CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 38
Object-oriented languages
26 Nov 07

Object-oriented languages

- Dominant programming paradigm for foreseeable future. Why?
  - Encapsulation/information hiding (∃)
    - Abstraction over implementations
  - Subtype polymorphism (≤)
  - Inheritance with open recursion/late binding
  - Static typing
  - Parametric polymorphism (C#, Java 1.5)
    - Abstraction over client
- Weaknesses:
  - Pattern matching, iteration (see: JMatch)
  - Type inference
  - Closures (but can encode as inner classes)
- Reading: Pierce 18, Abadi and Cardelli, Ch. 1-6

Classes

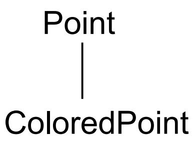
- Program is a set of classes [Simula67]
- Classes contain:
  - Static (class) fields
  - Static (class) methods
  - Constructors
    - Static methods that build a new object
  - Instance fields
  - Instance methods (may be “abstract”)
- Classes can *inherit* instance members from other classes
- Classes can *implement* interfaces

```
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);
}
```

ObjT(Pt) = μS.{ movex: int → 1,
movey: int → 1,
movexy: int*int → 1 }

- Interface extension is subtyping (aka “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
}

Sort of...

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. theEmpty,
   ListCons =
     λo: Object, t: List.
        rec this:ObjT(List) {
          hd = o, tl = t,
          head=λz:1.(this.hd),
          tail=…             }
}
```

Closed recursion ⇒ won't work with inheritance

Encapsulation mechanisms

- Class members usually can have access modifiers (public, private, protected)
  - Supports encapsulation (aka "information hiding")
- Can interpret as existential types or as subtyping:

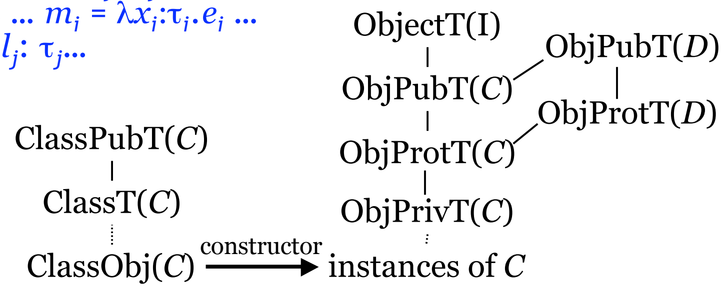
ObjPubT(*C*)
|
ObjProtT(*C*)
|
ObjPrivT(*C*)

- Public interface permits abstraction over clients, controlled exposure of implementation

Classes

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

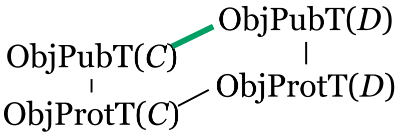
class C extends D implements I {
constructor $C(x_c{:}\tau_c) = D(e_D);$ … $l_j = e_j$ …
static methods … $m'_i = \lambda x_i{:}\tau_i.e_i$ …
static fields … $l'_j{:}\ \tau_j$…
methods … $m_i = \lambda x_i{:}\tau_i.e_i$ …
fields … $l_j{:}\ \tau_j$…
}



Subtyping vs. inheritance

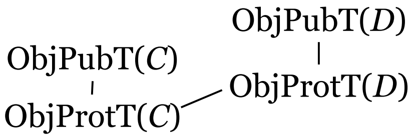
- Subclassing in Java creates subtype relation between object types of classes:

C extends D



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

C inherits D



Specialization interface

- C++, Java: methods may be marked "final" or "nonvirtual" -- cannot be overridden by subclasses
- "Virtual" methods form a *specialization interface* : contract between class and its subclass.
  - Abstracts with respect to superclasses being *extended* rather than code being called
  - Allows controlled exposure to subclasses
  - Why writing good frameworks is harder than writing good libraries…

Conformance

- "C extends D" requires *conformance* between two classes: types must have $C \leq D$ ($\mathrm{ObjProtT}(C) \leq \mathrm{ObjProtT}(D)$)
  - 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 (can access “this”)
- Can establish *representation invariants*
  - Methods can assume incoming objects of same class satisfy these invariants – simplifies code

