Solid Code with Liquid Types

An introduction to Liquid Haskell

Pablo Castellanos & Alessio Ferrarini

23rd October — Lambda World 2025

Contents

1 Motivation

1.1 The untyped world

In the untyped[1] world of programming languages, we can write a function that sums all the elements of a list:

```python
def sumlist(xs):
    total = 0
    for x in xs:
        total += x
    return total
```

[1]Actually, there are no un-typed languages: languages like Python, JavaScript, Ruby, etc. are really **mono**-typed.

And we can call it with a list of numbers:

```
print(sumlist([1, 2, 3, 4]))
```

```
10
```

But we operate under the assumption that `xs` is always a list of numbers. As an example if we call `sumlist("Hello, Cadiz!")` we will receive a runtime error.

```
try:
    print(sumlist("Hello, Cadiz!"))
except Exception as e:
    print(e)
```

```
unsupported operand type(s) for +=: 'int' and 'str'
```

So we can either trust ourselves, our colleagues, or the users of our code to always call `sumlist` with a list of numbers, or we can add some checks to ensure that the input is valid:

```
def sumlist(xs):
    if not isinstance(xs, list):
        raise ValueError("xs must be a list")
    for x in xs:
        if not isinstance(x, (int, float)):
            raise ValueError("all elements of xs must be numbers")

    total = 0
    for x in xs:
        total += x
    return total
```

Now if we call `sum("Hello, Cadiz!")` we will get a more informative error:

```
try:
  sumlist("Hello, Cadiz!")
except Exception as e:
  print(e)
```

```
xs must be a list
```

Having a program crash unexpectedly in production with an "`unsupported operand type(s)`" error is bad enough, but there are many other ways in which programs can go wrong. For instance, when accessing a position in a list, the index must be smaller than the length of the list:

```
xs = [1,2,3]

try:
    print(xs[7])
except Exception as e:
    print(e)
```

```
list index out of range
```

In this case, the Python interpreter makes sure that the index is valid while the program is running; other languages, like C, don't do this check, which can have serious consequences: for example, this was the cause of the *Heartbleed* vulnerability[2] that went undetected for more than two years and which allowed attackers to access unauthorized information.

Of course, this problem can be fixed with a simple check on the length:

```
xs = [1,2,3]

try:
    if 7 >= len(xs):
        raise ValueError("The index 7 is too big")
    else:
        print(xs[7])
except Exception as e:
    print(e)
```

```
The index 7 is too big
```

This approach is often called **defensive programming**, and while it certainly has its merits, it also has its downsides:

- It clutters our code with checks that are not part of the core logic of our program.
- It pushes the error to runtime, which means that we will only discover the error when we run the code, which might be in production.
- We need to know ahead of time what kind of errors we want to catch, and add checks for them.

1.2 The typed world

But if you're here at Lambda World I hope I don't need to convince you that types are a good thing. In a typed language like Haskell or Scala we have a type system that can help us catch these errors at compile time. For example in Haskell we can write:

```
sumlist :: [Int] -> Int
sumlist []     = 0
sumlist (x:xs) = x + sumlist xs
```

And as before we can call it with a list of numbers:

```
print $ sumlist [1, 2, 3, 4]
```

```
10
```

But if we try to call it with a string we will get a compile time error:

[2]https://en.wikipedia.org/wiki/Heartbleed

```haskell
print $ sum "Hello, Cadiz!"
```

```
<interactive>:326:9-11: error: [GHC-39999]
    • No instance for ‘Num Char’ arising from a use of ‘sum’
    • In the second argument of ‘($)’, namely ‘sum "Hello, Cadiz!"’
      In the expression: print $ sum "Hello, Cadiz!"
      In an equation for ‘it’: it = print $ sum "Hello, Cadiz!"
```

1.3 Are we typed enough?

But are we typed enough? What if we want to extract the first element of a list? We can write:

```haskell
head :: [a] -> a
head (x:_) = x
```

But the function `head` is partial, and if we call it with an empty list we will get a runtime error:

```haskell
print $ head []
```

```
*** Exception: <interactive>:52:1-14: Non-exhaustive patterns in function head
```

So we can go back to defensive programming and write:

```haskell
head :: [a] -> a
head []    = error "empty list"
head (x:_) = x
```

```haskell
print $ head []
```

```
*** Exception: empty list
```

But we are back to the same problems as before.

