CS152: Programming Languages

Lecture 21 — Object-Oriented Programming

Dan Grossman
Spring 2011

Don't Believe the Hype

OOP lets you:

1. Build some extensible software concisely
2. Exploit an intuitive analogy between interaction of physical entities and interaction of software pieces

It also:

- Raises tricky semantic and style issues that deserve careful PL study
- Is more complicated than functions
  - Not necessarily worse, but I'm skeptical that all those accessor methods are "productive"

So what is OOP?

OOP "looks like this", but 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; }
}
```


Class-based OOP

In (pure) *class-based OOP*:

1. Everything is an object
2. Objects communicate via messages (handled by methods)
3. Objects have their own state
4. Every object is an instance of a class
5. A class describes its instances' behavior

Why is this *approach* such a popular way to structure software?...

OOP can mean many things

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

Design question: Better to have small orthogonal features or one "do it all" feature?

Anyway, let's consider how "unique to OO" each is. . .

OO as ADT-focused

Fields, methods, constructors often have *visibilities*

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 cast to supertype. With *interfaces* (which are more like record types), we can hide members more selectively:

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

(This all assumes no downcasts, reflection, etc.)

Previously: objects are a bad match for "strong binary methods"

- `distance` takes a `Point1`, not an `I`

Records with private fields

If OOP = functions + private fields, we already have it

- But it's more (e.g., inheritance)

```
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
```


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

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

- Often in favor of static overloading

Some bend over backward to be more flexible

- At expense of run-time checks/casts/failures

It's good we studied this in a simpler setting

Inheritance and Override

Subclasses:

1. *inherit* fields and methods of superclass
2. can *override* methods
3. can use *super* calls (a.k.a. *resends*)

Can we code this up in Caml?

- No because of field-name reuse and lack of subtyping, but ignoring that and trying is illuminating...

Attempting Inheritance

```
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

(* note: adding get_y prevents type-checking in Caml *)
let point2_constructor () =
  let r = point1_constructor () in
  let y = ref 0 in
  let rec self =
  { get_x    = (fun () -> 34 + r.get_x());
    set_x = r.set_x;
    distance = r.distance;
    get_y    = (fun () -> !y) } in self
```


Problems

Small problems:

- Have to change point2 code when point1 changes.
  - But OOPs have tons of "fragile base class" issues too
- No direct access to "private fields" of super-class

Big problem:

- Distance method in point2 code doesn't behave how it does in OOP!!!
  - We do not have late-binding of self (i.e., dynamic dispatch)

The essence

Claim: Class-based object are:

- So-so ADTs
- Same-old record and function subtyping
- Some syntactic sugar for extension and override

And:

- *A fundamentally different rule for what* `self` *maps to in the environment*

More on late binding

Late-binding, dynamic dispatch, and open recursion are all essentially synonyms. The simplest example I know:

Functional (even still $O(n)$)

```
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
```

OO (even now $O(1)$):

```
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.

Next:

- How exactly do we define method dispatch?
- How do we use overriding for extensible software?
- Revisiting "subtyping is subclassing"
  - Usually convenient
  - Not necessary
  - Not always desirable

Defining Dispatch

Focus on 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 bound to the right thing

Approach 1:

- Each object has 1 "code pointer" per method
- For `new C()` where `C` extends `D`:
  - Start with code pointers for `D` (inductive definition!)
  - If `C` adds `m`, add code pointer for `m`
  - If `C` overrides `m`, change code pointer for `m`
- `self` bound to the (whole) object in method body

Dispatch continued

Approach 2:

- Each object has 1 "run-time tag"
- For `new C()` where `C` extends `D`, tag is `C`
- `self` bound to the (whole) object in method body
- Method call to `m` reads tag, looks up (tag,m) in a global table

Both approaches model dynamic-dispatch and are routinely formalized in PL papers. Real implementations a bit more clever.

Difference in approaches only observable in languages with run-time adding/removing/changing of methods.

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

- Else it's not OOP and overriding leads to faulty reasoning

Overriding and Hierarchy Design

Subclass writer decides what to override to modify behavior

- Often-claimed unchecked style issue: overriding should *specialize behavior*

But superclass writer often has ideas on what will be overridden

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

- An abstract class has > 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 or loop forever), but uses static checking to enforce an idiom and saves you a handful of keystrokes

Overriding for Extensibility

A PL example:

```
class Exp {
  abstract Exp eval(Env);
  abstract Typ typecheck(Ctxt);
  abstract Int toInt();
}
class IntExp extends class Exp {
  Int i;
  Exp eval(Env e)        { self }
  Typ typecheck(Ctxt c) { new IntTyp() }
  Int toInt()            { i }
  constructor(Int _i)    { i=_i }
}
```


Example Continued

```
class AddExp extends class Exp {
  Exp e1;
  Exp e2;
  Exp eval(Env e) {
    new IntExp(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 throw new TypeError() }
  Int toInt() { throw new BadCall() }
}
```

"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 the lambda-calculus encoding of booleans

- Closures are just objects with one method, perhaps called "apply"

Extending the example

Now suppose we want `MultExp`

- No change to existing code, unlike ML
- In ML, we would have to "prepare" with an "`Else of 'a`" variant and make `Exp` a type-constructor
  - In general, requires very fancy acrobatics

Now suppose we want a `toString` method

- Must change all existing classes, unlike ML
- In OOP, we would have to "prepare" with a "Visitor pattern"
  - In general, requires very fancy acrobatics

Extensibility has many dimensions — most require forethought!

- See picture...

Variants and Operations

- Given a type with several variants/subtypes and several functions/methods, there's a 2D-grid of code you need:

	Int	Negate	Add	Mult
eval				
toString				
hasZero				

- OO and FP just lay out the code differently!
- Which is more convenient depends on what you're doing and how the variants/operations "fit together"
- Often, tools let you view "the other dimension"
- Opinion: Dimensional structure of code is greater than 2–3, so we'll never have exactly what we want in text

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