Practical Verification of the Inductive Graph Library

Rico van Buren, Jesper Cockx, and Lucas Escot

TU Delft

June 27, 2021

Abstract

Formal verification works better than testing, since the correctness of a program is proven. It is researched if it is possible and feasible to formally verify the Inductive Graph Library. The library is an abstract class in Haskell and is ported manually to Agda. Agda is a total and dependently typed language and thus can be used as a proof assistant. The functions are first converted to total functions and the preconditions the are proven. Verifying an abstract class is time consuming, since it requires an implemented instance of the abstract class. Stating the properties of the library is possible, but difficult since it requires generalised properties that need to be valid for all instances of the abstract class. The verification process is not completed yet, so no definitive conclusions are made, but the properties that are verified did not produce any issues. agda2hs is used to compile the Agda code to Haskell, this ensures that the verified Agda code is verified Haskell code. This requires the library to fall within the common subset of agda2hs.

1 Introduction

This research is about the verification of functional programming libraries written in Haskell [14]. Verification, of software libraries in general, is important because it reduces the chance of software bugs in the library, but completely verifying a library is difficult. The standard way of doing this is with testing, but it is nearly impossible to fully test a program. A more complete approach is formal verification, this is the process of proving correctness of programs with respect to certain formal specifications. So, the goal is to see if such a verification is possible in practice and if the time and effort required is reasonable.

In Agda [8], this type of verification is possible, because it is a dependently typed, which are types that depend on elements of other types [1], and a total language. So, there are no run-time errors, no incomplete pattern matches and no non-terminating functions and thus, Agda can be used as a proof assistant as discussed in section 2.3. This proof is checked automatically by the type-checker of Agda, and re-checked every time the code is changed. So, it provides an unusually high degree of confidence in the correctness of the program. However, Agda is not a popular language so most programmers do not understand the code that is written in it. agda2hs [9] tries to solve this problem by compiling Agda code to readable Haskell code such that the code is readable and works in Haskell. agda2hs does this by operating on a subset of Agda so the library must fall within this subset, or be changed accordingly.

In this project the focus lies on verifying the Inductive Graph Library, this library is contained in the Functional Graph Library [5], fgl for short. This library is created by Martin Erwig and builds upon a functional view of graphs and graph algorithms. This view is motivated and explained in his paper "Inductive graphs and functional algorithms" [6].

The aim of this research project is to answer the following summarised research questions: (1) Does the implementation of this library fall within the common subset of Haskell and Agda as identified by agda2hs? (2) What kind of properties and invariants does the library guarantee? And (3) is it possible and feasible to formally state and prove the correctness of Haskell libraries that are ported to Agda? The complete research questions can be found in section 4.1.

To answer these questions, the library is manually ported to Agda, and the non-total functions are made total by stating the right preconditions. Here, the first issues were encountered. Since, the Inductive Graph Library is an abstract class some preconditions required properties that could not be proven without an implemented instance of the abstract class. To solve this, an instance is implemented based on the PatriciaTree which is also contained in the fgl library. After this, the properties of the Inductive Graph are found and most of them are stated as abstract properties, you can read more about them in section 4.3. Some of these properties are proven, but most of them are left as future work. Finally, the Inductive Graph Library is ported back to Haskell with agda2hs.

The answers to these research questions can be found in section 8, they are summarised here. The library falls within the common subset of agda2hs with some exceptions and changes. The properties can be taken from the QuickCheck properties and the graph has two invariants. Namely, the graph should not contain duplicated nodes, and edges need to reference nodes that are contained in the graph. However, the original library only guarantees that the outgoing node is referenced in the graph and does not guarantee the other invariants. The verification process is not complete, so the third research question cannot be fully answered, but the following can be concluded. Verifying an abstract class, is more work than verifying a non-abstract class, since the verification requires an implemented instance of the abstract class. However, some basic properties can be proven as mentioned in section 5.5. If the only goal is to verify the abstract class, it is probably not feasible to use formal verification. Stating the correctness is possible, but mistakes can be made. So, valid instances could be classified as invalid when trying to prove invalid properties.

The paper starts by giving background information in section 2, in section 3 the method is described. Then in section 4, the problems that were encountered during the project are discussed and solutions are proposed. After that, section 5 states the results of this research. Section 6 describes the reproducibility and section 7 discusses the results and techniques used in this project. Finally, section 8 concludes the results and section 9 formulates future research.

2 Preliminary

In this section the basics of the Inductive Graph library are explained, then it is discussed how one could write Haskell code as Agda code. After that the basics of Agda as a proof assistant is explained.

2.1 The Inductive Graph Library

An inductive graph, is a graph data structure that decomposes in to contexts. A context is a node, together with its label, its incoming edges (predecessors) and outgoing edges (successors). To built the graph, a context is merged with an empty or an already existing graph. Reading the graph, decomposes the graph into a context and the remaining graph. This is done with pattern matching on the merge function. To make this pattern match possible, the programming language should support pattern matching on functions. Haskell does not support this, so the Haskell implementation uses an explicit match function to read the graph.

Internally, this is done by splitting the implementation of the graph into two classes, the Graph class and the DynGraph class. The Graph class defines the match function and has functions to create the graph. However, once the graph is created, it cannot be altered. The DynGraph solves this issue by defining the merge function, which can add a context to the graph.

