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, 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] 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

2

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:[1]

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

[1] You can read the type of `Zero` as: "`v` is an `Int` *such that* `v` equals `0`" and `NonZero` as : "`v` is an `Int` *such that* `v` does not equal `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 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 Lt100  = {v:Int | v < 100}         @-}
```

Subtyping and Implication

`Zero` is the most precise type for `0::Int`, as it is a *subtype* of `Nat`, `Even` and `Lt100`. 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
```

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.

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.

Predicates with variables that depend on each other

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 2.1 (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;*

Try using the aliases created before.

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


clacPer 10 5  -- should be correct
calcPer 10 11 -- should be incorrect
```

Answer

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

Next

3
Polymorphism

Refinement types shine when we want to establish properties of *polymorphic* datatypes and higher-order functions. Rather than be abstract, let's illustrate this with a classic use-case.

Array Bounds Verification aims to ensure that the indices used to retrieve values from an array are indeed *valid* for the array, i.e. are between `0` and the *size* of the array. For example, suppose we create an `array` with two elements:

```
twoLangs  = fromList ["haskell", "javascript"]
```

Lets attempt to look it up at various indices:

```
eeks      = [ok, yup, nono]
  where
    ok    = twoLangs ! 0
    yup   = twoLangs ! 1
    nono  = twoLangs ! 3
```

If we try to *run* the above, we get a nasty shock: an exception that says we're trying to look up `twoLangs` at index `3` whereas the size of `twoLangs` is just `2`.

```
Prelude> :l 03-poly.lhs
[1 of 1] Compiling VectorBounds     ( 03-poly.lhs, interpreted )
Ok, modules loaded: VectorBounds.
*VectorBounds> eeks
Loading package ... done.
"*** Exception: ./Data/Vector/Generic.hs:249 ((!)): index out of bounds (3,2)
```

Specification: Vector Bounds

First, let's see how to *specify* array bounds safety by *refining* the types for the key functions exported by `Data.Vector`, i.e. how to

1. *define* the size of a `Vector`
2. *compute* the size of a `Vector`
3. *restrict* the indices to those that are valid for a given size.

Imports

We can write specifications for imported modules – for which we *lack* the code – either directly in the client's source file or better, in `.spec` files which can be reused across multiple client modules.

```
-- | Define the size
measure vlen :: Vector a -> Int

-- | Compute the size
assume length :: x:Vector a -> {v:Int | v = vlen x}

-- | Lookup at an index
assume (!) :: x:Vector a -> {v:Nat | v < vlen x} -> a
```

Measures are used to define *properties* of Haskell data values that are useful for specification and verification. Think of `vlen` as the *actual* size of a `Vector` regardless of how the size was computed.

Assumes are used to *specify* types describing the semantics of functions that we cannot verify e.g. because we don't have the code for them. Here, we are assuming that the library function `Data.Vector.length` indeed computes the size of the input vector. Furthermore, we are stipulating that the lookup function `(!)` requires an index that is between `0` and the real size of the input vector `x`.

Dependent Refinements are used to describe relationships *between* the elements of a specification. For example, notice how the signature for `length` names the input with the binder `x` that then appears in the output type to constrain the output `Int`. Similarly, the signature for `(!)` names the input vector `x` so that the index can be constrained to be valid for `x`. Thus, dependency lets us write properties that connect *multiple* program values.

ALIASES are extremely useful for defining *abbreviations* for commonly occurring types. Just as we enjoy abstractions when programming, we will find it handy to have abstractions in the specification mechanism. To this end, LiquidHaskell supports *type aliases*. For example, we can define `Vectors` of a given size `N` as:

```
{-@ type VectorN a N = {v:Vector a | vlen v == N} @-}
```

and now use this to type `twoLangs` above as:

```
{-@ twoLangs :: VectorN String 2 @-}
twoLangs     = fromList ["haskell", "javascript"]
```

Similarly, we can define an alias for `Int` values between `Lo` and `Hi`:

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

after which we can specify `(!)` as:

```
(!) :: x:Vector a -> Btwn 0 (vlen x) -> a
```

Verification: Vector Lookup

Let's try to write some functions to sanity check the specifications. First, find the starting element – or `head` of a `Vector`