We can do better, we can use the type system to make the error part of the type system. We can use the `Maybe` type to indicate that the function might fail:

```haskell
head :: [a] -> Maybe a
head []    = Nothing
head (x:_) = Just x
```

```haskell
print $ head []
```

```
Nothing
```

But this comes with its own problems, now the user of `head` has to handle the `Nothing` case. This can be mitigated by using monadic programming and do notation, but it still clutters the code with boilerplate. In principle we would like to avoid the `Nothing` completely and push the error boundaries to the edges of our program.

The seasoned Haskeller knows the slogan **"Make illegal states unrepresentable"** and uses the type system to ensure that the list is not empty:

```haskell
data NonEmptyList a = Singleton a | Cons a (NonEmptyList a)

head :: NonEmptyList a -> a
head (Singleton x) = x
head (Cons x _)    = x
```

There is no possible way to construct an empty `NonEmptyList` so `head` is total and we can call it without worrying about runtime errors:

```haskell
print $ head (Cons 1 (Singleton 2))
```

```
1
```

But also this comes with its own problems, now we have to duplicate all the functions that work on lists to work on non-empty lists.

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

```haskell
print $ map (+1) (Cons 1 (Singleton 2))
```

```
<interactive>:232:19-38: error: [GHC-83865]
    • Couldn't match expected type: [b0]
                  with actual type: NonEmptyList a0
    • In the second argument of ‘map’, namely ‘(Cons 1 (Singleton 2))’
      In the second argument of ‘($)’, namely
        ‘map (+ 1) (Cons 1 (Singleton 2))’
      In the expression: print $ map (+ 1) (Cons 1 (Singleton 2))
```

```haskell
mapNE :: (a -> b) -> NonEmptyList a -> NonEmptyList b
mapNE f (Singleton x) = Singleton (f x)
mapNE f (Cons x xs)   = Cons (f x) (map f xs)
```

Is there a better solution?

1.4 Refinement types

`NonEmptyList` felt like the right solution, but it doesn't scale. We would have to define a new type for every possible invariant that we want to enforce. What if we want to ensure that a list has at least `n` elements? Do we have to define a new type for every possible `n`?

Refinement types extend the type system by allowing us to attach predicate to types. Usually they are written as:

$$\{v : T \mid \phi(v)\}$$

Where T is a type, v is a variable that represents the values of type T, and $\phi(x)$ is a predicate that must hold for all values.

To explain it better let's go back to our original example of the non empty list. We can define the type of non-empty lists as:

$$\{v : [a] \mid \text{length}(v) > 0\}$$

These types also support subtyping so we can say that:

$$\{v : [a] \mid \text{length}(v) > 3\} \preceq \{v : [a] \mid \text{length}(v) > 0\}$$

As any list with more than 3 elements is also a non-empty list.

The subtyping relationship is carried through logical implication:

$$\frac{\forall v.\ \phi_1(v) \Rightarrow \phi_2(v)}{\{v : T \mid \phi_1(v)\} \preceq \{v : T \mid \phi_2(v)\}} \quad \text{(Subtyping)}$$

2 Haskell and Liquid Haskell basics

```
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}
```

2.1 Basic idea

Liquid Haskell is a refinement type system for Haskell. It is implemented as an extension to GHC, and it uses SMT solvers to check the validity of the refinement types. All Liquid Haskell code is still valid Haskell code, and can be compiled and run without Liquid Haskell.

Let's see some examples of how to use Liquid Haskell. We start with a simple Haskell definition:

```
positiveInteger :: Int
positiveInteger = 42
```

This is a standard Haskell definition of a variable, the first line is its type signature, `name :: type`, and the second line is its definition, `name = value`.

Now `positiveInteger` is just an integer, and we can assign any integer even a negative one contrary to its name:

```
positiveInteger' :: Int
positiveInteger' = -1
```

But we can use refinement types to ensure that the value of `positiveInteger` is always positive, Liquid Haskell annotations are written as comments that start with `{-@` and end with `@-}`, inside the comment we can write the refinement type annotation:

```haskell
{-@ positiveInteger'' :: {v:Int | v > 0} @-}
positiveInteger'' :: Int
positiveInteger'' = 42
```

We repeat the type signature but now instead of writing just `Int` we write `{v:Int | v > 0}`, which means that type is an integer called `v` such that the expression `v > 0` is true.

