RANJIT JHALA, ERIC SEIDEL, NIKI VAZOU

PROGRAMMING WITH REFINEMENT TYPES

AN INTRODUCTION TO LIQUIDHASKELL

Contents

1 Introduction 9

Well-Typed Programs Do Go Wrong 9

Refinement Types 11

Audience 11

Getting Started 11

Sample Code 12

2 Refinement Types 13

Defining Types 13

Errors 14

Subtyping 14

Writing Specifications 15

Refining Function Types: Pre-conditions 16

Refining Function Types: Post-conditions 17

Testing Values: Booleans and Propositions 17

Putting It All Together 19

Recap 19

3 *Polymorphism* 21

Specification: Vector Bounds 22

Verification: Vector Lookup 23

Inference: Our First Recursive Function 24

Higher-Order Functions: Bottling Recursion in a `loop` 25

Refinements and Polymorphism 27

Recap 28

4 *Refined Datatypes* 29

Sparse Vectors Revisited 29

Ordered Lists 31

Ordered Trees 34

Recap 37

5 *Boolean Measures* 39

Partial Functions 39

Lifting Functions to Measures 40

A Safe List API 42

Recap 45

6 *Numeric Measures* 47

Wholemeal Programming 47

Specifying List Dimensions 49

Lists: Size Preserving API 50

Lists: Size Reducing API 52

Dimension Safe Vector API 53

Dimension Safe Matrix API 55

Recap 57

7 *Elemental Measures* 59

Talking about Sets 59

Proving QuickCheck Style Properties 60

Content-Aware List API 62

Permutations 64

Uniqueness 65

Unique Zippers 67

Recap 69

8 *Case Study: Associative Maps* 71

Specifying Maps 71

Using Maps: Well Scoped Expressions 72

Implementing Maps: Binary Search Trees 76

Recap 80

9 *Case Study: Pointers and ByteStrings* 81

HeartBleeds in Haskell 81

Low-level Pointer API 82

A Refined Pointer API 84

Assumptions vs Guarantees 87

ByteString API 87

Application API 91

Nested ByteStrings 92

Recap: Types Against Overflows 94

List of Exercises

2.1 Exercise (List Average) 17
2.2 Exercise (Propositions) 18
2.3 Exercise (Assertions) 18

3.1 Exercise (Vector Head) 24
3.2 Exercise (Unsafe Lookup) 24
3.3 Exercise (Safe Lookup) 24
3.4 Exercise (Guards) 25
3.5 Exercise (Absolute Sum) 25
3.6 Exercise (Off by one?) 25
3.7 Exercise (Using Higher-Order Loops) 26
3.8 Exercise (Dot Product) 26

1

Introduction

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, if we call it with an empty list, we get a rather unpleasant crash:

```
ghci> average []
*** Exception: divide by zero
```

[0] We might solve this problem by writing `average` more *defensively*, perhaps 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.

Missing Keys Associative key-value maps are the new lists; they come "built-in" with modern languages like Go, Python, JavaScript and Lua; and of course, they're widely used in Haskell too.

```
ghci> :m +Data.Map
ghci> let m = fromList [ ("haskell", "lazy")
                       , ("ocaml"  , "eager")]

ghci> m ! "haskell"
"lazy"
```

Alas, maps are another source of vexing errors that are tickled when we try to find the value of an absent key:

```
ghci> m ! "javascript"
"*** Exception: key is not in the map
```

[o] Again, one could use a `Maybe` but its just deferring the inevitable.

SEGMENTATION FAULTS Say what? How can one possibly get a segmentation fault with a *safe* language like Haskell. Well, here's the thing: every safe language is built on a foundation of machine code, or at the very least, `C`. Consider the ubiquitous `vector` library:

```
ghci> :m +Data.Vector
ghci> let v = fromList ["haskell", "ocaml"]
ghci> unsafeIndex v 0
"haskell"
```

However, invalid inputs at the safe upper levels can percolate all the way down and stir a mutiny down below:

```
ghci> unsafeIndex v 3
'ghci' terminated by signal SIGSEGV ...
```

[o] Why use a function marked `unsafe`? Because it's very fast! Furthermore, even if we used the safe variant, we'd get a *run-time* exception which is only marginally better. Finally, we should remember to thank the developers for carefully marking it unsafe, because in general, given the many layers of abstraction, it is hard to know which functions are indeed safe.

HEART BLEEDS Finally, 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.

[0] If you are familiar with the notion of Dependent Types, for example, as in the Coq proof assistant, then Refinement Types can be thought of as restricted class of the former where the logic is restricted, at the cost of expressiveness, but with the reward of a considerable amount of automation.

Audience

Do you

- know a bit of basic arithmetic and logic?
- know the difference between a `nand` and an `xor`?
- know any typed languages e.g. ML, Haskell, Scala, F# or (Typed) Racket?
- know what `forall a. a -> a` means?
- like it when your code editor politely points out infinite loops?
- like your programs to not have bugs?

  Then this tutorial is for you!

Getting Started

First things first; lets see how to install and run LiquidHaskell.

LIQUIDHASKELL REQUIRES in addition to the cabal dependencies the binary executable for an `SMTLIB2` compatible solver, e.g. one of

- Z3
- CVC4
- MathSat

TO INSTALL LiquidHaskell, just do:

```
$ cabal install liquidhaskell
```

COMMAND LINE execution simply requires you type:

```
$ liquid /path/to/file.hs
```

You will see a report of `SAFE` or `UNSAFE` together with type errors at various points in the source.

EMACS AND VIM have LiquidHaskell plugins, which run `liquid` in the background as you edit any Haskell file, highlight errors, and display the inferred types, all of which we find to be extremely useful. Hence we *strongly recommend* these over the command line option.

- Emacs' `flycheck` plugin is described here
- Vim's `syntastic` checker is described here

Sample Code

This tutorial is written in literate Haskell and the code for it is available here. We *strongly* recommend you grab the code, and follow along, and especially that you do the exercises.

2
Refinement Types

WHAT IS A REFINEMENT TYPE? In a nutshell,

$$\text{Refinement Types} = \text{Types} + \text{Predicates}$$

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 USE these types we can write:

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

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

[0] We will use `@`-marked comments to write refinement type annotations the Haskell source file, making these types, quite literally, machine-checked comments!

Errors

If we try to say nonsensical things like:

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

LiquidHaskell will complain with an error message:

```
02-basic.lhs:58:8: Error: Liquid Type Mismatch
   Inferred type
     VV : Int | VV == (1  :  int)

   not a subtype of Required type
     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 Even  = {v:Int | v mod 2 == 0 } @-}
{-@ type Lt100 = {v:Int | v < 100}       @-}
```

What is the type of `zero`? `Zero` of course, but also `Nat`:

```
{-@ zero' :: Nat @-}
zero'     = zero
```

and also `Even`:

```
{-@ zero'' :: Even @-}
zero''     = zero
```

and also any other satisfactory refinement, such as:

```
{-@ zero''' :: Lt100  @-}
zero'''     = zero
```

[0] We use a different names ‘zero’‘, ‘zero’’‘ etc. as (currently) LiquidHaskell supports *at most* one refinement type for each top-level name.

Subtyping and Implication `Zero` is the most precise type for `0::Int`, as it is *subtype* of `Nat`, `Even` and `Lt100`. This is because the set of values defined by `Zero` is a *subset* of the values defined by `Nat`, `Even` and `Lt100`, as the following *logical implications* are valid:

- $v = 0 \Rightarrow 0 \leq v$
- $v = 0 \Rightarrow v \bmod 2 = 0$
- $v = 0 \Rightarrow v < 100$

Composing Refinements If $P \Rightarrow Q$ and $P \Rightarrow R$ then $P \Rightarrow Q \wedge R$. Thus, when a term satisfies multiple refinements, we can compose those refinements with `&&`:

\begin{comment} ES: this is confusingly worded \end{commment}

```
{-@ zero'''' :: {v:Int | 0 <= v && v mod 2 == 0 && v < 100} @-}
zero''''     = 0
```

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.

[0] Dually, a standard Haskell type, has the trivial refinement `true`. For example, `Int` is equivalent to `{v:Int|true}`.

Writing Specifications

Let’s write some more interesting specifications.

Typing Dead Code We can wrap the usual `error` function in a function `die` with the type:

```
{-@ die :: {v:String | false} -> a  @-}
die msg = error msg
```

The interesting thing about `die` is that the input type has the refinement `false`, meaning the function must only be called with `String`s that satisfy the predicate `false`. This seems bizarre; isn’t it *impossible* to satisfy `false`? Indeed! Thus, a program containing `die` typechecks *only* when LiquidHaskell can prove that `die` is *never called*. For example, LiquidHaskell will *accept*

```
cantDie = if 1 + 1 == 3
            then die "horrible death"
            else ()
```

by inferring that the branch condition is always `False` and so `die` cannot be called. However, LiquidHaskell will *reject*

```
canDie = if 1 + 1 == 2
           then die "horrible death"
           else ()
```

as the branch may (will!) be `True` and so `die` can be called.

Refining Function Types: Pre-conditions

Let's use `die` to write a *safe division* function that *only accepts* non-zero denominators.

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

From the above, it is clear to *us* that `div` is only called with non-zero divisors. However, LiquidHaskell reports an error at the call to `"die"` because, what if `divide'` is actually invoked with a `0` divisor?

We can specify that will not happen, 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
```

TO VERIFY that `divide` never calls `die`, LiquidHaskell infers that `"divide by zero"` is not merely of type `String`, but in fact has the the refined type `{v:String | false}` *in the context* in which the call to `die'` occurs. LiquidHaskell arrives at this conclusion by using the fact that in the first equation for `divide` the *denominator* is in fact

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

which *contradicts* the pre-condition (i.e. input) type. Thus, by contradition, LiquidHaskell deduces that the first equation is *dead code* and hence `die` will not be called at run-time.

ESTABLISHING PRE-CONDITIONS The above signature forces us to ensure that that when we *use* `divide`, we only supply provably `NonZero` arguments. Hence, these two uses of `divide` are fine:

```
avg2 x y   = divide (x + y) 2
avg3 x y z = divide (x + y + z) 3
```

Exercise 2.1 (List Average)**.** *Consider the function* avg*:*

1. *Why does LiquidHaskell flag an error at* n *?*
2. *How can you change the code so LiquidHaskell verifies it?*

```
avg       :: [Int] -> Int
avg xs    = divide total n
  where
    total = sum xs
    n     = length xs
```

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.

[9] LiquidHaskell is able to automatically make these arithmetic deductions by using an SMT solver which has built-in decision procedures for arithmetic, to reason about the logical refinements.

Testing Values: Booleans and Propositions

In the above example, we *compute* a value that is guaranteed to be a Nat. Sometimes, we need to *test* if a value satisfies some property, e.g., is NonZero. For example, let's write a command-line *calculator*:

```
calc = do putStrLn "Enter numerator"
          n <- readLn
          putStrLn "Enter denominator"
          d <- readLn
          putStrLn (result n d)
          calc
```

which takes two numbers and divides them. The function `result` checks if `d` is strictly positive (and hence, non-zero), and does the division, or otherwise complains to the user:

```
result n d
  | isPositive d = "Result = " ++ show (n `divide` d)
  | otherwise    = "Humph, please enter positive denominator!"
```

Finally, `isPositive` is a test that returns a `True` if its input is strictly greater than `0` or `False` otherwise:

```
isPositive :: Int -> Bool
isPositive x = x > 0
```

To VERIFY the call to `divide` inside `result` we need to tell LiquidHaskell that the division only happens with a `NonZero` value `d`. However, the non-zero-ness is established via the *test* that occurs inside the guard `isPositive d`. Hence, we require a *post-condition* that states that `isPositive` only returns `True` when the argument is positive:

```
{-@ isPositive :: x:Int -> {v:Bool | Prop v <=> x > 0} @-}
```

In the above signature, read `Prop v` as "`v` is `True`"; dually, read `not (Prop v)` as "`v` is `False`". Hence, the output type (post-condition) states that `isPositive x` returns `True` if and only if `x` was in fact strictly greater than `0`. In other words, we can write post-conditions for plain-old `Bool`-valued *tests* to establish that user-supplied values satisfy some desirable property (here, `Pos` and hence `NonZero`) in order to then safely perform some computation on it.

Exercise 2.2 (Propositions)**.** *What happens if you* delete *the type for* `isPositive` *? Can you* change *the type for* `isPositive` *(i.e. write some other type) to while preserving safety?*

Exercise 2.3 (Assertions)**.** *Consider the following* assert *function, and two use sites. Write a suitable refinement type signature for* `lAssert` *so that* `lAssert` *and* `yes` *are accepted but* `no` *is rejected.*

```
{-@ lAssert  :: Bool -> a -> a @-}
lAssert True  x = x
lAssert False _ = die "yikes, assertion fails!"

yes = lAssert (1 + 1 == 2) ()
no  = lAssert (1 + 1 == 3) ()
```

Hint: You need a pre-condition that `lAssert` is only called with `True`.

Putting It All Together

Let's wrap up this introduction with a simple `truncate` function that connects all the dots.

