Verification of Haskell programs using Liquid Haskell

Morten Aske Kolstad

Det matematisk-naturvitenskapelige fakultet

UNIVERSITETET I OSLO

November 15, 2019

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Contributions . . . 4
- 1.3 Chapter overview . . . 4

2 Haskell **5**
- 2.0.1 Curried functions . . . 5
- 2.0.2 Kinds and kind polymorphism . . . 7

3 Liquid Haskell **11**
- 3.1 Liquid Haskell . . . 11
- 3.2 Termination and totality . . . 13
  - 3.2.1 Termination . . . 13

4 Comparison of verification in Haskell vs Liquid Haskell **21**
- 4.1 Natural numbers . . . 21
  - 4.1.1 Natural numbers in Liquid Haskell . . . 22
  - 4.1.2 Natural numbers in Dependent Haskell . . . 24
  - 4.1.3 Comparison . . . 27
- 4.2 Compile time formula validity checking . . . 30
  - 4.2.1 Formula validation in Liquid Haskell . . . 31
  - 4.2.2 Type level formula validity checking in Dependent Haskell . . . 32
  - 4.2.3 Comparison . . . 36

5 Case study : Verifying finger trees **39**
- 5.1 Proving the splitting theorem . . . 47
  - 5.1.1 Proving the splitDigit theorem . . . 47
  - 5.1.2 Proof in Liquid Haskell . . . 48
  - 5.1.3 Split tree proof - by hand . . . 52
  - 5.1.4 Mutual recursion . . . 53

6 Conclusions and Future Work **57**
- 6.1 Conclusions . . . 57
- 6.2 Future work . . . 57
- 6.3 Related work . . . 59

Bibliography **61**

Chapter 1

Introduction

1.1 Motivation

Issues with formal verification

Formal verification gives us the obvious benefits of reducing errors, increasing reliability and an increased trust in the proven system. However, formal verification of programs are not common-place and are mostly reserved for some special use cases. The use of programming languages with the posibility to do formal verification, such as dependently typed languages such as Coq,Agda or Idris, are mostly reserved for academia. There are probably many reasons why this is the case, but one of the most important reason is that formal verification is hard. The techniques used to do formal verification in dependently typed languages use advanced type system and complicated proof tactics. The process of doing formal verifction also takes a lot of effort because of the preciceness needed, and it often leads to code that is a lot more complex than the code without verification. The performance of formally verified code written in a dependent language also suffers.

Liquid Haskell

Liquid Haskell, a verifier for Haskell programs, tries to avoid these issues. With the use of refinement types and an SMT-solver, it tries to automate a lot of the verification. Instead of using advanced proof tactics, it uses equational reasoning. It avoids the performance problem, by being a stand-alone type checker, that does not impact the execution of the program, and therefore the code will be as performant as normal Haskell code.

It uses the most widely used kind of verification today, the type checker, by extending normal Haskell types with predicates. This is done by using extending the Haskell type system with LIQUID types, which are a restricted form of refinement types designed to increase refinement inference.

Liquid Haskell has been used to verify algebraic laws of widely used type class instances[],

One of the biggest Haskell projects, Cardano, a public blockchain protocal and cryptocurrency project, stated that they would add Liquid Haskell to all of their code base [car17].

1.2 Contributions

My contributions are the following:

- A comparison between verification done in Haskell and Liquid Haskell
- A type level sequent calculus for propositional logic in Haskell
- Verified size properties of the multi-purpose data structure finger trees
- Shown the possibility of using Liquid Haskell to prove properties of non-trival datastructures
-

1.3 Chapter overview

Chapter 2

Haskell

The reader of this thesis is expected to have a decent understanding of functional programming and be somewhat familiar with Haskell, or at least be familiar with a somewhat similar language, like SML, OCaml, Idris or Coq.

Therefore basic syntax and functionality, such as type classes, higher order functions, pattern matching, Algebraic Data Types, will not be covered in detail. Knowledge of perhaps the most famous and infamous aspect of Haskell, Monads, is however not needed, because they are not used in this thesis. For a good introduction to Haskell and functional programming, the reader is referred to [Hut16].

However, in this thesis we will look into advanced techniques and functionality, such as type families, generalized algebraic data types and kinds. These techniques are neceassary when doing the type level programming that we will be doing in Chapter 3. These topics will be explained.

In this section we first look into curried functions, before we delve into the more advanced topics of kinds, kind polymorphism, generalized algebraic data types and type families.

2.0.1 Curried functions

One aspect of Haskell that is distinct compared to most other programming languages is that functions are automaticly curried.

A curried function means that instead of taking all the arguments at once, it takes one argument at a time and returns a function that takes the next argument until all the arguments has been provided and then it returns the value.

So if we have a non-curried function that adds numbers

```
add :: (Int,Int) -> Int
add (x,y) = x + y
```

To use this function, we would provide a tuple containing the two numbers and it evalutes to the expected result.

```
> add (1,2)
3
```

Both of the numbers are provided at the same time. This can lead to some inconvinencies when we want to use this function as a higher order function. F.ex. if we want to use our add-function to add 1 to every number in a list by using the `map` function, which takes a function and a list and returns the resulting list of applying the function to every element of the list. Because one of the arguments needs to be 1, we need to create another function that takes in a number and then applies the the add function to 1 and this number. This can be done with an anonymous function like this:

```
> map (\x -> add (1,x)) [1,2,3]
[2,3,4]
```

This is because the lambda adds extra syntactic noise, and makes the code less concise. It also introduces a variable name, that adds extra mental overhead. It can also can lead to less performant code. For example, if instead of 1, we would like to use the result of an expensive computation, lets say the sum of the numbers from 1 to 10000000. In a lambda, this would be evaluated for every time the function is evaluated. This should be optimazed away by turning on the optimazation flags in the compiler, but would lead to very slow code when running interpreted in the repl or without optimazation.

These problems go away if we redefine our add-function to be a curried function. Now it should take one number, and then return a function that takes the second number and then returns the sum of these numbers.

We could do the currying explicitly:

```
add :: Int -> (Int -> Int)
add x = \y -> x + y
```

or take advantage of the syntatic sugaring and do it the most common way, implicit :

```
add :: Int -> Int -> Int
add x y = x + y
```

One thing to notice is that in the second version, the parens around the `Int -> Int` was removed. This is possible, because `->` is a right-associative type operator, so it binds to the right even without the explicit parens.

```
> :t add 5
add 5 :: Int -> Int

> (add 5) 3
8
```

```
> add 5 3
8
```

We see that the the last two examples behaves similar, even though the parens are different. That is because function application is left-associative in Haskell. That results in `add 5 3` being treated as `(add 5) 3`.

2.0.2 Kinds and kind polymorphism

In a typed language like Haskell, every value is of a type. We can classify values by their type. But how can we classify types? What are the "types" of types?

If we have the following data types

```
data T f = MakeT (f Int)

data Identity a = Identity a

data MyInt = MyInt Int
```

`makeT (Identity 1)` would typecheck, but `makeT (MyInt 1)` would not. Why is that? Because the "type" of MyInt does not fit the "type" of f in T.

To understand why it does not fit, we need to understand Kinds.

Kinds are the "types of types". That means that a type has kind, which is analogue to a term having a type. In basic Haskell (Haskell2010), Kinds can be inductivly defined as either `Type` or `k1 -> k2` where `k1` and `k2` are Kinds. The arrow is lifted to the kind-level and is analogue to of the`->` for types. `k1 -> k2` means that : if you apply this kind to a type of kind `k1`, then it will give you a type of kind `k2`.

`*` is the Kind that is inhabited by values. Int is of kind `*` and [Int] is of kind `*`. [] is of kind (`*` -> `*`). [] is not an inhabited type, because it needs to be applied to a Type, and then it will have the kind `*`, which is inhabited.

Higher kind polymorphism leads to the possibility of powerful abstractions such as the Functor,Applicative and Monad type classes. For example, the fmap-function from the Functor class : `fmap :: Functor f => f a -> (a -> b) -> f b` here the f type variable is of kind (`*` -> `*`).

Polymorphic kinds and the Proxy data type

By turning on the extension PolyKinds, Haskell allows us to work with polymorphic kinds. That means that we can quanitfy over kinds, and make data types that accept types of different kinds.

`data Proxy t = Proxy` is probably the most common type that uses this feature. Proxy is a type that holds no data. It only holds a phantom

parameter of arbitary type or kind. It is used when one wants to provide type information, but has no value of that type.

The type of the Proxy data constructor is now `forall t . (t :: k) . Proxy t`. k is a kind variable, and we see now that the phantom parameter in Proxy ranges over all kinds. The kind of Proxy becomes k -> *, because if provided any type or kind, Proxy becomes a value. This makes these definitions possible:

```
p1 :: Proxy Int
p1 = Proxy

p2 :: Proxy Maybe
p2 = Proxy

p3 :: Proxy Proxy
p3 = Proxy
```

Datatype promotion using DataKinds

For the same reasons as we want the type system to constrain the number of valid values , when doing type level programming, we want to constrain the number of valid types. Therefore we need to create custom kinds.

This is done by using the DataKinds extension, which automatically promotes every suitable datatype to be a kind[GHC15a]. The constructors then becomes type constructors.

So in a simple example, we can make a kind to that represent the boolean values at the type level.

```
data B = F | T
```

`B` now becomes a kind. `F` and `T` becomes types of kind `B`.

DataKinds are crucial in order to do type level programming in Haskell. Later in this section we will see how DataKinds togheter with GADTs can encode singleton types.

Generalized Algebraic Data Types - GADTs

Now we will look at Generalized Algebraic Data Types , commonly refered to as `GADTs`[VWPJ06]. GADTs are a generalization of the standard algebraic data types you find in languages like Haskell and ML. They give the added power to specify the type parameter of the different constructors, whereas in normal ADTs they all have to be the same. We will shortly see how to use GADTs to define singleton types

Singleton type

A singleton type is a type with only one inhabitant. That means that if we know the value, we know that type and if we know the type, we know the

value. So a singleton type has a bijective mapping from type to value.

```
data Bool = False | True
```

is not a singleton type, because if we have a value of type `Bool`, we can't decide whetever the value is False or True, based on the type.

But if we use GADTs to add a type parameter and specify the type

```
--B is to be used as a Kind
data B = F | T

data GBool b where
    False :: GBool F
    True :: GBool T
```

But the GBool data type that was defined using GADTs would have been a singleton type. Because we know that if the value is `False`, then the type is `Bool F`, and if the type is `Bool F`, the value must be `False`. The same holds for `True` and `Bool T`.

We will now see how to define a singleton type for natural numbers.

Singleton type without GADTs? If we try to make a singleton data type for natural numbers using normal ADTs, a first effort would maybe look something like this:

```
data Natural n = Zero | Succ (Natural n)
```

The types given to the data constructors now will be : `Zero :: forall n . Natural n` and `Succ :: forall n . Natural n -> Natural n`. This makes non-sensical types (in this context) such as `Zero :: Natural String` and `Succ Zero :: Natural [Bool]` accepted. To avoid this problem, we try to constrain the kind of the type parameter to be of kind `Nat`.

```
data Natural (n :: Nat) = Zero | Succ (Natural n)
```

With the kind annotation, we specify that the type parameter needs to be of kind Nat, so that it represents a natural number at the type level. The types given now will be : `Zero :: forall (n :: Nat) . Natural n` and `Succ :: (n :: Nat) . Natural n -> Natural n`. Now types such as `Zero :: Natural String` and `Succ Zero :: Natural [Bool]` won't be accepted, because the type parameters in those examples are not of kind `Nat`.

