Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

Producing a verified implementation of sequences using agda2hs

Shashank Anand[1], Jesper Cockx[1], Lucas Escot[1]

[1]TU Delft

Abstract

Purely functional languages are advantageous in that it is easy to reason about the correctness of functions. Dependently typed programming languages such as Agda enable us to prove properties in the language itself. However, dependently typed programming languages have a steep learning curve and are usable only by expert programmers. The agda2hs project identifies a common subset of Agda and Haskell and translates this subset from Agda to readable Haskell, which enables programmers to write and verify code in Agda, and use it in Haskell. This provides the guarantees of verification to the translated code in Haskell, a language that does not support verification. The objective of this paper is to investigate the possibility of producing a verified implementation of the Haskell library `Data.Sequence` using the common subset identified by agda2hs. A description of the implementation and verification of `Data.Sequence` in Agda is provided. A proof technique using arbitrarily nested types is examined in detail. Possible reductions to the cost of the verification process are discussed.

1 Introduction

Haskell[1] is a strongly typed purely functional programming language. One of the big advantages of purity is that it allows reasoning about the correctness of algorithms and data structures. This is demonstrated for example in Chapter 16 of the book Programming in Haskell (2nd edition) by Graham Hutton[2], where he uses equational reasoning, case analysis, and induction to prove properties of Haskell functions. However, since these proofs are done ‘on paper’, there is always a risk that the proof contains a mistake, or that the code changes to a new version but the proof is not updated.

In a dependently typed programming language such as Agda[3], it is possible to actually write a formal proof of correctness in the language itself[4][5]. This proof is checked automatically by the typechecker, and re-checked every time the code is changed. So it provides a high degree of confidence in the correctness of the algorithm. Unfortunately, despite several big successes in dependently typed programming such as the CompCert verified C compiler[6], these languages remain hard to use. So these guarantees are currently only available to expert users.

The agda2hs project[7] is a recent effort to combine the best parts of Haskell and Agda. In particular, it identifies a common subset of the two languages, and provides a faithful translation of this subset from Agda to Haskell. This allows library developers to implement libraries in Agda and verify their correctness, and then translate the result to Haskell so it can be used and understood by Haskell programmers.[1] However, agda2hs does not completely identify the common subset of Haskell and Agda, and the implications of the missing subset on the expressive ability of Agda is unknown.

Sequence[8] is a Haskell library that represents lists internally as finger trees[9]. Sequences are more efficient than traditional lists for a variety of operations such as concatenation, splitting, index access, etc.

The objective of the paper is to discuss the implementation of a verified implementation of the Sequence library using the common subset identified by agda2hs. Particularly, the following questions are answered.

1. Does the implementation of this library fall 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?

2. What kind of properties and invariants does the library guarantee? Do the functions in the library require certain preconditions? Are there any ways the internal invariants of the library can be violated?

3. Is it possible to formally state and prove the correctness of Haskell libraries that are ported to Agda? How much time and effort does it take to verify the implementation compared to implementing the algorithm itself? What kind of simplifications could be made to reduce the cost of verification?

Section 2 discusses related work in verifying Haskell code, and Section 3 follows with a recap of the important concepts used in the paper. Section 4 describes the implementation in Agda and section 5 follows with an examination of the verification process. The ethical implications of this paper are discussed in section 6. Section 7 describes the results and discusses some possible reductions to the cost of verification, and section 8 concludes with future extensions.

2 Related Work

2.1 hs-to-coq

One of the most important contributions relevant to our work is the hs-to-coq project[10]. hs-to-coq is a tool that translates Haskell code to Coq[11]. A major difference between hs-to-coq and agda2hs is that the former relies on first coding in Haskell, and then translating it to a language with support for verification to prove properties about the code. In agda2hs, verification can be performed alongside the development process. This is useful as testing alongside the development process is very effective[12].
hs-to-coq has been used to verify the Haskell containers library[13]. The variants and properties of the containers library have been identified via type class laws, properties stated in the comments of the Haskell implementation, invariants guaranteed by the data types,

[1]Project description as provided in the project Practical Verification of Functional Libraries on Project Forum.

and QuickCheck[14][15] properties. These methods are very similar to those described in this paper.

2.2 Liquid Haskell

Liquid Haskell[16] enables verification in Haskell by allowing developers to specify properties using refinement types. It enables support for totality, termination checking, and equational reasoning, bringing many features of dependently typed languages to Haskell. However, it does not provide support for dependent types. While Liquid Haskell has been shown to be effective for verification, dependently typed languages such as Coq provide advantages in the form of a large pool of developed theorems, tactics, and proof methodologies[17].

3 Background

3.1 Finger tree

Finger trees are purely functional data structures. They provide constant time access to the 'fingers' (leaves) of the tree. The number of elements in a finger increases exponentially as we recursively go down the sub-trees. This facilitates logarithmic time access to any finger in a tree[9]. An illustration of a finger tree can be seen in Figure 1. The implementation is described in section 4.1.