```
truncate :: Int -> Int -> Int
truncate i max
  | i' <= max' = i
  | otherwise  = max' * (i `divide` i')
    where
      i'       = abs i
      max'     = abs max
```

The expression `truncate i n` evaluates to `i` when the absolute value of `i` is less the upper bound `max`, and otherwise *truncates* the value at the maximum `n`. LiquidHaskell verifies that the use of `divide` is safe by inferring that:

1. `max' < i'` from the branch condition,
2. `0 <= i'` from the `abs` post-condition, and
3. `0 <= max'` from the `abs` post-condition.

From the above, LiquidHaskell infers that `i' /= 0`. That is, at the call site `i' :: NonZero`, thereby satisfying the pre-condition for `divide` and verifying that the program has no pesky divide-by-zero errors.

Recap

This concludes our quick introduction to Refinement Types and LiquidHaskell. Hopefully you have some sense of how to

1. **Specify** fine-grained properties of values by decorating their types with logical predicates.
2. **Encode** assertions, pre-conditions, and post-conditions with suitable function types.
3. **Verify** semantic properties of code by using automatic logic engines (SMT solvers) to track and establish the key relationships between program values.

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 and concrete 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 and then attempt to look it up at various indices:

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

```
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)
```

In a suitable Editor e.g. Vim or Emacs, you will you will literally see the error *without* running the code. Next, let's see how LiquidHaskell checks `ok` and `yup` but flags `nono`, and along the way, learn how LiquidHaskell reasons about *recursion*, *higher-order functions*, *data types*, and *polymorphism*.

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. For example, we can write specifications for `Data.Vector` inside `include/Data/Vector.spec` which contains:

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

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

-- | Restrict the indices
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 betwen `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 is essential for writing properties that connect different program values.

ALIASES are extremely useful for defining *abbreviations* for commonly occuring 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 `Vector`s 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 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)
```

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 `Vector`s

```
{-@ 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 `Vector`s *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
```

Exercise 3.4 (Guards). *What happens if you* replace *the guard with* `i <= sz`*?*

Exercise 3.5 (Absolute Sum). *Write a variant of the above function that computes the* `absoluteSum` *of the elements of the vector.*

```
-- >>> absoluteSum (fromList [1, -2, 3])
-- 6
{-@ absoluteSum :: Vector Int -> Nat @-}
absoluteSum     = undefined
```

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*

[0] In your editor, click on go to see the inferred type.

```
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.

Exercise 3.6 (Off by one?). *Why does the type of* `go` *have* `v <= sz` *and not* `v < sz` *?*

Higher-Order Functions: Bottling Recursion in a `loop`

Let's refactor the above low-level recursive function into a generic higher-order `loop`.

```
loop :: Int -> Int -> a -> (Int -> a -> a) -> a
loop lo hi base f =  go base lo
  where
    go acc i
      | i < hi    = go (f i acc) (i + 1)
      | otherwise = acc
```

We can now use `loop` to implement `vectorSum`:

```
vectorSum'      :: Vector Int -> Int
vectorSum' vec  = loop 0 n 0 body
  where
    body i acc  = acc + (vec ! i)
    n           = length vec
```

INFERENCE is a convenient option. LiquidHaskell finds:

```
loop :: lo:Nat -> hi:{Nat|lo <= hi} -> a -> (Btwn lo hi -> a -> a) -> a
```

In english, the above type states that

- `lo` the loop *lower* bound is a non-negative integer
- `hi` the loop *upper* bound is a greater than `lo`,
- `f` the loop *body* is only called with integers between `lo` and `hi`.

It can be tedious to have to keep typing things like the above. If we wanted to make `loop` a public or exported function, we could use the inferred type to generate an explicit signature.

At the call `loop 0 n 0 body` the parameters `lo` and `hi` are instantiated with `0` and `n` respectively, which, by the way is where the inference engine deduces non-negativity. Thus LiquidHaskell concludes that `body` is only called with values of `i` that are *between* `0` and `(vlen vec)`, which verifies the safety of the call `vec ! i`.

Exercise 3.7 (Using Higher-Order Loops)**.** *Complete the implementation of* `absoluteSum'` *below. When you are done, what is the type that is inferred for* `body`*?*

```
-- >>> absoluteSum' (fromList [1, -2, 3])
-- 6
{-@ absoluteSum' :: Vector Int -> Nat @-}
absoluteSum' vec = loop 0 n 0 body
  where
    n             = length vec
    body i acc    = undefined
```

Exercise 3.8 (Dot Product)**.** *The following function uses* `loop` *to compute* `dotProducts`*. Why does LiquidHaskell flag an error? Fix the code or specification so that LiquidHaskell accepts it.*

```
-- >>> dotProduct (fromList [1,2,3]) (fromList [4,5,6])
-- 32
{-@ dotProduct :: x:Vector Int -> y:Vector Int -> Int @-}
dotProduct x y = loop 0 sz 0 body
  where
    sz         = length x
    body i acc = acc + (x ! i)  *  (y ! i)
```

Refinements and Polymorphism

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:

```
{-@ type SparseN a N = [(Btwn 0 N, a)] @-}
```

Implicitly, all indices *other* than those in the list have the value `0` (or the equivalent value for the type `a`).

ALIAS `SparseN` is just a shorthand for the (longer) type on the right, it does not *define* a new type. If you are familiar with the *index-style* length encoding e.g. as found in DML or Agda, then note that despite appearances, our `Sparse` definition is *not* indexed.

SPARSE PRODUCTS Let's write a function to compute a sparse product

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

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.

FOLDS The sharp reader will have undoubtedly noticed that the sparse product can be more cleanly expressed as a fold:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
```

We can simply fold over the sparse vector, accumulating the `sum` as we go along

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

LiquidHaskell digests this without difficulty. The main trick is in how the polymorphism of `foldl'` is instantiated.

1. GHC infers that at this site, the type variable `b` from the signature of `foldl'` is instantiated to the Haskell type `(Int, a)`.

2. Correspondingly, LiquidHaskell infers that in fact `b` can be instantiated to the *refined* `(Btwn 0 v (vlen x), a)`.

Thus, the inference mechanism saves us a fair bit of typing and allows us to reuse existing polymorphic functions over containers and such without ceremony.

Recap

This chapter gave you an idea of how one can use refinements to verify size related properties, and more generally, to specify and verify properties of recursive and polymorphic functions. Next, let's see how we can use LiquidHaskell to prevent the creation of illegal values by refining data type definitions.

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 postconditions 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 the `data` definition, making it impossible to create illegal inhabitants.

Sparse Vectors Revisited

As our first example of a refined datatype, let's revisit the sparse vector representation that we saw earlier. The `SparseN` type alias we used got the job done, but is not pleasant to work with because we have no way of determining the *dimension* of the sparse vector. Instead, 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*.

[0] 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, the 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
```

{#autosmart} 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:

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

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.

Exercise 4.1. **[Sanitization]** Invariants are all well and good for data computed *inside* our programs. The only way to ensure the legality of data coming from *outside*, i.e. from the "real world", is to writing a sanitizer that will check the appropriate invariants before constructing a `Sparse` vector. Write the specification and implementation of a sanitizer `fromList`, so that the following typechecks:

```
fromList          :: Int   -> [(Int, a)] -> Maybe (Sparse a)
fromList dim elts = undefined

{-@ test1         :: SparseN String 3 @-}
test1             = fromJust $ fromList 3 [(0, "cat"), (2, "mouse")]
```

Exercise 4.2. **[Addition]** Write the specification and implementation of a function `plus` that performs the addition of two `Sparse` vectors of the *same* dimension, yielding an output of that dimension. When you are done, the following code should typecheck:

```
plus     :: (Num a) => Sparse a -> Sparse a -> Sparse a
plus x y = undefined

{-@ test2 :: SparseN Int 3 @-}
test2    = plus vec1 vec2
  where
    vec1 = SP 3 [(0, 12), (2, 9)]
    vec2 = SP 3 [(0, 8),  (1, 100)]
```

Ordered Lists

As a second example of refined data types, let's consider a different problem: representing *ordered* sequences. Here's a type for sequences that mimics the classical list:

```
data IncList a = Emp
               | (:<) { hd :: a, tl :: IncList a }

infixr 9 :<
```

The Haskell type above does not state that the elements be in order of course, but we can specify that requirement by refining *every* element in `tl` to be *greater than* `hd`:

```
{-@ data IncList a = Emp
                   | (:<) { hd :: a, tl :: IncList {v:a | hd <= v} }
  @-}
```

Refined Data Constructors Once again, the refined data definition is internally converted into a "smart" refined data constructor

```
-- Generated Internal representation
data IncList a where
  Emp  :: IncList a
  (:<) :: hd:a -> tl:IncList {v:a | hd <= v} -> IncList a
```

which ensures that we can only create legal ordered lists.

```
okList  = 1 :< 2 :< 3 :< Emp      -- accepted by LH

badList = 2 :< 1 :< 3 :< Emp      -- rejected by LH
```

Its all very well to *specify* ordered lists. Next, lets see how its equally easy to *establish* these invariants by implementing several textbook sorting routines.

Insertion Sort First, lets implement insertion sort, which converts an ordinary list `[a]` into an ordered list `IncList a`.

```
insertSort        :: (Ord a) => [a] -> IncList a
insertSort []     = Emp
insertSort (x:xs) = insert x (insertSort xs)
```

The hard work is done by `insert` which places an element into the correct position of a sorted list. LiquidHaskell infers that if you give `insert` an element and a sorted list, it returns a sorted list.

```
insert             :: (Ord a) => a -> IncList a -> IncList a
insert y Emp       = y :< Emp
insert y (x :< xs)
  | y <= x         = y :< x :< xs
  | otherwise      = x :< insert y xs
```

EXERCISE 4.3. Complete the implementation of the function below to use `foldr` to eliminate the explicit recursion in `insertSort`.

```
insertSort'     :: (Ord a) => [a] -> IncList a
insertSort' xs  = foldr f b xs
  where
     f          = undefined    -- Fill this in
     b          = undefined    -- Fill this in
```

MERGE SORT Similarly, it is easy to write merge sort, by implementing the three steps. First, we write a function that *splits* the input into two equal sized halves:

```
split          :: [a] -> ([a], [a])
split (x:y:zs) = (x:xs, y:ys)
  where
    (xs, ys)   = split zs
split xs       = (xs, [])
```

Second, we need a function that *combines* two ordered lists

```
merge         :: (Ord a) => IncList a -> IncList a -> IncList a
merge xs  Emp = xs
merge Emp ys  = ys
merge (x :< xs) (y :< ys)
  | x <= y    = x :< merge xs (y :< ys)
  | otherwise = y :< merge (x :< xs) ys
```

Finally, we compose the above steps to divide (i.e. `split`) and conquer (`sort` and `merge`) the input list:

```
{-@ mergeSort :: (Ord a) => [a] -> IncList a @-}
mergeSort []  = Emp
mergeSort [x] = x :< Emp
mergeSort xs  = merge (mergeSort ys) (mergeSort zs)
  where
    (ys, zs)  = split xs
```

EXERCISE 4.4. Why is the following implementation of `quickSort` rejected by LiquidHaskell? Modify it so it is accepted.

```
quickSort           :: (Ord a) => [a] -> IncList a
quickSort []        = Emp
quickSort (x:xs)    = append lessers greaters
  where
```

```
    lessers          = quickSort [y | y <- xs, y < x ]
    greaters         = quickSort [z | z <- xs, z >= x]

append               :: (Ord a) => IncList a -> IncList a -> IncList a
append Emp        ys = ys
append (x :< xs) ys = x :< append xs ys
```

Ordered Trees

As a last example of refined data types, let us consider binary search ordered trees, defined thus:

```
data BST a = Leaf
           | Node { root  :: a
                  , left  :: BST a
                  , right :: BST a }
```

BINARY SEARCH TREES enjoy the property that each `root` lies (strictly) between the elements belonging in the `left` and `right` subtrees hanging off the the root. The ordering invariant makes it easy to check whether a certain value occurs in the tree. If the tree is empty i.e. a Leaf, then the value does not occur in the tree. If the given value is at the root then the value does occur in the tree. If it is less than (respectively greater than) the root, we recursively check whether the value occurs in the left (respectively right) subtree.

Figure 4.1 shows a binary search tree whose nodes are labeled with a subset of values from `1` to `9`. We might represent such a tree with the Haskell value:

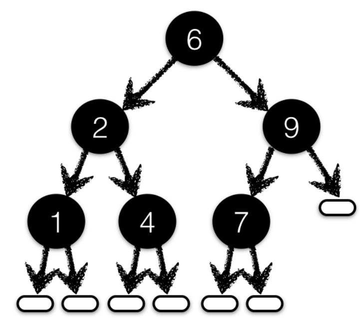

Figure 4.1: A Binary Search Tree with values between 1 and 9. Each root's value lies between the values appearing in its left and right subtrees.

```
okBST :: BST Int
okBST =  Node 6
             (Node 2
                 (Node 1 Leaf Leaf)
                 (Node 4 Leaf Leaf))
             (Node 9
                 (Node 7 Leaf Leaf)
                 Leaf)
```

REFINED DATA TYPE The Haskell type says nothing about the ordering invariant, and hence, cannot prevent us from creating illegal `BST` values that violate the invariant. We can remedy this with a refined data definition that captures the invariant:

```
{-@ data BST a = Leaf
               | Node { root  :: a
                      , left  :: BST {v:a | v < root}
                      , right :: BST {v:a | root < v} }
  @-}
```

Refined Data Constructors As before, the above data definition creates a refined "smart" constructor for `BST`

```
data BST a where
  Leaf :: BST a
  Node :: r:a -> BST {v:a | v < r} -> BST {v:a | r < v} -> BST a
```

which *prevents* us from creating illegal trees

```
badBST :: BST Int
badBST =  Node 6
             (Node 4
                 (Node 1 Leaf Leaf)
                 (Node 2 Leaf Leaf))    -- Out of order, rejected by LH
             (Node 9
                 (Node 7 Leaf Leaf)
                 Leaf)
```

Exercise 4.5. Can a `BST Int` contain duplicates?

Membership Lets write some functions to create and manipulate these trees. First, a function to check whether a value is in a `BST`:

```
mem                  :: (Ord a) => a -> BST a -> Bool
mem _ Leaf           = False
mem k (Node k' l r)
  | k == k'          = True
  | k <  k'          = mem k l
  | otherwise        = mem k r
```

Singleton Next, another easy warm-up: a function to create a `BST` with a single given element:

```
one   :: a -> BST a
one x = Node x Leaf Leaf
```

Insertion Next, lets write a function that adds an element to a `BST`.

[0] Amusingly, while typing out the below I swapped the `k` and `k'` which caused LiquidHaskell to complain.

