Verifying correctness of *Haskell* programs using the programming language *Agda* and framework *agda2hs*

Dixit Sabharwal[1], Jesper G.H. Cockx[1], Lucas F.B. Escot[1]
[1]TU Delft

June 27, 2021

Abstract

Equational reasoning based verification address some of the limitations of classical testing. The Curry-Howard correspondence shows a direct link between type systems and mathematical logic based proofs. *Agda* is a language with totality and dependent types which makes use of the CH isomorphism to support equational reasoning in its programs. '*agda2hs*' attempts to bring this formal verification to the *Haskell* ecosystem, by providing a translation between Haskell and *Agda* programs. This project will serve to test the viability of this framework by re-writing the Haskell library '*Data.Map*' in the subset of Agda defined by *agda2hs* and verifying properties of various functions in the library using Agda and dependent types.

1 Introduction

The importance of testing code is well known to all. Usually this is accomplished by identifying the different execution paths in your program and passing through each ensuring it works as expected. However, such methods leave a lot of room for human error — the programmer may not identify all execution paths correctly, the inputs used might not be sufficient to see unexpected behaviour etc.

Equational reasoning addresses some of these concerns of classical testing[16]. Properties of the program can be written as statements of propositional and predicate logic, and proofs of these properties can be derived through natural deduction and equational reasoning. This allows one to verify the correctness of a program not by running it on a selected sample of inputs, but by logically reasoning about the workings of each function. The Curry-Howard isomorphism [22] defines a direct link between such mathematical proofs and the type systems used in programming languages. It shows a correspondence between the formulas in logic systems and the elements of a type system, and also between the proof systems used and respective models of computation. This implies a programming language might be able to support writing and checking such mathematical proofs, if it uses a model of computation described in the isomorphism and its type-system supports all elements described in the correspondence.

Haskell[5] is a strongly-typed purely-functional programming language. In his book ”Programming in Haskell” Hutton [12] provides an example of using mathematical proofs for formally verifying properties of Haskell functions. The main limitation in his approach is that the given proofs are external verifications on paper, completely independent from the program itself. Hence the proof could contain a mistake or the definition of the function might be changed without respective effects on the proof. Due to limitations in Haskell, with regards to totality and termination checking, it is not possible to program these mathematical proofs in Haskell itself.

1.1 Exiting works and motivation

There are now frameworks which allow for such equational reasoning based proofs to be coded in Haskell, alongside the functions and algorithms they verify. One example of such a framework is *LiquidHaskell*[6], which uses ‘refinement types’ (see [20]) to enforce totality in Haskell and provide support for equational proofs. Refinement types are used to apply input contracts and correctness properties to function parameters eliminating the use of error clauses. It also packs a stronger termination checker to warn about infinite loops in functions.

Another approach is taken by the *agda2hs* project[2], which is to use ‘dependent types’. It defines a common subset between *Agda*[1], a dependently-typed and total programming language, and Haskell. This allows users to write the programs in a subset of Agda, and then use Agda’s support for equational proofs to verify properties about said program. As the base program itself is limited to the common subset language, the framework then provides a dictionary translation of the program into Haskell.

The reason for translating back to Haskell code is to gain access to the already existing ecosystem that comes with a mature language like Haskell. Developer familiarity and lots of existing libraries, frameworks, and projects currently exist utilising Haskell. While growing in popularity, the Agda ecosystem is currently vastly smaller and mostly research based with very few commercial/consumer applications.

Previous attempts have been made using a similar method of translating Haskell programs to a language that is more suited to be a proof assistant, for example the *hs-to-coq* framework (see [18]), which translates a Haskell program to the Coq language[4]. A major difference is that agda2hs works by writing the programs in Agda where the verification can be performed alongside development of the program, similar to the Test Driven Development process used in software engineering, and then provides a translation to Haskell code. Whereas, *hs-to-coq* takes existing Haskell code and translates it to Coq, primarily a proof-assistant, which has support for equational proofs. The similarity of the Agda syntax to its Haskell counterpart might also make Agda a better/easier candidate for this use case.

1.2 Research question

This project will utilize the Haskell library *Data.Map*[3] to analyse if the implementation of this library falls within the common subset of Haskell and Agda as identified by *agda2hs*. If not, is there an alternative implementation possible that does or else, what extensions to *agda2hs* are needed to implement the full functionality of the library.

If successful, the project will further the analysis to identifying, formally stating, and proving properties and in-variants guaranteed by the library. It will empirically observe the amount of time and effort required for implementation of proofs and how the verification process affects the program implementation itself.

The findings from the project will be used to further the development of the *agda2hs* framework by eliciting new requirements as well as identifying current limitations.

2 Preliminaries

2.1 Base library Data.Map

Data.Map is a Haskell library from the *containers*[1] package which provides an efficient implementation of the map data-structure for Haskell programmers. A Map in this library is internally represented as a balanced binary search tree (BBST) :

```
data Map k a = Bin Nat k a (Map k a) (Map k a) | Tip
```

Where `Tip` is the constructor for a leaf node in the tree and doesn't store any data. And `Bin` is a constructor for an internal node of the tree, with the parameters in order being;

- value of type Nat which stores the size of the tree where current node is root,
- values of type *'k'* and *'a'* which represent key and value elements resp. stored in that node, and finally
- the left and right sub-trees which again have type `Map k a`.

The library consists of around 170 functions which perform various operations such as insertion, deletion, folds, list conversions, merges etc. on the `Map` datatype.

2.2 Curry-Howard based Verification

Users of this library will undeniably want the operations conducted in the library to be error-free. One way to ensure that the library code functions as expected is classical testing, where a function is executed with a variety of inputs and the test suite checks for expected outcomes. This approach however is heavily reliant on the generation of an input space that covers all possible execution paths of the function, this step in itself is vulnerable to errors and mistakes on the part of the programmer. Incorrect expectations on the outcome of test cases is another possible error that might creep into the test suite.