But we have another problem The problem is that we can't specify what the type in the type parameter should be for the different constructors Zero and Succ.

That means that we can have `Zero :: Natural (S Z)` or `Zero :: Natural (S (S Z))` or `Succ Zero :: Natural Z`

This means that we don't have a singleton type. We don't have a one-to-one mapping from the type to the value, because as we can see, the value `Zero` is of both type `Natural (S Z)` and `Natural (S (S Z))`

Singleton type using GADTs

What we need is some way to specify that Zero is a constructor with the type Natural Z and Succ is a constructor that takes a natural number with type parameter n, and then returns a natural number with the type parameter (S n), to represent the additional layer of Succ.

```
data Natural (n :: Nat) where
    Zero :: Natural Z
    Succ :: Natural n -> Natural (S n)
```

Now the data constructors have the appropriate types, namely `Zero :: Natural Z` and `Succ :: forall (n :: Nat) . Natural n -> Natural (S n)`.

Now we have an actual singleton type. If the value is `Zero`, we know that the type is `Natural Z`, and if the type is `Natural Z`, we know that the value is `Zero`. If the value is `Succ (nat :: Natural n)`, we know that the type is `Natural (S n)`, and if the type is `Natural (S n)`, we know that the the value is `Succ (nat :: n)`, by using the inductive hypothesis.

Type families

To enable type level programming in Haskell, we have so far introduced a way to make custom kinds, which enables the creation of data types at the type level. But to do interesting stuff at the type level, we also need functions that work on types.

Type families are type level functions. They are enabled by the TypeFamilies extension[GHC15c]

```
type familiy Not b where
    Not T = F
    Not F = T
```

In chapter 3, we will look at more complex type families, such as a type familiy representing the addition of two natural numbers, and one type family that represents two inference rules in a logical calculus.

Overview

We have now introduced singleton types, kinds, DataKinds and GADTs, and combined all of theszse to

Chapter 3

Liquid Haskell

3.1 Liquid Haskell

Liquid Haskell (LH) is a refinement type checker that uses LiquidTypes. In this section we will look at an overview of the features in Liquid Haskell. For a detailed look at LiquidHaskell, the reader is referred to [VSJ14a].

Staticly typed languages , like Haskell, can prevent a lot of runtime errors at compile time, by type checking. That is, to verify the type safety of a program. That a program is type safe means that is does not contain type errors . (TAPL!) With this, you can catch errors like if you have a function that expects a string, but you give it an integer as an argument instead.

But what about if you have a function that expects a number between 0 and 300. Or a function that expects a list with 2 or more elements. How would one use a normal type system to verify that these properties are being held.

This is where refinement types come into play.

Refinement types

Refinement types makes it possible to encode invariants by combing types and logical predicates [VSJ+14b]. These predicates needs to be SMT-decidable, which means that they can only include formulas from decidable logics. This is to help automation of the type checking.

SMT : Satisfiablity modulo theories

Satisfiablity modulo theories (SMT) is a generalization of boolean satisfiability. It does that by adding additional first-order theories, such as equality reasoning, arithmetic, arrays and more [RR08].

Logically Qualified Data Types - Liquid types

Logically Qualified Data Types, abbreviated to Liquid types, was introduced in [RKJ08]. Liquid types are a restricted form of dependent types,

namely refinement types. Which means that the predicates used to refine the types must be created from a decidable sublanguage, and cannot be arbitrary expressions, as in dependent types. This constraint makes it possible to use SMT solvers to implement a decidable type checking. Liquid Types were created specificly to enable automatic inferring of these refinements, to reduce the number of type annotations the user has to specify and therefore hoping to increase the adoption of refinement types [RKJ08].

A liquid type has the form `{V : T | e}`, where `T` is a Hindley-Milner type, which means a standard algebraic data type, and `e` is a boolean expression which may contain the `V` variable and free variables[PM16].

A detailed investigation into the inner details of Liquid types is beyond the scope of this thesis. For a thorough and detailed overview of the inner workings of Liquid types, the interested reader is referred to [RKJ08].

Liquid Haskell - misc

All specifications appear within comments of the form `{-@ ... @-}`. These comments are ignored by Haskell compilers.

One example of a type in Liquid Haskell is the following type, that represents a refined Int type, where the numbers range from 0 to 300.

```
{-@ type Num0To300 = {v:Int | 0 <= v && v >= 300}} @-}
```

Lifting functions into the logic

To make Liquid Haskell able to reason about your functions, they needed to be encoded in the logic. This is done by lifting the functions into the logic. Depending on the complexity of your function, you can either use a measure or reflect the function into the logic.

measure

To describe properties of algebraic data types, LH has `measures`. Measures are inductively defined functions on algebraic data types. They are restriced in the form that they can only have a single equation per constructor and the right-hand side of the equation must be a term in the restrucited refinement logic [VSJ14a]. Each equation in a measure creates a refined type for the corresponding data constructor [VSJ+14b]. Due to the restriction of measures, they are able to by automatically unfolded into the SMT logic.

reflect

For functions that does not fit the restricted subset of measure functions, it is possible to lift arbitrary functions into the logic, using the `reflect` annotation. These functions are not automatically unfolded into the logic. This makes reflected functions more cumbersome to use.

Proofs in Liquid Haskell

When doing non-trivial proofs in Liquid Haskell, we need to use proof combinators. These enables us to do equational reasoning.

The ones used in this thesis are : `===` : the key combinator in equational reasoning. It ensures that the left and right hand side is equal.

`?` : adds a proof fact from the right side to the left side. Can be seen as the combinator that lets us use other proofs or lemmas.

`***` and `QED` : creates a proof from any value.

Examples of equational proofs in Liquid Haskell will be seen in chapter 3, when proving properties of natural numbers, and in chapter 4, when proving properties of finger trees.

3.2 Termination and totality

The standard definition of a total function is a function that is defined for all possible input values. That means that it terminates and returns a value for all arguments. In an LH context, when referring to totality and total functions, the part about being defined for all input values is seperated out. So in an LH context totality checking refers to checking that the functions have a case or guard for every possible value of the input type. This is also a convention I will use in this thesis.

Totality and termination are important aspects in programming. We want to reason about the totality of our functions to ensure that they are defined for all the cases and we want to be sure that our functions terminates and not ends up in infite loops. In verified programs and theorem proving, there is also the extra dimension, in that they are required to make the verification sound.
The following section will go into the following aspects of termination and totality

- different techniques used in LH to prove termination
- how to use non-terminating functions in LH
- how non-termination can lead to falsehoods
- The difference between the GHC exhaustiveness checker and LH totality checker
- how partial function can lead to falsehoods

3.2.1 Termination

To ensure the soundness of the LH refinements, LH requires proof of termination for functions.

To say that a function terminates means that for all arguments, the function does not lead to infinite computation. That means a function that terminates will never loop for ever, but reach a base case, for every argument. Liquid Haskell uses a well founded metric, such as the lexiographic order of natural numbers or the structural size of ADTs, to verify that functions are terminating.

Structural termination

The structural termination checker is an automatic checker that can automaticly prove termination by detecting the common pattern where the argument to the recursive call is a subterm of the original function argument [VBK+18]. Most of the functions in this thesis are accepted by the structural termination checker.

To compute the length of a linked-list is common in functional programming.

```
length [] = 0
length (x:xs) = 1 + length xs
```

The argument to the recursive call `xs` is a subterm of the original function argument `x:xs` This is recognized by the structural termination checker, and therefore it accepts the definition as terminating.

More complex termination and semantic termination

But the structual termination checker is not always enough. There are a lot of functions that terminates, but where the recursion is more complex than just using subterms as arguments. Even a simple function such as

```
range n m = if n <= m then n : f (n+1) m else []
```

can not be proven to terminate using a structural checker. The first argument in the recursive call of `range` is not a subterm of the original first argument. So the structural size of the arguments does not decrease, but is there anything else that decreases? Yes, the difference between the two arguments. Since the recursion only happens when `n <= m`, that means that when `n` increases, `n-m` decreases. `n-m` is the metric that shows that this function terminates.

We have now moved from structural recursion to a more advanced termination checking that the LH authors calls `Semantic termination`[VBK+18]. This is done by providing an explicit termination argument, which is an expression that decreases in each call and calculated from the function argument. The termination argument is on the form `[e_1, e_2, ...., e_n]` where the expressions `e_i` most often depends on the function arguments (we will see one example later where we use a constant expression to prove the termination of mutual recursive functions). The expressions must evaluate to natural numbers and they must lexicographically decrease at each recursive function call.

Non-termination can lead to falsehoods

To demonstrate why proof of termination is needed to unsure soundness, we will look at an example that manages to get LH to accept the falsehood `True == False`, by disabling termination checking.

```
{-@ LIQUID "--no-termination" @-}

import Proof ((?))

{-@ f :: a -> {v:a | True == False} @-}
f a = a ? f a
```

The function `f` has a type that says : if we give it any value, it will give us back a value of the same type that also carries a proof that `True == False`. Because of the "no-termination" flag, which disables termination checking for the whole file, LH will not complain about the argument to recursive call not decreasing in size. Thus it is deemed safe by LH.
What happens is that the refinement `True == False` holds because in the recursive call to `f`, the refinement is assumed in the induction hypothesis. This is sound if the recursive call is terminating, but in this case it is not. The refined type is added as a proof fact to the return type by the proof the infix function `?`, which "adds" the proof fact from the right operand to the left. Therefore the function type checks and is deemed safe.

Lazyness

As Haskell is a language with non-strict semantics, data structures such as infinite lists and non-terminating functions are commonplace. If we try to use non-terminating functions in LH, the termination checker will deem the code unsafe. But there is safe code that can come from the use of non-terminating functions. Such as taking the first 5 elements of an infinite list of zeros :

```
take 5 (repeat 0))
```

where `take` is a terminating function but `repeat` is a non-terminating function with the following implementation :

```
repeat n = n : repeat n
```

LH would deem this code unsafe, but in reality it is safe. So we would like force LH to accept that this, by removing the termintation checking for `repeat`, but still regain termination checking for the remaining function definitions in the file. The above-mentioned "no-termination"-flag, disables termination checking for the whole file. To disable termination checking for a specific function, instead of the whole file, LH provides a "lazy"-flag.

An example of the lazy flag in use:

```
{-@ lazy repeatValue @-}
```

```
{-@ repeatValue :: a -> {v:[a] | len v >= 5} @-}
repeatValue a = a : repeatValue a
```

`repeatValue` is a non-terminating function that takes an argument and "repeats" it infinitely many times in a lazy list, exactly like the previous `repeat`. The difference is in the the refinement on the return type which in this case says that the length of the list is greater or equal to 5. This is a valid refinement, because the list is infinite in length. This refinement is checked correctly and LH marks it as safe. If we had tried to say that the length is equal to 5, LH would correctly deem it unsafe.

