CSE 341: Programming Languages

Dan Grossman

Spring 2008

Lecture 22— Defining and Implementing Dynamic-Dispatch

Where are We?

In 8 weeks, we've picked up enough ML, Scheme, and Ruby to talk intelligently about modern, general-purpose PLs.

- Congratulations to us!

Now we need to:

- Consider OO semantics as carefully as we did FP semantics
- Consider various OO extensions and design decisions
- Consider OO type systems as carefully as we did FP type systems
- Compare OO and FP, specifically extensibility and polymorphism
- Discuss memory management and garbage collection

Today: Ruby look-up rules, a lower-level view of dynamic dispatch

Look-up rules

How we *resolve* various "symbols" is a key part of language definition.

- In many ways, FP boils down to first-class functions, *lexical scope*, and immutability.

In Ruby, we *syntactically distinguish* instance fields (`@x`) and class fields (`@@x`) from method/block variables (`x`) and method names (`x`).

- Unlike Java, no shadowing of fields.
  - Makes lookup rules for fields easy.
- Unlike Java, can shadow method names (does `m+2` read a variable or call a method)
  - Rather clumsy since variables aren't declared.
  - Will ignore confusion today; see book for the rules.

"First-class"

If something can be computed, stored in fields, passed as arguments, returned as results, etc., we say it is "first-class".

All objects in Ruby are first-class (and most things are objects).

These things are not:

- Message names: Must write `if b then x.m else x.n end`, not `x.(if b then m else n end)`
- Blocks (hence conversion to `Proc` instances)
- Argument lists

Variable lookup

To *resolve* a variable (e.g., `x`):

- Inside a code block that defines `x` (`{|x| e}`), `x` resolves to the local variable of the block (i.e., the argument).
- Else inside a code block, `x` resolves to an `x` that is defined in the enclosing method.
  - Lexical scope, just like in ML and Scheme
  - Ruby implementation must build closures (those pairs of code and environment you built for homework 5)

Nothing really new here

Message lookup

To resolve a message (e.g., `m`):

- A message is sent to an object (e.g., `e.m`), so first evalaute expression `e` to an object `obj`.
- Get the class of `obj` (e.g., `A`) (every object has a class).
- If `m` is defined in `A` (check instance methods first, then class methods), invoke that method, else recur with superclass of `A`.
  - Will slightly complicate this story next time with mixins

What about `self`?

As always, evaluation takes place in an environment.

In every environment, `self` is always bound to some object. (This determines message resolution for `self` and `super`.)

Key principles of OOP:

- Inheritance and override (last slide)
- Private fields (just abstraction)
- *The semantics of message send*

To send `m` to `obj` means evaluate the body of the method `m` resolves to for `obj` in an environment with argument names mapped to actual arguments *and `self` bound to `obj`*.

That last phrase is exactly what "late-binding", "dynamic dispatch", and "virtual function call" mean. It is why code defined in superclasses can invoke code defined in subclasses.

A Simple Example, part 1

```
fun even x = if x=0 then true else odd (x-1)
and odd  x = if x=0 then false else even (x-1)

(* does not change behavior of odd *)
fun even x = (x mod 2) = 0

(* neither does this *)
fun even x = false
```

A Simple Example, part 2

```
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
  def even x # changes B's odd too!
     x % 2 = 0
  end
end
```

Some Perspective on Late-Binding

Some opinions:

- Late-binding makes a more complicated semantics
  - Ruby without `self` is easier to define and reason about
  - It takes months in 142 to get to where we can explain it
  - It makes it harder to reason about programs
- But late-binding is often an elegant pattern for reuse
  - OO without `self` is not OO
  - Late-binding fits well with the "object analogy"
  - Late-binding can make it easier to localize specialized code even when other code wasn't expecting specialization
    * More reuse/abuse.

A Lower-Level View

Ruby clearly encourages late-binding with its message-send semantics.

But a definition in one language is often a pattern in another...

We can simulate late-binding in Scheme easily enough

And sketch how compilers/interpreters implement objects

- A naive but *accurate* view of implementation can give an alternate way to reason about programs

The Key Idea

The key to implementing late-binding is extending all the methods to take an extra argument (for `self`).

So an object is implemented as a record holding methods and fields, where methods are passed self explicitly.

And message-resolution always uses `self`.

What about classes and performance?

This approach, while a fine pattern, has some problems:

- It doesn't model Ruby, where methods can be added/removed from classes dynamically and an object's class determines behavior.
- It is space-inefficient: all objects of a class have the same methods.
- It is time-inefficient: message-send should be constant-time, not list traversals.

We fix the first two by adding a level of indirection: all instances made from same "constructor" share list of methods, and we can mutate the list.

We fix the third with better data structures (array or hash) and various tricks (so subclassing works).

Static typing gets in the way

- We have seen late-binding as a Scheme pattern
- In reality, we have learned roughly how OO implementations do it, without appealing to assembly code (where it really happens)
- Using ML instead of Scheme would have been a pain:
  - The ML type system is "unfriendly" for `self`.
  - We would have roughly taken the "embed Scheme in ML" approach, giving *every* object the same ML type.
  - But to be fair, basic OO typing (coming soon) is "unfriendly" to ML datatypes, first-class functions, and parametric polymorphism.