The library implements the Graph and DynGraph as abstract classes and provides a few default functions. This makes the Inductive Graph Library an interesting case, since how could one verify functions that have no implementation? In section 4.4, a solution is proposed that reduces this problem in a way that allows the verification of the libraries default functions and could make it feasible to implement the abstract class.

2.2 Porting Haskell code to Agda

Agda and Haskell are syntax wise closely related, however Agda is a total language and Haskell is not. A total language, is a language that does not allow the programming of exceptions, infinite loops and incomplete pattern matches, so it guarantees that programs written in it do not crash and that they always terminate. This requires adding preconditions to non-total functions to make them total. These preconditions are given with Instance Arguments [3]. For an example, see the following function from the Graph class:

```
matchAny : (grab : gr a b) -> {{ IsFalse (isEmpty grab) }} -> GDecomp gr a b
```

This function requires an instance of type IsFalse (isEmpty grab). So, to use the matchAny function you need to supply a proof that isEmpty grab returns false.

These preconditions can use functions that are defined as abstract. This is the case for the isEmpty function defined in the Inductive Graph Library. Since, the isEmpty function is not implemented, it makes it impossible to prove this precondition without any additional information from the abstract class or an actual implementation of the isEmpty function. In this paper, this additional information is given by properties that should be implemented by an instance of the abstract class and they are called abstract properties. They are discussed in section 4.3.

2.3 Agda as proof assistant

According to the Curry-Howard correspondence [10], it is possible to write mathematical proofs as programs. A proposition is a type, a proof is a program and simplifying a proof is evaluating a program. When evaluating the program, the program should not loop nor throw an exception. So, proofs need to cover all cases and inductive proofs need to reduce to the base case. That is why, it is important to use a total language to program proofs. So, if it is possible to implement a function of a specific type and evaluate that function, it is a proof for that type. For example, proving that A implies B, is the same as implementing a function with the type A -> B.

It is also possible to prove equivalence. This is done by defining equivalence with the use of reflexivity and writing functions that imply equivalence. The definition of reflexivity and an example is shown below:

```
-- reflexivity of equality
data ≡__ {a} {A : Set a} (x : A) : A → Set a where
    instance refl : x ≡ x

-- example
example : {a : Set} -> (x : a) -> (y : a) -> (if true then x else y) ≡ (if false
    then y else x)
example x y = refl
```

So, as shown the example, if two functions with a given input unify to the same value they are equivalent. With equivalence, one can also define symmetry, transitivity and congruence.

3 Method

With Agda as a proof assistant, it is possible to formally verify software libraries. The method that is used is listed below. After completing the tasks that are defined in the method, the research questions are answered.

1. Code the library in Agda and make functions total.
2. Define the properties of the library.
3. Verify the library.
4. Port the library back to Haskell with agda2hs.

The Haskell syntax and the Agda syntax are fairly similar, so the total functions can be easily written in Agda. However, The non-total functions need to be converted, this is explained in section 4.2. The properties can be taken from the QuickCheck properties, as mentioned in section 4.4. But, most of them need to be defined as abstract properties, they are discussed in section 4.3. With Agda, these properties are then verified by programming proofs. This is done with the use of induction and the help of other properties. Finally, the functions that need to be compiled to Haskell are annotated with {-# COMPILE AGDA2HS < FUNCTION > #-}. Then agda2hs is run, and the Haskell code is checked with GHC.

4 Problem analysis

In this chapter the research questions are stated and the problems that have arisen during the project are discussed. First, the problem of converting non-total to total functions is discussed. Then it is discussed why abstract properties form a problem. Finally, it is discussed how properties can be found.

4.1 Research Questions

In this section the research questions are listed. The goal of this research is to answer the following questions.

RQ1. Does the implementation of the Inductive Graph 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?

RQ2. 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?

RQ3. 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?

4.2 Non-total to total

Some functions in the inductive graph library are non-total. They throw an exception if an unsupported input is supplied. An example is pattern matching on only the Just constructor of the Maybe type. This is an incomplete pattern match. So, when it is called on Nothing, it throws an exception. In the Inductive Graph Library this pattern is commonly used in combination with the match function. Match takes a node and matches it against the graph. It then returns a decomposition of the graph, which only contains the context if the node is found and it contains the remaining graph. Since, it is possible that the node does not exist in the graph, it wraps the context in a Maybe type. However, some functions think that their supplied node is always contained in the graph and unwrap the Maybe type directly into a Just. If that is not possible, an exception is thrown. So, functions that use the match function without checking if the node is contained in the graph, need to have a precondition that the node exists or the function itself needs to guarantee that the node is contained.

Fortunately, most of these functions can proof that the supplied node exists in the graph if it is nonempty. The graph should be nonempty, because no nodes are contained in an empty graph. So, these functions require the precondition that a graph is nonempty. A nonempty graph then implies that its list of nodes is also nonempty. But, both of these functions are abstract, so without any additional information, there is no way of knowing if these functions are correct. And if they are incorrect, this implication does not hold. The additional information can be given with abstract properties, which are explained in the next section.

