Functional Programming

Higher-Order Programming and the Environment Model

Joey W. Coleman, Stefan Hallerstede

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

28 April 2014

Admin

Review

Higher-Order Programming

Environment Model

Next Week

Admin

Review

Higher-Order Programming

Environment Model

Next Week

Admin items

- FP1 Assignment due: 23:59 Friday 2 May
- Will post the Multi-paradigm assignment next week.

Admin

Review

Higher-Order Programming

Environment Model

Next Week

Comprehension test

```
(define double
  (lambda (x)
    + x x))
```

- What does this procedure do?
- Is it syntactically correct?

Why the substitution model?

- Full model for computation
- In principle, you need nothing more than the whole program
- No side-effects (so, interaction is out of scope for this course)
- Expression optimisation
- Is `(+ x x)` the same as `(* 2 x)`? `(left-shift 2 x)`? Always?
- Basis for the Environment model
- . . . functional programming is more than just syntax!

Admin

Review

Higher-Order Programming

Environment Model

Next Week

Higher-Order Programming

- What does “procedures are data” mean?
- Essence of Higher-Order Programming:
  - Treat procedures like any other data type
  - Create them when needed
  - Treat functionality as something that can be parameterised
- Reflection libraries approach this
- Function pointers in C give part of this
  - Both this and reflection only work on what is already there
  - Java 8 lambdas are “better” (but the syntax is weird)
- Self-modifying code
- Critical: generalisation of algorithms

Digression: Abstraction

- Computer Science and Software Engineering
- One core mental tool: abstraction
- Arguably every gain in practical reasoning power comes from abstraction
- Higher-Order programming is a direct use of this
- Caveat: use the *correct* abstractions!

Example: (simple-)map

```
(define simple-map
  (lambda (proc lst)
    (if (null? lst)
        '()
        (cons (proc (car lst))
              (simple-map proc (cdr lst))))))
```

- We saw this last week (more or less)
  - Tail recursive?
- Irritating to write every time you need it
  - Aside: how often do you think about `for` loops?
  - Other than to be sure you're not off by 1?
- Abstract what you need done into a specific procedure
- Call the generic procedure with your procedure and the data

Example: filter

```
(define filter
  (lambda (pred? lst)
    (cond ((null? lst) '())
          ((pred? (car lst)) (cons (car lst)
                                   (filter pred? (cdr lst))))
          (else (filter pred? (cdr lst))))))
```

- Takes a predicate and a list, returns a list with only the elements that satisfy the predicate
- Consider a priority queue

Example: foldl

```
(define foldl
  (lambda (proc acc lst)
    (if (null? lst)
        acc
        (foldl proc
               (proc acc (car lst))
               (cdr lst)))))
```

- "folds" an operator into a list
- `(foldl + 0 '(1 2 3))` $\rightarrow 0+1+2+3$
  i.e. `(+ (+ (+ 0 1) 2) 3)`
- `(foldl cons '() '(1 2 3))` $\rightarrow$ `...`

Newton's Method

- Standard approximation for finding roots of a function, f
- Formula for derivative

$$Df(x) = \frac{f(x + dx) - f(x)}{dx}$$

  where dx is the infinitesimal, and $Df(x)$ is the derivative of $f(x)$
- Start with a guess, r
- Check the guess: if $f(r) = 0$, done
- Otherwise, improve the guess, making r_{n+1}:

$$r_{n+1} = r_n - \frac{f(r_n)}{Df(r_n)}$$

- Normally, r_n converges on a root

What do we need for Newton's Method?

- Some value for the infinitesimal
- A way of computing derivatives
- A way of checking the accuracy of the guess
- A way to improve the guess
- . . . and a way of defining the function of interest
- So, let's implement it.

Summary for Higher-Order Programming

- `map`, `filter`, `foldl`
- Newton's Method
  - Applicable to any numeric function
  - More in SICP, Section 1.3
- Related:
  - First-order/Higher-order logic
  - Functors
  - Domain Specific Languages
  - Callbacks/Listener pattern
  - Design Patterns (failure of abstraction mechanisms)

