CS450

Structure of Higher Level Languages

Lecture 13: Reduction

Tiago Cogumbreiro

Today we will learn...

1. Revise general recursion patterns
2. Implement general recursion patterns
3. Refactor code to reduce code repetition
4. Refactor code to improve performance

Functional patterns: Reduction

Concatenate two lists

Concatenate two lists

Implement function (`append l1 l2`) that appends two lists together.

Spec

```
(check-equal?
  (append (list 1 2) (list 3 4))
  (list 1 2 3 4))
```

Concatenate two lists

Implement function `(append l1 l2)` that appends two lists together.

Spec

```
(check-equal?
  (append (list 1 2) (list 3 4))
  (list 1 2 3 4))
```

Solution

```
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else (cons (first l1) (append (rest l1) l2))]))
```

> Is it tail recursive?

Concatenate two lists

Implement function (`append l1 l2`) that appends two lists together.

Spec

```
(check-equal?
  (append (list 1 2) (list 3 4))
  (list 1 2 3 4))
```

Solution

```
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else (cons (first l1) (append (rest l1) l2))]))
```

> Is it tail recursive? **No!**

Generalizing reduction

A pattern arises

```
; Example 1:
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
; Example 2:
(define (filter to-keep? l)
  (cond [(empty? l) empty]
    [else
      (cond [(to-keep? (first l))
             (cons (first l)
               (filter to-keep? (rest l)))]
            [else (filter to-keep? (rest l))])]))
; Example 3:
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

A pattern arises

```
; Example 1:
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
; Example 2:
(define (filter to-keep? l)
  (cond [(empty? l) empty]
    [else
      (cond [(to-keep? (first l))
               (cons (first l)
                 (filter to-keep? (rest l)))]
            [else (filter to-keep? (rest l))])]))
; Example 3:
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

General recursion pattern for lists

```
(define (rec l)
  (cond
    [(empty? l) base-case]
    [else (step (first l) (rec (rest l)))]))
```

For instance,

```
(cons (f (first l)) (map f (rest l)))
```

maps to

```
(step (first l) (rec (rest l)))
```

Implementing this recursion pattern

Recursive pattern for lists

```
(define (rec l)
  (cond
    [(empty? l) base-case]
    [else (step (first l)
                (rec (rest l)))]))
```

Fold right reduction

```
(define (foldr step base-case l)
  (cond
    [(empty? l) base-case]
    [else (step (first l)
                (foldr step base-case (rest l)))]))
```

Implementing map with foldr

```
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
```

Implementing map with foldr

```
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
```

Solution

```
(define (map f l)
  (define (on-elem elem new-list)
    (cons (f elem) new-list))
  (foldr on-elem empty l))
```

Implementing append with foldr

```
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
         (cons (first l1)
               (append (rest l1) l2))]))
```

Implementing append with foldr

```
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

Solution

```
(define (append l1 l2)
  (foldr cons l2 l1))
```

Implementing filter with foldr

```
(define (filter to-keep? l)
  (cond
    [(empty? l) empty]
    [else
      (cond [(to-keep? (first l))
             (cons (first l)
               (filter to-keep? (rest l)))]
            [else (filter to-keep? (rest l))])]))
```

Solution

```
(define (filter to-keep? l)
  (define (on-elem elem new-list)
    (cond [(to-keep? elem) (cons elem new-list)]
          [else new-list]))
  (foldr on-elem empty l))
```

Contrasting the effect of using foldr

```
; Example 1:
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
; Example 2:
(define (filter to-keep? l)
  (cond
    [(empty? l) empty]
    [else
      (cond [(to-keep? (first l))
             (cons (first l)
               (filter to-keep? (rest l)))]
            [else (filter to-keep? (rest l))])]))
; Example 3:
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

Contrasting the effect of using foldr

```
; Example 1:
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
; Example 2:
(define (filter to-keep? l)
  (cond
    [(empty? l) empty]
    [else
      (cond [(to-keep? (first l))
             (cons (first l)
              (filter to-keep? (rest l)))]
            [else (filter to-keep? (rest l))])]))
; Example 3:
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

```
; Example 1:
(define (map f l)
  (define (on-elem elem new-list)
    (cons (f elem) new-list))
  (foldr on-elem empty l))
; Example 2:
(define (filter to-keep? l)
  (define (on-elem elem new-list)
    (cond [(to-keep? elem) (cons elem new-list)]
          [else new-list]))
  (foldr on-elem empty l))
; Example 3:
(define (append l r)
  (foldr cons r l))
```

What about the fold left reduction?

Reversing a list

Implement function (`reverse l`) that reverses a list.

Spec

```
(check-equal? (list 4 3 2 1) (reverse (list 1 2 3 4)))
```

Reversing a list

Implement function `(reverse l)` that reverses a list.

Spec

```
(check-equal? (list 4 3 2 1) (reverse (list 1 2 3 4)))
```

Solution

```
(define (reverse l)
  (define (rev l accum)
    (cond [(empty? l) accum]
          [else (rev (rest l) (cons (first l) accum))]))
  (rev l empty))
```

Another pattern arises

```
; Example 1
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l)))
          (rest l))]))
  (concat-nums-aux ">" l))
; Example 2
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum)
                     (rest l))]))
  (rev empty l))
```

A generalized recursion pattern for lists

