RANJIT JHALA, ERIC SEIDEL, NIKI VAZOU

PROGRAMMING WITH REFINEMENT TYPES

AN INTRODUCTION TO LIQUIDHASKELL

Version 13, July 20th, 2020.

HTTPS://UCSD-PROGSYS.GITHUB.IO/LIQUIDHASKELL-BLOG/

Contents

List of Exercises

1

Introduction

Welcome to the LiquidHaskell Short Tutorial Cheat Sheet!

Here are the main concepts and examples you can use to complete the exercises.

Adding Specifications

LiquidHaskell specifications in functions are written between `{-@ spec @-}`.

For example:

```
`{-@ calcPer ::  a:Int -> {b:Int | 0 <= b && b <= a} -> c:Int @-}`
calcPer       :: Int -> Int -> Int
calcPer a b    = (b * 100) `div` a
```

Alias

To reuse a specification we can use aliases.

For example:

```
{-@ type Nat      = {v:Int | 0 <= v}        @-}
{-@ type Positive = {v:Int | 0 < v}         @-}
```

Liquid types in Datatypes

To add a specification to the datatypes, first create the datatype in Haskell and then add the specification inside `{-@ @-}`.

For example:

1) In Haskell

```
data Sparse a = SP { spDim   :: Int
                   , spElems :: [(Int, a)] }
```

2) Adding the LiquidHaskell specifiaction

```
{-@ data Sparse a = SP { spDim   :: Nat
                       , spElems :: [(Btwn 0 spDim, a)]} @-}
```

Measures

Measures lift an Haskell function to the refinements logic. It is first created as a Haskell function, sinalizing that it is a measure and adding liquid types to the signature. Then, it can be used inside other refinements.

For example:

```
{-@ measure size @-}
{-@ size :: [a] -> Nat @-}
size []     = 0
size (_:rs) = 1 + size rs
```

And then, `size` can be used in:

```
{-@ type ListN a N = {v:[a] | size v == N} @-}
```

2

Introduction

Welcome to the LiquidHaskell Short Tutorial, where you will learn the basic workings of LiquidHaskell and complete some exercises. The full version of the tutorial can be found in the project's website.

One of the great things about Haskell is its brainy type system that allows one to enforce a variety of invariants at compile time, thereby nipping in the bud a large swathe of run-time errors.

Well-Typed Programs Do Go Wrong

Alas, well-typed programs *do* go quite wrong, in a variety of ways.

Division by Zero This innocuous function computes the average of a list of integers:

```
average    :: [Int] -> Int
average xs = sum xs `div` length xs
```

We get the desired result on a non-empty list of numbers:

```
ghci> average [10, 20, 30, 40]
25
```

However, this program crashes with certain arguments. From the following options, what argument would make `average` crash?

[1] [] [1,1,1,1,1,1,1,1,1,1,1] Submit

Answer

If we call it with an empty list, we get a rather unpleasant crash: *** Exception: divide by zero. We could write `average` more *defensively*, returning a `Maybe` or `Either` value. However, this merely kicks the can down the road. Ultimately, we will want to extract the `Int` from the `Maybe` and if the inputs were invalid to start with, then at that point we'd be stuck.

Heart Bleeds

For certain kinds of programs, there is a fate worse than death. `text` is a high-performance string processing library for Haskell, that is used, for example, to build web services.

```
ghci> :m +Data.Text Data.Text.Unsafe
ghci> let t = pack "Voltage"
ghci> takeWord16 5 t
"Volta"
```

A cunning adversary can use invalid, or rather, *well-crafted*, inputs that go well outside the size of the given `text` to read extra bytes and thus *extract secrets* without anyone being any the wiser.

```
ghci> takeWord16 20 t
"Voltage\1912\3148\SOH\NUL\15928\2486\SOH\NUL"
```

The above call returns the bytes residing in memory *immediately after* the string `Voltage`. These bytes could be junk, or could be either the name of your favorite TV show, or, more worryingly, your bank account password.