```
head     :: Vector a -> a
head vec = vec ! 0
```

When we check the above, we get an error:

```
    src/03-poly.lhs:127:23: Error: Liquid Type Mismatch
      Inferred type
        VV : Int | VV == ?a && VV == 0

      not a subtype of Required type
        VV : Int | VV >= 0 && VV < vlen vec

      In Context
        VV  : Int | VV == ?a && VV == 0
        vec : Vector a | 0 <= vlen vec
        ?a  : Int | ?a == (0  :  int)
```

What is the problem that the message is describing?

It does not know what is the ! operator The index should be greater than o because the head is not accessible Zero is not a valid index if the list is empty. Submit

Answer

LiquidHaskell is saying that `0` is *not* a valid index as it is not between `0` and `vlen vec`. Say what? Well, what if `vec` had *no* elements! A formal verifier doesn't make *off by one* errors.

To Fix the problem we can do one of two things.

1. *Require* that the input `vec` be non-empty, or
2. *Return* an output if `vec` is non-empty, or

Here's an implementation of the first approach, where we define and use an alias `NEVector` for non-empty `Vectors`

```
{-@ type NEVector a = {v:Vector a | 0 < vlen v} @-}

{-@ head' :: NEVector a -> a @-}
head' vec = vec ! 0
```

Exercise 3.1 (Vector Head). *Replace the* `undefined` *with an* implementation *of* `head''` *which accepts* all `Vectors` *but returns a value only when the input* `vec` *is not empty.*

```
head''     :: Vector a -> Maybe a
head'' vec = undefined
```

Exercise 3.2 (Unsafe Lookup). *The function* `unsafeLookup` *is a wrapper around the* `(!)` *with the arguments flipped. Modify the specification for* `unsafeLookup` *so that the* implementation *is accepted by LiquidHaskell.*

```
{-@ unsafeLookup :: Int -> Vector a -> a @-}
unsafeLookup index vec = vec ! index
```

Exercise 3.3 (Safe Lookup). *Complete the implementation of* `safeLookup` *by filling in the implementation of* `ok` *so that it performs a bounds check before the access.*

```
{-@ safeLookup :: Vector a -> Int -> Maybe a @-}
safeLookup x i
  | ok        = Just (x ! i)
```

```
  | otherwise = Nothing
  where
    ok        = undefined
```

Inference: Our First Recursive Function

Ok, let's write some code! Let's start with a recursive function that adds up the values of the elements of an `Int` vector.

```
-- >>> vectorSum (fromList [1, -2, 3])
-- 2
vectorSum         :: Vector Int -> Int
vectorSum vec     = go 0 0
  where
    go acc i
      | i < sz    = go (acc + (vec ! i)) (i + 1)
      | otherwise = acc
    sz            = length vec
```

Inference

LiquidHaskell verifies `vectorSum` – or, to be precise, the safety of the vector accesses `vec ! i`. The verification works out because LiquidHaskell is able to *automatically infer*

```
go :: Int -> {v:Int | 0 <= v && v <= sz} -> Int
```

which states that the second parameter `i` is between `0` and the length of `vec` (inclusive). LiquidHaskell uses this and the test that `i < sz` to establish that `i` is between `0` and `(vlen vec)` to prove safety.

Next Refined Datatypes {#refineddatatypes} ==========

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. First, the dimension stored in `spDim` is non-negative. Second, 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*.[1]

[1] The standard approach is to use abstract types and smart constructors but even then there is only the informal guarantee that the smart constructor establishes the right invariants.

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`:

```
-- Generated Internal representation
data Sparse a where
  SP :: spDim:Nat
     -> spElems:[(Btwn 0 spDim, a)]
     -> Sparse a
```

In other words, 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") ]
```

Field Measures It is convenient to write an alias for sparse vectors of a given size `N`. We can use the field name `spDim` as a *measure*, like `vlen`. That is, we can use `spDim` inside refinements[2]

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

[2] Note that *inside* a refined `data` definition, a field name like `spDim` refers to the value of the field, but *outside* it refers to the field selector measure or function.

