CSE 413 Autumn 2008

Objects & Dynamic Dispatch

Plan

- We’ve learned a great deal about functional and object-oriented programming
- Now,
  - Look at semantics and principles more carefully
  - Look at O-O and functional programming – what are the essential differences and similarities

Look-Up Rules (1)

- Key idea in any language: how are “symbols” (names, identifiers) resolved
- Functional programming – first-class functions, lexical scope, immutability (i.e., don’t use set!)

Look-Up Rules in Ruby (2)

- In Ruby, use *syntactic* distinctions
  - instance fields (@x), class fields (@@x) vs method/block variables and method names (x)
- No shadowing of fields, unlike Java
- Can shadow method names with variables
  - So: is m+2 a variable lookup or a method call?
  - We won’t worry about this for the most part

“First-Class”

- If something can be computed, stored in fields/ variables, used as arguments, returned as results, we say it is “first-class”
- All objects in Ruby are first-class
  - & most things are objects
- Things that are not:
  - Message names
    - can’t write x.(if b then m else n end)
  - Blocks (but procs are)
  - Argument lists

Variable Lookup in Ruby

- To resolve a variable (e.g., x)
  - Inside a code block { |x| e }, x resolves to local variable (the argument)
    - Not strictly true in Ruby 1.8 & earlier if x already exists in the surrounding block
  - Else x resolves to x defined in enclosing method
    - Lexical scope, as in Scheme
    - Implies Ruby implementation needs to build closures at least some of the time

Message Lookup in Ruby

- To resolve a message (e.g., m)
  - All messages are sent to an object (e.g., e.m), so first evaluate e to get object obj
  - Get class of obj (e.g., A)
    - Every object has a class and carries a reference to the corresponding class object
  - If m defined in A (instance methods first, then class methods), call it, otherwise recursively look in superclasses
    - Mixins complicate this somewhat (later)
    - If no match up the chain, method not found error

What is self?

- Evaluation always takes place in an environment
- self is always bound to some object in any environment
  - Determines resolution for self and super

OOP Principles

- Inheritance and override
- Private fields (just abstraction)
- *The semantics of message send*
  - To send m to obj means evaluate body of method m resolves to in environment where parameters map to arguments *and self is bound to obj*
  - This is exactly “late binding”, “dynamic dispatch”, “virtual function call”
    - And why superclass code can call code defined in subclasses

An Example (Scheme)

- Suppose this is defined at top-level

  ```
  (define (even x) (if (= x 0) #t (odd (- x 1))))
  (define (odd x)   (if (= x 0) #f (even (- x 1))))
  ```

- Suppose we evaluate (odd 42) in an inner scope where even is defined to be

  ```
      (define (even x) (= 0 (modulo x 2)))
  ```

  - Nothing changes – odd calls original even (static scope)

Example (Ruby – Subclasses)

```
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
    x % 2 == 0
  end
end
```

- Now odd, as well as even, is changed for instances of B

Perspectives on Late Binding

- More complicated semantics
  - Ruby without self is easier to define and reason about
  - Seems “natural” only because you have had months of this in previous courses
  - Hard to reason about code – “which method is really called here?”

Perspectives on Late Binding

- But often an elegant pattern for reuse
  - OO without self is not OO
  - Fits well with “object analogy”
  - Can make it easier to add/localize specialized code even when other code wasn’t written to be specialized
    - More reuse/abuse

Lower-Level View

- A definition in one language is often a pattern in another…
- Can simulate late-binding in Scheme easily enough
- And it provides a mental model for how objects and late binding are implemented
  - Naïve, but accurate view can give a way to reason about programs, even if “real” implementations contain more sophisticated engineering

Late Binding in Scheme

- Key idea: extend all methods to take an extra argument (i.e., self)
- An object is a record (closure) holding methods and fields
- Self is passed as an explicit argument everywhere
- Message resolution always uses self

Is This Real?

- It’s a fine pattern, but…
  - It doesn’t model Ruby, where methods can be added/removed dynamically and an object’s class determines behavior
    - In the example we model “classless” objects
  - Space inefficient – duplicate methods
  - Time inefficient – method lookup needs to be constant time in real systems

Better Engineering, Better Reality

- To model classes, add a level of indirection: all instances created by the same “constructor” share a list of methods
  - And for Ruby, we can change the list
- Use better data structures (array or hash) to get constant-time method dispatch
  - And add tricks so subclassing works