Higher-order List Functions in Racket

SOLUTIONS

CS251 Programming Languages
Spring 2018, Lyn Turbak
Department of Computer Science
Wellesley College

Higher-order List Functions

A function is **higher-order** if it takes another function as an input and/or returns another function as a result. E.g. app-3-5, make-linear-function, flip2 from the previous lecture

We will now study **higher-order list functions** that capture the recursive list processing patterns we have seen.

Recall the List Mapping Pattern

\[(\text{map} F \ (\text{list} \ v_1 \ v_2 \ ... \ v_n))\]

Express Mapping via Higher-order **my-map**

\[
\begin{align*}
\text{(define } (\text{my-map} \ f \ xs) \\
\text{ (if } (\text{null?} \ xs) \ \\
\text{ null} \\
\text{ (cons } (f \ (\text{first} \ xs)) \ \\
\text{ (my-map} \ f \ (\text{rest} \ xs)))))
\end{align*}
\]
my-map Examples

> (my-map (λ (x) (* 2 x)) '(7 2 4))
  '(14 4 8)
> (my-map first '((2 3) (4) (5 6 7)))
  '(2 4 5)
> (my-map (make-linear-function 4 7) '(0 1 2 3))
  '(7 11 15 19)
> (my-map app-3-5 (list sub2 + avg pow (flip2 pow)
  make-linear-function))
  '(-2 8 4 243 125 #<procedure:...t-class-funs.rkt:17:4>)

Higher-order Liss Funs

Currying

A curried binary function takes one argument at a time.

(define (curry2 binop)
  (λ (x) (λ (y) (binop x y))))
(define curried-mul (curry2 *)
  > ((curried-mul 5) 4)
    20
> (my-map (curried-mul 3) '(1 2 3))
  '(3 6 9)
> (my-map ((curry2 pow) 4) '(1 2 3))
  '(16 64)
> (my-map ((curry2 (flip2 pow)) 4) '(1 2 3))
  '(256 65536)
> (define LOL '((2 3) (4) (5 6 7)))
> (my-map ((curry2 cons) 8) LOL)
  '((2 3 8) (4 8) (8 5 6 7))
> (my-map (curry2 snoc) 8) LOL) ; fill in the blank
  '((2 3 8) (4 8) (5 6 7 8))

Higher-order Liss Funs

map-scale

Define (map-scale n nums), which returns a list that results from scaling each number in nums by n.

> (map-scale 3 '(7 2 4))
  '(21 6 12)
> (map-scale 6 (range 0 5))
  '(0 6 12 18 24)

(map-scale n nums)
(my-map (λ (num) (* n num))
  nums))

Higher-order Liss Funs

Mapping with binary functions

(define (my-map2 binop xs ys)
  (if (or (null? xs) (null? ys)) ; design decision:
    null
    (cons (binop (first xs) (first ys))
      (my-map2 binop (rest xs) (rest ys)))))

> (my-map2 pow '(2 3 5) '(6 4 2))
  '(64 81 25)
> (my-map2 cons '(2 3 5) '(6 4 2))
  '((2 . 6) (3 . 4) (5 . 2))
> (my-map2 + '(2 3 4 5) '(6 4 2))
  '((2 . 6) (3 . 4) (5 . 2))

Higher-order Liss Funs
**Built-in Racket map Function**
Maps over Any Number of Lists

> (map (λ (x) (* x 2)) (range 1 5))
' (2 4 6 8)

> (map pow '(2 3 5) '(6 4 2))
' (64 81 25)

> (map (λ (a b x) (+ (* a x) b))
' (2 3 5) '(6 4 2) '(0 1 2))
' (6 7 12)

> (map pow '(2 3 4 5) '(6 4 2))
ERROR: map: all lists must have same size;
arguments were: #<procedure:pow> '(2 3 4 5) '(6 4 2)

**Express Filtering via Higher-order my-filter**

```scheme
(define (my-filter pred xs)
  (if (null? xs)
      null
      (if (pred (first xs))
        (cons (first xs)
          (my-filter pred (rest xs))))
      (my-filter pred (rest xs))))
```

Built-in Racket filter function acts just like my-filter

**Recall the List Filtering Pattern**

\[
\text{filterP (list } v_1 v_2 \ldots v_n)\]

Higher-order Lists

**filter Examples**

> (filter (λ (x) (> x 0)) '(7 -2 -4 8 5))
' (7 8 5)