Sparse Products

Let's write a function to compute a sparse product

```
{-@ 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.

Next Boolean Measures {#boolmeasures} ================

In the last two chapters, we saw how refinements could be used to reason about the properties of basic `Int` values like vector indices, or the elements of a list. Next, let's see how we can describe properties of aggregate structures like lists and trees, and use these properties to improve the APIs for operating over such structures.

Partial Functions

As a motivating example, let us return to the problem of ensuring the safety of division. Recall that we wrote:

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

The Precondition asserted by the input type `NonZero` allows LiquidHaskell to prove that the `die` is *never* executed at run-time, but consequently, requires us to establish that wherever `divide` is *used*, the second parameter be provably non-zero. This requirement is not onerous when we know what the divisor is *statically*

```
avg2 x y   = divide (x + y)     2

avg3 x y z = divide (x + y + z) 3
```

However, it can be more of a challenge when the divisor is obtained *dynamically*. For example, let's write a function to find the number of elements in a list

```
size        :: [a] -> Int
size []     =  0
size (_:xs) =  1 + size xs
```

and use it to compute the average value of a list:

```
avgMany xs = divide total elems
  where
    total  = sum  xs
    elems  = size xs
```

Uh oh. LiquidHaskell wags its finger at us!

```
src/04-measure.lhs:77:27-31: Error: Liquid Type Mismatch
  Inferred type
    VV : Int | VV == elems

  not a subtype of Required type
    VV : Int | 0 /= VV

  In Context
    VV    : Int | VV == elems
    elems : Int
```

We cannot prove that the divisor is `NonZero`, because it *can be* `0` – when the list is *empty*. Thus, we need a way of specifying that the input to `avgMany` is indeed non-empty!

Lifting Functions to Measures

How shall we tell LiquidHaskell that a list is *non-empty*? Recall the notion of `measure` previously introduced to describe the size of a `Data.Vector`. In that spirit, let's write a function that computes whether a list is not empty:

```
notEmpty       :: [a] -> Bool
notEmpty []    = False
notEmpty (_:_) = True
```

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 notEmpty @-}
```

Non-Empty Lists can now be described as the *subset* of plain old Haskell lists `[a]` for which the predicate `notEmpty` holds

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

We can now refine various signatures to establish the safety of the list-average function.

SIZE returns a non-zero value *if* the input list is not-empty. We capture this condition with an implication in the output refinement.

```
{-@ size :: xs:[a] -> {v:Nat | notEmpty xs => v > 0} @-}
```

AVERAGE is only sensible for non-empty lists. Add a specification to average using the type NEList.

```
{-@ average :: NEList Int -> Int @-}
average xs = divide total elems
  where
    total  = sum xs
    elems  = size xs
```

Answer

```
{-@ average :: NEList Int -> Int @-}
```

Exercise 3.4 (Debugging Specifications). *An important aspect of formal verifiers like LiquidHaskell is that they help establish properties not just of your* implementations *but equally, or more importantly, of your* specifications. *In that spirit, can you explain why the following two variants of* `size` *are* rejected *by LiquidHaskell?*

```
{-@ size1    :: xs:NEList a -> Pos @-}
size1 []     =  0
size1 (_:xs) =  1 + size1 xs

{-@ size2    :: xs:[a] -> {v:Int | notEmpty xs => v > 0} @-}
size2 []     =  0
size2 (_:xs) =  1 + size2 xs
```

Of course, we can do a lot more with measures, so let's press on!

Next Numeric Measures {#numericmeasure} ================

Many of the programs we have seen so far, for example those in here, suffer from *indexitis*. This is a term coined by Richard Bird which describes a tendency to perform low-level manipulations to iterate over the indices into a collection, opening the door to

various off-by-one errors. Such errors can be eliminated by instead programming at a higher level, using a wholemeal approach where the emphasis is on using aggregate operations, like `map`, `fold` and `reduce`.

WHOLEMEAL PROGRAMMING IS NO PANACEA as it still requires us to take care when operating on *different* collections; if these collections are *incompatible*, e.g. have the wrong dimensions, then we end up with a fate worse than a crash, a possibly meaningless result. Fortunately, LiquidHaskell can help. Lets see how we can use measures to specify dimensions and create a dimension-aware API for lists which can be used to implement wholemeal dimension-safe APIs.[3]

[3] In a later chapter we will use this API to implement K-means clustering.

