CS152: Programming Languages

Lecture 23 — Advanced Concepts in Object-Oriented Programming

Dan Grossman
Spring 2011

So far. . .

The difference between OOP and "records of functions with shared private state" is *dynamic-dispatch* (a.k.a. *late-binding*) of `self`

(Informally) defined *method-lookup* to implement dynamic-dispatch correctly: use run-time tags or code-pointers

Now: Subclassing vs. subtyping

Then fancy stuff: multiple-inheritance, interfaces, static overloading, multiple dispatch

Next lecture: Bounded polymorphism and classless OOP

Type-Safety in OOP

Should be clearer about what type-safety means…

- "Not getting stuck" has meant "don't apply numbers", "don't add functions", "don't read non-existent record fields", etc.
- Pure OO has only method calls (and maybe field access)
  - Stuck if method-lookup fails (no method matches)
  - Stuck if method-lookup is ambiguous (no best match)

So far only failure is receiver has no method with right name/arity

Revisiting Subclassing is Subtyping

Recall we have been "confusing" classes and types: C is a class and a type and if C extends D then C is a subtype of D

Therefore, if C overrides `m`, the type of `m` in C must be a subtype of the type of `m` in D

Just like functions, method-subtyping is contravariant arguments and covariant results

- If code knows it has a C, it can call methods with "more" arguments and know there are "fewer" results

Subtyping and Dynamic Dispatch

We defined dynamic dispatch in terms of functions taking `self` as an argument

But unlike other arguments, `self` is *covariant*!!

- Else overriding method couldn't access new fields/methods
- Sound because `self` must be passed, not another value with the supertype

This is the key reason *encoding* OO in a *typed* $\boldsymbol{\lambda}$-calculus requires ingenuity, fancy types, and/or run-time cost

- We won't attempt it

More subtyping

With single-inheritance and the class/type confusion, we don't get all the subtyping we want

- Example: Taking any object that has an `m` method from `int` to `int`

Interfaces help somewhat, but class declarations must still *say* they implement an interface

Object-types bring the flexibility of structural subtyping to OOP

With object-types, "subclassing *implies* subtyping"

More subclassing

Breaking one direction of "subclassing = subtyping" allowed more subtyping (so more code reuse)

Breaking the other direction ("subclassing does not imply subtyping") allows more inheritance (so more code reuse)

Simple idea: If C extends D and overrides a method in a way that makes $C \leq D$ unsound, then $C \not\leq D$. This is useful:

```
class P1 { ... Int get_x(); Bool compare(P1); ... }
class P2 extends P1 { ... Bool compare(P2); ... }
```

But this is *not* always correct...

Subclass not a subtype

```
class P1 {
  Int x;
  Int get_x() { x }
  Bool compare(P1 p) { self.get_x() == p.get_x() }
}
class P2 extends P1 {
  Int y;
  Int get_y() { y }
  Bool compare(P2 p) { self.get_x() == p.get_x() &&
                       self.get_y() == p.get_y() }
}
```

- As expected, P2≤P1 is *unsound* (assuming `compare` in `P2` is overriding unlike in Java or C++)

Subclass not a subtype

- Can still inherit implementation (need not reimplement `get_x`)
- We cannot always do this: what if `get_x` called `self.compare`? Possible solutions:
  - Re-typecheck `get_x` in subclass
  - Use a "Really Fancy Type System"

I see little use in allowing subclassing that is not subtyping

But I see much use in understanding that typing is about interfaces and inheritance is about code-sharing

Where we are

Summary of last few slides: Separating types and classes expands the language, but clarifies the concepts:

- Typing is about interfaces, subtyping about broader interfaces
- Inheritance (a.k.a. subclassing) is about code-sharing

Combining typing and inheritance restricts both

- Most OO languages purposely confuse subtyping (about type-checking) and inheritance (about code-sharing), which is reasonble in practice

Multiple Inheritance

Why not allow `class C extends C1,C2,...{...}`
(and C$\leq$C1 and C$\leq$C2)?

