Pure Functional Data Structures

Mitch Stevens

Pure Function Data Structures

- Tonight: Queues
- Some other night: Finger Tree, Graphs, Catenatable

Queue

```haskell
data Queue a = Queue { f :: [a], r :: [a] }
```

- `f` is the **front** of a queue
- `r` is the **rest** of the queue
- We are assuming that length of the list is cached
- *Invariant: length f >=length r*

Queue

```haskell
data Queue a = Queue { f :: [a], r :: [a] }
```

- `f` is the **front** of a queue
- `r` is the **rest** of the queue
- We are assuming that length of the list is cached
- *Invariant: length f >=length r*

- If `null f` then the queue is empty
- This invariant is enforced with a pseudo-constructor

Queue

```haskell
rotate :: [a] -> [a] -> [a]
rotate f r = f <> reverse r

queue :: [a] -> [a] -> Queue a
queue f r =
  if length f < length r -- O(1)
    then Queue (rotate f r) []
    else Queue f r
```

Queue Operations

```
snoc :: Queue a -> a -> Queue a
snoc (Queue f r) x = queue f (x:r)

head :: Queue a -> a
head (Queue f r) = Prelude.head f

tail :: Queue a -> Queue a
tail (Queue f r) = queue (Prelude.tail f) r
```

- Invariant corollary: `null f` => queue is empty

Queue operations

- These operations (`snoc`, `head`, `tail`) are amortised O(1)
- `queue` reshuffles the elements whenever `length f - length r < 0`
- let `s := length f - length r`

Queue operations

- These operations (`snoc`, `head`, `tail`) are amortised O(1)
- `queue` reshuffles the elements whenever `length f - length r < 0`
- let `s := length f - length r`

- `tail` decrements `s` by removing an element from `f`
- `snoc` decrements `s` by adding an element to `r`

Queue operations

- These operations (`snoc`, `head`, `tail`) are amortised O(1)
- `queue` reshuffles the elements whenever `length f - length r < 0`
- let `s := length f - length r`
- `tail` decrements `s` by removing an element from `f`
- `snoc` decrements `s` by adding an element to `r`
- After `queue` reshuffles the elements, there will be n operations before the next reshuffling.

Lazy evaluation and amortisation

- In strict languages, time complexity $\sim$ num of operation
- In a lazy context this is not true

Lazy evaluation and amortisation

- In strict languages, time complexity $\sim$ num of operation
- In a lazy context this is not true

```haskell
foo :: Num a => [a] -> [a] -> a
foo l1 l2 = sum $ take 10 (l1 <> reverse l2)
```

- If `len l1 <= 10` then $\sim$10 operations are performed

Lazy evaluation and amortisation

- In strict languages, time complexity $\sim$ num of operation
- In a lazy context this is not true

```haskell
foo :: Num a => [a] -> [a] -> a
foo l1 l2 = sum $ take 10 (l1 <> reverse l2)
```

- If `len l1 <= 10` then ~10 operations are performed
- Otherwise we have to evaluate `reverse l2`, which is an O(n) operation
- Often unclear how many operations are performing here

Realtime Queue

```haskell
data Queue a = Queue { f :: [a], r :: ![a], s :: [a] }
```

- **Invariant:** `length s = length f - length r`

Realtime Queue

```haskell
data Queue a = Queue { f :: [a], r :: ![a], s :: [a] }
```

- **Invariant:** `length s = length f - length r`
- Why is `r` strict?
- Why are we storing the invariant as a list, instead of an `Int`?

Realtime Queue

```haskell
data Queue a = Queue { f :: [a], r :: ![a], s :: [a] }
```

- **Invariant:** `length s = length f - length r`
- Why is `r` strict?
- Why are we storing the invariant as a list, instead of an `Int`?
- `s` is short for "schedualer"

Realtime Queue

- We again use a constructor to enforce the invariant

```haskell
rotate :: [a] -> [a] -> [a] -> [a]
rotate f r a = f <> reverse r <> a

queue :: [a] -> [a] -> [a] -> Queue a
queue f r s = case s of
  (x:s') -> Queue f r s'
  []     -> Queue f' [] f'
    where f' = rotate f r []
```

Rotating a Queue

```
rotate f r a = case (f, r) of
  (Nil,  y:_)  -> y:a
  (x:f', y:r') -> x : rotate f' r' (y : a)
```

- Why go to all this trouble?

Rotating a Queue

```
rotate f r a = case (f, r) of
  (Nil,  y:_)  -> y:a
  (x:f', y:r') -> x : rotate f' r' (y : a)
```

- Why go to all this trouble?
- Example: `queue [1, 2] [5, 4, 3] [] = Queue f r s`
- `f = rotate [1, 2] [5, 4, 3] []`
- `r = []`
- `s = f`

Rotating a Queue

Need to ensure that pattern matching `f` and `r` is a single operation

- `f` is itself a suspension of `rotate`
- `r` is strict