Wholemeal Programming

Indexitis begone! As an example of wholemeal programming, let's write a small library that represents vectors as lists and matrices as nested vectors:

```
data Vector a = V { vDim  :: Int
                  , vElts :: [a]
                  }
              deriving (Eq)

data Matrix a = M { mRow  :: Int
                  , mCol  :: Int
                  , mElts :: Vector (Vector a)
                  }
              deriving (Eq)
```

THE DOT PRODUCT of two `Vectors` can be easily computed using a fold:

```
dotProd       :: (Num a) => Vector a -> Vector a -> a
dotProd vx vy = sum (prod xs ys)
  where
    prod      = zipWith (\x y -> x * y)
    xs        = vElts vx
    ys        = vElts vy
```

THE ITERATION embodied by the `for` combinator, is simply a `map` over the elements of the vector.

```
for            :: Vector a -> (a -> b) -> Vector b
for (V n xs) f = V n (map f xs)
```

WHOLEMEAL PROGRAMMING FREES us from having to fret about low-level index range manipulation, but is hardly a panacea. Instead, we must now think carefully about the *compatibility* of the various aggregates. For example,

- `dotProd` is only sensible on vectors of the same dimension; if one vector is shorter than another (i.e. has fewer elements) then we will won't get a run-time crash but instead will get some gibberish result that will be dreadfully hard to debug.
- `matProd` is only well defined on matrices of compatible dimensions; the number of columns of `mx` must equal the number of rows of `my`. Otherwise, again, rather than an error, we will get the wrong output.[4]

[4] In fact, while the implementation of `matProd` breezes past GHC it is quite wrong!

Specifying List Dimensions

In order to start reasoning about dimensions, we need a way to represent the *dimension* of a list inside the refinement logic. [5]

[5] We could just use `vDim`, but that is a cheat as there is no guarantee that the field's value actually equals the size of the list!

MEASURES are ideal for this task. Previously we saw how we could lift Haskell functions up to the refinement logic. Lets write a measure to describe the length of a list: [6]

[6] Recall that these must be inductively defined functions, with a single equation per data-constructor

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

MEASURES REFINE CONSTRUCTORS

As with refined data definitions, the measures are translated into strengthened types for the type's constructors. For example, the `size` measure is translated into:

```
data [a] where
  []  :: {v: [a] | size v = 0}
  (:) :: a -> xs:[a] -> {v:[a]|size v = 1 + size xs}
```

MULTIPLE MEASURES may be defined for the same data type. For example, in addition to the `size` measure, we can define a `notEmpty` measure for the list type:

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

WE COMPOSE DIFFERENT MEASURES

simply by *conjoining* the refinements in the strengthened constructors. For example, the two measures for lists end up yielding the constructors:

```
data [a] where
  []  :: {v: [a] | not (notEmpty v) && size v = 0}
  (:) :: a
      -> xs:[a]
      -> {v:[a]| notEmpty v && size v = 1 + size xs}
```

We are almost ready to begin creating a dimension aware API for lists; one last thing that is useful is a couple of aliases for describing lists of a given dimension.

TO MAKE SIGNATURES SYMMETRIC let's define an alias for plain old (unrefined) lists:

```
type List a = [a]
```

A LISTN is a list with exactly `N` elements, and a `ListX` is a list whose size is the same as another list `X`. Note that when defining refinement type aliases, we use uppercase variables like `N` and `X` to distinguish *value* parameters from the lowercase *type* parameters like `a`.

```
{-@ type ListN a N = {v:List a | size v = N} @-}
{-@ type ListX a X = ListN a {size X}        @-}
```

Lists: Size Preserving API

With the types and aliases firmly in our pockets, let us write dimension-aware variants of the usual list functions. The implementations are the same as in the standard library i.e. `Data.List`, but the specifications are enriched with dimension information.

Exercise 3.5 (Map). MAP *yields a list with the same size as the input. Fix the specification of* `map` *so that the* `prop_map` *is verified.*

```
{-@ map      :: (a -> b) -> xs:List a -> List b @-}
map _ []     = []
map f (x:xs) = f x : map f xs

{-@ prop_map :: List a -> TRUE @-}
prop_map xs = size ys == size xs
  where
    ys      = map id xs
```

Answer

```
{-@ map      :: (a -> b) -> xs: List a -> ListX b xs @-}
```

Now that we have seen the basics of LiquidHaskell, let us try a more complex exercise.

