CSE341: Programming Languages

Lecture 21
Late Binding;
OOP as a Racket Pattern

Dan Grossman

Fall 2011

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`**
- Most unique 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

Then encoding OOP / dynamic dispatch with pairs and functions

- In Racket
- Complement Lecture 9's encoding of closures in Java or C

Resolving identifiers

The rules for "looking up" various symbols in a PL is a key part of the language's definition

- So discuss in general before considering dynamic dispatch

- ML: Look up variables in the appropriate environment
  - Key point of closures' lexical scope is defining "appropriate"
  - Field names (for records) are different

- Racket: Like ML plus let, letrec, and hygienic macros

- Ruby:
  - Local variables and blocks mostly like ML 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 (more like ML field 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, part 1

In ML (and other languages), closures are closed

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

So we can shadow **odd**, but any call to the closure bound to **odd** above will "do what we expect"

- Doesn't matter if we shadow **even** or not

```
(* does not change odd - too bad; this would
   improve it *)
fun even x = (x mod 2)=0
```

```
(* does not change odd - good thing; this would
   break it *)
fun even x = false
```

A simple example, part 2

In Ruby (and other 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

Manual dynamic dispatch

Rest of lecture: 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 even though not everything is an object)

- Demonstrates how one language's *semantics* is an idiom in another language
- Understand dynamic dispatch better by coding it up
  - Roughly similar to how an interpreter/compiler would do it

Analogy: In Lecture 9, we encoded higher-order functions using objects and explicit environments

Our approach

Many ways to achieve our aim. Code in **`lec21.rkt`** 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)
```

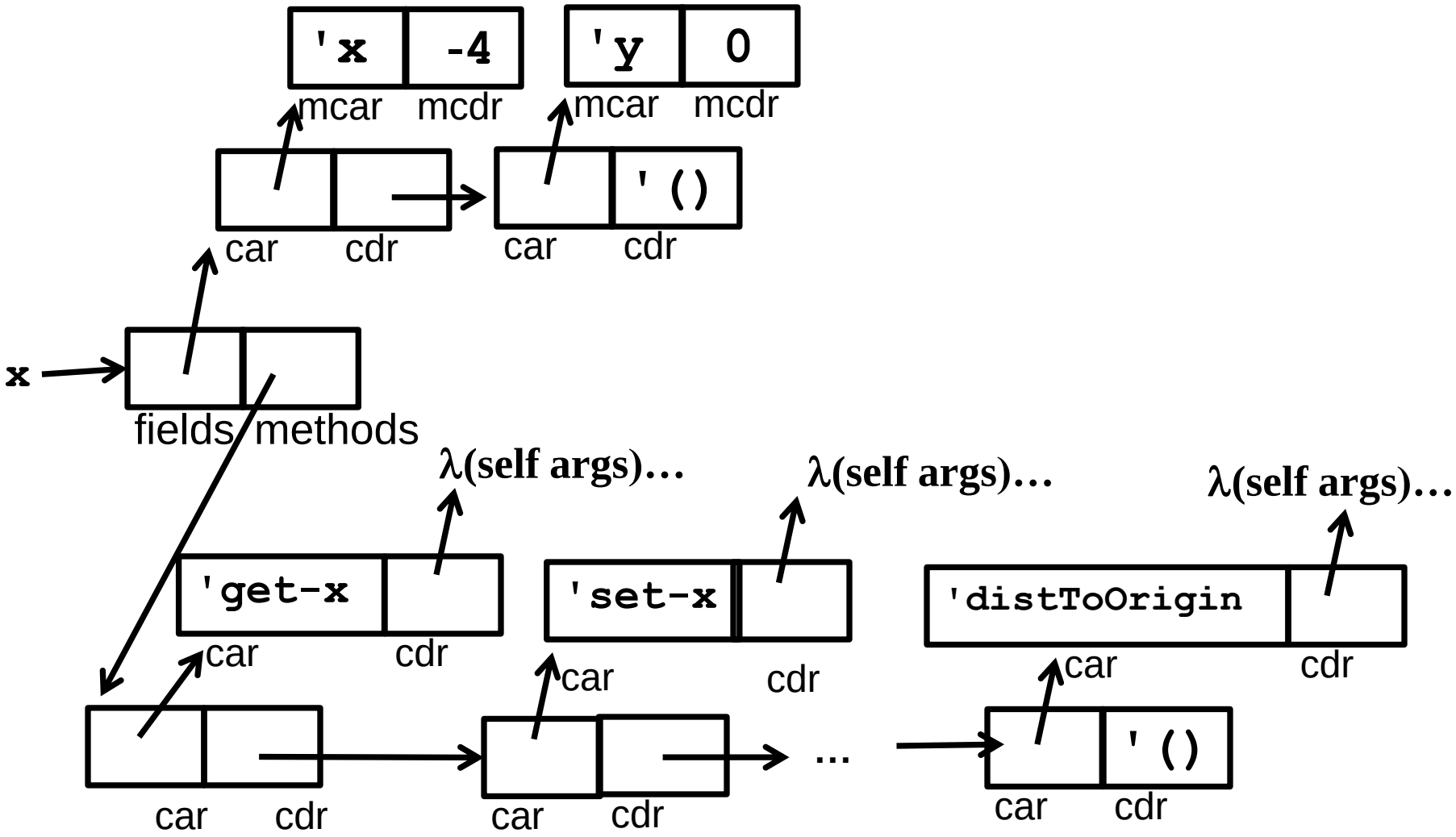
- Method-list element example:

