CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

The University of Arizona

Version: 1 **February 24, 2020**

Today

- **Closures to create "private" variables**
- **High-order function curry**
- **Reasoning about functions**
- **Useful higher-order functions**

Closures represent escaping functions

Function value representation: $(\!|\lambda x.e, \rho|\!)$

(ρ binds the free variables of e)

A closure is a heap-allocated record containing

- a pointer to the code
- an environment storing free variables

$(\!|\bullet, \{n \mapsto 3, k \mapsto 27\}|\!)$

code for `x-to-the-n-minus-k`

Exercises, vulnerable variables?

- **What is the problem with this?**

```
-> (val seed 1)
-> (val rand (lambda ()
        (set seed (mod (+ (* seed 9) 5) 1024)))))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
1
-> (rand)
14
```

Exercises, Lambda as abstraction barrier

The University of Arizona
Computer Science

- **Instead use lambda to give a variable "private" access**

```
-> (val mk-rand (lambda (seed)
        (lambda () (set seed (
                       mod (+ (* seed 9) 5) 1024)))))
-> (val rand (mk-rand 1))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
error: set unbound variable seed
-> (rand)
160
```

Currying

- **Currying converts a binary function f(x,y) to**
  - **A function f' that takes x and returns …**
  - **a function f'' that takes y and returns f(x,y)**

- **Handy: now all functions just take one parameter!**

```
-> (val positive? (lambda (y) (< 0 y)))
-> (positive? 3)
#t
-> (val <-c (lambda (x) (lambda (y) (< x y))))
-> (val positive? (<-c 0)) ;; partial application
-> (positive? 0)
#f
```

What's the algebraic law for curry?

- **Keep in mind that you can apply a function**

```
... (curry f) ... = ... f ...
```

```
(((curry f) x) y) = (f x y)
```

No need to curry by hand!

```
;; curry : binary function -> value -> function
-> (val curry
     (lambda (f)
       (lambda (x)
         (lambda (y) (f x y)))))
-> (val positive? ((curry <) 0))
-> (positive? -3)
#f
-> (positive? 11)
#t
```

Exercises

- **What is the result of the following expressions?**

```
-> (map ((curry +) 4) '(1 2 3 4 5))
?
-> (exists? ((curry =) 4) '(1 2 3 4 5))
?
-> (filter ((curry >) 4) '(1 2 3 4 5))
?
```

```
;; How would we define curry3?
```

Reasoning about code

- **Reasoning principles for lists:**
  - recursive function that consumes list A has the same structure as a proof about A
  - ➔ Q1: How to prove two lists are equal?

- **Reasoning principle for functions**
  - Q2: Can you do case analysis on a function?
  - A2: No!
  - Q3: So what can you do to them?
  - Q4: Apply it!
    - Q5: How to prove two functions are equal?
    - A5: Prove that when applied to equal arguments, they produce equal results.

Higher-Order Functions

- **Goal: start with functions on elements, end up with functions on lists**
  - Generalizes to sets,
  - arrays,
  - search trees,
  - hash tables, …

- **Goal: Capture common patterns of computation or algorithms**
  - `exists?` (**example: is there a number?**)
  - `all?` (**example: is everything a number?**)
  - `filter` (**example: take only the numbers**)
  - `map` (**example: add 1 to every element**)
  - `foldr` (**general: can do all of the above and more**)

List search: `exists?`

- **Algorithm encapsulated: linear search**
- **Example: Is there an even element in the list**
- **Algebraic laws, all possible forms of list input**

```
    (exists? p? '()) == ???

    (exists? p? (cons a as)) == ???
```

Defining `exists?`

```
-> (define exists? (p? xs)
       (if (null? xs)
           #f
           (or (p? (car xs))
               (exists? p? (cdr xs)))))
-> (exists? even? '(1 3))
??
-> (exists? even? '(1 2 3))
??
-> (exists? ((curry =) 0) '(1 2 3))
??
-> (exists? ((curry =) 0) '(1 2 3 0))
??
```

List search: `all?`

- **Algorithm encapsulated: linear checking**
- **Example: Is every element in the list even?**
- **Algebraic laws, all possible forms of list input**

```
(all? p? '()) == ???

(all? p? (cons a as)) == ???
```