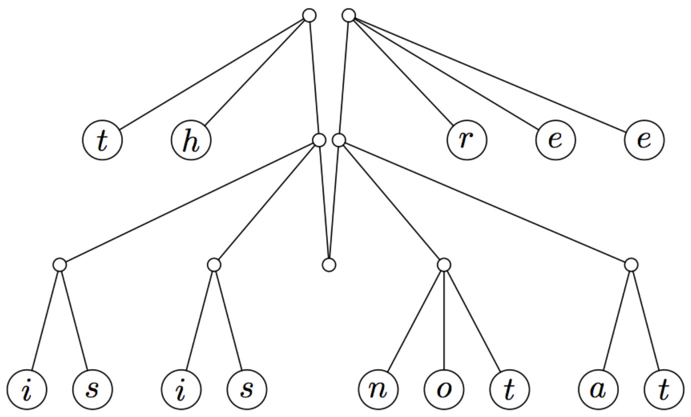

Figure 1: Finger Tree[18]. Note that the equivalent string would be 'thisisnotatree'

3.2 Sequence

Sequence is a Haskell library that provides methods to create and operate on lists that are internally represented as finger trees[9]. Sequences support operations on lists that are more efficient than on traditional lists. Most notably they are :

- Constant-time access to both the front and the rear end of the list.

- Logarithmic-time concatenation
- Logarithmic-time splitting, take, drop.
- Logarithmic-time random access

However, the finger tree implementation leads to reduced efficiency of some operations. Most notably, prepending an element to a finger tree has worst case time complexity $\mathcal{O}(\log n)$ Sequences are also typically slower than lists for operations with the same time complexity[8].

3.3 Curry-Howard correspondence

The Curry-Howard correspondence or the Curry-Howard isomorphism is the direct relationship that exists between mathematical proofs and type systems in programming languages[19]. It is useful for representing propositional logic as types in code. The Curry-Howard correspondence can be used to infer the relationship between propositional logic and types as presented in table 1.

Propositional logic		**Type system**
proposition	P	type
proof of proposition	p : P	program of type
implication	P → 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 1: Curry-Howard correspondence between propositional logic and types in Agda[20].

3.4 Agda basics

We list and briefly describe some basic Agda concepts used widely throughout the paper.

1. **Universes and Set**[2] : A type whose elements are types is called a universe. Agda provides an infinite number of universes. The first universe level is Set.
2. **Indexed data types**[3] : Data types in Agda can have indices. Indices can vary from constructor to constructor. Indexed data types are used to represent dependent types.
3. **Implicit arguments**[4] : Implcit arguments are arguments that the type checker can figure out by itself. These are supplied within {}. For example, the id function has the type of the parameter as an implicit argument as the first argument can be inferred from the second argument.

```
id : {A : Set} → A → A
```

[2]https://agda.readthedocs.io/en/latest/language/universe-levels.html
[3]https://agda.readthedocs.io/en/latest/language/data-types.html
[4]https://agda.readthedocs.io/en/latest/language/implicit-arguments.html

3.5 Dependent types

Dependent types are types that depend on a value of a type[21]. Dependent types are widely used to prove properties/encode proofs in dependently typed programming languages. Two important dependent types used in this paper are `IsTrue` and ≡.

`IsTrue` is defined as follows. It is available in the agda2hs prelude[5].

```
data IsTrue : Bool -> Set where
  instance itsTrue : IsTrue true
```

`IsTrue p` can be used in the type of a function to ensure that the function only accepts inputs for which the predicate `p` holds. As `IsTrue false` is an empty type, the function is not defined for inputs for which the predicate does not hold.

The type ≡ is defined as follows. It is provided as part of the module Equality[22].

```
data _≡_ {A : Set} : (x y : A) -> Set where
  refl : (x : A) -> x ≡ x
```

A ≡ B can be used to assert that A is strongly equivalent to B. This dependent type is used in this paper to prove properties on sequences. This can be achieved by defining and implementing functions that return a value of type A ≡ B

3.6 Totality of functions in Agda

Total functions are functions that are defined for all values over the domain of the input, never throw an exception, and always terminate and return a value[23]. In agda, all functions are required to be total[24].
For example, take the following definition of head of a list.

```
head : {a : Set} -> List a -> a
head (x :: xs) = x
```

This definition is invalid because the case `head []` is missing. However, it is impossible to get the head of an empty list. Therefore, this function can only be defined if the given list is non-empty. Thus, it is necessary to limit the domain of the function by supplying a proof that the list is non-empty as an implicit argument. This can be done using the `IsTrue` dependent type described in section 3.2.

```
head : {a : Set} -> (xs : List a) -> {IsTrue (isNonEmpty xs)} -> a
head (x :: xs) = x
```

As `IsTrue false` is an empty type, the function only accepts non-empty lists, thus making a non-total function total.

[5]https://github.com/agda/agda2hs/blob/master/lib/Haskell/Prim.agda

4 Implementation

This section describes the implementation of the Sequence library in Agda. A general overview of the implementation is provided, and the challenges faced in adapting the library for implementation in Agda are discussed. The implementation described in this paper is available in the public domain [6].

