CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2006

Lecture 16

Object-Oriented Programming

Don't Believe the Hype

OOP provides concise ways to build extensible software and exploit a sometimes-useful analogy between interaction of objects in physical systems and interaction of software parts.

It also raises tricky semantic and style issues that require careful PL investigation. (Good thing we're doing it near the end!)

Personally, I am skeptical about:

- The coding factor (X lines/day of accessor methods!)
- The Barnes&Noble factor (certified insane in X days!)
- The intro factor (by week X you can write your own class!)

If it takes an advanced degree to understand objects, I've got the right audience!

So what is OOP?

It seems to look like this... That's a lot; what's the essence?

```
class Point1 extends Object {
 int x;
 int  get_x() { x }
 unit set_x(int y) { self.x = y }
 int distance(Point1 p) { p.get_x() - self.get_x() }
 constructor() { x = 0; }
}
class Point2 extends Point1 {
 int y;
 int get_y() { y }
 int get_x() { 34+super.get_x() }
 constructor() { super(); y=0; }
}
```

OOP can mean many things

- An ADT (private fields)
- Subtyping
- Inheritance, method/field extension, method override
- Implicit this/self
- Dynamic dispatch
- All the above (plus constructor(s)) with 1 class declaration

Let's consider how OO each of these is...

Side question: Is "good design" many combinable constructs or one "do it all" construct?

OO as ADT-focused

Object/class members (fields, methods, constructors) often have *visibilities* and "more private" is sort of "more abstract"

What code can invoke a method/access a field? Other methods in same object, other methods in same class, a subclass, within some other boundary (e.g., a package), any code, ...

With just classes, the only other way to hide a member is upcasting. With *interfaces* (which are more like record types), we can hide members more selectively:

```
interface I { int distance(Point1 p); }
class Point1 { ... I f() { self } ... }
```

Previously we saw objects are a bad match for "strong binary methods" (we'll come back to this)

Records with private fields

Let's assume all fields are visible only to self object. Then objects are just a record of methods with private state.

So far, this is no need for a fancy OO language:

```
type t = { get_x : unit -> int;
           set_x : int -> unit;
           distance : t -> int }
let point1_constructor () =
  let x = ref 0 in
  let rec self =
  { get_x    = (fun () -> !x);
    set_x    = (fun y  -> x := y);
    distance = (fun p  -> p.get_x() - self.get_x() )
  } in self
```

But there's more to it (haven't considered inheritance yet)

Subtyping

Most class-based OO languages "confuse" classes and types:

- If C is a class, then C is a type.
- If C extends D (via declaration), then $C \leq D$.
- Subtyping is (only) the reflexive, transitive closure of this.

Is this novel? If C adds members, that's width subtyping.

This is "by name" subtyping. If classes $C1$ and $C2$ are *incomparable in the class hierarchy* they are *incomparable types*, even if they have the same members.

We will definitely revisit this "subclassing is subtyping" assumption

- It restricts subtyping *and* subclassing!

Subtyping, continued

If C extends D and *overrides* a method of D, what restrictions should we have?

- Argument types contravariant (assume less about arguments)
- Result type covariant (provide more about result)

Many "real" languages are even more restrictive.

Some bend over backward to be more flexible. (Don't!)

It's good we studied this in a simpler setting.

Inheritance and Override

A subclass *inherits* the fields and methods of its superclass. It can override some methods and have special "super" (a.k.a. resend) calls.

This isn't hard in ML either, (if the type system let you reuse field names):

```
let point1_constructor () =
  let x = ref 0 in
  let rec self =
  { get_x    = (fun () -> !x);
    set_x    = (fun y  -> x := y);
    distance = (fun p  -> p.get_x() - self.get_x() )
  } in self
```

continued...

Continued...

```
let point2_constructor () =
    let r = point1_constructor () in
    let rec self =
    {get_x    = (fun () -> 34 + r.get_x());
     distance = r.distance;
     y        = ref 0;
     ... } in self
```

Also have to change point2 code when point1 changes, but often true in OO too ("fragile base class" issues).