Refinement Types

Refinement types allow us to enrich Haskell's type system with *predicates* that precisely describe the sets of *valid* inputs and outputs of functions, values held inside containers, and so on. These predicates are drawn from special *logics* for which there are fast *decision procedures* called SMT solvers.

BY COMBINING TYPES WITH PREDICATES you can specify *contracts* which describe valid inputs and outputs of functions. The refinement type system *guarantees at compile-time* that functions adhere to their contracts. That is, you can rest assured that the above calamities *cannot occur at run-time*.

LiquidHaskell is a Refinement Type Checker for Haskell, and in this tutorial we'll describe how you can use it to make programs better and programming even more fun.

As a glimpse of what LiquidHaskell can do, run the average example below by pushing the green triangle on the top, and try to read the error message. Since `div` cannot take a zero value as the second argument, and LiquidHaskell sees that it is a possibility in this function, an error will be raised.

```
average'    :: [Int] -> Int
average' xs =  sum xs `div` length xs
```

In this tutorial you will learn how to add and reason about refinement types in Haskell, and how it can increase the realiability of Haskell problems.

Next

3

Refinement Types

What is a Refinement Type? In a nutshell,

> *Refinement Types = Types + Predicates*

That is, refinement types allow us to decorate types with *logical predicates*, which you can think of as *boolean-valued* Haskell expressions, that constrain the set of values described by the type. This lets us specify sophisticated invariants of the underlying values.

Defining Types

Let us define some refinement types:

```
{-@ type Zero    = {v:Int | v == 0} @-}
{-@ type NonZero = {v:Int | v /= 0} @-}
```

The Value Variable `v` denotes the set of valid inhabitants of each refinement type. Hence, `Zero` describes the *set of* `Int` values that are equal to `0`, that is, the singleton set containing just `0`, and `NonZero` describes the set of `Int` values that are *not* equal to `0`, that is, the set `{1, -1, 2, -2, ...}` and so on.

To indicate that these specifications are for LiquidHaskell specifications we write them like `{-@ spec @-}`.

Now, to use these types we can write:

```
{-@ zero :: Zero @-}
zero  = 0 :: Int

{-@ one, two, three :: NonZero @-}
one   = 1 :: Int
two   = 2 :: Int
three = 3 :: Int
```

Errors

If we try to say nonsensical things like:

```
nonsense :: Int
nonsense = one'
  where
  {-@ one' :: Zero @-}
  one' = 1
```

LiquidHaskell will complain with an error message:

```
../liquidhaskell-tutorial/src/03-basic.lhs:72:3-6: Error: Liquid Type Mismatch

72 |   one' = 1 :: Int
       ^^^^

  Inferred type
    VV : {VV : Int | VV == (1 : int)}

  not a subtype of Required type
    VV : {VV : Int | VV == 0}
```

The message says that the expression `1 :: Int` has the type

```
    {v:Int | v == 1}
```

which is *not* (a subtype of) the *required* type

```
    {v:Int | v == 0}
```

as `1` is not equal to `0`.

Subtyping

What is this business of *subtyping*? Suppose we have some more refinements of `Int`

```
{-@ type Nat      = {v:Int | 0 <= v}        @-}
{-@ type Positive = {v:Int | 0 < v}         @-}
{-@ type Even     = {v:Int | v mod 2 == 0 } @-}
{-@ type BtwZeroHundred = {v:Int | v >= 0 && v <= 100}         @-}
```

Subtyping and Implication

`Zero` is the most precise type for `0::Int`, as it is a *subtype* of `Nat`, `Even` and `BtwZeroHundred`. However, it is not a subtype of `Positive`.

