Modelling cyclic structures in Agda

Evaluating Agda's coinduction through modelling graphs

Faizel Mangroe
Supervisor(s): Jesper Cockx, Bohdan Liesnikov
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Faizel Mangroe
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Diomidis Spinellis

Abstract

Graphs are a widely used concept within computer science. Modelling graphs can be done in various ways, but the most popular approach is doing so inductively. When graphs contain cycles modelling them becomes less intuitive. A solution for this is using the dual of induction called coinduction, which has not been as well researched as induction. In this paper I explored the possibilities and limitations of coinduction in Agda by modelling graphs using coinduction. I looked at the struggles I encountered while coding in Agda. I also provide implementations of the graphs encodings. Suitability of the encodings is determined through experiments, in which properties about graphs are proven. Both guarded coinduction and musical coinduction were successful in all of the experiments. Creating an implementation using sized types was not successful. The main improvements I identified are concerning the ease of use for a new user of Agda. I recommend improving the documentation as well as the clarity of the error messages.

1 Introduction

Graph theory plays an important role in a multitude of different fields such as, but not limited to, the field of chemistry, sociology, biology, operations research, and even war science and is widely used in computer science as well [SVE10]. Therefore, it is no surprise that there are a multitude of different ways to model graphs in programming languages. The simplest representation of graphs is through a list of pairs to represent the edges of a graph. This is demonstrated by King in his work about graph algorithms in functional programming [Kin96]. You could also choose to model a graph by creating a function that returns the adjacent vertices using an index. Another popular approach is through an inductive way of modelling graphs [Erw01]. However, the inductive approach becomes less intuitive when the graphs contain cycles. Cycles occur when a substructure occurs on a path down from the root [Ham+06].

Instead of using induction to model the graph, we can also use the dual principle of induction, coinduction. When using interactive proof assistants, such as Agda, to reason about graphs that demonstrate cyclic behaviour it is more intuitive to use a coinductive approach in modelling the graph. Viewing graphs as coinductive structures is not as well researched as the inductive variant. Nevertheless, there are works exploring this approach in programming languages like Agda. One of such works is by Celia Picard and Ralph Matthes who managed to create a coinductive representation of graphs in the functional language and proof assistant Rocq [PM11]. Another paper by Donnacha Oisín Kidney and Nicolas Wu also demonstrates an approach of representing graphs in the coinductive style [KW25]. This approach uses the library cubical Agda, which also extends the type checker.

I will attempt to address the following question:

- What are different encodings of graphs in Agda and which are suitable for the proof assistant?
- What properties of graphs can be proven using the various encodings (e.g. has s-t path, has node)?
- What improvements should be made to Agda in order for it to handle modelling with coinduction more easily?

I will do this by first trying to replicate the results of Picard's and Matthes' paper in Agda, and then translating these results to the other flavours of coinduction. The goal of this paper is to evaluate the process of modelling graphs through coinduction and analysing the possibilities and limitations of coinduction in Agda. I also provide concrete implementations of the found encodings and proofs of the graph properties, as well as listing the struggles I encountered while trying to model graphs in Agda.

I will start by analysing how responsible engineering applies to my research in section 2. In section 3 the necessary background information will be discussed. Following this in section 4 I will go over the different encodings I created using the various methods of coinduction. In this section I will provide the implementation as well. Afterwards I will shed light on the experiments I ran using the different encodings in section 5. In section 6 I will give a summary of the difficulties I encountered while coding in Agda. I will conclude with a conclusion and some future work in section 7.

2 Responsible Research

In this section I will be discussing three topics concerning responsible research:

- The use of generative AI
- The clarity of what is proven by a proof
- Replicability & Reproducibility

2.1 The use of generative AI

Generative AI can be a helpful tool in coding and research. However, when using such LLMs for instance to create ideas it is important to properly fact check and attribute these ideas within a research. During my research I have made use of generative AI for a couple of cases:

- To help translate Rocq code to Agda
- To interpret error messages
- To clarify the syntax of Agda
- To help with latex

I tried to limit my use of AI throughout the research and only resort to using AI when I could not produce an answer through the conventional route of searching for information. In my case I could easily verify the validity of the responses generated by the LLM. I could apply the explanations to my code or use the intuition I gained to create code and run the type checker to see if the program would run. In Appendix A I listed the prompts I used to gather information trough generative AI.

2.2 The clarity of what is proven by a proof

In my research I intend to prove properties of graphs using the encodings I create. So, it is very important to know exactly what a particular proof is actually proving as to not create misunderstandings regarding the capabilities of the encodings. An example of this is how I created a function that automatically generates a proof when provided a trace, a target and a node. I first believed this function to prove that the target is present in a particular graph. However, this was not the case. The function actually proves that the trace provided results in a valid path to the target in a particular graph. This is why it is important to know what a program is proving, such that the capabilities of the encodings are not misrepresented.

2.3 Replicability & Reproducibility

When researching a topic it is very important that the results presented and acquired are reproducible. The research does not serve the public as it should, if the results can only be created by the author. So in order to make sure that my research is completely reproducible and replicable by another individual I have taken the following steps:

- **Availability:** all of the code presented and additional code used during the research is publicly available in the repository.
- **Test of time:** in order to make sure the code can be ran in the future, I have added specifications of the software used within the repository in the README.md file.