Then what is it?

I claim class-based objects are poor (maybe okay) ADTs, same old subtyping, and a little syntactic sugar for extension and override.

So what is that makes OO different in an intellectually interesting way?

Answer: The "late" binding of self and the dynamic dispatch that results. (Fundamentally different rule for what `self` maps to in the environment.)

The difference between `point2_constructor()` and an object of class `Point2` is in the behavior of `distance`.

More on late binding

Late-binding, dynamic dispatch, and open recursion are all closely related ideas. The simplest example I know:

Functional (even still $O(n)$) vs. OO (even now $O(1)$):

```
let c1() = let rec r = {
  even i = if i > 0 then r.odd  (i-1) else true;
  odd  i = if i > 0 then r.even (i-1) else false} in r
let c2() = let r1 = c1() in
  let rec r = {even = r1.even; odd i = i % 2 == 1} in r
class C1 {
  int even(int i) {if i>0 then odd(i-1)  else true}
  int odd(int i)  {if i>0 then even(i-1) else false}}
class C2 extends C1 {
  int odd(int i) {i % 2 == 1} }
```

The big debate

Open recursion:

- Code reuse: improve even by just changing odd
- Superclass has to do less extensibility planning

Closed recursion:

- Code abuse: break even by just breaking odd
- Superclass has to do more abstraction planning

Reality: Both have proved very useful; should probably just argue over "the right default"

Where We're Going

Now we know overriding and dynamic dispatch is the interesting part of the expression language. Now:

- How exactly do we define method dispatch?
- How do we use overriding for extensible software?
- Revisiting "subtyping is subclassing"
  - Why contra/covariance is useful
  - Interfaces or object types for more subtyping
  - Subclassing-not-subtyping for more code reuse

Defining Dispatch

We want correct definitions, not super-efficient compilation techniques.

Methods take "self" as an argument. (Compile down to functions taking an extra argument.) So just need self to refer to right thing.

Approach 1: Each object has a "code pointer" for each method. For object returned by `new C()` where C extends D, use code pointers for D (inductive definition!) but:

- If C overrides `f` replace code pointer for `f`
- If C adds `f`, then add code pointer for `f`
- In method body, self is bound to whole object.

Dispatch continued

Approach 2: Each object has a "type tag". Object returned by `new C()` has tag C. Program state also has a "class table" mapping tags and method-names to code. For dispatch, look up (tag,name) in table. Self still bound to whole object.

Approaches are equivalent, model dynamic dispatch correctly, and are routinely formalized in PL papers.

First approach is "more eager" and consumes more space.

Real implementations get best of both worlds with just a little more complication.

Informal claim: This is hard to explain to freshmen, but in the presence of overriding, no simpler definition is correct.

- And *not* explaining this is not OOP and leads to faulty reasoning if methods are overridden.

Overriding and Hierarchy Design

Subclass writer decides what to override to modify behavior. (Style: Modification should be specialization, but language doesn't check that.)

Superclass writer often has ideas on what will be overridden.

Leads to abstract methods (*must* override) and abstract classes:

- An abstract class has $> \mathbf{0}$ abstract methods
- Overriding an abstract method makes it non-abstract
- Cannot call constructor of an abstract class

Adds no expressiveness (superclass could implement method to raise an exception), but uses static checking to enforce an idiom and saves you a handful of keystrokes.

Overriding for Extensibility

A PL example:

```
class Exp {
  abstract Value eval(Env);
  abstract Typ   typecheck(Ctxt);
}
class IntExp extends class Exp {
  Int   i;
  Value eval(Env e)       { new IntValue(self.i) }
  Typ   typecheck(Ctxt c) { new IntTyp() }
}
```

Example Continued

```
class AddExp extends class Exp {
  Exp e1; Exp e2;
  Value eval(Env e) {
    new IntValue(e1.eval(e).toInt().add(
                 e2.eval(e).toInt())); }
  Typ typecheck(Ctxt c) {
    if(e1.typecheck(c).equals(new IntTyp()) &&
       e2.typecheck(c).equals(new IntTyp()))
      new IntTyp()
    else raise new TypeError() }
}
```

`toInt` may raise an exception (`Value` definition not shown)

"Impure" OO may have a plus primitive (not a method call)

Pure OO continued

Can make everything an object and all primitives method calls (cf. Smalltalk, Ruby, ...)