Equational reasoning based verification of properties is a possible alternative to classical testing which doesn't suffer from the concern of generating a comprehensive input space. As defined by the Curry-Howard isomorphism, there is a correspondence between such mathematical proofs and type systems. This correspondence is summarised in tables 2 and 1. Due to Agda's totality and its support for dependent types[15], it is possible to write properties of the program, defined as statements of propositional or predicate logic, as functions of a respective type. These properties can then be proved by defining a function of that respective type[17].

[1]containers - Data.Map : `https://hackage.haskell.org/package/containers`

Logic side	Programming side
hypothesis	free variables
implication elimination	application
implication introduction	abstraction

Table 1: Correspondence between natural deduction proof systems and lambda calculus based programming systems.[23]

Propositional logic		Type system
proposition	P	type
proof of proposition	$p : P$	program of type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	bottom/empty type
universal quantification	(x : A) -> P x	dependent function type
existential quantification	Σ A (λ x -> P x)	dependent pair type

Table 2: Correspondence between logic formulae and elements of type systems.[9]

Consider an example of a simple program based on lists. Given a list of integers defined as :

```
data List : Set where
  []    : List {- empty list -}
  _::_ : Int -> List -> List {- prepend to list -}
```

Properties of the program which can be considered statements of equational reasoning and predicate logic, should be expressed equivalently as the type signature of a function. Given a concatenation function '++', associativity is a property that might be expected of the function. Expressed in predicate logic —

$$\forall\, xs\ ys\ zs\ (xs \mathbin{++} ys) \mathbin{++} zs \equiv xs \mathbin{++} (ys \mathbin{++} zs)$$

and expressed as a type signature —

```
_++_ : (p : List) -> (q : List ) -> List
[] ++ xs = xs
(y :: ys) ++ xs = y :: (ys ++ xs)

property : (xs ys zs : List) -> (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
```

Proof of a property is then equivalent to the definition of a function with that type signature. So for our implementation of concatenation we can prove this property as such —

```
property [] ys zs =
  begin
    ([] ++ ys) ++ zs
  ≡⟨⟩  {- by definition of the ++ function -}
```

```
    (ys) ++ zs
  ≡⟨⟩  {- by definition of the ++ function -}
    [] ++ (ys ++ zs)
  ∎

property (x :: xs) ys zs =
  begin
    ((x :: xs) ++ ys) ++ zs
  ≡⟨⟩  {- by definition of the ++ function -}
    (x :: (xs ++ ys)) ++ zs
  ≡⟨⟩  {- by definition of the ++ function -}
    x :: ((xs ++ ys) ++ zs)
  ≡⟨ cong (\ ts -> x :: ts ) (property xs ys zs) ⟩
  {- by inductively applying the same property -}
    x :: (xs ++ (ys ++ zs))
  ≡⟨⟩ {- by definition of the ++ function -}
    (x :: xs) ++ (ys ++ zs)
  ∎
```

2.3 Identity type and Equational Reasoning

The equational proof demonstrated above uses constructs in Agda such as ≡, ≡⟨⟩, *cong* etc. A brief introduction to these elements is provided below.

The identity type $x \equiv y$ represents equality between two propositions x and y. It is defined as an indexed datatype with a single constructor and is bound to the built-in Equality of Agda.

```
data _≡_ {A : Set} : (x y : A) → Set where
  refl : (x : A) → x ≡ x
{-# BUILTIN EQUALITY _≡_  #-}
```

We can match on a proof of type $x \equiv y$ by either instantiating the variables such that Agda can unify x and y and match on *refl* or x and y become obviously different and the equality cannot be instantiated leading to an absurd pattern.

Using this identity type for equality of propositions, properties such as symmetry, transitivity and congruence can also be proved.

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl

cong : {A B : Set} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

The transitivity function is then re-written in the form of direct implication and induction operators which were used in our example equational proof.

```
begin_ : {A : Set} → {x y : A} → x ≡ y → x ≡ y
begin p = p

_end : {A : Set} → (x : A) → x ≡ x
x end = refl
```

```
_≡⟨_>_ : {A : Set} → (x : A) → {y z : A} → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ p ⟩ q = trans p q

_≡⟨⟩_ : {A : Set} → (x : A) → {y : A} → x ≡ y → x ≡ y
x ≡⟨⟩ q = x ≡⟨ refl ⟩ q
```

See [21] for a more detailed explanation of the constructs used for equational reasoning in Agda and how to use them.

Advantages of programming these equational proofs alongside the implementation of the program are — firstly the proof is type-checked by Agda to be correct[17] and secondly these proofs will be checked each time the library is compiled ensuring that any updates to the function require upkeep of the proofs as well.

The *'agda2hs'* framework attempts to make such verification techniques available to Haskell programmers. It does so by defining a subset of the Agda language, which ideally would contain the entire core Haskell language. This enables any program currently written in Haskell to have an equivalent definition in Agda. With the support of the complete Agda type-system including dependent types, properties of the program can then be formally verified through equational proofs. *agda2hs* then preforms a dictionary translation of the program to Haskell code, which is possible as the program itself is limited to the subset language and any instance of the verification process is erased during translation.

3 Implementation

The agda2hs framework covers a large enough part of the Haskell language that the core functionality of the library could be ported into the Agda subset without any major changes to the structure of the code. This section will describe the process of how the 'Data.Map' library was implemented in Agda and agda2hs.

The code for this implementation is available at `https://github.com/dxts/agda2hs-map` and the code for the agda2hs framework used can be found at `https://github.com/agda/agda2hs`.