```
(define (rec base-case l)
  (cond
    [(empty? l) base-case]
    [else
      (rec (step (first l) base-case)
           (rest l))]))
```

For instance,

```
(cons (first l) accum)
```

maps to

```
(step (first l) accum)
```

Implementing this recursion pattern

Recursive pattern for lists

```
(define (rec accum l)
  (cond
    [(empty? l) accum]
    [else
      (rec (step (first l) accum)
           (rest l))]))
```

Fold left reduction

```
(define (foldl step base-case l)
  (cond
    [(empty? l) base-case]
    [else (foldl step
                 (step (first l) base-case)
                 (rest l))]))
```

Implementing concat-nums with foldl

Before

```
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
           (string-append accum (f (first l)))
           (rest l))]))
  (concat-nums-aux ">" l))
```

Implementing concat-nums with foldl

Before

```racket
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l)))
          (rest l))]))
  (concat-nums-aux ">" l))
```

After

```racket
(define (concat-nums l)
  (define (f n a)
    (string-append a " "
      (number->string n)))
  (foldl f ">" l))
```

Implementing reverse with foldl

Original

```
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum)
                     (rest l))]))
  (rev empty l))
```

Implementing reverse with foldl

Original

```
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum)
                     (rest l))]))
  (rev empty l))
```

Solution

```
(define (reverse l)
  (foldl cons empty l))
```

Contrasting the effect of using foldl

Before

```
; Example 1
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l)))
          (rest l))]))
  (concat-nums-aux ">" l))
; Example 2
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum)
                     (rest l))]))
  (rev empty l))
```

Contrasting the effect of using foldl

Before

```
; Example 1
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l)))
          (rest l))]))
  (concat-nums-aux ">" l))
; Example 2
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum)
                     (rest l))]))
  (rev empty l))
```

After

```
; Example 1
(define (concat-nums l)
  (define (f n a)
    (string-append a " "
      (number->string n)))
    (foldl f ">" l))
; Example 2
(define (reverse l)
  (foldl cons empty l))
```

What about tail-recursive optimization?

What about tail-recursive optimization?

- We note that `foldl` is tail-recursive already
- However, our original implementation of `foldr` is not tail recursive

Can't we implement the tail-recursive optimization pattern?

Unoptimized

```
(define (rec l)
  (cond
    [(empty? l) base-case]
    [else (step (first l) (rec (rest l)))]))
```

Optimized

```
(define (rec l)
  (define (rec-aux accum l)
    (cond
      [(empty? l) (accum base-case)]
      [else
        (rec-aux
          (lambda (x)
            (accum (step (first l) x)))
          (rest l))]))
  (rec-aux (lambda (x) x) l))
```

Optimized foldr

Generalized pattern

```
(define (rec l)
  (define (rec-aux accum l)
    (cond
      [(empty? l) (accum base-case)]
      [else
        (rec-aux
          (lambda (x)
            (accum (step (first l) x)))
          (rest l))]))
  (rec-aux (lambda (x) x) l)
```

Implementation

```
(define (foldr step base-case l)
  (define (foldr-iter accum l)
    (cond
      [(empty? l) (accum base-case)]
      [else
        (foldr-iter
          (lambda (x)
            (accum (step (first l) x)))
          (rest l))]))
  (foldr-iter (lambda (x) x) l))
```

Benchmark evaluation

- Unoptimized `foldr`
- Tail-recursive `foldr`

```
Processing a list of size: 1000000
-------------------------------------------------------
Throughoutput (unopt): 7310 elems/ms
Mean (unopt): 136.8±7.56ms
-------------------------------------------------------
Throughoutput (tailrec): 12349 elems/ms
Mean (tailrec): 80.98±1.49ms
Speed-up (tailrec): 1.7
```

A speed improvement of 1.7

What if we use foldl + reverse?

What if we use foldl + reverse?

- Instead of creating nested functions,
- We reverse the list and apply foldl

```
(define (foldr step base-case l)
  (foldl step base-case (reverse l)))
```

What if we use foldl + reverse?

- Instead of creating nested functions,
- We reverse the list and apply foldl

```
(define (foldr step base-case l)
  (foldl step base-case (reverse l)))
```

Simpler implementation!

But is it faster?

Rev+fold runs the slower (0.7)

```
Processing a list of size: 1000000
-------------------------------------------------------
Throughoutput (unopt): 7310 elems/ms
Mean (unopt): 136.8±7.56ms
-------------------------------------------------------
Throughoutput (tailrec): 12349 elems/ms
Mean (tailrec): 80.98±1.49ms
Speed-up (tailrec): 1.7
-------------------------------------------------------
Throughoutput (rev+foldl): 4846 elems/ms
Mean (rev+foldl): 206.34±3.33ms
Speed-up (rev+foldl): 0.7
```

Conclusion

We learned to generalize two reduction patterns (foldl and foldr)

- **Pro:** generalizing code can lead to a central point to optimize code
- **Pro:** generalizing code can reduce our code base
  (less code means less code to maintain)
- **Con:** one level of indirection increases the cognitive code
  (more cognitive load, code harder to understand)

Easier to understand (self-contained)

```
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))
```

Harder to understand (what is foldr?)

```
(define (map f l)
  (define (on-elem elem new-list)
    (cons (f elem) new-list))
  (foldr on-elem empty l))
```