Admin

Review

Higher-Order Programming

Environment Model

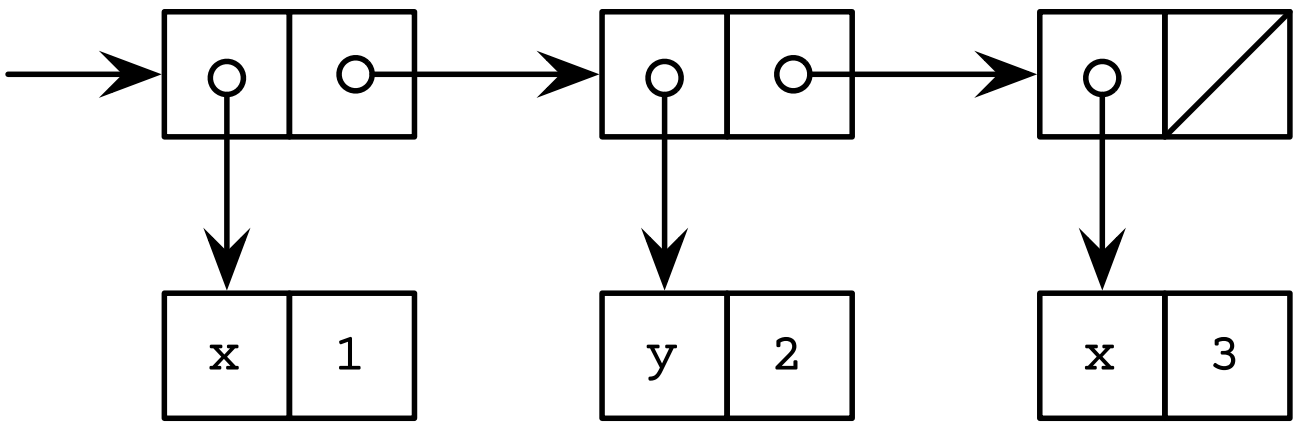
Next Week

The Environment Model

- SICP section 3.2 on the Environment Model
- Elaboration of the Substitution Model
- Provides a place to store definitions
- Allows the possibility of “commands”
  - i.e. things that are not expressions

Environments

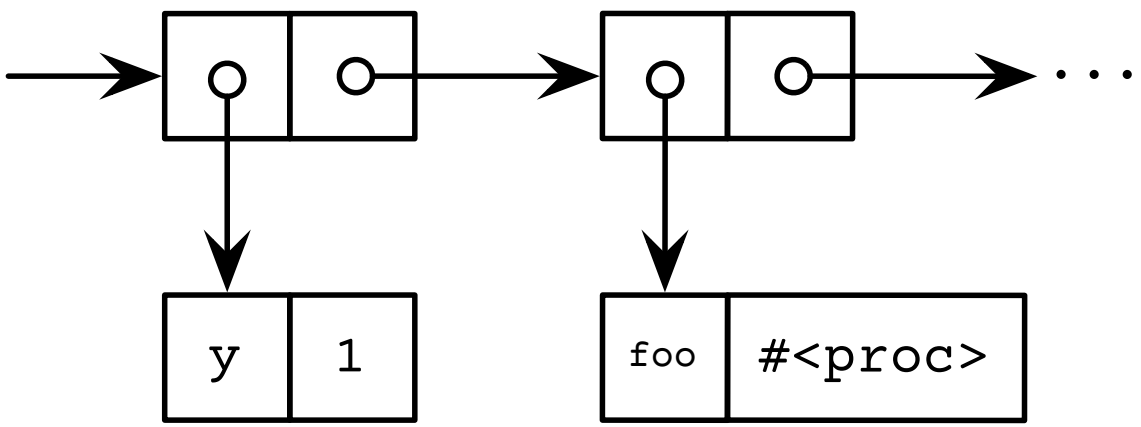
- An environment is a list of pairs
- The `car` is a symbol
- The `cdr` is the value
- Say we have an environment where `x`, `y`, `z` are defined to 1, 2, 3, respectively
- `'((x . 1) (y . 2) (z . 3))`



Environment from a Lambda

```
(define foo
  (lambda (y) y))
(foo 1)
```

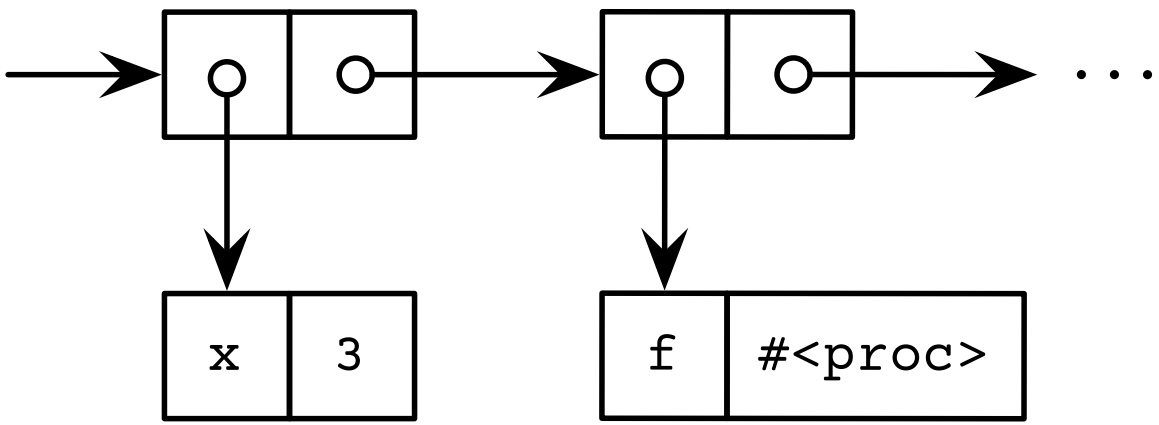
- The environment at `(foo 1)` is:
  `'((foo . #<proc>) ...)`
- Once inside the `lambda`, at `y`, the environment is:
  `'((y . 1) (foo . #<proc>) ...)`
- visually:



