CSE 341:
Programming Languages

Dan Grossman
Spring 2004
Lecture 25— 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.:

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 suggests static overloading is not very “interesting”, just convenient.

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., ('a * ('a -> 'b)) -> 'b).
What are “forall” types good for?

Some good uses for forall types:

- Combining functions:
  \[
  \lambda ((a \to b) \times (b \to c)) \to (a \to c)
  \]
  \[
  \text{let compose } (f,g) x = g (f x)
  \]

- Operating on generic container types:
  \[
  \text{isempty : (a list) \to bool}
  \]
  \[
  \text{map : (a list) \times (a \to b) \to b list}
  \]

- Passing private data (unnecessary with closures):
  \[
  \lambda (a \times (a \times \text{string}) \to \text{int}) \to \text{int}
  \]
  \[
  \text{let f (env, g) =}
  \]
  \[
  \text{let val s1 = getString(37)}
  \]
  \[
  \text{val s2 = getString(49)}
  \]
  \[
  \text{in g(env,s1) + g(env,s2) end}
  \]
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 the ’a” via a closure (a free variable in the function passed in.
  - 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

```plaintext
bool isXPos(Pt p) { p.x > 0; } // works fine for a Pt3D
```

But in ML, we end up encoding coercive subtyping using regular ML functions that build new values:

```plaintext
type pt = { x : real, y : real}
type pt3D = { x : real, y : real, z : real }
fun isXPos (p:pt) = (#x p) > 0.0
val p3:pt3D = { x=4.0, y=3.0, z=5.0}
fun pt (p:pt3D) = { x=(#x p), y=(#y p)}
val _ = isXPos ((pt) p3)
```
What else is subtyping good for?

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

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

In ML, we encode private state using closures.
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 P(0,p.y); }

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

Here’s an excellent start:

```java
interface I { Pt copy(Pt p); }
Pt withXZero(Pt p, I i) {
    Pt ans = i.copy(p); ans.x = 0; return ans;
}
```

But consider using it for a Pt3D:

- copy method will have to downcast argument.
- user of withXZero will have to downcast result.

Enter bounded quantification:

```java
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;
}
```
How did that work?

- No downcasts.
- Without the bound, ans.x would not typecheck.
- At call-sites of withXZero, just check the instantiation for 'a is a subtype of Pt.

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

Advanced point: When is forall 'a<:t1. t2 a subtype of forall 'a<:t3. t4.

- Sound answer: contravariant bounds; covariant body.
- But that answer makes the subtyping question (for any two types, is one a subtype of the other) undecidable! (1992 result)
- A common restriction in practice is invariant bounds.