RANJIT JHALA, ERIC SEIDEL, NIKI VAZOU

PROGRAMMING WITH REFINEMENT TYPES

AN INTRODUCTION TO LIQUIDHASKELL

Version 13, July 20th, 2020.

https://ucsd-progsys.github.io/liquidhaskell-blog/

Contents

List of Exercises

1

Introduction

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

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

Well-Typed Programs Do Go Wrong

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

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

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

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

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

What should be the predicate of div to make it impossible to divide by zero?

Yes, e.g., the list [1] Yes, e.g., the list [] No, it should not crash. Submit

Answer

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

Heart Bleeds

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

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

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

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

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

Refinement Types

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

By combining types with predicates you can specify *contracts* which describe valid inputs and outputs of functions. The refinement type system *guarantees at compile-time* that functions adhere to their contracts. That is, you can rest assured that the above calamities *cannot occur at run-time*.

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

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

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

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

To get started, open the Web Demo and see what is the result when you `Check` the code from the first example.

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[2] `c` is simply one of the numeric values:

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

[2] When you see := you should read it as "is defined to be"

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)                -- multiplication by constant
   | (v e1 e2 ... en)       -- uninterpreted function application
   | (if p then e else e)   -- if-then-else
```

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 [3]), `<=>` (if and only if [4]), and `not`.

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

```
p := (e r e)                -- binary relation
   | (v e1 e2 ... en)       -- predicate (or alias) application
   | (p && p)               -- and
   | (p || p)               -- or
   | (p => p) | (p ==> p)   -- implies
   | (p <=> p)              -- iff
   | (not p)                -- negation
   | true | True
   | false | False
```

EXAMPLES OF PREDICATES

Can you select which of the following ones is not a valid predicate?

What should be the predicate of div to make it impossible to divide by zero?

```
x /= 3 x + y <= 3 && y < 1 x < 10 ==> y < 10 ==> x + y < 20
x ** y > 0 0 < x + y <=> 0 < y + x
```

Submit

Answer

All of them are valid syntatic expressions, except for `x ** y > 0` since the operator ** is not part of the language.

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 with `&&`, `||`, `not` being the usual operators and ==> and <=> being two special operators.

A Predicate is Satisfiable if *there exists* an assignment that makes the predicate evaluate to `True`. For example, with the following assignments of x, y and z, the predicate bellow is satisfiable.

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

x + y == z
```

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 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.
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.

THE SMT SOLVER DECIDES whether a predicate (VC) is valid *without enumerating* and evaluating all assignments. The SMT solver uses a variety of sophisticated *symbolic algorithms* to deduce whether a predicate is valid or not.

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, let's 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`:[6]

[6] This syntax will be discussed in greater detail soon

```
{-@ type TRUE  = {v:Bool | v    } @-}
{-@ type FALSE = {v:Bool | not 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' :: FALSE @-}
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`. Write a predicate for ex2 with that meaning:

```
ex2 b = b && not b
```

Answer

The correct answer would be: {-@ ex2 :: Bool -> FALSE @-}

Examples: Arithmetic

Next, let's 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. Run the example and change it to be correct:

```
{-@ 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.

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.

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

```
{-@ measure size @-}
{-@ size :: [a] -> Nat @-}
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`. [7]

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

```
{-@ fx2VC :: _ -> _ -> _ -> TRUE @-}
fx2VC x xs ys =   (0 <= size xs)
              ==> (size ys == 1 + size xs)
              ==> (0 < size ys)
```

Next, let's 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:[1]

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

[1] You can read the type of `Zero` as: "`v` is an `Int` *such that* `v` equals `0`" and `NonZero` as : "`v` is an `Int` *such that* `v` does not equal `0`"

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

[2] We will use `@`-marked comments to write refinement type annotations in 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:

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

LiquidHaskell will complain with an error message:

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

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

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

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

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

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

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

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

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

Subtyping

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

```
{-@ type Nat   = {v:Int | 0 <= v}        @-}
{-@ type 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 [3]

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

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

SUBTYPING AND IMPLICATION

`Zero` is the most precise type for `0::Int`, as it is a *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:

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

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

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

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

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.[4]

[4] 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*

```
cannotDie = 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 contradiction, 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
```

Answer

Add a case for the empty list that does not call upon divide.