```
add                  :: (Ord a) => a -> BST a -> BST a
add k' Leaf          = one k'
add k' t@(Node k l r)
  | k' < k           = Node k (add k' l) r
  | k  < k'          = Node k l (add k' r)
  | otherwise        = t
```

MINIMUM Next, lets write a function to delete the *minimum* element from a BST. This function will return a *pair* of outputs – the smallest element and the remainder of the tree. We can say that the output element is indeed the smallest, by saying that the remainder's elements exceed the element. To this end, lets define a helper type:

[0] This helper type approach is rather verbose. We should be able to just use plain old pairs and specify the above requirement as a *dependency* between the pairs' elements. Later, we will see how to do so using abstract refinements.

```
data MinPair a = MP { minElt :: a, rest :: BST a }
```

We can specify that `minElt` is indeed smaller than all the elements in `rest` via the data type refinement:

```
{-@ data MinPair a = MP { minElt :: a, rest :: BST {v:a | minElt < v} } @-}
```

Finally, we can write the code to compute `MinPair`

```
delMin                 :: (Ord a) => BST a -> MinPair a
delMin (Node k Leaf r) = MP k r
delMin (Node k l r)    = MP k' (Node k l' r)
  where
    MP k' l'           = delMin l
delMin Leaf            = die "Don't say I didn't say I didn't warn ya!"
```

EXERCISE 4.6. **[Deletion]** Use `delMin` to complete the implementation of `del` which *deletes* a given element from a `BST`, if it is present.

```
del                   :: (Ord a) => a -> BST a -> BST a
del k' t@(Node k l r) = undefined
del _  t              = t
```

EXERCISE 4.7. The function `delMin` is only sensible for non-empty trees. Read ahead to learn how to specify and verify that it is only called with such trees, and then apply that technique here to verify the call to `die` in `delMin`.

EXERCISE 4.8. Complete the implementation of `toIncList` to obtain a `BST` based sorting routine `bstSort`.

```
bstSort   :: (Ord a) => [a] -> IncList a
bstSort   = toIncList . toBST

toBST     :: (Ord a) => [a] -> BST a
toBST     = foldr add Leaf

toIncList :: BST a -> IncList a
toIncList = undefined
```

Hint: This exercise will be a lot easier after you finish the `quickSort` exercise. Note that the signature for `toIncList` does not use `Ord` and so you cannot use a sorting procedure to implement it.

Recap

In this chapter we saw how LiquidHaskell lets you refine data type definitions to capture sophisticated invariants. These definitions are internally represented by refining the types of the data constructors, automatically making them "smart" in that they preclude the creation of illegal values that violate the invariants. We will see much more of this handy technique in future chapters.

One recurring theme in this chapter was that we had to create new versions of standard datatypes, just in order to specify certain invariants. For example, we had to write a special list type, with its own *copies* of nil and cons. Similarly, to implement `delMin` we had to create our own pair type.

THIS DUPLICATION of types is quite tedious. There should be a way to just slap the desired invariants on to *existing* types, thereby facilitating their reuse. In a few chapters, we will see how to achieve this reuse by *abstracting refinements* from the definitions of datatypes or functions in the same way we abstract the element type `a` from containers like `[a]` or `BST a`.

5
Boolean Measures

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, lets 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 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 is requirement is not onerous when we know exactly 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, lets 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, lets 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 To use the newly defined measure, we define an alias for non-empty lists, i.e. 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 First, we specify that `size` returns a non-zero value when the input list is not-empty:

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

Average Second, we specify that the `average` is only sensible for non-empty lists:

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

Exercise 5.1. Fix the code below to obtain an alternate variant `average'` that returns `Nothing` for empty lists:

```
average'      :: [Int] -> Maybe Int
average' xs
  | ok        = Just $ divide total elems
  | otherwise = Nothing
  where
    total     = sum  xs
    elems     = size xs
    ok        = True    -- What expression goes here?
```

Exercise 5.2. 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
```

***TODO* solution**

A Safe List API

Now that we can talk about non-empty lists, we can ensure the safety of various list-manipulating functions which are only well-defined on non-empty lists and which crash with unexpected run-time errors otherwise.

Heads and Tails For example, we can type the potentially dangerous `head` and `tail` as:

```
{-@ head    :: NEList a -> a @-}
head (x:_)  = x
head []     = die "Fear not! 'twill ne'er come to pass"

{-@ tail    :: NEList a -> [a] @-}
tail (_:xs) = xs
tail []     = die "Relaxeth! this too shall ne'er be"
```

LiquidHaskell deduces that the second equations are *dead code* thanks to the precondition, which ensures callers only supply non-empty arguments.

Exercise 5.3. Write down a specification for `null` such that `safeHead` is verified:

```
safeHead      :: [a] -> Maybe a
safeHead xs
  | null xs   = Nothing
  | otherwise = Just $ head xs

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

GROUPS Lets use the above to write a function that chunks sequences into non-empty groups of equal elements:

```
{-@ groupEq         :: (Eq a) => [a] -> [NEList a] @-}
groupEq []          = []
groupEq (x:xs)      = (x:ys) : groupEq zs
  where
    (ys, zs)        = span (x ==) xs
```

By using the fact that *each element* in the output returned by `groupEq` is in fact of the form `x:ys`, LiquidHaskell infers that `groupEq` returns a `[NEList a]` that is, a list of *non-empty lists*.

We can use `groupEq` to write a function that eliminates stuttering from a String:

```
-- >>> eliminateStutter "ssstringssss liiiiiike thisss"
-- "strings like this"
eliminateStutter xs = map head $ groupEq xs
```

LiquidHaskell automatically instantiates the type parameter for `map` in `eliminateStutter` to `notEmpty v` to deduce that `head` is only called on non-empty lists.

FOLDS One of my favorite folds is `foldr1` which uses the first element of the sequence as the initial value. Of course, it should only be called with non-empty sequences!

```
{-@ foldr1      :: (a -> a -> a) -> NEList a -> a @-}
foldr1 f (x:xs) = foldr f x xs
foldr1 _ []     = die "foldr1"

foldr                :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc []       = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

SUM Thanks to the precondition, LiquidHaskell will prove that the `die` code is indeed dead. Thus, we can write

```
{-@ sum :: (Num a) => NEList a -> a  @-}
sum []  = die "cannot add up empty list"
sum xs  = foldr1 (+) xs
```

Consequently, we can only invoke `sum` on non-empty lists, so:

```
sumOk  = sum [1,2,3,4,5]    -- accepted by LH

sumBad = sum []             -- rejected by LH
```

Exercise 5.4. The function below computes a weighted average of its input. Unfortunately, LiquidHaskell is not very happy about it. Can you figure out why, and fix the code or specification appropriately?

```
{-@ wtAverage :: NEList (Pos, Pos) -> Int @-}
wtAverage wxs = divide totElems totWeight
  where
    elems     = map (\(w, x) -> w * x) wxs
    weights   = map (\(w, _) -> w    ) wxs
    totElems  = sum elems
    totWeight = sum weights
    sum       = foldr1 (+)

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

Hint: On what variables are the errors? How are those variables' values computed? Can you think of a better specification for the function(s) doing those computations?

Exercise 5.5. Non-empty lists pop up in many places, and it is rather convenient to have the type system track non-emptiness without having to make up special types. Consider the risers function:

[0] Popularized by [Neil Mitchell](http://neilmitchell.blogspot.com/2008/03/sortin at-speed.html)

```
risers           :: (Ord a) => [a] -> [[a]]
risers []        = []
risers [x]       = [[x]]
risers (x:y:etc)
  | x <= y       = (x:s) : ss
  | otherwise    = [x] : (s : ss)
    where
      (s, ss)    = safeSplit $ risers (y:etc)

{-@ safeSplit    :: NEList a -> (a, [a]) @-}
safeSplit (x:xs) = (x, xs)
safeSplit _      = die "don't worry, be happy"
```

The call to safeSplit requires its input be non-empty, and LiquidHaskell does not believe that the call inside risers meets this requirement. Can you devise a specification for risers that allows LiquidHaskell to verify the call to safeSplit that risers will not die?

Recap

In this chapter we saw how LiquidHaskell lets you

1. Define *structural properties* of data types,

2. Use refinements over these properties to describe key invariants that establish, at compile-time, the safety of operations that might otherwise fail on unexpected values at run-time, all while,

3. Working with plain Haskell types, here, Lists, without having to make up new types which can have the unfortunate effect of adding a multitude of constructors and conversions which often clutter implementations and specifications.

Of course, We can do a lot more with measures, so lets press on!

6
Numeric Measures

Many of the programs we have seen so far, for example those in here, suffer from *indexitis*

[0] A term coined by Richard Bird

a tendency to perform low-level manipulations to iterate over the indices into a collection, which opens the door to various off-by-one errors. Such errors can be entirely 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`. However, wholemeal programming requires us to take care when operating on multiple collections; if these collections are *incompatible*, e.g. have the wrong dimensions, then we end up with a fate worse than a crash, a *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.

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

Wholemeal Programming

Indexitis begone! As an example of wholemeal programming, lets 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)
```

VECTOR PRODUCT We can write the dot product of two Vectors 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
```

MATRIX PRODUCT Similarly, we can compute the product of two matrices in a wholemeal fashion, without performing any low-level index manipulations, but instead using a high-level "iterator" over the elements of the matrix.

```
matProd       :: (Num a) => Matrix a -> Matrix a -> Matrix a
matProd (M rx _ xs) (M _ cy ys)
                 = M rx cy elts
  where
    elts         = for xs $ \xi ->
                     for ys $ \yj ->
                       dotProd xi yj
```

ITERATION In the above, the "iteration" embodied in for is simply a map over the elements of the vector.

```
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.

[0] 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.

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:

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

```
{-@ measure size @-}
{- size    :: xs:[a] -> {v:Nat | v = size xs && v = len xs} @-}
{-@ size    :: xs:[a] -> Nat @-}
size        :: [a] -> Int
size (_:rs) = 1 + size rs
size []     = 0
```

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

MULTIPLE MEASURES We can write several different measures for a datatype. 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
```

COMPOSING MEASURES LiquidHaskell lets you *compose* the 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}
  (:) :: x:a -> xs:[a] -> {v:[a] | notEmpty v && size v = 1 + size xs}
```

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

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

This is a very significant advantage of using measures instead of indices as in DML or Agda, as *decouples property from structure*, which crucially enables the use of the same structure for many different purposes. That is, we need not know *a priori* what indices to bake into the structure, but can define a generic structure and refine it *a posteriori* as needed with new measures.

Lets use `size` to create a dimension-aware API for lists. To get the ball rolling, lets defining a few helpful type aliases:

An 'N'-List is a list with exactly `N` elements:

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

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

To make the signatures symmetric, lets use an alias for plain old Lists:

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

Lists: Size Preserving API

With the types firmly in hand, 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.

'map' yields a list with the same size as the input:

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

zipWith requires both lists to have the *same* size, and produces a list with that same size.

º Note that as made explicit by the call to 'die', the input type *rules out* the case where one list is empty and the other is not, as in that case the former's length is zero while the latter's is not, and hence, different.

```
{-@ invariant {v:[a] | 0 <= size v} @-}

{-@ zipWith :: _ -> xs:List a -> ListN b (size xs) -> ListN c (size xs) @-}
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ [] []         = []
zipWith _ _  _          = die "no other cases"
```

unsafeZip The signature for `zipWith` is quite severe – it rules out the case where the zipping occurs only upto the shorter input. Here's a function that actually allows for that case, where the output type is the *shorter* of the two inputs:

```
{-@ zip :: as:[a] -> bs:[b] -> {v:[(a,b)] | Min (size v) (size as) (size bs)} @-}
zip (a:as) (b:bs) = (a, b) : zip as bs
zip [] _          = []
zip _  []         = []
```

The output type uses the following which defines X to be the smaller of Y and Z.

```
{-@ predicate Min X Y Z = (if X < Y then X = Y else X = Z) @-}
```

[0] Note that if p then q else r is simply an abbreviation for p => q && not p => r

Exercise 6.1. [Zip Unless Empty] In my experience, zip as shown above is far too permissive and lets all sorts of bugs into my code. As middle ground, consider zipOrNull below. Write a specification for zipOrNull such that the code below is verified by LiquidHaskell:

```
zipOrNull       :: [a] -> [b] -> [(a, b)]
zipOrNull [] _  = []
zipOrNull _ []  = []
zipOrNull xs ys = zipWith (,) xs ys

{-@ test1 :: {v: _ | size v = 2} @-}
test1     = zipOrNull [0, 1] [True, False]

{-@ test2 :: {v: _ | size v = 0} @-}
test2     = zipOrNull [] [True, False]

{-@ test3 :: {v: _ | size v = 0} @-}
test3     = zipOrNull ["cat", "dog"] []
```

Hint: Yes, the type is rather gross; it uses a bunch of disjunctions || , conjunctions && and implications =>.

Exercise 6.2. [Reverse] Consider the code below that reverses a list using the tail-recursive go. Fix the signature for go so that LiquidHaskell can prove the specification for reverse.

```
{-@ reverse       :: xs:[a] -> {v:[a] | size v = size xs} @-}
reverse xs        = go [] xs
  where
    {-@ go        :: xs:[a] -> ys:[a] -> [a] @-}
    go acc []     = acc
    go acc (x:xs) = go (x:acc) xs
```

Hint: How big is the list returned by go?

Lists: Size Reducing API

Next, lets look at some functions that truncate lists, in one way or another.