4.1 Sequence ADT

The sequence algebraic data type is defined as a finger tree of elements. The Elem type is used as a wrapper around a polymorphic type to implement the Sized type class as described in section 4.2.1. The Seq data type is described as follows.

```
data Seq (a : Set) : Set where
  Sequence : (xs : FingerTree (Elem a))  -> Seq a

data FingerTree (a : Set) : Set where
  EmptyT  : FingerTree a
  Single : a -> FingerTree a
  Deep   : Int -> Digit a -> FingerTree (Node a) -> Digit a -> FingerTree a

data Node (a : Set) : Set where
  Node2 : Int -> a -> a -> Node a
  Node3 : Int -> a -> a -> a -> Node a

data Digit (a : Set) : Set where
  One   : a -> Digit a
  Two   : a -> a -> Digit a
  Three : a -> a -> a -> Digit a
  Four  : a -> a -> a -> a -> Digit a

data Elem (a : Set) : Set where
  Element : a -> Elem a
```

4.2 Type class instances

Ad hoc polymorphism, also known as overloading, is a type of polymorphism in which polymorphic functions can be applied to arguments of different types. Type classes are type system constructs that enable ad-hoc polymorphism[25]. Type class instances can be defined in Haskell using the `instance` keyword[26]. Type classes are important as they provide a structured way to control ad hoc polymorphism.

The sequence library has instances declared for the type classes as seen in table 2.

[6] https://github.com/Sh-Anand/practical-verification-of-sequences-

Type class	Minimal complete definition(s)
Foldable	foldMap
Functor	fmap
Traversable	traverse
Semigroup	<>
Monoid	mempty
Applicative	pure || $< * >$
Monad	»=

Table 2: Type classes implemented by Sequence

The instances for Foldable, Functor, and Traversable are defined similar to each other. foldMap, fmap, and traverse requires a traversal of the entire tree in the order from start to end. To ensure that the functions are applied in order of the list, as seen in Figure 1, the functions generate outputs by first applying the function on the left finger, then recursing on the sub-tree, and then finally applying the function to the right finger. For the sake of brevity, only the functor instances for Sequence and FingerTree are presented.

```
instance
  SeqFunctor : Functor Seq
  SeqFunctor .fmap f (Sequence xs) = Sequence ((fmap (fmap f) xs))

instance
  FingerTreeFunctor : Functor FingerTree
  FingerTreeFunctor .fmap _ EmptyT = EmptyT
  FingerTreeFunctor .fmap f (Single a) = Single (f a)
  FingerTreeFunctor .fmap f (Deep v pre tree suf) =
                      Deep v (fmap f pre) (fmap (fmap f) tree) (fmap f suf)
```

4.2.1 Sized

Types that implement the Sized type class must define the size function. For Finger Trees, Nodes, Digits and Elems, size can be obtained in $\mathcal{O}(1)$ time. The size of values of the Elem type is always 1, which ensures that the size of a sequence is equivalent to its corresponding list. Thus, the polymorphic type does not have to implement Sized when Elem is used as a wrapper around it. For the sake of brevity, only the FingerTree and Elem instances for Sized are provided.

```
record Sized (a : Set) : Set where
  field
    size : a -> Int

instance
  FingerTreeSized : {{Sized a}} -> Sized (FingerTree a)
  FingerTreeSized .size EmptyT = 0
  FingerTreeSized .size (Single a) = size a
  FingerTreeSized .size (Deep v pr m sf) = v

instance
  ElemSized : Sized (Elem a)
  ElemSized .size _ = 1
```

However, as observed from the instances of Sized for FingerTree, it is possible to instantiate a FingerTree of invalid size using the `Deep` constructor. This can be solved by not exporting the internal definitions out of the module to prevent users from constructing illegal FingerTrees. We also describe proofs on properties of Sized in section 5.4 to ensure that functions modifying the structure of FingerTrees produce valid FingerTrees.

4.3 Challenges

Several obstacles were faced in the process of porting the sequence library to Agda. To overcome these issues, several functions had their definitions tweaked or changed.

4.3.1 Totality

As described in section 3.4, functions in Agda must be total. For instance, take the function `scanl1` in Sequence.

```
scanl1 : {a : Set} -> (a -> a -> a) -> Seq a -> Seq a
```

The `scanl1` function throws an error when the sequence is empty. However, since functions are required to be total, the domain of the function must be shrunk. Thus, the definition of `scanl1` is modified to only take non empty sequences as inputs.

```
isNotEmptySequence : {a : Set} -> Seq a -> Bool
isNotEmptySequence (Sequence EmptyT) = false
isNotEmptySequence _ = true

scanl1 : {a : Set} -> (a -> a -> a) -> (xs : Seq a)
         -> {IsTrue (isNotEmptySequence xs)} -> Seq a
```

4.3.2 Throwing Errors

In section 4.3.1, it was discussed that functions that are not defined (or throw errors) for certain inputs must have their domain restricted to be total. However, it may be preferable sometimes to throw an error in Haskell. For this purpose, the error function is defined in the agda2hs prelude. The agda2hs compiler converts this to the error function in Haskell.