The user has to be careful though, LH is not sound with the use of the lazy flag and it can easily lead to falsehoods and unsafe code. We can look at an example of this:

```
{-@ lazy repeatValueBad @-}
{-@ repeatValueBad :: a -> {v:[a] | True == False} @-}
repeatValueBad a = a : repeatValueBad a
```

Because of the same reasons as for the `f`-function in the previous-example, repeatValueBad here carries a proof of the falsehood `True == False`. The user needs to be careful about the refinements that is added to the return value of a lazy function.

```
{-@ lazy repeatNat @-}
{-@ repeatNat :: n:Nat -> {v:[Nat] | n == 2} @-}
repeatNat a = a : repeatNat a

{-@ p :: n:Nat -> {v:Nat | v == n && n == 2} @-}
p n = n ? repeatNat n

{-@ f :: {v:Nat | v == 2} -> () @-}
f :: Int -> ()
f 2 = ()

runFwith3 = f (p 3)
```

We can use this unsoundness to make LH accept unsafe code that crashes at runtime. The lazy `repeatNat` is the culprit. `p` takes a `Nat` and returns that same `Nat`, but with the added proof that this `Nat` is equal to `2`, regardless of the actual value of the `Nat`. The proof is obtained by using the refinement from `repeatNat` as an additional proof fact, which says that any argument given to `repeatNat` is equal to `2`.

The domain of `f` is refined to `{2}`. In `runFwith3`, we are able to "trick" LH to accept the call `f` with the result of `p 3`, which is 3. But this value also carries a proof that it is equal to 2 and if this is executed, this will lead to a non-exhaustive pattern error at runtime.

Totality

When using the refined types in LH, the values that a function needs to be defined for is narrowed down to the values in the refined type.

The following function

```
f :: Int -> Int
f x | x == 0 = 0
    | x >  0 = 1
```

would not be total, since it is not defined when x is a negative number. But if we refine the type of the input argument to be a natural number `{v:Int | v >= 0}` the function is now total because now the input number can't be negative, so the two guards covers all the input values.

```
{-@ f :: {v:Int | v >= 0} -> Int} @-}
f x | x == 0 =
    | x >  0 = 1
```

Liquid Haskell checks for totality by default. Total functions are needed in LH in order to preserve soundness [VBK+18]. GHC can already check for totality to some degree, but because this check is not so fine tuned and the types are not as fine grained as refinement types, so this check leads to over-approximation.

For an example that shows the difference between the GHC exhaustiveness checker and the LH totality checker, we look at these 3 implementations of a function that gives the absolute value of an integer:

```
abs :: Int -> Int
abs i | i >= 0 = i
      | i < 0 = negate i

abs' :: Int -> Int
abs' i | i >= 0 = i
       | not (i >= 0) = negate i

abs'' :: Int -> Int
abs'' i | i >= 0 = i
        | i < 0 = negate i
        | otherwise = error "should and won't ever happen"
```

GHC will complain about both abs and abs' not being total, because it thinks pattern matches are non-exhaustive. The error message is

```
Pattern match(es) are non-exhaustive
In an equation for "abs": Patterns not matched: _
| abs i | i >= 0 = i

Pattern match(es) are non-exhaustive
In an equation for abs': Patterns not matched: _
```

```
| abs' i | i >= 0 = i
```

It complains that the wildcard (_) case has not been matched. That is because it can't reason that an `Int` is either greater or equal to 0, or lesser than 0. It therefore think it needs an default case, an otherwise-guard, in case none of the two cases are true.

The same thing happens in the `abs'` function. Haskell does not have the possibilty to reason that a boolean value and the negation of the same boolean value covers every case.

When we add the otherwise-case (which is just constant defined `otherwise = True`) in `abs`", GHC accepts the definition as exhaustive.

LH will accept all the 3 functions when checked for totality. In the `abs` and `abs'` function, the SMT-solver is able to use the basic facts about integers and booleans, to conclude that the two guards are exhaustive for both functions.

The refined error function

Liquid Haskell has refined the type type of the standard `error` function to be

```
{s:String | False} -> a
```

which means that LH will only accept a call to `error` if the context it is called from leads to a contradiction and therefore is unreachable. In this case, LH sees that in order to call `error` it means that

```
    not (i >= 0) && not (i < 0)
```

which the SMT-solver figures out leads to False, and then accepts the call to error as safe.

Partial functions and unsoundness

Checking for totality is not only important to avoid errors, but also to keep soundness.

We can see how partial functions, which means functions that are not total, can lead to unsoundness. By turning on the "no-totality"-flag, LH will disable totality checking.

```
{-@ LIQUID "--no-totality" @-}

import Proof

{-@ f :: b:Bool -> {v:Bool | v == False && b == v} @-}
f False = False

{-@ g :: b:Bool -> {v:Bool | v == True && b == v} @-}
g True = True
```

```
{-@ unsound :: Bool -> { True == False } @-}
unsound True = () ? f True
unsound False = () ? g False
```

Here there are two partial functions, `f` and `g`. The `unsound`-function, even though being a total function itself, ends up with a contradiction in the refined return type. The refinements in the return types of `f` and `g` only holds because they are partial and by using them both for the different cases of the boolean input in `unsound`, we can create the contradiction that `True == False`. In the case of `f True` what happens is that `f` returns a Boolean value with proof that `v == False && b == v`. Since we also have `v == False && b == True` this leads to `False == False && True == False` which makes the `True == False` in the refinement hold. What happens in the False-case is similar.

Chapter 4

Comparison of verification in Haskell vs Liquid Haskell

One of the benefits of using Liquid Haskell is the ability to do theorem proving and verification. But one can also do theorem proving and verification in Haskell without Liquid Haskell. To do theorem proving and verification in Haskell, we need to emulate dependent typing. This part of Haskell is sometimes refered to as *Dependent Haskell*[Si14], which is also the term we will be using in this thesis and also the abbreviation DH.

This brings up some questions:

- What are the benefits to using LH in regard to theorem proving and verification compared to DH?
- Are dere any downsides?
- How do they compare?

To try to answer these questions, we will look two small case studies, one that focusing on proving that commutativity of addition of natural numbers, and one that focuses on being able to verify validity of propositional formulas at compile time.

4.1 Natural numbers

Natural numbers are a common sight in programming languages with dependent types. They are often used at the type level. For instance, to define lists where the length is encoded in the type, instead of something resembling `List a`, you would have `List a n` where `n` would be a type level representation of a natural number.

To prove and verify properties of functions that uses natural numbers or uses types that uses natural numbers, the properties of natural numbers themselves becomes important. For example, with pseudo-code in a Haskell-like language, we would like to prove that for every pair of

two lists, `xs :: List a n` and `ys :: List a m`, we want to show that `length (xs ++ ys) = length (ys ++ xs)` where the appending operator has the type `(++) :: List a n -> List a m -> List a (n+m)` and the length function has the type `length :: List a n -> n`. We see that the proof follows from the fact that `(n+m) = (m+n)`, which is exactly the commutative property of the natural numbers.

This is just one of many examples where the properties of the natural numbers are important in theorem proving and verification using dependent types. Therefore, it would be interesting to see how to we can use LH and Haskell to prove one of these properties.

So in the following section we will look at an inductive definition of the natural numbers, addition of two natural numbers, and finally prove that addition is a commutative operation.

4.1.1 Natural numbers in Liquid Haskell

We start by defining a data type to represent the natural numbers.

```
data Nat = Zero | Succ Nat
```

So a value of type `Nat` needs to be created by the data constructor `Zero` or the data constructor `Succ` given an argument of type `Nat`.

To define addition of two natural numbers, we do induction on the first number.

```
add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n') m = Succ (add n' m)
```

Liquid Haskell will accept the function definition as terminating without any annotations or explicit termination metric. As explained in the section about termination in LH, the structural termination checker will automaticly see that the first argument becomes structurally smaller in size in the recursive call.

What we now want to do is to show that `add` is commutative, meaning that for all natural numbers `n` and `n`, `add n m` should give the same result as `add m n`.
A proof of this in LH would be to implement a function (that type checks) with the following type:

```
addComm :: a:Nat -> b:Nat -> {add a b == add b a }
```

The type of `addComm` says : if we give it a value `a` of type `Nat` and a value `b` of type `Nat`, it will give us a value that carries a proof of `add a b = add b a`.

Before we can implement that function, we need two lemmas.

Lemma : `add n Zero = n`

One of the lemmas is to prove that Zero is the right identity for `add`, meaning that for every natural number `n` , we have that `add n Zero = n`.

To prove the lemma we implement the following function

```
addIdRight :: a:Nat -> { add a Zero == a }
```

The base case, when the argument is `Zero`, follows from applying the definition of `add`. The inductive case needs to use the induction hypothesis, which in this case is : for every `s`, where `s` is value of type `Nat` , `add s Zero == s`. To make use of the induction hypothesis, we simply do a recursive call. After we have used the induction hypothesis, we are done.

Because we removed one layer of Succ in the argument of the recursive call, `s` becomes structurally smaller, so this recursive call will not lead to non-termination, and LH will deem this safe automaticly.

```
{-@ addIdRight :: a:Nat -> { add a Zero == a } @-}
addIdRight Zero
    =   add Zero Zero
    === Zero
    *** QED
addIdRight (Succ s)
    =   add (Succ s) Zero
    === Succ (add s Zero) ? addIdRight s
    === Succ s
    *** QED
```

The inductive case, is the case with `Succ a` and `b`, and then the inductive hypothesis will be that `add a b = add b a`.

Lemma : `Succ (add b a) = add b (Succ a)`

We see that we also need a lemma to say that `Succ (add b a)` is equal to `add b (Succ a)`. This is done by the following proof, which follow the same structure as the proof for the previous lemma :

```
{-@ addSucc :: a:Peano -> b:Peano ->
    {Succ (add a b) == add a (Succ b)} @-}
addSucc Zero b
    =   Succ (add Zero b)
    === Succ b
    === add Zero (Succ b)
    *** QED
addSucc (Succ a) b
    =   Succ (add (Succ a) b)
    === Succ (Succ (add a b)) ? addSucc a b
    === Succ (add a (Succ b))
    === add (Succ a) (Succ b)
    *** QED
```

Theorem : `add a b = add b a`

Now we can start with the actual commutativity proof. To prove that `add` is commutative, we do induction on the first number. The base case then becomes straightforward with the use of the `addIdRight`-lemma:

```
addComm Zero b
    =   add Zero b
    === b ? addIdRight b
    === add b Zero
    *** QED
```

The addComm-inductive case then becomes

```
{-@ addComm ::
        a:Nat
    ->  b:Nat
    ->  {add a b == add b a } @-}
addComm (Succ a) b
    =   add (Succ a) b
    === Succ (add a b) ? addComm a b
    === Succ (add b a) ? addSucc b a
    === add b (Succ a)
    *** QED
```

PLE : Proof by Logical Evaluation

Liquid Haskell can use Proof by Logical Evaluation(PLE)[VBK+18] , which is a powerful tool to automate the trivial parts of proofs. This is done by automatic evaluation and unfolding of reflected function calls. This leads to the possibility of eliminating the tedious work of writing very rigerous proofs and lead to more concise proofs. The downside is that the proofs can be harder to understand because of the hiding of function expansions [VBK+18]. So if we enable PLE, we now only need to use the different lemmas, and you can skip all the applying and un-applying of the function definitions, so the implementation of the commutative proof now becomes

```
addComm Zero b = addIdRight b
addComm (Succ a) b = addComm a b &&& addSucc b a
```

4.1.2 Natural numbers in Dependent Haskell

We can do the same proof in Haskell without using Liquid Haskell. To do this, we need to use the techniques introduced in the Haskell background section.

