

CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 18— Bounded Polymorphism and Classless OOP

Revenge of Type Variables

Sorted lists in ML (partial):

```
type 'a slist
make : ('a -> 'a -> int) -> 'a slist
cons : 'a slist -> 'a -> 'a slist
find : 'a slist -> 'a -> 'a option
```

Getting by with OO subtyping:

```
interface Cmp { Int f(Object, Object); }
class SList {
  ... some field definitions ...
  constructor (Cmp x) {...}
  Slist cons(Object x) {...}
  Object find(Object x) {...}
}
```

Wanting Type Variables

Will downcast (potential run-time exception) the arguments to `f` and the result of `find`.

We are not enforcing list-element type-equality.

OO-style subtyping is no replacement for parametric polymorphism; we can have both:

```
interface 'a Cmp { Int f('a,'a); } // Cmp not a type
class 'a SList { // SList not a type (Int SList e.g. is)
  ... some field definitions (can use type 'a) ...
  constructor ('a Cmp x) {...}
  'a Slist cons('a x)      {...}
  'a          find('a)     {...}
}
```

Same Old Story

- Interface and class declarations are *parameterized*; they produce types.
- The constructor is polymorphic
 - For all T, given a T Cmp, it makes a T SList
- If o has type T SList, its cons method:
 - takes a T
 - returns a T SList

No more downcasts; the best of both worlds.

Complications

“Interesting” interaction with overloading and multimethods

```
class B {  
  unit f(Int C x) {...}  
  unit f(String C x) {...}  
}  
class 'a C { unit g(B x) { x.f(self); } }
```

For $T \ C$ where T is neither `Int` nor `String`, can have no match

- Cannot resolve static overloading at compile-time without code duplication and no abstraction (C++)
- To resolve overloading or multimethods at run-time, need run-time type information *including the instantiation* T (C#)
- Could disallow such overloading (Java)
- Or could just reject this sort of call as unresolvable (?)

Wanting bounds

There are compelling reasons to *bound* the instantiation of type variables.

Simple example: Use at supertype without losing that it's a subtype

```
interface I { unit print(); }  
class ('a < I) Logger { // must apply to subtype of I  
    'a item;  
    'a get_it() { syslog(item.print()); item }  
}
```

Without polymorphism or downcasting, client could only use `get_it` result for printing.

Without bound or downcasting, `Logger` could not print.

Fancy Example

With forethought and structural (non-named) subtyping, bounds can avoid some subtyping limitations.

(Example lifted from “A Theory of Objects” Abadi/Cardelli)

```
interface Omnivore { unit eat(Food); }
interface Herbivore { unit eat(Veg); } // Veg <= Food
```

Allowing $\text{Herbivore} \leq \text{Omnivore}$ could make a vegetarian eat meat (unsound)! But this works:

```
interface ('a < Food) Omnivore { unit eat('a); }
interface ('a < Veg) Herbivore { unit eat('a); }
```

If $T \text{ Herbivore}$ is legal, then $T \text{ Omnivore}$ is legal *and* $(T \text{ Herbivore}) \leq (T \text{ Omnivore})!$

Useful for `unit feed('a food, 'a Omnivore animal) {...}`.

Bounded Polymorphism

This “bounded polymorphism” is useful in any language with universal types and subtyping. Instead of $\forall\alpha.\tau$ and $\Lambda\alpha.e$, we have $\forall\alpha < \tau'.\tau$ and $\Lambda\alpha < \tau'.e$:

- Change Δ to be a list of bounds ($\alpha < \tau$) instead of a set of type variables
- In e you can subsume from α to τ'
- $e_1[\tau_1]$ typechecks only if τ_1 “satisfies the bound” in type of e_1

One metatheory drawback: When is $(\forall\alpha_1 < \tau_1.\tau_2) \leq (\forall\alpha_2 < \tau_3.\tau_4)$?

- Contravariant bounds and covariant bodies assuming bound are sound, but makes subtyping undecidable
- Requiring invariant bounds and covariant bodies regains decidability, but obviously allows less subtyping

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.

We will make up syntax as we go along...

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

Extension

But that trick doesn't work to add slots to an object, a common use of subclassing.

Having something like “extend e1 (x=e2)” that mutates e1 to have a new slot is problematic semantically (what if e1 has a slot named x) and for efficiency (may not be room where e1 is allocated)

Instead, we can build a new object with a *special parent slot*:

```
[parent=e1; x=e2]
```

parent is very special because definition of method-lookup (*the* issue in OO) depends on it (else this isn't inheritance)

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:

```
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

Without separating first two *roles*, objects don't share method slots (wastes space), but immutability avoids danger.

Late-binding still makes method-override work correctly.

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.

A word on types

Untyped languages work (the OO of Scheme) – may get a “no match found” exception at run-time. Very flexible.

But we can develop type systems that restrict the language and prevent getting stuck without developing a class system.

Can base types on “derived from the same object,” which can form the basis for multimethods.

- See Cecil (Chambers et al)

Summary: Pure classless OOP a liberating way to think, especially if you learn workarounds in more restrictive languages.