Take lets us grab the first `k` elements from a list:

```
{-@ take'     :: n:Nat -> {v:List a | n <= size v} -> ListN a n @-}
take' 0 _      = []
take' n (x:xs) = x : take' (n-1) xs
take' _ _      = die "won't  happen"
```

Exercise 6.3. [Drop] is the yang to `take`'s yin: it returns the remainder after extracting the first `k` elements. Write a suitable specification for it so that the below typechecks:

```
drop 0 xs     = xs
drop n (_:xs) = drop (n-1) xs
drop _ _      = die "won't happen"

{-@ test4 :: ListN String 2 @-}
test4 = drop 1 ["cat", "dog", "mouse"]
```

Exercise 6.4. [Take it easy] The version `take'` above is too restrictive; it insists that the list actually have at least `n` elements. Modify the signature for the *real* `take` function so that the code below is accepted by LiquidHaskell:

```
take 0 _       = []
take _ []      = []
take n (x:xs)  = x : take (n-1) xs

{-@ test5 :: [ListN String 2] @-}
test5 = [ take 2  ["cat", "dog", "mouse"]
        , take 20 ["cow", "goat"]        ]
```

Partition As one last example, lets look at the function that partitions a list using a user supplied predicate:

```
partition _ []     = ([], [])
partition f (x:xs)
  | f x            = (x:ys, zs)
  | otherwise      = (ys, x:zs)
  where
    (ys, zs)       = partition f xs
```

We would like to specify that the *sum* of the output tuple's dimensions equal the input list's dimension. Lets write measures to access the elements of the output:

```
{-@ measure first @-}
first  (x, _) = x

{-@ measure second @-}
second (_, y) = y
```

We can use the above to type `partition` as

```
{-@ partition :: (a -> Bool) -> xs:_ -> ListPair a (size xs) @-}
```

using an alias for a pair of lists whose total dimension equals `N`

```
{-@ type ListPair a N = {v:([a], [a]) | size (first v) + size (second v) = N} @-}
```

EXERCISE 6.5. [QUICKSORT] Use the `partition` function above to implement `quickSort`:

```
-- >> quickSort [1,4,3,2]
-- [1,2,3,4]

{-@ quickSort    :: (Ord a) => xs:List a -> ListN a (size xs) @-}
quickSort []     = []
quickSort (x:xs) = undefined

{-@ test10 :: ListN String 2 @-}
test10 = quickSort test4
```

Dimension Safe Vector API

We can use the dimension aware lists to create a safe vector API.

LEGAL VECTORS are those whose `vDim` field actually equals the size of the underlying list:

```
{-@ data Vector a = V { vDim  :: Nat
                      , vElts :: ListN a vDim }
  @-}
```

The refined data type prevents the creation of illegal vectors:

```
okVec  = V 2 [10, 20]       -- accepted by LH

badVec = V 2 [10, 20, 30]   -- rejected by LH
```

Access Next, lets write some functions to access the elements of a vector:

```
{-@ vCons        :: a -> x:Vector a -> {v:Vector a | vDim v = vDim x + 1} @-}
vCons x (V n xs) = V (n+1) (x:xs)

{-@ type VectorNE a = {v:Vector a | vDim v > 0} @-}

{-@ vHd :: VectorNE a -> a @-}
vHd (V _ (x:_)) = x
vHd _           = die "nope"

{-@ vTl          :: x:VectorNE a -> {v:Vector a | vDim v = vDim x - 1} @-}
vTl (V n (_:xs)) = V (n-1) xs
vTl _           = die "nope"
```

Iteration It is straightforward to see that:

```
{-@ for :: x:Vector a -> (a -> b) -> VectorN b (vDim x) @-}
```

Binary Operations We want to apply various binary operations to *compatible* vectors, i.e. vectors with equal dimensions. To this end, it is handy to have an alias for vectors of a given size:

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

We can now write a generic binary operator:

```
{-@ vBin :: (a -> b -> c) -> vx:Vector a -> vy:VectorN b (vDim vx) -> VectorN c (vDim vx) @-}
vBin     :: (a -> b -> c) -> Vector a -> Vector b -> Vector c
vBin op (V n xs) (V _ ys) = V n (zipWith op xs ys)
```

Dot Product Finally, we can implement a wholemeal, dimension safe dot product operator as:

```
{-@ dotProduct :: (Num a) => x:Vector a -> y:VectorN a (vDim x) -> a @-}
dotProduct x y = sum $ vElts $ vBin (*) x y
```

Exercise 6.6. [Vector Constructor] Complete the *specification* and *implementation* of `vecFromList` which *creates* a `Vector` from a plain old list.

```
vecFromList     :: [a] -> Vector a
vecFromList xs  = undefined

test6  = dotProduct vx vy    -- should be accepted by LH
  where
    vx = vecFromList [1,2,3]
    vy = vecFromList [4,5,6]
```

Dimension Safe Matrix API

The same methods let us create a dimension safe Matrix API which ensures that only legal matrices are created and that operations are performed on compatible matrices.

LEGAL MATRICES are those where the dimension of the outer vector equals the number of rows `mRow` and the dimension of each inner vector is `mCol`. We can specify legality in a refined data definition:

```
{-@ data Matrix a = M { mRow  :: Pos
                      , mCol  :: Pos
                      , mElts :: VectorN (VectorN a mCol) mRow
                      }
  @-}
```

Notice that we avoid disallow degenerate matrices by requiring the dimensions to be positive.

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

It is convenient to have an alias for matrices of a given size:

```
{-@ type MatrixN a R C = {v:Matrix a | mRow v = R && mCol v = C} @-}
```

after LiquidHaskell accepts:

```
ok23      = M 2 3 (V 2 [ V 3 [1, 2, 3]
                       , V 3 [4, 5, 6] ])
```

EXERCISE 6.7. [LEGAL MATRIX] Modify the definitions of `bad1` and `bad2` so that they are legal matrices accepted by LiquidHaskell.

```
bad1 :: Matrix Int
bad1 = M 2 3 (V 2 [ V 3 [1, 2    ]
                  , V 3 [4, 5, 6]])
```

```
bad2 :: Matrix Int
bad2 = M 2 3 (V 2 [ V 2 [1, 2]
                  , V 2 [4, 5] ])
```

Exercise 6.8. [Matrix Constructor] ⋆ Write a function to construct a `Matrix` from a nested list.

```
matFromList      :: [[a]] -> Maybe (Matrix a)
matFromList []   = Nothing                       -- no meaningful dimensions!
matFromList xss@(xs:_)
  | ok           = Just (M r c vs)
  | otherwise    = Nothing
  where
    r            = size xss
    c            = size xs
    ok           = undefined
    vs           = undefined
```

Exercise 6.9. [Refined Matrix Constructor] ⋆⋆ Refine the specification for `matFromList` so that the following is accepted by LiquidHaskell:

```
{-@ mat23 :: Maybe (MatrixN Integer 2 2) @-}
mat23     = matFromList [ [1, 2]
                        , [3, 4] ]
```

Hint: It is easy to specify the number of rows from `xss`. How will you figure out the number of columns? A measure may be useful.

Matrix Multiplication Ok, lets now implement matrix multiplication. You'd think we did it already, but in fact the implementation at the top of this chapter is all wrong. Indeed, you cannot just multiply any two matrices: the number of *columns* of the first must equal to the *rows* of the second – after which point the result comprises the `dotProduct` of the rows of the first matrix with the columns of the second.

[0] You could run it of course, or you could just replace 'dotProd' with our type-safe 'dotProduct' and see what happens!

```
{-@ matProduct   :: (Num a) => x:Matrix a
                            -> y:{Matrix a  | mCol x = mRow y}
                            -> MatrixN a (mRow x) (mCol y)
  @-}
matProduct (M rx _ xs) my@(M _ cy _)
                 = M rx cy elts
```

```
  where
    elts         = for xs $ \xi ->
                     for ys' $ \yj ->
                       dotProduct xi yj
    M _ _ ys'    = transpose my
```

Transposition To iterate over the columns of `my` we just `transpose` it so the columns become rows.

```
-- >>> ok32 == transpose ok23
-- True
ok32 = M 3 2 (V 3 [ V 2 [1, 4]
                  , V 2 [2, 5]
                  , V 2 [3, 6] ])
```

Exercise 6.10. [Matrix Transposition] ⋆⋆ Use the `Vector` API to Complete the implementation of `txgo`. For inspiration, you might look at the implementation of `Data.List.transpose` from the prelude. Better still, don't.

```
{-@ transpose          :: m:Matrix a -> MatrixN a (mCol m) (mRow m) @-}
transpose (M r c rows) = M c r $ txgo c r rows

{-@ txgo      :: c:Nat -> r:Nat
              -> VectorN (VectorN a c) r
              -> VectorN (VectorN a r) c @-}
txgo c r rows = undefined
```

Hint: As shown by `ok23` and `ok32`, `transpose` works by stripping out the `heads` of the input rows, to create the corresponding output rows.

Recap

In this chapter, we saw how to use measures to describe numeric properties of structures like lists (`Vector`) and nested lists (`Matrix`). To recap:

1. Measures are *structurally recursive* functions, with a single equation per data constructor,
2. Measures can be used to create refined data definitions that prevent the creation of illegal values,
3. Measures can then be used to enable safe wholemeal programming, via dimension-aware APIs that ensure that operators only apply to compatible values.

We can use numeric measures to encode various other properties of structures; in subsequent chapters we will see examples ranging from high-level height-balanced trees, to low-level safe pointer arithmetic.

7
Elemental Measures

Often, correctness requires us to reason about the *set of elements* represented inside a data structure, or manipulated by a function.

Sets appear everywhere. For example, we'd like to know that:

- *sorting* routines return permutations of their inputs – i.e. return collections whose elements are the same as the input' set,
- *resource management* functions do not inadvertently create duplicate elements or drop elements from set of tracked resources.
- *syntax-tree* manipulating procedures create well-scoped trees where (the set of) used variables are (contained within the set of variables) previously defined.

SMT Solvers support rather expressive logics. In addition to the linear arithmetic and uninterpreted functions, they can efficiently decide formulas over sets. Next, lets see how LiquidHaskell lets us exploit this fact to develop types and interfaces that guarantee invariants over the (set of) elements of a structures.

Talking about Sets

First, we need a way to talk about sets in the refinement logic. We could roll our own special Haskell type , but for now, lets just use the `Set a` type from the prelude's `Data.Set`.

[0] See [this](http://goto.ucsd.edu/ rjhala/liquid/haskell/blog/b about-sets.lhs/) for a brief description of how to do so

Lifted Operators The LiquidHaskell prelude *lifts* the basic set operators from `Data.Set` into the refinement logic, i.e. defines the following logical functions that correspond to the Haskell functions of the same name:

```
measure empty        :: Set a
measure singleton    :: a -> Set a
```

```
measure member       :: a -> Set a -> Bool
measure union        :: Set a -> Set a -> Set a
measure intersection :: Set a -> Set a -> Set a
measure difference   :: Set a -> Set a -> Set a
```

INTERPRETED OPERATORS The above operators are *interpreted* by the SMT solver. That is, just like the SMT solver "knows", via the axioms of the theory of arithmetic that:

$$x = 2 + 2 \Rightarrow x = 4$$

is a valid formula, i.e. holds for all x, the solver "knows" that:

$$x = (\texttt{singleton}\ 1) \Rightarrow y = (\texttt{singleton}\ 2) \Rightarrow x = (\texttt{intersection}\ x\ (\texttt{union}\ y\ x))$$

This is because, the above formulas belong to a decidable Theory of Sets reduces to McCarthy's more general Theory of Arrays.

[0] See [this recent paper](http://research.microsoft.com/en-us/um/people/leonardo/fmcad09.pdf) to learn how modern SMT solvers prove equalities like the above.

Proving QuickCheck Style Properties

To get the hang of whats going on, lets do a few warmup exercises, using LiquidHaskell to prove various simple "theorems" about sets and operations over them.

REFINED SET API To make it easy to write down theorems, we've refined the types of the operators in `Data.Set` so that they mirror their logical counterparts:

```
empty        :: {v:Set a | v = empty}
singleton    :: x:a -> {v:Set a | v = singleton x}
union        :: x:Set a -> y:Set a -> {v:Set a | v = union x y}
intersection :: x:Set a -> y:Set a -> {v:Set a | v = intersection x y}
difference   :: x:Set a -> y:Set a -> {v:Set a | v = difference x y}
member       :: x:a -> s:Set a -> {v:Bool | Prop v <=> member x s}
```

ASSERTING PROPERTIES Lets write our theorems as QuickCheck style *properties*, that is, as functions from arbitrary inputs to a `Bool` output that must always be `True`. Lets define aliases for the singletons `True` and `False`:

```
{-@ type True  = {v:Bool |      Prop v } @-}
{-@ type False = {v:Bool | not (Prop v)} @-}
```

We can use `True` to state and prove theorems. For example, something (boring) like the arithmetic equality above becomes:

```
{-@ prop_one_plus_one_eq_two :: _ -> True @-}
prop_one_plus_one_eq_two x   = (x == 1 + 1) `implies` (x == 2)
```

Where `implies` is just the implication function over `Bool`

```
{-@ implies        :: p:_ -> q:_ -> {v:Bool | Prop v <=> (Prop p => Prop q)} @-}
implies False _     = True
implies _     True = True
implies _     _     = False
-- implies p q = not p || q
```

EXERCISE 7.1. [BOUNDED ADDITION] Write a QuickCheck style proof of the fact that $x < 100 \wedge y < 100 \Rightarrow x + y < 200$.

```
{-@ prop_x_y_200 :: _ -> _ -> True @-}
prop_x_y_200 x y = False -- fill in the appropriate body to obtain the theorem.
```

INTERSECTION IS COMMUTATIVE Ok, lets prove things about sets and their operators! First, lets check that `intersection` is commutative:

```
{-@ prop_intersection_comm :: _ -> _ -> True @-}
prop_intersection_comm x y
  = (x `intersection` y) == (y `intersection` x)
```

UNION IS ASSOCIATIVE Similarly, we might verify that union is associative:

```
{-@ prop_intersection_comm :: _ -> _ -> True @-}
prop_union_assoc x y z
  = (x `union` (y `union` z)) == (x `union` y) `union` z
```

UNION DISTRIBUTES OVER INTERSECTION and while we're at it, check various distributivity laws of Boolean algebra:

```
{-@ prop_intersection_dist :: _ -> _ -> _ -> True @-}
prop_intersection_dist x y z
  =  x `intersection` (y `union` z) == (x `intersection` y) `union` (x `intersection` z)
```

NON-THEOREMS Of course, while we're at it, let's make sure LiquidHaskell doesn't prove anything that *isn't* true . . .

```
{-@ prop_cup_dif_bad :: _ -> _ -> True @-}
prop_cup_dif_bad x y
```

```
  = pre `implies` (x == ((x `union` y) `difference` y))
  where
    pre = True  -- Fix this with a non-trivial precondition
```

Exercise 7.2. [Set Difference] Do you know why the above fails? 1. Use QuickCheck to find a *counterexample* for the property `prop_cup_dif_bad`, and, 2. Use the counterexample to assign `pre` a non-trivial (i.e. non `False`) condition so that the property can be proved.

