Observational Equivalence

Last time, we saw ADTs and algebraic specifications, and we saw that when implementing an ADT we want to ensure that it satisfies its specification. We looked last at the queue ADT, along with a straightforward implementation in terms of a simple list.

That implementation of queues is fine and dandy, but it's pretty inefficient. Think about it: every time you enqueue, you need to walk over the whole list and attach the new element at the end—using `append`. If we enqueue elements a lot faster than we dequeue, then the list keeps growing, and it takes longer and longer to enqueue elements. Roughly, if we do not dequeue and perform n enqueue operations, the time taken to do those enqueue operations (where a time unit is the time needed to, say, visit one cell in the list) is on the order of n^2. That can get big pretty fast if the queue is long.

We can do better though, if we're willing to be a bit clever. Let's consider a queue implemented not as a single list, but as two lists, one on which to enqueue elements, and one from which to dequeue elements. To enqueue an element, you cons it at the head of the enqueue list. To dequeue an element, you remove it from the head of the dequeue list. Voilà. Both operations are fast. Of course, you may run out of elements on the dequeue list. When you do, you just take your enqueue list, reverse it, and present it as your new dequeue list. Think about it, draw pictures. It seems to work. Let's convince ourselves by implementing and proving that the implementation satisfies the specification.

The representation of a queue, now, is as a pair of lists of integers. I will use a list to holds those two lists. Let me use a `*` to distinguish these operations from the ones from last time, just so that we can compare them later.

```
(defun queuep* (q) (and (int-true-listp (first x))
                        (int-true-listp (second x))))

(defun empty* () (list nil nil))

(defun enqueue* (a q) (list (cons a (first q))
                            (second q)))

(defun isEmpty* (q) (and (endp (first q))
                         (endp (second q))))
```

```
(defun front* (q)
  (if (endp (second q))
      (car (rev (first q)))
      (car (second q))))

(defun dequeue* (q)
  (if (endp (second q))
      (list nil (cdr (rev (first q))))
      (list (first q) (cdr (second q)))))
```

Queue equality, in this implementation, is not trivial. It is *not* just equality of structures. For example, `((1 2) ())` and `(() (2 1))` should be two representations of the same queue—if we look at the front of either queue, we get `2`. Similarly, `((1 2) (4 3))` and `((1) (4 3 2))` also represent the same queue: if we repeatedly look at the front and dequeue, we get the same elements `4`, `3`, `2`, and `1`, in that order. A moment's thought give us the following definitions of queue equality for this representation:

```
(defun queue*-= (q1 q2)
  (= (app (first q1) (rev (second q1)))
     (app (first q2) (rev (second q2)))))
```

Recall the algebraic specifications for the queue ADT, re-expressed in ACL2, and using the `*` operations:

$$\texttt{(isEmpty* (empty*))}$$

$$\texttt{(integerp a)} \wedge \texttt{(queuep* q)} \Longrightarrow \neg\texttt{(isEmpty* (enqueue* a q))}$$

$$\texttt{(integerp a)} \wedge \texttt{(queuep* q)} \wedge \texttt{(isEmpty* q)}$$
$$\Longrightarrow \texttt{(= (front* (enqueue* a q)) a)}$$

$$\texttt{(integerp a)} \wedge \texttt{(queuep* q)} \wedge \neg\texttt{(isEmpty* q)}$$
$$\Longrightarrow \texttt{(= (front* (enqueue* a q)) (front* q))}$$

$$\texttt{(integerp a)} \wedge \texttt{(queuep* q)} \wedge \texttt{(isEmpty* q)}$$
$$\Longrightarrow \texttt{(queue*-= (dequeue* (enqueue* a q)) (empty*))}$$

$$\texttt{(integerp a)} \wedge \texttt{(queuep* q)} \wedge \neg\texttt{(isEmpty* q)}$$
$$\Longrightarrow \texttt{(queue*-= (dequeue* (enqueue* a q)) (enqueue* a (dequeue* q)))}$$

We can prove that these hold. Let's try the last one, just for kicks. The context is

A1 : `(integerp a)`

A2 : `(queuep* q)`

A3 : ¬`(isEmpty* q)`

Note that by expanding A3, we get:

A4 : ¬`(endp (first q))` ∨ ¬`(endp (second q))`

and let's prove:

```
(queue*-= (dequeue* (enqueue* a q)) (enqueue* a (dequeue* q)))
```

by definition of `enqueue*`

```
(queue*-= (dequeue* (list (cons a (first q)) (second q)))
          (enqueue* a (dequeue* q)))
```

At this point, we do a case analysis on `(endp (second q))`.
First case: we add to the context:

A5 : `(endp (second q))`

and note that A5 and A4 together imply that ¬`(endp (first q))`, which we can add to the context:

A6 : ¬`(endp (first q))`

and continue the proof:

```
(queue*-= (dequeue* (list (cons a (first q)) (second q)))
          (enqueue* a (dequeue* q)))
```

by definition of `dequeue*` *and A5*

```
(queue*-= (list nil (cdr (rev (cons a (first q)))))
          (enqueue* a (dequeue* q)))
```

by definition of `rev`

```
(queue*-= (list nil (cdr (rev (cons a (first q)))))
          (enqueue* a (dequeue* q)))
```

by definition of `dequeue*` *and A5*

```
(queue*-= (list nil (cdr (rev (cons a (first q)))))
          (enqueue* a (list nil (cdr (rev (first q))))))
```

by definition of `enqueue*`

```
(queue*-= (list nil (cdr (rev (cons a (first q)))))
          (list (list a) (cdr (rev (first q)))))
```