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. [5]

[5] 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 | v <=> x > 0} @-}
```

In the above signature, 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.

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.

Include directories can be specified when checking a file. Suppose we want to check some file `target.hs` that imports an external dependency `Data.Vector`. 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 between `0` and the real size of the input vector `x`.

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

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

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

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

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

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

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

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

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

Verification: Vector Lookup

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

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

When we check the above, we get an error:

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

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

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

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.

Note you need to run `liquid` with the option `--no-termination` or make sure your source file has `{-@ LIQUID "--no-termination" @-}`, otherwise the code forgo' fails the now default termination check. We will come back to this example later to see how to verify termination using metrics.

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 then or equal to `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
    body i acc   = undefined
    n            = length vec
```

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
    body i acc = acc + (x ! i)  *  (y ! i)
    sz         = length x
```

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 post-conditions on the outputs. Very often, we wish to define *datatypes* that satisfy certain invariants. In these cases, it is handy to be able to directly refine the `data` definition, making it impossible to create illegal inhabitants.

Sparse Vectors 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} @-}
```

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

SPARSE PRODUCTS

Let's write a function to compute a sparse product

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

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

Folded Product We can port the `fold`-based product to our new representation:

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

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

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)]
```

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, let's see how we can describe properties of aggregate structures like lists and trees, and use these properties to improve the APIs for operating over such structures.

Partial Functions

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

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

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

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

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

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

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

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

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

Uh oh. LiquidHaskell wags its finger at us!

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

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

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

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

Lifting Functions to Measures

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

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

A measure is a *total* Haskell function,

1. With a *single* equation per data constructor, and

2. Guaranteed to *terminate*, typically via structural recursion.

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

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

NON-EMPTY LISTS can now be described as the *subset* of plain old Haskell lists `[a]` for which the predicate `notEmpty` holds

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

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

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

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

AVERAGE is only sensible for non-empty lists. 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        = True   -- 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, its type should say that it works on* all *lists and returns* `False` 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.

Foldl1 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!

```
{-@ foldl1         :: (a -> a -> a) -> NEList a -> a @-}
foldl1 f (x:xs)    = foldl f x xs
foldl1 _ []        = die "foldl1"
```

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

To SUM a non-empty list of numbers, we can just perform a `foldl1` 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  = foldl1 (+) 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       = foldl1 (+)

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) => [a] -> [[a]] @-}
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 let's 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, let's write a small library that represents vectors as lists and matrices as nested vectors:

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

data Matrix a = M { mRow  :: Int
                  , mCol  :: Int
```

```
                , mElts :: Vector (Vector a)
                }
            deriving (Eq)
```

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

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

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 aggregates. For example,

- `dotProd` is only sensible on vectors of the same dimension; if one vector is shorter than another (i.e. has fewer elements) then we

will won't get a run-time crash but instead will get some gibberish result that will be dreadfully hard to debug.

- `matProd` is only well defined on matrices of compatible dimensions; the number of columns of `mx` must equal the number of rows of `my`. Otherwise, again, rather than an error, we will get the wrong output.[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 :: [a] -> Nat @-}
size []     = 0
size (_:rs) = 1 + size rs
```

Measures Refine Constructors

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

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

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

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

We Compose Different Measures

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

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

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 let's define an alias for plain old (unrefined) lists:

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

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

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

Lists: Size Preserving API

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

Exercise 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]

```
{-@ 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"
```

[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.

unsafeZip The signature for `zipWith` is quite severe – it rules out the case where the zipping occurs only up to 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, let's 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 (drop 1 ["cat", "dog", "mouse"])
```

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, let's 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, let's 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, let's just use the `Set a` type from the prelude's `Data.Set`.[1]

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:

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

```
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:

$$x = 1 + 1 \Rightarrow x = 2$$

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, let's do a few warm up 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 | 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 `Bool`eans that are always `True` or `False`

```
{-@ type True  = {v:Bool |     v} @-}
{-@ type False = {v:Bool | not 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:_ | v <=> (P => 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_union_assoc :: _ -> _ -> _ -> 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, let's check that `intersection` distributes over `union`:

```
{-@ 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 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 [a] where
  []  :: ListEmp a
  (:) :: x:a -> xs:[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:[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 _ []     = False
elem x (y:ys) = x == y || elem x ys

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

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

Permutations

Next, let's 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, let's 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:[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:[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:[a] -> ys:[a] -> [a] @-}
merge [] ys          = ys
merge xs []          = xs
merge (x:xs) (y:ys)
  | x <= y           = x : merge xs (y:ys)
  | otherwise        = y : merge (x:xs) ys

{-@ 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 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:[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, let's 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) => [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:[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`*? (When you are done, you should be able to remove the* `assume` *from the signature below, and still have LH verify the code.)*

```
{-@ assume reverse    :: xs:UList a -> UList a    @-}
reverse :: [a] -> [a]
reverse         = go []
  where
    {-@ go     :: acc:[a] -> xs:[a] -> [a] @-}
    go acc []     = acc
    go acc (x:xs) = go (x:acc) 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              :: [a] -> UList a @-}
nub xs                = go [] xs
  where
    {-@ go :: UList a -> xs:[a] -> UList a / [len xs] @-}
    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 | 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 structure that is used to arbitrarily traverse the structure and update its contents. For example, a zipper for a list is a data type that contains an element (called `focus`) that we are currently `focus`-ed on, a list of elements to the `left` of (i.e. before) the focus, and a list of elements to the `right` (i.e. after) the focus.