3.1 Total functions

The implementation process is straightforward for total functions, i.e. functions that do not error or infinitely loop. For example the following Haskell function would undergo minimal changes to be written in Agda —

```
{- haskell -}
singleton :: k -> a -> Map k a
singleton k x = Bin 1 k x Tip Tip

{- agda -}
singleton : {k a : Set} → k → a → Map k a
singleton k x = Bin 1 k x Tip Tip
```

3.2 Case matching

Another commonly used Haskell construct is the `` `case_of_' `` pattern-matching clause. This is supported in agda2hs using a function of the form `` `case_of_ : a → (a → b) → b' ``, which takes as parameters a single value and a pattern matching lambda. Agda2hs handles conversion of this idiom to native `case_of_' clause in Haskell.

```
{- haskell -}
findWithDefault :: Ord k => a -> k -> Map k a -> a
findWithDefault def !_ Tip = def
findWithDefault def k (Bin _ kx x l r) = case compare k kx of
      LT → findWithDefault def k l
      GT → findWithDefault def k r
      EQ → x

{- agda2hs -}
findWithDefault : {k a : Set} {{_ : Ord k}} → a → k → Map k a → a
findWithDefault def _ Tip               = def
findWithDefault def k (Bin _ kx x l r)  = case (compare k kx) of
  λ {
    LT → findWithDefault def k l
  ; GT → findWithDefault def k r
  ; EQ → x
  }
```

3.3 Adding totality to a function

Haskell doesn't enforce totality on functions, this means that functions are allowed to throw program-crashing errors and/or be non-terminating. Agda however doesn't allow such behaviour of functions, and enforces strict totality.

Firstly, implementation of error-throwing functions in Agda can be achieved by adding pre-conditions to the type signature of the function. These pre-conditions can be used to limit the parameters to a subset of the elements of their respective type. These pre-conditions can also serve to verify that the function throws an error only in the cases intended.

```
{- haskell -}
find :: Ord k => k -> Map k a -> a
find _ Tip = error "Given key is not an element in the map"
find k (Bin _ kx x l r) = case compare k kx of
    LT → go k l
    GT → go k r
    EQ → x

{- agda2hs -}
error : {@(tactic absurd) i : ⊥} → String → a
error {i = ()} err

_□_ : (key : k) → Map k a → Set
_ □ Tip = ⊥
k □ (Bin _ kx _ l r) with (compare k kx)
... | LT = k □ l
... | GT = k □ r
... | EQ = ⊤

find :  (key : k) (map : Map k a) → {prf : key □ map} → a
find key Tip = error "Given key is not an element in the map"
find key (Bin sz kx x l r) {prf} = match (compare key kx) {refl}
  where
    match : (o : Ordering) → {eq : compare key kx ≡ o} → a
    match LT {eq} = find key l {□L sz key kx x l r eq prf}
    match GT {eq} = find key r {□R sz key kx x l r eq prf}
    match EQ {_} = x
```

The `error` function in Agda is only a placeholder for translation back to Haskell. The function requires an implicit argument of the bottom type, '⊥', which does not have a constructor. Since this function can only be invoked from an absurd pattern clause, the function is unreachable during runtime.

Here `` `key ∈ map' `` is a type that can only be instantiated when the ∈ function doesn't return the ⊥ type. A key can never be present in the leaf nodes (`Tip`) hence `key ∈ tip` will always return the type ⊥, an element of this type cannot be instantiated by Agda making the case unreachable. This allows us to add the placeholder error statement.

3.4 Non-termination warnings

Secondly, since we cannot solve the Halting problem to ensure termination of functions, the Agda type-checker is sound but not complete, i.e. it might flag valid programs as non-terminating. This can usually be fixed by re-writing the function such that it is clear to Agda that at least one parameter in the recursive call is smaller than the original input to the function. For example :

```
{- Termination checking fails for this impl. -}
insertMax : {k a : Set} → k → a → Map k a → Map k a
insertMax kx x t = case t of
  λ {
    Tip → singleton kx x
  ; (Bin _ ky y l r) → balanceR ky y l (insertMax kx x r)
  }

{- pattern matching directly convinces Agda
  that the recursive call uses a reduced input -}
insertMax : k → a → Map k a → Map k a
insertMax kx x Tip = singleton kx x
insertMax kx x (Bin _ ky y l r) = balanceR ky y l (insertMax kx x r)
```

3.5 Type-class instances

Type-classes in Haskell serve a similar purpose as 'interfaces' in OOP languages. They define a list of function signatures, which must be defined for an instance of the type-class [11]. Agda2hs supports Haskell type-classes through record types in Agda. The Map library defines instances of type-classes such Eq, Functor, Monoid, Foldable etc. and these were successfully implemented in agda2hs.

```
instance
  iSemigroupMap : {k a : Set} → {{ _ : Ord k }} {{ _ : Eq a }}
    {{ _ : Eq (Map k a) }} → Semigroup (Map k a)
  iSemigroupMap ._<>_ = union

  iMonoidMap : {k a : Set} → {{ _ : Ord k }} {{ _ : Eq a }}
    {{ _ : Eq (Map k a) }} → Monoid (Map k a)
  iMonoidMap .mempty = empty

  iFoldableMap : {k : Set} {{ _ : Ord k }} → Foldable (Map k)
  iFoldableMap .foldMap f Tip = mempty
  iFoldableMap .foldMap f (Bin _ _ v l r) =
    mappend (foldMap f l) (mappend (f v) (foldMap f r))

  ...
```

4 Limitations and Upgrades

This section will discuss limitations faced while programming using the current Agda subset defined by agda2hs. It will also talk about Haskell constructs and extensions which are currently not supported by agda2hs. Added support for these can make implementations in agda2hs closer to how the functions should be implemented in native Haskell.