Now to test it out we can try to declare another variable with the same type but with a negative value:

```haskell
{-@ positiveInteger''' :: {v:Int | v > 0} @-}
positiveInteger''' :: Int
positiveInteger''' = -1
```

And it will not compile, saying that the type of integers that are equal to -1 is not a subtype of the type of integers that are greater than 0:

Liquid Type Mismatch

```
The inferred type
{v : GHC.Types.Int | v == (-(1 : int))}

is not a subtype of the required type
{v : GHC.Types.Int | v > 0}


  positiveInteger''' = -1
  ^^^^^^^^^^^^^^^^^^^^^^^
```

In the same way as we declare variables, we can declare functions: We start with the type signature, here → is the type constructor for functions, and when defining the function after the name we can use introduce the arguments of the function:

```haskell
percentage :: Int -> Int -> Int
percentage total part = (part * 100) `div` total
```

Now this declaration in Haskell is perfectly valid, but it has a pretty big flaw: calling the function like

```haskell
percentage 0 10
```

gives a run-time exception:

Liquid Type Mismatch

```
  *** Exception: divide by zero
```

But when using Liquid Haskell we would have already being stopped at compile time, when writing the definition of `percentage`:

Liquid Type Mismatch

```


    The inferred type
      {v : GHC.Types.Int | v == total}

    is not a subtype of the required type
    {v : GHC.Types.Int | v ≠ 0}

  in the context
    total  : GHC.Types.Int


   percentage total part = (part * 100) `div` total
                                             ^^^^^
```

We can fix this by refining the type of `percentage` to ensure that the first argument is not zero:

```
{-@ percentage :: {total:Int | total ≠ 0} -> Int -> Int @-}
```

Now if we try to call `percentage` with a zero as the first argument we will get a compile time error:

```
percentage 0 10
```

```
                              Liquid Type Mismatch
  The inferred type
    {v : GHC.Types.Int | v == (0 : int)}

  is not a subtype of the required type
    {v : GHC.Types.Int | v ≠ 0}


   percentage 0 10
              ^
```

Refinement types are not limited to ensuring correctness in the sense of not crashing, they can also be used to ensure some logical properties coming from the domain of the problem. For example we can ensure that `part` is always less than or equal to `total` and that both `total` and `part` are non negative:

```
{-@ percentage :: {total:Int | total ≥ 0} -> {part:Int | part ≤ total && part ≥ 0}
               -> Int @-}
```

Notice how we can refer tot `total` in the refinement of `part`.

And now we can ensure that the result of `percentage` is always between 0 and 100:

```
{-@ percentage :: {total:Int | total ≥ 0} -> {part:Int | part ≤ total && part ≥ 0}
               -> {r:Int | r ≥ 0 && r ≤ 100} @-}
```

We define aliases for refinement types using the `type` keyword:

```
{-@ type Positive      = {v:Int | v >  0} @-}
{-@ type NonNegative   = {v:Int | v ≥ 0} @-}
{-@ type NonZero       = {v:Int | v ≠ 0} @-}
{-@ type Between L U   = {v:Int | v ≥ L && v ≤ U} @-}
```

And use them in our annotations:

```
{-@ percentage :: total:Positive -> part:Between 0 total -> Between 0 100 @-}
```

In Haskell we can also define function using pattern matching:

```
head :: [a] -> a
head (x:_) = x
```

Which is equivalent to:

```
head :: [a] -> a
head xs = case xs of
  (x:_) -> x
```

But Liquid Haskell will reject both definitions:

```
                                  Liquid Type Mismatch
    The inferred type
      {v : GHC.Prim.Addr# | v == "Main.hs:23:1-14|function head"}

    is not a subtype of the required type
      {v : GHC.Prim.Addr# | totalityError "Pattern match(es) are non-exhaustive"}


     head (x:_) = x
     ^^^^^^^^^^^^^^
```

As they are not total.

2.2 Lists

In Haskell we can define ADTs[3] using the `data` keyword. For example we can define a list as:

```
data List a = Nil | Cons a (List a)
```

And define functions that operate on them by pattern matching:

```
{-@ length :: List a -> Nat @-}
length :: List a -> Int
length Nil         = 0
length (Cons _ xs) = 1 + length xs
```

Functions with only one argument that are defined by structural induction can be made part of the refinement logic with the `measure` annotation:

```
{-@ measure length @-}
```

Now we can defined the head function, safely:

[3]ADTs is probably the most useless acronym in programming, it stands for Algebraic Data Types or Abstract Data Types depending on who you ask and anyway they are not Algebraic nor Abstract.

```
{-@ head :: {xs:List a | length xs > 0} -> a @-}
head :: List a -> a
head (Cons x _) = x
```

We can define the take function:

```
{-@ take :: n:Nat -> {xs:List a | length xs ≥ n}
         -> List a @-}
take :: Int -> List a -> List a
take 0 _           = Nil
take n (Cons x xs) = Cons x (take (n - 1) xs)
```

Exercise

Refine the type of the function `take` to ensure that the result has length equal to `n`.