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

In the above definition, `@(tactic absurd)` makes sure that the unreachable cases are automatically resolved by the `absurd` tactic. Thus, the error function can only be used when a case is impossible. That is, it can only be used for cases that have been restricted implicitly. For instance, the lookup function presented in section 4.3.1 can be modified to throw an error for the empty sequence case.

```
lookup : {a : Set} -> Int -> (xs : Seq a)
        -> {IsTrue (isNotEmptySequence xs)} -> Maybe a
lookup i (Sequence EmptyT) = error "undefined"
```

4.3.3 Termination checker

Agda only accepts functions that can be proven to be terminating by the termination checker[27]. A recursive function passes the termination check only if it recurses on a subexpression of the argument(s), while provably terminating functions can fail the termination check. However, Agda provides a way out in the form of the TERMINATING pragma.
For example, the following definition of cycleTaking fails the termination check.

```
cycleTaking : {a : Set} -> {{Sized a}} -> (n : Int) -> Seq a -> Seq a
cycleTaking n xs = if n < (size xs)
                    then xs >< (take n xs)
                    else cycleTaking (n - size xs) (xs >< xs)
```

It can be shown that cycleTaking terminates for all inputs. However, due to the way Ints are defined in the agda2hs prelude[7], Agda is unable to see that subtracting is the same as reducing the input. To convince Agda, {-# TERMINATING #-} is added before the function definition.

4.3.4 Agda2Hs limitations

agda2hs does not completely identify the common subset of Agda and Haskell. Thus, certain features of Agda could not be used. Most importantly, `case_of_` λ `where` is not supported by agda2hs. This required a large rewrite of many functions to instead use `let`, or to move the definitions in the `where` block.

5 Verification

This section describes the process of verification of the sequence library. Verification is performed entirely on the Agda side, and is not translated to Haskell using agda2hs. Haskell lacks support for dependent types. While progress is being made on adding dependent types to Haskell[28], it would still not be a suitable proof assistant since it is not a total language. Non-total functions do not provide the guarantee of being valid proofs.
The properties proven for sequences can be divided into preconditions, and equational properties that hold for all values of a given type. A description of the $Node\hat{}$method for proving properties is provided before equational properties are discussed.

5.1 Preconditions

Preconditions are properties or conditions that must hold when a function is called. If the preconditions do not hold, it is not possible to call the function as it is impossible to construct a value of the empty type. Preconditions effectively restrict the domain of the function. Some important preconditions used are discussed.

[7]https://github.com/agda/agda2hs/blob/master/lib/Haskell/Prim/Int.agda

5.1.1 Non Empty

The non-empty preconditions are used to restrict a function from taking empty sequences/finger trees as an input.

```
isNotEmptySequence : {a : Set} -> Seq a -> Bool
isNotEmptySequence (Sequence EmptyT) = false
isNotEmptySequence _ = true

isNotEmptyFingerTree : {a : Set} -> FingerTree a -> Bool
isNotEmptyFingerTree EmptyT = false
isNotEmptyFingerTree _ = true
```

Most functions in the sequence library make use of auxiliary functions that operate on the finger trees. For instance, the lookup function calls the lookupTree function on the finger tree of the sequence. Only the function types are provided here for the sake of brevity.

```
lookupTree : {a : Set} -> {{Sized a}} -> Int -> (xs : FingerTree a)
            -> {IsTrue (isNotEmptyFingerTree xs)} -> Int × a

lookup : {a : Set} -> Int -> (ss : Seq a)
        -> {IsTrue (isNotEmptySequence ss)} -> Maybe a
```

Thus, when lookup is called with a non empty sequence, lookupTree must be called with a corresponding non empty finger tree. It can be seen from the above definitions of isNotEmptySeqeuence and isNotEmptyFingerTree that a non empty sequence implies a non empty finger tree. However, this is not obvious to Agda, and the supplied implicit proof cannot be given to lookupTree. This is solved by defining a function that when given a non empty sequence, constructs a proof that the underlying finger tree is non empty as well.

```
getTreeFromSequence : {a : Set} -> Seq a -> FingerTree (Elem a)
getTreeFromSequence (Sequence xs) = xs

nonEmptySeq->nonEmptyFingerTree : {a : Set} -> (xs : Seq a)
-> {IsTrue (isNotEmptySequence xs)}
-> IsTrue (isNotEmptyFingerTree (getTreeFromSequence xs))
nonEmptySeq->nonEmptyFingerTree (Sequence (Single x)) = IsTrue.itsTrue
nonEmptySeq->nonEmptyFingerTree (Sequence (Deep v pr m sf)) =
                                                    IsTrue.itsTrue
```

5.1.2 Not Nothing

Another function that selectively throws an error is errorIfNothing.

```
errorIfNothing : {a : Set} -> (m : Maybe a) -> {IsTrue (isNotNothing m)}
                                            -> a
errorIfNothing Nothing = error "index out of bounds"
errorIfNothing (Just x) = x
```