```
class Rational {
  int num, den; // rep invariant: den > 0,
                //   num≠0 ⇒ (gcd(num,den)=1)
Rational(int p, int q) {
  int g = gcd(p,q);
  num = p/g; den = q/g;
  if (den < 0) {  num = -num; den = -den;
}
Rational plus(Rational r) {// assume RI(this), RI(r)
  …
```

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 $e+\{\ldots l_i=e_i\ldots\}$:

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

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

$$\{\, x_1 = r.x_1,\ldots,\, x_m=r.x_m,\, \ldots l_\mathrm{i} = e_\mathrm{i}\ldots\}$$

(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 $C(x_c{:}\tau_c)$ = { $D(e_D)$; … $l_j = e_j$ …}

  - 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 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…
}

Java constructors:

Constructor as *initializer*

$\mathrm{Constr}(C) : \tau_c \to \underline{\mathrm{ObjPrivT}(C) \to \mathrm{ObjPrivT}(C)}$ preobject

$= \lambda x_c{:}\tau_c.\ \lambda \mathit{this}\text{: ObjPrivT(C).}$

$\mathrm{Constr}(D)(e_D, \mathit{this} + \{..m_i = \lambda x_i{:}\tau_i.e_i..\}) + ..l_j = e_j..\}$

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

Constructor as creator

- Fixed point needs bottom element at *every* type…null/0 (more observable than nontermination…can see uninit fields in Java!

A problematic Java 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(); }
}
```

- A “final” field appears to change!
- Need to know which methods are called from superclass constructors…

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: τ_j… }

this not in scope in e_D

- Pro: Expressions e_D, e_i evaluated in context of completed object so far—cannot see uninitialized fields or methods
- Con: 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 no constructors at all (Modula-3)

CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 39: Beyond classes
28 Nov 07

Prototype-based languages

- So far, have discussed *class-based* languages
  - Classes are second-class values, objects are first-class
  - Objects only produced by class constructors
- Another option: *object-based/prototype-based* languages
  - No classes (can be simulated via *template* objects)
  - Inheritance by *cloning* other objects, overriding fields & methods
  - Examples: SELF, Cecil, JavaScript, object calculus

Object calculus

- 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/self) in expression e_i (implicit recursion)
  - Can extend object expression with +, automatically rebind recursion:

not xx or r.x*!*

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

Untyped object calculus

Syntax

$$e ::= x \mid o \mid e.l \mid e + \{x.l = e'\}$$

$$v ::= \{x_i.l_i = e_i \ ^{i \in 1..n}\} \ (n \geq 0)$$

Reductions

$$\left(o = \{x_i.l_i = e_i \ ^{i \in 1..n}\} \ ^{(n \geq 0)}\right)$$

$$o.l_i \longrightarrow e_i\{o/x_i\}$$

$$o + \{x.l = e\} \longrightarrow \{x.l = e, x_i.l_i = e_i \ ^{\forall l_i \in \{l_1,\ldots,l_n\} - \{l\}}\})$$

- Can encode untyped lambda calculus
- Can encode classes as objects

Typed object calculus

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

$$v, o ::= \{x_i.l_i = e_i \ ^{i \in 1..n}\} \ (n \geq 0)$$

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

$$o.l_i \longrightarrow e_i\{o/x_i\}$$

$$o + \{x.l_j = e\} \longrightarrow \{x.l_j = e, x_i.l_i = e_i \ ^{\forall i \in (1..n) - \{j\}}\}) \text{ (where } j \in 1..n)$$

$$\frac{\Gamma, x_i : \tau_o \vdash e_i : \tau_i \ ^{(\forall i \in 1..n)}}{\Gamma \vdash o : \tau_o}$$

$$(o \triangleq \{x_i.l_i = e_i \ ^{\forall i \in 1..n}\})$$

$$(\tau_o \triangleq \{l_i{:}\tau_i \ ^{\forall i \in 1..n}\})$$

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

Prototype example

In untyped object calculus:

point = {p.movex = λd. p + {q.x = p.x+d, q.y=p.y}}
constr_point = λp,x,y. p + {p.x = x, p.y=y}
new_point = λx,y. constr_point(point, x, y)

colored_point = point + {cp.draw = … cp.color…}
constr_cp = λp,x,y,c. constr_point(p, x, y) + {cp.color = c}
new_cp = λx,y,c. constr_cp(colored_point,x,y,c)
a_cp = new_cp(10,10,red) = { p.movex = …, p.x = 10,
p.y = 10, cp.draw = …, cp.color = red }

Inheritance without classes!
(Java-like constructor semantics)
Methodology: *template/traits* superobjects

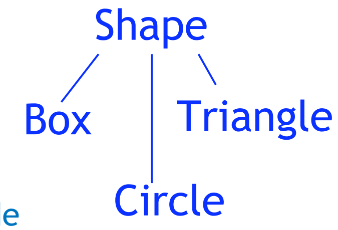
Implementing classes (typed)