Definition of natural numbers in DH

First we need a way to make it so that the values of type Natural also carries some proof of which value it has. So that if we have the value, we know

the type and vice versa. This is called a singleton-type [singleton-sorce, Eisenberg], because this leads to that each type only has one inhabitant.

To do that in Haskell, we start by defining a kind, so that we have a representation of the natural numbers at the type level.

```
data Nat = Z | S Nat
```

This looks like a normal sum type, but with the DataKinds extension turned on, this is lifted to the type level, and will create a new Kind. This Kind will be inhabited? by the types Z and S n where n also is a Nat.

To define the actual values to represent the natural numbers we reach for the power of GADTs.

GADTs is an abbreviation for Generalized Abstract Data Type. They give the added power to specify the type parameter of the different constructors, whereas in normal ADTs they all have to be the same.

```
data Natural (n :: Nat) where
    Zero :: Natural Z
    Succ :: Natural n -> Natural (S n)
```

We then define addition for the `Nat` kind, by defining a type family.

For our type family to represent addition, we make an infix type family `+` , that takes in two types of kind `Nat`, and returns a type of kind `Nat`. (The Kinds are here annotated for clarity, but would be inferred if not annotated)

```
type family ((a::Nat) + (b::Nat)) :: Nat where
    Z + n = n
    (S n) + m = S (n + m)
```

As we can see, the computation and recursion are similar to the definition of add used in our Liquid Haskell version

Type equality in Haskell

For proof of equality of types in Haskell, we will use

```
data a :~: b where
    Refl :: a :~: a
```

which is defined in `base` (the Haskell standard library) which says that you can create a proof that the a `a` is equal to a type `b` using the Refl constructor which gives you a proof that some type is equal to itself. Refl is short for reflexive, and represents the reflexivity axiom. In this context the reflexivity axiom represents that every type is equal to itself.

To prove that + is commutative, we need the same lemmas as we used in the Liquid Haskell version. but changed to use our new data type.

To prove that `Z` is the right identity for +, we implement a function with the type `Natural n -> n :~: (n + Z)`.

In the base case, `Zero`, n will be `Z`. GHC will evaluate the type family, and `Z+Z` will evaluate to `Z`. To make the function type-check we then need to return a value with the type `Z :~: Z`. Since Refl has the type `forall a . a :~: a` , we can get a valid proof by returning the Refl-constructor and the Haskell type checker will apply the correct type to the constructor.

```
rightId Zero = Refl
```

The inductive case will be of the `Succ s`. Here we type parameter will be of the form `S n`, and the type of the proof needs to be `S n :~: S (n + Z)` By the induction hypothesis, we get a proof that `n :~: n + Z`. So we only need to apply `S` to both sides of this equality. To do this we implement the congruence axiom.

```
cong :: (a :~: b) -> f a :~: f b
cong Refl = Refl
```

Then we use `cong` on the recursive call and Haskell infers that that `f` must be`S`, and the proof type checks.

```
rightId (Succ s) = cong (rightId s)
```

The proof for the lemma of the swapping of the `S` is similar.

```
sSwap :: Natural n -> Natural m ->  (S n + m) :~: (n + S m)
sSwap Zero m = Refl
sSwap (Succ n) m = cong (sSwap n m)
```

The proof of commutativity becomes very similar to the one in Liquid Haskell with PLE enabled.

```
plusComm :: Natural n -> Natural m -> (n+m) :~: (m+n)
plusComm Zero m
    =   rightId m
plusComm (Succ n) m
    =   trans (cong (plusComm n m)) (sSwap m n)
```

The inductive case uses
`trans :: (a :~: b)-> (b :~: c)-> a :~: c`, which corresponds to the transitivity of equality. That case needs to return a proof of the type equality `S (n1 + m):~: (m + S n1)`.

cong (plusComm n m) gives a value of type `S (n1 + m):~: S (m + n1)` and `sSwap m n` gives a value of type `S m + n1):~: (m + S n1)` . The use of `trans` then returns a proof of `S (n1 + m):~: (m + S n1)`, which is what we needed, and the proof is done.

cong - kind polymorphic

It is interesting to take a close look at `cong`.

The full type of `cong` actually needs to use kind polymorphism, which is enabled through the PolyKind extension. The basics of kind polymorphism was explained in the Haskell background section earlier in this thesis.

```
cong ::
    forall k1 k2 (a :: k2) (b :: k2) (f :: k2 -> k1).
    (a :~: b) -> f a :~: f b
```

We see here that the the `a` and `b` type parameters can be of any kind (as long as it is the same). Without kind polymorphism, the only legal kind for the type parameters would have been `*`, as seen here :

```
cong ::
    (a :: *) (b :: *) (f :: * -> *).
    (a :~: b) -> f a :~: f b
```

This version of `cong` is useless for our needs. And this shows the importance how kind polymorphism when doing type level programming in Haskell.

4.1.3 Comparison

Expressiveness/Readability?

The main difference between these implementations are the data type definition. In LH we used a straight forward inductive definition, because LH enables us to reflect this definition and work with a normal data type at the type level. In Haskell we had to represent the data type at both the type and term level, by defining a new kind and then defining the data type using GADTs. So at this point it is fair to say that LH has a clear benefit compared to Haskell.

Regarding the actual proof and the lemmas, we see that the implementations in LH and Haskell are pretty similar. Both use structural induction, the same lemmas and follow the same structure. In LH there are actually 2 versions, one without PLE and one with PLE. The one without PLE is naturally more verbose because it needs to show all the unrolling.

Which one that is more readable depends on the purpose of the reader. If the reader wants to study the proof carefully, the non-PLE version provides a more explicit way to read the proof, and therefore also an easier way to understand every part. But if the reader instead wants to have a more high-level understanding of the proof or just wants to skim the proof, all the function unrolling and verboseness can clutter up the proof and can make it hard for the reader to know which parts of the proof which are the most important.

Reliability? of the proofs

An important point to mention, that since Haskell is a non-total language, the proofs in Haskell does not check for termination. So even though the compiler accepts the proof, it doesn't mean that it is a valid proof, because we could have easily made an incorrect non-terminating proof ,like `plusComm = plusComm`. Haskell does not also check for totality(exhaustiveness)

by default, but this can be easily turned on by a compiler flag and is commonly always enabled. So we have to check it manually or assume that it is correct. One of the most important reasons to do interactive theorem proving is the ability to be checked automatically, so this a big drawback for theorem proving in Haskell.

Ease of use

Forgetting one case To do a simple comparison of the error message displayed when the proofs are missing a case, we can simpy remove the `Succ`-case in the final commutativity proof in both LH and Haskell.

The error message from LH :

```
Error: Liquid Type Mismatch

27 | addComm Zero b
28 |     =   addIdLeft b

    Inferred type
    VV : {v : GHC.Prim.Addr# | v == "PeanoLH.hs:(27,1)-(28,19)|
        function addComm"}

    not a subtype of Required type
    VV : {VV : GHC.Prim.Addr# | 5 < 4}
```

and the one from Haskell:

```
    Pattern match(es) are non-exhaustive
    In an equation for  plusComm': Patterns not matched: (Succ _
        ) _
   |
34 | plusComm Zero m
   | ^^^^^^^^^^^^^^^...
```

To me, the Haskell one is clear and obvious. It says that we are missing a pattern match where the first argument is on the form `Succ _`, which is exactly what we removed to create the error. The LH one is confusing and unclear about why there is an error. It mentions that the function is not a subtype of an `Prim.Addr# where 5 < 4`, which isn't an intuitive way to say that the function is partial and probably only makes sense to the creators of LH.

Debugging Another important aspect to help the user be able to complete the proofs, is the error messages when the proofs are wrong and incomplete. We can look at what the error messages are if we remove the use of the addSucc lemma, in the inductive case of the plusComm implementation, and instead uses the addIdRight-lemma. This makes our proof wrong and therefore we can look at the error messages that hopefully we explain why they are wrong

LH:

```
Error: Liquid Type Mismatch

30 |     =   addComm a b &&& addIdRight a
                ^^^^^^^^^^^^^^^^^^^^^^^^^

    Inferred type
    VV : ()

    not a subtype of Required type
    VV : {VV : () | Main.add ?a b == Main.add b ?a}

    In Context
    b : Main.Peano

    ?a : Main.Peano
```

This just says that the type is wrong. It says that the inferred type does not contain any proof facts, which is weird, because it should contain `add a b == add b a &&& add a Zero == a`

Haskell:

```
* Could not deduce: m ~ 'S ('S ('S t0))
from the context: n ~ 'S n1
    bound by a pattern with constructor:
                Succ :: forall (n :: Nat). Peano n -> Peano ('S
                    n),
             in an equation for  plusComm'
    at C:\\Users\MortenAske\master-temp\code\PeanoEq.hs:36:11-16
 m' is a rigid type variable bound by
    the type signature for:
    plusComm :: forall (n :: Nat) (m :: Nat).
                Peano n -> Peano m -> (n + m) :~: (m + n)
    at C:\\Users\MortenAske\master-temp\code\PeanoEq.hs:33:1-49
Expected type: 'S ('S ('S t0)) :~: 'S ('S ('S t0 + 'S n1))
    Actual type: m :~: (m + 'Z)

37 |     =   cong (plusComm n m)  trans  rightId m
```

Here we get some more indication about what is wrong, but it is hard to figure out exactly what went wrong.

Typed holes But by using a *typed hole*[Gis18], which is a placeholder that the compiler tries to figure out, we can get a much clearer understanding of what is going on and what we need. By placing a hole instead of `rightId m`, we get the following information:

```
* Found hole: _ :: 'S (m + n1) :~: (m + 'S n1)
```

```
Where:  n1' is a rigid type variable bound by
    a pattern with constructor:
    Succ :: forall (n :: Nat). Peano n -> Peano ('S n),
    in an equation for  plusComm'
    at C:\Users\MortenAske\master-temp\code\PeanoEq.hs:37:11-16
 m' is a rigid type variable bound by
    the type signature for:
    plusComm :: forall (n :: Nat) (m :: Nat).
                Peano n -> Peano m -> (n + m) :~: (m + n)
    at C:\Users\MortenAske\master-temp\code\PeanoEq.hs:34:1-49
```

where the first list is the most informative. It says that to make our proof type check, we need a value of type `'S (m + n1) : :  (m + 'S n1)`. In contrast to the LH error message and the previous Haskell one, this is actually helpful. We can now see easily that we need to use the addSucc-lemma.