4.3 Abstract properties

An abstract property is a property that is not proven in the abstract class. Instead, it is implemented by the class that implements the abstract class. When working with the abstract class, these properties can be used to prove preconditions or to prove other properties. The abstract properties of the Graph class and the DynGraph class are defined below. The first two properties are properties of the Graph class and the other property is a property of the DynGraph class. labNodes is a function that returns all the nodes in the graph with their labels.

```
-- | The property that the graph and the labNodes share the same nodes
propGraphAndLabNodesSameNodes : (n : Node) -> (grab : gr a b) -> isJust (fst
    (match n grab)) ≡ elem n (map fst (labNodes grab))

```

```
-- | The property that an empty graph has no nodes and a nonempty graph does
     contain nodes
propEmptyGraphIsEmptyLabNodes : (grab : gr a b) -> isEmpty grab ≡ null (labNodes
     grab)

-- | Updating a context should not remove nodes
propUpdatingContextKeepsNodes : (cxt : Context a b) -> (nUpdate : Node) -> (nKeep
     : Node) -> (grab : gr a b) -> {{ _ : IsTrue (nodeInContext nUpdate cxt) }} ->
     {{ _ : IsTrue (gelem nKeep grab) }} -> gelem nKeep (cxt & (snd (match nUpdate
     grab))) ≡ true
```

These properties are required to transform the non-total functions into total functions. The abstract properties of the Graph class are used to guarantee that the graph and the list of nodes contain the same nodes and that the isEmpty function is equivalent to checking that labNodes is an empty list. The abstract property of the DynGraph class is used to ensure that if the context of a node is updated it does not change the nodes. This is required for inserting multiple edges into the graph, since the nodes that are referenced by an edge need to be contained in the graph.

Abstract properties can also be used to prove other properties. Take as example, two functions that insert nodes:

```
-- | Insert a labeled node into the 'Graph'.
insNode : {{ DynGraph gr }} -> LNode a -> gr a b -> gr a b
insNode (v , l) = ((_,_,_,_ [] v l [] ) &_)

-- | Insert multiple labeled nodes into the 'Graph'.
insNodes : {{ DynGraph gr }} -> List (LNode a) -> gr a b -> gr a b
insNodes vs g = foldl' (flip insNode) g vs
```

Here the merge function, defined as _&_, is an abstract function. Nothing can be proven about this function, since its implementation is unknown. insNode uses this function and thus the complete implementation of this function is also unknown. Now suppose that there is an abstract property that proofs that insNode works as intended. This property can then be used to prove that insNodes works as intended, since most of the implementation is known. Only the insNode part is unknown, but with the abstract property it can be assumed that it works as intended.

Only there is a problem, defining abstract properties is dangerous. It is possible to define any abstract property and to use that property. For example, take a look at the following two abstract properties.

```
-- | Abstract property that true is equivalent to false.
propTrueIsFalse : true ≡ false

-- | Abstract property that isEmpty is equivalent to calling null on a list of
     nodes
propEmptyGraphIsNullNodes : (ns : List Node) -> (grab : gr a b) -> isEmpty grab ≡
     null ns
```

The first abstract property is obviously false, but it is valid Agda code. No exceptions or warnings are thrown and the abstract property can be used like any other property. With this property it is possible to prove anything about the abstract class, since true is equivalent to false. The second abstract property contains an easy to make mistake, the property should use the node list of the graph but instead it uses an arbitrary list. Since this property is not correct, a counterexample can be found. With this counterexample it is possible to prove that true ≡ false, given a graph instance, as shown below:

```
postulate
    -- Property of the graph, calling isEmpty on empty returns true.
    propEmptyIsEmpty : {{ _ : Graph gr }} -> isEmpty {gr} {a} {b} empty ≡ true

propTrueIsFalse : {{ _ : Graph gr }} -> gr a b -> true ≡ false
propTrueIsFalse {gr} {a} {b} _ = helper (0  []) empty {{ subst IsTrue (sym
    propEmptyIsEmpty) itsTrue }}{{  itsNonEmpty }}
    where
        helper : (ns : List Node) -> (grab : gr a b) -> {{ IsTrue (isEmpty grab)
            }} -> {{ NonEmpty ns }} -> true ≡ false
        helper (n  ns) gr {{ emptyGraph }} =
            begin
                true
            =< sym (propIsTrueIsJustTrue (isEmpty gr) {{ emptyGraph }}) >
                isEmpty gr
            =< propEmptyGraphIsNodes (n  ns) gr >
                null (n  ns)
            =<>
                false
            end
```

This proof uses the fact that it is always possible to construct an empty graph and a nonempty list of nodes. The helper function requires isEmpty to be true and a nonempty list of nodes. Since these can be constructed, it is possible to prove that true is equivalent to false.

With this mistake it is not possible to prove that true is equivalent to false in general. That is because these properties require an instance of the abstract class, which can only be created when the abstract class is implemented. This faulty abstract property is impossible to implement, it is incorrect, and thus it is impossible to actually implement an instance. So, to make sure that an abstract class with abstract properties can be implemented, an implementation should be given to poof that this is possible.