```
(cons 'get-x (lambda (self args) …))
```

Notes:

- Association lists sufficient but not efficient
- This is not class-based: object has a list of methods, not a class that has a list of methods [could do it that way instead]
- The key trick is having lambdas take an extra **`self`** argument
  - All "regular" arguments put in a list **`args`** for simplicity

A point object bound to **x**



Key helper functions

Given the object representation on previous slide, define plain-old Racket functions to get field, set field, send message

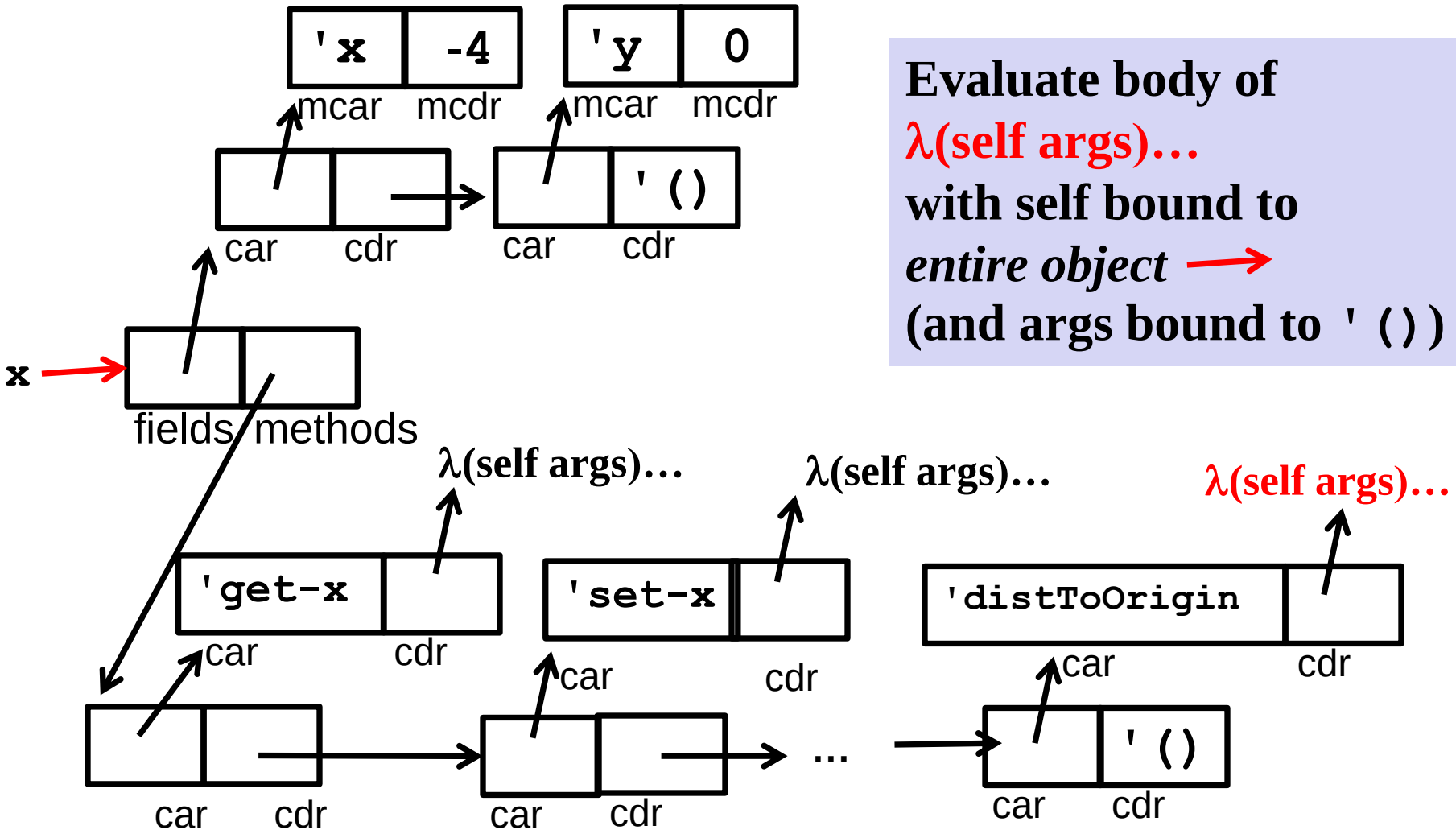
```
(define (assoc-m v xs)
   …) ; assoc for list of mutable pairs

(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 (see **`lec21.rkt`**)
  - Use functions **`get`**, **`set`**, and **`send`** on result and in "methods"
  - Call to self: **`(send self 'm …)`**
  - Real arguments in **`(car args)`**, **`(cadr args)`**, etc.

```
(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 to do this exercise in Racket, not ML
- ML's type system doesn't have subtyping for declaring a polar-point type and a point type *and* treating one as the other
  - Various workarounds possible (e.g., 1 type for all objects)
  - Without workarounds, no good type for those **`self`** arguments to functions
    - Type depends on "what class" is "using" the method (and whole purpose is to support dynamic dispatch)
- In fairness, languages with subtyping but not generics make it analogously awkward to write generic code
- Future lecture will discuss subtyping, contrast it with generics, and discuss how a language (e.g., Java) can support both