Typed holes are also capable of suggesting values that "fits" in the holes, so-called *valid hole fits* . If we would instead of using a hole for the whole value, we could put the hole with m and n as arguments. The mesage from the compiler then becomes:

```
    * Found hole:
    _ :: Peano m -> Peano n1 -> 'S (m + n1) :~: (m + 'S n1)
  ....
  Valid hole fits include
    addSucc :: forall (n :: Nat) (m :: Nat).
               Peano n -> Peano m -> ('S n + m) :~: (n + 'S m)
      with addSucc @m @n1
      (bound at C:\Users\MortenAske\master-temp\code\PeanoEq.hs
          :31:1)
|
38 |     =    cong (plusComm n m)  trans  _ m n
```

The compiler suggest addSucc as a valid hole fit, which is exactly what we were looking for.

As we can see from these examples, which was also my experience using these tools, it is easier to work with Haskell instead of LH in regards to debugging because of the more helpful error messages and especially the typed holes. Doing proofs in LH is a neat excperience when it works fine, but as soon at it isn't correct, there is often a whole lot of guessing involved. So in this aspect, normal Haskell has a clear advantage.

4.2 Compile time formula validity checking

ML was introduced to be able to assist theorem proving. As Haskell is heavily influenced by the ideas of ML, this spirit of logic and theorem proving has been a important part of the use of Haskell, especially in

academia. Therefore it would be an interesting case study to compare LH and Haskell on a problem in this domain.

The problem I have chosen is compile time validation of formulas from propositional logic.

In the LH version we will see how we can take advantage of the SMT-solver to do the check for validity. In the Haskell version we will use a singleton representation of the propositional formulas and then introduce a type level sequent calculus that can be used to check validity.

4.2.1 Formula validation in Liquid Haskell

To start, we define a data type to represent formulas in propositional logic. This is just a straight forward inductive definition that follows from the actual definition in logic. The logical operators will give rise to one binary data constructor each. The one difference is that instead of atomic symbols, variables will be represented by Int's instead.

```
data Formula =
        Var  Int
    |   Not  Formula
    |   And  Formula Formula
    |   Or   Formula Formula
    |   Impl Formula Formula
```

So our task now is to make it so that we can have a compile time checked type for values of type Formula which evaluates to true for every variable assignment.

We can take advantage of the fact the the Liquid Haskell uses a SMT-solver to solve and type check programs. SMT-solvers have built in optimized SAT-solvers, which can be used to check the validity of a formula.

We take advantage of that by defining a measure for the Formula data type.

```
{-@ measure isValid :: Formula -> Bool
isValid (Var i) = truthValue i
isValid (Not s) = not (isValid s)
isValid (And l r) = isValid l && isValid r
isValid (Or l r) = isValid l || isValid r
isValid (Impl l r) = isValid l => isValid r
@-}

{-@ measure truthValue :: Int -> Bool @-}
```

This corresponds nicely to the standard interpretation of propositional logic. It is a simple recursive function that replaces the binary data constructors that represents the different operators with the actual logical operators. The one part that is not straight forward is that the truth assignments are represented by the uninterpreted function `truthValue`.

To understand how the measure will work, we can look at how it would measure `Impl (Var 1) (Var 1)`

```
Impl (Var 1) (Var 1)
    =
isValid (Var 1) => isValid (Var 1)
    =
truthValue 1 => truthValue 1
```

The SMT-solver can reason that `truthValue 1` either evaluates to `True` or `False` at both sides of the implication arrow, and therefore evaluate to `True` in either case. Therefore the measure will return `True`.

Then we can just create a type `Theorem` , that says that a value of type `Formula` is a `Theorem`, if the measure `isValid` evaluates to `True`. `-@ type Theorem = v : Formula | isValid v @-`

And then we can use this type to refine a Formula to be a Theorem.

```
{-@ theo1 :: Theorem @-}
theo1 = Or (Var 1) (Not (Var 1))
```

This will be reported as safe by the LH compiler.

```
{-@ nontheo1 :: Theorem @-}
nontheo = And (Var 1) (Not (Var 1))
```

This will be reported as unsafe.

4.2.2 Type level formula validity checking in Dependent Haskell

Validation of formulas in Haskell is a lot harder, but also more interesting.

As stated earlier, my approach will be introduce a singleton representation of the propositional formulas and then introduce a type level sequent calculus that can check for validity. Because we want to to check the formulas at compile time, a normal term level sequent calculus would not suffice. Therefore I will in this section introduce a relatively simple implementation of a type level sequent calculus.

Singleton data type

To define singleton version of the formula data type, we start by defining a kind F (for Formula) to represent the types that corresponds to the actual values. Then we create a GADT that for each data constructor uses the corresponding type of kind F in the type parameter.

```
data F a =
        V a           --Var
    |   A (F a) (F a) --And
    |   N (F a)       --Not
    |   O (F a) (F a) --Or
```

```
    |   I (F a) (F a) --Impl

data Formula s where
    Var :: (KnownNat n) => Formula (V n)
    Not :: Formula q -> Formula (N q)
    And :: Formula q -> Formula u -> Formula (A q u)
    Or :: Formula q -> Formula u -> Formula (O q u)
    Impl :: Formula q -> Formula u -> Formula (I q u)
```

The KnownNat constraint on n makes is so that the type n must be of kind Nat. It also gives a way to extract the integer associated with the type-level natural number at runtime. In other words, it makes it possible to get a value representation of the type level natural number. df

Sequent calculus

Sequent calculus is a proof calculus introduced by Gentzen[CITE]. For a thorough introduction to this system, the reader is advised to read [CITE] or most logic books. At its core it consists of a *sequent* and inference rules. A sequent consists of a list of formulas on the left, which are the assumptions, and a list of formulas on the right, which are the propositions. The inference rules are applied to both the assumptions and the propositions. The axiom of sequent calculus is when an atomic symbol appears as both an assumptions and a proposition.

$$\underbrace{A_1, A_2, ..., A_n}_{assumptions} \vdash \underbrace{B_1, B2, ..., B_n}_{propositions}$$

By using the DataKinds-extension, we can represent the a sequent at the type level. We do that by saying that something of kind Seq consists of two Side's created by the infix type constructor `:=>`,that represents the turnstile. Each Side contains a list of type level formulas and also a list of Nat's. The list of formulas are the representation of the comma-separated sequences of formulas at each side of the turnstile. The natural numbers are the atomic symbols collected at each side during the application of the inference rules.

```
data Seq = Side :=> Side

data Side = Side [F Nat] [Nat]
```

Find overlap between type level lists

The axiom of the sequent calculus, is when there is the same atomic symbol at the left and right side of the turnstile in the sequent. Because we store our atomic symbols in lists, we need to implement a way to check if there is any overlap between the atomic symbols at the left side and the right side. This is the same as checking that the intersection is non-empty. We defined two type families, `AnyOverlap` and `Find`, which works just like a normal

function on lists, but only at the type level on type level lists containing types of kind `Nat`.

```
type family AnyOverlap (xs :: [Nat]) (ys :: [Nat]) where
    AnyOverlap '[] ys      = 'False
    AnyOverlap (x : xs) ys = Find x ys || AnyOverlap xs ys

type family Find (x :: Nat) (xs :: [Nat]) where
    Find x '[]       = 'False
    Find x (x : ys) = 'True
    Find x (y : ys) = Find x ys
```

Implementation of the inference rules

To implement the inference rules, we implement a type family that applies every rule for formulas in the sequent calculus. When it finds an atomic symbol, it stores it in the variable-list for its corresponding side. When there are no more formulas left, it checks if there is any overlap between the variables on the left side and the right side, which represents the axiom in sequent calculus. If it is , the formula is valid and the type returned is Theo, and if not, the type returned is NonTheo.

It is implemented in a straight forward way by pattern matching on the types of kind F, which is the representation of the formulas at the type level. There is a one-to-one mapping between the inference rules of the sequent calculus and in the Sequent type family. When the sequent is split into two, the type level boolean and-operator `&&` is used.

One thing to mention is the use of `'` in front of constructors, as in `'Side` and `'[]`. This is sometimes needed when working with DataKinds because of ambiguity in the type checker. Sometimes it can't figure out it wheter to use the lifted type as a datakind or as a type operator. To disambiguate, we use `'`, which makes it explicitly refer to the lifted type generated from the us the DataKind extension.

Comparison of the logical inference rules and the implementation If we look at the definition of two following inference rules and their corresponding implementation at, we can see that the implementation closely follows the logical definition, and that should make the code easily understandable. Branching is done by using a type level boolean and-operator (`&&`).

$L\wedge$ rule: $\wedge\text{-L}\ \dfrac{\Gamma, A \wedge B \vdash \Delta}{\Gamma, A, B \vdash \Delta}$

```
--Left and rule
Sequent ('Side (A l r : as) avars :=> right) =
    Sequent ('Side (l : r : as) avars :=> right)
```

$R\wedge$ rule: $\wedge\text{-R}\ \dfrac{\Gamma \vdash \Delta, A \wedge B}{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}$

```
-- Right and rule
Sequent (left :=> 'Side (A l r : b) bvars) =
    Sequent (left :=> 'Side (l:b) bvars) && Sequent (left :=> '
       Side (r:b) bvars)
```

The full implementation of the all the inference rules then becomes :

```
type family Sequent (s :: Seq )  where
    Sequent ('Side '[] avars :=> 'Side '[] bvars) =
        AnyOverlap avars bvars

    --Left logical rules
    Sequent ('Side (V n : as) avars :=> right) =
        Sequent ('Side as (n : avars) :=> right)
    Sequent ('Side (N fm : as) avars :=> 'Side bs bvars) =
        Sequent ('Side as avars :=> 'Side (fm : bs) bvars)
    Sequent ('Side (A l r : as) avars :=> right) =
        Sequent ('Side (l : r : as) avars :=> right)
    Sequent ('Side (O l r : as) avars :=> right) =
        Sequent ('Side (l : as) avars :=> right)
        &&
        Sequent ('Side (r : as) avars :=> right)
    Sequent ('Side (I a b : as) avars :=> ('Side bs bvars)) =
        Sequent ('Side as avars :=> 'Side (a:bs) bvars)
        &&
        Sequent ('Side (b:as) avars :=> 'Side bs bvars)

    --Right logical rules
    Sequent ('Side as avars :=> 'Side (I a b : bs) bvars) =
        Sequent ('Side (a:as) avars :=> 'Side (b:bs) bvars)
    Sequent (left :=> 'Side (V n : b) bvars) =
        Sequent (left :=> 'Side b (n:bvars))
    Sequent (left :=> 'Side (A l r : b) bvars) =
        Sequent (left :=> 'Side (l:b) bvars) && Sequent (left :
           => 'Side (r:b) bvars)
    Sequent (left :=> 'Side (O l r : b) bvars) =
        Sequent (left :=> 'Side (l:r:b) bvars)
    Sequent ('Side as avars :=> 'Side (N fm : bs) bvars) =
        Sequent ('Side (fm : as) avars :=> 'Side bs bvars)
```

Listing 4.1: Implementation of all inference rules

Then we create a helper type family that that uses the "starts" evaluation of the Sequent type family by putting the type level formula at the right of the turnstile, and then returns Theo if the result is True and NonTheo if the result is False.