Such a Haskell based implementation is given in the fgl library, called the PatriciaTree. Of course, this needs to be converted to Agda to implement the Agda version of the Inductive Graph. This is not directly possible since the PatriciaTree depends on IntMap, it is the underlying data structure. The Agda Standard Library provides a Map data structure based on an AVL Tree. This could be converted to an IntMap but the Agda version of Map is difficult to use. Thus, it was opted to implement a simple version of IntMap based on lists and pairs. With this IntMap, the PatriciaTree is implemented.

4.4 Defining properties

Properties can directly be taken from the QuickCheck properties of the Inductive Graph Library. In theory these can be converted to be used as properties in Agda. However, they test a lot at once which makes writing proofs difficult. To solve this issue, they can be split in to multiple properties. Properties also arise when proving the preconditions. Most of the preconditions are indirectly tested in the QuickCheck properties so not all QuickCheck properties need to be converted.

Most properties should hold for instances of an abstract class. For example, inserting a node into an empty graph and then deleting that node should produce an empty graph. Normally, the general QuickCheck properties for abstract classes are only implemented once, since they can be reused by each instance. But this is not always the case in Agda, since each instance can differ. These differences will show up in the proofs and thus not all proofs can be interchanged between instances. However, this differs per instance, so if the basic implementation is roughly the same, proofs for basic properties could be interchanged. It is also possible to interchange proofs if the properties are based on the default implementation of the abstract class and the instance uses these default functions. It is also possible to prove properties that do not depend on the instance at all, then they do not need to be proven by an instance.

To guarantee that the abstract properties are implemented, an abstract properties class is created. The goal of this abstract class is to guarantee that all properties of the abstract class hold for the implementation of the instance. So, implementing the abstract properties class guarantees, that the implemented instance is valid.

However, to make this work, properties need to be generalised. For example, take a property of inserting a node. Inserting a node should insert only that node and should not alter the other nodes. The code snippet below shows two ways of defining this property. equalLists checks if the two lists contain exactly same elements, but does not care about the order.

```
propInsNode1 : {{ _ : Eq a }} -> (ln : LNode a) -> (grab : gr a b) -> equalLists
     (labNodes (insNode ln grab)) (ln :: (labNodes grab)) ≡ true

propInsNode2 : {{ _ : Eq a }} -> (ln : LNode a) -> (grab : gr a b) -> labNodes
     (insNode ln grab) ≡ ln :: (labNodes grab)
```

The first property should hold for all implementations of the graph, but the second property only holds for some implementations, since it is not required in general that the inserted node is the head of the labelled nodes list. However, the second property is easier to prove since the lists are equivalent. If this is the case for the first property, an additional property is needed which states that if two lists are equivalent, they are equal. The second property is also easier to use by properties that depend on it. Because, reflexivity is easier to use and more versatile than a property that is equivalent to true.

The properties from the abstract properties class can be used to prove properties about the abstract class, if they only depend on the abstract properties. For example, insNodes depends only on insNode, so if the insNode property is given, the insNodes property can be proven. But as mentioned earlier, using the generalised version of the property makes proving properties, that depend on it, also more difficult to prove. Therefore, two methods have been considered: (1) Adding the insNodes property to the abstract properties class, such that it needs to be proven for all instances and (2) making insNodes a general property such that it is directly proven for all instances that implement the insNode property. These methods are compared in section 5.5.

5 Results

In this section, the results of the project are discussed. It gives the results to the research questions and results of the proposed solutions of the problems discussed in section 4. First, the result of using agda2hs are shown. Then what invariants were found. After that, the preconditions that were required. Then, how a graph instance is implemented. After that, the properties that the inductive graph should satisfy and finally the simplifications that have been made.

5.1 agda2hs

After porting the library to Agda, it is converted back to Haskell with agda2hs. agda2hs operates on a subset of Agda, however the library does not completely fall within this subset. This resulted in some issues. These are listed in table 1, most of these issues were solved fairly easily. Some of them required the use of a foreign agda2hs pragma. This pragma allows Haskell code to be inserted into an Agda file. When agda2hs compiles the Agda file to Haskell, it will check the foreign Haskell code to see if the syntax is correct and place the code into the compiled Agda code.

Issue	**Solution**
Creating an instance, with an auxiliary function.	Use foreign agda2hs.
Case of combined with where clause[1].	Use a let.
Lambda with record syntax.	Add {-# LANGUAGE LambdaCase #-} with foreign agda2hs.
Splitting function definition and implementation together with using a where clause.	Use a let.
Properties accidentally compiling to Haskell.	Only use properties in implicit or instance arguments.
agda2hs tries to compile implicit fields in record instances.	Use foreign agda2hs.
Function definitions in where clauses do not compile correctly.	Change agda2hs implementation to ignore function definitions in where clauses[2]. See section 6.
Functions from dependencies compiling inline to Haskell.	Add the NOINLINE pragma to the function from the dependency.
Agda issue: Haskell's syntactic sugar for list creation.	Add a custom createList function. Use foreign agda2hs to compile this to the syntactic sugar.

