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) {...}
}

Dan Grossman

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

Dan Grossman

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 Logger< 'a :< I > { // 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

Issue isn’t special to OOP

Dan Grossman
**Fancy Example** from "A Theory of Objects" Abadi/Cardelli

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

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

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

Useful for unit `feed(\'a\ food, Omnivore<\'a\> 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 \( \Delta \) to be a list of bounds \( (\alpha < \tau) \) instead of a set of type variables
- In \( e \) you can subsume from \( \alpha \) to \( \tau' \)
- \( e_1[\tau_1] \) typechecks when \( \tau_1 \) “satisfies the bound” in type of \( e_1 \)

One limitation: 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

*Will make up syntax as we go...*

**Extension**

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

Having something like “\( \text{extend e1 (x:=e2)} \)” 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*:

\[ \text{[parent=e1; x=e2]} \]

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 \textit{inequivalent} ways to define \( \text{parent}=e1 \):

- Delegation: \( \text{parent} \) refers to result of \( e1 \)
- Embedding: \( \text{parent} \) refers to result of \( e1 \. \text{clone}() \)

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).
- \textit{parent} is called \textit{prototype}
- \texttt{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() {}  // prototype
        F.prototype = o;
        return new F();
    }
    ```

  - No \texttt{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 \textit{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 \textit{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