphism (C#, Java 1.5)
- Weaknesses:
  - Pattern matching, iteration (see: JMatch)
  - Type inference
  - Closures (can encode)
- Reading: Abadi and Cardelli, Ch. 1-6

Classes

- Program is a set of classes [Simula67]
- Classes contain:
  - Static fields
  - Static methods
  - Constructors
    - Static methods that build a new object
  - Instance fields
  - Instance methods
- Classes can *inherit* instance members from other classes
- Classes can *implement* interfaces
- Whew!

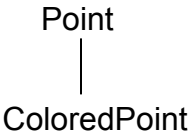
```
class List
    extends  AbstractCollection
    implements Collection {
    static List theEmpty = null;
    static List empty()
      { return theEmpty; }

    Object hd;
    List tl;
    List List(Object h, List t) {
      hd = h; tl = t;
    }
    Object head()
      { return this.hd; }
    Object tail();
      { return this.tl; }
}
```


Inheritance

```
class Point {
    int x, y;
    void movex(int d) { this.x = this.x + d; }
    void movexy(int dx, int dy) { movex(dx); movey(dy); }
}
class ColoredPoint extends Point {
    Color c;
    ColoredPoint(int x, int y, Color cc)
          { point(x,y); this.c = cc; }
    void movex(int d) { x = x + d; c = red; }
}

ColoredPoint p = new ColoredPoint(0, 0, black);
p.movex(1);
```



- Instances of ColoredPoint have all the fields, methods declared in Point, unless overridden
- Inheritance works like (efficient) copying
- Implicit *receiver object* method argument (this/self)

Interfaces as types

- Java interfaces are object types

```
interface Pt {
   void movex(int d);
   void movey(int d);
   void movexy(int dx, int dy);
}
```

ObjectT(Pt) = μS.{ movex: int -> unit,
movey: int -> unit,
movexy: int*int -> unit }

- Interface extension is subtyping ("interface inheritance")

Classes as types

- Class defines an object type and a class type

```
class List extends Collection {
    static List theEmpty = null;
    static List empty()
        { return theEmpty; }

    Object hd;
    List tl;
    List List(Object h, List t) {
        hd = h; tl = t;
    }
    Object head()
        { return this.hd; }
    Object tail();
        { return this.tl; }
}
```

ObjT(List) =
μS.{ hd: Object,
tl: S,
head: unit → Object,
tail: unit → S }

ClassT(List) = {
theEmpty: List,
empty: unit → List,
ListCons: Object * List → List
}

Class objects

- Class defines a singleton *value* of the class type
- Constructors build new object values

```
class List extends Collection {
    static List theEmpty = null;
    static List empty()
        { return theEmpty; }

    Object hd;
    List tl;
    List List(Object h, List t) {
        hd = h; tl = t;
    }
    Object head()
        { return this.hd; }
    Object tail();
        { return this.tl; }
}
```

```
ListClass: ClassT(List) = {
    theEmpty = inr(unit),
    empty = λu:1. theEmpty,
    ListCons =
    λo: Object, t: List.
        fold_List rec o:ObjT(List) {
            hd = o, tl = t, head=…, tail=…
        }
}
```


Information hiding

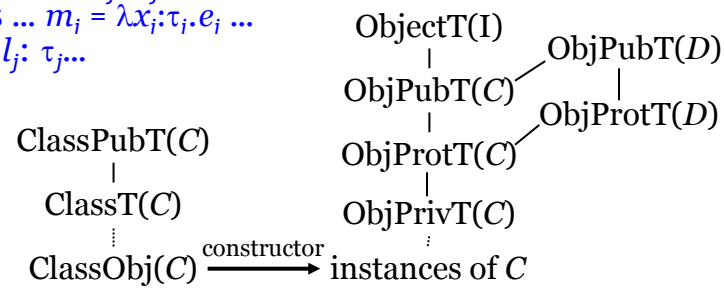
- Class members usually can have access modifiers (public, private, protected)
- Can interpret as existential types or as subtyping