Table 1: agda2hs results

5.2 Invariants

A graph is valid when the following two internal invariants are not violated:

1. A graph should not contain duplicate nodes.
2. Edges in the graph should only reference nodes that are in the graph.

The first invariant is not guaranteed, only a part of the second invariant is originally guaranteed by the Inductive Graph Library. It only throws an exception when the outgoing node referenced by the edge is not part of the graph. Throwing an exception is not allowed in Agda and thus it is converted to a precondition. When converting this exception, the

[1]Known Issue
[2]Solution sugested by project member

incoming node referenced by the edge was also taken care of. This guarantees the second invariant.

There is a problem with this solution, it only guarantees it for the insEdge and insEdges function. These functions rely on the merge function. This function is the source of the problem. It allows to merge any context into the graph. This context could contain invalid edges and or nodes and thus can break the invariants.

This could be solved in four ways. First it can be solved with the precondition that the node from the context is not contained in the graph and that the incoming and outgoing edges only reference nodes in the graph or the node in the context. The second option is that there is a safe and an unsafe DynGraph class. Then, if you want to use the graph, you first need to convert it to a safe version. The third option is a combination of these two. It could define some operations with preconditions such that they fall under the safe option and some operations without preconditions such that they fall under the unsafe version. At last, the merge function could be replaced by two functions that can safely insert a node or an edge, this can then be used as the safe version.

The first option is not feasible. Since some functions like gmap, which takes a function that operates on a context and maps that function over the graph, can alter the graph in such away that it is nearly impossible to state the correct precondition. For example, take a function that increments each node by one and updates the edges accordingly. This would then only be a valid graph after the gmap function is complete and breaks the invariant in between.

The other options could be possible, given enough time to implement. They take care of the inconsistencies that arise when using functions like gmap. But it comes at a cost, functions require more preconditions which need to be proven by functions that depend on these functions. However, none of these options are implemented yet in the current state of the Agda version of the Inductive Graph Library.

5.3 Preconditions

Some functions of the graph require preconditions. There are only three different kinds of preconditions required, they are listed below. Precondition three is extended in the Agda implementation, it also checks if the incoming node referenced by an edge is present in the graph. However, this extension is not required, because the library does not throw an exception for this invalid input, but it was easy to implement and it ensures the second invariant.

1. Calling functions that require a nonempty graph.
2. Calling functions that require that a specific node is contained in the graph.
3. Calling functions that insert an edge, require that the outgoing node referenced by the edge is present in the graph.

Proving the preconditions took more time then actually porting the Haskell code to Agda. This is due to the fact that the proofs for these preconditions are not always trivial, especially for abstract classes, since they can require other properties that cannot be proven from the abstract class directly, so they need to be defined as abstract properties.

5.4 Implementing a Graph Instance

As mentioned in section 4.3 it is important to give an implementation of a graph instance to ensure that the abstract properties are valid. The PatriciaTree library has been implemented and the abstract properties mentioned in that section have been proven.

Implementing the PatriciaTree took about three weeks. This was mostly due to the fact that the IntMap also needed to be implemented and that proving the abstract properties is not trivial. They also depend on the IntMap implementation, so the properties needed to be reduced to properties of the IntMap. These properties have been postulated since the IntMap implementation used in this project does not correspond to the Haskell implementation, so proving them does not proof the Haskell implementation.

5.5 Properties

In section 4.4 it was discussed how properties could be defined and how the implementation can be guaranteed. This idea is made concrete, two extra abstract classes have been created. They are named GraphProperties and DynGraphProperties. These define the abstract properties. They are taken from the QuickCheck properties. However, the GraphProperties and DynGraphProperties classes have not been fully implemented for the PatriciaTree instance of the graph. Implementing the PatriciaTree was not part of the initial project and took about the same time as implementing the Inductive Graph Library itself.

The abstract properties of the default Graph functions are implemented, they can directly be used in the abstract properties class if the instance uses the default functions. Two other abstract properties are also implemented, the insNode and the insNodes properties. The general version of the insNode property was quite easy to prove, using the property that two equivalent lists are equal. In the PatriciaTree implementation they are equivalent, since the PatriciaTree adds the node to the start of the list.

The insNodes property was more difficult to prove and the two methods that were proposed in section 4.4 are tried. Proving the insNodes property as an abstract property was easily reduced to a property of the IntMap. However, the generalisation of the abstract property still let to an issue, since the PatriciaTree adds all the nodes to the front of the IntMap. This causes the nodes inserted into the IntMap to be in the reverse order of the initial list of nodes that is inserted. This means that the insNode property cannot directly be used to prove the insNodes property, since the list is reverted. Proving the insNodes as a general property using the insNode has not been completed, since it required a lot of properties from the equalLists function.

In theory both of the above options are plausible, but the generalisation makes it difficult to prove these properties. In the case of the insNodes property, this is due to the fact that the order of the list should not matter. This could be solved by using a set instead of a list data structure, since the graph does not contain duplicate nodes, then the order would not matter and the sets would be reflective. This also eliminates the required equalLists function. However, it would still require additional properties of the set to make it possible to prove these kinds of properties. Also, this solution changes the original code, because the node list is originally a list, this would then become a node set. The generalisation creates also another issue. Validating the properties with an implementation does not guarantee that no mistakes are made in the generalisation, since there could be instances that satisfy the generalised property together with instances that do not satisfy the generalised property. That results in the following question. Is the instance incorrect, or is the generalised property incorrect?