One function that uses errorIfNothing is the index function. While lookup returns a Maybe type, returning a Nothing for indices out of range, the index function throws an error in case of an out of bounds index. Thus, the index function simply calls the lookup function, and throws an error if the result is Nothing.

```
isNotNothing : {a : Set} -> Maybe a -> Bool
isNotNothing Nothing = false
isNotNothing _ = true

index : {a : Set} -> (xs : Seq a) -> (i : Int)
        -> {notEmpty : IsTrue (isNotEmptySequence xs)}
        -> {IsTrue (isNotNothing (lookup i xs {notEmpty}))} -> a
index xs i {notEmpty} {notNothing}
        = errorIfNothing (lookup i xs {notEmpty}) {notNothing}
```

5.2 Node (Node ... (Elem a))

Many properties described in the following sections only hold for values of FingerTree (Elem a), and do not hold for values of FingerTree a, for a polymorphic type a. One prominent example is the equivalence of sizes of sequences and the lengths of their corresponding lists. This only holds when the size of an element of a sequence is 1.
However, there is a glaring issue when attempting to prove properties on values of FingerTree (Elem a). Assume that a function takes a FingerTree (Elem a) as input (among others), and returns a proof. The sub tree of this finger tree would be of type FingerTree (Node (Elem a)). This makes it impossible to recursively use the induction hypothesis.
A solution to this problem was devised in the form of the function `Node^`.

```
Node^ : Nat -> Set -> Set
Node^ zero a = a
Node^ (suc n) a = Node (Node^ n a)
```

`Node^ n` can be used to describe a type nested under n Nodes. Thus, we may now describe the type taken by proofs as a `FingerTree (Node^ n (Elem a))` for arbitrary n. This enables the usage of the induction hypothesis by a recursive call that takes an incremented n (suc n) in addition to the `FingerTree (Node^ (suc n))` value.

5.3 Type class laws

The following type classes and their laws have been verified.

Type class	Laws
Foldable	foldMap f = fold . fmap f foldMap f . fmap g = foldMap (f . g)
Functor	fmap id = id fmap f . fmap g = fmap (f . g)
Traversable	traverse Identity = Identity t . traverse f = traverse (t . f)
Semigroup	(x <> y) <> z = x <> (y <> z)
Monoid	mconcat [x] = x mconcat (map mconcat xss) = mconcat (concat xss)
Applicative	pure id <*> x = x pure (f x) = pure f <*> pure x mf <*> pure y = pure (-> g y) <*> mf f <*> (g <*> x) = (pure (.) <*> f <*> g) <*> x
Monad	return x »= f = f x

Table 3: Type class laws proven for Sequence

Proving the functor laws for Seq require proving the laws for FingerTree, Node, Digit, and Elem. A partial proof for the first functor law is as follows. The proofs for Node, Digit, and Elem are omitted as they are trivial.

```
functorDigitId : {a : Set} -> (d : Digit a) -> fmap id d ≡ id d

functorNodeId : {a : Set} -> (n : Node a) -> fmap id n ≡ id n

functorElemId : {a : Set} -> (e : Elem a) -> fmap id e ≡ id e
```

Armed with the proofs for Digit, Node, and Elem, we may now prove the property for FingerTree.

```
fingerTreeIdDigits : {a : Set} -> (v : Int) -> (pr : Digit a)
            -> (m : FingerTree (Node a)) -> (sf : Digit a)
            -> Deep v (fmap id pr) (fmap (fmap id) m) (fmap id sf)
            ≡ Deep v pr (fmap (fmap id) m) sf

functorFingerTreeId : {a : Set} -> (fT : FingerTree a)
                    -> fmap id fT ≡ id fT
functorFingerTreeId EmptyT = refl
functorFingerTreeId (Single x) = refl
functorFingerTreeId (Deep v pr m sf) =
    begin
        fmap id (Deep v pr m sf)
    =<>
        Deep v (fmap id pr) (fmap (fmap id) m) (fmap id sf)
    =< fingerTreeIdDigits v pr m sf >
        Deep v pr (fmap (fmap id) m) sf
    =< ? >
        Deep v pr (fmap id m) sf
    =< cong (λ mm → Deep v pr mm sf) (functorFingerTreeId m) >
        id (Deep v pr m sf)
    end
```

However, the function that fmap is called with on sub trees continuously grows as the finger tree becomes deeper. Thus, to use the induction hypothesis in the proof above, it is necessary to show that `fmap id = id` for all `Node`. While this cannot be proven in Agda, we are able to prove that `fmap id n = id n` for all `n` of type Node. We postulate a general rule that for any functions `f` and `g`, `f = g` if `f x = g x` for all `x`.

```
postulate
    funExt : {A B : Set} (f g : A -> B) -> ((x : A) -> f x ≡ g x) -> f ≡ g
```

We may now use `funExt` to fill in the '?' in the proof and complete it. Similarly, we obtain `fmap id = id` for all Elem, as we have already proven that `fmap id e = id e` for all e of type Elem. Using this postulate, we may prove the first functor law for Seq.
A similar method may be used to prove the Foldable and Traversable laws.
The Semigroup law is proved by proving the associativity of concatenation. This is discussed in section 5.5.
The proofs of the Monoid, Applicative, and Monad laws are excluded for the sake of conciseness. All these proofs are available in the public domain.[8]

5.4 Properties on sizes

This section describes general properties related to the sizes of sequences and the lengths of their corresponding lists. The proofs in this section illustrate the capabilities of types as first class citizens, and the `Node^` construct introduced in section 5.2.

