CS 6110
Advanced Programming Languages

Andrew Myers

Cornell University

Lecture 38

More about object-oriented languages

1 May 09

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

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

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

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

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

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

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

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

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

Implementing classes (typed)

$T_{Point} = \mu T.\{x\text{: int, y: int, movex: int} \to T\}$

$T_{ColoredPoint} = \mu T.\{x\text{: int, y: int, c: color, movex: int} \to T\text{, draw: } 1 \to 1\} \leq T_{Point}$

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} = { p.x: int = p.x, y: int = p.y,

      p.movex = λd:int. p + {q.x = p.x + d} }

Point
“class”

Need masked
types here!

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(CPTemplate, c),

}

CPTemplate : $T_{ColoredPoint}$ = PointTemplate + {

  p.c: color = p.c,

  p.movex = λd:int. p + {q.x = p.x + d, q.c = p.c},

  p.draw = λu:1. … }

ColoredPoint
“class”

Subtyping vs. inheritance

- Inheritance: an operation on *code*
  - A inherits B = “Code A is just like code B except for the following changes and additions.” A mechanism for *code reuse*.
  - Semantics: A is a distinct *copy* of B
  - Implementation: code of B reused where possible without breaking copying semantics

- Subtyping: a relation on *types*
  - $A \leq B$: “A value of type A can be used wherever a value of type B is expected”

Inheritance w/o subtyping

- Java's "class A extends B"
  - A inherits B *and* A ≤ B
- Can we have A inherits B *without* A ≤ B?
  - Yes: C++ "private" inheritance, Modula-3 type revelations
- Should we have A inherits B without A ≤ B?
  - If we want code reuse without subtyping.
  - **Behavioral subtyping**: A value of type A behaves like a value of type B (satisfies spec of B, not just types)
  - Good uses of subtyping are behavioral subtyping.
  - Good uses of inheritance need not be.

Specialization interface

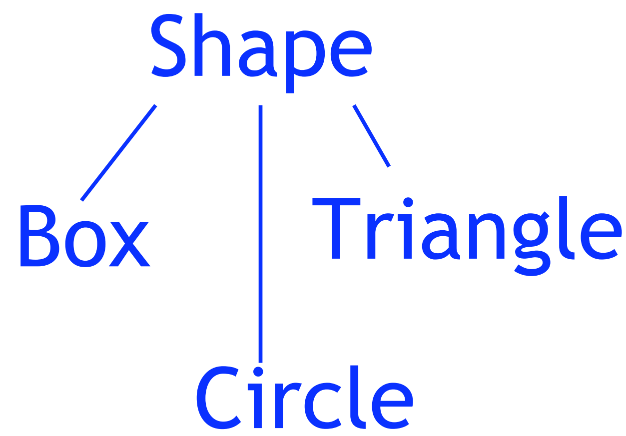
- C++, Java: methods may be marked "final" or "nonvirtual" -- cannot be overridden by subclasses
- Overridable "virtual" methods are a **specialization interface** : contract between class and its subclass.
  - Abstracts with respect to superclasses being *extended* rather than code being called
  - Controls exposure to subclasses
  - Why writing good OO libraries is hard.

Multimethods

- Objects 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) : dispatch on *all* arguments.

A multimethod on Shape

```
class Shape {
   boolean intersects(Shape s);
}
Class Triangle extends Shape {
   boolean intersects(Shape s) {
      if (s instanceof Box) … T/B code
      if (s instanceof Triangle) … T/T code
      if (s instanceof Circle) … T/C code
```



Problem: not extensible

Multimethods

intersects(Box b, Triangle t) { T/B code }
intersects(Triangle t1, Triangle t2) { T/T code }
intersects(Circle c, Triangle t) { T/C code }
Intersects(Shape s, Box b) { S/B code }
... more extensible!

	Shape	Box
Shape	✓	✓
Box	✓	

But...

- Semantics are tricky
  - scope of generic function?
  - encapsulation boundary?
  - ambiguities!
- Modular type-checking problematic -- whole program needed to see ambiguities.

