RANJIT JHALA, ERIC SEIDEL, NIKI VAZOU

PROGRAMMING WITH REFINEMENT TYPES

AN INTRODUCTION TO LIQUIDHASKELL

GOTO.UCSD.EDU/LIQUID

Contents

List of Exercises

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:[1]

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

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.

[1] We could write `average` more *defensively*, returning a `Maybe` or `Either` value. However, this merely kicks the can down the road. Ultimately, we will want to extract the `Int` from the `Maybe` and if the inputs were invalid to start with, then at that point we'd be stuck.

```
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:[2]

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

[2] 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:[3]

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

[3] 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.[4]

[4] 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) binary 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.

```
$ git clone https://github.com/ucsd-progsys/liquidhaskell-tutorial.git
$ cd liquidhaskell-tutorial/src
```

Note: This tutorial is a *work in progress*, and we will be **very** grateful for feedback and suggestions, ideally via pull-requests on github. Lets begin!

2
Logic & SMT

As we shall see shortly, a refinement type is:

Refinement Types = Types + Logical Predicates

Let us begin by quickly recalling what we mean by "logical predicates" in the remainder of this tutorial.[1] To this end, we will describe *syntax*, that is, what predicates *look* like, and *semantics*, which is a fancy word for what predicates *mean*.

[1] If you are comfortable with this material, e.g. if you know what the "S", "M" and "T" stand for in SMT, and what QF-UFLIA stands for i.e. the quantifier free theory of linear arithmetic and uninterpreted functions, then feel free skip to the next chapter.

Syntax

A *logical predicate* is, informally speaking, a Boolean valued term drawn from a *restricted* subset of Haskell. In particular, the expressions are drawn from the following grammar comprising *constants*, *expressions* and *predicates*.

A Constant `c` is simply one of the numeric values:

```
c := 0, 1, 2, ...
```

A Variable `v` is one of `x`, `y`, `z`, etc., these will refer to (the values of) binders in our source programs.

```
v := x, y, z, ...
```

An Expression `e` is one of the following forms; that is, an expression is built up as linear arithmetic expressions over variables and constants and uninterpreted function applications.

```
e := v                  -- variable
   | c                  -- constant
   | e + e              -- addition
   | e - e              -- subtraction
   | c * e              -- linear multiply
   | v e1 e2 ... en     -- unint. func. appl.
```

EXAMPLES OF EXPRESSIONS include the following:

- `x + y - z`
- `2 * x`
- `1 + size x`

A RELATION is one of the usual (arithmetic) comparison operators:

```
r := ==              -- equality
   | /=              -- disequality
   | >=              -- greater than or equal
   | <=              -- less than or equal
   | >               -- greater than
   | <               -- less than
```

A PREDICATE is either an atomic predicate, obtained by comparing two expressions, or, an application of a predicate function to a list of arguments, or the Boolean combination of the above predicates with the operators `&&` (and), `||` (or), `==>` (implies[2]), `<=>` (if and only if[3]), and `not`.

[2] Read `p ==> q` as "if `p` then `q`"

[3] Read `p <=> q` as "if `p` then `q` **and** if `q` then `p`"

```
p := true
   | false
   | e r e           -- atomic binary relation
   | v e1 e2 ... en  -- predicate application
   | p  && p         -- and
   | p  || p         -- or
   | p ==> p         -- implies
   | p <=> p         -- if and only if
   | not p           -- negation
```

EXAMPLES OF PREDICATES include the following:

- `x + y <= 3`
- `null x`
- `x < 10 ==> y < 10 ==> x + y < 20`
- `0 < x + y <=> 0 < y + x`

Semantics

The syntax of predicates tells us what they *look* like, that is, what we can *write down* as valid predicates. Next, let us turn our attention to what a predicate *means*. Intuitively, a predicate is just a Boolean valued Haskell function – `&&`, `||`, `not` are the usual operators and `==>` and `<=>` are two special operators.

The Implication operator ==> is equivalent to the Haskell function:

```
(==>) :: Bool -> Bool -> Bool
False ==> False = True
False ==> True  = True
True  ==> True  = True
True  ==> False = False
```

The If-and-only-if operator <=> is equivalent to the Haskell function:

```
(<=>)  :: Bool -> Bool -> Bool
False <=> False = True
False <=> True  = False
True  <=> True  = True
True  <=> False = False
```

An Environment is a mapping from variables to their Haskell types. For example, the environment `G` defined

```
x :: Int
y :: Int
z :: Int
```

maps each variable `x`, `y` and `z` to the type `Int`.

An Assignment under an environment, is a mapping from variables to values of the type specified in the environment. For example,

```
x := 1
y := 2
z := 3
```

is an assignment under `G` that maps `x`, `y` and `z` to the `Int` values `1`, `2` and `3` respectively.

A Predicate Evaluates to either `True` or `False` under a given assignment. For example, the predicate

```
x + y > 10
```

evaluates to `False` given the above assignment but evaluates to `True` under the assignment

```
x := 10
y := 10
```

A PREDICATE IS SATISFIABLE in an environment if *there exists* an assignment (in that environment) that makes the predicate evaluate to `True`. For example, in `G` the predicate

```
x + y == z
```

is satisfiable, as the above assignment makes the predicate evaluate to `True`.

A PREDICATE IS VALID in an environment if *every* assignment in that environment makes the predicate evaluate to `True`. For example, the predicate

```
x < 10 || x == 10 || x > 10
```

is valid under `G` as no matter what value we assign to `x`, the above predicate will evaluate to `True`.

Verification Conditions

LiquidHaskell works without actually *executing* your programs. Instead, it checks that your program meets the given specifications in roughly two steps.

1. First, LH combines the code and types down to a set of *Verification Conditions* (VC) which are predicates that are valid *only if* your program satisfies a given property.[4]
2. Next, LH *queries* an SMT solver to determine whether these VCs are valid. If so, it says your program is *safe* and otherwise it *rejects* your program.

[4] The process is described at length in this paper

THE SMT SOLVER DECIDES whether a predicate (VC) is valid *without enumerating* and evaluating all assignments. Indeed, it is impossible to do so as there are usually infinitely many assignments once the predicates refer to integers or lists and so on. Instead, the SMT solver uses a variety of sophisticated *symbolic algorithms* to deduce whether a predicate is valid or not. This somewhat magical process is the result of decades of work in mathematical logic and decision procedures; the Ph.D thesis of Greg Nelson is an excellent place to learn more about these beautiful algorithms.

WE RESTRICT THE LOGIC to ensure that all our VC queries fall within the *decidable fragment*. This makes LiquidHaskell extremely automatic – there is *no* explicit manipulation of proofs, just the specification of properties via types and of course, the implementation via Haskell

code! This automation comes at a price: all our refinements *must* belong to the logic above. Fortunately, with a bit of creativity, we can say a *lot* in this logic.[5]

[5] In particular, we will use the uninterpreted functions to create many sophisticated abstractions.

Examples: Propositions

Finally, lets conclude this quick overview with some examples of predicates, in order to build up our own intuition about logic and validity. Each of the below is a predicate from our refinement logic. However, we write them as raw Haskell expressions that you may be more familiar with right now, and so that we can start to use LiquidHaskell to determine whether a predicate is indeed valid or not.

Let 'TRUE' be a refined type for `Bool` valued expressions that *always* evaluate to `True`. Similarly, we can define `FALSE` for `Bool` valued expressions that *always* evaluate to `False`:

```
{-@ type TRUE  = {v:Bool | Prop v}       @-}
{-@ type FALSE = {v:Bool | not (Prop v)} @-}
```

Thus, a *valid predicate* is one that has the type `TRUE`. The simplest example of a valid predicate is just `True`:

```
{-@ ex0 :: TRUE @-}
ex0 = True
```

of course, `False` is *not valid*

```
{-@ ex0' :: TRUE @-}
ex0' = False
```

We can get more interesting predicates if we use variables. For example, the following is valid predicate says that a `Bool` variable is either `True` or `False`.

```
{-@ ex1 :: Bool -> TRUE @-}
ex1 b = b || not b
```

Of course, a variable cannot be both `True` and `False`, and so the below predicate is invalid:

```
{-@ ex2 :: Bool -> FALSE @-}
ex2 b = b && not b
```

The next few examples illustrate the `==>` operator. You should read `p ==> q` as *if* `p` is true *then* `q` must also be true. Thus, the below

predicates are valid as if both `a` and `b` are true, then well, `a` is true, and `b` is true.

```
{-@ ex3 :: Bool -> Bool -> TRUE @-}
ex3 a b = (a && b) ==> a

{-@ ex4 :: Bool -> Bool -> TRUE @-}
ex4 a b = (a && b) ==> b
```

Exercise 2.1 (Implications and Or). *Of course, if we replace the* `&&` *with* `||` *the result is* not valid. *Can you shuffle the variables around* – without changing the operators – *to make the formula valid?*

```
{-@ ex3' :: Bool -> Bool -> TRUE @-}
ex3' a b = (a || b) ==> a
```

The following predicates are valid because they encode modus ponens: if you know that `a` implies `b` and you know that `a` is true, then it must be that `b` is also true:

```
{-@ ex6 :: Bool -> Bool -> TRUE @-}
ex6 a b = (a && (a ==> b)) ==> b

{-@ ex7 :: Bool -> Bool -> TRUE @-}
ex7 a b = a ==> (a ==> b) ==> b
```

Recall that `p <=> q` (read `p` if and only iff `q`) evaluates to `True` exactly when `p` and `q` evaluate to the *same* values (`True` or `False`). It is used to encode *equalities* between predicates. For example, we can write down De Morgan's laws as the valid predicates:

```
{-@ exDeMorgan1 :: Bool -> Bool -> TRUE @-}
exDeMorgan1 a b = not (a || b) <=> (not a && not b)
```

Exercise 2.2 (DeMorgan's Law). *The following version of DeMorgan's law is wrong. Can you fix it to get a valid formula?*

```
{-@ exDeMorgan2 :: Bool -> Bool -> TRUE @-}
exDeMorgan2 a b = not (a && b) <=> (not a && not b)
```

Examples: Arithmetic

Next, lets look at some predicates involving arithmetic. The simplest ones don't have any variables, for example:

```
{-@ ax0 :: TRUE @-}
ax0 = 1 + 1 == 2
```

Again, a predicate that evaluates to `False` is not valid:

```
{-@ ax0 :: TRUE @-}
ax0' = 1 + 1 == 3
```

SMT Solvers determine Validity *without* enumerating assignments. For example, consider the predicate:

```
{-@ ax1 :: Int -> TRUE @-}
ax1 x = x < x + 1
```

It is trivially valid; as via the usual laws of arithmetic, it is equivalent to `0 < 1` which is `True` independent of the value of `x`. The SMT solver is able to determine this validity without enumerating the infinitely many possible values for `x`. This kind of validity checking lies at the heart of LiquidHaskell.

We can combine arithmetic and propositional operators, as shown in the following examples:

```
{-@ ax2 :: Int -> TRUE @-}
ax2 x = (x < 0) ==> (0 <= 0 - x)

{-@ ax3 :: Int -> Int -> TRUE @-}
ax3 x y = (0 <= x) ==> (0 <= y) ==> (0 <= x + y)

{-@ ax4 :: Int -> Int -> TRUE @-}
ax4 x y = (x == y - 1) ==> (x + 2 == y + 1)

{-@ ax5 :: Int -> Int -> Int -> TRUE @-}
ax5 x y z =   (x <= 0 && x >= 0)
          ==> (y == x + z)
          ==> (y == z)
```

Exercise 2.3 (Addition and Order)**.** *The formula below is* not *valid. Do you know why? Change the* hypothesis *i.e. the thing to the left of the* `==>` *to make it a valid formula.*

```
{-@ ax6 :: Int -> Int -> TRUE @-}
ax6 x y = True ==> (x <= x + y)
```

Examples: Uninterpreted Function

We say that function symbols are *uninterpreted* in the refinement logic, because the SMT solver does not "know" how functions are defined. Instead, the only thing that the solver knows is the *axiom of congruence* which states that any function `f`, returns equal outputs when invoked on equal inputs.

Let us define an uninterpreted function from `Int` to `Int`:

```
{-@ measure f :: Int -> Int @-}
```

We Test the Axiom of Congruence by checking that the following predicate is valid:

```
{-@ congruence :: Int -> Int -> TRUE @-}
congruence x y = (x == y) ==> (f x == f y)
```

Again, remember we are *not evaluating* the code above; indeed we *cannot* evaluate the code above because we have no definition of `f`. Still, the predicate is valid as the congruence axiom holds for any possible interpretation of `f`.

Here is a fun example; can you figure out why this predicate is indeed valid? (The SMT solver can. . .)

```
{-@ fx1 :: Int -> TRUE @-}
fx1 x =   (x == f (f (f x)))
      ==> (x == f (f (f (f (f x)))))
      ==> (x == f x)
```

To get a taste of why uninterpreted functions will prove useful lets write a function to compute the `size` of a list:

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

We can now verify that the following predicates are *valid*:

```
{-@ fx0 :: [a] -> [a] -> TRUE @-}
fx0 xs ys = (xs == ys) ==> (size xs == size ys)
```

Note that to determine that the above is valid, the SMT solver does not need to know the *meaning* or *interpretation* of `size` – merely that it is a function. When we need some information about the definition, of `size` we will put it inside the predicate. For example, in order to prove that the following is valid:

```
{-@ fx2 :: a -> [a] -> TRUE @-}
fx2 x xs = 0 < size ys
  where
    ys   = x : xs
```

LiquidHaskell actually asks the SMT solver to prove the validity of a VC predicate which states that sizes are non-negative and that since `ys` equals `x:xs`, the size of `ys` is one more than `xs`.[6]

[6] Fear not! We will describe how this works soon

```
fx2VC x xs ys =   (0 <= size xs)
              ==> (size ys == 1 + size xs)
              ==> (0 < size ys)
```

Recap

This chapter describes exactly what we, for the purposes of this book, mean by the term *logical predicate*.

1. We defined a grammar – a restricted subset of Haskell corresponding to `Bool` valued expressions.
2. The restricted grammar lets us use SMT solvers to decide whether a predicate is *valid* that is, evaluates to `True` for *all* values of the variables.
3. Crucially, the SMT solver determins validity *without enumerating* and evaluating the predicates (which would take forever!) but instead by using clever symbolic algorithms.

Next, lets see how we can use logical predicates to *specify* and *verify* properties of real programs.

3
Refinement Types

What is a Refinement Type? In a nutshell,

> *Refinement Types = Types + Predicates*

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

Defining Types

Let us define some refinement types:

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

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

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

To use these types we can write:

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

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

Errors

If we try to say nonsensical things like:

```
{-@ 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[2]

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

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

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 `&&`:

```
{-@ 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.[3]

[3] 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 `Strings` 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 3.1 (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.[4]

[4] 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 Liquid-Haskell 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 3.2 (Propositions). *What happens if you* delete *the type for* `isPositive` *? Can you* change *the type for* `isPositive` *(i.e. write some other type) while preserving safety?*

Exercise 3.3 (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 than 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.

4
Polymorphism

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

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

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

Lets attempt to look it up at various indices:

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

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

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

In a suitable Editor e.g. Vim or Emacs, or if you push the "play" button in the online demo, you will literally see the error *without* running the code. Lets see how LiquidHaskell checks `ok` and `yup` but flags `nono`, and along the way, learn how it 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}

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

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

ASSUMES are used to *specify* types describing the semantics of functions that we cannot verify e.g. because we don't have the code for them. Here, we are assuming that the library function `Data.Vector.length` indeed computes the size of the input vector. Furthermore, we are stipulating that the lookup function `(!)` requires an index that is 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 lets us write properties that connect *multiple* 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 `Vectors` of a given size `N` as:

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

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

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

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

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

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

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

Verification: Vector Lookup

Let's try 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 `Vectors`

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

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

Exercise 4.1 (Vector Head). *Replace the* `undefined` *with an* implementation *of* `head''` *which accepts* all `Vectors` *but returns a value only when the input* `vec` *is not empty.*

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

Exercise 4.2 (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 4.3 (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 4.4 (Guards). *What happens if you* replace *the guard with* `i <= sz`*?*

Exercise 4.5 (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*[1]

[1] 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 4.6 (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 4.7 (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 4.8 (Dot Product). *The following uses* `loop` *to compute* `dotProduct`*s. 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`).

The 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

```
{-@ 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.

5
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*.[1]

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

Data Invariants LiquidHaskell lets us enforce these invariants with a refined data definition:

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

Where, as before, we use the aliases:

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

REFINED DATA CONSTRUCTORS The refined data definition is internally converted into refined types for the data constructor `SP`:

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

In other words, by using refined input types for `SP` we have automatically converted it into a *smart* constructor that ensures that *every* instance of a `Sparse` is legal. Consequently, LiquidHaskell verifies:

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

but rejects, due to the invalid index:

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

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

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

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.

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

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 5.1 (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 write 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:*

Hint: You need to check that *all* the indices in `elts` are less than `dim`; the easiest way is to compute a new `Maybe [(Int, a)]` which is `Just` the original pairs if they are valid, and `Nothing` otherwise.

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

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

Exercise 5.2 (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 are 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 5.3 (Insertion Sort)**.** *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 5.4 (QuickSort). ⋆⋆ *Why is the following implementation of* `quickSort` *rejected by LiquidHaskell? Modify it so it is accepted.*

Hint: Think about how `append` should behave so that the `quickSort` has the desired property. That is, suppose that `ys` and `zs` are already in *increasing order*. Does that mean that `append x ys zs` are *also* in increasing order? No! What other requirement do you need? bottle that intuition into a suitable *specification* for `append` and then ensure that the code satisfies that specification.

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

{-@ append :: x:a -> IncList a
                  -> IncList a
                  -> IncList a
  @-}
append z Emp       ys = z :< ys
append z (x :< xs) ys = x :< append z 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 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 5.1 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:

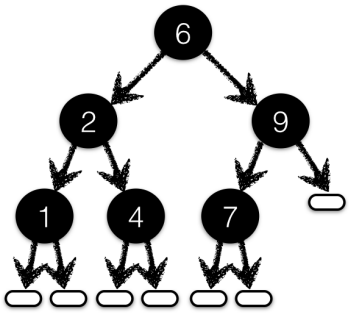

Figure 5.1: A Binary Search Tree with values between 1 and 9. Each root's value lies between the values appearing in its left and right subtrees.

```
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. The aliases BSTL and BSTR denote BSTs with values less than and greater than some X, respectively.[3]

[3] We could also just *inline* the definitions of BSTL and BSTR into that of BST but they will be handy later.

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

{-@ type BSTL a X = BST {v:a | v < X}             @-}
{-@ type BSTR a X = BST {v:a | X < 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 =  Node 66
             (Node 4
                 (Node 1 Leaf Leaf)
                 (Node 29 Leaf Leaf))  -- Out of order, rejected
```

```
            (Node 99
                (Node 77 Leaf Leaf)
                Leaf)
```

Exercise 5.5 (Duplicates). *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 Lets write a function that adds an element to a `BST`.[4]

[4] While writing this exercise I inadvertently swapped the `k` and `k'` which caused LiquidHaskell to protest.

```
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 For our next trick, 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:[5]

[5] 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 { mElt :: a, rest :: BST a }
```

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

```
{-@ data MinPair a = MP { mElt :: a, rest :: BSTR a mElt} @-}
```

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 warn ya!"
```

Exercise 5.6 (Delete). *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 _  Leaf           = Leaf
```

Exercise 5.7 (Safely Deleting Minimum). ⋆ *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 5.8 (BST Sort). *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 (Node x l r) = undefined
toIncList Leaf         = 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* (and *need not*) 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`.