Now let us try a new predicate. Write a type for the numbers that represent a percentage (between 0 and 100) by replacing the `true` predicate. Then run the code, and the first example should be correct and the second should not.

```
{-@ type Percentage = true  @-}

{-@ percentT  :: Percentage  @-}
percentT    = 10 :: Int
{-@ percentF  :: Percentage  @-}
percentF :: Int
percentF    = 10 + 99 :: Int
```

Answer

{-@ type Percentage = {v:Int | 0 <= v && v <= 100} @-}

In Summary the key points about refinement types are:

1. A refinement type is just a type *decorated* with logical predicates.
2. A term can have *different* refinements for different properties.
3. When we *erase* the predicates we get the standard Haskell types.

Writing Specifications

We can also add specifications as pre- and post-conditions of functions.

Remember the divide function from before? We can add the case of dividing by zero with this `die "message"` to indicate that this case should be handled before running the code.

```
divide'     :: Int -> Int -> Int
divide' n 0 = die "divide by zero"
divide' n d = n `div` d
```

So, now we can specify that the first case will never with a *pre-condition* that says that the second argument is non-zero:

```
{-@ divide :: Int -> NonZero -> Int @-}
divide _ 0 = die "divide by zero"
divide n d = n `div` d
```

You can run the both pieces of code and check that the first one throws an error while the second one does not since it can infer that the first case will not be called.

Establishing Pre-conditions

The above signature forces us to ensure that that when we *use* `divide`, we only supply provably `NonZero` arguments.

Select which of the following functions that call divide would raise an error:

abc x y = divide (x + y) 2 efg x y z = divide (divide (x + y) 3) 10 hij x y z = divide (x + y) z Submit

Answer

```
`hij` is the invocation that could trigger a crash since we have no guarantees that z is a `NonZero` value.

</div>
```

Refining Function Types: Post-conditions

Next, let's see how we can use refinements to describe the *outputs* of a function. Consider the following simple *absolute value* function

```
abs           :: Int -> Int
abs n
  | 0 < n     = n
  | otherwise = 0 - n
```

We can use a refinement on the output type to specify that the function returns non-negative values

```
{-@ abs :: Int -> Nat @-}
```

LiquidHaskell *verifies* that `abs` indeed enjoys the above type by deducing that `n` is trivially non-negative when `0 < n` and that in the `otherwise` case, the value `0 - n` is indeed non-negative.

Dependent Refinements

The predicates in pre- and post- conditions can also refer to previous arguments of the function.

For example, including that the output is greater than the input.

```
{-@ plus1 :: a:Int -> {b:Int |b > a}@-}
plus1 :: Int -> Int
plus1 a = a + 1
```

And the same could be done between input values.

Exercise 3.1 (List Average)**.** *Can you now put everything together?*

Write a specification for the method `calcPer` *that:*

1) *first receives a positive int;*

2) *then an int with a value between zero and the first int;*

3) *returns a percentage;*

Use the aliases created in the exercises you have completed before.

```
calcPer       :: Int -> Int -> Int
calcPer a b    = (b * 100) `div` a



cpc = calcPer 10 5 :: Int  -- should be correct
cpc = calcPer 10 11 :: Int -- should be incorrect
```

Answer

`{-@ calcPer :: a:Positive -> {b:Int | 0 <= b && b <= a} -> c:Percentage @-}`

You finished the first part of the Tutorial!

Before moving to the next part, answer some questions from our team.

Next

4

Refined Datatypes

So far, we have seen how to refine the types of *functions*, to specify, for example, pre-conditions on the inputs, or post-conditions on the outputs. Very often, we wish to define *datatypes* that satisfy certain invariants. In these cases, it is handy to be able to directly refine the `data` definition, making it impossible to create illegal inhabitants.

Sparse Vectors

As our first example of a refined datatype, let's see Sparse Vectors. While the standard Vector is great for dense arrays, often we have to manipulate sparse vectors where most elements are just 0. We might represent such vectors as a list of index-value tuples `[(Int, a)]`.