Solution

```
{-@ take :: n:Nat -> {xs:List a | length xs ≥ n}
         -> {ys:List a | length ys == n} @-}
```

Exercise

Define the function `drop` with the appropriate refinement type.

Solution

```
{-@ drop :: n:Nat -> {xs:List a | length xs ≥ n}
         -> {ys:List a | length ys == length xs - n} @-}
drop :: Int -> List a -> List a
drop 0 xs           = xs
drop n (Cons _ xs) = drop (n - 1) xs
```

Exercise

Define the function `lookup` that takes an index and a list and returns the element at that index, with the appropriate refinement type.

Solution

```
{-@ lookup :: i:Nat -> {xs:List a | length xs > i} -> a @-}
lookup :: Int -> List a -> a
lookup 0 (Cons x _)  = x
lookup n (Cons _ xs) = lookup (n - 1) xs
```

Exercise

Define the function `append` that takes two lists and returns the concatenation of the two lists, with the appropriate refinement type.

Solution

```
{-@ append :: xs:List a -> ys:List a
           -> {zs:List a | length zs == length xs + length ys} @-}
append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Exercise

Define the function `map` that takes a function and a list and returns the list obtained by applying the function to each element of the list, with the appropriate refinement type.

Solution

```
{-@ map :: (a -> b) -> xs:List a -> {ys:List b | length ys == length xs} @-}
map :: (a -> b) -> List a -> List b
map _ Nil         = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

2.3 Basic theorem proving and proof automation

We can also use Liquid Haskell to prove properties about our functions. First we enable the reflection flag:

```
{-@ LIQUID "--reflection" @-}
```

Now we can prove interesting properties about our functions, for example the function `map` that we defined before as an exercise is part of the functor typeclass[4] and must satisfy the functor laws[5].

First we need to be able to talk about the function `map` itself in the refinement types, but it is not defined as a measure, so we need to reflect it:

```
{-@ reflect map @-}
```

Now we can state the first functor law, that mapping the identity function over a list is the same as the original list:

```
{-@ reflect id @-}
id :: a -> a
id x = x

{-@ fmapId :: xs:List a -> { map id xs == xs } @-}
fmapId :: List a -> Proof
```

The type `Proof` is used to indicate that the function is only used for proving properties; in reality, it is just the unit type `()` under the hood, but we use it for clarity.

As the function `map` is defined by structural induction, we can prove the property by structural induction on the list.

[4]"Typeclass" is Haskell's name, but the concept is popular in many other languages under different names like trait or interface.
[5]If `Hask` was actually a category they would be validated by parametricity.

One of the ways to prove properties in Liquid Haskell is by equational reasoning, we can use the === operator to indicate that two expressions are equal.

Let's start with the base case, the empty list:

```
fmapId Nil
  =   map id Nil
  === Nil
  *** QED
```

We take the original expression and we rewrite it using the definition of `map`, and then we use the fact that `map` of the empty list is the empty list. We terminate the proof with `*** QED`. Liquid Haskell will check each step of the proof to ensure that it is valid, and that together they constitute a proof of our theorem.

As an example if we put a wrong step in the proof:

```
fmapId Nil
  =   map id Nil
  === Cons 1 Nil
  *** QED
```

```
                                    Liquid Type Mismatch
     === Cons 1 Nil
         ^^^^^^^^^^
```

Liquid Haskell will complain that the two sides of the equation are not equal.

Now we can do the inductive case, where the list is not empty:

```
fmapId (Cons x xs)
  = map id (Cons x xs)
  === Cons (id x) (map id xs)
  === Cons x (map id xs) ? fmapId xs
  === Cons x xs
  *** QED
```

Note that we use the `?` operator to indicate that we are using a previously proven lemma, in this case the inductive hypothesis, to go from one step to the next.

Now we can state and prove the second functor law, that mapping the composition of two functions is the same as first mapping one and then the other:

```
{-@ infix . @-}
{-@ reflect (.) @-}
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

{-@ fmapCompose :: f:(b -> c) -> g:(a -> b) -> xs:List a
                -> { map f (map g xs) = map (f . g) xs } @-}
fmapCompose :: (b -> c) -> (a -> b) -> List a -> Proof
```

As before we prove it by structural induction on the list.

```
fmapCompose f g xs = case xs of
  Nil          -> map f (map g Nil)
              === map f Nil
              === Nil
              === map (f . g) Nil
              *** QED
  (Cons x xs) -> map f (map g (Cons x xs))
              === map f (Cons (g x) (map g xs))
              === Cons (f (g x)) (map f (map g xs))
              === Cons ((f . g) x) (map f (map g xs))
                ? fmapCompose f g xs
              === Cons ((f . g) x) (map (f . g) xs)
              === map (f . g) (Cons x xs)
              *** QED
```

Exercise

Given `data Pair a b = Pair a b`, define the map function for `Pair` with respect to one of the type parameters, and prove that it satisfies the functor laws.

