Purely Functional Data Structures

CIS 352 — Spring 2020
Logistics

- e2/e3 released over weekend
  - Both .25% bonus (not all exercises will be)
- a2 released today: due Monday after next
  - a3 will likely be released before a2 due
- Do e2/e3 before attempting a2

- Coding exam 0 — Week after next
  - More logistics soon
  - In-class programming exam (roughly half of class)
  - Email me soon if you need anything special for this
Warmup (observations on folds)

Assignment 1 defines a portion of PageRank as a sum…

$$\sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$
Warmup (observations on folds)

Consider a mathematical sum over a set, $S$

$$\sum_{e \in S} f(e)$$

The summation is readily translated to using foldl:

```scheme
(define s (set 1 2 3))
(define (f x) (+ 1 x))
(foldl (λ (e acc) (+ (f e) acc)) 0 (set->list s))
```
Exercise

Write the following product using foldl and multiplication:

\[ \prod_{e \in \{1,2,3\}} 2e \]
Write the following product using foldl and multiplication:

$$
\prod_{e \in \{1,2,3\}} 2e
$$

(foldl (λ (e acc) (* 2 e acc)) 1
     (set->list (set 1 2 3)))
Data Structures

• A **data structure** is a representation of data

• **Constructors** build data

• **Destructors** (or matching) observes data
  • E.g., (empty?, cons?, car, cdr)
    • These four functions alone sufficient to define all functions that observe lists

• Defines various **operations** on the data

• **Abstract data type (ADT)** leaves form opaque, just operations
  • E.g., push, pop
  • Same ADT can have multiple concrete implementations
Purely Functional Data Structures

• A data structure is **purely functional** when all operations produce *new* data, rather than *changing* input data

• Otherwise the data structure is **imperative** or **stateful**

• Most of Racket’s data structures are purely functional:
  • Cons cells, Lists, Immutable hashes, etc…

• Imperative variants have some potential advantages
  • Can be faster, allow more flexible access

• Reasoning about imperative data structures requires reasoning about the temporal patterns in its shape
  • This can be tricky!
A **queue** is a first-in, first-out data structure:

- **Enqueue** insert an element into queue
- **First** retrieves first element of the queue
- **Rest** retrieves the rest of the queue
We can implement a queue as a list

```
(define (empty-queue) '())

(define (queue-add queue elt)
  (append queue `(,elt)))

(define (queue-first queue) (first queue))

(define (queue-rest queue) (rest queue))
```
Unfortunately this is **slow**, as `append` is $O(n)$. Thus `(queue-add)` is $O(n)$

```scheme
(define (empty-queue) '())

(define (queue-add queue elt)  
  (append queue `(,elt)))

(define (queue-first queue) (first queue))

(define (queue-rest queue) (rest queue))
```
Let’s build some code to test our queue

;; build a queue of size i
(define (build-random-queue i)
  (define (loop num-left acc)
    (match num-left
      [0 acc]
      [_ (loop (- num-left 1) (queue-add acc (random 0 200)))]))
  (loop i (empty-queue)))

;; get nth element from the head of the queue
(define (get-nth queue n)
  (match n
    [0 (queue-first queue)]
    [_ (get-nth (queue-rest queue) (- n 1))])))
And now build a queue of size 20,000, then retrieve its last element

;; build a queue of size n, then destruct it
(define (n-firsts-and-rests n)
  (get-nth (build-random-queue n) (- n 1)))

(time
  (n-firsts-and-rests 20000))

;; cpu time: 4885 real time: 4825 gc time: 2824

4.8 seconds!
Observation: to build queue $O(n)$ calls to $(\text{queue–add} \ldots)$, we do $O(n^2)$ work

`; build a queue of size n, then destruct it
(define (n-firsts-and-rests n)
  (get-nth (build-random-queue n) (- n 1)))

(time
  (n-firsts-and-rests 20000))

`; cpu time: 4885 real time: 4825 gc time: 2824

4.8 seconds!
Okasaki’s Lazy Queues

• Our queue is purely functional, but it is slow
  • (make-queue ...) is $O(n)$, which is unacceptable
  • Imperative implementations perform $O(1)$ insert
• Chris Oksaki presents lazy queues
  • Insert, first, and rest all have $O(1)$ amortized time.
    • $O(n)$ calls to insert (first, and reset) perform $O(n)$ work
    • But an individual call may take up to $O(n)$ time
  • Achieves this by using two lists rather than one
    • One you cons on to (the head) to insert
    • One you pull leaves from (call cdr on) to dequeue
Queue is a pair of a front (in order) and back (in reverse order)

Empty queue is just pair of empty lists

\[(\text{define \ (empty-lazy-queue) \ (cons \ '() \ '()))}\]

\[(\text{define \ (lqueue-add \ queue \ elt) \ (match \ queue}]
\[\[\text{[(cons \ '() \ '()) \ (cons \ '(() ,elt) \ '()))]}\]
\[\text{[(cons \ front \ end)}\]
\[\text{\ (cons \ front \ (cons \ elt \ end)))]\]

To add to queue: build new queue that conses new element to reversed end, O(1)
Tricky! Need to be careful when front is empty. In that case, first is end. We always want to be able to access first via car.

```
(define (empty-lazy-queue) (cons '() '()))

(define (lqueue-add queue elt)
  (match queue
    [(cons '() '()) (cons `(,elt) '())]
    [(cons front end)
      (cons front (cons elt end))]))
```
Front is kept in order, and using cons ensures we get O(1) time for first

\[
\text{(define (lqueue-first queue)
 (match queue
   [(cons front end) (car front)])})
\]
Rest must consider three cases:
• No more list left (heap underflow)
• Front empty, but back nonempty
  • Reverse back, make it front
• Front nonempty, pair its rest with back

(define (lqueue-rest queue)
 (match queue
     [(cons '() '()) (error 'underflow)]
     [(cons '() back)
      (queue-rest (cons '() (reverse back)))]
     [(cons front back)
      (cons (cdr front) back)]))
Consider a queue that looks like...

• (cons `(0 ... 10000) `(0 ... 10000))

• **Rest** will take $O(1)$ time for the first 10,001 calls

• Then, 10,002nd call will reverse `(0 ... 10000)` and make it `(10000 ... 0)`, taking time proportional to $10k$

• Then, 10,003rd call and onward take $O(1)$ time: as they are back in first case

```scheme
(define (lqueue-rest queue)
  (match queue
    [((cons '() '())) (error 'underflow)]
    [((cons '() back) (queue-rest (cons '() (reverse back))))]
    [(cons front back) (cons (cdr front) back)])
)```
Amortized Runtime

• **Amortization**: pay fee “up front” so next calls cheaper

• We say a function has **amortized** $O(1)$ complexity if:
  • $O(n)$ calls takes $O(n)$ time
  • $O(f(n))$ amortized if $O(n)$ calls take $O(f(n) \times n)$ time

• Several methods for reasoning about amortized data
  • Won’t discuss specifics in this class
  • Basis for several popular functional data structures

• Imperative languages can often achieve $O(1)$ complexity easier, as they can use pointers
  • But good functional data structures are usually fine