Let's create a new datatype to represent such vectors:

```
data Sparse a = SP { spDim   :: Int
                   , spElems :: [(Int, a)] }
```

Thus, a sparse vector is a pair of a dimension and a list of index-value tuples. Implicitly, all indices *other* than those in the list have the value `0` or the equivalent value type `a`.

Legal

`Sparse` vectors satisfy two crucial properties. 1) the dimension stored in `spDim` is non-negative; 2) every index in `spElems` must be valid, i.e. between `0` and the dimension.

Unfortunately, Haskell's type system does not make it easy to ensure that *illegal vectors are not representable*.

DATA INVARIANTS LiquidHaskell lets us enforce these invariants with a refined data definition:

```
{-@ data Sparse a = SP { spDim   :: Nat
                       , spElems :: [(Btwn 0 spDim, a)]} @-}
```

Where, as before, we use the aliases:

```
{-@ type Nat        = {v:Int | 0 <= v}            @-}
{-@ type Btwn Lo Hi = {v:Int | Lo <= v && v < Hi} @-}
```

REFINED DATA CONSTRUCTORS The refined data definition is internally converted into refined types for the data constructor `SP`. So, by using refined input types for `SP` we have automatically converted it into a *smart* constructor that ensures that *every* instance of a `Sparse` is legal. Consequently, LiquidHaskell verifies:

```
okSP :: Sparse String
okSP = SP 5 [ (0, "cat")
            , (3, "dog") ]
```

but rejects, due to the invalid index:

```
badSP :: Sparse String
badSP = SP 5 [ (0, "cat")
             , (6, "dog") ]
```

Write another example of a Sparse data type that is invalid.

```
badSP' :: Sparse String
```

Answer

e.g., `badSP' = SP -1 [(0, "cat")]`

FIELD MEASURES It is convenient to write an alias for sparse vectors of a given size `N`. So that we can easily say in a refinement that we have a sparse vector of a certain size.

For this we can use *measures*.

MEASURES are used to define *properties* of Haskell data values that are useful for specification and verification.

A MEASURE is a *total* Haskell function, 1. With a *single* equation per data constructor, and 2. Guaranteed to *terminate*, typically via structural recursion.

We can tell LiquidHaskell to *lift* a function meeting the above requirements into the refinement logic by declaring:

```
{-@ measure nameOfMeasure @-}
```

For example, for a list we can define a way to *measure* its size with the following function.

```
{-@ measure size @-}
{-@ size :: [a] -> Nat @-}
size []     = 0
size (_:rs) = 1 + size rs
```

Then, we can use this measure to define aliases.

But first, let's create another measure named `notEmpty` that takes a list as input and returns a `Bool` with the information if it is empty or not.

```
{-@ measure notEmpty @-}
```

Answer

`{-@ measure notEmpty @-}` `notEmpty       :: [a] -> Bool` `notEmpty []    = False` `notEmpty (_:_) = True`

We can now define a couple of useful aliases for describing lists of a given dimension.

And now, we can define that a list has exactly `N` elements.

```
{-@ type ListN a N = {v:[a] | size v == N} @-}
```

Note that when defining refinement type aliases, we use uppercase variables like `N` to distinguish *value* parameters from the lowercase *type* parameters like `a`.

Now, try to create an alias for an empty list, using the measure `notEmpty` created before. The first example should raise an error while the second should not.

```
{-@ type NEList a = {true} @-}

{-@ ne1 :: NEList Int@-}
ne1 = [] ::  [Int]
{-@ ne1 :: NEList Int@-}
ne2 = [1,2,3,4] :: [Int]
```

Answer

<div id="collapsibleDiv4">

```
{-@ type NEList a = {v:[a] | notEmpty v} @-}
```

Measures with Sparse Vectors

Similarly, the sparse vector also has a *measure* for its dimension, but in this case it is already defined by `spDim`, so we can use it to create the new alias of sparse vectors of size N.

