CS450

Structure of Higher Level Languages

Lecture 12: Reduction, thunks

Tiago Cogumbreiro

Today we will learn...

1. Optimizing code to be tail-recursive
2. List reduction (`append`, `foldl`)
3. Learn about delayed evaluation (thunks)

> Acknowledgment: Today's lecture is partially inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington: (Video 1) (Video 2)

Making map tail-recursive

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

Tail-recursive `map` run

```
(map f (list 1 2 3)) =
; First, build the pipeline accumulator
(define (accum0 x) x) (map-iter accum0 (list 1 2 3)) =
(define (accum1 x) (accum0 (cons (f 1) x))) (map-iter accum1 (list 2 3)) =
(define (accum2 x) (accum1 (cons (f 2) x))) (map-iter accum2 (list 3)) =
(define (accum3 x) (accum2 (cons (f 3) x))) (map-iter accum3 (list)) =
; Second, run the pipeline accumulator
(accum3 (list)) =
(accum2 (list (f 3))) =
(accum1 (list (f 2) (f 3))) =
(accum0 (list (f 1) (f 2) (f 3))) =
(list (f 1) (f 2) (f 3)))
```

Tail-recursive optimization pattern

To summarize, when a value has base case and an inductive case, we identified the following pattern for a tail-recursive optimization:

Unoptimized

```
(define (rec v)
  (cond
    [(base-case? v) (base v)]
    [else (step v (rec (dec v)))]))
```

Optimized

```
(define (rec v)
  (define (rec-aux accum v)
    (cond
      [(base-case? v) (accum (base v))]
      [else
        (rec-aux
          (lambda (x) (accum (step v x)))
          (dec v))]))
  (rec-aux (lambda (x) x) v))
```

Tail-recursive `map`, using the generalized tail-recursion optimization pattern

```
(define (map f l)
  (define (map-iter accum l)
    (cond [(empty? l) (accum l)]
          [else (map-iter (lambda (x) (accum (cons (f (first l)) x))) (rest l))]))
  (map-iter (lambda (x) x) l))
```

Scanning

Remove zeros from a list

Spec

```
(require rackunit)
(check-equal? (list 1 3 4) (remove-0 (list 0 1 3 0 4)))
(check-equal? (list 1 2 3) (remove-0 (list 1 2 3)))
```

Remove zeros from a list

Spec

```
(require rackunit)
(check-equal? (list 1 3 4) (remove-0 (list 0 1 3 0 4)))
(check-equal? (list 1 2 3) (remove-0 (list 1 2 3)))
```

Solution

```
(define (remove-0 l)
  (cond
    [(empty? l) l]
    [(not (equal? (first l) 0)) (cons (first l) (remove-0 (rest l)))]
    [else (remove-0 (rest l))]))
```

Can we generalize this functional pattern?

Original

```
(define (remove-0 l)
  (cond
    [(empty? l) l]
    [(not (equal? (first l) 0))
     (cons (first l) (remove-0 (rest l)))]
    [else (remove-0 (rest l))]))
```

Generalized

```
(define (filter to-keep? l)
  (cond
    [(empty? l) l]
    [(to-keep? (first l))
     (cons (first l)
           (filter1 to-keep? (rest l)))]
    [else (filter to-keep? (rest l))]))

;; Usage example
(define (remove-0 l)
  (filter
    (lambda (x) (not (equal? x 0))) l))
```

Is this function tail-recursive?

Can we generalize this functional pattern?

Original

```
(define (remove-0 l)
  (cond
    [(empty? l) l]
    [(not (equal? (first l) 0))
     (cons (first l) (remove-0 (rest l)))]
    [else (remove-0 (rest l))]))
```

Generalized

```
(define (filter to-keep? l)
  (cond
    [(empty? l) l]
    [(to-keep? (first l))
     (cons (first l)
           (filter1 to-keep? (rest l)))]
    [else (filter to-keep? (rest l))]))

;; Usage example
(define (remove-0 l)
  (filter
    (lambda (x) (not (equal? x 0))) l))
```

> Is this function tail-recursive? **No.** Function `cons` is a tail-call; `filter` is not.

Tail-recursive filter

Revisiting the tail call optimization

> Function `filter` has very similar shape than function `map`, so we can apply the same optimization pattern.

```
(define (filter to-keep? l)
  (define (filter-aux accum l)
    (cond
      [(empty? l) (accum l)] ; same as before
      [else
        (define hd (first l)) ; cache the head of the list
        (define tl (rest l))  ; cache the tail of the list
        (cond
          [(to-keep? hd) (filter-aux (lambda (x) (accum (cons hd x))) tl)]
          [else (filter-aux accum tl)])]))
  (filter-aux (lambda (x) x) l))
```