Some properties from the abstract properties class are proven using the default properties of the Inductive Graph. These are the node count and node range property. This is possible since the PatriciaTree uses the default functions from the library. Some basic functions of the abstract class can also be proven if they do not rely on an instance, for the Inductive Graph Library this is currently only the EdgeProjections property. This property checks if the edge label is removed and added correctly.

5.6 Simplifications

Some simplifications have been made to the PatriciaTree functions to make the verification process simpler. The function that adds or clears the predecessors and successors have been simplified to not depend the number of predecessors and successors. Before the simplification they had a more efficient implementation, since they used a faster method of inserting and deleting when the number of edges was above a certain threshold. Unfortunately, this optimisation required more properties of the IntMap and thus it was left out. The other four simplification can be found in appendix A.

The prime functions remove the graph constructor such that the properties can be reduced to properties of the IntMap. The matchGrLookup also adds the support to give it an explicit maybe context. This helps reducing properties that depend on matchGr, but only care about the maybe context, to actually focus on the maybe context. The insNode and insNodes where simplified and proven equivalent to the original functions such that the simplified versions can be used in the proofs.

6 Responsible Research

This section describes the responsibility of this research project and explains the steps taken to ensure that the project is reproducible. It also states the versions that are used and discusses the ethical part of this research.

The research questions are answered by implementing a functional library, namely the Inductive Graph Library. The process is described and the problems that have arisen have been discussed. Following this process, the reader should also be able to reproduce these results and be able to answer the research questions for another library. The code is open source and can be found on GitHub[3]. So, if there are doubts about the implementation it can be checked. The code can also be used as an example.

The code in this project depends on three programs: The Glorious Glasgow Haskell Compiler (GHC), Agda and agda2hs. When writing the code, the following versions were used:

- GHC: version 8.10.3
- Agda: version 2.6.1.3
- agda2hs:
  - Branch: master
  - Commit: d48f8648ebbf3c9aeef9eb642a8dda0b00c2373b

[3]GitHub Repository: `https://github.com/RicovanBuren/agda2hs-inductive-graph-public`

The fgl version which contains the Inductive Graph Library, where this is project based on, is version 5.7.0.3.

One change has been made to agda2hs. This change removes the compilation of the type signatures in the where clause. To make this change, line 618 in Main.hs should be replaced with the following:

```
whereDecls <- concat <$> mapM (\ x -> tail <$> ((uncurry compileFun') x) )
    children''
```

No ethical concerns have arisen during the span of this project. Data from humans has not been collected and all referenced authors have been credited. Comments are provided within the code. This makes the code and properties more understandable. All postulates that are used in the code contain a comment explaining why they are correct and they are discussed in more detail in section 7.2.

7 Discussion

In this section, other proof assistants are discussed and the used postulates are discussed.

7.1 Other Proof Assistants

In this project Agda is used to verify the Haskell library. It was opted to go with Agda since its syntax is very similar to Haskell and with agda2hs it is simple to port back to Haskell. However, there are other options available.

One of them is LiquidHaskell. As stated by the LiquidHaskell blog: "It refines Haskell's types system with logical predicates that let you enforce critical properties at compile time" [7]. Just like Agda it guarantees that functions are total and it checks that functions terminate [16]. Guaranteeing the totality of a function is not done within the function definition itself, but instead the function is annotated with a precondition. With this functionality LiquidHaskell can also be used as a proof assistant. However, LiquidHaskell uses a refined language and is thus restricted to the logical language determined by the underlying SMT solver [15]. This means that not every property can be verified. Since, LiquidHaskell is a superset of Haskell, it could be easier to learn then Agda, because standard Haskell code can be used and the LiquidHaskell blog also contains demos and tutorials.

Another option is Idris [4]. Its syntax is very similar to Haskell and Agda. It also guarantees total and terminating functions. So, it can also be used as a proof assistant. The benefit that Idris has, is that it supports, next to the Agda proving style, the proving style of Coq [13] with the use of elaborator reflection [2]. So, if you are familiar with Coq, this could be a better option. However, there does not exist an easy way to convert Idris code back to Haskell yet.

hs2coq [11] is also an option. The aim of this project is to convert Haskell code to Coq. Then Coq can be used to verify the Haskell code. However, Haskell is a non-total language, so hs-to-coq will add the patternFailure axiom if it detects that a function is non-total and the unsafeFix axiom if it is a non-terminating function [12]. This means that verification is partly possible with Coq, but both of these axioms are unsound and should be resolved to be able to completely verify the program.

7.2 Postulates

Postulates are functions that are defined but do not have an implementation. Like abstract properties, they can also be used to prove any property if they are stated incorrectly. Within this project there are postulated properties about lists, IntMap and two general ones. The correctness of each postulate is described in the code with a comment. There only should be one postulate that has a counterexample, so it should be used carefully. The postulate is listed below.

```
sameElemIsEq : {{ _ : Eq a }} -> (n : a) -> (n == n) ≡ true
```

This property depends on an Eq instance. If this instance is incorrectly defined, the same element does not have to be equal to itself. This problem could be solved by defining it as an abstract property in the Eq class, then the property needs to be verified for all Eq instances.