> (filter (λ (n) (= 0 (remainder n 2)))
' (7 -2 -4 8 5))
' (-2 -4 8)

> (filter (λ (xs) (>= (len xs) 2))
' ((2 3) (4) (5 6 7))
' ((2 3) (5 6 7))

> (filter number? '(17 #t 3.141 "a" (1 2) 3/4 5+6i))
' (17 3.141 3/4 5+6i)

> (filter (lambda (binop) (>= (app-3-5 binop) (app-3-5 (flip2 binop))))
  (list sub2 + * avg pow (flip2 pow)))
; The printed rep would show 4 #<procedure>s,
; but the returned list would be equivalent to
; (list + * avg pow)
Recall the Recursive List Accumulation Pattern

\[
\text{recf (list } v_1 \ v_2 \ldots \ v_n)\\
\]

\[
\begin{align*}
\text{v}_1 & \longrightarrow \text{v}_2 & \longrightarrow & \cdots & \longrightarrow & \text{v}_n & \longrightarrow & \bullet \\
\downarrow & \quad & \downarrow & \quad & \downarrow & \quad & \downarrow & \\
\text{combine} & \quad & \text{combine} & \quad & \cdots & \quad & \text{combine} & \quad & \text{nullval}
\end{align*}
\]

\[
\text{(define } \text{rec-accum } \text{xs) }\\
\begin{align*}
\text{(if (null? } \text{xs) } \\
\text{nullval} \\
\quad \text{(combine } \text{(first } \text{xs) } \\
\quad \quad \text{(rec-accum (rest } \text{xs))})
\end{align*}
\]

Express Recursive List Accumulation via Higher-order \text{my-foldr}

\[
\begin{align*}
\text{v}_1 & \longrightarrow \text{v}_2 & \longrightarrow & \cdots & \longrightarrow & \text{v}_n & \longrightarrow & \bullet \\
\downarrow & \quad & \downarrow & \quad & \downarrow & \quad & \downarrow & \quad & \downarrow & \\
\text{combine} & \quad & \text{combine} & \quad & \cdots & \quad & \text{combine} & \quad & \text{nullval}
\end{align*}
\]

\[
\text{(define } \text{my-foldr combine nullval } \text{vals) }\\
\begin{align*}
\text{(if (null? } \text{vals) } \\
\text{nullval} \\
\quad \text{(combine } \text{(first } \text{vals) } \\
\quad \quad \text{(my-foldr } \text{combine nullval } \text{(rest } \text{vals))})
\end{align*}
\]

\text{my-foldr Examples}

\[
\text{> (my-foldr } + 0 \ ' (7 2 4)) \\
\text{13 ; (+ 7 (+ 2 (+ 4 0))))}
\]

\[
\text{> (my-foldr } * 1 \ ' (7 2 4)) \\
\text{56 ; (* 7 (* 2 (* 4 1))))}
\]

\[
\text{> (my-foldr } - 0 \ ' (7 2 4)) \\
\text{9 ; (- 7 (- 2 (- 4 0))))}
\]

\[
\text{> (my-foldr min } \text{+inf.0 } ' (7 2 4)) \\
\text{2 ; (min 7 (min 2 (min 4 +inf.0))))}
\]

\[
\text{> (my-foldr max } \text{-inf.0 } ' (7 2 4)) \\
\text{7 ; (max 7 (max 2 (max 4 -inf.0))))}
\]

\[
\text{> (my-foldr cons } '(8) \ ' (7 2 4)) \\
\text{'(7 2 4 8) ; (cons 7 (cons 2 (cons 4 '(8))))}
\]

\[
\text{> (my-foldr append null } '\text{(2 3) (4 (5 6 7))} \\
\text{'(2 3 4 5 6 7)} \\
\text{; (append } '(2 3) \text{ (append } '(4 (append '}(5 6 7) '\text{()))))}
\]

\text{More my-foldr Examples}

\[
\text{> (my-foldr } \lambda (a \ b) \ (\text{and } a \ b)) \ #t \ (\text{list } \#t \ \#t \ \#t)) \\
\text{\#t ; (and } \#t \ (\text{and } \#t \ (\text{and } \#t \ \#t)))
\]

\[
\text{> (my-foldr } \lambda (a \ b) \ (\text{and } a \ b)) \ #t \ (\text{list } \#t \ \#f \ \#t)) \\
\text{\#f ; (and } \#t \ (\text{and } \#f \ (\text{and } \#t \ \#t)))
\]