Solution

```
data Pair a b = Pair a b

{-@ reflect mapFst @-}
{-@ mapFst :: (a -> c) -> Pair a b -> Pair c b @-}
mapFst :: (a -> c) -> Pair a b -> Pair c b
mapFst f (Pair x y) = Pair (f x) y

{-@ fmapFstId :: p:Pair a b -> { mapFst id p == p } @-}
fmapFstId :: Pair a b -> Proof
fmapFstId (Pair x y)
  =   mapFst id (Pair x y)
  === Pair (id x) y
  === Pair x y
  *** QED

{-@ fmapFstCompose :: f:(a -> c) -> g:(d -> a) -> p:Pair d b
                   -> { mapFst f (mapFst g p) = mapFst (f . g) p } @-}
fmapFstCompose :: (a -> c) -> (d -> a) -> Pair d b -> Proof
fmapFstCompose f g (Pair x y)
  =   mapFst f (mapFst g (Pair x y))
  === mapFst f (Pair (g x) y)
  === Pair (f (g x)) y
  === Pair ((f . g) x) y
  === mapFst (f . g) (Pair x y)
  *** QED
```

Exercise

Sate and prove the fact that append has the empty list as the identity element on the right side.

Solution

Right identity for any list xs means `append xs Nil == xs`.

```haskell
{-@ appendNilR :: xs:List a -> { append xs Nil == xs } @-}
appendNilR :: List a -> Proof
appendNilR xs = case xs of
  Nil         -> append Nil Nil
             === Nil
             *** QED
  (Cons x xs) -> append (Cons x xs) Nil
             === Cons x (append xs Nil)
               ? appendNilR xs
             === Cons x xs
             *** QED
```

Now these proofs are very long and tedious, but they can be automated away using Liquid Haskell's only tactic—*Proof by Logical Evaluation*, or `PLE` for short—that will perform the equational reasoning for us.

```haskell
{-@ LIQUID "--ple" @-}
```

And now we can prove the same properties without writing the steps:

```haskell
{-@ fmapCompose' :: f:(b -> c) -> g:(a -> b) -> xs:List a
                 -> { map f (map g xs) = map (f . g) xs } @-}
fmapCompose' :: (b -> c) -> (a -> b) -> List a -> Proof
fmapCompose' f g xs = case xs of
        Nil         -> trivial
        (Cons x xs) -> fmapCompose' f g xs
```

The `trivial` in the base case means the proof can be done just by expanding the function definitions, as we did manually before (in reality, it is just an alias for the unit value `()`). In the recursive case, we only need to provide the lemmas we used when simple equational reasoning was not enough (that is, whatever information we wrote to the right of the `?` operator).

Exercise

Use PLE to redo the proofs that you did before.

Solution

```
{-@ fmapId' :: xs:List a -> { map id xs == xs } @-}
fmapId' :: List a -> Proof
fmapId' xs = case xs of
        Nil         -> trivial
        (Cons x xs) -> fmapId' xs

{-@ mapFstId' :: p:Pair a b -> { mapFst id p == p } @-}
mapFstId' :: Pair a b -> Proof
mapFstId' (Pair x y) = trivial

{-@ fmapFstCompose' :: f:(a -> c) -> g:(d -> a) -> p:Pair d b
                   -> { mapFst f (mapFst g p) = mapFst (f . g) p } @-}
fmapFstCompose' :: (a -> c) -> (d -> a) -> Pair d b -> Proof
fmapFstCompose' f g (Pair x y) = trivial

{-@ appendNilR' :: xs:List a -> { append xs Nil == xs } @-}
appendNilR' :: List a -> Proof
appendNilR' xs = case xs of
        Nil         -> trivial
        (Cons x xs) -> appendNilR' xs
```

2.4 A note on termination

2.4.1 Termination checking

Most of the time, we want our programs to terminate in a finite amount of time, so that e.g. infinite loops are usually an indication that we did something wrong. Wouldn't it be great if the compiler was able to warn us when our programs contain a loop that will never exit?