6
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 the problem of ensuring the safety of division. Recall that we wrote:

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

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

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

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

However, it can be more of a challenge when the divisor is obtained *dynamically*. For example, 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 can now be described as the *subset* of plain old Haskell lists `[a]` for which the predicate `notEmpty` holds

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

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

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

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

AVERAGE is only sensible for non-empty lists. Happily, we can specify this using the refined `NEList` type:

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

Exercise 6.1 (Average, Maybe). *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 (sum xs) elems
  | otherwise = Nothing
  where
    elems     = size xs
    ok        = elems > 0 -- What expression goes here?
```

Exercise 6.2 (Debugging Specifications). *An important aspect of formal verifiers like LiquidHaskell is that they help establish properties not just of your* implementations *but equally, or more importantly, of your* specifications. *In that spirit, can you explain why the following two variants of* `size` *are* rejected *by LiquidHaskell?*

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

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

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 crash otherwise.

HEAD AND TAIL are two of the canonical *dangerous* functions, that only work on non-empty lists, and burn horribly otherwise. We can type them simple 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 uses the precondition to deduce that the second equations are *dead code*. Of course, this requires us to establish that *callers* of `head` and `tail` only invoke the respective functions with non-empty lists.

Exercise 6.3 (Safe Head). *Write down a specification for* `null` *such that* `safeHead` *is verified. Do* not *force* `null` *to only take non-empty inputs, that defeats the purpose. Instead, it's type should say that it works on* all *lists and returns* `True` if and only if *the input is non-empty.*

Hint: You may want to refresh your memory about implies `==>` and `<=>` from the chapter on logic.

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

{-@ null      :: [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*.

To ELIMINATE STUTTERING from a string, we can use `groupEq` to split the string into blocks of repeating `Chars`, and then just extract the first `Char` from each block:

```
-- >>> 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.

FOLDR1 is one of my favorite folds; it 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)
```

To SUM a non-empty list of numbers, we can just perform a `foldr1` with the `+` operator: 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]    -- is accepted by LH, but

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

Exercise 6.4 (Weighted Average). *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 6.5 (Mitchell's Risers). *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, popularized by Neil Mitchell.* `safeSplit` *requires its input be non-empty; but LiquidHaskell believes that the call inside* `risers` *fails this requirement. Fix the specification for* `risers` *so that it is verified.*

```
{-@ risers   :: (Ord a) => xs:[a] -> {v: [[a]] | notEmpty xs => notEmpty v} @-}
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"
```

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!

7
Numeric Measures

Many of the programs we have seen so far, for example those in here, suffer from *indexitis*. This is a term coined by Richard Bird which describes a tendency to perform low-level manipulations to iterate over the indices into a collection, opening the door to various off-by-one errors. Such errors can be eliminated by instead programming at a higher level, using a wholemeal approach where the emphasis is on using aggregate operations, like `map`, `fold` and `reduce`.

Wholemeal programming is no panacea as it still requires us to take care when operating on *different* collections; if these collections are *incompatible*, e.g. have the wrong dimensions, then we end up with a fate worse than a crash, a possibly meaningless result. Fortunately, LiquidHaskell can help. Lets see how we can use measures to specify dimensions and create a dimension-aware API for lists which can be used to implement wholemeal dimension-safe APIs.[1]

[1] 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)
```

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

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

MATRIX MULTIPLICATION can similarly be expressed in a high-level, wholemeal fashion, by eschewing low level index manipulations in favor of a high-level *iterator* over the `Matrix` elements:

```
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
```

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

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

WHOLEMEAL PROGRAMMING FREES us from having to fret about low-level index range manipulation, but is hardly a panacea. Instead, we must now think carefully about the *compatibility* of the various aggreates. 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.[2]

[2] 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.[3]

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

MEASURES are ideal for this task. Previously we saw how we could lift Haskell functions up to the refinement logic. Lets write a measure to describe the length of a list:[4]

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

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

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

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

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

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

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

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

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.

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

To MAKE SIGNATURES SYMMETRIC lets define an alias for plain old (unrefined) lists:

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

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

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

Lists: Size Preserving API

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

Exercise 7.1 (Map). MAP *yields a list with the same size as the input. Fix the specification of* `map` *so that the* `prop_map` *is verified.*

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

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

Exercise 7.2 (Reverse). ⋆ *We can* `reverse` *the elements of a list as shown below, using the tail recursive function* `go`*. Fix the signature for* `go` *so that LiquidHaskell can prove the specification for* `reverse`*.*

Hint: How big is the list returned by `go`?

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

zipWith requires both lists to have the *same* size, and produces a list with that same size.[5]

[5] 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.

```
{-@ zipWith :: (a -> b -> c) -> xs:List a
                             -> ListX b xs
                             -> ListX c 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)] | Tinier v as bs} @-}
zip (a:as) (b:bs) = (a, b) : zip as bs
zip [] _          = []
zip _  []         = []
```

The output type uses the predicate `Tinier Xs Ys Zs` which defines the length of `Xs` to be the smaller of that of `Ys` and `Zs`.[6]

[6] In logic, `if p then q else r` is the same as `p => q && not p => r`.

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

Exercise 7.3 (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 `=>`.

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 -> ListGE a n -> ListN a n @-}
take' 0 _      = []
take' n (x:xs) = x : take' (n-1) xs
take' _ _      = die "won't  happen"
```

The alias `ListGE a n` denotes lists whose length is at least `n`:

```
{-@ type ListGE a N = {v:List a | N <= size v} @-}
```

Exercise 7.4 (Drop). `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 7.5 (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"]        ]
```

THE PARTITION function breaks a list into two sub-lists of elements that either satisfy or fail a user supplied predicate.

```
partition          :: (a -> Bool) -> [a] -> ([a], [a])
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 fst @-}
fst  (x, _) = x

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

We can now refine the type of `partition` as:

```
{-@ partition :: _ -> xs:_ -> {v:_ | Sum2 v (size xs)} @-}
```

where `Sum2 V N` holds for a pair of lists dimensions add to `N`:

```
{-@ predicate Sum2 X N = size (fst X) + size (snd X) = N @-}
```

Exercise 7.6 (QuickSort). *Use* `partition` *to implement* `quickSort`.

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

{-@ quickSort    :: (Ord a) => xs:List a -> ListX a 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 }        @-}
```

When `vDim` is used a selector function, it returns the `vDim` field of `x`.

```
{-@ vDim :: x:_ -> {v: Nat | v = vDim x} @-}
```

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
```

As usual, it will be handy to have a few aliases.

```
-- | Non Empty Vectors
{-@ type VectorNE a  = {v:Vector a | vDim v > 0} @-}

-- | Vectors of size N
{-@ type VectorN a N = {v:Vector a | vDim v = N} @-}

-- | Vectors of Size Equal to Another Vector X
{-@ type VectorX a X = VectorN a {vDim X}        @-}
```

To CREATE a `Vector` safely, we can start with the empty vector `vEmp` and then add elements one-by-one with `vCons`:

```
{-@ vEmp :: VectorN a 0 @-}
vEmp = V 0 []

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

To ACCESS vectors at a low-level, we can use equivalents of *head* and *tail*, which only work on non-empty `Vectors`:

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

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

To ITERATE over a vector we can use the `for` combinator:

```
{-@ for        :: x:Vector a -> (a -> b) -> VectorX b x @-}
for (V n xs) f = V n (map f xs)
```

BINARY POINTWISE OPERATIONS should only be applied to *compatible* vectors, i.e. vectors with equal dimensions. We can write a generic binary pointwise operator:

```
{-@ vBin :: (a -> b -> c) -> x:Vector a
                          -> VectorX b x
                          -> VectorX c x
  @-}
vBin op (V n xs) (V _ ys) = V n (zipWith op xs ys)
```

THE DOT PRODUCT of two `Vectors` can be now implemented in a wholemeal *and* dimension safe manner, as:

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

Exercise 7.7 (Vector Constructor). *Complete the* specification *and* implementation *of* `vecFromList` *which* creates *a* `Vector` *from a plain 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]
```

Exercise 7.8 (Flatten). ⋆ *Write a function to* `flatten` *a nested* `Vector`.

```
{-@ flatten :: n:Nat
            -> m:Nat
            -> VectorN (VectorN a m) n
            -> VectorN a {m * n}
  @-}
flatten = undefined
```

THE CROSS PRODUCT of two vectors can now be computed in a nice wholemeal style, by a nested iteration followed by a `flatten`.

```
{-@ product   :: xs:Vector _
              -> ys:Vector _
              -> VectorN _ {vDim xs * vDim ys}
  @-}
product xs ys = flatten (vDim ys) (vDim xs) xys
  where
    xys       = for ys $ \y ->
                  for xs $ \x ->
                    x * y
```

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 | Dims v R C } @-}
{-@ predicate Dims M R C = mRow M = R && mCol M = C   @-}
```

For example, we can use the above to write type:

```
{-@ ok23 :: MatrixN _ 2 3 @-}
ok23     = M 2 3 (V 2 [ V 3 [1, 2, 3]
                      , V 3 [4, 5, 6] ])
```

Exercise 7.9 (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 7.10 (Matrix Constructor). ⋆ *Write a function to construct a* `Matrix` *from a nested list.*

```
matFromList      :: [[a]] -> Maybe (Matrix a)
matFromList []   = Nothing
matFromList xss@(xs:_)
  | ok           = Just (M r c vs)
  | otherwise    = Nothing
  where
    r            = size xss
    c            = size xs
    ok           = undefined
    vs           = undefined
```

Exercise 7.11 (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 Finally, lets implement matrix multiplication. You'd think we did it already, but in fact the implementation

at the top of this chapter is all wrong (run it and see!) We 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.

```
{-@ 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
```

To iterate over the *columns* of the matrix `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 7.12 (Matrix Transpose). ⋆⋆ *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`).

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 data structures. We will see examples ranging from high-level AVL trees, to low-level safe pointer arithmetic.

8
Elemental Measures

Often, correctness requires us to reason about the *set of elements* represented inside a data structure, or manipulated by a function. Examples of this abound: 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 very expressive logics. In addition to 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`.[1]

[1] See this for a brief description of how to work directly with the set operators natively supported by LiquidHaskell.

LiquidHaskell Lifts the basic set operators from `Data.Set` into the refinement logic. That is, the prelude 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.[2]

[2] See this recent paper 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.