T_{Point} = μT.{x: int, y: int, movex: int→T}
$T_{ColoredPoint}$ = μT.{x: int, y: int, c: color, movex: int→T, draw: 1→1} ≤ T_{Point}
Point = {
    cl.init : T_{Point}*int*int→T_{Point} = λt: T_{Point}, x:int, y:int .
        t + {p.x = x, p.y = y}
    cl.new : int*int→T_{Point} = λx:int, y:int . cl.init(PointTemplate, x, y)
}
PointTemplate: T_{Point} = { x: int = 0,   y: int = 0,
        p.movex = λd:int. p + {q.x = p.x + d} }
ColoredPoint = {
    cl.init : $T_{ColoredPoint}$*color→$T_{ColoredPoint}$ = λt: $T_{ColoredPoint}$, c: color .
        Point.init(t) + { p.color = c },
    cl.new : color→$T_{ColoredPoint}$ = λc:color. cl.init(ColoredPointTemplate, c),
}
ColoredPointTemplate : $T_{ColoredPoint}$ = PointTemplate + {
    c: color = black,
    p.movex = λd:int. p + {q.x = p.x + d, c = red},
    p.draw = λu:1. … }

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
  - Implementation: *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, MultiJava) : can dispatch on any argument

A multimethod on Shape



```
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, encapsulation boundary? Ambiguities!), modular type-checking problematic

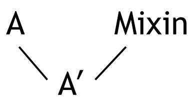
Predicate dispatch

- Multimethods let o.m(a,b,c) dispatch on one property of o, a, b, c (runtime class).

- *Predicate dispatch*: dispatch on general *predicates* over o, a, b, c.
  - Allows selective overriding of methods
  - Exposes assumptions to compiler (can reason about exhaustiveness)
  - Multimethod dispatch a special case

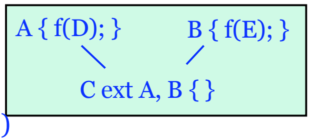
Mixins

- Code is expensive and slow to produce…
- Inheritance, polymorphism, functors are abstraction mechanisms, supporting:
  - modular programming
  - code reuse
  - *extensibility*
- Mixin: mechanism that allows functionality to be "mixed in" to existing class or code base
  - Multimethods: some support
  - Multiple inheritance:
    `class A′ extends A, Mixin`

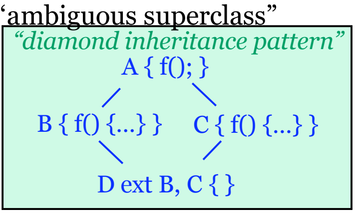


Multiple inheritance

- Multiple "interface inheritance" is mostly-harmless subtyping (e.g. Java, C#)
- Multiple class inheritance ⇒ name conflicts
- Diff. identity, same name:
  - Static error
  - Method renaming (underlying identity)
  - Can hide method at subtype ((A)o).f(D)



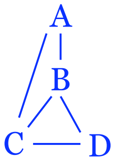
- Same identity, diff. value: real conflict
  - Static error: force override in D
  - Prevent invocation at D or cast to "ambiguous superclass"
- Repeated superclasses: how many copies?
  - C++: 1 if "virtual base class"
  - ...but impl. more complex


"diamond inheritance pattern"

Parametric mixins

```
class Mixin⟨T extends I⟩ extends T {
   new functionality
}
```

- Applying mixin to class C produces a new subclass of C! (not supported by Java 1.5)
- Problem with parametric reuse (also: SML functors): parameters proliferate



```
A[b,c]
B[c,d]
C[b,d]
```

...too much planning, clutter ahead of time!

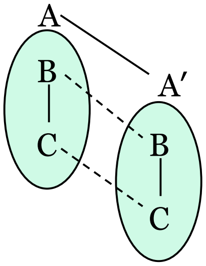
Virtual classes and superclasses

- Ordinary inheritance inherits fields, methods
  - Allows per-class extension of behavior, representation
- Sometimes want to inherit a whole body of code while preserving class relationships
- Virtual (super-)class mechanisms support this

(gBeta, Jx, J&)

```
class A {
    class B {
        void g() { f(); }
        void f();
    }
    class C extends B {
        …
    }
}
```

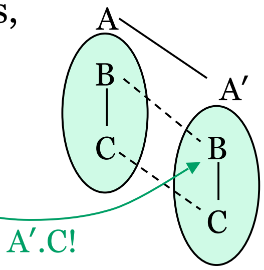
```
class A′ extends A {
    class B {
        int x;
    }
    class C {
        void f() { this.x = 0; }
    }
}
```



Nested inheritance

- Jx extends Java with *nested inheritance* : a type-safe virtual class mechanism
  - Dependent classes: A a = …; a.B b = …
  - *Prefix types* let classes name non-descendant relatives
  - Works with static nested classes, packages

```
class A {
  class B {
       A[this.class].C c = new C();
  }
  class C {…}
}
```



Final issues

- Final is Thursday, December 8, 9AM-11:30AM in Olin Hall 245
- Review session Tuesday, time/location TBA
- Related courses and seminars: CS 412, CS 612, CS 711, PLDG, LCS, Nuprl seminars