To be sure that the postulates are correct they should be implemented and proven. Since this is not the case, they can be invalid. Only the IntMap properties are not very useful to implement since the implementation is not based on the Haskell implementation. So, proving these postulated properties does not proof the properties of the Haskell version of IntMap.

8 Conclusions

Some issues were discovered when porting the library with agda2hs. These issues were circumvented fairly easily, but not all were solved completely. Issues that arise by the use of a where clause can be fixed with a let function. Other issues required the use of foreign agda2hs. One issue was solved by making a change to agda2hs. This change ignores the type signatures in where clauses, but it still compiles the implementation. So, the original Haskell implementation does not fall into the common subset of agda2hs, but with some small changes and the use of some foreign agda2hs code blocks, the library is still successfully implemented in Agda and successfully ported back to Haskell.

Two invariants of the graph are found, but the Haskell library only partly guarantees one of them. Namely that the outgoing node of an edge is present in the graph, but it does not guarantee that the incoming node is also present. The Agda implementation does guarantee this. The library requires three kinds of preconditions. With these preconditions all exceptions that are thrown by the library can be prevented. The Haskell library guarantees all the properties, that are defined by the QuickCheck properties, that are contained in the GitHub repository of the fgl library.

Stating the correctness of an abstract class is possible, but difficult. Properties need to be guaranteed by the instances that implement the abstract class. In this project, this is done by using an abstract properties class. Properties in this class are generalised to be valid for all instances of the abstract class. This makes stating the properties cumbersome and it may introduce mistakes. To make sure that all the properties are valid, an instance of the abstract property class is implemented. However, this implementation is not complete yet and this method does not ensure the correctness of the generalised properties, since there could be instances that are correct and satisfy the generalised properties together with instances that are also correct but are not able to satisfy the generalised properties if a mistake is made in the generalisation process.

The required implementation of an instance of the abstract class, makes the time and effort spend higher than verifying a non-abstract class and thus it makes it infeasible from a time and effort perspective, if the only goal is to verify the abstract class. If the goal is to also verify the instances, it will be more feasible, but since it is not always possible to reuse the proofs for those instances it will still increase the time and effort spend for each instance that is added. No statements can be made about the actual time and effort required for the verification process since this process is not completed in this project.

Work has gone into simplifying the verification process. Defining an abstract properties class helps to guarantee that all instances satisfy the properties. With this method the properties are only stated once, so the definitions do not have to be repeated for each instance. The abstract properties class can is then used to prove general properties about the abstract class. But since they are general, they are harder to prove and thus require more work. However, if the goal is to implement more instances, this extra effort can be worth it. It reduces the number of properties in the abstract properties class and thus it requires less properties to be implemented by the instance.

Another simplification that is made, is splitting functions in to multiple parts. This makes writing proofs easier and reading them more understandable. It also helps with reducing the original property to properties of the dependency, since the results of a function can than be stripped of their constructor. However, this comes at the cost of changing the original implementation. It is also possible to simplify function by removing optimisations. Unfortunately, this makes the implementation slower.

9 Future Work

As mentioned in section 5.1, some issues were discovered, the most important one being the compiling of instances that use auxiliary functions. This one is important because a foreign agda2hs block cannot be proven and mistakes can be made when manually porting agda code to Haskell code. An update to agda2hs that fixes this issue would be a welcome addition. Also, a change has been made to agda2hs, this change removes the type signature in where clauses when the Agda code is compiled to Haskell. This change can be improved by not removing the type signatures completely, but by only removing the explicit mention of the the parameter types that are shared with the parent function. Because, that is the source of the problem.

The verification process is not complete, so the abstract properties and general properties that follow from the abstract properties can be defined incorrectly. To make sure that the Inductive Graph Library is actually verifiable, the verification process should be finished. So, the instances that implement the abstract property classes need to be completed and the proofs of the general properties need to be finished. Also, some postulated properties are still used in this project. They should be proven to make sure that the implementation is correct.

Some functionality and optimisations have been left out of the Inductive Graph Library. Namely the show, read and the subgraph functions. The show and read functions are not required for the verification process and the subgraph depends on IntSet which makes the implementation more efficient, but in the Agda version the function is converted to only depend on lists. To make the Inductive Graph Library like the original Haskell implementation these issues need to be solved.

The PatriciaTree and the IntMap were not originally part of this project, so a simplified version has been implemented. The PatriciaTree has a lot of correspondents with the original Haskell implementation but lacks some optimisations. The IntMap is based on lists which is inefficient compared to the original implementation, which is based on a binary search tree and lacks some functionality. To make the PatriciaTree perform efficiently in Agda, these issues need to be addressed.

An open-source repository with common libraries and their proven properties could save a lot of time, since not every dependency needs to be implemented again. agda2hs already supplies a great part of the Haskell prelude, but does not supply their properties. A great start would be the common libraries in the Data library of Haskell, since a lot of libraries depend on it, including the Inductive Graph library. The code in this research project could also be apart of it, since it already implements the Inductive Graph Library and contains some functions of the Data libraries.

References

[1] Ana Bove and Peter Dybjer. *Dependent Types at Work*, pages 57–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[2] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552593, 2013.

