CSE 341:
Programming Languages

Dan Grossman
Fall 2004
Lecture 26— Static Overloading; Subtype vs. Parametric Polymorphism; Bounded Quantification
Static Overloading

Many OO languages allow methods in the same class to have the same “name” but different argument types. E.g.:

```java
void show(Window w) ...  
void show(DancingBear db) ...  
float distTo(Point p) ...  
float distTo(3DPoint p) ...
```

This complicates slightly the semantics of message send. As before, we:

- Use the class ("run-time type") of the receiver to pick a method.
- Call the method with the receiver bound to self.

But now there are multiple methods with the same name, so we:

- Use the (compile-time) types of the arguments to pick the “best match”.
A lower-level view

Here’s an equivalent way to think about it:

- A method’s name includes the types of its “formal” arguments (e.g., `show$Window`)
- A message send is rewritten with the types of its “actual” arguments after typechecking (e.g., `show(e)` becomes `show$Window(e)` if `e` has type `Window`.

This seems like an “ugly” view, but:

- It’s exactly how static overloading is implemented.
- It means the overloading is really resolved “at compile-time” (long before `e` is evaluated).

But... It interacts poorly with contravariant subtyping on method argument-types, which (I believe) is why Java and C++ use invariant subtyping there.
Static Overloading vs. Multimethods

A very simple difference: Multimethods choose the method at run-time using the class of the actuals.

Example: `e.distTo((Point)(new 3DPoint(3.0,4.0,2.0)))`

The same “no best match” errors arise, but with overloading they arise at compile-time (and can be resolved with explicit subsumption).
Static Typing and Code Reuse

Key idea: Scheme and Smalltalk are different but not *that* different:

- Scheme has arbitrarily nested lexical scope (so does Smalltalk, but only within a method)
- Smalltalk has subclassing and dynamic dispatch (but easy to code up what you need in Scheme)

Java and ML are a bit more different:

- ML has datatypes; Java has classes
- The ML default is immutable
- Java does not have first-class functions (but does have anonymous inner classes)

But the key difference is the *type system*: Java has subtyping; ML has parametric polymorphism (e.g., (\(\text{'a} \times (\text{'a} \to \text{'b}) \to \text{'b}\)).
### What are “forall” types good for?

Some good uses for forall types:

- **Combining functions:**
  
  \[
  \ast (('a\to'b)\ast('b\to'c)) \to ('a\to'c) \ast
  \]
  
  ```
  fun compose (f,g) x = g (f x)
  ```

- **Operating on generic container types:**
  
  ```
  isempty : ('a list) \to bool
  map : (('a list) \ast ('a \to 'b)) \to 'b list
  ```

- **Passing private data (unnecessary with closures):**
  
  ```
  (* ('a * (('a * string) \to int)) \to int *)
  let f (env, g) =
    let val s1 = getString(37)
    val s2 = getString(49)
    in g(env,s1) + g(env,s2) end
  ```
More on private data

(* ('a * ('a * string) -> int)) -> int *)

let f (env, g) =
    let val s1 = getString(37)
    val s2 = getString(49)
    in g(env,s1) + g(env,s2) end

The last point is important in safe, lower-level languages (related to my research), but is unnecessary in ML or Java:

• In ML, just use (string->int) -> int and have the caller pass in fn s => g(env,s) instead of (env,g).
  – This works because the types of free variables do not appear in a function type!

• In Java, just “pass the ’a” as a field in the object that implements the interface.
  – This works because subtyping lets us “forget” we have fields.
What is subtyping good for?

Passing in values with “extra” or “more useful” stuff

//can pass a Pt3D
boolean isXPos(Pt p){ return p.x > 0; }

But in ML, we cannot subsume record types to forget fields. We can write code that “looks like” explicit casting, but it “coerces” values by making new values.

end up encoding coercions to supertypes using regular ML functions that build new values. (See code)
What else is subtyping good for?

In addition to adding “public” fields, we can use it for private state:

```java
interface J { int f(int); }
class MaxEver implements J {
    private int m = 0;
    public int f(int i) { if(i > m) m = i; return m; }
}
```

In ML, we *encode* private state using closures:

```ml
(* closures over mutable fields act like objects,  
   but there is no dynamic dispatch here *)
type J = int -> int
val f : J =
    let val m = ref 0
    in fn i => ((if i > !m then m := i else ()); !m)
end
```
Wanting both

Could one language support subtype polymorphism and parametric polymorphism?

- Sure; and the next generation of OO languages will
- C++ templates are sort of like parametric polymorphism, but they duplicate code, so they're a bit like macros

More interestingly, you may want both at once!

Pt withXZero(Pt p) { return new Pt(0,p.y); }

How could we make a version that worked for subtypes too?
Need for Bounded Quantification

Best effort in Java:

```java
interface I { Pt copy(Pt p); }
Pt withXZero(Pt p, I i) {
    Pt ans = i.copy(p); ans.x = 0; return ans;
}
class A implements I {
    Pt copy(Pt p) { return new Pt3D(p.x,p.y,((Pt3D)p).z); }
    void f(Pt3D p) { Pt3D q = (Pt3D)withXZero(p,self); }
}
```

- copy method has to downcast argument.
- caller of withXZero has to downcast result.
Need for Bounded Quantification

Best effort in ML (Pt and Pt3D defined in lec26.sml)

(* withXZero : ((pt->'a) * ('a->pt) * 'a) -> 'a *)
fun withXZero (to,from,v) = 
  from v in to({x = 0, y = #y (from v)})

fun withXZeroPt p = withXZero(fn x=>x, fn x=>x, p)
fun withXZero3DPt p = withXZero(Pt3D, Pt, p)

• This is tricky.

• Makes 2 temporary “objects” to appease the type system.
interface I<>'a> { 'a copy('<a p); }  
'a withXZero('<a p, I<>'a> i) where '<a <: Pt {  
'a ans = i.copy(p); ans.x = 0; return ans; 
}  
class A implements I<Pt3D> {  
Pt3D copy(Pt3D p) { return new Pt3D(p.x,p.y,p.z); }  
void f(Pt3D p) { Pt3D q = withXZero(p,self); }  
}  

• No downcasts.  
• Without the bound, ans.x = 0 would not typecheck.  
• At call-sites of withXZero, just check the instantiation for '<a is a subtype of Pt
Bounded quantification in general

In general, in a language with subtyping (t1 <: t2) and parametric polymorphism, a useful generalization of \texttt{forall 'a. t} is \texttt{forall 'a <: t1 . t2}. This allows fewer instantiations for 'a.

It does raise interesting “beyond 341” questions, e.g., When is \texttt{forall 'a <: t1. t2} a subtype of \texttt{forall 'a <: t3. t4}?