Functions using the *Boolean-guard* construct of Haskell need to be re-written using `if_then_else_` functions. As Agda currently does not have a construct equivalent to Boolean-guards, the subset language cannot be extended to support this feature.

```
{- original haskell program -}
link :: k -> a -> Map k a -> Map k a -> Map k a
link kx x Tip r  = insertMin kx x r
link kx x l Tip  = insertMax kx x l
link kx x l@(Bin sizeL ky y ly ry) r@(Bin sizeR kz z lz rz)
  | delta*sizeL < sizeR  = balanceL kz z (link kx x l lz) rz
  | delta*sizeR < sizeL  = balanceR ky y ly (link kx x ry r)
  | otherwise            = bin kx x l r

{- agda2hs equivalent -}
link : {k a : Set} → k → a → Map k a → Map k a → Map k a
link kx x Tip r  = insertMin kx x r
link kx x l Tip  = insertMax kx x l
link kx x l@(Bin sizeL ky y ly ry) r@(Bin sizeR kz z lz rz) =
    if (delta * sizeL < sizeR)
    then (balanceL kz z (link kx x l lz) rz)
    else (if (delta * sizeR < sizeL)
          then (balanceR ky y ly (link kx x ry r))
          else (bin kx x l r))
```

With regards to the `case_of_` clause, a useful addition to the language would be support for translating `case_of_` clauses with the inspect idiom, which can be helpful in intrinsic verification techniques, to native Haskell case matching . Similar functionality should also be added to the `if_then_else_` clause.

```
{- case_of_ with inspect -}
case'_of_ : {A B : Set} → (a : A) → ((a' : A) → {eq : a ≡ a'} → B) → B
case' x of f = f x {refl}

{- if_then_else with inspect -}
if'_then_else_ : {A : Set} → (flg : Bool) → ({flg ≡ true} → A) → ({flg ≡ false} → A) → A
if' false then t else f = t {refl}
if' true  then t else f = t {refl}

{- use case -}
find : {k a : Set} {{_ : Ord k}} → (key : k) (map : Map k a) → {key □ map} → a
find key Tip = error "Map.!: given key is not an element in the map"
find key t@(Bin sz kx x l r) {prf} = case' compare key kx of
  λ {
    LT {eq} → find key l {□L sz key kx x l r eq prf}
  ; GT {eq} → find key r {□R sz key kx x l r eq prf}
  ; EQ {eq} → x
  }
```

Another improvement in this same construct would be to allow `where' blocks to appear in the same clause as a `case_of_' function.

```
{- with current agda2hs support -}
lookupLE : {k a : Set} {{_ : Ord k}} → k → Map k a → Maybe (k × a)
lookupLE {k} {a} = goNothing
  where
    goJust : k → k → a → Map k a → Maybe (k × a)
    goJust _ kx' x' Tip                  = Just (kx' , x')
    goJust key kx' x' (Bin _ kx x l r) = match (compare key kx)
      where {- match function should be a case_of_ statement -}
        match : Ordering → Maybe (k × a)
        match LT = goJust key kx' x' l
        match GT = Just (kx , x)
        match EQ = goJust key kx x r

    goNothing : k → Map k a → Maybe (k × a)
    goNothing _ Tip                  = Nothing
    goNothing key (Bin _ kx x l r) = match (compare key kx)
      where {- match function should be a case_of_ statement -}
        match : Ordering → Maybe (k × a)
        match LT = lookupLE key l
        match GT = Just (kx , x)
        match EQ = goJust key kx x r
```

4.1 Type-class methods

Due to the current implementation of type-classes and support for minimal complete definitions, it is not possible to override the default implementations of other methods in the type-class. Such as the `foldl` and `foldr` methods in the `iFoldableMap` instance cannot be overridden to the efficient implementations provided in the library, and must rely on the default definition derived from the `foldMap` function.

4.2 BangPatterns

The Data.Map library employs considerable use of the Haskell ‘BangPatterns‘ extension to control strictness of evaluation in the program. BangPatterns can be used to force strict evaluation of parameters during pattern matching. Since Agda currently does not provide support for this extension, it is also not supported in agda2hs.

4.3 Module exports

The considerable size of the Data.Map library, 200 functions, can cause the Agda type-checker to take around a minute when type-checking the complete implementation. It makes sense then to split the library into separate modules limiting the scope of the type-checker during interactive Agda development. This is an essential feature for development of large libraries, and Haskell and Agda both provide support for re-exporting functions defined in separate modules from a single combining module. Currently agda2hs does not support translating between these two representations. This functionality is however already on the pending features list of the agda2hs framework, and should be implemented before its first official release.

5 Verification

Once the library has been implemented in the subset of Agda, it is possible to verify various properties and to assert invariants about the data-structure used. For verification the complete functionality of Agda can be used, and one is not limited to the common subset with Haskell, since the proofs are utilised only during type checking the library and are erased/ignored during translation of the program to Haskell.

5.1 Preconditions

As stated in section 3.3, a precondition should a proposition which is false under the cases where an error is thrown. This will lead Agda to an absurd pattern in that particular case of the inputs, and a placeholder error statement can be written. Therefore pre-conditions are also a form of verification. The pre-condition should sufficiently limit the input space of the function, leaving out all the possible inputs which would have originally lead to the error statement. Often the pre-condition needs to be transformed to fit recursive calls to the same function or calls to other functions.

```
find : {k a : Set} {{_ : Ord k}} → (key : k) → (map : Map k a)
        → {prf : key □ map} → a
find key Tip = error "Given key is not an element in the map"
find key (Bin sz kx x l r) {prf} = match (compare key kx) {refl}
  where
    match : (o : Ordering) → {eq : compare key kx ≡ o} → a
    match LT {eq} = find key l {□L sz key kx x l r eq prf}
    match GT {eq} = find key r {□R sz key kx x l r eq prf}
    match EQ {_} = x

□L : {k a : Set} {{_ : Ord k}} →
    → (sz : Nat) (key kx : k) (x : a) (l r : Map k a)
    → (eq : compare key kx ≡ LT)
    → (prf : key □ (Bin sz kx x l r))
    → (key □ l)
□L sz key kx x l r eq prf with (compare key kx)
... | LT = prf
```

Considering the same example as section 3.3, the pre-condition `key` in `map` is sufficient to eliminate the input `Tip` as being a valid element of type `Map k a` to this function as it always leads to a false proposition. And in the case of `Bin` Agda provides a proof of this proposition by instantiating a value of the type `key` in `map`. For the recursive calls `find key l` and `find key r`, the programmer now needs to provide a proof of the propositions `key` in `l` and `key` in `r`. This can be accomplished through the functions ∈L and ∈R respectively. In inL function makes use of the with-abstraction clause of Agda. The value `eq` limits the cases of `compare key kx` to just `LT` and the with-abstraction adds this additional information to Agda's unification process transforming the proof of `key` in `map` to `key ∈ l`.

