CSE 341: Programming Languages

Dan Grossman

Winter 2008

Lecture 24— Named Types; Polymorphism vs. Subtyping; Polymorphism + Subtyping

Named Types

In Java/C++/C#/..., types don't look like
`{t10 m1:(t11,...), ..., tn0 mn(tn1,...)}`.

Instead they look like `C` where `C` is a class or interface.

But everything we learned about subtyping still applies!

Yet the only subtyping is (the transitive closure of) declared subtypes (e.g., `class C extends D implements I,J`).

- Having fewer subtypes is always sound; just allows fewer programs.

Given *types* `D`, `I`, and `J`, ensure objects produced by *class* `C`'s constructors can have subtypes (more methods, contra/co, etc.)

Named vs. Unnamed

For preventing "message not understood", unnamed ("structural") types worked fine.

But many languages have named ("nominal") types.

Which is better is an old argument with points on both sides.

Let's consider whether *subtyping* should be structural ("I have everything you need") or nominal ("I said I was a subtype explicitly")...

Some Fair Points

For structural subtyping:

- Allows more code reuse, while remaining sound.
- Does not require refactoring or adding "`implements` clauses" later when you discover you could share some implementation.

For nominal subtyping:

- Reject more code, which catches bugs and avoids accidental method-name clashes.
- Confusion with classes saves keystrokes and "doing everything twice"?
- Fewer subtypes makes type-checking and efficient code-generation easier.

The Grand Confusion

For convenience, many languages *confuse* classes and types:

- `C` is a class and a type
- If `C` extends `D`, then:
  - instances of the class `C` inherit from the class `D`
  - expressions of type `C` can be subsumed to have type `D`

Do you usually want this confusion? Probably.

Do you always want "subclass implies subtype"?

- No: Consider `distBetween` for `Point` and `3DPoint`.

Do you always want "subtype implies subclass"?

- No: Two classes with `display` methods and no inheritance relationship.

Untangling Classes and Types

- Classes define object behavior; subclassing inherits behavior
- Subtyping defines substitutability
- Most languages require subclasses to be subtypes

Now some other common features make more sense:

- "Abstract" methods:
  - Expand the supertype without providing behavior to subclass
  - Superclass does not implement behavior, so no constructors allowed (an additional static check; the *class* is abstract)
  - The static check is the only fundamental justification
    - Trivial to provide a method that raises an exception
    - In Ruby, just got with message-not-understood
- Interfaces (see previous lecture)

Static Typing and Code Reuse

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

- Scheme has arbitrarily nested lexical scope (so does Ruby via nested blocks within a method)
- Ruby 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
- ML has 1st-class functions (but see Java's inner classes)

But the key difference is the *type system*: ML has parametric polymorphism. Java has subtyping with paramteric polymorphism added on much later (combination greater than the sum of the parts)

What are "forall" types good for?

Some good uses for parametric polymorphism:

- Combining functions:

```
(* (('a->'b)*('b->'c)) -> ('a->'c) *)
fun compose (f,g) x = g (f x)
```

- Operating on generic container types:

```
isempty : ('a list) -> bool
map : (('a list) * ('a -> 'b)) -> 'b list
```

- Passing private data (unnecessary with closures though):

```
(* ('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
```

Subtyping is not right here

If you try to use subtyping for the previous examples, arguments get "upcast" results (to `Object`) get "downcast".

- Inconvenient and error-prone
- Don't get the static checks

In general, when different values can be "any type" but "the same as each other", you want parametric polymorphism.

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; }
```

- Passing private state like with closures

```
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; }
}
```

Parametric polymorphism is not the right thing here (there are sophisticated workarounds)

Wanting both

Could one language support subtyping and parametric polymorphism?

- Sure; Java and C# already do but they also let you "get around the checks" :-(

More interestingly, you may want *both at once*!

A simple (?) example: Making a copy of a mutable list.