CS-XXX: Graduate Programming Languages

Lecture 26 — Classless OOP

Dan Grossman
2012

Classless OOP

OOP gave us code-reuse via inheritance and extensibility via late-binding

Can we throw out classes and still get OOP? Yes

Can it have a type system that prevents "no match found" and "no best match" errors? Yes, but we won’t get there

This is mind-opening stuff if you’ve never seen it

Will make up syntax as we go...

Make objects directly

Everything is an object. You can make objects directly:

let p = [
    field x = 7;
    field y = 9;
    right_quad(){ x.gt(0) && y.gt(0) }  // cf. 0.lte(y)
]

p now bound to an object

▷ Can invoke its methods and read/write its fields

No classes: Constructors are easy to encode

let make_pt = [
    doit(x0,y0) { [ field x=x0; field y=y0;... ] }
]

Inheritance and Override

Building objects from scratch won’t get us late-binding and code reuse. Here’s the trick:

▷ clone method produces a (shallow) copy of an object
▷ method “slots” can be mutable

let o1 = [ // still have late-binding
    odd(x) {if x.eq(0) then false else self.even(x-1)}
    even(x) {if x.eq(0) then true else self.odd(x-1) }
]

let o2 = o1.clone()
o2.even(x) := {(x.mod(2)).eq(0)}

Language doesn’t grow: just methods and mutable “slots”

Can use for constructors too: clone and assign fields

Method Lookup

To find the m method of o:

▷ Look for a slot named m
▷ If not found, look in object held in parent slot

But we still have late-binding: for method in parent slot, we still have self refer to the original o.

Two inequivalent ways to define parent=e1:

▷ Delegation: parent refers to result of e1
▷ Embedding: parent refers to result of e1.clone()

Mutation of result of e1 (or its parent or grandparent or ...) exposes the difference

▷ We’ll assume delegation
Oh so flexible

Delegation is way more flexible (and simple!) (and dangerous!) than class-based OO: The object being delegated to is usually used like a class, but its slots may be mutable

▶ Assigning to a slot in a delegated object changes every object that delegates to it (transitively)
▶ Clever change-propagation but as dangerous as globals and arguably more subtle?

▶ Assigning to a parent slot is “dynamic inheritance” — changes where slots are inherited from

Classes restrict what you can do and how you think, e.g., never thinking of clever run-time modifications of inheritance

Javascript: A Few Notes

▶ Javascript gives assignment “extension” semantics if field not already there. Implementations use indirection (hashtables).
▶ parent is called prototype
▶ new F(...) creates a new object o, calls F with this bound to o, and returns o
▶ No special notion of constructor
▶ Functions are objects too
▶ This isn’t quite prototype-based inheritance, but can code it up:

```javascript
function inheritFrom(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```
▶ No clone (depending on version), but can copy fields explicitly

Rarely what you want

We have the essence of OOP in a tiny language with more flexibility than we usually want

Avoid it via careful coding idioms:
▶ Create trait/abstract objects: Just immutable methods
▶ Analogous role to virtual-method tables
▶ Extend with prototype/template objects: Add mutable fields but don’t mutate them
▶ Analogous role to classes
▶ Clone prototypes to create concrete/normal objects
▶ Analogous role to objects (clone is constructor)

Traits can extend other traits and prototypes other prototypes
▶ Analogous to subclassing

Coming full circle

This idiom is so important, it’s worth having a type system that enforces it

For example, a template object cannot have its members accessed (except clone)

We end up getting close to classes, but from first principles and still allowing the full flexibility when you want it