CSE341: Programming Languages

Late Binding in Ruby
Multiple Inheritance, Interfaces, Mixins

Alan Borning
Spring 2018

Today

Dynamic dispatch aka late binding aka virtual method calls
- Call to `self.m2()` in method `m1` defined in class `C` can *resolve to* a method `m2` defined in a subclass of `C`
- Important characteristic of OOP

Need to define the semantics of objects and method lookup as carefully as we defined variable lookup for functional programming

Then consider advantages, disadvantages of dynamic dispatch

Recall earlier encoding OOP / dynamic dispatch with functions in Racket (bank account example)

Resolving identifiers

The rules for "looking up" various symbols in a programming language is a key part of the language's definition
- So discuss in general before considering dynamic dispatch

- Haskell: Look up variables in the appropriate environment
  - Key point of closures' lexical scope is defining "appropriate"
- Racket: Like Haskell plus hygienic macros
- Ruby:
  - Local variables and blocks mostly like Haskell and Racket
  - But also have instance variables, class variables, and methods (all more like record fields)

Ruby instance variables and methods

- `self` maps to some "current" object
- Look up local variables in environment of method
- Look up instance variables using object bound to `self`
- Look up class variables using object bound to `self.class`

A syntactic distinction between local/instance/class means there is no ambiguity or shadowing rules
- Contrast: Java locals shadow fields unless use `this.f`

But there is ambiguity/shadowing with local variables and zero-argument no-parenthesis calls
- What does `m+2` mean?
  - Local shadows method if exists unless use `m()+2`
  - Contrast: Java forces parentheses for syntactic distinctions

Method names are different

- self, locals, instance variables, class variables all map to objects
- Have said "everything is an object" but that's not quite true:
  - Method names
  - Blocks
  - Argument lists
- *First-class* values are things you can store, pass, return, etc.
  - In Ruby, only objects (almost everything) are first-class
  - Example: cannot do `e.(if b then m1 else m2 end)`
    - Have to do `if b then e.m1 else e.m2 end`
  - Example: can do `(if b then x else y).m1`

Ruby message lookup

The semantics for method calls aka message sends

`e0.m(e1,…,en)`

1. Evaluate `e0`, `e1`, …, `en` to objects `obj0`, `obj1`, …, `objn`
   - As usual, may involve looking up `self`, variables, fields, etc.
2. Let `C` = the class of `obj0` (every object has a class)
3. If `m` is defined in `C`, pick that method, else recur with the superclass of `C` unless `C` is already `Object`
   - If no `m` is found, call `method_missing` instead
     - Definition of `method_missing` in `Object` raises an error
4. Evaluate body of method picked:
   - With formal arguments bound to `obj1`, …, `objn`
   - With `self` bound to `obj0` -- this implements dynamic dispatch!

Note: Step (3) complicated by mixins: will revise definition later

Java method lookup (very similar)

The semantics for method calls aka message sends

`e0.m(e1,…,en)`

1. Evaluate `e0`, `e1`, …, `en` to objects `obj0`, `obj1`, …, `objn`
   - As usual, may involve looking up `this`, variables, fields, etc.
2. Let `C` = the class of `obj0` (every object has a class)
3. [Complicated rules to pick "the best `m`" using the static types of `e0`, `e1`, …, `en`]
   - Static checking ensures an `m`, and in fact a best `m`, will always be found
   - Rules similar to Ruby except for this *static overloading*
   - No mixins to worry about (interfaces irrelevant here)
4. Evaluate body of method picked:
   - With formal arguments bound to `obj1`, …, `objn`
   - With `this` bound to `obj0` -- this implements dynamic dispatch!

The punch-line again

`e0.m(e1,…,en)`

To implement dynamic dispatch, evaluate the method body with `self` mapping to the receiver

- That way, any `self` calls in the body use the receiver's class,
  - Not necessarily the class that defined the method
- This much is the same in Ruby, Java, C#, Smalltalk, etc.

Comments on dynamic dispatch

- This is why last lecture's **`distFromOrigin2`** worked in **`PolarPoint`**
  - **`distFromOrigin2`** implemented with **`self.x`**, **`self.y`**
  - If receiver's class is **`PolarPoint`**, then will use **`PolarPoint`**'s **`x`** and **`y`** methods because **`self`** is bound to the receiver
- More complicated than the rules for closures
  - Have to treat **`self`** specially
  - May seem simpler only because you learned it first
  - Complicated doesn't imply superior or inferior
    - Depends on how you use it…
    - Overriding does tend to be overused

A simple example

In Ruby (and other object-oriented languages), subclasses can change the behavior of methods they don't override

```
class A
  def even x
    if x==0 then true  else odd  (x-1) end
  end
  def odd x
    if x==0 then false else even (x-1) end
  end
end
class B < A  # improves odd in B objects
  def even x ; x % 2 == 0 end
end
class C < A  # breaks odd in C objects
  def even x ; false end
end
```

The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in subclasses even if it is not overridden
  - Maybe on purpose, maybe by mistake

- Makes it harder to reason about "the code you're looking at"
  - Can avoid by disallowing overriding (Java **`final`**) of helper methods you call
- Makes it easier for subclasses to specialize behavior without copying code
  - Provided method in superclass isn't modified later

What next?