WE REFINE THE SET API to make it easy to write down theorems. That is, we give the operators in `Data.Set` refinement type signatures that precisely track their set-theoretic behavior:

```
empty        :: {v:Set a | v = empty}
member       :: x:a
             -> s:Set a
             -> {v:Bool | Prop v <=> member x s}

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}
```

We Can Assert 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 the `Bool`eans that are always `True` or `False`

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

We can use `True` to state theorems. For example, the unexciting 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:Bool -> q:Bool -> Implies p q  @-}
implies False _    = True
implies _     True = True
implies _    _     = False
```

and `Implies p q` is defined as

```
{-@ type Implies P Q = {v:_ | Prop v <=> (Prop P => Prop Q)} @-}
```

Exercise 8.1 (Bounded Addition). *Write and prove a QuickCheck style theorem that:* $\forall x, y. x < 100 \wedge y < 100 \Rightarrow x + y < 200$.

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

The Commutativity of Intersection can be easily stated and proved as a QuickCheck style theorem:

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

The Associativity of Union can similarly be confirmed:

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

The Distributivity Laws for Boolean Algebra can be verified by writing properties over the relevant operators. For example, we lets check that `union` distributes over `intersection`:

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

NON-THEOREMS should be rejected. So, 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 8.2 (Set Difference). *Why does the above property fail?*

1. *Use QuickCheck (or your own little grey cells) to find a* counterexample *for the property* `prop_cup_dif_bad`.
2. *Use the counterexample to assign* `pre` *a non-trivial (i.e. other than* `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.[3]

[3] The SBV and Leon projects describe a different DSL based approach for using SMT solvers from Haskell and Scala respectively.

Content-Aware List API

Lets return to our real goal, which is to to verify properties of programs. First, we need a way to refine the list API to precisely track the set of elements in a list.

THE ELEMENTS OF A LIST can be described by a simple recursive measure that walks over the list, building up the set:

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

Lets write a few helpful aliases for various refined lists that will then make the subsequent specifications pithy and crisp.

- A list with elements `S`

```
{-@ type ListS a S = {v:[a] | elts v = S} @-}
```

- An *empty* list

```
{-@ type ListEmp a = ListS a {Set_empty 0} @-}
```

- A list whose contents *equal* those of list `X`

```
{-@ type ListEq a X = ListS a {elts X}    @-}
```

- A list whose contents are a *subset* of list `X`

```
{-@ type ListSub a X = {v:[a]| Set_sub (elts v) (elts X)} @-}
```

- A list whose contents are the union of lists `X` and `Y`

```
{-@ type ListUn a X Y = ListS a {Set_cup (elts X) (elts Y)} @-}
```

- A list whose contents are exactly `X` and the contents of `Y`

```
{-@ type ListUn1 a X Y = ListS a {Set_cup (Set_sng X) (elts Y)} @-}
```

The Measures strengthens the data constructors for lists. That is we get the automatically refined types for "nil" and "cons":

```
data List a where
  []  :: ListEmp a
  (:) :: x:a -> xs:List a -> ListUn1 a x xs
```

Lets take our new vocabulary out for a spin!

The Append function returns a list whose elements are the *union* of the elements of the input Lists:

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

Exercise 8.3 (Reverse). *Write down a type for* `revHelper` *so that* `reverse'` *is verified by LiquidHaskell.*

```
{-@ reverse' :: xs:List a -> ListEq a xs @-}
reverse' xs = revHelper [] xs

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

Exercise 8.4 (Halve). ⋆ *Write down a specification for* `halve` *such that the subsequent "theorem"* `prop_halve_append` *is proved by LiquidHaskell.*

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

{-@ prop_halve_append  :: _ -> _ -> True @-}
prop_halve_append n xs = elts xs == elts xs'
  where
    xs'      =  append' ys zs
    (ys, zs) =  halve n xs
```

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

Exercise 8.5 (Membership). *Write down a signature for* `elem` *that suffices to verify* `test1` *and* `test2`.

```
{-@ 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 sorting routines return *permutations* of their inputs, that is, return output lists whose elements are the *same as* those of the input lists.[4]

[4] Since we are focusing on the elements, lets not distract ourselves with the ordering invariant and reuse plain old lists. See this for how to specify and verify order with plain old lists.

INSERTION SORT 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:List a -> ListUn1 a x xs @-}
```

The above signature lets us prove that the output of the sorting routine indeed has the elements of the input:

```
{-@ insertSort :: (Ord a) => xs:List a -> ListEq a xs @-}
insertSort []     = []
insertSort (x:xs) = insert x (insertSort xs)
```

Exercise 8.6 (Merge). *Fix the specification of* `merge` *so that the subsequent property* `prop_merge_app` *is verified by LiquidHaskell.*

```
{-@ merge :: xs:List a -> ys:List a -> List a @-}
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 = elts zs == elts zs'
  where
    zs              = append' xs ys
    zs'             = merge   xs ys
```

Exercise 8.7 (Merge Sort). ⋆⋆ *Once you write the correct type for* `merge` *above, you should be able to prove the er, unexpected signature for* `mergeSort` *below.*

1. *Make sure you are able verify the given signature.*

2. *Obviously we don't want* `mergeSort` *to return the empty list, so there's a bug. Find and fix it, so that you* cannot *prove that the output is empty, but* can *instead prove that the output is* `ListEq a xs`.

```
{-@ mergeSort :: (Ord a) => xs:List a -> ListEmp a @-}
mergeSort []  = []
mergeSort xs  = merge (mergeSort ys) (mergeSort zs)
  where
   (ys, zs)   = halve mid xs
   mid        = length xs `div` 2
```

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 the zipper contains no duplicates. Next, lets see how to specify and verify this invariant using LiquidHaskell, first for lists, and then for a simplified zipper.

To SPECIFY UNIQUENESS we need a way of saying that a list has *no duplicates*. There are many ways to do so; the simplest is a *measure*:

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

We can use the above to write an alias for duplicate-free lists

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

Lets quickly 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
```

THE FILTER function returns a *subset* of its elements, and hence, *preserves* uniqueness. That is, if the input is unique, the output is too:

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

Exercise 8.8 (Filter). *It seems a bit draconian to require that* `filter` *only be called with unique lists. Write down a more permissive type for* `filter'` *below such that the subsequent uses are verified by LiquidHaskell.*

```
filter' _ []   = []
filter' f (x:xs)
  | f x       = x : xs'
  | otherwise = xs'
  where
    xs'       = filter' f xs

{-@ test3 :: UList _ @-}
test3     = filter' (> 2) [1,2,3,4]

{-@ test4 :: [_] @-}
test4     = filter' (> 3) [3,1,2,3]
```

Exercise 8.9 (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     :: a:List a -> xs:List a -> List a @-}
    go a []     = a
    go a (x:xs) = go (x:a) xs
```

The Nub function constructs a `unique` list from an arbitrary input by traversing the input and tossing out elements that are already `seen`:

```
{-@ nub              :: List a -> UList a @-}
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.[5]

[5] Which should be clear by now, if you did a certain exercise above ....

```
-- FIXME
{-@ predicate In X Xs = Set_mem X (elts Xs) @-}

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

Exercise 8.10 (Append). ⋆ *Why does* `append`*ing 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 8.11 (Range). ⋆⋆ `range i j` *returns the list of* `Int` *between* `i` *and* `j`*. LiquidHaskell refuses to acknowledge that the output is indeed a* `UList`*. Fix the code so that LiquidHaskell verifies that it implements the given signature (and of course, computes the same result.)*

```
{-@ 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 = []
```

Hint: This may be easier to do *after* you read this chapter about lemmas.

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   :: List a
  , right  :: List 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, that is, be free of duplicates.

WE REFINE ZIPPER to capture the requirement that legal zippers are unique. To this end, we state that the `left` and `right` lists are unique, disjoint, and do not contain `focus`.

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

{-@ predicate Disj X Y = Disjoint (elts X) (elts Y)       @-}
```

OUR REFINED ZIPPER CONSTRUCTOR makes *illegal states unrepresentable*. That is, by construction, we will ensure that every `Zipper` is free of duplicates. For example, 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 8.12 (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)
```

WE CAN SHIFT THE FOCUS element to the left or right while preserving the uniqueness invariant. Here's the code that shifts the focus to the left:

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

To shift to the right, we simply *reverse* the elements and shift to the left:

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

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

To FILTER elements from a zipper, we need to take care when the `focus` itself, or all the elements get eliminated. In the latter case, there is no `Zipper` and so the operation returns a `Maybe`:

```
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'
      []     -> case filter p ls of
                  f':ls' -> Just $ Zipper f' ls' []
                  []     -> Nothing
```

Thus, by using LiquidHaskell's refinement types, and the SMT solvers native reasoning about sets, we can ensure the key uniqueness invariant holds in the presence of various tricky operations that are performed over `Zippers`.

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. In particular, we saw how to:

- *Lift set-theoretic primitives* to refined Haskell functions from the `Data.Set` library,
- *Define measures* like `elts` that characterize the set of elements of structures, and `unique` that describe high-level application specific properties about those sets,
- *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 longer *case-studies* that illustrate the techniques developed so far on particular application domains.

9
Case Study: Okasaki's Lazy Queues

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

Queues

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

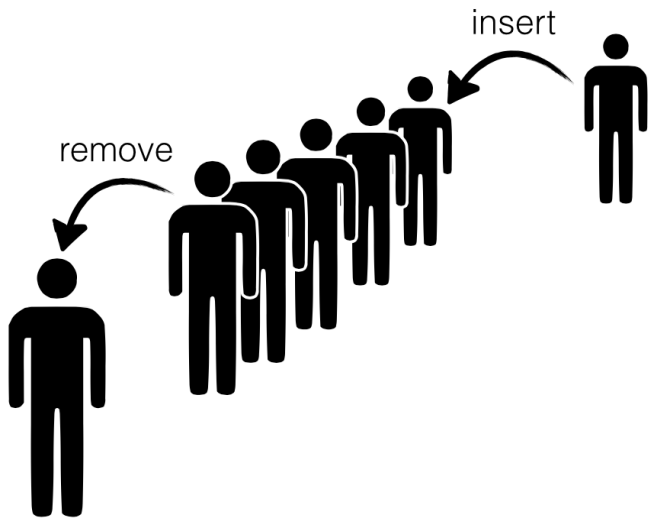

Figure 9.1: A Queue is a structure into which we can insert and remove elements. The order in which the elements are removed is the same as the order in which they were inserted.

To EFFICIENTLY IMPLEMENT a queue we need to have rapid access to both the front as well as the back because we `remove` elements from former and `insert` elements into the latter. This is quite straightforward with explicit pointers and mutation – one uses an old school

linked list and maintains pointers to the head and the tail. But can we implement the structure efficiently without having stoop so low?

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

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

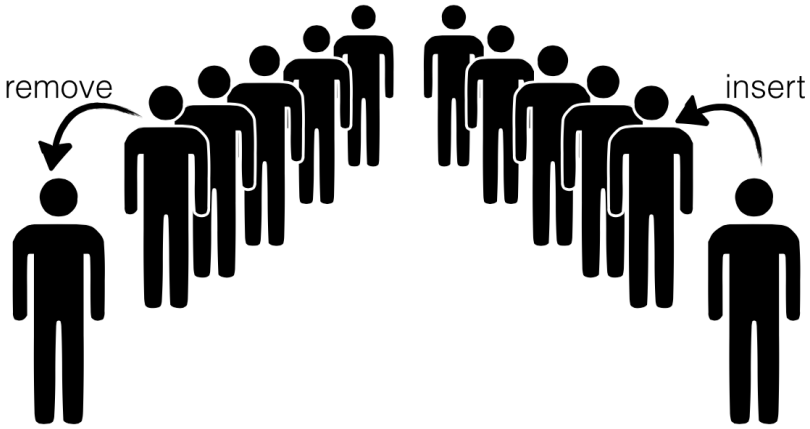

Figure 9.2: We can implement a Queue with a pair of lists; respectively representing the front and back.

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

1. a `remove` call is triggered, and
2. the `front` list is empty.

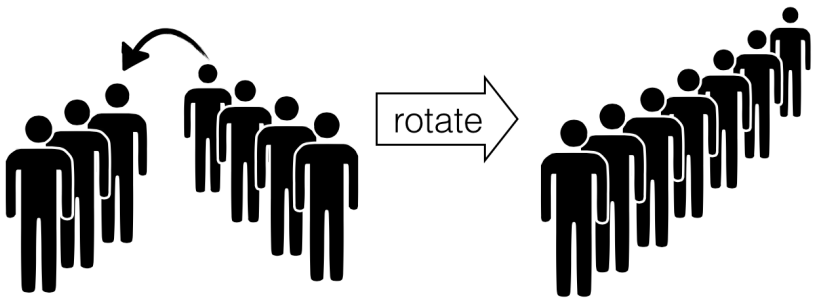

Figure 9.3: Transferring Elements from back to front.

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

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

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

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

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

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

Sized Lists

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

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

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

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

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

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

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

As a sanity check, consider this:

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

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

LETS DEFINE AN ALIAS for lists of a given size `N`:

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

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

To CONSTRUCT LISTS, we use `nil` and `cons`:

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

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

Exercise 9.1 (Destructing Lists). *We can destruct lists by writing a* `hd` *and* `tl` *function as shown below. Fix the specification or implementation such that the definitions typecheck.*

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

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

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

```
okHd  = hd okList       -- accepted

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

Queue Type

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

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

The alias `SListLE a L` corresponds to lists with at most `N` elements:

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

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

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

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

Queue Operations

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

The Empty Queue is simply one where both `front` and `back` are both empty:

```
emp = Q nil nil
```

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

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

Exercise 9.2 (Whither pattern matching?)**.** *Can you explain why we (or Okasaki) didn't use pattern matching here, and have instead opted for the explicit* `hd` *and* `tl`*?*

Exercise 9.3 (Queue Sizes)**.** *If you did the* List Destructing *exercise above, then you will notice that the code for* `remove` *has a type error: namely, the calls to* `hd` *and* `tl` *may fail if the* `f` *list is empty.*

1. *Write a* measure *to describe the queue size,*
2. *Use it to complete the definition of* `QueueN` *below, and*
3. *Use it to give* `remove` *a type that verifies the safety of the calls made to* `hd` *and* `tl`.

Hint: When you are done, `okRemove` should be accepted, `badRemove` should be rejected, and `emp` should have the type shown below:

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

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

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

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

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

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

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

Exercise 9.4 (Insert). *Write down a type for* `insert` *such that* `replicate` *and* `y3` *are accepted by LiquidHaskell, but* `y2` *is rejected.*

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

{-@ y3 :: QueueN _ 3 @-}
y3     = replicate 3 "Yeah!"

{-@ y2 :: QueueN _ 3 @-}
y2     = replicate 1 "No!"
```

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

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

Exercise 9.5 (Rotate). ⋆⋆ *The Rotate function* `rot` *is only called when the* `back` *is one larger than the* `front` *(we never let things drift beyond that). It is arranged so that it the* `hd` *is built up fast, before the entire computation finishes; which, combined with laziness provides the efficient worst-case guarantee. Write down a type for* `rot` *so that it typechecks and verifies the type for* `makeq`.

Hint: You may have to modify a precondition in `makeq` to capture the relationship between `f` and `b`.

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

Exercise 9.6 (Transfer). *Write down a signature for* `take` *which extracts* `n` *elements from its input* `q` *and puts them into a new output Queue. When you are done,* `okTake` *should be accepted, but* `badTake` *should be rejected.*

```
take           :: Int -> Queue a -> (Queue a, Queue a)
take 0 q       = (emp          , q)
take n q       = (insert x out , q')
  where
    (x  , q')  = remove q
    (out, q'') = take (n-1) q'

{-@ okTake :: (QueueN _ 2, QueueN _ 1) @-}
okTake   = take 2 exampleQ  -- accept

badTake  = take 10 exampleQ -- reject

exampleQ = insert "nal" $ insert "bob" $ insert "alice" $ emp
```

Recap

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

1. The refinements express invariants that are critical for efficiency,

2. The code introspects on the `size` to guarantee the invariants, and
3. The code is quite simple and we hope, easy to follow!

10
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 and refined data types, to analyze programs that *implement* and *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 a type for `Maps`. As usual, lets parameterize the type with `k` for the type of keys and `v` for the type of values:

```
data Map k v    -- Currently left abstract
```

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`.[1]

[1] Recall that `Empty`, `Union`, `In` and the other `Set` operators are described here.

EMPTY `Maps` have no keys in them. Hence, we type the empty `Map` as:

```
emp :: {m:Map k v | Empty (keys m)}
```

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 :: k:k -> v -> m:Map k v -> {n: Map k v| AddKey k m n}

predicate AddKey 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 = In 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 :: k:k -> m:Map k v -> {v:Bool|Prop v <=> HasKey k m}

-- | Lookup key's value
get :: 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:[2]

[2] 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 `Maps` were invented! Lets define our environments as `Maps` from `Variables` to `Values`:

```
{-@ 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)  = xs1 `union`  xs2
  where
    xs1            = free e1
    xs2            = free e2
free (Let x e1 e2) = xs1 `union` (xs2 `difference` xs)
  where
    xs1            = free e1
    xs2            = free e2
    xs             = 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 10.1 (Wellformedness Check). *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 10.2 (Closures). ⋆⋆ *Extend the language above to include functions. That is, extend* `Expr` *as below, (and* `eval` *and* `free` *respectively.)*

```
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 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) = ks `union` kl `union` kr
  where
    kl             = keys l
    kr             = keys r
    ks             = singleton k
```

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

Exercise 10.3 (Empty Maps). *To make sure you are following, fill in the definition for an* `empty` *Map:*

```
{-@ emp :: {m:Map k v | Empty (keys m)} @-}
emp     = undefined
```

Exercise 10.4 (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 | AddKey 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:

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

CONVERSION LEMMAS Unfortunately, LiquidHaskell *cannot automatically* deduce facts like the above, as they relate refinements of a container's *type parameters* (here: $\texttt{key}' \neq \texttt{key}$, which refines the `Map`s first type parameter) with properties of the entire container (here: `HasKey key t`). Fortunately, it is easy to *state*, *prove* and *use* facts like the above, via *lemmas* which are just functions.[3]

[3] Why does LiquidHaskell not automatically deduce this information? 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's how to use lemmas to verify `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` refined with the fact `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 in `assert`:

```
assert _ x = x
```

GHOST VALUES This technique of materializing auxiliary facts via *ghost values* is a well known idea in program verification. 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 wreck the efficient logarithmic lookup time, assuming we kept the trees balanced, as we would traverse the *entire* tree instead of just the short path to a node.[4]

[4] Which is what makes dynamic contract checking inefficient 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 10.5 (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:_ | 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 10.6 (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  = In 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.

11
Case Study: Pointers & Bytes

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.[1] 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.

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.

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

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` `Character`s 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 Figure 11.1, 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"
```

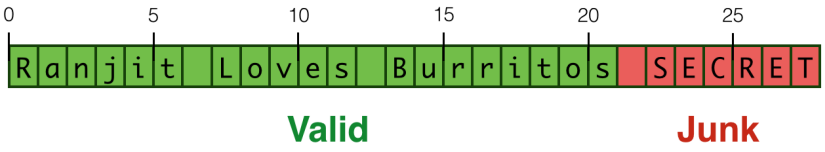

Figure 11.1: Can we prevent the program from leaking secrets via overflows?

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.[2]

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

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

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?[3] We will use the following strategy to turn LiquidHaskell into such an assistant:

[3] In Vim or Emacs or online, you'd see the error helpfully highlighted.

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.[4]

[4] This is another *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 it is always safe to run the `act` on a larger buffer. For example, the below variant of `zero4` where we only set the first three bytes is fine as the `act`, namely the function `\p -> ...`, can be typed with the requirement that the buffer `p` has size `4`, even though only `3` bytes are actually touched.

```
zero3 = do fp <- mallocForeignPtrBytes 4
           withForeignPtr fp $ \p -> do
             poke (p `plusPtr` 0) zero
             poke (p `plusPtr` 1) zero
             poke (p `plusPtr` 2) zero
           return fp
        where
           zero = 0 :: Word8
```

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:[5]

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

using the alias `BNat`, defined as:

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

6

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     -- LH complains
           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}
```

[5] This signature precludes *left* or *backward* shifts; for that there is an analogous `minusPtr` which we elide for simplicity.

[6] Did you notice 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.

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). The latter equals 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 be sure 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.[7]

[7] 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 of three fields:

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

```
  , bLen :: !Int
  }
```

- `bPtr` is a *pointer* to a block of memory,
- `bOff` is the *offset* in the block where the string begins,
- `bLen` is the number of bytes from the offset that belong to the string.