Thus, LiquidHaskell's refined types offer a nice interface for interacting with the SMT solvers in order to *prove* theorems, while letting us use QuickCheck to generate counterexamples.

[0] The [SBV](https://github.com/LeventErkok/sbv) and [Leon](http://lara.epfl.ch/w/leon) projects describe a different DSL based approach for using SMT solvers from Haskell and Scala respectively.

Content-Aware List API

Our overall goal is to verify properties of programs. Lets start off by refining the list API to precisely track the list elements.

Elements of a List To specify the permutation property, we need a way to talk about the set of elements in a list. At this point, hopefully you know what we're going to do: write a measure!

```
{-@ measure elems @-}
elems        :: (Ord a) => [a] -> Set a
elems []     = empty
elems (x:xs) = singleton x `union` elems xs
```

Strengthened Constructors Recall, that as before, the above definition automatically strengthens the types for the constructors:

```
data [a] where
  []  :: {v:[a] | v = empty }
  (:) :: x:a -> xs:[a] -> {v:[a] | elems v = union (singleton x) (elems xs)}
```

Next, to make the specifications concise, let's define a few predicate aliases:

```
{-@ predicate EqElts  X Y   = elems X = elems Y                         @-}
{-@ predicate SubElts X Y   = Set_sub (elems X) (elems Y)               @-}
{-@ predicate DisjElts X Y  = Set_empty 0 = Set_cap (elems X) (elems Y) @-}
{-@ predicate Empty   X     = elems X = Set_empty 0                     @-}
{-@ predicate UnElts  X Y Z = elems X = Set_cup (elems Y) (elems Z)     @-}
{-@ predicate UnElt   X Y Z = elems X = Set_cup (Set_sng Y) (elems Z)   @-}
{-@ predicate Elem    X Y   = Set_mem X (elems Y)                       @-}
```

APPEND First, here's good old `append`, but now with a specification that states that the output indeed includes the elements from both the input lists.

```
{-@ append'       :: xs:[a] -> ys:[a] -> {v:[a] | UnElts v xs ys} @-}
append' []     ys = ys
append' (x:xs) ys = x : append' xs ys
```

EXERCISE 7.3. [REVERSE] Write down a type for `revHelper` so that `reverse'` is verified by LiquidHaskell:

```
{-@ reverse' :: xs:[a] -> {v:[a] | EqElts v xs} @-}
reverse' xs = revHelper [] xs

revHelper acc []     = acc
revHelper acc (x:xs) = revHelper (x:acc) xs
```

EXERCISE 7.4. [PARTITION] ⋆ Write down a specification for `split` such that the subsequent "theorem" `prop_partition_appent` is proved by LiquidHaskell.

```
split            :: Int -> [a] -> ([a], [a])
split 0 xs       = ([], xs)
split n (x:y:zs) = (x:xs, y:ys) where (xs, ys) = split (n-1) zs
split _ xs       = ([], xs)

{-@ prop_split_append  :: _ -> _ -> True @-}
prop_split_append n xs = elems xs == elems xs'
  where
    xs'      =  append' ys zs
    (ys, zs) =  split n xs
```

Hint: You may want to remind yourself about the "dimension-aware" signature for `partition` from the earlier chapter.

EXERCISE 7.5. [MEMBERSHIP] Write down a signature for `elem` that suffices to verify `test1` and `test2` by LiquidHaskell.

```
{-@ elem      :: (Eq a) => a -> [a] -> Bool @-}
elem x (y:ys) = x == y || elem x ys
elem _ []     = False

{-@ test1 :: True @-}
test1      = elem 2 [1,2,3]
```

```
{-@ test2 :: False @-}
test2      = elem 2 [1,3]
```

Permutations

Next, lets use the refined list API to prove that various list-sorting routines return *permutations* of their inputs, that is, return output lists whose elements are the *same as* those of the input lists. Since we are focusing on the elements, lets not distract ourselves with the ordering invariant just, and reuse plain old lists.

[0] See [this](http://goto.ucsd.edu/ rjhala/liquid/haskell/blog/b things-in-order.lhs/) for how to specify and verify order with plain old lists.

InsertionSort is the simplest of all the list sorting routines; we build up an (ordered) output list inserting each element of the input list into the appropriate position of the output:

```
insert x []     = [x]
insert x (y:ys)
  | x <= y      = x : y : ys
  | otherwise   = y : insert x ys
```

Thus, the output of insert has all the elements of the input xs, plus the new element x:

```
{-@ insert :: x:a -> xs:[a] -> {v:[a] | UnElt v x xs } @-}
```

Which then lets us prove that the output of the sorting routine indeed has the elements of the input:

```
{-@ insertSort    :: (Ord a) => xs:[a] -> {v:[a] | EqElts v xs} @-}
insertSort []     = []
insertSort (x:xs) = insert x (insertSort xs)
```

Exercise 7.6. [Merge] Write down a specification of merge such that the subsequent property is verified by LiquidHaskell:

```
{-@ merge :: xs:_ -> ys:_ -> {v:_ | UnElts v xs ys} @-}
merge (x:xs) (y:ys)
  | x <= y           = x : merge xs (y:ys)
  | otherwise        = y : merge (x:xs) ys
merge [] ys          = ys
merge xs []          = xs

{-@ prop_merge_app   :: _ -> _ -> True   @-}
```

```
prop_merge_app xs ys = elems zs == elems zs'
  where
    zs               = append' xs ys
    zs'              = merge   xs ys
```

Exercise 7.7. [MergeSort] ⋆⋆ Once you write the correct type for `merge` above, you should be able to prove the surprising signature for `mergeSort` below.

```
{-@ mergeSort :: (Ord a) => xs:[a] -> {v:[a] | Empty v} @-}
mergeSort []  = []
mergeSort xs  = merge (mergeSort ys) (mergeSort zs)
  where
   (ys, zs)   = split mid xs
   mid        = length xs `div` 2
```

First, make sure you are able verify the given signature. Next, obviously we don't want `mergeSort` to return the empty list, so there's a bug somewhere in the code. Find and fix it, so that you *cannot* prove that the output is empty, but *can* prove that `EqElts v xs`.

Uniqueness

Often, we want to enforce the invariant that a particular collection contains *no duplicates*; as multiple copies in a collection of file handles or system resources can create unpleasant leaks. For example, the XMonad window manager creates a sophisticated *zipper* data structure to hold the list of active user windows, and carefully maintains the invariant that that there are no duplicates. Next, lets see how to specify and verify this invariant using LiquidHaskell, first for lists, and then for a simplified zipper.

Specifying Uniqueness How would we even describe the fact that a list has no duplicates? There are in fact multiple different ways, but the simplest is a *measure*:

```
{-@ measure unique @-}
unique         :: (Ord a) => [a] -> Bool
unique []      = True
unique (x:xs) = unique xs && not (member x (elems xs))
```

We can define an alias for duplicate-free lists

```
{-@ type UList a = {v:[a] | unique v }@-}
```

and then do a quick sanity check, that the right lists are indeed `unique`

```
{-@ isUnique    :: UList Int @-}
isUnique        = [1, 2, 3]         -- accepted by LH

{-@ isNotUnique :: UList Int @-}
isNotUnique     = [1, 2, 3, 1]      -- rejected by LH
```

Filter Lets write some functions that preserve `uniqueness`. For example, `filter` returns a subset of its elements. Hence, if the input was unique, the output is too:

```
{-@ filter      :: _ -> xs:UList a -> {v: UList a | SubElts v xs} @-}
filter _ []     = []
filter f (x:xs)
  | f x         = x : xs'
  | otherwise   = xs'
  where
    xs'         = filter f xs
```

Exercise 7.8. [Reverse] ⋆ When we `reverse` their order, the set of elements is unchanged, and hence unique (if the input was unique). Why does LiquidHaskell reject the below? Can you fix things so that we can prove that the output is a `UList a`?

```
{-@ reverse     :: xs:UList a -> UList a    @-}
reverse         = go []
  where
    {-@ go      :: acc:[a] -> xs:[a] -> [a] @-}
    go a []     = a
    go a (x:xs) = go (x:a) xs
```

Nub One way to create a `unique` list is to start with an ordinary list and throw away elements that we have seen already.

```
nub xs                = go [] xs
  where
    go seen []        = seen
    go seen (x:xs)
      | x `isin` seen = go seen     xs
      | otherwise     = go (x:seen) xs
```

The key membership test is done by `isin`, whose output is `True` exactly when the element is in the given list.

[0] Which should be clear by now, if you did the exercise above ...

```
{-@ isin :: x:_ -> ys:_ -> {v:Bool | Prop v <=> Elem x ys }@-}
isin x (y:ys)
  | x == y    = True
  | otherwise = x `isin` ys
isin _ []     = False
```

EXERCISE 7.9. [APPEND] ⋆ Why does appending two `UList`s not return a `UList`? Fix the type signature below so that you can prove that the output is indeed `unique`.

```
{-@ append       :: UList a -> UList a -> UList a @-}
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

EXERCISE 7.10. [RANGE] ⋆⋆ In the below `range i j` returns the list of all `Int` between `i` and `j`. Yet, LiquidHaskell refuses to acknowledge that the output is indeed a `UList`. Modify the specification and implementation, if needed, to obtain an equivalent of `range` which *provably* returns a `UList Int`.

```
{-@ type Btwn I J = {v:_ | I <= v && v < J} @-}

{-@ range     :: i:Int -> j:Int -> UList (Btwn i j) @-}
range i j
  | i < j     = i : range (i + 1) j
  | otherwise = []
```

Unique Zippers

A zipper is an aggregate data stucture that is used to arbitrarily traverse the structure and update its contents. For example, a zipper for a list is a data type that contains an element (called `focus`) that we are currently `focus`-ed on, a list of elements to the `left` of (i.e. before) the focus, and a list of elements to the `right` (i.e. after) the focus.

```
data Zipper a = Zipper {
    focus  :: a
  , left   :: [a]
  , right  :: [a]
  }
```

XMonad is a wonderful tiling window manager, that uses a zipper to store the set of windows being managed. Xmonad requires the crucial invariant that the values in the zipper be unique, i.e. have no duplicates.

Refined Zipper

We can specify that all the values in the zipper are unique by refining the `Zipper` data declaration to express that both the lists in the structure are unique, disjoint, and do not include `focus`.

```
{-@ data Zipper a = Zipper {
      focus :: a
    , left  :: {v: UList a | not (Elem focus v)}
    , right :: {v: UList a | not (Elem focus v) && DisjElts v left }
    } @-}
```

Constructing Zippers Our refined type makes *illegal states unrepresentable*; by construction, we will ensure that every `Zipper` is free of duplicates. Of course, it is straightforward to create a valid `Zipper` from a `unique` list:

```
{-@ differentiate    :: UList a -> Maybe (Zipper a) @-}
differentiate []     = Nothing
differentiate (x:xs) = Just $ Zipper x [] xs
```

Exercise 7.11. [Deconstructing Zippers] ⋆ Dually, the elements of a unique zipper tumble out into a unique list. Strengthen the types of `reverse` and `append` above so that LiquidHaskell accepts the below signatures for `integrate`:

```
{-@ integrate            :: Zipper a -> UList a @-}
integrate (Zipper x l r) = reverse l `append` (x : r)
```

Shifting Focus We can shift the focus element left or right while preserving the invariants:

```
focusLeft                      :: Zipper a -> Zipper a
focusLeft (Zipper t [] rs)     = Zipper x xs []     where (x:xs) = reverse (t:rs)
focusLeft (Zipper t (l:ls) rs) = Zipper l ls (t:rs)

focusRight                     :: Zipper a -> Zipper a
focusRight                     = reverseZipper . focusLeft . reverseZipper

reverseZipper                  :: Zipper a -> Zipper a
reverseZipper (Zipper t ls rs) = Zipper t rs ls
```

FILTER Finally, using the filter operation on lists allows LiquidHaskell to prove that filtering a zipper also preserves uniqueness.

```
filterZipper :: (a -> Bool) -> Zipper a -> Maybe (Zipper a)
filterZipper p (Zipper f ls rs) = case filter p (f:rs) of
    f':rs' -> Just $ Zipper f' (filter p ls) rs'      -- maybe move focus right
    []     -> case filter p ls of                     -- filter back left
                   f':ls' -> Just $ Zipper f' ls' [] -- else left
                   []     -> Nothing
```

Recap

In this chapter, we saw how SMT solvers can let us reason precisely about the actual *contents* of data structures, via the theory of sets. We can

- Lift the set-theoretic primitives to (refined) Haskell functions from the `Data.Set` library,
- Use the functions to define measures like `elems` that characterize the contents of structures, and `unique` that describe high-level application specific properties.
- Use LiquidHaskell to then specify and verify that implementations enjoy various functional correctness properties, e.g. that sorting routines return permutations of their inputs, and various zipper operators preserve uniqueness.

Next, we present a variety of *case-studies* illustrating the techniques so far on particular application domains.

8
Case Study: Associative Maps

Recall the following from the introduction.