What everyone agrees: C++ has it and Java doesn't

All we'll do: Understand some basic problems it introduces and how interfaces get most of the benefits and some of the problems

Problem sources:

- Class hierarchy is a dag, not a tree (not true with interfaces)
- Subtype hierarchy is a dag, not a tree (true with interfaces)

Diamond Issues

If C extends $C1$ and $C2$ and $C1$,$C2$ have a common superclass D (perhaps transitively), our class hierarchy has a diamond

- If D has a field f, should C have one field f or two?
- If D has a method m, $C1$ and $C2$ will have a clash
- If subsumption is coercive (changing method-lookup), how we subsume from C to D affects run-time behavior (incoherent)

Diamonds are common, largely because of types like `Object` with methods like `equals`

Multiple Inheritance, Method-Name Clash

If C extends $C1$ and $C2$, which both define a method m, what does C mean?

Possibilities:

1. Reject declaration of C (Too restrictive with diamonds)
2. Require C to override m (Possibly with *directed resends*)
3. "Left-side" ($C1$) wins (Must decide if upcast to "right-side" ($C2$) coerces to use $C2$'s m or not)
4. C gets both methods (Now upcasts definitely coercive and with diamonds we lose coherence)
5. Other? (I'm just brainstorming based on sound principles)

Implementation Issues

This isn't an implementation course, but many semantic issues regarding multiple inheritance have been heavily influenced by clever implementations

- In particular, accessing members of `self` via compile-time offsets...
- ... which won't work with multiple inheritance unless upcasts "adjust" the `self` pointer

That's one reason C++ has different kinds of casts

Better to think semantically first (how should subsumption affect the behavior of method-lookup) and implementation-wise second (what can I optimize based on the class/type hierarchy)

Digression: Casts

A "cast" can mean many things (cf. C++).

At the language level:

- upcast: no run-time effect until we get to static overloading
- downcast: run-time failure or no-effect
- conversion: key question is round-tripping
- "reinterpret bits": not well-defined

At the implementation level:

- upcast: usually no run-time effect but see last slide
- downcast: usually only run-time effect is failure, but...
- conversion: same as at language level
- "reinterpret bits": no effect (by definition)

Least Supertypes

Consider `if` e_1 `then` e_2 `else` e_3 (or in C++/Java, e_1 ? e_2 : e_3)

- We know e_2 and e_3 must have the same type

With subtyping, they just need a common supertype

- Should pick the least (most-specific) type
- Single inheritance: the closest common ancestor in the class-hierarchy tree
- Multiple inheritance: there may be no least common supertype

Example: $C1$ extends $D1, D2$ and $C2$ extends $D1, D2$

Solutions: Reject (i.e., programmer must insert explicit casts to pick a common supertype)

Multiple Inheritance Summary

- Method clashes (what does inheriting m mean)
- Diamond issues (coherence issues, shared (?) fields)
- Implementation issues (slower method-lookup)
- Least supertypes (may be ambiguous)

Complicated constructs lead to difficult language design

- Doesn't necessarily mean they are bad ideas

Now discuss *interfaces* and see how (and how not) multiple interfaces are simpler than multiple inheritance...

Interfaces

An interface is *just a (named) (object) type*. Example:

```
interface I { Int get_x(); Bool compare(I); }
```

A class can *implement* an interface. Example:

```
class C implements I {
  Int x;
  Int get_x() {x}
  Bool compare(I i) {...} // note argument type
}
```

If C implements I, then $C \leq I$

Requiring *explicit* "implements" hinders extensibility, but simplifies type-checking (a little)

Basically, C implements I if C could extend a class with all *abstract* methods from I

Interfaces, continued

Subinterfaces (`interface J extends I { ...}`) work exactly as subtyping suggests they should

An unnecessary addition to a language with abstract classes and multiple inheritance, but what about single inheritance and multiple interfaces:

```
class C extends D implements I1,I2,...,In
```