Pre-conditions were also used for the functions 'elemAt', 'updateAt' and 'deleteAt'. Here the pre-condition limits the index parameter to the valid range of indices that is from 0 to the size of the Map. Similar functions were used to then transform the proof of this pre-condition to the proof needed for the recursive calls in the function.

```
elemAt : ... (n : Nat) → (map : Map k a) → {(n < size map) ≡ true} → k × a
updateAt : ... (k → a → Maybe a) → (n : Nat) → (map : Map k a)
    → {(n < size map) ≡ true} → Map k a
deleteAt : ... (n : Nat) → (map : Map k a) → {(n < size map) ≡ true} → Map k a
```

5.2 Type-class laws

As seen in section 3.5, the Data.Map library defines instances of a few type-classes. In Haskell, type-classes usually include a list of properties that the function definitions must satisfy, for example an instance of the Eq type-class must have the definition of the '_==_' function satisfying the properties of reflexivity, transitivity and symmetry.

For the Data.Map library, properties of the Functor, Monoid, Foldable, Traversable classes etc. were verified. Some properties such as the identity and composition properties of Functor were fairly easy to verify.

```
functorIdentityMap : {k a : Set} {{_ : Ord k}} → (m : Map k a) → fmap id m ≡ id m
functorIdentityMap Tip = refl
functorIdentityMap (Bin sz kx x l r) =
  begin
    fmap id (Bin sz kx x l r)
  ≡⟨ cong (λ t →  Bin sz kx x t (map id r)) (functorIdentityMap l) ⟩
    Bin sz kx x l (map id r)
  ≡⟨ cong (λ t →  Bin sz kx x l t) (functorIdentityMap r) ⟩
    Bin sz kx x l r
  ∎

functorCompositionMap : {k a : Set} {{_ : Ord k}} → (f : b → c) (g : a → b) (m : Map k a)
                    → fmap (f ∘ g) m ≡ (fmap f ∘ fmap g) m
functorCompositionMap f g Tip = refl
functorCompositionMap f g m@(Bin sz kx x l r) = begin
    fmap (f ∘ g) (Bin sz kx x l r)
  ≡⟨⟩ { application of fmap }
    Bin sz kx ((f ∘ g) x) (map (f ∘ g) l) (map (f ∘ g) r)
  ≡⟨ cong (λ t →  Bin sz kx ((f ∘ g) x) t (map (f ∘ g) r))
      (functorCompositionMap f g l) ⟩
    Bin sz kx ((f ∘ g) x) (map f (map g l)) (map (f ∘ g) r)
  ≡⟨ cong (λ t →  Bin sz kx ((f ∘ g) x) (map f (map g l)) t)
      (functorCompositionMap f g r) ⟩
    Bin sz kx (f (g x)) (map f (map g l)) (map f (map g r))
  ≡⟨⟩ { un-application of fmap }
    fmap f (fmap g m)
  ∎
```

The process usually follows a similar pattern. The input parameters are case-split into their different constructors and the base case (smallest case) can usually be instantiated through `refl` — Agda can automatically infer the equality by unifying the constraints using the definitions of the functions involved. The recursive case can be re-written through a few applications of the functions involved. Once a recursive call to the same function has been reached, an inductive hypothesis about the property in question can be applied on this function with the reduced inputs.

A similar process was used for the following properties —

```
monoidRightIdentityMap : ... → (m : Map k a) → m <> mempty ≡ m
monoidLeftIdentityMap : ... → (m : Map k a) → mempty <> m ≡ m
monoidConcatenationMap : ... → (ms : List (Map k a)) → mconcat ms ≡ foldr (_<>_) mempty ms

foldableFunctorMap : ... {{_ : Monoid b}} → (f : a → b) → (m : Map k a)
    → foldMap f m ≡ (fold ∘ fmap f) m

traverseNaturalityMap : {k a b : Set} {{_ : Ord k}}
    → {p q : Set → Set} {{ap : Applicative p}} {{aq : Applicative q}}
    → (t : {x : Set} → p x → q x) (preservePure : {A : Set} → (a : A) → t (pure a) ≡ pure a)
    → (preserveApp : {A B : Set} → (g : p (A → B)) (a : p A) → t (g <*> a) ≡ (t g <*> t a))
    → (f : a → p b) → (m : Map k a)
    → (t ∘ traverse f) m ≡ traverse (t ∘ f) m

traversePurityMap : {k a : Set} {{_ : Ord k}}
    → {p : Set → Set} {{ap : Applicative p}}
    → (m : Map k a) → traverse (pure {p}) m ≡ pure m
traverseIdentityMap : {k a : Set} {{_ : Ord k}}
    → (m : Map k a) → traverse IdentityCon m ≡ IdentityCon m

traverseCompositionMap : {k a b c : Set} {{_ : Ord k}}
    → {p q : Set → Set} {{ap : Applicative p}} {{aq : Applicative q}}
    → (g : b → q c) (f : a → p b) (m : Map k a)
    → traverse (ComposeCon ∘ fmap g ∘ f) m ≡ (ComposeCon ∘ fmap (traverse g) ∘ traverse f) m
```

5.2.1 Functions with many case-splits

A major hurdle faced during this verification process was the incompatibility of the efficiency focused function definitions with the verification process. The `union` function, used in the definition of the Semigroup class, uses 5 separate 'overlapping' cases for pattern matching the inputs. Re-writing the pattern matching to obtain mutually exclusive patterns leads to 15 cases.

```
union : {{ Eq a }} → {{ Eq (Map k a) }} → Map k a → Map k a → Map k a
union Tip t2  = t2
union t1 Tip  = t1
union t1 (Bin _ k x Tip Tip) = insertR k x t1
union (Bin _ k x Tip Tip) t2 = insert k x t2
union t1@(Bin _ k1 x1 l1 r1) t2 = case (split k1 t2) of
  λ {
    (l2 , r2) → let l1l2 = union l1 l2
                     r1r2 = union r1 r2
                   in link k1 x1 l1l2 r1r2
  }

semigroupAssociativityMap : {k a : Set} {{_ : Ord k}} {{_ : Eq a}}
          → (m1 m2 m3 : Map k a) → (m1 <> m2) <> m3 ≡ m1 <> (m2 <> m3)
```

Therefore to prove the associativity property of the Semigroup class we would need at least, 15×15 as the associativity property involves two applications of the union function, 225 pattern-matched cases in the definition of `semigroupAssociativityMap`. Consequently this property could not be verified given the time and resource bounds of this project.