Have used classes for OOP's essence: inheritance, overriding, dynamic dispatch

Now, what if we want to have more than *just 1 superclass*

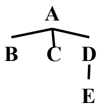
- *Multiple inheritance*: allow > 1 superclasses
  - Useful but has some problems (see C++)
- Java-style *interfaces*: allow > 1 types
  - Mostly irrelevant in a dynamically typed language, but fewer problems
- Ruby-style *mixins*: 1 superclass; > 1 method providers
  - Often a fine substitute for multiple inheritance and has fewer problems

Multiple Inheritance

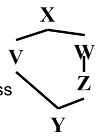
- If inheritance and overriding are so useful, why limit ourselves to one superclass?
  - Because the semantics is often awkward (next couple of slides)
  - Because it makes static type-checking harder (not discussed)
  - Because it makes efficient implementation harder (not discussed)
- Is it useful? Sure!
  - Example: Make a `ColorPt3D` by inheriting from `Pt3D` and `ColorPt` (or maybe just from `Color`)
  - Example: Make a `StudentAthlete` by inheriting from `Student` and `Athlete`
  - With single inheritance, end up copying code or using non-OOP-style helper methods

Trees, dags, and diamonds

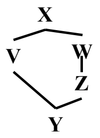
- Note: The phrases *subclass*, *superclass* can be ambiguous
  - There are *immediate* subclasses, superclasses
  - And there are *transitive* subclasses, superclasses
- Single inheritance: the *class hierarchy* is a tree
  - Nodes are classes
  - Parent is immediate superclass
  - Any number of children allowed



- Multiple inheritance: the class hierarchy no longer a tree
  - Cycles still disallowed (a directed-acyclic graph)
  - If multiple paths show that *X* is a (transitive) superclass of *Y*, then we have *diamonds*



What could go wrong?



- If *V* and *Z* both define a method `m`, what does *Y* inherit? What does `super` mean?
  - *Directed resends* useful (e.g., `Z::super`)
- What if *X* defines a method `m` that *Z* but not *V* overrides?
  - Can handle like previous case, but sometimes undesirable (e.g., `ColorPt3D` wants `Pt3D`'s overrides to "win")
- If *X* defines fields, should *Y* have one copy of them (`f`) or two (`V::f` and `Z::f`)?
  - Turns out each behavior is sometimes desirable (next slides)
  - So C++ has (at least) two forms of inheritance

3DColorPoints

If Ruby had multiple inheritance, we would want `ColorPt3D` to inherit methods that share one `@x` and one `@y`

```
class Pt
  attr_accessor :x, :y
  …
end
class ColorPt < Pt
 attr_accessor :color
 …
end
class Pt3D < Pt
 attr_accessor :z
 … # override methods like distance?
end
class ColorPt3D < Pt3D, ColorPt # not Ruby!
end
```

ArtistCowboys

This code has **`Person`** define a pocket for subclasses to use, but an **`ArtistCowboy`** wants *two* pockets, one for each **`draw`** method

```
class Person
  attr_accessor :pocket
  …
end
class Artist < Person # pocket for brush objects
 def draw # access pocket
 …
end
class Cowboy < Person # pocket for gun objects
 def draw # access pocket
 …
end
class ArtistCowboy < Artist, Cowboy # not Ruby!
end
```

Java interfaces

Recall (?), Java lets us define *interfaces* that classes explicitly *implement*

```
interface Example {
  void   m1(int x, int y);
  Object m2(Example x, String y);
}

class A implements Example {
  public void m1(int x, int y) {…}
  public Object m2(Example e, String s) {…}
}
class B implements Example {
  public void m1(int pizza, int beer) {…}
  public Object m2(Example e, String s) {…}
}
```

What is an interface?

```
interface Example {
  void   m1(int x, int y);
  Object m2(Example x, String y);
}
```

- An interface is a type!
  - Any implementer (including subclasses) is a *subtype* of it
  - Can use an interface name wherever a type appears
  - (In Java, classes are also types in addition to being classes)
- An implementer type-checks if it defines the methods as required
  - Parameter names irrelevant to type-checking; it's a bit strange that Java requires them in interface definitions
- A user of type **`Example`** can objects with that type have the methods promised
  - I.e., sending messages with appropriate arguments type-checks

Multiple interfaces

- Java classes can implement any number of interfaces
- Because interfaces provide no methods or fields, no questions of method/field duplication arise
  - No problem if two interfaces both require of implementers and promise to clients the same method
- Such interfaces aren't much use in a dynamically typed language
  - We don't type-check implementers
  - We already allow clients to send any message
  - Presumably these types would change the meaning of **`is_a?`**, but we can just use **`instance_methods`** to find out what methods an object has

Why no interfaces in C++?

If you have multiple inheritance and abstract methods (called pure virtual methods in C++), there is no need for interfaces

- *Abstract method*: A method declared but not defined in a class. All instances of the (sub)class must have a definition
- *Abstract class*: Has one or more abstract methods; so disallow creating instances of this exact class
  - Have to subclass and implement all the abstract methods to create instances
- Little point to abstract methods in a dynamically typed language
- In C++, instead of an interface, make a class with all abstract methods and inherit from it – same effect on type-checking

Mixins

- A *mixin* is (just) a collection of methods
  - Less than a class: no fields, constructors, instances, etc.
  - More than an interface: methods have bodies
- Languages with mixins (e.g., Ruby modules) typically allow a class to have one superclass but any number of mixins
- Semantics: *Including a mixin makes its methods part of the class*
  - Extending or overriding in the order mixins are included in the class definition
  - More powerful than helper methods because mixin methods can access methods (and instance variables) on `self` not defined in the mixin

Example

```
module Doubler
  def double
    self + self # assume included in classes w/ +
  end
end
class String
  include Doubler
end
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other
    ans = AnotherPt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
end
```