Efficiency, Persistence, and Side-Effects

Efficiency & Side-Effects

Features of Side-Effects:

- Can avoid copying with side-effects, more efficient.
- Program analysis is more difficult.
- Introduces bugs.
- Separate copies must be kept of “old” versions of data.
- Compare Queue in Miranda to Queue in Scheme.

Scheme Queue – Side-Effects

Specification:

queue-create `(queue-create)` ⇒ new empty queue.

queue-empty? `(queue-empty? <queue>)` ⇒ boolean.

queue-dequeue! `(queue-dequeue! <queue>)` ⇒ head of `<queue>` (and removes this element the queue).

queue-enqueue! `(queue-enqueue! <queue> <element>)` ⇒ no return result (adds `<element>` to `<queue>`).

Scheme Queue Implementation

Implementation: head and tail pointers.

```
(define (queue-create) (cons (list) (list)))
(define (queue-empty? q) (null? (car q)))

(define (queue-dequeue! q)
   (let ((head (caar q)))
        (set-car! q (cdar q))
        (if (queue-empty? q)
            (set-cdr! q (list))
        )
        head
   )
)
```

Scheme Queue (cont)

```
(define (queue-enqueue! q data)
   (if (queue-empty? q)
       (begin
           (set-car! q (list data))
           (set-cdr! q (car q))
       )
       (begin
           (set-cdr! (cdr q) (list data))
           (set-cdr! q (cddr q))
       )
   )
   'ok
)
```


Miranda Queue Spec

```
abstype queue *
    with emptyqueue :: queue *
         isempty :: queue * -> bool
         head :: queue * -> *
         enqueue :: queue * -> * -> queue *
         dequeue :: queue * -> queue *
```


Miranda Queue 1

```
queue * == [*]

emptyqueue = []

isempty [] = True
isempty (x:xs) = False

head (x:xs) = x

enqueue q x = q ++ [x]

dequeue (x:xs) = xs
```


Scheme Q vs. Miranda Q 1

What are the runtimes of

Creation?

Empty?

Dequeue?

Enqueue?

Seems pretty good.

Drawbacks

Drawbacks of implementation of Scheme Queue:

- There is only one “live” copy of each queue.
- To share the queue, it must be copied.
- Accidental sharing of queue might cause bugs.

Drawbacks of implementation of Miranda Queue 1:

- *Enqueue* takes $O(n)$ time.

Notes

Notes:

- You can implement Miranda Queue 1 in Scheme.
- You cannot implement Scheme Queue in Miranda.

Goals:

- How can we implement efficient Queues in Miranda?
- How can we implement efficient persistent Queues in Scheme?

Amortized Analysis

(an introduction)

To find “average” (*amortized*) runtime.

- Charge a certain amount for each operation.
- For cheap operations, we make a profit.
- For expensive operations, we have a loss.
- The trick is to make sure we never go negative.
- Success implies amortized runtime is amount we charged.

Miranda Queue 2 – Amortized

Implementation: With *front* list and *rear* list.

- first element of *front* is head of queue.
- first element of *rear* is last element in queue – it’s backwards.
- “proper” non-empty queue never has empty *front*.

```
queue * == ([*],[*])                    || (front,rear)
emptyqueue = ([],[])
isempty ([],[]) = True
isempty q = False
head ([x:front],rear) = x
enqueue (front,rear) x = make_proper_queue (front,x:rear)
dequeue (x:front,rear) = make_proper_queue (front,rear)

make_proper_queue ([],x:rear) = (reverse (x:rear),[])
make_proper_queue q = q
```

Efficiency

Worst-case runtimes for queue containing n elements?

emptyqueue

isempty

queue-make-proper

head

dequeue

enqueue

Better Analysis

Let n be number of items in queue.

- Charge \$2 for each `queue_make_proper` call.
- If `queue_make_proper q` returns `q`, cost is \$1.
- If `queue_make_proper q` calls `reverse`, cost is \$$n$.

If we start with an empty queue, after any set of operations, will we always have profit?

The answer is yes (if we never use “old” queues)!

Example 1

```
q1 = foldl enqueue emptyqueue [1..6]

;; what does q1 look like?  How much $ do we have?

q2 = dequeue q1

;; what does q2 look like?  How much $ do we have?

q3 = dequeue dequeue q2

;; what does q3 look like?  How much $ do we have?
```


Using “Old” Queues

What can happen if we use “old” queues?

```
q1 = foldl enqueue emptyqueue [1..6]

;; what does q1 look like?  How much $ do we have?

q2 = dequeue q1

;; what does q2 look like?  How much $ did this operation cost?

dequeue q1
dequeue q1
dequeue q1

;; how much $ do we have now?
```

Miranda Q 2 vs Scheme Q

Recall the Scheme Queue using side-effects.

After dequeue, original queue is no longer accessible.

Thus, if usage patterns of queue do not use "old" copies, then Scheme Queue and Miranda Queue 2 are comparable: Their average runtime is about the same.

If usage patterns do use "old" copies, then Scheme Queue breaks unless explicit copies are made.

Miranda Queue 2 will not break, but we might pay for expensive operation twice.

(Note: Can implement Miranda Queue 2 in Scheme)

Persistence

Two types of data structures

ephemeral: Only access to a single version of data structure, old copies not accessed.
persistent: Access to past copies of data structure allowed.

Can we build an efficient *persistent* queue?

Memoization

Consider:

```
<some-code>
   where
      x = <some-expression>
```

Because Miranda is lazy, `x` will only be evaluated if needed.
When `x` is first evaluated, its value will be *memoized*.

Subsequent references to `x` will not re-evaluate `<some-expression>`.
Instead, they will return `x`'s memoized value.

Lazyness & Memoization

What is the runtimes of the following?

1. `reverse [1..n] !  0`
2. `([1,2,3] ++ reverse [1..n]) !  0`
3. `([1,2,3] ++ reverse [1..n]) !  3`

Here `reverse` is expensive.

We can use this idea to improve our queue implementation.

Miranda Queue 3

Instead of copying `reverse rear` to `front` when `front` is empty, copy it when `length rear > length front`.

Length is expensive to compute so store lengths in queue tuple.

```
queue * == (num,[*],num,[*])           || (front-len,front,rear-len,rear)

emptyqueue = (0,[],0,[])
isempty (0,[],0,[]) = True
isempty q = False
head (fl,(x:front),rl,rear) = x

enqueue (fl,front,rl,rear) x
       = make_proper_queue (fl,front,rl+1,x:rear)

dequeue (fl,x:front,rl,rear)
       = make_proper_queue (fl-1,front,rl,rear)

make_proper_queue (fl,front,rl,rear)
       = (fl+rl, front ++ reverse rear,0,[]), if rl > fl
       = (fl,front,rl,rear), otherwise
```


Amortized Analysis

What part of `make_proper_queue` is expensive?

It is `reverse`.

For queue length n, must call `dequeue` $n/2$ times before `reverse` happens because it is lazy!

Charge \$2 for `make_proper_queue` call.
Pay \$1 for no-op call.
Pay \$$k$ for call to reverse of k item list.

We must have $n/2$ no-op calls before \$$n/2$ `reverse`.

A persistent queue with amortized $O(1)$ cost per operation!!