```
ghci> :m +Data.Map
ghci> let m = fromList [ ("haskell"   , "lazy")
                       , ("javascript", "eager")]

ghci> m ! "haskell"
"lazy"

ghci> m ! "python"
"*** Exception: key is not in the map
```

The problem illustrated above is quite a pervasive one; associative maps pop up everywhere. Failed lookups are the equivalent of `NullPointerDereference` exceptions in languages like Haskell. It is rather difficult to use Haskell's type system to precisely characterize the behavior of associative map APIs as ultimately, this requires tracking the *dynamic set of keys* in the map.

In this case study, we'll see how to combine two techniques – measures for reasoning about the *sets* of elements in structures, and refined data types for reasoning about order invariants – can be applied to programs that use associative maps (e.g. `Data.Map` or `Data.HashMap`).

Specifying Maps

Lets start by defining a *refined API* for Associative Maps that tracks the set of keys stored in the map, in order to statically ensure the safety of lookups.

Types First, we need an (currently abstract) type for `Map`s. As usual, lets parameterize the type with `k` for the type of keys and `v` for the type of values.

```
-- | Data Type
data Map k v
```

KEYS To talk about the set of keys in a map, we will use a *measure*

```
measure keys :: Map k v -> Set k
```

that associates each `Map` to the `Set` of its defined keys. Next, we use the above measure, and the usual `Set` operators to refine the types of the functions that *create*, *add* and *lookup* key-value bindings, in order to precisely track, within the type system, the `keys` that are dynamically defined within each `Map`.

EMPTY `Maps` have no keys in them. Hence, we defined a predicate alias, `NoKey` and use it to type `emp` which is used to denote the empty `Map`:

```
emp :: {m:Map k v | NoKey m}

predicate NoKey M = keys M = Set_empty 0
```

ADD The function `set` takes a key k a value v and a map `m` and returns the new map obtained by extending `m` with the binding $k \mapsto v$. Thus, the set of `keys` of the output `Map` includes those of the input plus the singleton k, that is:

```
set :: (Ord k) => k:k -> v -> m:Map k v -> {n: Map k v | PlusKey k m n}

predicate PlusKey K M N = keys N = Set_cup (Set_sng K) (keys M)
```

QUERY Finally, queries will only succeed for keys that are defined a given `Map`. Thus, we define an alias:

```
predicate HasKey K M    = Set_mem K (keys M)
```

and use it to type `mem` which *checks* if a key is defined in the `Map` and `get` which actually returns the value associated with a given key.

```
-- | Check if key is defined
mem :: (Ord k) => k:k -> m:Map k v -> {v:Bool | Prop v <=> HasKey k m}

-- | Lookup key's value
get :: (Ord k) => k:k -> {m:Map k v | HasKey k m} -> v
```

Using Maps: Well Scoped Expressions

Rather than jumping into the *implementation* of the above `Map` API, lets write a *client* that uses `Maps` to implement an interpreter for a tiny language. In particular, we will use maps as an *environment* containing the values of *bound variables*, and we will use the refined API to ensure that *lookups never fail*, and hence, that well-scoped programs always reduce to a value.

Expressions Lets work with a simple language with integer constants, variables, binding and arithmetic operators:

[0] Feel free to embellish the language with fancier features like functions, tuples etc.

```
type Var  = String

data Expr = Const Int
          | Var   Var
          | Plus  Expr Expr
          | Let   Var  Expr Expr
```

Values We can use refinements to formally describe *values* as a subset of `Expr` allowing us to reuse a bunch of code. To this end, we simply define a (`measure`) predicate characterizing values:

```
{-@ measure val @-}
val              :: Expr -> Bool
val (Const _)    = True
val (Var _)      = False
val (Plus _ _)   = False
val (Let _ _ _ ) = False
```

and then we can use the lifted `measure` to define an alias for `Val` denoting values:

```
{-@ type Val = {v:Expr | val v} @-}
```

we can use the above to write simple *operators* on `Val`, for example:

```
{-@ plus                 :: Val -> Val -> Val @-}
plus (Const i) (Const j) = Const (i+j)
plus _         _         = die "Bad call to plus"
```

Environments let us save values for the "local" i.e. *let-bound* variables; when evaluating an expression `Var x` we simply look up the value of `x` in the environment. This is why `Map`s were invented! Lets define our environments as `Map`s from `Var`iables to `Val`ues:

```
{-@ type Env = Map Var Val @-}
```

The above definition essentially specifies, inside the types, an *eager* evaluation strategy: LiquidHaskell will prevent us from sticking unevaluated `Exprs` inside the environments.

Evaluation proceeds via a straightforward recursion over the structure of the expression. When we hit a `Var` we simply query its value from the environment. When we hit a `Let` we compute the bound expression and tuck its value into the environment before proceeding within.

```
eval _ i@(Const _)   = i
eval g (Var x)       = get x g
eval g (Plus e1 e2)  = plus  (eval g e1) (eval g e2)
eval g (Let x e1 e2) = eval g' e2
  where
    g'               = set x v1 g
    v1               = eval g e1
```

The above `eval` seems rather unsafe; whats the guarantee that `get x g` will succeed? For example, surely trying:

```
ghci> eval emp (Var "x")
```

will lead to some unpleasant crash. Shouldn't we *check* if the variables is present and if not, fail with some sort of `Variable Not Bound` error? We could, but we can do better: we can prove at compile time, that such errors will not occur.

Free Variables are those whose values are *not* bound within an expression, that is, the set of variables that *appear* in the expression, but are not *bound* by a dominating `Let`. We can formalize this notion as a (lifted) function:

```
{-@ measure free @-}
free               :: Expr -> (Set Var)
free (Const _)     = empty
free (Var x)       = singleton x
free (Plus e1 e2)  = (free e1) `union` (free e2)
free (Let x e1 e2) = (free e1) `union` ((free e2) `difference` (singleton x))
```

An Expression is Closed with respect to an environment `G` if all the *free* variables in the expression appear in `G`, i.e. the environment contains bindings for all the variables in the expression that are *not* bound within the expression. As we've seen repeatedly, often a whole pile of informal handwaving, can be succinctly captured by a type definition that says the `free` variables in the `Expr` must be contained in the `keys` of the environment `G`:

```
{-@ type ClosedExpr G = {v:Expr | Subset (free v) (keys G)} @-}
```

CLOSED EVALUATION never goes wrong, i.e. we can ensure that `eval` will not crash with unbound variables, as long as it is invoked with suitable environments:

```
{-@ eval :: g:Env -> ClosedExpr g -> Val @-}
```

We can be sure an `Expr` is well-scoped if it has *no* free variables.Lets use that to write a "top-level" evaluator:

```
{-@ topEval :: {v:Expr | Empty (free v)} -> Val @-}
topEval     = eval emp
```

EXERCISE 8.1. Complete the definition of the below function which *checks* if an `Expr` is well formed before `eval`uating it:

```
{-@ evalAny   :: Env -> Expr -> Maybe Val @-}
evalAny g e
  | ok        = Just $ eval g e
  | otherwise = Nothing
  where
    ok        = undefined
```

Proof is all well and good, in the end, you need a few sanity tests to kick the tires. So:

```
tests   = [v1, v2]
  where
    v1  = topEval e1          -- Rejected by LH
    v2  = topEval e2          -- Accepted by LH
    e1  = (Var x) `Plus` c1
    e2  = Let x c10 e1
    x   = "x"
    c1  = Const 1
    c10 = Const 10
```

EXERCISE 8.2. [FUNCTIONS AND CLOSURES] ⋆⋆ Extend the language above to include functions. That is, extend

```
data Expr = ... | Fun Var Expr | App Expr Expr
```

Just focus on ensuring the safety of variable lookups; ensuring full type-safety (i.e. every application is to a function) is rather more complicated and beyond the scope of what we've seen so far.

Implementing Maps: Binary Search Trees

We just saw how easy it is to *use* the Associative Map API to ensure the safety of lookups, even though the `Map` has a "dynamically" generated set of keys. Next, lets see how we can *implement* a `Map` library that respects the API using Binary Search Trees

Data Type First, lets provide an implementation of the (hitherto abstract) data type for `Map`. We shall use Binary Search Trees, wherein, at each `Node`, the `left` (resp. `right`) subtree has keys that are less than (resp. greater than) the root `key`.

```
{-@ data Map k v = Node { key   :: k
                        , value :: v
                        , left  :: Map {v:k | v < key} v
                        , right :: Map {v:k | key < v} v }
                 | Tip
  @-}
```

[Recall](#binarysearchtree) that the above refined data definition yields strengthened data constructors that statically ensure that only legal, *binary-search ordered* trees are created in the program.

Defined Keys Next, we must provide an implementation of the notion of the `keys` that are defined for a given `Map`. This is achieved via the (lifted) measure function:

```
{-@ measure keys @-}
keys                :: (Ord k) => Map k v -> Set k
keys Tip            = empty
keys (Node k _ l r) = union (singleton k) (union (keys l) (keys r))
```

Armed with the basic type and measure definition, we can start to fill in the operations for `Maps`.

Exercise 8.3. [Empty Maps] To make sure you are following, fill in the definition for an `empty` Map:

```
{-@ emp :: {m:Map k v | NoKey m} @-}
emp     = undefined
```

Exercise 8.4. [Insert] To add a key `k'` to a `Map` we recursively traverse the `Map` zigging `left` or `right` depending on the result of comparisons with the keys along the path. Unfortunately, the version below has an (all too common!) bug, and hence, is *rejected* by LiquidHaskell. Find and fix the bug so that the function is verified.

```
{-@ set :: (Ord k) => k:k -> v -> m:Map k v -> {n: Map k v | PlusKey k m n} @-}
set k' v' (Node k v l r)
  | k' == k   = Node k v' l r
  | k' <  k   = set k' v l
  | otherwise = set k' v r
set k' v' Tip = Node k' v' Tip Tip
```

Lookup Next, lets write the `mem` function that returns the value associated with a key `k'`. To do so we just compare `k'` with the root key, if they are equal, we return the binding, and otherwise we go down the `left` (resp. `right`) subtree if sought for key is less (resp. greater) than the root `key`. Crucially, we want to check that lookup *never fails*, and hence, we implement the `Tip` (i.e. empty) case with `die` gets LiquidHaskell to prove that that case is indeed dead code, i.e. never happens at run-time.

```
{-@ get' :: (Ord k) =>  k:k -> m:{Map k v | HasKey k m} -> v @-}
get' k' m@(Node k v l r)
  | k' == k   = v
  | k' <  k   = get' k' l
  | otherwise = get' k' r
get'  _ Tip   = die  "Lookup Never Fails"
```

Unfortunately the function above is *rejected* by LiquidHaskell. This is a puzzler (and a bummer!) because in fact it *is* correct. So what gives? Well, lets look at the error for the call `get' k' l`

```
src/07-case-study-associative-maps.lhs:411:25: Error: Liquid Type Mismatch
  Inferred type
    VV : (Map a b) | VV == l

  not a subtype of Required type
    VV : (Map a b) | Set_mem k' (keys VV)

  In Context
    VV : (Map a b) | VV == l
    k  : a
    l  : (Map a b)
    k' : a
```

LiquidHaskell is *unable* to deduce that the the key `k'` definitely belongs in the `left` subtree `l`. Well, lets ask ourselves: *why* must `k'` belong in the left subtree? From the input, we know `HasKey k' m` i.e. that `k'` is *somewhere* in `m`. That is *one of* the following holds:

1. `k' == k` or,
2. `HasKey k' l` or,
3. `HasKey k' r`.

As the preceding guard `k' == k` fails, we (and LiquidHaskell) can rule out case (1). Now, what about the `Map` tells us that case (2) must hold, i.e. that case (3) cannot hold? The *BST invariant*, all keys in `r` exceed `k` which itself exceeds `k'`. That is, all nodes in `r` are *disequal* to `k'` and hence `k'` cannot be in `r`, ruling out case (3). Formally, we need the fact that:

$$\forall\ \mathtt{key},\ \mathtt{t}.\mathtt{t} :: \mathtt{Map}\ \{\mathtt{key}'{:}k \mid \mathtt{key}' \neq \mathtt{key}\}\ v\ \Rightarrow\ \neg(\mathtt{HasKey}\ \mathtt{key}\ \mathtt{t})$$

Conversion Lemmas Unfortunately, LiquidHaskell *cannot automatically* deduce facts like the above, as they relate refinements of a container's *type parameters* (here: $\mathtt{key}' \neq \mathtt{key}$, which refines the `Maps` first type parameter) with properties of the entire container (here: `HasKey key t`). Fortunately, it is both easy to *state*, *prove* and *use* facts like the above.

[0] Why not? This is tricky to describe. Intuitively, because there is no way of automatically connecting the *traversal* corresponding to ‘keys‘ with the type variable ‘k‘. I wish I had a better way to explain this rather subtle point; suggestions welcome!

Defining Lemmas To state a lemma, we need only convert it into a type by viewing universal quantifiers as function parameters, and implications as function types:

```
{-@ lemma_notMem :: key:k -> m:Map {k:k | k /= key} v -> {v:Bool | not (HasKey key m)} @-}
lemma_notMem _   Tip            = True
lemma_notMem key (Node _ _ l r) = lemma_notMem key l && lemma_notMem key r
```

Proving Lemmas Note how the signature for `lemma_notMem` corresponds exactly to the missing fact from above. The "output" type is a `Bool` refined with the proposition that we desire. We *prove* the lemma simply by *traversing* the tree which lets LiquidHaskell build up a proof for the output fact by inductively combining the proofs from the subtrees.

Using Lemmas To use a lemma, we need to *instantiate* it to the particular keys and trees we care about, by "calling" the lemma function, and forcing its result to be in the *environment* used to typecheck the expression where we want to use the lemma. Say what? Here is a verified `get`:

```
{-@ get :: (Ord k) => k:k -> m:{Map k v | HasKey k m} -> v @-}
get k' (Node k v l r)
  | k' == k   = v
  | k' <  k   = assert (lemma_notMem k' r) $
```

```
                  get k' l
  | otherwise = assert (lemma_notMem k' l) $
                  get k' r
get _ Tip     = die "Lookup failed? Impossible."
```

By calling `lemma_notMem` we create a dummy `Bool` that carries the desired refinement that tells LiquidHaskell that `not (HasKey k' r)` (resp. `not (HasKey k' l)`). We force the calls to `get k' l` (resp. `get k' r`) to be typechecked using the materialized refinement by wrapping the calls within a function `assert`