5.2.2 Equality of Maps

```
map1 = Bin 2 "b" "val" (singleton "a" "val") (Tip)
map2 = Bin 2 "a" "val" (Tip) (singleton "b" "val")

iEqMap : {k a : Set} {{ _ : Ord k }} {{ _ : Eq a }} → Eq (Map k a)
iEqMap ._==_ t1 t2 = (size t1 == size t2) && (toAscList t1 == toAscList t2)

{-  Substitutivity
    if x == y = True, then f x == f y = True -}

f : {k a : Set} → Map k a → Nat
f Tip = 0
f (Bin _ _ _ l r) = 1 + (f l)
```

The Map datatype uses a weak form of equality which only depends on the data stored in the Map. `map1` and `map2` defined above are different structurally but, as per the definition of the _==_ function, they are equivalent — `map1 == map2 = True`. Functions dependent on the structure of the Map, such as function ‘f’ above violate the law of substitutivity. Hence the Eq instance for Map does not satisfy one of the required properties.

5.3 Intrinsic verification of ordering

The style of proofs discussed in the previous subsection is called external verification, as the proofs are external to the program whose properties they work on. This style of verification is useful for proving algebraic properties such as associativity of an operator or composition of a function. When functions take in proofs of pre-conditions or they are defined with an invariant the internal verification style is more fitting.[19]

A binary search tree has an invariant data-ordering property on it’s nodes — the left sub-tree must contain smaller elements while the right sub-tree contains larger. This property must not be violated in any reasonable use of the data-structure. It makes sense to then enforce this property internally through a more semantically expressive constructor for Map.

```
data [_]∞ (A : Set) : Set where
  -∞  :     [ A ]∞
  [_] : A → [ A ]∞
  +∞  :     [ A ]∞

_≤_ : A → A → Set
x ≤ y = (x <= y) ≡ true

data Map (k : Set) (a : Set) {{ kOrd : Ord k }} {lower upper : [ k ]∞} : Set where
  Bin : (sz : Nat) → (kx : k) → (x : a)
        → (l : Map k a {lower} {[ kx ]}) → (r : Map k a {[ kx ]} {upper})
        → {{_ : sz ≡ (size l) + (size r) + 1}}
        → Map k a {lower} {upper}
  Tip : {{ l≤u : lower ≤ upper }} → Map k a

dict : Map Int String -∞ +∞
dict = Bin 2 9 "nine" (singleton 6 "six") (Tip)

dict2 : Map Int String -∞ +∞
dict2 = Bin 2 9 "nine" (singleton 11 "eleven") (Tip)
```

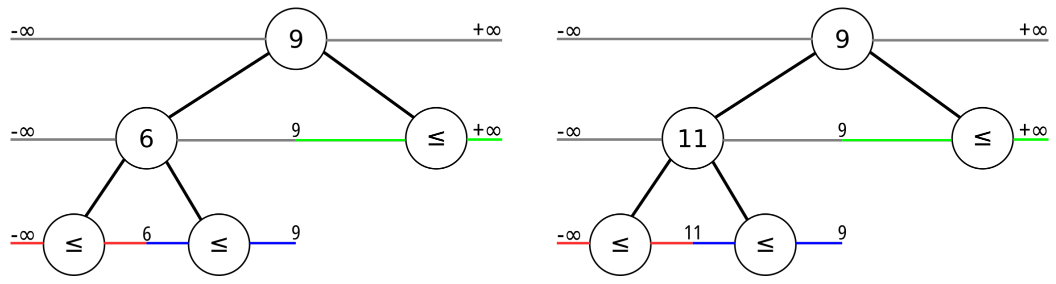

Figure 1: Valid and in-valid BSTs.

Firstly, since a BST need not have bounds on the data-structure as a whole, we define a new datatype `[_]∞`. As can be seen from its definition above, this datatype extends any existing type with the `+∞` and `-∞` elements. An Eq and Ord instance for this new type is also defined. Secondly, for convenience, we define the function '`_≤_`' which allows the re-write of the proposition '`x <= y ≡ true`' as '`x ≤ y`'. Inspiration for these data-types was taken from [8] and [14].

The constructor for the BST representing a Map is now re-defined as above. A Map instance, `dict`, previously of type '`Map Int String`' now becomes '`Map Int String -∞ +∞`' — where (`lower = -∞`) and (`upper = +∞`) represent the bounds on the BST. This doesn't enforce any additional constraint on the elements that can be contained in `dict`, which is desired, but the type of the left sub-tree of `dict` is now '`Map Int String -∞ [9]`' and that of the right sub-tree is '`Map Int String [9] +∞`'. This splitting of bounds continues down till the leaf nodes, `Tip`, where a proof of the proposition '`lower ≤ upper`' is finally required ensuring the validity of the bounds.

For example, consider `dict2` which violates the ordering imposed on the sub-trees of the Map. The left sub-tree of `dict2` should only accept values in the range of `-∞` to `[9]`. Adding the integer 11 to this sub-tree leads to the constraint `11 ≤ 9` on one of the leaf nodes, as can be seen in figure 1. This proposition can never be proved causing the invalidation of `dict2`.