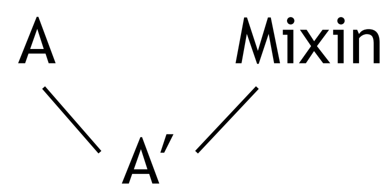
Predicate dispatch

- 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 (use automatic theorem prover to reason about exhaustiveness)
  - Multimethod dispatch is a special case

Mixins

- Code is expensive and slow to produce. Reuse?
- Inheritance, polymorphism, functors are abstraction mechanisms, supporting modular code reuse.
  – Also want *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 via *intersection types*
- Multiple class inheritance ⇒ name conflicts
- Diff. identity, same name:
  - Static error
  - Method renaming (underlying identity)
  - Can hide method at subtype ((A)o).f(D)

```
A { f(D); }        B { f(E); }
        \            /
       C ext A, B { }
```

- 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

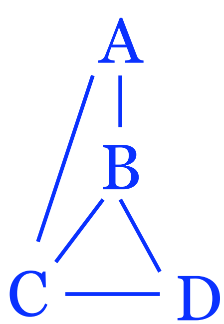
“diamond inheritance pattern”

```
            A { f(); }
           /          \
B { f() {…} }    C { f() {…} }
           \          /
          D ext B, C { }
```

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: ML functors): parameters proliferate




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

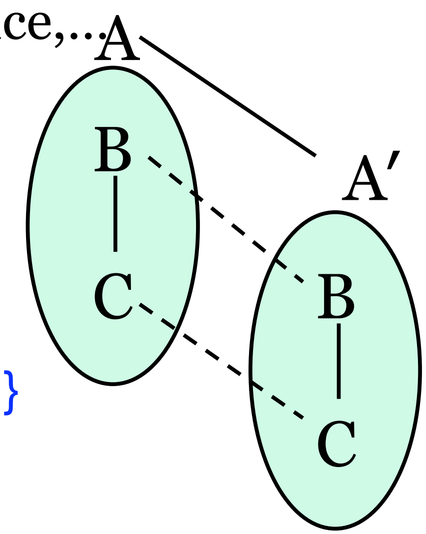
…too much planning, clutter ahead of time!

Family inheritance mechanisms

- 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
- Family inheritance mechanisms support this (gBeta, Jx, J&) -- virtual classes, nested inheritance,…

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

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

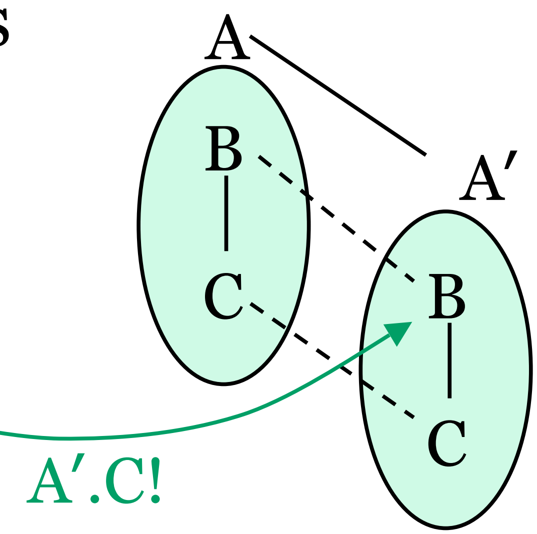


$A'.B \not\leq A.B$ (consider $A'.B.f$)

Nested inheritance

- J& extends Java with *nested inheritance* : a type-safe family inheritance mechanism
  - Dependent classes: A a = ...; a.B b = ...
  - Works with static nested classes, packages
  - Example: composing compilers (package-level mixins)

```
class/package A {
   class B {
        C c = new C();
   }
   class C {…}
}
```



Some things we didn't cover

- Concurrency mechanisms and reasoning techniques
- Abstract interpretation
- Information flow types
- Functors
- Monads
- Intersection/union types
- Singleton types
- Generalized ADTs
- Logic programming
- Polarity for co/contravariant subtyping
- Mechanized proof techniques

What we did have time for

- Thinking about programs and languages formally and precisely
  - Operational semantics
  - Axiomatic semantics
  - Denotational semantics (translation)
  - Type systems
- Studied language features in isolation
- Learned how to prove properties of languages and programs

- Useful?

Final issues

- Final is Monday, May 11
  9AM-11:30AM in 206 Hollister Hall
  - Open book

- Related courses and seminars:
  CS 4120, [CS 6120], [CS 7110], PLDG/LCS