These entities are illustrated in Figure 11.2; the green portion represents the actual contents of a particular `ByteString`. This representation makes it possible to implement various operations like computing prefixes and suffixes extremely quickly, simply by pointer arithmetic.

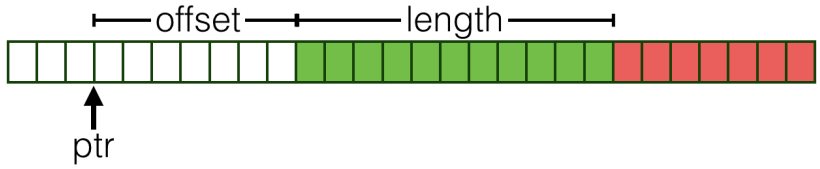

Figure 11.2: Representing ByteStrings in memory.

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 `ByteStrings`:

```
{-@ 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 `ByteStrings` 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 create `ByteStrings` whose length is less than the size of the allocated block, as shown in `good2` whose length is `2` while the allocated block has size `5`.

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

Illegal Bytestrings are rejected by LiquidHaskell. For example, `bad1`'s length is exceeds its 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 11.1 (Legal ByteStrings)**.** *Modify the definitions of* `bad1` *and* `bad2` *so they are* accepted *by LiquidHaskell.*

Measures are generated from Fields in the datatype definition. As GHC lets us use the fields as accessor functions, we can *refine* the types of those functions to specify their behavior to LiquidHaskell. For example, we can type the (automatically generated) field-accessor function `bLen` so that it actually returns the exact size of the `ByteString` argument.

```
{-@ bLen :: b:ByteString -> {v: Nat | v = bLen b} @-}
```

To 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 11.2 (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:[8]

[8] 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 11.3 (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.*

```
{-@ prop_pack_length  :: String -> TRUE @-}
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 11.4 (Pack Invariant). EXERCISE 11.1. ⋆ *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 11.5 (Unsafe Take and Drop). *The functions* `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:_ -> ByteStringN n @-}
unsafeTake n (BS x s _) = BS x s n

{-@ unsafeDrop :: n:Nat -> b:_ -> 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 = peekAt p 0 >>= \e -> return (w2c e : acc)
    go p n acc = peekAt p n >>= \e -> go p (n-1) (w2c e : acc)
    peekAt p n = peek (p `plusPtr` n)
```

Exercise 11.6 (Unpack). ⋆ *Fix the specification for* unpack *so that the below QuickCheck style property is proved by LiquidHaskell.*

```
{-@ prop_unpack_length :: ByteString -> TRUE @-}
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. We require that the prefix size n be less than the size of the input string s:

```
{-@ chop :: s:String -> n:BNat (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 (cf. Figure 11.1). In the code below, the first use of chop which defines ex6 is accepted as `6 <= len ex` but the second call is rejected as `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
```

Fix the specification for chop so that the following property is proved:

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

Exercise 11.7 (Checked Chop). *In the above, we know statically that the string is longer than the prefix, but what if the string and prefix are obtained* dynamically*, e.g. as inputs from the user? Fill in the implementation of* `ok` *below to ensure that* `chop` *is called safely with user specified values:*

```
safeChop      :: String -> Int -> String
safeChop str n
  | ok        = chop str n
  | otherwise = ""
  where
    ok        = True

queryAndChop  :: IO String
queryAndChop  = do putStrLn "Give me a string:"
                   str  <-  getLine
                   putStrLn "Give me a number:"
                   ns   <-  getLine
                   let n =  read ns :: Int
                   return $ safeChop str n
```

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:

```
{-@ predicate Null B  = bLen B == 0                    @-}
{-@ type ByteStringNE = {v:ByteString | not (Null v)} @-}
```

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

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

This check is used to determine if it is safe to extract the head and tail of the `ByteString`. we can use refinements to ensure the safety of the operations and also track the sizes.[9]

[9] `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:_ -> {v: [ByteStringNE] | bsLen 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 bsLen @-}
bsLen        :: [ByteString] -> Int
bsLen []     = 0
bsLen (b:bs) = bLen b + bsLen 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 $ splitAt i
```

```
                        else go p (i+1)
    splitAt i     = (unsafeTake i ps, unsafeDrop i ps)
```

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 that we can:

1. *compose* larger systems from layers of smaller ones,
2. *refine* APIs for each layer, which can be used to
3. *design and validate* the layers above.

We saw this recipe 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.

12
Case Study: AVL Trees

One of the most fundamental abstractions in computing is that of a *collection* of values – names, numbers, records – into which we can rapidly `insert`, `delete` and check for `membership`.

Trees offer an an attractive means of implementing collections in the immutable setting. We can *order* the values to ensure that each operation takes time proportional to the *path* from the root to the datum being operated upon. If we additionally keep the tree *balanced* then each path is small (relative to the size of the collection), thereby giving us an efficient implementation for collections.

As in real life maintaining order and balance is rather easier said than done. Often we must go through rather sophisticated gymnastics to ensure everything is in its right place. Fortunately, LiquidHaskell can help. Lets see a concrete example, that should be familiar from your introductory data structures class: the Georgy Adelson-Velsky and Landis' or AVL Tree.

AVL Trees

An `AVL` tree is defined by the following Haskell datatype:[1]

[1] This chapter is based on code by Michael Beaumont.

```
data AVL a =
    Leaf
  | Node { key :: a      -- value
         , l   :: AVL a  -- left subtree
         , r   :: AVL a  -- right subtree
         , ah  :: Int    -- height
         }
    deriving (Show)
```

While the Haskell type signature describes any old binary tree, an

AVL tree like that shown in Figure 12.1 actually satisfies two crucial invariants: it should be binary search ordered and balanced.

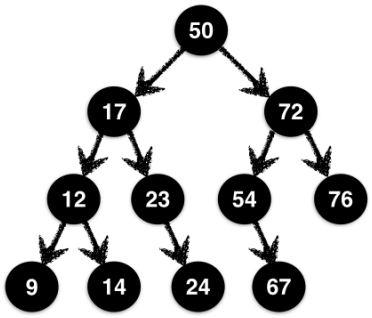

Figure 12.1: An AVL tree is an ordered, height-balanced tree.

A Binary Search Ordered tree is one where at *each* Node, the values of the left and right subtrees are strictly less and greater than the values at the Node. In the tree in Figure 12.1 the root has value 50 while its left and right subtrees have values in the range 9-23 and 54-76 respectively. This holds at all nodes, not just the root. For example, the node 12 has left and right children strictly less and greater than 12.

A Balanced tree is one where at *each* node, the *heights* of the left and right subtrees differ by at most 1. In Figure 12.1, at the root, the heights of the left and right subtrees are the same, but at the node 72 the left subtree has height 2 which is one more then the right subtree.

The Invariants Lead To Fast Operations. Order ensures that there is at most a single path of left and right moves from the root at which an element can be found; balance ensures that each such path in the tree is of size $O(\log\ n)$ where n is the numbers of nodes. Thus, together they ensure that the collection operations are efficient: they take time logarithmic in the size of the collection.

Specifying AVL Trees

The tricky bit is to ensure order and balance. Before we can ensure anything, lets tell LiquidHaskell what we mean by these terms, by defining legal or valid AVL trees.

To Specify Order we just define two aliases AVLL and AVLR – read *AVL-left* and *AVL-right* – for trees whose values are strictly less than and greater than some value X:

```
-- | Trees with value less than X
{-@ type AVLL a X = AVL {v:a | v < X}  @-}

-- | Trees with value greater than X
{-@ type AVLR a X = AVL {v:a | X < v}  @-}
```

The Real Height of a tree is defined recursively as 0 for Leafs and one more than the larger of left and right subtrees for Nodes. Note that we cannot simply use the ah field because thats just some arbitrary Int – there is nothing to prevent a buggy implementation

from just filling that field with `0` everywhere. In short, we need the ground truth: a measure that computes the *actual* height of a tree.[2]

[2] **FIXME** The `inline` pragma indicates that the Haskell functions can be directly lifted into and used inside the refinement logic and measures.

```
{-@ measure realHeight @-}
realHeight               :: AVL a -> Int
realHeight Leaf          = 0
realHeight (Node _ l r _) = nodeHeight l r

{-@ inline nodeHeight @-}
nodeHeight l r = 1 + max hl hr
  where
    hl         = realHeight l
    hr         = realHeight r

{-@ inline max @-}
max :: Int -> Int -> Int
max x y = if x > y then x else y
```

A Reality Check predicate ensures that a value `v` is indeed the *real* height of a node with subtrees `l` and `r`:

```
{-@ inline isReal @-}
isReal v l r = v == nodeHeight l r
```

A Node is n-Balanced if its left and right subtrees have a (real) height difference of at most n. We can specify this requirement as a predicate `isBal l r n`

```
{-@ inline isBal @-}
isBal l r n = 0 - n <= d && d <= n
  where
    d       = realHeight l - realHeight r
```

A Legal AVL Tree can now be defined via the following refined data type, which states that each `Node` is 1-balanced, and that the saved height field is indeed the *real* height:

```
{-@ data AVL a = Leaf
               | Node { key :: a
                      , l   :: AVLL a key
                      , r   :: {v:AVLR a key | isBal l v 1}
                      , ah  :: {v:Nat        | isReal v l r}
                      }                                  @-}
```

Smart Constructors

Lets use the type to construct a few small trees which will also be handy in a general collection API. First, lets write an alias for trees of a given height:

```
-- | Trees of height N
{-@ type AVLN a N = {v: AVL a | realHeight v = N} @-}

-- | Trees of height equal to that of another T
{-@ type AVLT a T = AVLN a {realHeight T} @-}
```

An Empty collection is represented by a `Leaf`, which has height `0`:

```
{-@ empty :: AVLN a 0 @-}
empty = Leaf
```

Exercise 12.1 (Singleton). *Consider the function* `singleton` *that builds an* `AVL` *tree from a single element. Fix the code below so that it is accepted by LiquidHaskell.*

```
{-@ singleton :: a -> AVLN a 1 @-}
singleton x =  Node x empty empty 0
```

As you can imagine, it can be quite tedious to keep the saved height field `ah` *in sync* with the *real* height. In general in such situations, which arose also with lazy queues, the right move is to eschew the data constructor and instead use a *smart constructor* that will fill in the appropriate values correctly.[3]

[3] Why bother to save the height anyway? Why not just recompute it instead?

The Smart Constructor `node` takes as input the node's value `x`, left and right subtrees `l` and `r` and returns a tree by filling in the right value for the height field.

