RANJIT JHALA, ERIC SEIDEL, NIKI VAZOU

PROGRAMMING WITH REFINEMENT TYPES

AN INTRODUCTION TO LIQUIDHASKELL

Version 13, July 20th, 2020.

Copyright © 2024 Ranjit Jhala

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

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

1 Introduction 7

Well-Typed Programs Do Go Wrong 7

Refinement Types 8

2 Logic & SMT 11

Syntax 11

Semantics 13

Verification Conditions 14

Examples: Propositions 14

Examples: Arithmetic 15

Examples: Uninterpreted Function 16

3 Refinement Types 17

Defining Types 17

Errors 18

Subtyping 18

Writing Specifications 20

Refining Function Types: Pre-conditions 20

Refining Function Types: Post-conditions 22

4 Polymorphism 23

Specification: Vector Bounds 24

Verification: Vector Lookup 25

Inference: Our First Recursive Function 27

Sparse Vectors 27

5 Boolean Measures 31

Partial Functions 31

Lifting Functions to Measures 32

6 Numeric Measures 35

Wholemeal Programming 35

Specifying List Dimensions 36

Lists: Size Preserving API 38

7 Case Study: Okasaki's Lazy Queues 39

Queues 39

Sized Lists 41

Queue Type 43

List of Exercises

3.1 Exercise (List Average) 21

4.1 Exercise (Vector Head) 26
4.2 Exercise (Unsafe Lookup) 26
4.3 Exercise (Safe Lookup) 26

5.1 Exercise (Debugging Specifications) 33

6.1 Exercise (Map) 38

7.1 Exercise (Destructing Lists) 43

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*.

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, ...
```

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.

---

[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.

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

```
    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`.

```
    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

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

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

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]

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]

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

[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)
```

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} @-}
```

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]

TO USE these types we can write:

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

{-@ one, two, three :: NonZero @-}
```

---

[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`"

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

```
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
```

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
```

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

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]

Writing Specifications

Let's write some more interesting specifications.

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

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

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

```
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.

---

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

```
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.

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)
```

Specification: Vector Bounds

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

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

IMPORTS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Verification: Vector Lookup

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

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

When we check the above, we get an error:

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

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

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

What is the problem that the message is describing?

It does not know what is the `!` operator The index should be greater than 0 because the head is not accessible Zero is not a valid index if the list is empty. Submit

Answer

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

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

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

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

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

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

Exercise 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
```

INFERENCE

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

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

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

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

Sparse Vectors

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

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

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

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

Legal

`Sparse` vectors satisfy two crucial properties. First, the dimension stored in `spDim` is non-negative. Second, every index in `spElems` must be valid, i.e. between `0` and the dimension. Unfortunately, Haskell's type system does not make it easy to ensure that *illegal vectors are not representable*.[1]

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:

---

[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.

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

but rejects, due to the invalid index:

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

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

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

SPARSE PRODUCTS

Let's write a function to compute a sparse product

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

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

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.

[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.

5

Boolean Measures

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

Partial Functions

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

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

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

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

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

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

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

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

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

Uh oh. LiquidHaskell wags its finger at us!

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

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

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

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

Lifting Functions to Measures

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

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

A MEASURE is a *total* Haskell function,

1. With a *single* equation per data constructor, and
2. Guaranteed to *terminate*, typically via structural recursion.

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

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

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

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

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

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

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

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

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

Answer

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

Exercise 5.1 (Debugging Specifications)**.** *An important aspect of formal verifiers like LiquidHaskell is that they help establish properties not just of your* implementations *but equally, or more importantly, of your* specifications. *In that spirit, can you explain why the following two variants of* `size` *are* rejected *by LiquidHaskell?*

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

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

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

6

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]

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
```

---

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

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

The Dot Product of two `Vectors` can be easily computed using a fold:

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

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

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

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

- `dotProd` is only sensible on vectors of the same dimension; if one vector is shorter than another (i.e. has fewer elements) then we will won't get a run-time crash but instead will get some gibberish result that will be dreadfully hard to debug.