5.3.1 Redefinition of `insert`

Now that the constructor has been defined to include the ordering properties, proofs of these constraints will need to be provided in functions that operate on a Map. For example, inserting an element into the BST is now more involved, as the function needs to do some work to ensure that the ordering properties are always satisfied.

The insert function takes as input a new key '`ky`' and an existing map of type '`Map k a lower upper`'. If the value of `ky` does not fall within the existing bounds then the bounds of the new Map will need to change. Hence the return type of the insert function will be '`Map k a min lower [ky] max upper [ky]`'.

```
compareLT-≤ : {A : Set} {{_ : Ord A}} → (x y : A) → compare x y ≡ LT → x ≤ y
compareGT-≤ : {A : Set} {{_ : Ord A}} → (x y : A) → compare x y ≡ GT → y ≤ x
compareEQ-≡ : {A : Set} {{_ : Ord A}} → (x y : A) → compare x y ≡ EQ → x ≡ y

min-≤1 : ... → (x y : A) → (min x y) ≤ x
min-≤2 : ... → (x y : A) → (min x y) ≤ y
```

```
max-≤1 : ... → (x y : A) → x ≤ (max x y)
max-≤2 : ... → (x y : A) → y ≤ (max x y)

≤-min : ... → (x y : A) → x ≤ y → (min x y) ≡ x
≤-max : ... → (x y : A) → y ≤ x → (max x y) ≡ x
```

We define helper methods which prove a few properties about the _≤_ operator that will come in handy during the definition of insert.

```
insert : {k a : Set} {{_ : Ord k}} {{_ : Eq a}}
  → {lower upper : [ k ]∞}
  → (ky : k) → (y : a) → (m : Map k a {lower} {upper})
  → Map k a {min lower [ ky ]} {max upper [ ky ]}

insert {lower = lower} {upper = upper} ky y Tip =
  singleton ky y {{min-≤2 lower [ ky ]}} {{max-≤2 upper [ ky ]}}

insert {k} {a} {lower} {upper} ky y (Bin sz kx x l r) = match (compare ky kx) {refl}
  where
    match : (o : Ordering) → {eq : compare ky kx ≡ o}
          → Map k a {min lower [ ky ]} {max upper [ ky ]}
    match LT {eq} = helper
                    {≤-max [ kx ] [ ky ] ([]-≤-I {{x≤y = compareLT-≤ ky kx eq}})}
                    {≤-max upper [ ky ] (≤-trans {[ k ]∞} {z = upper}
                      ([]-≤-I {{x≤y = compareLT-≤ ky kx eq}}) _)}
      where
        l' = insert ky y l
        helper : {max [ kx ] [ ky ] ≡ [ kx ]} → {max upper [ ky ] ≡ upper}
               → Map k a {min lower [ ky ]} {max upper [ ky ]}
        helper {prf1} {prf2} = balance {l1 = min lower [ ky ]} {u1 = [ kx ]}
                               {l2 = [ kx ]} {u2 = upper}
                               kx x {{≤-refl {x = [ kx ]}}} {{≤-refl {x = [ kx ]}}} l' r
    match GT {eq} = helper
                    {≤-min lower [ ky ] (≤-trans {[ k ]∞} {x = lower} _
                      ([]-≤-I {{x≤y = compareGT-≤ ky kx eq }}))}
                    {≤-min [ kx ] [ ky ] ([]-≤-I {{x≤y = compareGT-≤ ky kx eq }})}
      where
        r' = insert ky y r
        helper : {min lower [ ky ] ≡ lower} → {min [ kx ] [ ky ] ≡ [ kx ]}
               → Map k a {min lower [ ky ]} {max upper [ ky ]}
        helper {prf1} {prf2} = balance {l1 = lower} {u1 = [ kx ]}
                               {l2 = [ kx ]} {u2 = max upper [ ky ]}
                               kx x {{≤-refl {x = [ kx ]}}} {{≤-refl {x = [ kx ]}}} l r'
    match EQ {eq} = helper {compareEQ-≡ ky kx eq}
      where
        helper : { ky ≡ kx } → Map k a {min lower [ ky ]} {max upper [ ky ]}
        helper {prf} = Bin sz ky y l r
```

The first case of insertion into an empty Map is fairly simple where a singleton BST is constructed. The above mentioned properties are used to prove to Agda that the new element ‘ky’ satisfies the bounds (`min lower [ky]`) $\leq$ [ky] and [ky] $\leq$ (`max upper [ky]`).

The second case of insertion into an existing BST is a bit more involved. The new key ‘ky’ is compared to the existing key in the current node leading to one of three cases - insertion continues in the left sub-tree, the right sub-tree, or the insertion is done at the current node.

If the new key is smaller than the existing key, a recursive call to insert is created with the left sub-tree as the input. The return value of this call, `l'`, is a Map of type ‘`Map k a min` lower [ky] max [kx] [ky]'. The new left sub-tree, l', is passed onto the function balance along with the existing right sub-tree and current node values.

```
balance : ∀ {l1 u1 l2 u2 : [ k ]∞} → (kx : k) → a
          → {{u1≤k : u1 ≤ [ kx ]}} {{k≤l2 : [ kx ] ≤ l2}}
          → Map k a {l1} {u1} → Map k a {l2} {u2}
          → Map k a {l1} {u2}
```

The function balance takes as input a de-structured BST, i.e. values for the root node and two BSTs where one contains values smaller than the other, and performs rotations to re-balance the tree.

Once again, the above mentioned properties about _≤_ have to be utilised to convince Agda that — given 'compare ky kx ≡ LT' the propositions 'max [kx] [ky] ≡ [kx]' and 'max upper [ky] ≡ upper' hold. Since the with-abstraction construct of Agda cannot be used in parts of the program that will be translated to Haskell, the existence of these properties can be indicated to the Agda type-checker through the use of locally defined functions with implicit parameters.