ObjPubT(*C*)
|
ObjProtT(*C*)
|
ObjPrivT(*C*)

Classes

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

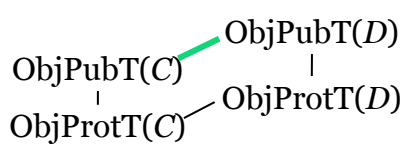
class C extends D implements I {
  constructor $C(x_c{:}\tau_c) = D(e_D)$; … $l_j = e_j$ …
  static methods … $m'_i = \lambda x_i{:}\tau_i.e_i$ …
  static fields … $l'_j{:}\ \tau_j$…
  methods … $m_i = \lambda x_i{:}\tau_i.e_i$ …
  fields … $l_j{:}\ \tau_j$…
}



Subtyping vs. inheritance

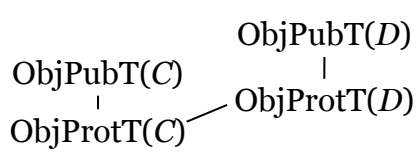
- Subclassing in Java creates subtype relation between object types of classes:

C extends D



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

C inherits D



Conformance

- "C extends D" requires conformance between two classes: types must have C ≤ D (ObjProtT(C)≤ObjProtT(D))
  - Methods: covariant return types, contravariant arguments
- What conformance is required for inheritance without subtyping?
  - Can introduce "self type" type variable This/Self representing subclass when inherited
  - Value of type C will not be used at type D: can relax checking. Covariant argument types ok!

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


Constructors

- Static on the outside, non-static on the inside
- Useful for establishing *data structure invariants*
  - Methods can assume incoming objects of same class satisfy these invariants – simplifies code

Inheritance

```
class ColoredPoint extends Point
   { Color c;
     ColoredPoint(int x, int y, Color cc)
        { super(x,y); c = cc; }
```

- How to define ColoredPoint constructor while using Point constructor?
- Assume record extension operator $e+\{\ldots l_i=e_i\ldots\}$:

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

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

  $\{ x_1 = r.x_1,\ldots, x_m=r.x_m, \ldots l_i = e_i\ldots\}$

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

Failed encoding

```
new Point(x1,y1) = rec this {x = ref x1, y = ref y1,
        movex = λd:int. this.x := (!this.x) + d }
new ColoredPoint(xx,yy,cc) = new Point(xx,yy) +
                { c = cc, movex = ? }
```

- No way to bind "this" in movex to result of record extension
- No way to rebind "this" in inherited methods from new_point to result of record extension
  - Simple closed recursive record model is broken
  - How to open up & rebind recursion of this reference?

Constructor Implementation

- C++/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 $C(x_c{:}\tau_c) = \{ D(e_d);\ e_b\}$
  methods … $m_i = \lambda x_i{:}\tau_i.e_i$ …
  fields … l_j: τ_j…
}

Constructor as initializer

Java constructors:

Constr(C) : ObjPrivT(C)*$\tau_c \rightarrow$ ObjPrivT(C)
  $= \lambda x_c{:}\tau_c$, *this*: ObjPrivT(C).
    Constr(D)(e_D, *this* + {..$m_i = \lambda x_i{:}\tau_i.e_i$..}) + ..$l_j = e_j$..}

new $C(e_c)$= rec *this*: ObjPrivT(C). Constr(C) (e_c, *this*)

Constructor as creator

- Fixed point; need bottom element at *every* type…null/0 (more observable than nontermination…oops)

A problematic example

```
class A {
        A() { if (!checkOK()) throw error; }
        checkOK() { return true; }
}
class B extends A {
        final SecurityTag y;
        B() { A(); y = new SecurityTag() }
        checkOK() { return this.y.saysOK(); }
}
```


C++ constructors

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

this not in scope in e_D

- Expressions e_D, e_i 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!
  - Can't see uninitialized fields, but methods change
- Other options: *makers* initialize fields first (Theta, Moby), or don't have constructors at all (Modula-3)