Constructors for FingerTree and Node are defined to enable constant time access to the size of the data type. The constructors Deep, Node2, and Node3 contain an integer in the constructor to facilitate this, as seen in section 4.1. However, it is possible to instantiate a value of FingerTree or Node with an integer that is not equivalent to the actual size of the underlying data structure. For example, the following is an invalid node that is accepted by the typechecker. Validity is defined as a structure with a valid size value.

```
invalidNode : {a : Set} -> Node a
invalidNode = Node2 1 0 0
```

There are two possible solutions to this problem. The first would be to show using proofs that functions that alter the structure of a Seq/FingerTree/Node do not invalidate a valid sequence. The second would be to include invariants as implicit arguments in the data type itself. The former is explored in detail in section 5.4.1. A brief description of the latter is provided in section 5.4.2.

[8]https://github.com/Sh-Anand/practical-verification-of-sequences-/blob/master/code/src/ProofsTypeClass.agda

5.4.1 Proofs of validity

We describe a proof of a straightforward property : the size of a sequence must be equivalent to the length of its corresponding list. The definition of `toList` is provided in the agda2hs prelude[9].

```
toList : t a -> List a
toList = foldr _::_ []

sizeUnchangedToList : {a : Set} -> (xs : Seq a)
            -> {IsTrue (isValidSeq xs)} -> size xs ≡ lengthList (toList xs)

sizeUnchangedToList (Sequence xs) {valid} =
    begin
        size (Sequence xs)
    =<>
        size xs
    =< sym (identityInt (size xs)) >
        0 + size xs
    =< commutativeInt 0 (size xs) >
        size xs + 0
    =< ? >
        lengthList (foldr (flip (foldr _□_)) [] xs)
    =<>
        lengthList (toList (Sequence xs))
    end
```

However, we see that it is necessary to now prove that:
`size xs = lengthList (foldr (flip (foldr _::_)) [] xs)`
Note that the right hand side is not equivalent to `toList xs`. We can show that it holds by attempting to prove the general property that:
`size xs + lengthList z = lengthList (foldr (flip (foldr _::_) z xs))`
for all xs of type FingerTree (Elem a) and z of type List a

We attempt a proof as follows.

```
sizeFingerTreeFoldSplit : {a : Set} -> (z : List a)
                -> (xs : FingerTree (Elem a))
                -> {IsTrue (isValidFingerTree  xs)}
                -> size xs + lengthList z
                ≡ lengthList (foldr (flip (foldr _::_)) z xs)
```

However, it would be impossible to complete this proof. As described in section 5.2, it would not be possible to use the induction hypothesis for the given definition. Thus, we modify the proof to instead prove it on a `FingerTree (Node^ n)` for an arbitrary `n`. We now encounter another issue, namely the function to fold with. The number of nested `flip (foldr _::_)` increases linearly with `n`. To solve this problem, we define a function that when given an `n`, generates the appropriate folding function.

```
node^Folder : {a : Set} -> (n : Nat) -> Node^ n (Elem a) -> List a -> List a
node^Folder zero = flip (foldr _::_)
```

[9] https://github.com/agda/agda2hs/blob/master/lib/Haskell/Prim/Foldable.agdaL55

```
node^Folder (suc n) = flip (foldr (node^Folder n))

sizeFingerTreeFoldSplit : {a : Set} -> (n : Nat) -> (z : List a)
-> (xs : FingerTree (Node^ n (Elem a)))
-> {IsTrue (isValidFingerTree {{ sizedNode^ n }} xs)}
-> size {{ FingerTreeSized {{ sizedNode^  n}} }} xs + lengthList z
≡ lengthList (foldr (node^Folder n) z xs)
```

`sizedNode^` in the above snippet generates the appropriate Sized instances for `Node^`. This property can now be easily proven by further proving similar properties on `Digit (Node^ n (Elem a))` and `Node^ n` for arbitrary `n`. These proofs are omitted for the sake of brevity and can be found in the repository[10].

Some other properties proven on sizes are shown below. They are proven using techniques similar to the previous proof.