```
{-@ mkNode :: a -> l:AVL a -> r:AVL a
           -> AVLN a {nodeHeight l r}
  @-}
mkNode v l r = Node v l r h
 where
   h       = 1 + max hl hr
   hl      = getHeight l
   hr      = getHeight r
```

Exercise 12.2 (Constructor). *Unfortunately, LiquidHaskell rejects the above smart constructor* `node`. *Can you explain why? Can you fix the code (implementation or specification) so that the function is accepted?*

Hint: Think about the (refined) type of the actual constructor `Node`, and the properties it requires and ensures.

Inserting Elements

Next, lets turn our attention to the problem of *adding* elements to an `AVL` tree. The basic strategy is this:

1. *Find* the appropriate location (per ordering) to add the value,
2. *Replace* the `Leaf` at that location with the singleton value.

If you prefer the spare precision of code to the informality of English, here is a first stab at implementing insertion:[4]

[4] `node` is a fixed variant of the smart constructor `mkNode`. Do the exercise *without* looking at it.

```
{-@ insert0    :: (Ord a) => a -> AVL a -> AVL a @-}
insert0 y t@(Node x l r _)
  | y < x      = insL0 y t
  | x < y      = insR0 y t
  | otherwise  = t
insert0 y Leaf = singleton y

insL0 y (Node x l r _) = node x (insert0 y l) r
insR0 y (Node x l r _) = node x l (insert0 y r)
```

Unfortunately `insert0` does not work. If you did the exercise above, you can replace it with `mkNode` and you will see that the above function is rejected by LiquidHaskell. The error message would essentially say that at the calls to the smart constructor, the arguments violate the balance requirement.

Insertion Increases The Height of a sub-tree, making it *too large* relative to its sibling. For example, consider the tree `t0` defined as:

```
ghci> let t0 = Node { key = 'a'
                    , l   = Leaf
                    , r   = Node {key = 'd'
                                  , l  = Leaf
                                  , r  = Leaf
                                  , ah = 1 }
                    , ah = 2}
```

If we use `insert0` to add the key `'e'` (which goes after `'d'`) then we end up with the result:

```
ghci> insert0 'e' t0
  Node { key = 'a'
       , l   = Leaf
       , r   = Node { key = 'd'
                    , l   = Leaf
                    , r   = Node { key = 'e'
                                 , l   = Leaf
                                 , r   = Leaf
                                 , ah  = 1   }
                    , ah = 2                 }
       , ah = 3}
```

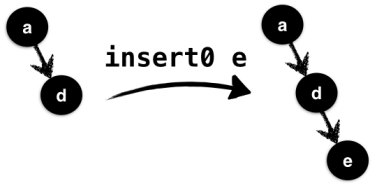

Figure 12.2: Naive insertion breaks balancedness

In the above, illustrated in Figure 12.2 the value `'e'` is inserted into the valid tree `t0`; it is inserted using `insR0`, into the *right* subtree of `t0` which already has height `1` and causes its height to go up to `2` which is too large relative to the empty left subtree of height `0`.

LIQUIDHASKELL CATCHES THE IMBALANCE by rejecting `insert0`. The new value `y` is inserted into the right subtree `r`, which (may already be bigger than the left by a factor of `1`). As insert can return a tree with arbitrary height, possibly much larger than `l` and hence, LiquidHaskell rejects the call to the constructor `node` as the balance requirement does not hold.

TWO LESSONS can be drawn from the above exercise. First, `insert` may *increase* the height of a tree by at most `1`. So, second, we need a way to *rebalance* sibling trees where one has height `2` more than the other.

Rebalancing Trees

The brilliant insight of Adelson-Velsky and Landis was that we can, in fact, perform such a rebalancing with a clever bit of gardening. Suppose we have inserted a value into the *left* subtree `l` to obtain a new tree `l'` (the right case is symmetric.)

THE RELATIVE HEIGHTS of `l'` and `r` fall under one of three cases:

- *(RightBig)* `r` is two more than `l'`,
- *(LeftBig)* `l'` is two more than `r`, and otherwise
- *(NoBig)* `l'` and `r` are within a factor of `1`,

WE CAN SPECIFY these cases as follows.

```
{-@ inline leftBig @-}
leftBig l r = diff l r == 2

{-@ inline rightBig @-}
rightBig l r = diff r l == 2

{-@ inline diff @-}
diff s t = getHeight s - getHeight t
```

the function `getHeight` accesses the saved height field.

```
{-@ measure getHeight @-}
getHeight Leaf           = 0
getHeight (Node _ _ _ n) = n
```

In `insL`, the *RightBig* case cannot arise as `l'` is at least as big as `l`, which was within a factor of `1` of `r` in the valid input tree `t`. In *NoBig*, we can safely link `l'` and `r` with the smart constructor as they satisfy the balance requirements. The *LeftBig* case is the tricky one: we need a way to shuffle elements from the left subtree over to the right side.

What is a LeftBig tree? Lets split into the possible cases for `l'`, immediately ruling out the *empty* tree because its height is `0` which cannot be `2` larger than any other tree.

- *(NoHeavy)* the left and right subtrees of `l'` have the same height,
- *(LeftHeavy)* the left subtree of `l'` is bigger than the right,
- *(RightHeavy)* the right subtree of `l'` is bigger than the left.

The Balance Factor of a tree can be used to make the above cases precise. Note that while the `getHeight` function returns the saved height (for efficiency), thanks to the invariants, we know it is in fact equal to the `realHeight` of the given tree.

```
{-@ measure balFac @-}
balFac Leaf           = 0
balFac (Node _ l r _) = getHeight l - getHeight r
```

Heaviness can be encoded by testing the balance factor:

```
{-@ inline leftHeavy @-}
leftHeavy  t = balFac t > 0

{-@ inline rightHeavy @-}
```

```
rightHeavy t = balFac t < 0

{-@ inline noHeavy @-}
noHeavy    t = balFac t == 0
```

Adelson-Velsky and Landis observed that once you've drilled down into these three cases, the shuffling suggests itself.

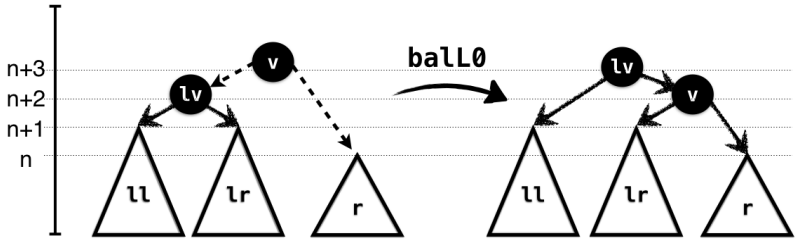

Figure 12.3: Rotating when in the LeftBig, NoHeavy case.

In the NoHeavy case, illustrated in Figure 12.3, the subtrees `ll` and `lr` have the same height which is one more than that of `r`. Hence, we can link up `lr` and `r` and link the result with `l`. Here's how you would implement the rotation. Note how the preconditions capture the exact case we're in: the left subtree is *NoHeavy* and the right subtree is smaller than the left by 2. Finally, the output type captures the exact height of the result, relative to the input subtrees.

```
{-@ balL0 :: x:a
          -> l:{AVLL a x | noHeavy l}
          -> r:{AVLR a x | leftBig l r}
          -> AVLN a {realHeight l + 1 }
  @-}
balL0 v (Node lv ll lr _) r = node lv ll (node v lr r)
```

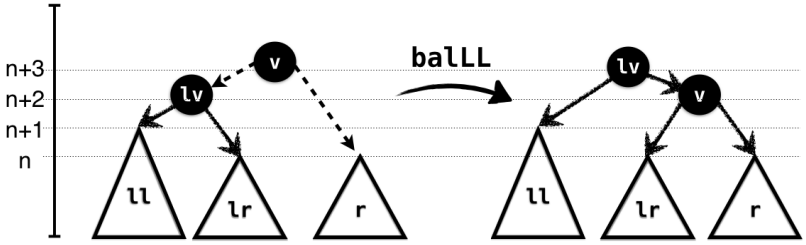

Figure 12.4: Rotating when in the LeftBig, LeftHeavy case.

In the LeftHeavy case, illustrated in Figure 12.4, the subtree `ll` is larger than `lr`; hence `lr` has the same height as `r`, and again we can

link up `lr` and `r` and link the result with `l`. As in the *NoHeavy* case, the input types capture the exact case, and the output the height of the resulting tree.

```
{-@ balLL :: x:a
          -> l:{AVLL a x | leftHeavy l}
          -> r:{AVLR a x | leftBig l r}
          -> AVLT a l
  @-}
balLL v (Node lv ll lr _) r = node lv ll (node v lr r)
```

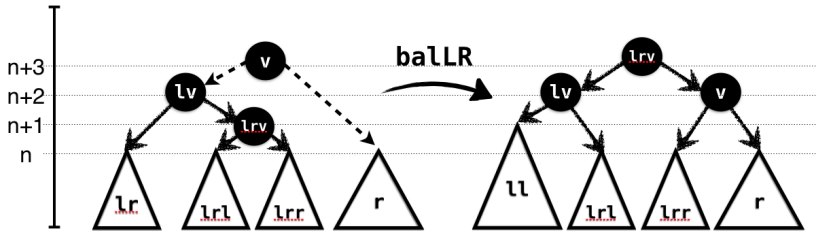

Figure 12.5: Rotating when in the LeftBig, RightHeavy case.

In the RightHeavy case, illustrated in Figure 12.5, the subtree `lr` is larger than `ll`. We cannot directly link it with `r` as the result would again be too large. Hence, we split it further into its own subtrees `lrl` and `lrr` and link the latter with `r`. Again, the types capture the requirements and guarantees of the rotation.