Now, following what we did with the lists, write the alias for sparse vector, using `spDim` instead of `size`.

```
{-@ type SparseN a N = {true} @-}
```

Answer

<div id="collapsibleDiv5">

```
{-@ type SparseN a N = {v:Sparse a | spDim v == N} @-}
```

```
/div>
```

`Vectors` are similar to Sparse Vectors, and therefore, have a *measure* of size named `vlen`.

Sparse Products

So, now, we can see that LiquidHaskell is able to compute a sparse product, making the product of all the same indexes and returning its sum. Run the code ahead.

```
{-@ dotProd :: x:Vector Int -> SparseN Int (vlen x) -> Int @-}
dotProd x (SP _ y) = go 0 y
  where
    go sum ((i, v) : y') = go (sum + (x ! i) * v) y'
    go sum []            = sum
```

LiquidHaskell verifies the above by using the specification to conclude that for each tuple `(i, v)` in the list `y`, the value of `i` is within the bounds of the vector `x`, thereby proving `x ! i` safe.

%

FOLDED PRODUCT We can port the `fold`-based product % to our new representation:

%

```
% {-@ dotProd' :: x:Vector Int -> SparseN Int (vlen x) -> Int @-}
% dotProd' x (SP _ y) = foldl' body 0 y
%   where
%     body sum (i, v) = sum + (x ! i)  * v
%
```

% As before, LiquidHaskell checks the above by % automatically instantiating refinements % for the type parameters of `foldl'`, saving us a fair % bit of typing and enabling the use of the elegant % polymorphic, higher-order combinators we know and love.

YOU FINISHED THE SECOND PART OF THE TUTORIAL!

Before moving to the next part, answer some questions from our team.

NEXT EXERCISE!

Now that you have learned the main blocks of LiquidHaskell, lets complete an exercise using all the concepts.

You can open a Cheat Sheet with examples of the main concepts on the side.

Next Case Study: Okasaki's Lazy Queues {#lazyqueue}
=================================

Lets test what we learned so far in a case study that is simple enough to explain without pages of code, yet complex enough to show off whats cool about dependency: Chris Okasaki's beautiful Lazy Queues. This structure leans heavily on an invariant to provide fast *insertion* and *deletion*. Let's see how to enforce that invariant with LiquidHaskell.

Queues

A queue is a structure into which we can `insert` and `remove` data such that the order in which the data is removed is the same as the order in which it was inserted.

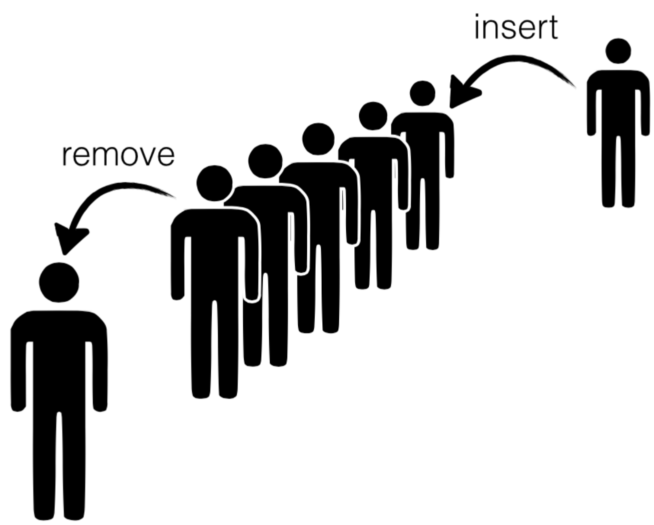

Figure 4.1: A Queue is a structure into which we can insert and remove elements. The order in which the elements are removed is the same as the order in which they were inserted.

To efficiently implement a queue we need to have rapid access to both the front as well as the back because we `remove` elements from former and `insert` elements into the latter. This is quite straightforward with explicit pointers and mutation – one uses an old school linked list and maintains pointers to the head and the tail. But can we implement the structure efficiently without having stoop so low?