[3] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 143155, New York, NY, USA, 2011. Association for Computing Machinery.

[4] J. Rudloff E. Brady. Idris. Retrieved June 13, 2021, from `https://www.idris-lang.org/index.html`.

[5] M. Erwig. A functional graph library for haskell. Retrieved May 30, 2021, from `https://web.engr.oregonstate.edu/~erwig/fgl/haskell/`.

[6] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[7] Bakst et al. liquidhaskell. Retrieved June 13, 2021, from `https://ucsd-progsys.github.io/liquidhaskell-blog/`.

[8] Norell et al. Agda. Retrieved May 30, 2021, from `https://agda.readthedocs.io/`.

[9] Norell et al. agda2hs. Retrieved May 30, 2021, from `https://github.com/agda/agda2hs`.

[10] W.H. Howard. To h.b. curry: Essays on combinatory logic, lambda calculus, and formalism. 1980.

[11] A. Spector-Zabusky S. Weirich, J. Breitner. hs-to-coq. Retrieved June 23, 2021, from `https://hs-to-coq.readthedocs.io/en/latest/`.

[12] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. *CoRR*, abs/1711.09286, 2017.

[13] The Coq Development Team. The coq proof assistant. Retrieved June 23, 2021, from `https://coq.inria.fr/`.

[14] van der Jeugt et al. Haskell. Retrieved May 30, 2021, from https://www.haskell.org/.

[15] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: Experience with refinement types in the real world. *SIGPLAN Not.*, 49(12):3951, September 2014.

[16] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282. ACM, September 2014.

A Code Simplifications

```
private
    matchGrLookUp' : Node -> GraphRep a b -> Maybe (Context' a b) -> Decomp GraphRep a b
    matchGrLookUp' n g Nothing = (Nothing , g)
    matchGrLookUp' n g (Just (p , label , s)) = let g1 = IM.delete n g
                                                    p' = IM.delete n p
                                                    s' = IM.delete n s
                                                    g2 = clearPred g1 n s'
                                                    g3 = clearSucc g2 n p'
                                                in (Just (_,_,_,_ (toAdj p') n label (toAdj s)) , g3)

    matchGrLookUp : Node -> GraphRep a b -> Maybe (Context' a b) -> Decomp PatriciaGr a b
    matchGrLookUp n g cxt = second λ( x -> Gr x) (matchGrLookUp' n g cxt)

matchGr : {a : Set} -> {b : Set} -> Node -> PatriciaGr a b -> Decomp PatriciaGr a b
matchGr node (Gr g) = matchGrLookUp node g (IM.lookup node g)

private
    patriciaMerge' : Context a b -> PatriciaGr a b -> GraphRep a b
    patriciaMerge' (_,_,_,_ p v l s) (Gr g) = let nppreds = fromAdjCounting p
                                                np = fst nppreds
                                                preds = snd nppreds
                                                nssuccs = fromAdjCounting s
                                                ns = fst nssuccs
                                                succs = snd nssuccs
                                                g1 = IM.insert v (preds , l , succs) g
                                                g2 = addSucc g1 v np preds
                                            in addPred g2 v ns succs

patriciaMerge : Context a b -> PatriciaGr a b -> PatriciaGr a b
patriciaMerge cxt g = Gr (patriciaMerge' cxt g)

private
    insNodeSimple' : LNode a -> GraphRep a b -> GraphRep a b
    insNodeSimple' vl intMap = IM.insert (fst vl) (Map [] , (snd vl) , Map []) intMap

insNodeSimple : LNode a -> PatriciaGr a b -> PatriciaGr a b
insNodeSimple (v , l) (Gr intMap) = Gr (insNodeSimple' (v , l) intMap)

```

```
insNodesSimple : List (LNode a) -> PatriciaGr a b -> PatriciaGr a b
insNodesSimple lns g = (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNodeSimple lns) g

propInsNodeInsNodeSimpleEquiv : (ln : LNode a) -> (gr : PatriciaGr a b) -> insNode ln gr ≡ insNodeSimple ln gr
propInsNodeInsNodeSimpleEquiv (n , l) (Gr (Map [])) = refl
propInsNodeInsNodeSimpleEquiv (n , l) (Gr (Map (m  ms))) = refl

propInsNodesInsNodesSimpleEquiv : (lns : List (LNode a)) -> (gr : PatriciaGr a b) -> (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNode lns) gr ≡ (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNodeSimple lns) gr
propInsNodesInsNodesSimpleEquiv [] (Gr (Map ms)) = refl
propInsNodesInsNodesSimpleEquiv (ln@(n , l)  lns) gr@(Gr (Map ms)) =
  begin
    (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNode (ln  lns)) (Gr (Map ms))
  =<>
    (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNode lns) (Gr (Map ((n , (Map [] , l , Map []))  ms)))
  =< propInsNodesInsNodesSimpleEquiv lns (Gr (Map ((n , (Map [] , l , Map []))  ms))) >
    (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNodeSimple lns) (Gr (Map ((n , (Map [] , l , Map []))  ms)))
  =<>
    (foldMap {{ iFoldableList }}{{  MonoidEndo }} insNodeSimple (ln  lns)) (Gr (Map ms))
```