CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 24— Method Subtyping; Named Types; Classes vs. Types; (Multiple) Interfaces; Coherence

Recall...

- OO static typing usually means no "message not understood" (except if receiver is `nil`).
- A subsumption relation `t1<:t2` and a subsumption rule can make a sound type system less restrictive.
- For records (objects with only getters/setters), subtypes can add fields or reorder fields, but cannot change the type of a field.

So field types must be *invariant*, else the getter or setter methods in the subtype will have an unsound type:

- If the field becomes a subtype, the getter is wrong (see last lecture).
- If the field becomes a supertype, the setter is wrong.

Methods

But this getter/setter stuff is really just an example of a more general phenomenon: If a supertype has a method m taking arguments of types $t_1, ..., t_n$ and returning an argument of type t_0, what can m take and return in a subtype?

Since this is more general, let's forget about fields:

```
t ::= [t10 m1:(t11,...), ..., tn0 mn(tn1,...)]
```

Now, when is `t1<:t2`?

Method Subtyping, part 1

One sound answer: A subtype can have more methods and rearrange methods, but a method m must take arguments of the same type and return arguments of the same type.

(This answer corresponds to Java and C++ because they also support *static overloading*, which we'll discuss later.)

Can we be less restrictive and still sound?

Yes: We can let the return type be a subtype. Why:

- Some code calling m will "know more" about what's returned.
- Other code calling m will "still work" because of substitutability.

But what about the argument types...

Allowing subtypes is not sound!

Method Subtyping, part 2

What if we allow argument types to be supertypes? It's sound! Why:

- Some code calling m can pass a larger collection of arguments.
- Other code calling m will "still work" because of substitutability.

The jargon: Method subtyping is "contravariant" in argument types and "covariant" in return types.

The point: One method is a subtype of another if the arguments are supertypes and the result is a subtype.

This is easily one of the 5 most important points in this course.

Never, ever think argument-types are covariant. You will be tempted many times. You will never be right. Tell your friends a guy with a PhD jumped up and down!

Connection to FP

Functions and methods are quite similar.

When is `t1->t2` a subtype of `t3->t4`?

When `t3` is a subtype of `t1` and `t2` is a subtype of `t4`.

Why the contravariance? For substitutability—a caller can "still" use a `t3`.

Advanced point: Is there any difference? Yes, remember methods also take a `self` argument bound late.

- And in a subtype, we can assume `self` has the subtype
- But that makes it a covariant argument-type!
- This is sound because cannot change the fact that a particular value (bound to `self`) is passed.
- This is roughly why encoding late-binding in ML is awkward.

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 just 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`).

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

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: Recall `distTo` for `Point` and `3DPoint`.

Do you always want "subtype implies subclass"?

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

Untangling Classes and Types

- Classes define object behavior; subclassing inherits behavior
- Subtyping defines substitutability
- You often want subclasses to be subtypes; most languages give you no choice.

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 because the *class* is abstract)
  - The static-check is the only fundamental justification (trivial to provide a method that raises an exception).
- Interfaces...

Interfaces

A Java interface is just a (named) object type.

By implementing an interface, you get subsumption but no behavior.

- Same thing with "multiple inheritance" when $n - 1$ superclasses have all abstract methods. Should be called "multiple subsumance", but *subsumance* is not a word. :)
- None of the semantic issues we previously discussed with multiple inheritance arise with interfaces.
- But there are two new issues we didn't discuss before because they're about typing...

Multiple Supertype Issues

Most types have multiple supertypes; the issues arise from multiple *immediate* supertypes.

- No least supertypes
  - Java ends up with a pretty *ad hoc* rule for `e1 ? e2 : e3`
- "Coherence" problems: With the subtype relationship a dag, there can be multiple ways to subsume from `C` to `D`.
  - No problem with subtyping as we've seen, but some languages have *coercive* subtyping
  - Coercive subtyping means subsuming `e` from `t1` to `t2` (e.g., `t2 x = e` where `e` has type `t1`) may evaluate `e` to an object and then assign `x` to a different (presumably related) object.

Implicit Coercions

Programmers just love the convenience:

- `Float x = 3;`
- `Int y = x * 1.4;`
- `String s = y;`

Languages end up with lots of rules to specify exactly where and how such coercions occur.

- Example: Narrowing to `int` for `y` happens "after" multiplication.

If we ban implicit narrowing, it's tempting to treat coercions as subtyping and forget all the extra rules.

- `Int<:Float`, `Int<:String`, `Float<:String`
- Language can provide "built-in" coercions and/or let programmers write their own (e.g., overload the cast operator in C++)

Coherence Problems

For `s=y`, a well-defined language will not allow an implementation to choose whether `s` holds `"4"` or `"4.0"`! Solutions:

- Make coercions explicit (don't treat as implicit subtyping) or require only when it's ambiguous.
- Go back to specifying how and where subsumption occurs (complicated rules about "shortest paths" and such?)
- Make it so it doesn't matter what subsumption is used; expression will still be contextually equivalent.
  - Suppose subsumption from `Int` to `String` always adds `".0"`.
  - A coercive subtyping system with this property (path doesn't matter) is called "coherent" (just jargon).
  - Impractical to check this for user-defined coercions, but a good thing for users (that's you) to think about.

Back to Named or Unnamed

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

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

Which is better is a tired old argument, but fortunately it has some interesting intellectual points (unlike emacs vs. vi).

First, frame the question more narrowly: Should subtyping be nominal or structural? (Named types don't preclude structural subtyping, e.g. casting between two otherwise-unrelated interfaces.)

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.
- A simpler system (type names are just an abbreviation and convenient way to write recursive types)

For nominal subtyping:

- Reject more code, which catches bugs and treating unmeaningful method-name clashes as significant.
- Confusion with classes saves keystrokes and "doing everything twice"?
- Fewer subtypes makes type-checking (??) and efficient code-generation easier.