CSE341: Programming Languages

Lecture 21
Dynamic Dispatch Precisely, and Manually in Racket

Eric Mullen
Autumn 2019
Dynamic dispatch

Dynamic dispatch
- Also known as *late binding* or *virtual methods*

- Call `self.m2()` in method `m1` defined in class `C` can *resolve* to a method `m2` defined in a subclass of `C`

- Most unique characteristic of OOP

Need to define the semantics of *method lookup* as carefully as we defined *variable lookup* for our PLs
Review: variable lookup

Rules for “looking things up” is a key part of PL semantics

• ML: Look up variables in the appropriate environment
  – Lexical scope for closures
  – Field names (for records) are different: not variables

• Racket: Like ML plus let, letrec

• Ruby:
  – Local variables and blocks mostly like ML and Racket
  – But also have instance variables, class variables, methods
    (all more like record fields)
    • Look up in terms of self, which is special
**Using self**

- `self` maps to some “current” object

- Look up instance variable `@x` using object bound to `self`

- Look up class variables `@@x` using object bound to `self.class`

- Look up methods…
Ruby method lookup

The semantics for method calls also known as message sends
\[ e_0 \cdot m(e_1, \ldots, e_n) \]

1. Evaluate \( e_0, e_1, \ldots, e_n \) to objects \( \text{obj}_0, \text{obj}_1, \ldots, \text{obj}_n \)
   - As usual, may involve looking up \texttt{self}, variables, fields, etc.
2. Let \( C \) be the class of \( \text{obj}_0 \) (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 \texttt{Object}
   - If no \( m \) is found, call \texttt{method_missing} instead
     - Definition of \texttt{method_missing} in \texttt{Object} raises an error
4. Evaluate body of method picked:
   - With formal arguments bound to \( \text{obj}_1, \ldots, \text{obj}_n \)
   - With \texttt{self} bound to \( \text{obj}_0 \) -- this implements dynamic dispatch!

Note: Step (3) complicated by \textit{mixins}: will revise definition later
Punch-line again

\[
e_0.m(e_1, \ldots, e_n)
\]

To implement dynamic dispatch, evaluate the method body with \texttt{self} mapping to the \textit{receiver} (result of \texttt{e0})

- That way, any \texttt{self} calls in body of \texttt{m} use the receiver's class,
  - Not necessarily the class that defined \texttt{m}

- This much is the same in Ruby, Java, C#, Smalltalk, etc.
Comments on dynamic dispatch

• This is why `distFromOrigin2` worked in `PolarPoint`

• More complicated than the rules for closures
  – Have to treat `self` specially
  – May seem simpler only if you learned it first
  – Complicated does not necessarily mean inferior or superior
**Static overloading**

In Java/C#/C++, method-lookup rules are similar, but more complicated because > 1 methods in a class can have same name

- Java/C/C++: Overriding only when number/types of arguments the same
- Ruby: same-method-name always overriding

Pick the “best one” using the static (!) types of the arguments

- Complicated rules for “best”
- Type-checking error if there is no “best”

Relies fundamentally on type-checking rules

- Ruby has none
A simple example, part 1

In ML (and other languages), closures are closed

\[
\begin{align*}
\text{fun even } x &= \text{if } x=0 \text{ then } \text{true} \text{ else } \text{odd } (x-1) \\
\text{and odd } x &= \text{if } x=0 \text{ then } \text{false} \text{ else } \text{even } (x-1)
\end{align*}
\]

So we can shadow `odd`, but any call to the closure bound to `odd` above will “do what we expect”

– Does not matter if we shadow `even` or not

\[
\begin{align*}
\text{fun even } x &= \text{if } x=0 \text{ then } \text{true} \text{ else } \text{odd } (x-1) \\
\text{and odd } x &= \text{if } x=0 \text{ then } \text{false} \text{ else } \text{even } (x-1)
\end{align*}
\]

(* does not change odd – too bad; this would improve it *)

\[
\text{fun even } x = (x \text{ mod } 2) = 0
\]

(* does not change odd – good thing; this would break it *)

\[
\text{fun even } x = \text{false}
\]
A simple example, part 2

In Ruby (and other OOP languages), subclasses can change the behavior of methods they do not 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
– Observable behavior includes calls-to-overridable methods

• So harder to reason about “the code you're looking at”
  – Can avoid by disallowing overriding
    • “private” or “final” methods

• So easier for subclasses to affect behavior without copying code
  – Provided method in superclass is not modified later
Manual dynamic dispatch

Now: Write Racket code with little more than pairs and functions that acts like objects with dynamic dispatch

Why do this?
   - (Racket actually has classes and objects available)

   • Demonstrates how one language's semantics is an idiom in another language
   • Understand dynamic dispatch better by coding it up
      - Roughly how an interpreter/compiler might

Analogy: Earlier optional material encoding higher-order functions using objects and explicit environments
Our approach

Many ways to do it; our code does this:

- An “object” has a list of field pairs and a list of method pairs

  (struct obj (fields methods))

- Field-list element example:

  (mcons 'x 17)

- Method-list element example:

  (cons 'get-x (lambda (self args) ...))