```
{-@ balLR :: x:a
          -> l:{AVLL a x | rightHeavy l}
          -> r:{AVLR a x | leftBig l r}
          -> AVLT a l
  @-}
balLR v (Node lv ll (Node lrv lrl lrr _) _) r
  = node lrv (node lv ll lrl) (node v lrr r)
```

The *RightBig* cases are symmetric to the above cases where the left subtree is the larger one.

Exercise 12.3 (RightBig, NoHeavy)**.** *Fix the implementation of* `balR0` *so that it implements the given type.*

```
{-@ balR0 :: x:a
          -> l: AVLL a x
          -> r: {AVLR a x | rightBig l r && noHeavy r}
          -> AVLN a {realHeight r + 1}
  @-}
balR0 v l r = undefined
```

Exercise 12.4 (RightBig, RightHeavy). *Fix the implementation of* `balRR` *so that it implements the given type.*

```
{-@ balRR :: x:a
          -> l: AVLL a x
          -> r:{AVLR a x | rightBig l r && rightHeavy r}
          -> AVLT a r
  @-}
balRR v l r = undefined
```

Exercise 12.5 (RightBig, LeftHeavy). *Fix the implementation of* `balRL` *so that it implements the given type.*

```
{-@ balRL :: x:a
          -> l: AVLL a x
          -> r:{AVLR a x | rightBig l r && leftHeavy r}
          -> AVLT a r
  @-}
balRL v l r = undefined
```

To Correctly Insert an element, we recursively add it to the left or right subtree as appropriate and then determine which of the above cases hold in order to call the corresponding *rebalance* function which restores the invariants.

```
{-@ insert :: a -> s:AVL a -> {t: AVL a | eqOrUp s t} @-}
insert y Leaf = singleton y
insert y t@(Node x _ _ _)
  | y < x     = insL y t
  | y > x     = insR y t
  | otherwise = t
```

The refinement, `eqOrUp` says that the height of `t` is the same as `s` or goes *up* by atmost `1`.

```
{-@ inline eqOrUp @-}
eqOrUp s t = d == 0 || d == 1
  where
    d      = diff t s
```

THE HARD WORK happens inside `insL` and `insR`. Here's the first; it simply inserts into the left subtree to get `l'` and then determines which rotation to apply.

```
{-@ insL :: x:a
         -> t:{AVL a | x < key t && 0 < realHeight t}
         -> {v: AVL a | eqOrUp t v}
  @-}
insL a (Node v l r _)
  | isLeftBig && leftHeavy l'  = balLL v l' r
  | isLeftBig && rightHeavy l' = balLR v l' r
  | isLeftBig                  = balL0 v l' r
  | otherwise                  = node  v l' r
  where
    isLeftBig                  = leftBig l' r
    l'                         = insert a l
```

Exercise 12.6 (InsertRight). ⋆ *The code for* `insR` *is symmetric. To make sure you're following along, why don't you fill it in?*

```
{-@ insR :: x:a
         -> t:{AVL a  | key t < x && 0 < realHeight t }
         -> {v: AVL a | eqOrUp t v}
  @-}
insR = undefined
```

Refactoring Rebalance

Next, lets write a function to `delete` an element from a tree. In general, we can apply the same strategy as `insert`:

1. remove the element without worrying about heights,
2. observe that deleting can *decrease* the height by at most `1`,
3. perform a rotation to fix the imbalance caused by the decrease.

WE PAINTED OURSELVES INTO A CORNER with `insert`: the code for actually inserting an element is intermingled with the code for determining and performing the rotation. That is, see how the code that determines which rotation to apply – `leftBig`, `leftHeavy`, etc. – is *inside* the `insL` which does the insertion as well. This is correct, but it means we would have to *repeat* the case analysis when deleting a value, which is unfortunate.

INSTEAD LETS REFACTOR the rebalancing code into a separate function, that can be used by *both* `insert` and `delete`. It looks like this:

```
{-@ bal :: x:a
        -> l:AVLL a x
        -> r:{AVLR a x | isBal l r 2}
        -> {t:AVL a | reBal l r t}
  @-}
bal v l r
  | isLeftBig  && leftHeavy l  = balLL v l r
  | isLeftBig  && rightHeavy l = balLR v l r
  | isLeftBig                  = balL0 v l r
  | isRightBig && leftHeavy r  = balRL v l r
  | isRightBig && rightHeavy r = balRR v l r
  | isRightBig                 = balR0 v l r
  | otherwise                  = node  v l r
  where
    isLeftBig                  = leftBig l r
    isRightBig                 = rightBig l r
```

The `bal` function is a combination of the case-splits and rotation calls made by `insL` (and ahem, `insR`); it takes as input a value `x` and valid left and right subtrees for `x` whose heights are off by at most `2` because as we will have created them by inserting or deleting a value from a sibling whose height was at most `1` away. The `bal` function returns a valid `AVL` tree, whose height is constrained to satisfy the predicate `reBal l r t`, which says:

- (`bigHt`) The height of `t` is the same or one bigger than the larger of `l` and `r`, and
- (`balHt`) If `l` and `r` were already balanced (i.e. within `1`) then the height of `t` is exactly equal to that of a tree built by directly linking `l` and `r`.

```
{-@ inline reBal @-}
reBal l r t = bigHt l r t && balHt l r t

{-@ inline balHt @-}
balHt l r t = not (isBal l r 1) || isReal (realHeight t) l r

{-@ inline bigHt @-}
bigHt l r t = lBig && rBig
  where
    lBig    = not (hl >= hr) || (eqOrUp l t)
    rBig    = (hl >= hr)     || (eqOrUp r t)
    hl      = realHeight l
    hr      = realHeight r
```

INSERT can now be written very simply as the following function that recursively inserts into the appropriate subtree and then calls `bal` to fix any imbalance:

```
{-@ insert' :: a -> s:AVL a -> {t: AVL a | eqOrUp s t} @-}
insert' a t@(Node v l r n)
  | a < v      = bal v (insert' a l) r
  | a > v      = bal v l (insert' a r)
  | otherwise  = t
insert' a Leaf = singleton a
```

Deleting Elements

Now we can write the `delete` function in a manner similar to `insert`: the easy cases are the recursive ones; here we just delete from the subtree and summon `bal` to clean up. Notice that the height of the output `t` is at most `1` *less* than that of the input `s`.

```
{-@ delete    :: a -> s:AVL a -> {t:AVL a | eqOrDn s t} @-}
delete y (Node x l r _)
  | y < x     = bal x (delete y l) r
  | x < y     = bal x l (delete y r)
  | otherwise = merge x l r
delete _ Leaf = Leaf

{-@ inline eqOrDn @-}
eqOrDn s t = eqOrUp t s
```

THE TRICKY CASE is when we actually *find* the element that is to be removed. Here, we call `merge` to link up the two subtrees `l` and `r` after hoisting the smallest element from the right tree `r` as the new root which replaces the deleted element `x`.

```
{-@ merge :: x:a -> l:AVLL a x -> r:{AVLR a x | isBal l r 1}
          -> {t:AVL a | bigHt l r t}
  @-}
merge _ Leaf r = r
merge _ l Leaf = l
merge x l r    = bal y l r'
  where
   (y, r')     = getMin r
```

`getMin` recursively finds the smallest (i.e. leftmost) value in a tree, and returns the value and the remainder tree. The height of each

remainder `l'` may be lower than `l` (by at most 1.) Hence, we use `bal` to restore the invariants when linking against the corresponding right subtree `r`.

```
getMin (Node x Leaf r _) = (x, r)
getMin (Node x l r _)    = (x', bal x l' r)
  where
    (x', l')             = getMin l
```

Functional Correctness

We just saw how to implement some tricky data structure gynastics. Fortunately, with LiquidHaskell as a safety net we can be sure to have gotten all the rotation cases right and to have preserved the invariants crucial for efficiency and correctness. However, there is nothing in the types above that captures "functional correctness", which, in this case, means that the operations actually implement a collection or set API, for example, as described here. Lets use the techniques from that chapter to precisely specify and verify that our AVL operations indeed implement sets correctly, by:

1. *Defining* the set of elements in a tree,
2. *Specifying* the desired semantics of operations via types,
3. *Verifying* the implemetation.[5]

[5] By adding ghost operations, if needed.

We've done this once before already, so this is a good exercise to solidify your understanding of that material.

The Elements of an `AVL` tree can be described via a measure defined as follows:

```
{-@ measure elems @-}
elems                :: (Ord a) => AVL a -> S.Set a
elems (Node x l r _) = (S.singleton x) `S.union`
                       (elems l)       `S.union`
                       (elems r)
elems Leaf           = S.empty
```

Let us use the above `measure` to specify and verify that our `AVL` library actually implements a `Set` or collection API.

Exercise 12.7 (Membership). *Complete the implementation of the implementation of* `member` *that checks if an element is in an* `AVL` *tree:*

```
-- FIXME https://github.com/ucsd-progsys/liquidhaskell/issues/332
{-@ member :: (Ord a) => x:a -> t:AVL a -> {v: Bool | Prop v <=> hasElem x t} @-}
member x t = undefined

{-@ type BoolP P = {v:Bool | Prop v <=> Prop P} @-}

{-@ inline hasElem @-}
hasElem x t = True
-- FIXME hasElem x t = S.member x (elems t)
```

Exercise 12.8 (Insertion). *Modify* `insert'` *to obtain a function* `insertAPI` *that states that the output tree contains the newly inserted element (in addition to the old elements):*

```
{-@ insertAPI :: (Ord a) => a -> s:AVL a -> {t:AVL a | addElem x s t} @-}
insertAPI x s = insert' x s

{-@ inline addElem @-}
addElem       :: Ord a => a -> AVL a -> AVL a -> Bool
addElem x s t = True
-- FIXME addElem x s t = (elems t) == (elems s) `S.union` (S.singleton x)
```

Exercise 12.9 (Insertion). *Modify* `delete` *to obtain a function* `deleteAPI` *that states that the output tree contains the old elements minus the removed element:*

```
{-@ deleteAPI :: (Ord a) => a -> s:AVL a -> {t: AVL a | delElem x s t} @-}
deleteAPI x s = delete x s

{-@ inline delElem @-}
delElem       :: Ord a => a -> AVL a -> AVL a -> Bool
delElem x s t = True
-- FIXME delElem x s t = (elems t) == (elems s) `S.difference` (S.singleton x)
```