The bad news is that Alan Turing proved the *Halting Problem*[6] to be undecidable (i.e. there will never be an algorithm that can determine if our program will halt in a finite amount of time or run forever). The good news is that tools like Liquid Haskell can detect when your program has an infinite loop:

```
{-@ wrongFactorial :: Nat -> Nat @-}
wrongFactorial :: Int -> Int
wrongFactorial 0 = 1
wrongFactorial n = n * wrongFactorial n
```

```
Liquid Type Mismatch
The inferred type
  {v : GHC.Types.Int | v == n && v ≥ 0}

is not a subtype of the required type
  {v : GHC.Types.Int | v < n}
.
in the context
  n : {n : GHC.Types.Int | n ≥ 0 && n ≠ 0}
```

[6] https://en.wikipedia.org/wiki/Halting_problem

```
Constraint id 4
```

1. What is going on here?
2. How is this possible? Doesn't this contradict Turing's theorem?

To answer the second question, remember that what Turing proved is that there can't exist an algorithm that rejects all non-terminating programs and simultaneously accepts all terminating programs. However, it is very easy to think of an algorithm that rejects all **non-terminating** programs but *is allowed to be wrong* about terminating ones:

```haskell
halts :: Program -> Bool
halts p = False
```

The only difference is that Liquid Haskell is a bit more clever in that it tries to reduce the amount of terminating programs that are incorrectly flagged as non-terminating.

This brings us back to the first question: how does Liquid Haskell do it? It tries to prove that there is some argument that

1. gets "smaller" in every recursive call (for some definition of "small"), and
2. cannot get arbitrarily small (its size is bounded below by 0).

It is easy to see why these two conditions together imply that the function terminates. Thanks to this, any function that has this property will be accepted by Liquid Haskell and so we can be sure that it will indeed terminate.

In some cases, proving these conditions may be too cumbersome. For this reason, Liquid Haskell provides the "lazy" annotation to disable termination checking for a particular function, as in

```haskell
{-@ lazy <name of the function> @-}
```

as well as the "—`no-termination`" pragma that affects all functions in that file.

As a rule of thumb, Liquid Haskell tries to prove these two conditions about the first argument of the function that can be given a notion of a "size". In the case of integers, the size is the value of the number itself. This is why, in the previous example, Liquid Haskell complained that `n` is not strictly smaller than `n`:

```
Liquid Type Mismatch
  The inferred type
    {v : GHC.Types.Int | v == n && v ≥ 0}

  is not a subtype of the required type
    {v : GHC.Types.Int | v < n}
  .
  in the context
    n : {n : GHC.Types.Int | n ≥ 0 && n ≠ 0}
  Constraint id 4
```

It is able to prove that the input is bounded below by 0 (this is what it means for the input n to be a Nat) but it fails to prove that the recursive call is made with a smaller number (which, indeed is not the case, even though it should be): there is a mistake in the code.

By fixing the recursive call in the second case, we get rid of the error and Liquid Haskell can check that the function terminates:

```haskell
{-@ factorial :: Nat -> Nat @-}
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

2.4.2 Termination measures

Liquid Haskell's rule of thumb has been enough to prove termination for every function we have seen so far, but this is not always the case. Let's look now at the range function that, given integers a and b, returns the range [a..b) as a list:

```haskell
{-@ range :: a:Int -> b:{Int | b ≥ a} -> [{n:Int | a ≤ n && n < b}] @-}
range :: Int -> Int -> [Int]
range a b = if a == b then [] else a : range (a+1) b
```

```
                                        Liquid Type Mismatch
  The inferred type
    {v : GHC.Types.Int | v == a + 1}

  is not a subtype of the required type
    {v : GHC.Types.Int | v < a}
  .
  in the context
    b : {b : GHC.Types.Int | b ≥ a}

    a : GHC.Types.Int

    ?b : {?b : GHC.Types.Bool | not (a == b)}
  Constraint id 11
```

Liquid Haskell is trying to prove that the first argument a is getting smaller, but it's not. In fact, the reason why we know that this function terminates is that the range (the difference between a and b) is getting smaller, but neither a nor b decreases. We can indicate this to Liquid Haskell by adding / [<quantity that decreases>] at the end of the refined signature:

```haskell
{-@ range :: a:Int -> b:{Int | b ≥ a} -> [{n:Int | a ≤ n && n < b}]
           / [b-a] @-}
range :: Int -> Int -> [Int]
range a b = if a == b then [] else a : range (a+1) b
```

This new quantity is called the **termination measure** for the function, and it must satisfy the same conditions as before:

1. `b-a` is getting smaller in the recursive call because we are excluding the first element `a`.

2. `b-a` is bounded below by 0 because we stated in the function precondition that `b ≥ a`.

When we add the termination measure as above, Liquid Haskell is able to prove that the function terminates.

Because the Halting Problem is undecidable, there is no hope of implementing an algorithm that always *finds* termination measures for us; however, if we can find them *on our own*, Liquid Haskell will happily check if they are indeed valid termination measures for the function or not.

2.4.3 Termination for data structures

More complex datatypes need a different notion of "size": the size of the *structure* itself.

As a simple example, let's take a look at the `List` data structure:

