FUNC Lecture 7
Purely Functional Queues

(lightly adapted for TFPIE'17)

Colin Runciman

Purely Functional . . .

Persistence by Non-Destruction

- A *persistent* implementation of a data structure is *non-destructive*. Operations such as insertion or deletion do not alter the original. They derive a new version from it.
- Parts of the structure affected by an operation are *copied*; but unchanged parts are *shared*.
- So multiple threads of computation can work independently on the same initial data structure.
- Or a failing path of computation can be abandoned without any need to reverse changes it has made.
- In imperative languages based on destructive assignment, programming a persistent data structure is a *delicate task*.
- In a *purely functional* language we have *persistence for free*! But the challenge is to make it efficient.

. . . Queues.

Breadth-First Search: a Motivating Application

```
breadthFirst :: (a -> [a]) -> a -> [a]
breadthFirst b r  =  bf [r]
  where
  bf []        =  []
  bf (x:xs)    =  x : bf (xs ++ b x)
```

eg. `breadthFirst (\n -> [(n*2)+1,(n+1)*2]) 0`
$\rightsquigarrow$ `[0,1,2,3,4,5,6,7,...`

- `breadthFirst` takes as arguments the specification of a tree by a *branching function* `b` and a *root* `r`. Its result is the list of items in the tree in *breadth-first* order.
- Auxiliary `bf` uses its list argument *as a queue*. Adding items to the queue by *concatenation* is expensive. For a large tree, `(++)` is applied many times and to long first arguments `xs`.
- The *cons-nil list* provides $O(1)$ access to the *front*, but only $O(n)$ access to the *rear*. It makes a good stack, but a poor queue.

A Type-Class Specification for Queues

```
class QueueSpec q where
  empty   :: q a
  snoc    :: q a -> a -> q a
  head    :: q a -> a
  tail    :: q a -> q a
  queue   :: [a] -> q a
  queue   =  foldl snoc empty
  items   :: q a -> [a]
  isEmpty :: q a -> Bool
  isEmpty =  null . items
```

- For any datatype constructor `q` used to implement a queue, we shall provide an instance `QueueSpec q`.
- The name `snoc` is `cons` in reverse — a traditional joke.
- The `queue` function translates whole lists of items into queues. It is not essential, but nice to have. Note the simple default.
- Conversely, the `items` function translates the other way. So `isEmpty` also has a simple default.

One List?

Lists as a Reference Implementation

```
data ListQ a  =  LQ [a]

instance QueueSpec ListQ where
  empty              =  LQ []
  snoc   (LQ xs) x   =  LQ (xs ++ [x])
  head   (LQ xs)     =  Prelude.head xs
  tail   (LQ xs)     =  LQ (Prelude.tail xs)
  queue              =  LQ
  items (LQ xs)      =  xs
```

- The `QueueSpec` class declaration only specifies methods by their types.
- A simple instance for list types serves to specify the *expected behaviour* of the `QueueSpec` methods.
- It also provides a benchmark against which more efficient alternatives can be measured.
- The *glaring inefficiency* is an $O(n)$ `snoc`.
- A default `isEmpty` is fine, but we improve on a default `queue`!

Two Lists.

Batched Queues (1)

```
data BatchedQ a  =  BQ [a] [a]

-- one possibility for items 1-6 queued in order
BQ [1,2,3] [6,5,4]
```

- A seminal idea, prompting numerous variations, is to split queued items into *two* lists: the *front items* `f` and the *rear items in reverse* `r`.
- The motivation is to make the end of the queue immediately accessible: for `snoc`, we can use `(:)` on the rear list.
- But the split into front and rear sections raises two issues:
  1. What rule determines how the queue is divided into front and rear sections?
  2. When and how should items transfer from one section to the other?