Intrinsic verification requires considerable use of implicit and instance parameters in data constructors and functions, as these arguments are wiped during translation of the program. During the course of the project this support was added to agda2hs[7]. Although from the above example we can see intrinsic verification can still take a lot of additional effort.

Providing a basic set of proofs about properties involving functions and data-types from the Haskell prelude can aid in the verification process. Improved 'automatic theorem provers' for Agda [13][10] can also help developers by automatically filling in the simple but tedious parts in the interactive verification process.

6 Responsible Research

The project mostly deals with the workings and usage of the 'agda2hs' framework. The research performed is mostly empirical observations about the functionality and limitations of the aforementioned framework. The main ethical concern of in the project is of reproducibility of the code and implementations off which the observations are made. All the code utilised in the implementation and verification of the library is made available to the public, free to use, modify and build upon. The Agda language and 'agda2hs' framework are also available as an open source projects for users interested in the workings of dependent types and the Agda type-checker. No other concerns with the workings of the project has been identified concerning the ethics of the process.

7 Conclusions and Future Work

The Curry-Howard isomorphism explains the correspondence between mathematical proofs and type systems. *agda2hs* is a framework that attempts to bring the strength of Agda as a proof-assistant to the vast ecosystem of Haskell programs. It designates a subset of Agda that can be dictionary translated to Haskell. This allows any Haskell program to be written equivalently in Agda, and then be verified.

The aim of the project was to empirically test the viability of the *agda2hs* framework in bringing proof based verification to the Haskell ecosystem. To this end an existing Haskell library, Data.Map, was chosen as a representation of Haskell programs that might benefit from use of this framework. By re-writing this library in agda2hs, it was shown that the common-subset is complete enough to accommodate most Haskell programs. Properties of functions contained in this library were also verified confirming the feasibility of this approach.

During the course of the project, various features were also recommended as additions to the framework to increase its functionality and improve its use for verification. Bugs in the current implementation of the framework were also identified during this process. Some of the issue identified were limited support for case matching clauses, inability to override default implementations of type-class methods, no support for module export lists etc.

The outcome of this project is also a verified version of the Data.Map library available to users of Agda and agda2hs. No bugs were found in the library itself as it's already thoroughly tested. However, future projects that seek to verify further properties of the library and/or programs that build upon this library can now utilise these proofs in their own formal verifications.

Considerations for future work on verification of Haskell programs and the *agda2hs* framework were also noted, namely :

- Classical testing methods are not as affected by the implementation details of a program, as they mostly rely on the intended effects of a function, which are measured using a test suite of various inputs and their respective outputs. However, verification based testing is intrinsically linked to the exact implementation of a function. Verifying properties of functions which have been heavily optimised to reduce duplicate evaluation can be a challenge. These implementations forgo the simplest possible implementation for a particular program and instead adopt one that tries to manually overcome the inherent inefficiency of Haskell, consequently adding additional steps to the verification process as well.

- The agda2hs framework can currently produce equivalent Haskell code from the programs written in its subset of Agda. Therefore there might also be demand for a supplemental framework which is able to translate existing Haskell code to its equivalent Agda form. This step was performed manually in the course of this project, but future attempts to verify other existing Haskell programs could benefit from automation of this step. This functionality is present in the similar framework *'hs-to-coq'*, whereby it can translate existing Haskell programs to the proof-assistant language Coq. Though this framework does not support translating the program from a subset of Coq to Haskell.

References

[1] Agda. `https://github.com/agda/agda`.

[2] Agda2hs. `https://github.com/agda/agda2hs`.

[3] containers : Data.map. `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Map-Internal.html`.

[4] The coq proof assistant. `https://coq.inria.fr/`.

[5] Haskell language. `https://www.haskell.org/`.

[6] Liquidhaskell, static verifier for haskell. `https://ucsd-progsys.github.io/liquidhaskell-blog/`.

[7] agda2hs contributors. Support for implicit and instance arguments in data constructors. https://github.com/agda/agda2hs/commit/1b350f85ed02f44dd24c07bf7f147055cac6da2e, 2021.

[8] Jesper Cockx. Formalize all the things (in agda). https://jesper.sikanda.be/posts/formalize-all-the-things.html, 10 2019.

[9] Jesper Cockx. Correspondences between logic systems and type systems in agda. `https://github.com/jespercockx/agda-lecture-notes/blob/master/agda.pdf`, 2021. Table 2.

[10] Simon Foster and Georg Struth. Integrating an automated theorem prover into agda. In *NASA Formal Methods Symposium*, pages 116–130. Springer, 2011.

[11] Paul Hudak, John Peterson, Joseph Fasel, and Reuben Thomas. Gentle introduction to haskell, version 98. https://www.haskell.org/tutorial/classes.html, 06 2000.

[12] G. Hutton. *Programming in Haskell*, chapter 16, pages 228–246. Programming in Haskell. Cambridge University Press, 2016.

[13] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In *International Workshop on Types for Proofs and Programs*, pages 154–169. Springer, 2004.

[14] Conor Thomas McBride. How to keep your neighbours in order. *ACM SIGPLAN Notices*, 49(9):297–309, 2014.

[15] Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

[16] Liam O'Connor-Davis and Willian DeMeo. Learn you an agda. https://williamdemeo.github.io/2014/02/27/learn-you-an-agda/#why-prove-when-you-can-just-test, 02 2014.

[17] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in agda. *Proceedings of the 7th workshop on Programming languages meets program verification - PLPV '13*, 2013.

[18] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Aaron Stump. *Verified functional programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016.

[20] Niki Vazou, Eric L Seidel, and Ranjit Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 39–51, 2014.

[21] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020.

[22] Wikibooks. Haskell/The Curry-Howard isomorphism — Wikibooks, the free textbook project, 2016. [Online; accessed 23-April-2021].

[23] Wikipedia contributors. Curry–howard correspondence — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Curry%E2%80%93Howard_correspondence&oldid=1029985989`, 2021. [Online; accessed 26-June-2021].