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

XMONAD is a wonderful tiling window manager, that uses a zipper to store the set of windows being managed. xmonad requires the crucial invariant that the values in the zipper be unique, 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.

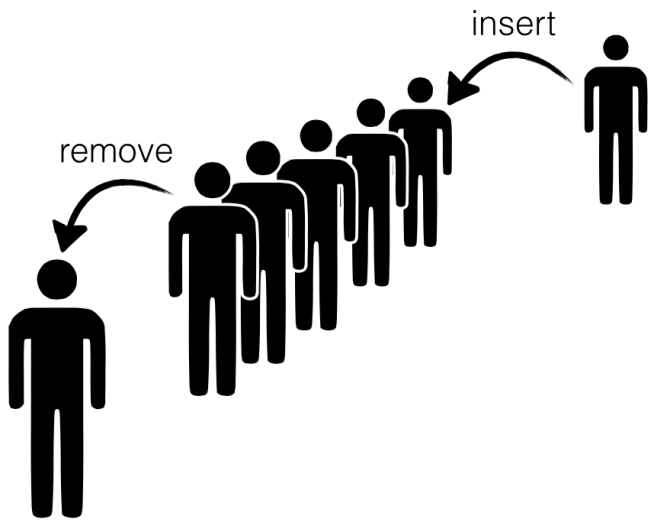

Figure 9.1: A Queue is a structure into which we can insert and remove elements. The order in which the elements are removed is the same as the order in which they were inserted.

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.

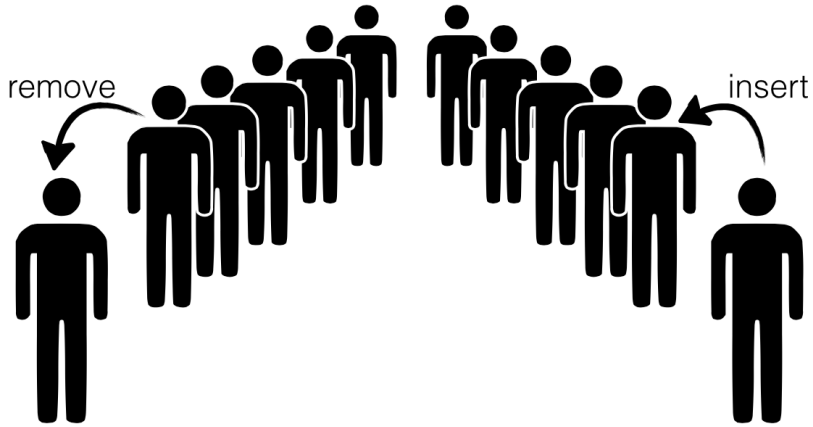

Figure 9.2: We can implement a Queue with a pair of lists; respectively representing the front and back.

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.

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

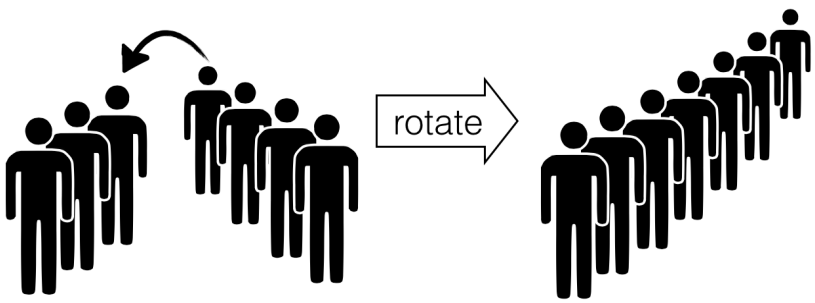

Figure 9.3: Transferring Elements from back to front.

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

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

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

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

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

Sized Lists

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

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

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

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

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

Answer

{-@ measure realSize @-} realSize :: [a] -> Int realSize [] = 0 realSize (_:xs) = 1 + realSize xs
Now, specify a *refined* type for `SList` that ensures that the *real* size is saved in the `size` field. Do it by replacing the questions marks below.

```
{-@ data SList a = SL {
      size  :: Nat
    , elems :: ??
    }
@-}
```

Answer

{-@ 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 on both functions so the definitions typecheck.*

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

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

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

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

okHd  = hd okList       -- accepted

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

Queue Type

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

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

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

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

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

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

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