$\boxed{\textit{by definition of } \texttt{rev}}$

```
(queue*-= (list nil (cdr (app (rev (first q)) (list a))))
          (list (list a) (cdr (rev (first q)))))
```

$\boxed{\textit{by definition of } \texttt{queue*-=}}$

```
(= (app nil (rev (cdr (app (rev (first q)) (list a)))))
   (app (list a) (rev (cdr (rev (first q))))))
```

$\boxed{\textit{by definition of } \texttt{app}}$

```
(= (rev (cdr (app (rev (first q)) (list a)))))
   (app (list a) (rev (cdr (rev (first q))))))
```

$\boxed{\textit{by definition of } \texttt{app} \textit{ and A6}}$

```
(= (rev (app (cdr (rev (first q))) (list a)))
   (app (list a) (rev (cdr (rev (first q))))))
```

$\boxed{\textit{by theorem for } \texttt{(rev (app a b))}}$

```
(= (app (rev (list a)) (rev (cdr (rev (first q)))))
   (app (list a) (rev (cdr (rev (first q))))))
```

$\boxed{\textit{by definition of } \texttt{rev}}$

```
(= (app (list a) (rev (cdr (rev (first q)))))
   (app (list a) (rev (cdr (rev (first q))))))
```

$\boxed{\textit{by reflexivity of } \texttt{=}}$

That's the first case.

We now go back to where we did case analysis and do the second case, by first adding to the context:

A5 : ¬`(endp (second q))`

and continue the proof:

```
(queue*-= (dequeue* (list (cons a (first q)) (second q)))
          (enqueue* a (dequeue* q)))
```

$\boxed{\textit{by definition of } \texttt{dequeue*} \textit{ and A5}}$

```
(queue*-= (list (cons a (first q)) (cdr (second q)))
          (enqueue* a (dequeue* q)))
```

by definition of `dequeue*` *and A5*

```
(queue*-= (list (cons a (first q)) (cdr (second q)))
          (enqueue* a (list (first q) (cdr (second q)))))
```

by definition of `enqueue*`

```
(queue*-= (list (cons a (first q)) (cdr (second q)))
          (list (cons a (first q)) (cdr (second q))))
```

by definition of `queue*-=`

```
(= (app (cons a (first q)) (rev (cdr (second q))))
   (app (cons a (first q)) (rev (cdr (second q)))))
```

by reflexivity of `=`

There. Not a hard proof, but somewhat long. Try to push it through in ACL2. (You'll need lemmas, as you realize if you work through the proofs above.) The other proofs are similar, and in fact easier, since they do not involve `queue*-=`.

So this implementation of queues also satisfies the queue ADT specification. Meaning, in particular, that if you apply any given sequence of operations starting from an empty queue, and you observe the resulting queue, either using `isEmpty` or `front`, you will get the same result irrespective of the queue implementation. Thus, if the queue library is a black box that doesn't let you look inside how queues are represented, the `isEmpty` and `front` can be called *observations*, because they let you observe something concrete about the queues. What we can conclude from the discussion above is that given two queues derived using the same sequence of operations, but each using one of the two implementations, are *observationally equivalent*: they cannot be distinguished using the observation functions. Observational equivalence tells you can swap implementations of ADTs without affecting what your program computes.

For the sake of completeness, let's suppose we have both implementations of queues, but we only have the signature of the queue ADT, but not a formal specification. I want to argue that we can still establish that queues constructed using the same sequence of operations are observationally equivalent.

The way to do that is to relate the representation of the queues in both implementations. In fact, if you start from an empty queue in both implementations, and apply the same sequence of operations, say enqueue 1, 2, 3, then dequeue once, then enqueue 4, you see that you get the following sequence of representations, with the representation of the simple list-based implementation in the first column and the two-lists implementation in the second column:

```
(empty)                    ()          ( () () )
```

```
(enqueue 1...           (1)        ( (1) () )
(enqueue 2...           (1 2)      ( (2 1) () )
(enqueue 3...           (1 2 3)    ( (3 2 1) () )
(dequeue...             (2 3)      ( () (2 3) )
(enqueue 4...           (2 3 4)    ( (4) (2 3) )
```

You see that there is an *equivalence* between the two representations: if take the second representation as `( L M )`, and you append `M` to the reverse of `L`, you get the first representation. What we want to show now is that this equivalence is an *invariant* of the queue operations in the respective implementations.

Let's formalize the equivalence above via a function `equiv`:

```
(defun equiv (q q*)
  (= q (app (second q*) (rev (first q*)))))
```

My claim is that all operations that return queues will return equivalent when they are given equivalent queues. This is captured by the following formulas:

`(equiv (empty) (empty*))`

`(integerp a)` ∧ `(queuep q)` ∧ `(queuep* q*)` ∧ `(equiv q q*)`
    ⟹ `(equiv (enqueue a q) (enqueue* a q*))`

`(queuep q)` ∧ `(queuep* q*)` ∧ `(equiv q q*)` ⟹ `(equiv (dequeue q) (dequeue* q*))`

As I said, these formulas say that `equiv` is an invariant of the queue operations. Thus, if we start with two empty queues, one in each implementation of queues, and we apply any sequence of enqueues and dequeues, we end up with equivalent resulting queues. Why is this interesting? Because equivalent queues yield the same observations, as the following provable formulas show:

`(queuep q)` ∧ `(queuep* q*)` ∧ `(equiv q q*)` ⟹ `(= (isEmpty q) (isEmpty* q*))`

`(queuep q)` ∧ `(queuep* q*)` ∧ `(equiv q q*)` ⟹ `(= (front q) (front* q*))`

Two equivalent queues yield the same observations. This is our notion of observational equivalence, which is here established for the two implementations of queues without relying on an externally-imposed algebraic specification.