```
assert _ x = x
```

GHOST VALUES This technique of materializing auxiliary facts via *ghost values* is a well known idea in the program verification literature. Usually, one has to take care to ensure that ghost computations do not interfere with the regular computations. If we had to actually *execute* `lemma_notMem` it would totally wreck the efficient logarithmic lookup times as we'd traverse the entire tree all the time

[0] Assuming we kept the trees balanced

[0] Which is what makes dynamic contract checking [rather slow](findler-contract) for such invariants

LAZINESS comes to our rescue: as the ghost value is (trivially) not needed, it is never computed. In fact, it is straightforward to entirely *erase* the call in the compiled code, which lets us freely `assert` such `lemmas` to carry out proofs, without paying any runtime penalty. In an eager language we would have to do a bit of work to specifically mark the computation as a ghost or irrelevant but in the lazy setting we get this for free.

EXERCISE 8.5. [MEMBERSHIP TEST] Capisce? Fix the definition of `mem` so that it verifiably implements the given signature:

```
{-@ mem :: (Ord k) => k:k -> m:Map k v -> {v:Bool | Prop v <=> HasKey k m} @-}
mem k' (Node k _ l r)
  | k' == k    = True
  | k' <  k    = mem k' l
  | otherwise = mem k' r
mem _ Tip     = False
```

EXERCISE 8.6. [FRESH] ⋆⋆ To make sure you really understand this business of ghosts values and proofs, complete the implementation of the following function which returns a `fresh` integer that is *distinct* from all the values in its input list:

```
{-@ fresh :: xs:[Int] -> {v:Int | not (Elem v xs)} @-}
fresh = undefined
```

To refresh your memory, here are the definitions for `Elem` we saw earlier

```
{-@ predicate Elem X Ys  = Set_mem X (elems Ys) @-}
{-@ measure elems @-}
elems []     = empty
elems (x:xs) = (singleton x) `union` (elems xs)
```

Recap

In this chapter we saw how to combine several of the techniques from previous chapters in a case study. We learnt how to:

1. **Define** an API for associative maps that used refinements to track the *set* of `keys` stored in a map, in order to prevent lookup failures, the `NullPointerDereference` errors of the functional world,

2. **Use** the API to implement a small interpreter that is guaranteed to never fail with `UnboundVariable` errors, as long as the input expressions were closed,

3. **Implement** the API using Binary Search Trees; in particular, using *ghost lemmas* to `assert` facts that LiquidHaskell is otherwise unable to deduce automatically.

9
Case Study: Pointers and ByteStrings

A large part of the allure of Haskell is its elegant, high-level ADTs that ensure that programs won't be plagued by problems like the infamous SSL heartbleed bug. However, another part of Haskell's charm is that when you really really need to, you can drop down to low-level pointer twiddling to squeeze the most performance out of your machine. But of course, that opens the door to the heartbleeds.

[0] Assuming, of course, the absence of errors in the compiler and run-time...

Wouldn't it be nice to have have our cake and eat it too? Wouldn't it be great if we could twiddle pointers at a low-level and still get the nice safety assurances of high-level types? Lets see how Liquid-Haskell lets us have our cake and eat it too.

HeartBleeds in Haskell

Modern Languages like Haskell are ultimately built upon the foundation of `C`. Thus, implementation errors could open up unpleasant vulnerabilities that could easily slither past the type system and even code inspection. As a concrete example, lets look at a a function that uses the `ByteString` library to truncate strings:

```
chop'     :: String -> Int -> String
chop' s n = s'
  where
    b     = pack s          -- down to low-level
    b'    = unsafeTake n b  -- grab n chars
    s'    = unpack b'       -- up to high-level
```

First, the function `pack`s the string into a low-level bytestring `b`, then it grabs the first `n` `Char`acters from `b` and translates them back into a high-level `String`. Lets see how the function works on a small test:

```
ghci> let ex = "Ranjit Loves Burritos"
```

We get the right result when we chop a *valid* prefix:

```
ghci> chop' ex 10
"Ranjit Lov"
```

But, as illustrated in fig. 9.1, the machine silently reveals (or more colorfully, *bleeds*) the contents of adjacent memory or if we use an *invalid* prefix:

```
ghci> heartBleed ex 30
"Ranjit Loves Burritos\NUL\201\&1j\DC3\SOH\NUL"
```

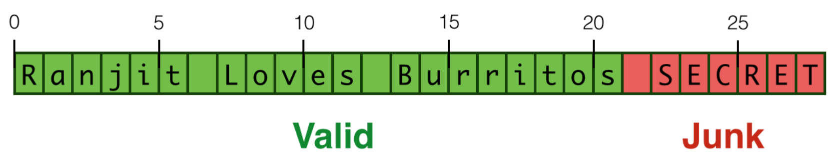

Figure 9.1: Can we prevent the program from leaking 'secret's?

TYPES AGAINST OVERFLOWS Now that we have stared the problem straight in the eye, look at how we can use LiquidHaskell to prevent the above at compile time. To this end, we decompose the system into a hierarchy of levels (i.e. modules). Here, we have three levels:

1. *Machine* level `Pointers`
2. *Library* level `ByteString`
3. *User* level `Application`

Our strategy, as before, is to develop an *refined API* for each level such that errors at each level are prevented by using the typed interfaces for the lower levels. Next, lets see how this strategy lets us safely manipulate pointers.

Low-level Pointer API

To get started, lets look at the low-level pointer API that is offered by GHC and the run-time. First, lets see who the *dramatis personae* are and how they might let heartbleeds in. Then we will see how to batten down the hatches with LiquidHaskell.

POINTERS are an (abstract) type `Ptr a` implemented by GHC.

```
-- | A value of type `Ptr a` represents a pointer to an object,
--   or an array of objects, which may be marshalled to or from
--   Haskell values of type `a`.

data Ptr a
```

FOREIGN POINTERS are *wrapped* pointers that can be exported to and from C code via the Foreign Function Interface.

```
data ForeignPtr a
```

TO CREATE a pointer we use `mallocForeignPtrBytes n` which creates a `Ptr` to a buffer of size `n` and wraps it as a `ForeignPtr`

```
mallocForeignPtrBytes :: Int -> ForeignPtr a
```

TO UNWRAP and actually use the `ForeignPtr` we use

```
withForeignPtr :: ForeignPtr a     -- pointer
               -> (Ptr a -> IO b)  -- action
               -> IO b             -- result
```

That is, `withForeignPtr fp act` lets us execute a action `act` on the actual `Ptr` wrapped within the `fp`. These actions are typically sequences of *dereferences*, i.e. reads or writes.

TO DEREFERENCE a pointer, i.e. to read or update the contents at the corresponding memory location, we use `peek` and `poke` respectively.

[0] We elide the `Storable` type class constraint to strip this presentation down to the absolute essentials.

```
peek :: Ptr a -> IO a           -- Read
poke :: Ptr a -> a -> IO ()     -- Write
```

FOR FINE GRAINED ACCESS we can directly shift pointers to arbitrary offsets using the *pointer arithmetic* operation `plusPtr p off` which takes a pointer `p` an integer `off` and returns the address obtained shifting `p` by `off`:

```
plusPtr :: Ptr a -> Int -> Ptr b
```

EXAMPLE That was rather dry; lets look at a concrete example of how one might use the low-level API. The following function allocates a block of 4 bytes and fills it with zeros:

```
zero4 = do fp <- mallocForeignPtrBytes 4
           withForeignPtr fp $ \p -> do
             poke (p `plusPtr` 0) zero
             poke (p `plusPtr` 1) zero
             poke (p `plusPtr` 2) zero
             poke (p `plusPtr` 3) zero
           return fp
        where
           zero = 0 :: Word8
```

While the above is perfectly all right, a small typo could easily slip past the type system (and run-time!) leading to hard to find errors:

```
zero4' = do fp <- mallocForeignPtrBytes 4
            withForeignPtr fp $ \p -> do
              poke (p `plusPtr` 0) zero
              poke (p `plusPtr` 1) zero
              poke (p `plusPtr` 2) zero
              poke (p `plusPtr` 8) zero
            return fp
         where
            zero = 0 :: Word8
```

A Refined Pointer API

Wouldn't it be great if we had an assistant to helpfully point out the error above as soon as we *wrote* it? We will use the following strategy to turn LiquidHaskell into such an assistant:

[0] In Vim or Emacs, you'd see the error helpfully underlined.

1. *Refine* pointers with allocated buffer size,
2. *Track* sizes in pointer operations,
3. *Enforce* pointer are valid at reads and writes.

To Refine Pointers with the *size* of their associated buffers, we can use an *abstract measure*, i.e. a measure specification *without* any underlying implementation.

```
-- | Size of `Ptr`
measure plen  :: Ptr a -> Int

-- | Size of `ForeignPtr`
measure fplen :: ForeignPtr a -> Int
```

It is helpful to define aliases for pointers of a given size `N`:

```
type PtrN a N        = {v:Ptr a        | plen v  = N}
type ForeignPtrN a N = {v:ForeignPtr a | fplen v = N}
```

Abstract Measures are extremely useful when we don't have a concrete implementation of the underlying value, but we know that the value *exists*. Here, we don't have the value – inside Haskell – because the buffers are manipulated within C. However, this is no cause for alarm as we will simply use measures to refine the API, not to perform any computations.

HEREHEREHEREHERE

[0] This is another example of a *ghost* specification.

To REFINE ALLOCATION we stipulate that the size parameter be non-negative, and that the returned pointer indeed refers to a buffer with exactly `n` bytes:

```
mallocForeignPtrBytes :: n:Nat -> ForeignPtrN a n
```

To REFINE UNWRAPPING we specify that the *action* gets as input, an unwrapped `Ptr` whose size *equals* that of the given `ForeignPtr`.

```
withForeignPtr :: fp:ForeignPtr a
               -> (PtrN a (fplen fp) -> IO b)
               -> IO b
```

This is a rather interesting *higher-order* specification. Consider a call `withForeignPtr fp act`. If the `act` requires a `Ptr` whose size *exceeds* that of `fp` then LiquidHaskell will flag a (subtyping) error indicating the overflow. If instead the `act` requires a buffer of size less than `fp` then via contra-variant function subtyping, the input type of `act` will be widened to the large size, and the code will be accepted.

To REFINE READS AND WRITES we specify that they can only be done if the pointer refers to a non-empty (remaining) buffer. That is, we define an alias:

```
type OkPtr a = {v:Ptr a | 0 < plen v}
```

that describes pointers referring to *non-empty* buffers (of strictly positive `plen`), and then use the alias to refine:

```
peek :: OkPtr a -> IO a
poke :: OkPtr a -> a -> IO ()
```

In essence the above type says that no matter how arithmetic was used to shift pointers around, when the actual dereference happens, the size "remaining" after the pointer must be non-negative (so that a byte can be safely read from or written to the underlying buffer.)

To REFINE THE SHIFT operations, we simply check that the pointer *remains* within the bounds of the buffer, and update the `plen` to reflect the size remaining after the shift:

[0] This signature precludes "left" or "backward" shifts; for that there is an analogous 'minusPtr' which we elide for simplicity

```
plusPtr :: p:Ptr a -> off:NatLE (plen p) -> PtrN b (plen p - off)
```

using the alias `NatLE`, defined as:

```
type NatLE N = {v:Nat | v <= N}
```

[9] The alert reader will note that we have strengthened the type of 'plusPtr' to prevent the pointer from wandering outside the boundary of the buffer. We could instead use a weaker requirement for 'plusPtr' that omits this requirement, and instead have the error be flagged when the pointer was used to read or write memory.

Types Prevent Overflows Lets revisit the zero-fill example from above to understand how the refinements help detect the error:

```
exBad = do fp <- mallocForeignPtrBytes 4
           withForeignPtr fp $ \p -> do
             poke (p `plusPtr` 0) zero
             poke (p `plusPtr` 1) zero
             poke (p `plusPtr` 2) zero
             poke (p `plusPtr` 5) zero
           return fp
        where
           zero = 0 :: Word8
```

Lets read the tea leaves to understand the above error:

```
  Error: Liquid Type Mismatch
   Inferred type
     VV : {VV : Int | VV == ?a && VV == 5}

   not a subtype of Required type
     VV : {VV : Int | VV <= plen p}

   in Context
     zero : {zero : Word8 | zero == ?b}
     VV   : {VV : Int | VV == ?a && VV == (5  :  int)}
     fp   : {fp : ForeignPtr a | fplen fp == ?c && 0 <= fplen fp}
     p    : {p  : Ptr a | fplen fp == plen p && ?c <= plen p && ?b <= plen p && zero <= plen p}
     ?a   : {?a : Int | ?a == 5}
     ?c   : {?c : Int | ?c == 4}
     ?b   : {?b : Integer | ?b == 0}
```

The error says we're bumping `p` up by `VV == 5` using `plusPtr` but the latter *requires* that bump-offset be within the size of the buffer referred to by `p`, i.e. `VV <= plen p`. Indeed, in this context, we have:

```
     p    : {p : Ptr a | fplen fp == plen p && ?c <= plen p && ?b <= plen p && zero <= plen p}
     fp   : {fp : ForeignPtr a | fplen fp == ?c && 0 <= fplen fp}
```

that is, the size of `p`, namely `plen p` equals the size of `fp`, namely `fplen fp` (thanks to the `withForeignPtr` call), and finally the latter is equal to `?c` which is `4` bytes. Thus, since the offset `5` is not less than the buffer size `4`, LiquidHaskell cannot prove that the call to `plusPtr` is safe, hence the error.

Assumptions vs Guarantees

At this point you ought to wonder: where is the *code* for `peek`, `poke` or `mallocForeignPtrBytes` and so on? How can we know that the types we assigned to them are in fact legitimate?

Frankly, we cannot as those functions are *externally* implemented (in this case, in `C`), and hence, invisible to the otherwise all-seeing eyes of LiquidHaskell. Thus, we are *assuming* or *trusting* that those functions behave according to their types. Put another way, the types for the low-level API are our *specification* for what low-level pointer safety. We shall now *guarantee* that the higher level modules that build upon this API in fact use the low-level function in a manner consistent with this specification.

Assumptions are a Feature and not a bug, as they let us to verify systems that use some modules for which we do not have the code. Here, we can *assume* a boundary specification, and then *guarantee* that the rest of the system is safe with respect to that specification.

[0] If we so desire, we can also *check* the boundary specifications at run-time, but that is outside the scope of LiquidHaskell.

ByteString API

Next, lets see how the low-level API can be used to implement to implement ByteStrings, in a way that lets us perform fast string operations without opening the door to overflows.

A ByteString is implemented as a record