Chris Okasaki came up with a very cunning way to implement queues using a *pair* of lists – let's call them `front` and `back` which represent the corresponding parts of the Queue.

- To `insert` elements, we just *cons* them onto the `back` list,
- To `remove` elements, we just *un-cons* them from the `front` list.

The catch is that we need to shunt elements from the back to the front every so often, e.g. we can transfer the elements from the `back` to the `front`, when:

1. a `remove` call is triggered, and

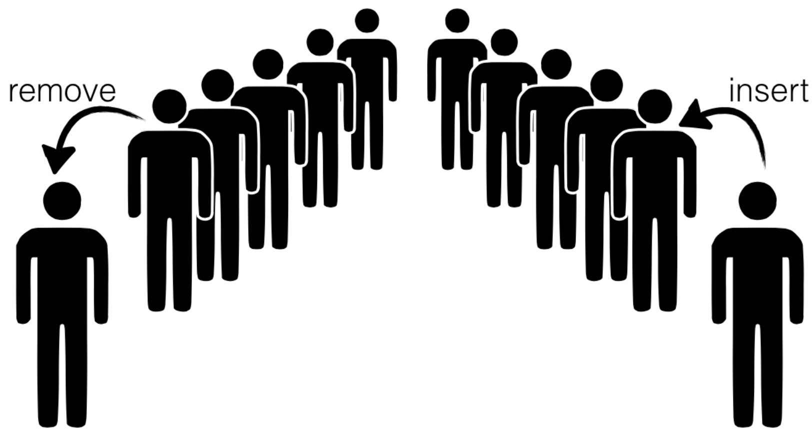

Figure 4.2: We can implement a Queue with a pair of lists; respectively representing the front and back.

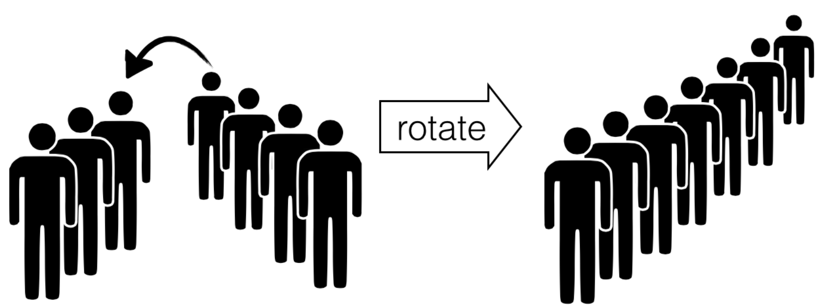

Figure 4.3: Transferring Elements from back to front.

2. the `front` list is empty.

OKASAKI'S FIRST INSIGHT was to note that every element is only moved *once* from the back to the front; hence, the time for `insert` and `remove` could be `O(1)` when *amortized* over all the operations. This is perfect, *except* that some set of unlucky `remove` calls (which occur when the `front` is empty) are stuck paying the bill. They have a rather high latency up to `O(n)` where `n` is the total number of operations.

OKASAKI'S SECOND INSIGHT saves the day: he observed that all we need to do is to enforce a simple *balance invariant*:

$$\text{Size of front} \geq \text{Size of back}$$

If the lists are lazy i.e. only constructed as the head value is demanded, then a single `remove` needs only a tiny `O(log n)` in the worst case, and so no single `remove` is stuck paying the bill.

LETS IMPLEMENT QUEUES and ensure the crucial invariant(s) with LiquidHaskell. What we need are the following ingredients:

1. A type for `List`s, and a way to track their `size`,
2. A type for `Queue`s which encodes the balance invariant
3. A way to implement the `insert`, `remove` and `transfer` operations.

Sized Lists

The first part is super easy. Let's define a type:

```haskell
data SList a = SL { size :: Int, elems :: [a] }
```

We have a special field that saves the `size` because otherwise, we have a linear time computation that wrecks Okasaki's careful analysis. (Actually, he presents a variant which does *not* require saving the size as well, but that's for another day.)