Then we can create a data type that only accepts values of type Formula fm where fm, the type level representation, is accepted as a theorem by the sequent calculus. We do this by constraining the result of applying the CheckTheo type family to the the type paramater to be equal to Theo.

```
type family CheckTheo fm where
    CheckTheo fm = If (Sequent ('Side '[] '[] :=> 'Side '[fm]
        '[])) Theo NonTheo

data Theorem (fm :: F Nat) where
    Theorem :: (CheckTheo fm ~ Theo) => Formula fm -> Theorem fm
```

To test that it works, we can define two formulas, theo1 and theo2, that represent theorems.

```
p1 :: Formula (V 1)
p1 = Var

-- not (p1 && not p1 )
theo1 = Theorem $ Not $ And p1 (Not p1)

-- p1 -> p1
theo2 = Theorem $ Impl p1 p1
```

The the type checker will evaluate the sequent calculus at compile time, since the sequent calculus will report that these formulas are valid, they will type check.

If we try to create a Theorem from a formula that does not represent a theorem, it the type checker will rightfully complain, because when checking the validity of this formula using CheckTheo, the result will be NonTheo, and to be a theorem it should have been Theo.

```
-- p1 -> p1
theo2 = Theorem $ Impl p1 p1
```

4.2.3 Comparison

In contrast to the natural numbers case study, the implementations are very different.

Expressiveness

In this case study, the expressivity of LH shines in comparison to that of Haskell. It shows in code size, where the LH implementation is only 15 lines of code compared to 55 lines of code. The Haskell version also includes the use of 6 language extensions and 2 modules from `base`, the standard library.

The LH solution is very straight forward, by mapping the formula data type to the logical operators and letting the SMT-solver do the validity

checking. The only non-trivial part is the uninterperated function, but even that should be decently understandable.

The Haskell version on the other hand implements a full logical calculus and also needs a singleton representation of the data type. Even though type level programming in Haskell isn't so different to normal term level Haskell programming when doing normal, it still adds a layer of complexity.

Ease of use

Reliability

The comparison of the reliability of these implementations are different than than in the natural numbers case study. Because in this case we don't really prove anything, we instead implement our own validity checker. So we can't use the type checkers to guarantee that out validity checkers are correct, at least not in an easy way. But there is still some question regarding termination. Because this our validity checker is run at compile time, it is the type checker that does the evalutation, and therefore it is the type checker that could loop forever, instead of the actual program.

In the case of LH, LH will check that our measure is indeed terminating, and it accepts it, because using the structural termination checker it sees that the arguments to the recursive calls decreases in size.

In the Haskell imlpementation on the other hand, we have no such termination checker. There actually is a simple termination checker for type families [GHC15b], but it does not accept more complex recursion, like in the Sequent type family, so it had to be disabled by using the extension UndecidableInstances. Therefore there is a possibility that the type checker will not terminate. But since the type family so closely mimics the actual inference rules which have been proven to be terminating, it should be easy to convince ourselves that it terminates.

It is important to note that if the Haskell type checker for some reason would would lead to infinite recursion, it would not mean that we would be able to come to any wrong conclusions, in our case that would mean to verify any formulas that aren't valid as theorems. It would simply fail to complete the type check. Infinite recursion would still terminate, because termination is still ensured by having a fixed-depth recursion stack[GHC15b].

Chapter 5

Case study : Verifying finger trees

Fingertrees

Persistant data structures

Standard data structures uses mutability, and therefore a change to the structure will destroy the old version. This does not work well in a functional setting because of the lack of mutability. And referential transparency. Data structures that does not destroy old versions, and allows access to all versions at any time are called *persistant* [DSST89].

All data structures in pure functional programming lanugages are persistant[Oka99].

Basic finger trees

```
type Digit a = [a]

data FingerTree a =
        Empty
    |   Single a
    |   Deep (Digit a) (FingerTree (Node a)) (Digit a)
```

So a fingertree is either empty (Empty), holds one value(Single), or contains left and right digits and another fingertree containing nodes of values.

Polymorphic recursion

If we call a tree of depth 0 a FingerTree with either no recursive FingerTree or an empty one, the maximum number of elements at depth 0 is 8, which is when both the left and right digits are full (4). Depth 1 can possibly include 3*8, since the size of a node is maximum 3. Depth 2 can then possible include 3*3*8, and generally Depth n can then possibly include $3^n * 8$ elements.

Monoids

Need more? In abstract algebra a monoid is a set with a associative binary operation and an identity element. Some standard examples would be addition over the natural numbers (including 0), with 0 as its identity element, multiplication over the natural numbers (including 0), with 1 as its identity element, and string concatenation over finite strings over a given alphabet with the empty string as its identity element.

This can represented in a Haskell context as a type, because you can look at a type as a set of values, with a function and a .

In the Haskell standard library they are implemented as a type class with a corresponding value `mempty` (the identity element) and an operation `mappend`, (the binary operation).

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

This is the same representation as used in the original paper and also in this thesis, except the use of unicode symbols and change from a prefix function to an infix function.

```
class Monoid a where
    ∅ :: a
    ⊕ :: a -> a -> a
```

Multiple instances

Multiple types can have different underlaying monoids. F.ex. addition and multiplication of integers. The way this is solved is to use a newtype-wrapper (refer to background), and make a monoid instance for the newtype. In the Haskell standard library, there is the newtype Sum for the addition-instance, and the newtype Product for the multiplication-instance.

Then one can use the different monoid instances by wrapping the values into a newtype with the desired monoid instance.

```
> Sum 1  mappend  Sum 2
Sum {getSum = 3}
> Product 1  mappend  Product 2
Product {getProduct = 2}
```

As mentioned earlier, newtypes does not have a runtime penalty, so there is no performance loss.

Annotated finger trees

```
data Node v a = node2 v a a | Node3 v a a a

data FingerTree v a =
    Empty
|   Simple a
|   Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)
```

Applications

Verifying the size of finger trees

To know information about the size of a container at compile time is useful in a lot of cases. It can avoid run-time errors by only allowing functions like `head` operate on containers with size greater than 0, i.e. containers with at least one element. It can also help avoid in having use types such as `Maybe` in return types, thats puts a burden on the caller of the function.

In this section we show how to use refinement types and equational proofs to prove that the size of a finger tree will have the same size as the list it was made from. We do that by

- finding a suitable size function for finger trees
- proving that adding an element at the left, increases the size by 1, using equational reasoning and proof combinators
- verifying the size of the finger tree returned from fromList

Refining the FingerTree data type

We start by creating the refined list type Dig to represent Digits, that is a list where the length is between 1 and 4.

```
{-@ type Dig a = {v:[a] | len v >= 1 && len v <= 4} @-}
type Digit a = [a]
```

This uses the pre-defined len measurement, which is just the straight forward length of a list. Because you can't use the same name for a refined type as the original Haskell type, we can't name the refined type Digit, so we go for Dig.

We need to refine the FingerTree data type to only only use actual Digit's, which so we say that that the left and right Digit needs to be of the abovementioned refined type Dig. This is done by just specifying the refinements of the data type in a LH type annotation.

```
{-@ data FingerTree a =
  Deep {
      lft::(Dig a) ,
      mid::(FingerTree (Node a)),
```

```
      rft :: (Dig a)
  }
  | Single a
  |   Empty
@-}
```

This makes it so that LH will check and verify that all the digits in FingerTree's actually have the valid length.

Size measurement

Now we need to create a way to to refer to the size of a fingertree, in the predicates of the refinements. (When talking about the size of a fingertree, I am refering to the number of values of type a in a fingertree of type FingerTree a).

We start by taking a naive approach to create a function that calculates the lgnth of a fingertree.

```
simpleSize :: FingerTree a -> Int
simpleSize Empty = 0
simpleSize (Single _) = 1
simpleSize (Deep l m r) = length l + simpleSize m + length r
```

This works fine at "level 0", but at level 1, a `Single (Node2 val1 val2)` will have the size 1, when we want it to have size2. The simple naive approach does not handle the polymorphic recursion. Due to the polymorphic recursion, there is no simple recursive function that can evaluate the size of a fingertree.

We need a way to calculate the number of a values in a node, even if the node is of type `Node a` or `Node (Node a)` or `Node (Node (Node a))` and so on.

We need to take advantage of higher order functions. The trick is to have a function argument should be called on each element in the node, and return the number of a-values.

The implementation of the node-size function is quite simple:

```
{-@ reflect nodeS @-}
nodeS :: (a -> Int) -> Node a -> Int
nodeS f (Node2 a b)   = f a + f b
nodeS f (Node3 a b c) = f a + f b + f c
```

It is just pattern matching and then summing up the results of applying the counting function on every element.

So when the type is Node a, every element contains 1 a-value, that counting-function is simply `_ -> 1` , the constant function returning 1. When we go up/down? a level, to `Node (Node a)`, we need to apply the node-size function that applies `_ -> 1` to every element. That is

accomplished by updating the counting function from `to1` to `nodeS to1`, by means of partial application.

Here is an example that shows how this works. The valued to be sized is `(Node2 (Node2 a2 b2) (Node3 a3 b3 c3))` of type `Node (Node a)`, and contains 5 a values. So in this context, the size is 5.

```
nodeS (nodeS to1) (Node2 (Node2 a2 b2) (Node3 a3 b3 c3))
nodeS to1 (Node2 a2 b2) + nodeS to1 (Node3 a3 b3 c3)
(to1 a2 + to1 b2) + (to1 a3 + to1 b3 + to1 c3)
(1 + 1) + (1 + 1 + 1)
5
```

We see here how the different function arguments makes it so that it is possible to calculate the correct amount of a values.

There is a equvivalent function for calculating the size of digits:

```
{-@ reflect digitS @-}
{-@ digitS :: (a -> Int) -> Dig a -> Int @-}
digitS :: (a -> Int) -> Digit a -> Int
digitS f [a]        = f a
digitS f [a,b]      = f a + f b
digitS f [a,b,c]    = f a + f b + f c
digitS f [a,b,c,d]  = f a + f b + f c + f d
```

So to fix our simple and naive finger tree size function, we add a function parameter, apply the function a the a values either directly (Single case) or with the digitS functions. In the recursive call, we need to send in a function that counts into the nodes of the recursive fingertree, and therefore we send in the result of partially applying nodeS to our function parameter. The general fingertree size function then becomes

```
{-@ reflect ftSizeGen @-}
ftSizeGen ::
    (a -> Int) -> FingerTree a -> Int
ftSizeGen _ Empty        = 0
ftSizeGen f (Single a)   = f a
ftSizeGen f (Deep l m r) =
    digitS f l + ftSizeGen (nodeS f) m + digitS f r
```

with an additional helper function, which calls the the general function with the correct first function argument to1.

```
{-@ reflect fingerTreeSize @-}
fingerTreeSize :: FingerTree a -> Int
fingerTreeSize t = ftSizeGen to1 t

{-@ reflect ftSizeGen @-}
ftSizeGen ::
    (a -> Int) -> FingerTree a -> Int
```

```
ftSizeGen _ Empty        = 0
ftSizeGen f (Single a)   = f a
ftSizeGen f (Deep l m r) =
    digitS f l + ftSizeGen (nodeS f) m + digitS f r
```

Size verification

Due to this fingerTreeSize-function not being a measure, we do not get the data constructor refinements which could lead to more automatic verification. We instead have to provide more manual proofs to verify our desired size properties.