1. `size (x <| xs) ≡ 1 + size xs`
2. `size (x <| xs) ≡ lengthList (x :: toList xs)`
3. `size (xs |> x) ≡ size xs + 1`
4. `size (xs |> x) ≡ lengthList (toList xs ++ (x :: []))`
5. `lengthList xs ≡ size (fromList xs)`
6. `size xs ≡ size (fromList (toList xs))`

5.4.2 Implicit arguments in constructors

Invariants may be proven by including proofs as implicit arguments in the constructors of the data types. This ensures that an invalid value of a type cannot be constructed. For example, the `Deep` constructor of FingerTree may be modified as follows to include the size invariant.

```
Deep : (v : Int) -> (pr : Digit a)
    -> (m : FingerTree (Node a)) -> (sf : Digit a)
    -> {IsTrue (v == size pr + size m + size sf} -> FingerTree a
```

The primary advantage of this method is that it is no longer required to prove that functions preserve the validity of sizes as shown in section 5.4.1. The proof is now encoded in the constructor itself, and it would be impossible to construct an invalid value. However, a disadvantage would be that all functions that operate on finger trees would have to be modified.

5.5 Other properties

In this section, some general properties that were proven on the structure of sequences and their corresponding lists are listed. We omit the proofs of these properties as they are proven using the techniques described in sections 5.2 and 5.4.1.

[10] https://github.com/Sh-Anand/practical-verification-of-sequences-/blob/master/code/src/ProofsSized.agdaL84

1. `toList (x <| xs) ≡ x :: (toList xs)`
2. `xs ≡ toList  (fromList xs)`
3. `toList xs ≡ toList (fromList (toList xs))`
4. `toList ((xs >< ys) >< zs) ≡ toList (xs >< (ys >< zs))`

6 Responsible Research

The paper primarily describes the implementation and verification of Agda code and its subsequent translation to Haskell using agda2hs. These proofs are validated by the Agda typechecker. As a result, there are no issues regarding validity and bias.
All code discussed in the paper is made available in the public domain. Furthermore, a detailed description of the verification process is presented. The agda2hs framework is also an open source project. Thus, reproducibility of the process and results are ensured. No other ethical concerns regarding the paper have been identified.

7 Results and Discussion

The implementation of the Sequence library falls under the common subset of Haskell and Agda as identified by agda2hs. No extensions to the current version of agda2hs were required to implement the library.
The library contained some functions that only operated on non-empty sequences, and would throw an error otherwise. In a total language such as Agda, this would have to be encoded as a precondition as it is not possible to throw an error. Thus, we restrict the domain of the function by implicitly including a proof of the input sequence being non-empty. Furthermore, the library guarantees that the size value included in the sequence data type is valid. While it is possible to violate the invariant by instantiating an invalid sequence, it cannot be done by users of the library in Haskell as the internal structure is not exported for use. Thus, it suffices to prove that functions that modify the input sequences return valid sequences. In the future, this proof method may be applied to extend the number of functions that have been verified.
We have formally stated and proven some properties of the Sequence library. The general proof method of `Node^` may be used to extend the set of verified functions of the library.
The time taken to implement the library was three weeks, while it took five weeks to verify some properties of the library. The time taken to implement the sequence library was much shorter than verifying it because of the availability of a reference implementation in Haskell.
A discussion of possible ways to reduce the cost of verification, and of other possible improvements to the process are provided as follows.

7.1 Reducing cost of verification

The time taken for verification of code is much higher than the time taken to implement the code itself. It is of utmost importance to reduce this cost to make verification feasible. Some possibilities are presented as follows.

1. Postulate trivial properties and properties of imported types. For example, some properties on lists were postulated (such as the associativity of concatenation) to prove properties on sequences.

2. Prove the smallest/innermost properties first. Take the property $f\, p1 \equiv f\, p2$. If we possess a proof that $p1 \equiv p2$, then $f\, p1 \equiv f\, p2$ trivially follows. Thus, to prevent redundacy and save time, it is useful to identify the innermost properties and prove them first.

3. Always begin the proof by using the case-split feature of Agda. More often than not case splitting enables trivially completing a proof using $refl$

7.2 Other improvements

Some improvements to Agda that would optimize the verification process are identified here.

1. Boolean blindness is the condition caused by boolean types not carrying any information beyond their value[29]. A boolean is one of either `true` or `false`, and does not provide any more information, such as the context or the condition. Boolean blindness makes proving properties using if_then_else difficult. Agda is unable to infer constraints imposed by the condition in if.

   ```
   head : {a : Set} -> (xs : List a) -> {IsTrue (isNotEmpty xs)} -> a

   maybeHead : {a : Set} -> List a -> Maybe a
   maybeHead xs = if (isNotEmpty xs) then Just (head xs)
                                     else Nothing
   ```

   The code above does not type check, as Agda is unable to infer the `IsTrue (isNotEmpty xs)` implicit argument from the condition.

2. Type checking proofs in Agda is slow if the size of the file exceeds a few hundred lines. A short term solution to the problem is to split proofs across multiple files. However, this leads to fragmentation.

8 Conclusions and Future Work

In this paper, the Sequence library is implemented. A description of the challenges faced during implementation is given, and the changes made to implement the library in Agda are discussed. The properties verified are listed, and a brief description of proof techniques is provided. The technique of using an arbitrary number of nest nodes is examined. Finally, a discussion on reducing the cost of verification is given.