How can we be sure that `size` is indeed the *real size* of `elems`? Write a function to *measure* the real size:

```
{-@ measure realSize @-}
```

Answer

`{-@ measure realSize @-} realSize :: [a] -> Int realSize [] = 0 realSize (_:xs) = 1 + realSize xs`

Now, we can specify a *refined* type for `SList` that ensures that the *real* size is saved in the `size` field.

```
{-@ data SList a = SL {
      size  :: Nat
    , elems :: {v:[a] | realSize v = size}
    }
@-}
```

As a sanity check, consider this:

```
okList  = SL 1 ["cat"]    -- accepted

badList = SL 1 []         -- rejected
```

Lets define an alias for lists of a given size `N`:

```
{-@ type SListN a N = {v:SList a | size v = N} @-}
```

Now define an alias for lists that are not empty:

```
{-@ type NEList a = ?? @-}
```

Answer

`{-@ type NEList a = {v:SList a | size v > 0} @-}`

Finally, we can define a basic API for `SList`.

To Construct lists, we use `nil` and `cons`:

```
{-@ nil :: SListN a 0 @-}
nil = SL 0 []

{-@ cons :: a -> xs:SList a -> SListN a {size xs + 1} @-}
cons x (SL n xs) = SL (n+1) (x:xs)
```

Exercise 4.1 (Destructing Lists)**.** *We can destruct lists by writing a* `hd` *and* `tl` *function as shown below. Now, fix the specification on both functions so the definitions typecheck.*

```
{-@ tl           :: xs:SList a -> SListN a {size xs + 1}  @-}
tl (SL n (_:xs)) = SL (n-1) xs
tl _             = die "empty SList"

{-@ hd           :: xs:SList a -> a @-}
hd (SL _ (x:_))  = x
hd _             = die "empty SList"
```

Hint: When you are done, `okHd` should be verified, but `badHd` should be rejected.

```
{-@ okList :: SListN String 1 @-}

okHd  = hd okList       -- accepted

badHd = hd (tl okList)  -- rejected
```

Answer

```
{-@ tl           :: xs:NEList a -> SListN a {size xs - 1}
@-}
```

```
tl (SL n (_:xs)) = SL (n-1) xs
```

```
{-@ hd           :: xs:NEList a -> a @-}
```

```
hd (SL _ (x:_))  = x
```

Queue Type

It is quite straightforward to define the `Queue` type, as a pair of lists, `front` and `back`, such that the latter is always smaller than the former:

```
{-@ data Queue a = Q {
      front :: SList a
    , back  :: SListLE a (size front)
    }
@-}
data Queue a = Q
  { front :: SList a
  , back  :: SList a
  }
```

THE ALIAS `SListLE a L` corresponds to lists with at most `N` elements:

```
{-@ type SListLE a N = {v:SList a | size v <= N} @-}
```

As a quick check, notice that we *cannot represent illegal Queues*:

```
okQ  = Q okList nil  -- accepted, |front| > |back|

badQ = Q nil okList  -- rejected, |front| < |back|
```

Queue Operations

Almost there! Now all that remains is to define the `Queue` API. The code below is more or less identical to Okasaki's (I prefer `front` and `back` to his `left` and `right`.)