- Method clashes (no problem, inherit from D)
- Diamond issues (no problem, no implementation diamond)
- Implementation issues (still a "problem", different object of type I will have different layouts)
- Least supertypes (still a problem, this *is* a typing issue)

Using Interfaces

Although it requires more keystrokes and makes efficient implementation harder, it may make sense (be more extensible) to:

- Use interface types for all fields and variables
- Don't use constructors directly: For class C implementing I, write:

```
I makeI(...)  { new C(...)  }
```

This is related to "factory patterns"; constructors are behind a level of indirection

It is using named object-types instead of class-based types

Static Overloading

So far, we have assumed every method had a different name

- Same name implied overriding and required a subtype

Many OO languages allow the same name for methods with *different argument types*:

```
A f(B x) { ... }
C f(D x, E y) { ... }
F f(G x, H z) { ... }
```

Complicates definition of method-lookup for `e1.m(e2,...,en)`

Previously, we had dynamic-dispatch on `e1`: method-lookup a function of the *class* of the object `e1` evaluates to (*at run-time*)

We now have *static overloading*: Method-lookup is *also* a function of the *types* of `e2,...,en` (*at compile-time*)

Static Overloading Continued

Because of subtyping, multiple methods can match!

"Best-match" can be roughly "Subsume fewest arguments. For a tie, allow subsumption to *immediate* supertypes and recur"

Ambiguities remain (no best match):

- `A f(B)` vs. `C f(B)` (usually rejected)
- `A f(I)` vs. `A f(J)` for `f(e)` where `e` has type T, $T \leq I$, $T \leq J$ and I,J are incomparable (We saw this before)
- `A f(B,C)` vs. `A f(C,B)` for `f(e1,e2)` where $B \leq C$, and `e1` and `e2` have type B

Type systems often reject ambiguous calls or use *ad hoc* rules to give a best match (e.g., "left-argument precedence")

Multiple Dispatch

Static overloading saves keystrokes from shorter method-names

- We know the compile-time types of arguments at each call-site, so we could call methods with different names

Multiple (dynamic) dispatch (a.k.a. multimethods) is much more interesting: Method-lookup a function of the run-time types of arguments

It's a natural generalization: the "receiver" argument is no longer treated differently!

So `e1.m(e2,...,en)` is just sugar for `m(e1,e2,...,en)`

- It wasn't before, e.g., when `e1` is `self` and may be a subtype

Example

```
class A { int f; }
class B extends A { int g; }
Bool compare(A x, A y) { x.f == y.f }
Bool compare(B x, B y) { x.f == y.f && x.g == y.g }
Bool f(A x, A y, A z) { compare(x,y) && compare(y,z) }
```

Neat: late-binding for both arguments to `compare` (choose second method if both arguments are subtypes of B, else first method)

With power comes danger. Tricky question: Can we add "`&& compare(x,z)`" to body of `f` and have an equivalent function?

- With static overloading?
- With multiple dispatch?

Pragmatics

Not clear where multimethods should be defined

- No longer "belong to a class" because receiver isn't special

Multimethods are "more OO" because dynamic dispatch is the essence of OO

Multimethods are "less OO" because without a distinguished receiver the analogy to physical objects is reduced

Nice paper in OOPSLA08: "Multiple Dispatch in Practice"

Revenge of Ambiguity

The "no best match" issues with static overloading exist with multimethods and ambiguities arise at run-time

It's undecidable if "no best match" will happen:

```
// B <= C
A f(B,C) {...}
A f(C,B) {...}
unit g(C a, C b) { f(a,b); /* may be ambiguous */ }
```

Possible solutions:

- Raise exception when no best match
- Define "best match" such that it always exists
- A conservative type system to reject programs that might have a "no best match" error when run

Summary so far

Sketched several advanced issues in class-based OOP:

- multiple inheritance — thorny semantics
- interfaces — less thorny, but no least supertypes
- static overloading — reuse method names, get ambiguities
- multimethods — generalizes late-binding, ambiguities at run-time

But there's still no good way to define a container type such as homogeneous lists

- Add back in parametric polymorphism