Environment from a Let

```
(let ((x 3)
      (f (lambda (y) y)))
  (f x))
```

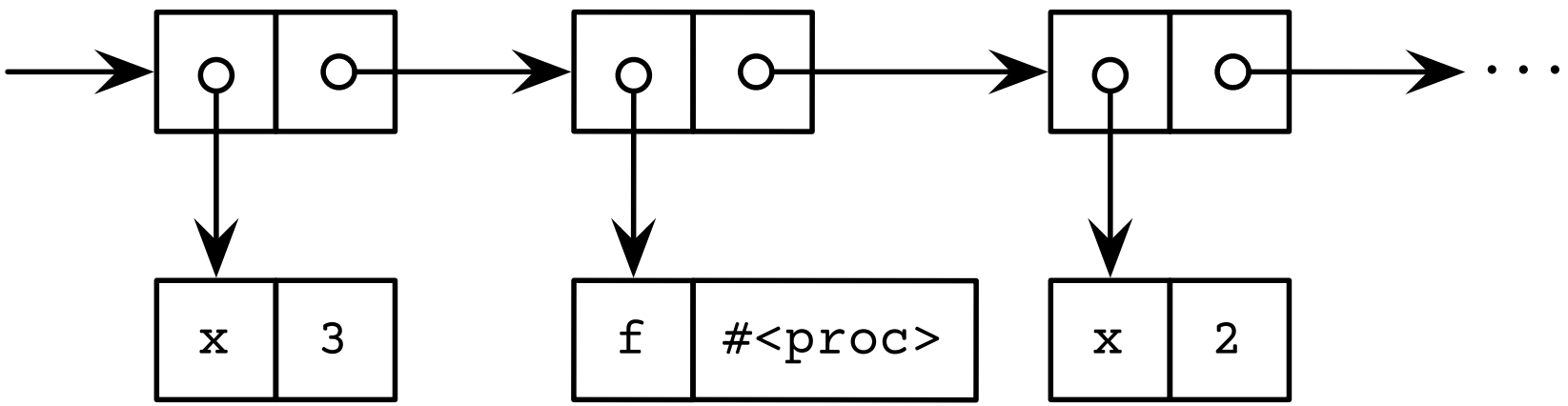
- The environment at `(f x)` is:
  `'((x . 3) (f . #<proc>) ...)`
- visually:



Shadowing

```
(let ((x 2))
  (+ x (let ((x 3)
             (foo (lambda (y) y)))
         (foo x))))
```

- The environment at `(foo x)` is:
  `'((x . 3) (foo . #<proc>) (x . 2) ...)`
- or, visually:



Define, Set!

- `(define <symbol> <expression>)`
  - *Adds* a new definition to the current environment
- `(set! <symbol> <expression>)`
  - *Changes* a definition in the current environment

Letrec

syntax

(`letrec` ((⟨variable$_1$⟩ ⟨expression$_1$⟩) ...)
  ⟨body⟩)

substitution

(`let` ((⟨variable$_1$⟩ `'*dummy*`) ...)
  (`set!` ⟨variable$_1$⟩ ⟨expression$_1$⟩)
  ...
  ⟨body⟩)

- Why the nested `set!` commands?

Closures

- Procedure + Environment at definition
- Defining a procedure “captures” the environment at its definition
- All definitions in the environment may be used.

```
(define gravitational-force
  (let ((G 6.674e-11))
    (lambda (m1 m2 r)
      (/ (* G m1 m2)
         (* r r)))))
```

- Note that the `G` is captured in the definition
- Is effectively a constant
- Iterative helpers
- `let` inside versus outside the `lambda`

Consequences

- What does the environment model allow, as a result?
- Imagine a “counter” generator
- Call `(make-counter 0)` and create a counter
- How do we use it?

Admin

Review

Higher-Order Programming

Environment Model

Next Week

Homework

- Read: SICP section 3.2 on the Environment Model
- Note: SICP uses an older-style “nested define” notation

Next Week

- Multi-paradigm development
  - General difficulties
  - Functional + Imperative
- Feedback on Assignment 1
- FP assignment 2 posted by Monday
- Multi-paradigm assignment post early next week.