THE EMPTY QUEUE is simply one where both `front` and `back` are both empty:

```
emp = Q nil nil
```

Exercise 4.2 (Queue Sizes)**.** *For the remaining operations we need some more information. Do the following steps:*

1. *Write a* measure *qsize to describe the queue size,*
2. *Use it to complete the definition of* `QueueN` *below, and*
3. *In the next step use QueueN.*

```
-- | create measuere qsize here

-- | Queues of size `N`
{-@ type QueueN a N = {v:Queue a | true} @-}

{-@ emp :: QueueN _ 0 @-}

{-@ example2Q :: QueueN _ 2 @-}
example2Q = Q (1 `cons` (2 `cons` nil)) nil

{-@ example0Q :: QueueN _ 0 @-}
example0Q = Q nil nil
```

Answer

{-@ measure qsize @-} qsize :: Queue a -> Int qsize (Q l r) = size l + size r

{-@ type QueueN a N = {v:Queue a | qsize v = N} @-}

To Remove an element we pop it off the `front` by using `hd` and `tl`. Notice that the `remove` is only called on non-empty `Queues`, which together with the key balance invariant (`makeq` that we will see later), ensures that the calls to `hd` and `tl` are safe.

Add a LiquidHaskell signature to remove using QueueN. When you are done, `okRemove` should be accepted, `badRemove` should be rejected.

```
remove (Q f b)   = (hd f, makeq (tl f) b)


okRemove  = remove example2Q   -- accept
badRemove = remove example0Q   -- reject
```

Answer

`{-@ remove       :: q:NEQueue a -> (a, QueueN a {qsize q - 1}) @-}`

`remove (Q f b)   = (hd f, makeq (tl f) b)`

To Insert an element we just `cons` it to the `back` list, and call the *smart constructor* `makeq` to ensure that the balance invariant holds:

Exercise 4.3 (Insert)**.** *Write down a type for* `insert` *such that* `replicate` *and* `okReplicate` *are accepted by LiquidHaskell, but* `badReplicate` *is rejected.*

```
insert e (Q f b) = makeq f (e `cons` b)

{-@ replicate :: n:Nat -> a -> QueueN a n @-}
replicate 0 _ = emp
replicate n x = insert x (replicate (n-1) x)

{-@ okReplicate :: QueueN _ 3 @-}
okReplicate = replicate 3 "Yeah!"  -- accept

{-@ badReplicate :: QueueN _ 3 @-}
badReplicate = replicate 1 "No!"   -- reject
```

Answer

```
{-@ insert       :: a -> q:Queue a -> QueueN a {qsize q + 1}
@-}
```

```
insert e (Q f b) = makeq f (econsb)
```

To Ensure the Invariant we use the smart constructor `makeq`, which is where the heavy lifting happens. The constructor takes two lists, the front `f` and back `b` and if they are balanced, directly returns the `Queue`, and otherwise transfers the elements from `b` over using the rotate function `rot` described next.

```
{-@ makeq :: f:SList a -> b:SListLE a {size f + 1 } -> QueueN a {size f + size b} @-}
makeq f b
  | size b <= size f = Q f b
  | otherwise        = Q (rot f b nil) nil
```

Exercise 4.4 (Rotate). ⋆⋆ *The Rotate function* `rot`*:*
1) is only called when `back` *is one larger than the* `front` *(we never let things drift beyond that).*

2) And the return size is the sum of the size in front, back and the additional to be rotated.

It is arranged so that it the `hd` *is built up fast, before the entire computation finishes; which, combined with laziness provides the efficient worst-case guarantee.*

As a last exercise, write down a type for `rot` so that it typechecks and verifies the type for `makeq`.

```
rot f b acc
  | size f == 0 = hd b `cons` acc
  | otherwise   = hd f `cons` rot (tl f) (tl b) (hd b `cons` acc)
```

Answer

```
{-@ rot :: f:SList a -> b:SListN _ {1 + size f} -> a:SList _
-> SListN _ {size f + size b + size a} @-}
```

Recap of everyting!

Well there you have it; Okasaki's beautiful lazy Queue, with the invariants easily expressed and checked with LiquidHaskell. This example is particularly interesting because

1. The refinements express invariants that are critical for efficiency,
2. The code introspects on the `size` to guarantee the invariants, and
3. The code is quite simple and we hope, easy to follow!

This exercise concludes the Short Tutorial of LiquidHaskell. Thank you for tagging along!