CS 624: Notes 16

Ryan Culpepper

November 9, 2022

1 Persistent Data Structures

So far we have discussed algorithms on *mutable* data structures. In particular, both the *contents* and the *meaning* of an instance of the data structure changes over time.

In an *immutable* data structure, both the content and meaning of an instance are fixed. Generally, to perform an "update", you must create a new instance of the data structure.

In a *persistent* data structure, the meaning of an instance is fixed, but the (internal) contents may change.

Immutable and persistent data structures are especially important to functional languages like Lisp (and Scheme and Clojure and Racket), ML, and Haskell.

2 Singly-linked Lists

```
A List[X] is either
- Empty
- Cons(x, rest)
  where x : X, rest : List[X]
```

For example, the list (List[Integer]) containing 1, 2, and 3 is

```
Cons(1, Cons(2, Cons(3, Empty)))
```

In examples, I'll write lists more compactly using square brackets, like `[1,2,3]`.

3 Mutable vs Immutable Stacks

Here is a *signature* (or *interface*) for the mutable version of the Stack abstract data type.

```
type Stack[X]
push : X, Stack[X] -> Void
top  : Stack[X] -> X
pop  : Stack[X] -> Void
```

Implementations:

growable array, box holding singly-linked list

3.1 Signature for Immutable Stacks

```
type Stack[X]
push : X, Stack[X] -> Stack[X]
top  : Stack[X] -> X
pop  : Stack[X] -> Stack[X]
```

$\forall\ x\ \forall\ stk:$ `top(push(`x`,`stk`))` $=\ x$
$\forall\ stk:$ `pop(push(`x`,`stk`))` $=\ stk$

3.2 Implementation for Immutable Stacks

```
type Stack[X] = List[X]

push(x, stk) = Cons(x, stk)

top( Empty ) = ERROR
top( Cons(x,rest) ) = x

pop( Empty ) = ERROR
pop( Cons(x, rest) ) = rest
```

3.3 Asymptotic Performance of Stacks

```
| Operation | Mutable | Immutable |
|-----------+---------+-----------|
| push      | O(1)  * | O(1)      |
| top       | O(1)    | O(1)      |
| pop       | O(1)    | O(1)      |
```

4 FIFO Queues

Signature for mutable FIFO (first-in, first-out) queues:

```
type Queue[X]
enqueue : X, Queue[X] -> Void
dequeue : Queue[X] -> X
```

Implementations:

ring buffer, box to singly-linked list, doubly-linked list

Signature for immutable FIFO queues:

```
type Queue[X]
enqueue : X, Queue[X] -> Queue[X]
dequeue : Queue[X] -> (X, Queue[X])
```

4.1 Implementation for FIFO Queues, v1

```
type Queue[X] = List[X]  -- in "dequeue order"

enqueue(x, q) = append(q, Cons(x, Empty))
where append(Empty, ys) = ys
      append(Cons(x, xs), ys) = Cons(x, append(xs, ys))

dequeue( Cons(x,xs) ) = (x, xs)
```

I've left out the error cases.

4.2 Asymptotic Performance of Queues (v1)

```
| Operation | Mutable | Immutable |
|-----------+---------+-----------|
| enqueue   | O(1)  * | O(n)  !!! |
| dequeue   | O(1)    | O(1)      |
```

n is the size of the queue

4.3 Implementation of Queues (v2)

Strategy: we'll store the queue as *two* stacks (lists):

- one for quickly enqueuing (pushing) onto, and
- one for quickly dequeuing (popping) from.

```
type Queue[X] = Q(List[X], List[X])
```

For example, if we repeatedly dequeue from `Q([6,5,4], [1,2,3])` until it is empty, we get `1,2,3,4,5,6`.

We can enqueue by just pushing onto the "enqueue" list:

```
enqueue( e, Q(es,ds) ) = Q( Cons(e,es), ds )
```

We can dequeue by just popping from the "dequeue" list:

```
dequeue( Q(es, Cons(d,ds)) ) = (d, Q(es,ds))
```

But what if there is nothing in the "dequeue" list?

We need to grab the enqueued elements and move them over to the dequeue list. But we need to reverse the list so that elements get dequeued in FIFO order.

```
dequeue( Q(es, Empty) ) = dequeue( Q(Empty, reverse(es)) )
```

For example:

```
  dequeue( Q([3,2,1], []) )
= dequeue( Q([], [1,2,3]) )
= (1, Q([], [2,3]))
```

So `dequeue` could take $O(n)$ time, where n is the number of elements in the queue!

Have we accomplished nothing?!

Let's apply amortized analysis.

The actual cost of `enqueue` is \$3:

- pattern matching (`Q`, \$1 total)
- allocate a new `Cons` object (\$1)
- allocate a new `Q` object (\$1)

We'll change an amortized cost of \$5, though, and we'll save the extra \$2 as part of the enqueue list.

The actual cost for `dequeue` (in the fast case) is \$4:

- pattern matching (`Q` and `Cons`, \$2 total)
- allocate a new `Q` object (\$1)
- allocate a pair for the result (\$1)

In the slow case for `dequeue`, we also have to reverse the list, but we've saved \$2n to pay for it.

So (modulo exactly how we count the costs for pattern matching, reversing, the temporary `Q` object, etc), the amortized cost for dequeue is $4 — constant.

4.4 Asymptotic Performance of Queues (v2)

```
| Operation | Mutable | Immutable |
|-----------+---------+-----------|
| enqueue   | O(1)  * | O(1)      |
| dequeue   | O(1)    | O(1)  am  |
```

Better!

Sadly, this amortized analysis doesn't predict the time cost of the following program:

```
q = Q([3,2,1],[])  -- immutable queue!
for i = 1 to 10
  dequeue(q)       -- ignore result
```

Why? Because the amortized analysis assumes that we only use each intermediate queue object once, but this program repeatedly dequeues from the same immutable queue, in its slow-dequeue state, so we actually perform the reversal 10 times!

4.5 Immutable vs Persistent

One fix is to allow "internal" mutation. That is, the queue data structure pretends to be immutable (the *meaning* of an instance never changes), but it actually uses mutation to move the reversed enqueue list to the dequeue slot.

Another fix is to use *laziness*, a feature where we can *delay* the evaluation of a computation and then *force* is later, where the result of forcing the computation is guaranteed to be remembered, and so the computation is actually only evaluated once. Haskell and R are examples of programming languages that use lazy evaluation by default; other languages offer it as an opt-in feature. (Note: laziness is implemented by "internal" mutation, so it's a specialization of the previous solution.)

5 References

Purely Functional Data Structures by Chris Okasaki