Notes:

- Lists sufficient but not efficient
- Not class-based: object has a list of methods, not a class that has a list of methods [could do it that way instead]
- Key trick is lambdas taking an extra self argument
  - All “regular” arguments put in a list args for simplicity
A point object bound to \( x \)

\[
\begin{array}{c}
\text{'x} & -4 \\
\text{mcar} & \text{mcdr} \\
\end{array}
\quad
\begin{array}{c}
\text{'y} & 0 \\
\text{mcar} & \text{mcdr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{car} & \text{cdr} \\
\end{array}
\quad
\begin{array}{c}
\text{car} & \text{cdr} \\
\end{array}
\quad
\begin{array}{c}
\text{car} & \text{cdr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{'()'} \\
\text{car} & \text{cdr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{fields, methods} \\
\end{array}
\quad
\begin{array}{c}
\lambda(\text{self args})… \\
\end{array}
\quad
\begin{array}{c}
\lambda(\text{self args})… \\
\end{array}
\quad
\begin{array}{c}
\lambda(\text{self args})… \\
\end{array}
\]

\[
\begin{array}{c}
\text{'get-x} \\
\text{car} & \text{cdr} \\
\end{array}
\quad
\begin{array}{c}
\text{'set-x} \\
\text{car} & \text{cdr} \\
\end{array}
\quad
\begin{array}{c}
\text{'distToOrigin} \\
\text{car} & \text{cdr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{car} & \text{cdr} \\
\end{array}
\quad
\begin{array}{c}
\text{car} & \text{cdr} \\
\end{array}
\quad
\begin{array}{c}
\text{car} & \text{cdr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{'()} \\
\text{car} & \text{cdr} \\
\end{array}
\]
Key helper functions

Now define plain Racket functions to get field, set field, call method

```
(define (assoc-m v xs)  
  ...) ; assoc for list of mutable pairs
(define (get obj fld)  
  (let ([[pr (assoc-m fld (obj-fields obj))]])  
    (if pr (mcdr pr) (error ...))))
(define (set obj fld v)  
  (let ([[pr (assoc-m fld (obj-fields obj))]])  
    (if pr (set-mcdr! pr v) (error ...))))
(define (send obj msg . args)  
  (let ([[pr (assoc msg (obj-methods obj))]])  
    (if pr ((cdr pr) obj args) (error ...))))
```
(send x 'distToOrigin)

Evaluate body of λ(self args)...
with self bound to entire object →
(and args bound to '() )
Constructing points

- Plain-old Racket function can take initial field values and build a point object
  - Use functions `get`, `set`, and `send` on result and in “methods”
  - Call to self: `(send self 'm ...)`
  - Method arguments in `args` list

```scheme
(define (make-point _x _y)
  (obj
    (list (mcons 'x _x)
      (mcons 'y _y))
    (list (cons 'get-x (λ(self args)(get self 'x)))
      (cons 'get-y (λ(self args)(get self 'y)))
      (cons 'set-x (λ(self args)(...)))
      (cons 'set-y (λ(self args)(...)))
      (cons 'distToOrigin (λ(self args)(...))))))
```
“Subclassing”

• Can use `make-point` to write `make-color-point` or `make-polar-point` functions (see code)

• Build a new object using fields and methods from “super” “constructor”
  – Add new or overriding methods to the `beginning of the list`
    • `send` will find the first matching method
  – Since `send` passes the entire receiver for `self`, dynamic dispatch works as desired
Why not ML?

• We were wise not to try this in ML!

• ML's type system does not have subtyping for declaring a polar-point type that “is also a” point type
  – Workarounds possible (e.g., one type for all objects)
  – Still no good type for those self arguments to functions
    • Need quite sophisticated type systems to support dynamic dispatch if it is not built into the language

• In fairness, languages with subtyping but not generics make it analogously awkward to write generic code