```
data List a = Nil | Cons a (List a)
```

The empty list `Nil` can't possibly get any smaller, so we can treat it as our "lower bound". The `Cons` constructor, on the other hand, takes a list and adds an element to it creating a bigger list: in this sense, we say that the list `Cons x xs` is *structurally larger* than the list `xs` that appears inside it / as a part of it. Liquid Haskell can deal with this notion of structural size automatically, in the same way as it did for integers; this is why the definition of the `append` function can be proven to terminate without any help from us:

```
append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

However, if we use `ys` instead of `xs` in the recursive call by mistake, Liquid Haskell will rightly complain that there is no parameter that is getting smaller:

```
append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append ys ys)
```

```
Liquid Type Mismatch
  Termination.hs:22:1: error:
      Termination Error
  Termination.append
  No decreasing parameter
```

2.4.4 Termination in proofs

Infinite loops in programs can be a big problem, but in math they completely break the validity of a proof.

Suppose that we want to prove that all integers are equal to zero. We proceed like this:

Theorem 1. Let $n \in \mathbb{Z}$ be an integer. Then, $n = 0$.

Proof: Take n to be any integer. By Theorem 1, we can conclude that n is indeed equal to 0. This completes the proof. □

Of course, this is not a valid proof because it is **circular**: it refers back to something that has not been proven yet. If we allowed non-terminating proofs in Liquid Haskell, we could easily encode this kind of proofs, so we would be able to prove any statement at all, even false ones:

```
{-@ lazy nIsZero @-}
{-@ nIsZero :: n:Int -> { n == 0 } @-}
nIsZero :: Int -> Proof
nIsZero n = nIsZero n
```

Of course, by removing the `lazy` annotation Liquid Haskell complains about non-termination, which is just the circularity of the argument in disguise:

```
{-@ nIsZero :: n:Int -> { n == 0 } @-}
nIsZero :: Int -> Proof
nIsZero n = nIsZero n
```

```
                         Liquid Type Mismatch
The inferred type
  {v : GHC.Types.Int | v == n}

is not a subtype of the required type
  {v : GHC.Types.Int | v < n}
.
in the context
  n : GHC.Types.Int
Constraint id 4
```

2.4.5 Lexicographic termination & mutual recursion

Sometimes, the termination of a function can't be proven just by looking at one of the arguments. This is the case of the Ackermann function[7], where either the first argument decreases, or it stays the same while the second one decreases:

```
ack :: Int -> Int -> Int
ack 0 n = n + 1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

This termination measure based on the "lexicographic" order of the arguments can be given to Liquid Haskell by separating the measures with a comma:

```
{-@ ack :: m:Nat -> n:Nat -> Nat / [m,n] @-}
```

[7]https://en.wikipedia.org/wiki/Ackermann_function

Exercise

Why does the follow implementation of the greatest common divisor terminate? Find a termination measure.

```
{-@ gcd :: a:Nat -> b:Nat -> Int @-}
gcd :: Int -> Int -> Int
gcd 0 b = 0
gcd a 0 = a
gcd a b | a == b = a
        | a >  b = gcd (a - b) b
        | a <  b = gcd a (b - a)
```

Solution

```
{-@ gcd :: a:Nat -> b:Nat -> Int / [a,b] @-}
```

A particular case of lexicographic termination measures comes with mutually recursive functions:

```
{-@ isEven :: Nat -> Bool @-}
isEven :: Int -> Bool
isEven 0 = True
isEven n = isOdd (n-1)

{-@ isOdd :: Nat -> Bool @-}
isOdd :: Int -> Bool
isOdd n = not (isEven n)
```

`isEven` and `isOdd` depend on one another, so how can we be sure that any call to either function is guaranteed to terminate?

- `isEven` calls `isOdd` with a decreased value for `n`, so the parameter decreases sometimes.
- however, `isOdd` calls `isEven` with the same value it got, so the parameter does not decrease in this case.

We can solve this by establishing an "order" between the functions: if we give the value 2 to `isOdd` and the value 1 to `isEven`, whenever `isOdd` calls `isEven` we can see that the (arbitrary) value that we made up decreases in the mutually recursive call.

