CSE505: Programming Languages

Lecture 18: Bounded polymorphism, Classless OOP

Dan Grossman
Spring 2006
Revenge of Type Variables

- Sorted lists in ML (partial):

  make : (′a -> ′a -> int) -> ′a slist
  cons : ′a slist -> ′a -> ′a slist
  find : ′a slist -> (′a -> bool) -> ′a option

- Sorted lists with OO subtyping (null has any type)

  interface Cmp   { Int f(Object, Object); }  
  interface Pred { Bool g(Object); } 
  class SList { 
    constructor (Cmp x)     { ... } 
    SList  cons (Object x) { ... } 
    Object find (Pred x)   { ... } 
  }
Still want generics

- Will downcast args to f, arg to g, result of a find
- Not enforcing type equality of list elements
- OO subtyping no replacement for parametric polymorphism

So have both:

```haskell
interface 'a Cmp { Int f('a, 'a); } // not a type
interface 'a Pred { Bool g('a); } // not a type
class 'a SList { // not a type (but Int SList is)
    constructor ('a Cmp x)  { ... }
    SList  cons ('a x)     { ... }
    Object find ('a Pred x) { ... }
}
```
Same Old Story

- Interface and class declarations are parameterized; they produce types
- The constructor is polymorphic (for all $T$, given $T \text{ Compare}$, it makes a $T \text{ SList}$)
- If $o$ has type $T \text{ SList}$, its cons method takes $T$ and returns a $T \text{ SList}$

No more downcasts; the best of both worlds
Complications

“Interesting” interaction with overloading or multimethods

```java
class B {
    Int f(Int C x){1}
    Int f(String C x){2}
    Int g(’a C x) { self.f(x) }
}
```

Whether match is found depends on instantiation of ’a

Cannot resolve static overloading at compile-time without
code duplication

At run-time, need run-time type information
   – Including instantiation of type constructors
   – Or restrict overloading enough to avoid it
Wanting bounds

As expected, with subtyping and generics, want bounded polymorphism

Example:

```java
interface I { unit print(); } class ('a: I) Logger {
    'a item;
    'a get() { item.print(); item }
}
```

w/o polymorphism, `get` would return an `I` (not useful)
w/o the bound, `get` could not `send print` to `item`
Fancy example

With forethought, can use bounds to avoid some subtyping limitations
(Example lifted from Abadi/Cardelli text)

/* Herbivore1 ≤ Omnivore1 unsound */
interface Omnivore1 { unit eat(Food); }
interface Herbivore1 { unit eat(Veg); }
/* T Herbivore2 ≤ T Omnivore2 sound for any T */
interface ('a≤Food) Omnivore2 { unit eat('a); }
interface ('a≤Veg) Herbivore2 { unit eat('a); }
/* subtyping lets us pass herbivores to feed
   but only if food is a Veg */
unit feed('a food, 'a Omnivore animal) {
   animal.eat(food);
}
Bounded Polymorphism

Is useful in any language with universal types and subtyping. Instead of $\forall \alpha. \tau$ and $\land \alpha. e$, we have $\forall \alpha \lessdot \tau'. \tau$ and $\land \alpha \lessdot \tau'. e$:

- Change $\Delta$ to be a set of bounds ($\alpha \lessdot \tau$) not just a set of type variables
- In $e$ you can subsume from $\alpha$ to $\tau'$
- $e_1[\tau_1]$ typechecks only if $\tau_1$ “satisfies the bound” in the type of $e_1$

One meta-theory drawback: When is $\forall \alpha \lessdot \tau_1. \tau_2 \leq \forall \alpha \lessdot \tau_3. \tau_4$?
Contravariant bounds (and covariant bodies assuming bound) are sound, but makes subtyping undecidable.
Requiring invariant bounds (more restrictive) regains decidability.
Classless OOP

• OOP gave us code-reuse via inheritance and extensibility via late-binding
• But it also gave us a clunky, heavyweight class and named-type mechanism
• 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” error?
  – yes, but we won’t get there.
• We will make up syntax as we go along
  – with subtitles in JavaScript
• This is mind-opening/bending stuff if you’ve never seen it
Making objects directly

- Everything is an object. You can make objects directly:

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

- No classes. Constructors are easy to encode:

```javascript
let make_p = [
    doit(x0, y0) { [field x=x0; field y=y0; ... ] }
]
```
Making objects directly (JavaScript)

- Everything is an object. You can make objects directly:

```javascript
p = new Object;
p.x = 7;
p.y = 9;
p.right_quad =
function() { return (this.x > 0) && (this.y > 0); }
```

- No classes. Constructors are easy to encode:

```javascript
function point(x, y) {
    this.x = x;
    this.y = y;
    function right_quad(){return (this.x>0 && this.y>0);}
}  
p = new point(7,9);
```
Inheritance and Override

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

- clone method produces a (shallow) copy of an object.
- method slots can be mutable

```javascript
let o1 = [   // still have late binding
    odd = { if x.eq(0) then false else self.even(x-1) }
    even = { if x.eq(0) then true else self.odd(x-1) }
];
let o2 = o1.clone()
o2.even := fun (x) { (x.mod(2)).eq(0) }
```

Language does not grow (just methods and mutable “slots”)
Inheritance and Override (JavaScript)

Building objects from scratch won’t get us late-binding and code reuse. Here is the trick:
- clone method produces a (shallow) copy of an object.
- method slots can be mutable

```javascript
ol = new Object;

ol.odd = function (x)
    {if (x == 0) {return false;} else {return this.even(x-1);}}
ol.even = function (x)
    {if (x == 0) {return true;} else {return this.odd(x-1);}}

o2 = ol.clone(); // not built-in in JavaScript
o2.even = function (x) { return x - (x / 2) == 0; }
```

Language does not grow (just methods and mutable “slots”)
clone() in JavaScript

```javascript
Object.prototype.clone = function() {
    var newObj = new this.constructor();
    newObj.__proto__ = this;  // not in IE
    return newObj;
};
```
Extension

- But that trick does not work to add slots to an object, a common use of subclassing.
- Having something like “\texttt{extend e1 (x=e2)}” that mutates \( e_1 \) to have a new slot is problematic semantically (what if \( e_1 \) has a slot named \( x \)) and for efficiency (may not be room where \( e_1 \) is allocated)
- Instead, we can build a new object with special parent slot:

\[
[parent = e_1; \ x = e_2]
\]

- parent is very special because definition of method lookup (the issue in OO) depends on it.
Extension (in JavaScript)

Extending an object:

```javascript
function Parent() { ... }
function Child(e1) { this.x = e1; }
c = new Child(0);
c.__proto__ = new Parent;  // Not in IE
```

Another way of extending: via constructors

```javascript
function Parent() { ... }
function Child(e1) { this.x = e1; }
Child.prototype = new Parent;

c = new Child(0);
```
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 the 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 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 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 (never thinking of clever run-time modifications of inheritance)
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 (cf. abstract classes)
• extend with *prototype/template* objects: add mutable fields but don’t mutate them (cf. classes)
• clone prototype to create *concrete/normal* objects (cf. constructors)

Traits can extend other traits and prototypes other prototypes (cf. subclasses)
Coming Full Circle

• Without separating first two roles, objects don’t share method slots (wastes space), but immutably 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 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
- Summary: pure classless OO is a liberating way to think, especially if you learn workarounds in more restrictive languages.