The time taken to verify the library was almost twice that of implementing the library. However, it is to be noted that the availability of a reference implementation in Haskell greatly simplified the porting process. Although verification is an expensive process, it is useful as it is much stronger than testing, and does not rely on random inputs generated by QuickCheck for example.

In the future, it would be beneficial to redefine the FingerTree and Node types, and the subsequent implementations, to use an implicit proof of validity of size. This would greatly reduce the number of properties that must be proven. Furthermore, the set of identified and proven properties may be expanded to further verify the correctness of the implementation of sequences. It should be investigated if the arbitrary nested node method would streamline the process of proving properties of other libraries. Finally, it would be useful to optimize the proofs to reduce the time taken to type check and verify that the proofs hold.

References

[1] *Haskell language*, `https://www.haskell.org/`.

[2] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2016.

[3] *Agda*, `https://github.com/agda/agda`.

[4] C. Schwaab and J. G. Siek, "Modular type-safety proofs in agda," *Proceedings of the 7th workshop on Programming languages meets program verification - PLPV '13*, 2013. DOI: `10.1145/2428116.2428120`.

[5] P. van der Walt and W. Swierstra, "Engineering proof by reflection in agda," *Implementation and Application of Functional Languages*, pp. 157–173, 2013. DOI: `10.1007/978-3-642-41582-1_10`.

[6] F. Besson, S. Blazy, and P. Wilke, "Compcerts: A memory-aware verified c compiler using pointer as integer semantics," *Interactive Theorem Proving Lecture Notes in Computer Science*, pp. 81–97, 2017. DOI: `10.1007/978-3-319-66107-0_6`.

[7] *Agda2hs*, `https://github.com/agda/agda2hs`.

[8] *Sequence*, `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Sequence.html`.

[9] R. Hinze and R. Paterson, "Finger trees: A simple general-purpose data structure," *Journal of Functional Programming*, vol. 16, no. 02, p. 197, 2005. DOI: `10.1017/s0956796805005769`.

[10] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total haskell is reasonable coq," *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan. 2018. DOI: `10.1145/3167092`. [Online]. Available: `http://dx.doi.org/10.1145/3167092`.

[11] *What is coq?* `https://coq.inria.fr/about-coq`.

[12] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 76–85, Jul. 2004, ISSN: 0163-5948. DOI: `10.1145/1013886.1007523`. [Online]. Available: `https://doi.org/10.1145/1013886.1007523`.

[13] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich, "Ready, set, verify! applying hs-to-coq to real-world haskell code (experience report)," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. DOI: `10.1145/3236784`. [Online]. Available: `https://doi.org/10.1145/3236784`.

[14] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000, ISSN: 0362-1340. DOI: 10.1145/357766.351266. [Online]. Available: https://doi.org/10.1145/357766.351266.

[15] J. Cockx, *An introduction to property-based testing with quickcheck*, https://jesper.sikanda.be/posts/quickcheck-intro.html, 2020.

[16] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 269–282, ISBN: 9781450328739. DOI: 10.1145/2628136.2628161. [Online]. Available: https://doi.org/10.1145/2628136.2628161.

[17] N. Vazou, L. Lampropoulos, and J. Polakow, "A tale of two provers: Verifying monoidal string matching in liquid haskell and coq," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2017, Oxford, UK: Association for Computing Machinery, 2017, pp. 63–74, ISBN: 9781450351829. DOI: 10.1145/3122955.3122963. [Online]. Available: https://doi.org/10.1145/3122955.3122963.

[18] A. Gibiansky, *Finger trees*, https://andrew.gibiansky.com/blog/haskell/finger-trees.

[19] P. Wadler, "Propositions as types," *Commun. ACM*, vol. 58, no. 12, pp. 75–84, Nov. 2015, ISSN: 0001-0782. DOI: 10.1145/2699407. [Online]. Available: https://doi.org/10.1145/2699407.

[20] J. Cockx, *Programming and proving in agda*, https://github.com/jespercockx/agda-lecture-notes/blob/master/agda.pdf, Page 20, Table 2, 2021.

[21] *Dependent type*, https://en.wikipedia.org/wiki/Dependent_type, Apr. 2021.

[22] *Built-ins*, https://agda.readthedocs.io/en/latest/language/built-ins.html#equality.

[23] E. Normand, *What is a total function?* https://lispcast.com/what-is-a-total-function/, Sep. 2019.

[24] *What is agda?* https://agda.readthedocs.io/en/v2.6.1.3/getting-started/what-is-agda.html.

[25] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89, Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76, ISBN: 0897912942. DOI: 10.1145/75277.75283. [Online]. Available: https://doi.org/10.1145/75277.75283.

[26] *Type classes and overloading*, https://www.haskell.org/tutorial/classes.html.

[27] *Termination checking*, https://agda.readthedocs.io/en/v2.6.1.3/language/termination-checking.html.

[28] R. Eisenberg, "Dependent types in haskell: Theory and practice," *ArXiv*, vol. abs/1610.07978, 2016.

[29] R. Harper, *Boolean blindness*, `https://existentialtype.wordpress.com/2011/03/15/boolean-blindness`, 2011.