We now have a way to refer to the size of a fingertree, in the types. Since our goal is to verify that when you create a fingertree from a list, the number of elements in the list and fingertree are equal. So our goal is to verify the following type.

```
{-@ fromList :: xs:[a] ->
    {t:FingerTree a | fingerTreeSize t == len xs} @-}
```

The implementation is a straight forward recursive function on a list, where addLeft, for now, is just another binding for `<|`.

```
fromList []     = Empty
fromList (x:xs) = addLeft x (fromList xs)

addLeft = (<|)

{-@ infix <| @-}
{-@ reflect <| @-}
(<|) :: a -> FingerTree a -> FingerTree a
a <| Empty                    =
    Single a
a <| Single b                 =
    Deep [a] Empty [b]
a <| Deep [b,c,d,e] m sf      =
    Deep [a, b] (Node3 c d e <| m) sf
a <| Deep l m r               =
    Deep (consDigit a l) m r
```

To see what needs to be done, we can start by trying to prove this property. For the base case : [] `fingerTreeSize Empty == len xs` by just applying the functions, we have that `0 == 0`, and the base case is done.

For the inductive case : (x:xs)

```
len (x:xs) == addLeft x (fromList xs)

1 + len xs == addLeft x (fromList xs)
```

we know from the induction hypothesis that `len xs == fingerTreeSize (fromList xs)`. So if we could use a lemma that says that `fingerTreeSize (addLeft x (fromList xs)) == 1 + fromList xs` we would reach our goal. So lets prove that lemma.

Generalized finger tree size proof

We start by verifying the generalized size function, the one with the counting function argument. We specify that that when you add an element to the left of a fingertree, the generalized size should be equal to the generalized size of the original tree plus the count of the new element. The type of the proof then becomes:

```
{-@ lem_add_l_gen ::
    f:(a -> Int) -> a:a -> t:FingerTree a ->
        { ftSizeGen f (a <| t) == ftSizeGen f t + f a }
@-}
```

All the base cases are trivial, and are automaticly proven by PLE. The inductive case is not automaticly proven, and here we need to show a equational proof and use the inductive hypothesis.

```
lem_add_l_gen f a t@(Deep [b,c,d,e] m sf)
  =   ftSizeGen f t + f a
  === digitS f [b,c,d,e] + ftSizeGen (nodeS f) m + digitS f sf +
       f a
  === f a + f b + digitS f sf + f c + f d + f e + ftSizeGen (
      nodeS f) m
  === f a + f b + digitS f sf + nodeS f (Node3 c d e) +
      ftSizeGen (nodeS f) m
        ? lem_add_l_gen (nodeS f) (Node3 c d e) m
  === f a + f b + digitS f sf + ftSizeGen (nodeS f) ((Node3 c d
      e) <| m)
  === ftSizeGen f (Deep [a, b] (Node3 c d e <| m) sf)
  === ftSizeGen f (a <| (Deep [b,c,d,e] m sf))
  === ftSizeGen f (a <| t)
  *** QED
```

Now we need to verify that the size of a finger tree increases by 1, when adding at the left. By using the previous lemma, this comes easy.

```
{-@ lem_add_l :: a:_ -> t:_ -> { fingerTreeSize (a <| t) == 1 +
    fingerTreeSize t} @-}
lem_add_l :: a -> FingerTree a -> Proof
lem_add_l a t
  =   fingerTreeSize (a <| t)
  === ftSizeGen to1 (a <| t)
        ? lem_add_l_gen to1 a t
  === ftSizeGen to1 t + to1 a
  === fingerTreeSize t + 1
```

```
  *** QED
```

We just carry out the stepwise function application and use the `lem_add_l_gen` lemma.

Now we can add the refined type to add left that the resulting fingertree increases by one in size, by using the `lem_add_l` lemma.

```
{-@ addLeft :: a:_ -> t:_ -> { v: _ | fingerTreeSize v == 1 +
    fingerTreeSize t} @-}
addLeft :: a -> FingerTree a -> FingerTree a
addLeft a t = (a <| t) ? lem_add_l a t
```

Now our fromList definition will type check without further proof, because of the refinement in addLeft.

```
{-@ fromList :: xs:_ ->
  {t:_ | fingerTreeSize t == len xs} @-}
fromList :: [a] -> FingerTree a
fromList []     = Empty
fromList (x:xs) = addLeft x (fromList xs)
```

To see an example of the verified property in use, we can use fromList on a list of length 10 and specify that the resulting fingertree also has? the same length.

```
{-@ ft1 :: {v:_ | fingerTreeSize v == 10} @-}
ft1 :: FingerTree Int
ft1 = fromList [1,2,3,4,5,6,7,8,9,10]
```

This type checks, and our job is done.

Note: fromList can also be defined in terms of addRight. A proof of the verification using addRight instead is included in the appendix.

Thoughts

Even though we did not get much help from the automatic capabilities of Liquid Haskell due to the fact that our function was not in the restricted subset of measures, the verification went smoothly. We did have to write some extra lines of code in the proofs, but nothing in this verification was hard.

It is promising that Liquid Haskell can prove properties on advanced data structures defined using features such polymorphic recursion, with such ease.

5.1 Proving the splitting theorem

5.1.1 Proving the splitDigit theorem

Finger trees is a general purpose data structure. To be able to use finger trees as more specific data structures, such as sets or max prioirty queues, we need to be able to split the fingertree based on a predicate.

The function that splits a finger tree based on a predicate is called splitTree and is implemented in the finger tree paper. In this section we will not be focusing on that function, but instead on the splitDigit function, which is a analaogue function that insteads splits a digit.

The implementation in the paper uses the Measured type class mentioned earlier in this chapter.

```
splitDigit :: Measured a v => (v -> Bool) -> v -> [a] -> DSplit
    a
splitDigit p i [a] = DSplit [] a []
splitDigit p i (a:as)
    | p i' = DSplit [] a as
    | otherwise =
            let
                DSplit l x r = splitDigit p i' as
            in
                DSplit (a:l) x r
        where
            i' = i <> measure a
```

In regards to proving the split Lemma theorem, an important aspect of this implementation, is that the `v` type needs to be a monoid. This is important because we need to use the fact that `<>` is associative and that the the measurement of an empty list returns the identity element for that monoid.

Due to restrictions in LH regarding type classes, I found out that the easiest way to handle this was to change from a type class to normal function arguments.

So the implementation of splitDigit that will be used in this chapter have `ap` (the "monoid append" `<>`) and measure provided as arguments to splitDigit.

```
splitDigit :: (v -> v -> v) -> (a -> v) -> (v -> Bool) -> v -> [
    a] -> DSplit a
splitDigit ap measure p i [a] = DSplit [] a []
splitDigit ap measure p i (a:as)
    | p i' = DSplit [] a as
    | otherwise =
            let
                DSplit l x r = splitDigit ap measure p i' as
            in
```

```
            DSplit (a:l) x r
    where
        i' = ap i (measure a)
```

$$\neg p\ i \wedge p(i \oplus \|d\|)$$
$$\Longrightarrow$$
$$let\ Split\ l\ x\ r\ =\ splitDigit\ p\ i\ d$$
$$in$$
$$l +\!\!+ [x] +\!\!+ r = d\ \wedge\ \neg(p(i \oplus \|l\|))\ \wedge p(i \oplus \|l\| \oplus \|x\|)$$

2

5.1.2 Proof in Liquid Haskell

Setup, the the argument passing. Free monoid, list ++. DSplit?

(Remove this?) The definition of splitDigit is the same as in the paper, except the added arguments "ap" and "measure" and the use of DSPlit instead of Split.

```
splitDigit ::
    (v -> v -> v) -> (a -> v) -> (v -> Bool) -> v -> [a] ->
       DSplit a
splitDigit ap measure p i [a] = DSplit [] a []
splitDigit ap measure p i (a:as)
| p i' = DSplit [] a as
| otherwise =
        let
            DSplit l x r = splitDigit ap measure p i' as
        in
            DSplit (a:l) x r
    where
        i' = ap i (measure a)
```

For clearity and to reduce the complexity of the code, theorem are split into 3 parts, one for each conjuct. So lets look at the not-p conjuct.

```
{-@
split_lemma_notp ::
       ap:(v -> v -> v)
    -> e : v
    -> assoc: (x:v -> y:v -> z:v -> {ap x (ap y z) = ap (ap x y)
        z})
    -> identity : (x:v-> {ap x e = x && ap e x = x})
    -> mes:(a -> v)
    -> p:(v -> Bool)
    -> i:v
    -> t:Digit a
    -> prnp:{ Proof | not (p i) }
    -> prp :{ Proof | p (ap i (measureList ap e mes t)) }
    -> { (not (p i) && p (ap i (measureList ap e mes t)))
        =>
            not (p (ap i (measureList ap e mes (getL (splitDigit
                ap mes p i t)))))
        }
@-}
```

The return type states the proof that we want, which is the second conjuct. The syntatic? difference, is because of the type class issues with Liquid Haskell, we have to specify the measure ourselves, and also because of the lack of pattern matching inside Liquid types, we use a function to extract the left part of the split.

Except the aforementioned extra arguments, we have also added the two assumptions? as arguments to the proof. That makes it so that we can be explicit about where to use the facts that the assumptions carry, and therefore help LH to type check, instead to having LH to guess where to use those assumptions. When we take the assumptions as arguments, and using the fact that in curry howard, function arrows are implication arrows, we could have dropped left side of the arrow of the theorem in the return type. But it remains, due to the fact that it makes it easier to see the comparison to the theorem in the paper.

Base case : [a] and Inductive case : (a:as) and p i' holds

The base case is very straigth forward, is is just applications of the function definitions and use of the identity proof?

```
=   not (p (i  ap  measureList ap e measure (getL (splitDigit ap
    measure p i [a]))))
=== not (p (i  ap  e)) ? identity i
=== not (p i)
*** QED
```

The proof of the inductive case where p i' holds are equal , you just have to change [a] to (a:as).

Inductive case : (a:as) and not (p i') holds

Here is where the proof gets a little trickier, in terms of making LH accept the proof.

```
let
    DSplit l x r = splitDigit ap measure p i' as
    lemmaL
            =   p (ap i (measureList ap e measure (a:l)))
            === p (ap i (ap (measure a) (measureList ap e
                measure l))) ? assoc i (measure a) (measureList
                ap e measure l)
            === p (ap (ap i (measure a)) (measureList ap e
                measure l))
            *** QED
in
    lemmaL
    &&&
    split_lemma_notp
        ap
        e
        assoc
        identity
        measure
        p
        i'
        as
        (not (p i') *** QED)
        lemma_i_a_as_assoc
    where
        i' = i  ap  measure a
        lemma_i_a_as_assoc
            =   p (ap i (measureList ap e measure (a:as)))
            === p (ap i (ap (measure a) (measureList ap e
                measure as))) ? assoc i (measure a) (measureList
                 ap e measure as)
            === p (ap (ap i (measure a)) (measureList ap e
                measure as))
            *** QED
```

The let in the theorem and the let in the splitDigit-case, leads to a nested let, which makes it knotete.