3 Background

In this section I will go over the necessary prior knowledge and concepts expected throughout the paper. I will begin with some brief definition of a graph and list some properties of graphs, that can be used to reason about graphs. Afterwards I will touch upon the three different flavours of coinduction within Agda.

3.1 Graphs

In order to create an encoding for a graph we first need to understand what a graph is.

"...a graph G is simply a way of encoding pairwise relationships among a set of objects: it consists of a collection V of nodes and a collection E of edges, each of which "joins" two of the nodes. [KT06]"

In order to make it a little less abstract I will give you an example. The Dutch railway network can be represented as a graph where all stations are nodes and the tracks connecting the stations are the edges. There are various categories of graphs. You could talk about infinite and finite graphs. Graphs without cycle and graphs that contain cycles. It is also important to make a distinction between directed and undirected graphs because some graph algorithms depend on this characteristic. Trees can also be seen as graphs, but in this paper I will not be considering modelling trees since they do not contain cycles. The graphs I will discuss will be infinite graphs that contain cycles and are directed, unless explicitly stated otherwise.

Graphs can have certain properties and I will be using the following subset to test the different encodings I propose:

- isPresent: when provided a node and a graph, it tells us whether the node is present in this graph.
- hasCycle: when provided a graph, it tell us whether there exists a cycle in this graph.
- hasPath: when provided a start node, a goal node and a graph, it tells us whether the goal node can be reached from the start node in this graph.

I chose isPresent as this can be seen as the most simple property of a graph and a lot of other properties can be defined in terms of this property. I chose the cycle property since ultimately I want to reason about cyclic structures to evaluate the use of coinduction in Agda. Lastly, I chose hasPath as this is a property that represents one of the most fundamental problems within graph theory, path finding.

3.2 Agda coinduction

Agda is a pure, total, dependently typed, functional programming language. Agda being pure means that every function defined in this language behaves like a mathematical function, so when you call it with the same input you are guaranteed to get the same outputs [Stu16]. Secondly, Agda is a total language this means that every function made in Agda must terminate. Lastly, Agda is also dependently typed. This means that a type can be indexed by the objects of another type. As a consequence Agda can be used as a proof assistant. Agda has an interactive mode that can help you identify the goals of a clause containing holes. These holes can be refined or resolved interactively. An in depth example of this can be found in Agda's documentation [Tea24a].

There are three flavours of coinduction in Agda: Guarded, Musical and Sized.

3.2.1 Guarded coinduction

Guarded coinduction makes use of coinductive record types to define a type in terms of the destructors of that data type. An example taken from the documentation of Agda is the Stream type [Tea24b].

```
record Stream (A : Set) : Set where
  coinductive
  field
    hd : A
    tl : Stream A
```

The stream is defined by its fields hd and tl which represent the head and the tail of the stream. Using these fields you can now use copatterns to create Streams. Copatterns are a nice way to define and reason about infinite structures [CA18]. In the example the stream is defined by matching on the destructor copatterns [And+13] of the stream, hd and tl.

```
repeat : {A : Set} (a : A) → Stream A
hd (repeat a) = a
tl (repeat a) = repeat a
```

The stream repeat is defined by defining what is returned when applying the destructors to the stream. In this case applying hd to the repeat function returns the element a of type A and the tl function returns the stream itself. In other words repeat is a stream that repeats a given element an infinite amount of times.

3.2.2 Musical coinduction

The second flavour of coinduction is musical coinduction. This form of coinduction is deemed the old way and is deprecated. Its notation uses three symbols, namely ∞, $\sharp$, and $\flat$. Coinductive occurences in the type are labelled with the delay operator ∞. The ∞ suspends the evaluation of the type it is placed in front of. The operator comes with two functions, namely $\sharp$ and $\flat$. They are called delay and force respectively. The sharp function wraps the input into a suspended type and the flat function does the opposite. The musical equivalent of a stream taken from the Agda standard library is defined as follows.

```
data Stream (A : Set a) : Set a where
  _::_ : (x : A) (xs : ∞ (Stream A)) → Stream A
```

Here, you can see how the ∞ is used to suspend the stream type in the type signature of the constructor. It makes sure we do not try to evaluate the whole infinite stream at once.

```
zipWith : (A → B → C) → Stream A → Stream B → Stream C
zipWith _·_ (x :: xs) (y :: ys) = (x · y) :: ♯ zipWith _·_ (♭ xs) (♭ ys)
```

In the zipWith function you can see the use of the force and delay functions. It uses the delay operator on the zipWith call to make sure it is delayed, in other words it tells Agda the evaluation will be done later. Since the zipWith function itself takes stream types as input parameters the force operator is used to unwrap the suspended stream data types into stream types.