- `matProd` is only well defined on matrices of compatible dimensions; the number of columns of `mx` must equal the number of rows of `my`. Otherwise, again, rather than an error, we will get the wrong output.[2]

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]

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

[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]

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

MEASURES REFINE CONSTRUCTORS

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

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

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

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

WE COMPOSE DIFFERENT MEASURES

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

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

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

---

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

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 6.1 (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
```

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

7

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.

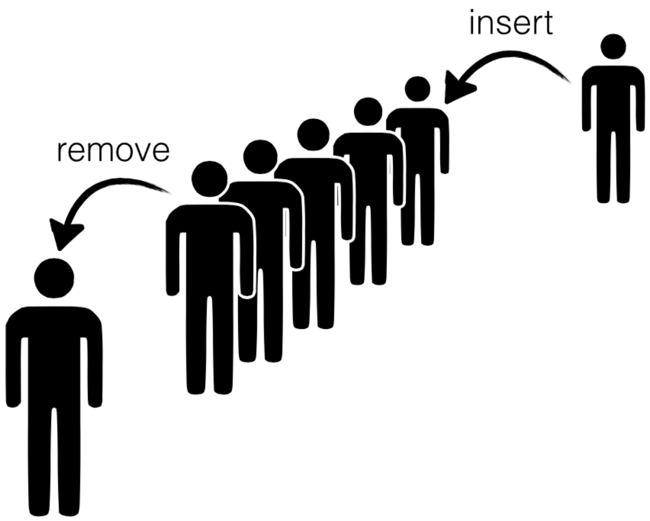

Figure 7.1: A Queue is a structure into which we can insert and remove elements. The order in which the elements are removed is the same as the order in which they were inserted.

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.

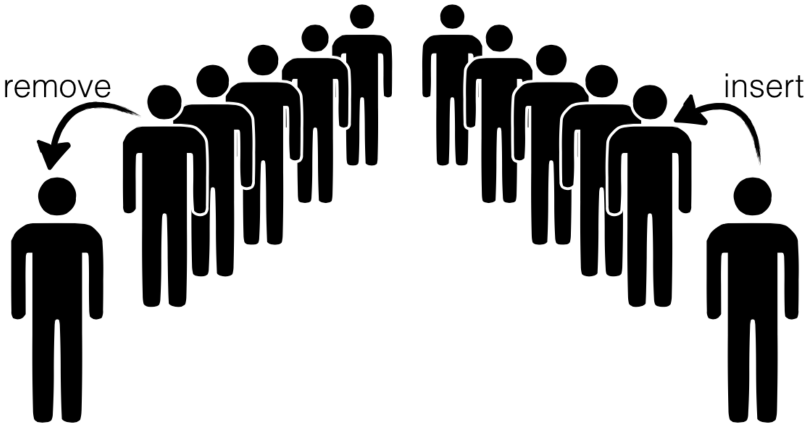

Figure 7.2: We can implement a Queue with a pair of lists; respectively representing the front and back.

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.

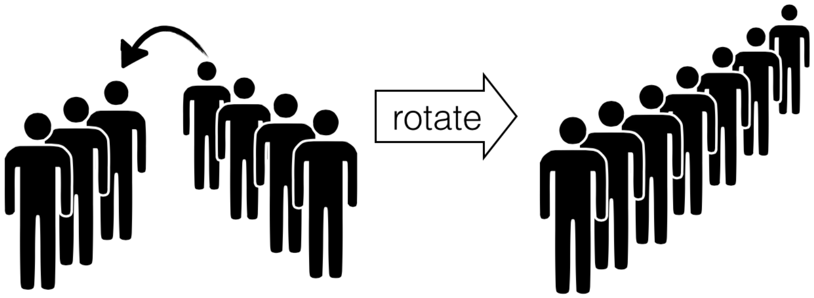

Figure 7.3: Transferring Elements from back to front.

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

Sized Lists

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

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

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

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

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

Answer

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

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

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

As a sanity check, consider this:

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

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

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

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

NOW DEFINE AN ALIAS for lists that are not empty:

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

Answer

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

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

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

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

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

Exercise 7.1 (Destructing Lists)**.** *We can destruct lists by writing a* `hd` *and* `tl` *function as shown below. Now, fix the specification on both functions so the definitions typecheck.*

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

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

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

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

okHd  = hd okList       -- accepted

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

Answer

{-@ tl :: xs:NEList a -> SListN a {size xs - 1} @-} tl (SL n (_:xs)) = SL (n-1) xs

{-@ hd :: xs:NEList a -> a @-} hd (SL _ (x:_)) = x

Queue Type

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

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

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

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

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

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

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