```
{-@ isEven :: n:Nat -> Bool / [n, 1] @-}
isEven :: Int -> Bool
isEven 0 = True
isEven n = isOdd (n-1)

{-@ isOdd :: n:Nat -> Bool / [n, 2] @-}
isOdd :: Int -> Bool
isOdd n = not (isEven n)
```

All in all, this is saying that either the argument `n` gets smaller, or it stays the same while the value we gave the functions goes down.

2.5 Deep verification

Refinement types can be used to prove that our functions do exactly what we want them to do.

As an example, let's take this implementation of the `rotate` function:

```haskell
{-@ reflect rotate @-}
{-@ rotate :: xs:List a -> n:Between 0 (length xs) -> List a @-}
rotate :: List a -> Int -> List a
rotate xs n = append (drop n xs) (take n xs)
```

What do we want the `rotate` function to do? Intuitively, the element at the `i`-th position of the rotated list should be the same as the element at the `(i+n)`-th position of the original list, wrapping back to the start of the list when necessary:

Exercise

```haskell
{-@ rotateCorrect :: xs:List a -> n:Between 0 (length xs)
                  -> i:Between 0 (length xs)
                  -> {lookup i (rotate xs n) = lookup ((i+n) mod (length xs)) xs} @-}
rotateCorrect :: List a -> Int -> Int -> Proof
rotateCorrect xs n i = let ys = take n xs; zs = drop n xs in undefined
```

Can you find a proof of this theorem?
Hint: you may find these lemmas useful:

```haskell
{-@ lookupAppendLeft :: xs:List a -> ys:List a -> i:Between 0 (length xs)
                     -> {lookup i (append xs ys) = lookup i xs} @-}
lookupAppendLeft :: List a -> List a -> Int -> Proof
lookupAppendLeft (Cons x xs) ys 0 = undefined
lookupAppendLeft (Cons _ xs) ys i = undefined

{-@ lookupAppendRight :: xs:List a -> ys:List a -> i:Between 0 (length ys)
                      -> {lookup (i + length xs) (append xs ys) = lookup i ys} @-}
lookupAppendRight :: List a -> List a -> Int -> Proof
lookupAppendRight Nil         ys i = undefined
lookupAppendRight (Cons x xs) ys i = undefined

{-@ takeDropAppend :: xs:List a -> n:Between 0 (length xs)
                   -> {append (take n xs) (drop n xs) = xs} @-}
takeDropAppend :: List a -> Int -> Proof
takeDropAppend (Cons x xs) 0 = undefined
takeDropAppend (Cons x xs) n = undefined
```

Solution

```
{-@ rotateCorrect :: xs:List a -> n:Between 0 (length xs)
                  -> i:Between 0 (length xs)
                  -> {lookup i (rotate xs n) = lookup ((i+n) mod (length xs)) xs} @-}
rotateCorrect :: List a -> Int -> Int -> Proof
rotateCorrect xs n i = let ys = take n xs; zs = drop n xs in
  if i < length xs - n
  then lookupAppendLeft  zs ys i  ? takeDropAppend xs n ? lookupAppendRight ys zs i
  else lookupAppendRight zs ys i' ? takeDropAppend xs n ? lookupAppendLeft  ys zs i'
    where i' = i + n - length xs

{-@ lookupAppendLeft :: xs:List a -> ys:List a -> i:Between 0 (length xs)
                     -> {lookup i (append xs ys) = lookup i xs} @-}
lookupAppendLeft :: List a -> List a -> Int -> Proof
lookupAppendLeft (Cons x xs) ys 0 = trivial
lookupAppendLeft (Cons _ xs) ys i = lookupAppendLeft xs ys (i-1)

{-@ lookupAppendRight :: xs:List a -> ys:List a -> i:Between 0 (length ys)
                      -> {lookup (i + length xs) (append xs ys) = lookup i ys} @-}
lookupAppendRight :: List a -> List a -> Int -> Proof
lookupAppendRight Nil         ys i = trivial
lookupAppendRight (Cons x xs) ys i = lookupAppendRight xs ys i

{-@ takeDropAppend :: xs:List a -> n:Between 0 (length xs)
                   -> {append (take n xs) (drop n xs) = xs} @-}
takeDropAppend :: List a -> Int -> Proof
takeDropAppend (Cons x xs) 0 = trivial
takeDropAppend (Cons x xs) n = takeDropAppend xs (n-1)
```

3 Acknowledgements

Partially funded by the European Union (GA 101039196). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the European Research Council can be held responsible for them.

Funded by the European Union

European Research Council
Established by the European Commission