\[
\text{> (my-foldr } \lambda (a \ b) \ (\text{or } a \ b)) \ #f \ (\text{list } \#t \ \#f \ \#t)) \\
\text{\#t ; (or } \#t \ (\text{or } \#f \ (\text{or } \#t \ \#t)))
\]

\[
\text{> (my-foldr } \lambda (a \ b) \ (\text{or } a \ b)) \ #f \ (\text{list } \#f \ \#f \ \#f)) \\
\text{\#f ; (or } \#f \ (\text{or } \#f \ (\text{or } \#f \ \#f)))
\]

\[
\text{;; This doesn’t work. Why not? } \\
\text{> (my-foldr and } \#t \ (\text{list } \#t \ \#t)) \\
\text{; and is a syntactic sugar construct, not a function, } \\
\text{; so get the following error: } \\
\text{and: bad syntax in: and}
\]
Your turn: sumProdList

Define sumProdList (from scope lecture) in terms of foldr. Is let necessary here like it was in scoping lecture?

\[
\text{(define (sumProdList '()} -> '(0 . 1))}
\]

\[
\text{(define (sumProdList nums)}
\text{ (foldr (λ (fst subres) ; combiner}
\text{ (cons (+ fst (car subres)))
\text{ (* fst (cdr subres)))
\text{ '()} ; nullval)
\text{ nums))}
\text{; (1) Good idea to begin combiner (λ (fst subres) ... )
\text{; (2) Use “pretty printing” indentation to align
\text{ 3 args to foldr and 2 args to cons)}
\]

Mapping & Filtering in terms of my-foldr

\[
\text{(define (my-map f xs)}
\text{ (my-foldr (λ (fst subres) ; combiner
\text{ (cons (f fst) subres))
\text{ '(); nullval
\text{ xs))}
\]

\[
\text{(define (my-filter pred xs)}
\text{ (my-foldr (λ (fst subres) ; combiner
\text{ (if (pred fst)
\text{ (cons fst subres)
\text{ subres))
\text{ '(); nullval
\text{ xs))}
\]

Built-in Racket foldr Function

Folds over Any Number of Lists

\[
> (foldr + '() '(7 2 4))
13
> (foldr (lambda (a b sum) (+ (* a b) sum))
0
 ' (2 3 4)
 ' (5 6 7))
56
> (foldr (lambda (a b sum) (+ (* a b) sum))
0
 ' (1 2 3 4)
 ' (5 6 7))
\]

ERROR: foldr: given list does not have the same size as the first list: ' (5 6 7)

Problematic for foldr

\[
\text{(keepBiggerThanNext nums) returns a new list that keeps all nums that are bigger than the following num. It never keeps the last num.}
\]

\[
> (keepBiggerThanNext '() '(7 5 3 9 8))
' (7 5 9)
> (keepBiggerThanNext '() '(7 5 3 9 8))
' (7 5 9)
> (keepBiggerThanNext '() '(7 5 3 9 8))
' (4 7 5 9)
\]

keepBiggerThanNext cannot be defined by fleshing out the following template. Why not?

\[
\text{(define (keepBiggerThanNext nums)}
\text{ (foldr <combiner> <nullvalue> nums))}
\]
keepBiggerThanNext with foldr

keepBiggerThanNext needs (1) next number and (2) list result from below. With foldr, we can provide both #1 and #2, and then return #2 at end

(define (keepBiggerThanNext nums)
  (second
    (foldr (λ (thisNum nextNum&subResult) ; combiner
              (let ([nextNum (first nextNum&subResult)] ; arg #1
                    [subResult (second nextNum&subResult)]) ; arg #2
                   (list thisNum ; becomes nextNum for elt to left
                       (if (> thisNum nextNum) ; special case for singleton list
                           (cons thisNum subResult) ; keep
                           subResult)))))) ; don’t keep
    (list +inf.0 '()); +inf.0 guarantees last num
    ; in nums won’t be kept
    nums)))

Higher-order Lisc Funs  21

foldr-ternop: more info for combiner

In cases like keepBiggerThanNext, it helps for the combiner to also take rest of list as an extra arg

(foldr-ternop ternop nullval (list v1 v2 ... vn))

arg #1 arg #2 arg #3 nullval

(define (foldr-ternop ternop nullval vals)
  (if (null? vals)
      nullval
      (ternop (first vals) ; arg #1
               (rest vals) ; extra arg # 2 to ternop
               ; arg #3
               (foldr-ternop ternop nullval (rest vals)))))

Higher-order Lisc Funs  23