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

Lets start with 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.

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.

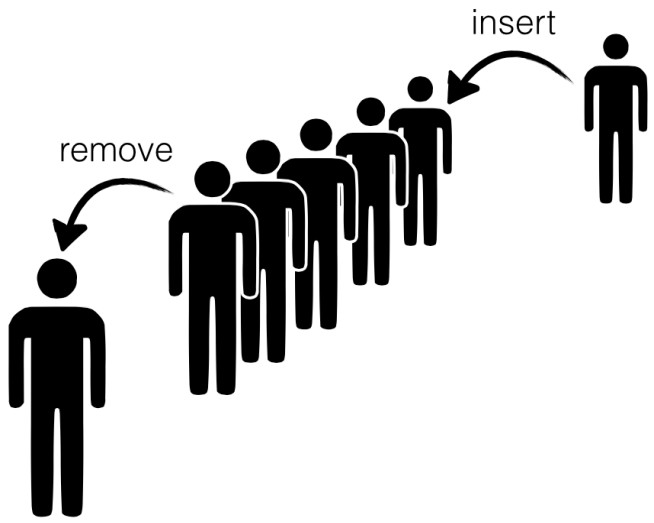

Figure 3.1: A Queue is a structure into which we can insert and remove elements. The order in which the elements are removed is the same as the order in which they were inserted.

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

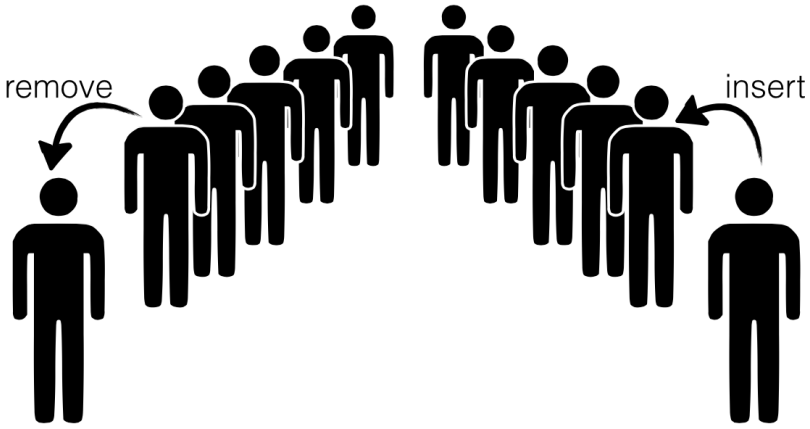

Figure 3.2: We can implement a Queue with a pair of lists; respectively representing the front and back.

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
2. the `front` list is empty.

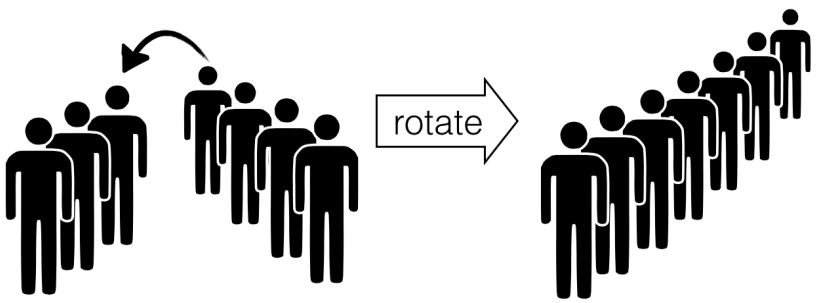

Figure 3.3: Transferring Elements from back to front.

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 `Lists`, and a way to track their `size`,
2. A type for `Queues` 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:

```
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 3.6 (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 3.7 (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. *Use it to give* `remove` *a type that verifies the safety of the calls made to* `hd` *and* `tl`*.*

```
-- | 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
```

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.

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

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


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

Hint: When you are done, `okRemove` should be accepted, `badRemove` should be rejected.

Answer

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

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

{-@ 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 3.8 (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 (e cons b)

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

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!