Defining `all?`

THE UNIVERSITY OF ARIZONA
Computer Science

```
-> (define all? (p? xs)
        (if (null? xs)
             #t
             (and (p? (car xs))
                  (all? p? (cdr xs)))))
-> (all? even? '(1 3))
??
-> (all? even? '(2))
??
-> (all? ((curry =) 0) '(0 1 2 3))
??
-> (all? ((curry =) 0) '(0 0))
??
```

List search: `filter?`

- **Algorithm encapsulated: linear filtering**
- **Example: Given a list of numbers, return only the even ones**
- **Algebraic laws, all possible forms of list input**

```
    (filter p? '()) == ???

    (filter p? (cons a as)) == ???
```

- **What are the restrictions on `p?` for `exists?`, `all?`, and `filter?`**

Defining `filter`


```
-> (define filter (p? xs)
      (if (null? xs)
         '()
         (if (p? (car xs))
            (cons (car xs) (filter p? (cdr xs)))
            (filter p? (cdr xs)))))
-> (filter (lambda (n) (> n 0)) '(1 2 -3 -4 5))
??
-> (filter (lambda (n) (<= n 0)) '(1 2 -3 -4 5))
??
-> (filter ((curry <) 0) '(1 2 -3 -4 5))
??
-> (filter ((curry >=) 0) '(1 2 -3 -4 5))
??
```

Composition Revisited: List Filtering

```
-> (val positive? ((curry <) 0))

-> (filter positive? '(1 2 -3 -4 5))
??
-> (filter (o not positive?) '(1 2 -3 -4 5))
??
```

List search: `map`

- "Lifting" functions to lists
- Algorithm encapsulated: transform every element
- Example: square every number of a list
- Algebraic laws, all possible forms of inputs

```
(map f '()) == ???

(map f (cons a as)) == ???
```

Defining map

THE UNIVERSITY OF ARIZONA
Computer Science

```
-> (define map (f xs)
       (if (null? xs)
           `()
           (cons (f (car xs))
                 (map f (cdr xs)))))
-> (map number? `(3 a b (5 6)))
??
-> (map ((curry *) 10) `(3 7 2))
??
-> (val square*
        ((curry map) (lambda (n) (* n n))))
-> (square* `(1 2 3))
??
```

The universal list function: fold

THE UNIVERSITY OF ARIZONA.
Computer Science

Algebraic laws for foldr

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
(foldr (plus zero '()))              = zero
(foldr (plus zero (cons y ys))) =
                          (plus y (foldr plus zero ys))
```

Note: Binary operator **+** associates to the right.

Note: `zero` might be identity of `plus`.

`foldr`: the universal list function

- **`foldr` takes two arguments**
  - `plus`: **how to combine elements with running results**
  - `zero`: **what to do with the empty list**

- **Example: foldr plus zero '(a b)**

```
    cons a (cons b `())
     |        |      |
     v        v      v
    plus a (plus b zero)
```

The universal list function: fold

Code for foldr

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
-> (define foldr (plus zero xs)
     (if (null? xs)
       zero
       (plus (car xs) (foldr plus zero (cdr xs)))))
-> (val sum  (lambda (xs) (foldr + 0 xs)))
-> (sum '(1 2 3 4))
10

-> (val prod (lambda (xs) (foldr * 1 xs)))
-> (prod '(1 2 3 4))
24
```

The universal list function: fold

Another view of operator folding

```
'(1 2 3 4)   =   (cons 1 (cons 2 (cons 3 (cons 4 '()))))
(foldr + 0 '(1 2 3 4))
             =   (+    1 (+    2 (+    3 (+    4 0  ))))
(foldr f z '(1 2 3 4))
             =   (f    1 (f    2 (f    3 (f    4 z  ))))
```

Exercise

Idea: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

```
-> (define combine (x a) (+ 1 a))
-> (foldr combine 0 '(2 3 4 1))
???
```

Studying for the midterm

- **Implement each of the following using foldr**
  - exists?
  - all?
  - filter
  - map

- **Feel free to post possible answers on piazza**