Problems with The Lazy Queue

The lazy queue operations are sometimes expensive because rotation is done monolithically:

\[ \ldots \text{fs ++ reverse bs } \ldots \]

To achieve \( O(1) \) worst-case time bound, we need to:

- do rotation incrementally
- schedule to evaluate it often

This spreads out the cost of the expensive operation.
Incremental Rotation

Rather than relying on reverse to rotate, we write our own rotation:

\[
\begin{align*}
\text{rotate} \; [] \; (y:\[]) \; a & = y:a \\
\text{rotate} \; (x:xs) \; (y:ys) \; a & = x : \text{rotate} \; xs \; ys \; (y:a)
\end{align*}
\]

(We ignore \( \text{rotate} \; [] \; y:ys \; a \) with \( ys \neq [] \). We will make sure it never happens.)

So \( \text{rotate} \; xs \; ys \; [] \) has the same value as \( xs \; ++ \; \text{reverse} \; ys \), except that it is incrementally lazy. The parameter \( a \) acts as a kind of accumulator.

More generally, \( \text{rotate} \; xs \; ys \; a \; == \; xs \; ++ \; \text{reverse} \; ys \; ++ \; a \).
**O(1) Lazy Queue**

The queue has a front list, a back list, and a schedule:

\[
data\ SQueue\ a = SQ\ [a]\ [a]\ [a]\end{equation}

The schedule holds an unevaluated `rotate` expression. It is usually a suffix of the front list. By evaluating it once in a while, we discharge the rotation incrementally.

\[
\text{snoc}\ (SQ\ f\ b\ s)\ x = \text{exec}\ f\ (x:b)\ s
\]
\[
\text{head}\ (SQ\ (h:f)\ b\ s) = h
\]
\[
\text{tail}\ (SQ\ (h:f)\ b\ s) = \text{exec}\ f\ b\ s
\]

Here `exec` evaluates the schedule once and returns the new queue.
Scheduling

exec evaluates the schedule by matching it against patterns. It also forgets the head so that the rest gets evaluated next time.

If the schedule becomes empty, we simply create a new rotation as the new schedule (and as the new front list). It is this time when the back list is emptied.

\[
\text{exec } f \ b \ (x:s) = SQ \ f \ b \ s \\
\text{exec } f \ b \ [] = SQ \ f' \ [] \ f' \ \text{where } f' = \text{rotate} \ f \ b \ []
\]

Now each operation takes \(O(1)\) time.