Example: true and false are objects with ifThenElse methods

```
e1.typecheck(c).equals(new IntTyp()).ifThenElse(
e2.typecheck(c).equals(new IntTyp()).ifThenElse(
(fun () -> new IntTyp()),
(fun () -> throw new TypeError())),
(fun () -> throw new TypeError()))
```

Essentially identical to our encoding of booleans in lecture 6 with explicitly delayed evaluation.

Extending the example

If we add a new variant of expression (e.g., `MultExp`) we need not change any existing code. In ML-style, we do.

If we add a new operation (e.g., `toString`) we need to change `Exp` and all subclasses. In ML, no change to existing code.

If we add a new type of value (e.g., Bool):

- ML patterns need new case but _ may avoid it
- Value subclasses need new method (e.g., `toBool`) but concrete method in superclass (to raise an exception) may localize the change.

Extensibility has many dimensions — most require forethought! (Some work at UW on allowing OO and FP style extension)

Yet more example

Now consider actually adding `MultExp`.

If you have `MultExp` extend `Exp`, you will *copy* `typecheck` from `AddExp`.

If you have `MultExp` extend `AddExp`, you don't copy. The `AddExp` implementer was not expecting that. May be brittle; generally considered bad style.

Best (?) of both worlds by *refactoring* with an abstract `BinIntExp` class implementing `typecheck`. So we *choose* to change `AddExp` when we add `MultExp`.

This intermediate class is a fairly heavyweight way to use a helper function.

Revisiting Subclassing is Subtyping

Recall we have been "confusing" classes and types: C is a class and a type and if C extends D then C is a subtype of D.

Therefore, if C overrides `f`, the type of `f` in C must be a subtype of the type of `f` in D. Just like functions, method-subtyping is contravariant arguments and covariant results.

If code knows it has a C, it can call `f` with "more" arguments and know there are "fewer" results.

Subtyping and Dynamic Dispatch

We defined dynamic dispatch in terms of functions taking self as an argument.

But unlike other arguments, *self is covariant*! (Else overriding method couldn't access new fields/methods.)

This is sound because self must be passed, not another value with the supertype.

This is the key reason encoding OO in a typed $\boldsymbol{\lambda}$-calculus requires ingenuity, fancy types, and/or run-time cost.

(We won't even attempt it.)

More subtyping

With single-inheritance and the class/type confusion, we don't get all the subtyping we want. Example: Taking any object that has an `f` method from `int` to `int`.

Interfaces help somewhat, but class declarations must still *say* they implement an interface.

Object-types bring the flexibility of structural subtyping to OO. For example, `class Exp` has a type with two methods (certain names, certain types) and several supertypes (fewer methods, methods taking more restricted arguments, etc.)

With object-types, "subclassing *implies* subtyping"

More subclassing

Breaking one direction of "subclassing = subtyping" allowed more subtyping (so more code reuse).

Breaking the other direction ("subclassing does not imply subtyping") allows more inheritance (so more code reuse).

Simple idea: If C extends D and overrides a method in a way that makes $C \leq D$ unsound, then $C \not\leq D$. This is useful:

```
class P1 { ... Int get_x(); Int compare(P1); ... }
class P2 extends Point1 { ... Int compare(P2); ... }
```

This is *not* always correct – may need to re-typecheck `get_x` in `P2` in case it assumes a type for `compare`.

Where we are

Summary of last 4 slides: Separating types and classes expands the language, but clarifies the concepts:

- Typing is about interfaces, subtyping about wider interfaces
- Inheritance is about code-sharing

Combining typing and inheritance restricts both.

Where we are going: multiple inheritance, multiple dispatch, bounded polymorphism, classless OO languages.