The assumption-proofs comes from the guard? and the assoc-proof. Besides the use of the induction hypothesis from the recursive call, the most important part is lemmaL . This is inside the scope of where l was defined, so that it has access to l. It then states that `p (ap i (measureList ap e measure (a:l)))` is equal to `p (ap (ap i (measure a)) (measureList ap e measure l))` by associativity, and then with automatic application of congruence with not and using the induction hypothesis it leads to the proof we wanted. (Clean up)

5.1.3 Split tree proof - by hand

Base case : Single x

notP and p both holds trivially from identity. The order also holds from the fact that toList of Empty is [] and then that [] is the left and right identity.

Inductive case : Deep v pr m sf : p vpr

From the split digit theorem, we have that $\neg p(i \oplus \|l\|)$. measure l = measure (toTree l), any name for this? some morphishm?. So then $\neg p(i \oplus \|toTree\ l\|)$ holds.

Inductive case : Deep v pr m sf : p vm

From the fact that this guard is below the *p vpr* guard means that we have $\neg p\ vpr$. This makes the notP assumtion to the recursive call hold, and that gives us $\neg p(vpr \oplus \|m\|)$. This makes the assumption to the splitDigit call with the `vpr` $\oplus$`|ml|` accumulator hold.
That gives us $\neg p(vpr \oplus \|ml\| \oplus \|l\|)$ and then

```
vpr ⊕ |ml| ⊕ |l|
    =   {   expanding of vpr and associativty }
i ⊕ |pr| ⊕ |ml| ⊕ |l|
    =   { deepR lemma }
i ⊕ |deepR pr ml l|
```

$$
\begin{aligned}
& vpr \oplus \|ml\| \oplus \|l\| \\
& \quad = \{\ expanding\ of\ vpr\ and\ associativty\ \} \\
& i \quad \oplus \|pr\| \oplus \|ml\| \oplus \|l\| \\
& \quad = \{\ deepR\ lemma\ \} \\
& i \quad \oplus \|deepR\ pr\ ml\ l\|
\end{aligned}
$$

Inductive case : Deep v pr m sf : otherwise

in the otherwise case we have not (p vpr) and not (p vm).

Since splitDigit is called with vm as its accumulator, the lemma-assumption holds from the case. From the split digit lemma we have `vm` $\oplus$ `|li|` and

```
vm ⊕ |li|
    =    { inling vm }
(i ⊕ |pr| ⊕ |m|) ⊕ |li|
    =    { by assicativity }
i ⊕ |pr| ⊕ |m| ⊕ |li|
    =   { deepR pr m l == |pr| ⊕ |m| ⊕ |l| : need more details
        }
i ⊕ |deepR pr m l|
```

which means that we have

```
¬ p (i ⊕ |deepR pr m l|)
```

which is what we wanted.

5.1.4 Mutual recursion

To implement deque operations, the authors of [HP06] define what they call "views". These views represents a way to look at the left or right end of a sequence. They are either empty or contain the element at the end togheter with the rest of the sequence. The following is the data type to represent a view of the left end:

```
data ViewL s a = NilL | ConsL a (s a)
```

`NilL` represents an empty view, where the sequence is empty, so there is no element at the end. In the other case, when the sequence is not empty , the first argument `a` of `ConsL` represents the element at the left end, and the second argument `s a` represents the rest of the sequence.

One thing to note, if we remember kinds from the Haskell background, is that the first type parameter to `ViewL`, `s`, is an example of a higher kinded type with kind `* -> *`. This actually allows this view to be used for other "containers" than fingertrees, like lists

```
listViewExample :: ViewL [] Int
listViewExample = ConsL 1 [2,3,4]
```

or even functions

```
listViewExample :: ViewL ((->) Int) Int
listViewExample = ConsL 0 (+1)
```

Implementation

The actual values of the data type ViewL are created by a function `viewL`, which uses the function `deepL`.

```
viewL Empty          = NilL
viewL (Single x)     = ConsL x Empty
viewL (Deep (p:pr) m sf) = ConsL p (deepL pr m sf)

deepL :: [a] -> FingerTree (Node a) -> Digit a -> FingerTree a
deepL [] m sf  = case viewL m of
    NilL -> toTree sf
    ConsL a m' -> Deep (nodeToList a) m' sf
deepL pr m sf = Deep pr m sf
```

Mutual recursion

We see here that `viewL` and `deepL` are mutually recursive, meaning that they are defined in terms of each other. If we assume that `toTree` and `nodeToList` will terminate, we can see `viewL` will terminate if `deepL` terminates and vice versa. If we try to typecheck this definition without giving an explicit termination metric LH will reject the definitions, because the automatic termination checker is not able to find the termination metric itself.

Termination metric

So how can we make LH see that these mutual recursive functions are terminating?

To begin, we notice that the structural size of the fingertree-argument to `viewL` is smaller than the fingertree-argument in in the mutually recursive call to `deepL`. But in `deepL`, the fingertree-argument is the same as in the call to `viewL`, so the argument does not get smaller.

So the size of the fingertree-argument alone does not work as a decreasing metric. We notice that either the size of the fingertree-argument stays the same or there is a call to `viewL`.

From this we can assign `viewL` the number 1 and `deepL` the number 0, and have a termination metric that says : either the fingertree-argment decreases in size, or it stays the same and the assigned number for the function in the mutually recursive call decreases.

If we then define a measure for the structural size of a fingertree and add with the explicit termination metric, LH will accept the definitions as terminating.

```
{-@ measure ftStructuralSize ::
    FingerTree v a -> {v:Int| v >= 0} @-}
ftStructuralSize Empty = 0
ftStructuralSize Single{} = 1
ftStructuralSize (Deep _ _ m _) = 1 + ftStructuralSize m

viewL :: ft:FingerTree v a
    -> ViewL (FingerTree v) a / [ftStructuralSize ft, 0]

deepL ::
       Digit a
    -> m:FingerTree v (Node v a)
    -> {v:[a] | len v <= 4 }
    -> FingerTree v a / [ftStructuralSize m, 1]
```

Now `viewL` will pass the termination check because

```
\texttt{[ftStructuralSize m, 1] < [ftStructuralSize (Deep (p:pr)
    m sf), 0]}
```

and deepL will pass because

```
\texttt{[ftStructuralSize m, 0] < [ftStructuralSize m, 1]}
```

Chapter 6

Conclusions and Future Work

Throughout this thesis we have investegated the usability and usefullness of Liquid Haskell in different contexts. This concluding chapter will summarize and evaluate the results we have gotten from the case studies and also include my persons conclusive summarized experience.

At the end of this chapter we will also present some interesting ideas that we did not have the time to pursue, that are eligible candidates for future work.

6.1 Conclusions

6.2 Future work

In this section we summarize possible future work.

Comparison of Liquid Haskell and Dependent Haskell

Since the two comparisons in this thesis tackles problems that are kinda small, it would be natural to look at a more complex example. To connect this part of the thesis to the finger trees chapter, it would be interesting to try to use Dependent Haskell on finger trees. To use Dependent Haskell to implement size verified finger trees and then compare the implementation to the Liquid Haskell version would probably give meaningful insights into the different techniques.

Finger trees

Use different representation of Digit

One interesting aspect of our use of Liquid Haskell on finger trees, is that the representation of digits in our implementation and actual practical implementations of finger trees are different.

We use the representation that is used in the finger tree paper, which is lists with 1,2,3 or 4 elements. This can easiliy be encoded as a refinenement type in Liquid Haskell in the following way:

```
{-@ type Digit a = {v:[a] | len v >= 1 && len v <= 4 @-}
```

This representation makes the implementation of finger trees, and especially the implementation of the appending of finger trees, more concise and readable. The downside is that lists in Haskell are recursivly defined, and that leads to worse performance than a non-recursive data type.

So the representation used in an actual practical implementations of finger trees use the following representation :

```
data Digit a =
        One a
    |   Two a a
    |   Three a a a
    |   Four a a a a
```

This implementation is more effienct and more accurate, but makes the number of different cases explode. The implementation of append goes from 6 lines of code in [HP06] to over 200 [Ro06]. Every other function that deals with digits also inceases in implementation size due to the added number of cases.

Therefore I made the choice to use the implementation in the paper.

It would be interesting to try do the size verification and splitDigit proof using the non-recurisve Digit data type, to compare the implementation and proofs and also test the performance of the Liquid Haskell type checker.

Size verification

We did manage to verify important properties regarding the size of a finger tree, like that adding an element at the left increases the size by one and also that that the size of a finger tree built from a list, corresponds to the length of the list. What is remaining is to verify the size of appending finger trees, which would make Liquid Haskell accept a definiton of append with the following refined type :

```
{-@ app ::
    ft1:FingerTree a ->
    ft2:FingerTree a ->
    {ft3 : FingerTree a | ftSize ft1 + ftSize ft2 == ftSize ft3}
        @-}
app :: FingerTree a -> FingerTree a -> FingerTree a
app = ...
```

.

It would also be interesting to do the size verification by verifying a finger tree that uses the Size-monoid from the finger tree paper as measurement.

splitTree theorem

In this thesis we only managed to prove the splitDigit theorem, not the actual splitTree theorem. It would be natural complete that proof.

Proving the splitTree theorem would also verify that the sizes of the resulting split, because of the first conjunct that says that the resulting split is actually a split of the original finger tree.

6.3 Related work

Dependent Haskell, singletons.

Dependently typed languages, Idris,Agda,Coq.

Bibliography

[Ro06] Ross Paterson, Ralf Hinze. fingertree hackage package, 2006. http://hackage.haskell.org/package/fingertree, Last accessed on 2019-11-13.

[Si14] Simon Peyton Jones. Ghc wiki : dependent haskell, 2014. https://gitlab.haskell.org/ghc/ghc/wikis/dependent-haskell, Last accessed on 2019-11-13.

[car17] Why we are building cardano, 2017. https://whycardano.com/science-and-engineering/, Last accessed on 2019-11-13.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[GHC15a] GHC Team. 7.8. kind polymorphism and promotiom chapter 7. ghc language features, 2015. https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/kind-polymorphism-and-promotion.html, Last accessed on 2019-11-13.

[GHC15b] GHC Team. Glasgow haskell compiler user's guide - 9.8.3.5. undecidable instances, 2015. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#undecidable-instances, Last accessed on 2019-11-13.

[GHC15c] GHC Team. Glasgow haskell compiler user's guide - 9.9. type families, 2015. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#type-families, Last accessed on 2019-11-13.

[Gis18] Matthías Páll Gissurarson. Suggesting valid hole fits for typed-holes (experience report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 179–185, 2018.

[HP06] Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

[Hut16] Graham Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[PM16] Ricardo Peña-Marí. An introduction to liquid haskell. In *PROLE*, 2016.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.

[RR08] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[VBK+18] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 132–144, 2018.

[VSJ14a] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: experience with refinement types in the real world. In *Haskell*, 2014.

[VSJ+14b] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282, 2014.

[VWPJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Simple unification-based type inference for gadts. Technical Report MS-CIS-05-22, April 2006. ACM SIGPLAN International Conference on Functional Programming (ICFP'06).