```
data ByteString = BS {
    bPtr :: ForeignPtr Word8
  , bOff :: !Int
  , bLen :: !Int
  }
```

comprising

- a *pointer* `bPtr` to a contiguous block of memory,
- an *offset* `bOff` that denotes the position inside the block where the string begins, and
- a *length* `bLen` that denotes the number of bytes (from the offset) that belong to the string.

These entities are illustrated in Figure~9.2; the green portion represents the actual contents of a particular `ByteString`. This representation makes it possible to implement various operations like

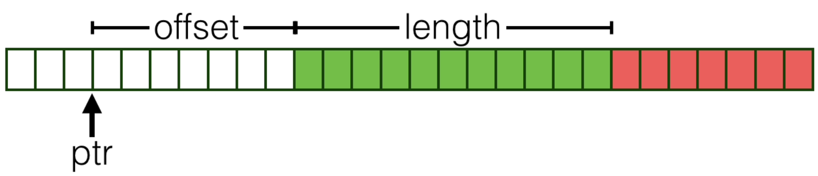

Figure 9.2: Representing ByteStrings in memory.

computing prefixes and suffixes extremely quickly, simply by pointer arithmetic.

In a Legal ByteString the *start* (`bOff`) and *end* (`bOff + bLen`) offsets lie inside the buffer referred to by the pointer `bPtr`. We can formalize this invariant with a data definition that will then make it impossible to create illegal `ByteString`s:

```
{-@ data ByteString = BS {
      bPtr :: ForeignPtr Word8
    , bOff :: {v:Nat| v        <= fplen bPtr}
    , bLen :: {v:Nat| v + bOff <= fplen bPtr}
    }
  @-}
```

The refinements on `bOff` and `bLen` correspond exactly to the legality requirements that the start and end of the `ByteString` be *within* the block of memory referred to by `bPtr`.

For brevity lets define an alias for `ByteString`s of a given size:

```
{-@ type ByteStringN N = {v:ByteString | bLen v = N} @-}
```

Legal Bytestrings can be created by directly using the constructor, as long as we pass in suitable offsets and lengths. For example,

```
{-@ good1 :: IO (ByteStringN 5) @-}
good1 = do fp <- mallocForeignPtrBytes 5
           return (BS fp 0 5)
```

creates a valid `ByteString` of size `5`; however we need not start at the beginning of the block, or use up all the buffer, and can instead do:

```
{-@ good2 :: IO (ByteStringN 2) @-}
good2 = do fp <- mallocForeignPtrBytes 5
           return (BS fp 3 2)
```

Note that the length of good2 is just 2 which is *less than* allocated size 5.

ILLEGAL BYTESTRINGS are rejected by LiquidHaskell. For example, bad1's length is rather more than the buffer size, and is flagged as such:

```
bad1 = do fp <- mallocForeignPtrBytes 3
          return (BS fp 0 10)
```

Similarly, bad2 does have 2 bytes but *not* if we start at the offset of 2:

```
bad2 = do fp <- mallocForeignPtrBytes 3
          return (BS fp 2 2)
```

EXERCISE 9.1. [FIX THE BYTESTRING] Modify the definitions of bad1 and bad2 so they are *accepted* by LiquidHaskell.

TO FLEXIBLY BUT SAFELY CREATE a ByteString the implementation defines a higher order create function, that takes a size n and accepts a fill action, and runs the action after allocating the pointer. After running the action, the function tucks the pointer into and returns a ByteString of size n.

```
{-@ create :: n:Nat -> (Ptr Word8 -> IO ()) -> ByteStringN n @-}
create n fill = unsafePerformIO $ do
  fp  <- mallocForeignPtrBytes n
  withForeignPtr fp fill
  return (BS fp 0 n)
```

EXERCISE 9.2. [CREATE] ⋆ Why does LiquidHaskell *reject* the following function that creates a ByteString corresponding to "GHC"?

```
bsGHC = create 3 $ \p -> do
  poke (p `plusPtr` 0) (c2w 'G')
  poke (p `plusPtr` 1) (c2w 'H')
  poke (p `plusPtr` 2) (c2w 'C')
```

Hint: The function writes into 3 slots starting at p. How big should plen p be to allow this? What type does LiquidHaskell infer for p above? Does it meet the requirement? Which part of the *specification* or *implementation* needs to be modified so that the relevant information about p becomes available within the do-block above? Make sure you figure out the above before proceeding.

To 'PACK' a `String` into a `ByteString` we simply call `create` with the appropriate fill action:

[0] The code uses 'create'' which is just 'create' with the *correct* signature in case you want to skip the previous exercise. (But don't!)

```
pack str      =  create' n $ \p -> go p xs
  where
  n           = length str
  xs          = map c2w str
  go p (x:xs) = poke p x >> go (plusPtr p 1) xs
  go _ []     = return  ()
```

EXERCISE 9.3. [PACK] We can compute the size of a `ByteString` by using the function:

Fix the specification for `pack` so that (it still typechecks!) and furthermore, the following QuickCheck style *property* is proved by LiquidHaskell:

```
{-@ prop_pack_length  :: [Char] -> {v:Bool | Prop v} @-}
prop_pack_length xs   = bLen (pack xs) == length xs
```

Hint: Look at the type of `length`, and recall that `len` is a numeric measure denoting the size of a list.

THE MAGIC OF INFERENCE ensures that `pack` just works. Notice there is a tricky little recursive loop `go` that is used to recursively fill in the `ByteString` and actually, it has a rather subtle type signature that LiquidHaskell is able to automatically infer.

EXERCISE 9.4. ⋆ Still, we're here to learn, so can you *write down* the type signature for the loop so that the below variant of `pack` is accepted by LiquidHaskell (Do this *without* cheating by peeping at the type inferred for `go` above!)

```
packEx str     = create' n $ \p -> pLoop p xs
  where
  n            = length str
  xs           = map c2w str

{-@ pLoop      :: (Storable a) => p:Ptr a -> xs:[a] -> IO () @-}
pLoop p (x:xs) = poke p x >> pLoop (plusPtr p 1) xs
pLoop _ []     = return ()
```

Hint: Remember that `len xs` denotes the size of the list `xs`.

EXERCISE 9.5. ['UNSAFETAKE' AND 'UNSAFEDROP'] respectively extract the prefix and suffix of a `ByteString` from a given position.

They are really fast since we only have to change the offsets. But why does LiquidHaskell reject them? Can you fix the specifications so that they are accepted?

```
{-@ unsafeTake          :: n:Nat -> b:ByteString -> ByteStringN n @-}
unsafeTake n (BS x s _) = BS x s n

{-@ unsafeDrop          :: n:Nat -> b:ByteString -> ByteStringN {bLen b - n} @-}
unsafeDrop n (BS x s l) = BS x (s + n) (l - n)
```

Hint: Under what conditions are the returned ByteStrings legal?

To 'unpack' a ByteString into a plain old String, we essentially run pack in reverse, by walking over the pointer, and reading out the characters one by one till we reach the end:

```
unpack              :: ByteString -> String
unpack (BS _  _ 0)  = []
unpack (BS ps s l)  = unsafePerformIO $ withForeignPtr ps $ \p ->
    go (p `plusPtr` s) (l - 1) []
  where
    {-@ go     :: p:_ -> n:_ -> acc:_ -> IO {v:_ | true } @-}
    go p 0 acc = peek p >>= \e -> return (w2c e : acc)
    go p n acc = peek (p `plusPtr` n) >>=  \e -> go p (n-1) (w2c e : acc)
```

EXERCISE 9.6. [UNPACK] ⋆ Fix the specification for unpack so that the below QuickCheck style property is proved by LiquidHaskell.

```
{-@ prop_unpack_length :: ByteString -> {v:Bool | Prop v} @-}
prop_unpack_length b   = bLen b  == length (unpack b)
```

Hint: You will also have to fix the specification of the helper go. Can you determine the output refinement should be (instead of just true?) How *big* is the output list in terms of p, n and acc.

Application API

Finally, lets revisit our potentially "bleeding" chop function to see how the refined ByteString API can prevent errors.
The signature specifies that the prefix size n must be less than the size of the input string s.

```
{-@ chop :: s:String -> n:NatLE (len s) -> String @-}
chop s n = s'
  where
    b    = pack s          -- down to low-level
    b'   = unsafeTake n b  -- grab n chars
    s'   = unpack b'       -- up to high-level
```

Overflows are prevented by LiquidHaskell, as it rejects calls to `chop` where the prefix size is too large (which is what led to the overflow that spilled the contents of memory after the string, as illustrated in Figure~9.1). Thus, in the code below, the first use of `chop` which defines `ex6` is accepted as `6 <= len ex` but the second call is rejected because `30 > len ex`.

```
demo     = [ex6, ex30]
  where
    ex   = ['L','I','Q','U','I','D']
    ex6  = chop ex 6   -- accepted by LH
    ex30 = chop ex 30  -- rejected by LH
```

Exercise 9.7. [Chop] Fix the specification for `chop` so that the following property is proved:

```
{-@ prop_chop_length  :: String -> Nat -> {v:Bool | Prop v} @-}
prop_chop_length s n
  | n <= length s     = length (chop s n) == n
  | otherwise         = True
```

Nested ByteStrings

For a more in-depth example, let's take a look at `group`, which transforms strings like

`"foobaaar"`

into *lists* of strings like

`["f","oo", "b", "aaa", "r"]`.

The specification is that `group` should produce a

1. list of *non-empty* `ByteStrings`,
2. the *sum of* whose lengths equals that of the input string.

Non-empty ByteStrings are those whose length is non-zero:

```
{-@ type ByteStringNE = {v:ByteString | bLen v /= 0} @-}
```

We can use these to define enrich the ByteString API with a `null` check

```
{-@ null               :: b:ByteString -> {v:Bool | Prop v <=> bLen b == 0} @-}
null (BS _ _ l)        = l == 0
```

This check is used to determine if it is safe to extract the head and tail of the `ByteString`. Notice how we can use refinements to ensure the safety of the operations, and also track the sizes.

° 'peekByteOff p i' is equivalent to 'peek (plusPtr p i)'

```
{-@ unsafeHead        :: ByteStringNE -> Word8 @-}
unsafeHead (BS x s _) = unsafePerformIO $
                          withForeignPtr x $ \p ->
                            peekByteOff p s

{-@ unsafeTail         :: b:ByteStringNE -> ByteStringN {bLen b - 1} @-}
unsafeTail (BS ps s l) = BS ps (s + 1) (l - 1)
```

THE 'GROUP' function recursively calls `spanByte` to carve off the next group, and then returns the accumulated results:

```
{-@ group :: b:ByteString -> {v: [ByteStringNE] | bLens v = bLen b} @-}
group xs
    | null xs   = []
    | otherwise = let  y        = unsafeHead xs
                       (ys, zs) = spanByte y (unsafeTail xs)
                  in (y `cons` ys) : group zs
```

The first requirement, that the groups be non-empty is captured by the fact that the output is a `[ByteStringNE]`. The second requirement, that the sum of the lengths is preserved, is expressed by a writing a numeric measure:

```
{-@ measure bLens @-}
bLens        :: [ByteString] -> Int
bLens []     = 0
bLens (b:bs) = bLen b + bLens bs
```

'SPANBYTE' does a lot of the heavy lifting. It uses low-level pointer arithmetic to find the *first* position in the `ByteString` that is different from the input character `c` and then splits the `ByteString` into a pair comprising the prefix and suffix at that point.

```
{-@ spanByte :: Word8 -> b:ByteString -> ByteString2 b @-}
spanByte c ps@(BS x s l) = unsafePerformIO $ withForeignPtr x $ \p ->
    go  (p `plusPtr` s) 0
  where
    go p i | i >= l    = return (ps, empty)
           | otherwise = do c' <- peekByteOff p i
                            if c /= c'
                                then return (unsafeTake i ps, unsafeDrop i ps)
                                else go  p (i+1)
```

LiquidHaskell infers that `0 <= i <= l` and therefore that all of the memory accesses are safe. Furthermore, due to the precise specifications given to `unsafeTake` and `unsafeDrop`, it is able to prove that the output pair's lengths add up to the size of the input `ByteString`.

```
{-@ type ByteString2 B = {v:_ | bLen (fst v) + bLen (snd v) = bLen B} @-}
```

Recap: Types Against Overflows

In this chapter we saw a case study illustrating how measures and refinements enable safe low-level pointer arithmetic in Haskell. The take away messages are:

1. larger systems are *composed of* layers of smaller ones,
2. we can write *refined APIs* for each layer,
3. that can be used to inform the *design* and ensure *correctness* of the layers above.

We saw this in action by developing a low-level `Pointer` API, using it to implement fast `ByteStrings` API, and then building some higher-level functions on top of the `ByteStrings`.

THE TRUSTED COMPUTING BASE in this approach includes exactly those layers for which the code is *not* available, for example, because they are implemented outside the language and accessed via the FFI as with `mallocForeignPtrBytes` and `peek` and `poke`. In this case, we can make progress by *assuming* the APIs hold for those layers and verify the rest of the system with respect to that API. It is important to note that in the entire case study, it is only the above FFI signatures that are *trusted*; the rest are all verified by LiquidHaskell.