Batched Queues (2)

```
bq :: [a] -> [a] -> BatchedQ a
bq [] r     =  BQ (reverse r) []
bq f  r     =  BQ f r

instance QueueSpec BatchedQ where
  empty                 =  BQ [] []
  snoc   (BQ f r) x     =  bq f (x:r)
  head   (BQ (x:_) _)   =  x
  tail   (BQ (_:f) r)   =  bq f r
  queue  xs             =  BQ xs []
  items  (BQ f r)       =  f ++ reverse r
```

- A *smart constructor* `bq` keeps an *invariant rule* for a batched queue `BQ f r` that `null f ==> null r`.
- The motivation is to ensure $O(1)$ access to the head.
- When a `snoc` or `tail` operation threatens to break this rule, `bq` reverses the whole *batch* of rear items to form a new front.
- Instead of an $O(n)$ operation for *every* `snoc`, there are only *occasional* $O(n)$ batch reversals.

Amortized Complexity versus Worst-Case Complexity

- Still, in the *worst-case*, `tail` is $O(n)$. So have we really made any progress?
- *Amortized complexity* is concerned with the *overall cost* of a *sequence of operations* rather than the division of costs among them.
- If a sequence of n operations $op_1 \ldots op_n$ has worst-case complexity $O(n)$, then the amortized complexity of each op_i is $O(1)$ even though the worst-case op_i may be more costly.
- We can often obtain *simpler and faster* implementations by aiming for low *amortized* complexity than for low *worst-case* complexity of individual operations.
- For the `BatchedQ` implementation, both `snoc` and `tail` have amortised complexity $O(1)$.

The Nemesis of Batched Queues: Multi-Threading

- More precisely, the `BatchedQ` implementation achieves $O(1)$ amortised complexity for *single-threaded* queue computations using the basic operations `empty`, `snoc`, `head` and `tail`.
- Consider `q :: BatchedQ` of the form `BQ [i] r`, with a one-element front list. If the next operation applied to `q` is `tail`, it involves the $O(n)$ reversal of `r`.
- Suppose q is used in a *multi-threaded* way — *ie.* in an expression referring to q more than once, where each q is needed.
- In *each thread*, if the next operation on q is `tail`, an $O(n)$ cost is incurred.
- For *multi-threaded* computations we *cannot* claim $O(1)$ *amortised complexity* for the `BatchedQ` operations.

Three Lists!

Incremental Rotating Queues (1)

```
data RotatingQ a  =  RQ [a] [a] [a]

instance QueueSpec RotatingQ where
  empty                  =  RQ [] [] []
  snoc  (RQ f r s) x     =  rq f (x:r) s
  head  (RQ (x:_) _ _)   =  x
  tail  (RQ (_:f) r s)   =  rq f r s
  queue xs               =  RQ xs [] xs
  items (RQ f r _)       =  f ++ reverse r
```

- Our goal is to perform reversals *incrementally*. We aim to split the task over several operations, each making only a small constant contribution.
- We introduce *another* list, *s*. It will always be some *shared suffix of f*. Specifically, our invariant for `RQ f r s` is: `length f >= length r && s == drop (length r) f`.
- The suffix *s* is used by *smart constructor* `rq` when the difference `length f - length r` decreases by one.

Incremental Rotating Queues (2)

```
rq :: [a] -> [a] -> [a] -> RotatingQ a
rq f r (x:s)  =  RQ f  r  s
rq f r []     =  RQ f' [] f'
  where   f'  =  rotate f r []

rotate :: [a] -> [a] -> [a] -> [a]
rotate []    [y]   a  =  y : a
rotate (x:f) (y:r) a  =  x : rotate f r (y:a)
```

- If the suffix is non-empty, `rq` simply discards its head to restore the invariant.
- If the suffix is empty, `rq` starts an *incremental* reversal. We know `length r == length f + 1`.
- On this condition `rotate f r a` gives `f ++ reverse r ++ a`. So if `a == []` it gives `f ++ reverse r` as required.
- *Crucially*, `rotate` is *lazy*. It takes only a single step to produce each successive element.

Acknowledgements and Further Reading

- The first published work on front-and-reversed-rear representations of queues:

  Robert Hood and Robert Melville, *Real-time queue operations in pure* LISP, Information Processing Letters, 13(2), pp50–54, 1981.

- For a fuller explanation of amortization, and two different methods for reasoning about amortized complexity, with queues among the illustrative examples, see:

  Chris Okasaki, *Fundamentals of Amortization*, Chapter 5 in his book *Purely Functional Data Structures*, Cambridge University Press, 1998.