3.2.3 Sized types

The last flavour uses sized types to implement coinduction. Sized types are types that have sizes added to them indicating how big they are or how deep they go. The syntax is mostly the same as the guarded variant except that the extra Size notation is added. An example from the documentation is a representation of decidable languages as infinite trees [Tea24c].

```
record Lang (i : Size) (A : Set) : Set where
  coinductive
  field
    ν : Bool
    δ : ∀{j : Size< i} → A → Lang j A

open Lang
```

The parameter i now indicates the size of the type. The second field will now return a language of type Lang j A where j is of type Size< i. The type Size<_ ensures that j is

strictly smaller than i. There are other operators for sized types such as ∞ that indicates that something is of infinite size, which means that the structure has infinite depth or has infinite values. The $\uparrow$ indicating a type of size one bigger than the other. Sized types help the termination checker by making the size of types explicit.

4 Encodings

In this section I will go over the steps I have taken in finding the graph encodings. I will start by discussing my first ideas and the results of this intuition. After this I will continue with the encodings I ended up with using guarded coinduction and musical coinduction. I will also give an example on how to use the encodings. Finally, I will discuss the results of the sized type encoding.

4.1 A first intuition

My first intuition was to look at the literature available. I started by analysing the paper by Picard and Matthew [PM11] and following along with the steps they took in creating the coinductive graph encoding in Rocq. I did this to see if Agda's termination checker would behave similarly to Rocq's. The first idea mentioned in the paper was to use a normal adjacency list representation of a graph. I tried this and this type checked, as expected from the results of the paper. The paper then mentions that the problem with this encoding arises when trying to use a map function on graph, since the termination checker would not allow for such a call. This call is not guarded because the co-recursive call in the implementation is an argument of the map function. The map function was under a constructor, but this is too indirect to satisfy the guardedness conditions in Rocq. For good measure I tried to define a map function over the graph, but indeed the termination checker in Agda complained for the same reason as in Rocq. I wanted to use the map function in order to determine the presence of a label in the nodes of the graph. However, since this map function did not compile I needed a different approach. The paper then continues with changing the list representation of the adjacency list in the graph. They define an ilist as a function that returns a node given an index. They claimed that this will not violate the termination constraint in Rocq. I once again followed along with the steps provided in the paper and translated them to Agda code. However, the termination checker in Agda still did not accept the mapping function over the new ilist representation of the adjacency list. Even though the representation of the adjacency list changed the co-recursive call still does not satisfy the guardedness condition in Agda. This is where I took a step back and after some insightful discussions with one of my peers, I came to the conclusion that the original classical adjacency list representation might still work.

4.2 Guarded encoding

I defined the graph using a coinductive record type modelled after the classical adjacency list representation mentioned in the previous subsection.

```
record Node : Set where
  coinductive
  constructor g
  field
    cur : N
    adj : List Node
```

As presented in the code snippet, the coinductive record has a cur field and an adj field. The cur field acts as a label and for testing purposes I gave it the type Nat. The adj is where the cyclic behaviour occurs and is a list of Node types. In order to now check the suitability of the encoding with the proof assistant, I tried defining a data type that encodes the presence of a label in a graph. The first version I came up with after the discussion with my peer was a data type called isPresent that takes as input a Nat and a Node and has a constructor called node-present that takes these input parameters and checks if the provided Nat equals the label of the provided Node. If it does, it returns the data type.

```
data isPresent : N → Node → Set where
  node-present : {n : N} {g : Node} → cur g ≡ n → isPresent n g
```

This data type successfully captures the presence of a label in the cur field of a node. However, as you probably already noticed, it overlooks the case where the label is inside a node within the adjacency list. My first intuition was to somehow use a map functionality to check for node-present inside the adjacency list, but I already mentioned that this intuition did not lead anywhere. After some research, I stumbled upon the mutual functionality of Agda. This keywords allows for type, record, or function definitions that depend on each other. With this functionality I defined the adj-present constructor of the isPresent type.

```
mutual
  data isPresentInList : N → List Node → Set where
    here : {n : N} {g : Node} {gs : List Node} →
                          isPresent n g → isPresentInList n (g :: gs)
    there : {n : N} {g : Node} {gs : List Node} →
                          isPresentInList n gs → isPresentInList n (g :: gs)

  data isPresent : N → Node → Set where
    node-present : {n : N} {g : Node} → cur g ≡ n → isPresent n g
    adj-present : {n : N} {g : Node} → isPresentInList n (adj g) → isPresent n g
```

In order to evaluate whether a label is present in a node within the adjacency list, I defined another type that checks explicitly if a label is present in a list of nodes. The here constructor then looks at the head of the list and checks for isPresent given the head of the list and the label provided, which makes the types mutually dependent. When isPresent returns successful the here constructor returns an isPresentInList type with the label and

the list of nodes with the head. If it is not present in that node it has the there constructor to go into. There the head is omitted and a recursive call is made on the isPresentInList type with the same label and the list excluding the head. Now using this type the adj-present can evaluate if the node is present in the adjacency list of the graph. Using these mutual data types I was able to write manual proofs that reason about the presence of a Nat label in a graph, showing that the encoding is suitable for the proof assistant.

4.2.1 Example graphs

For this encoding, the destructor copatterns, cur and adj, can be used to define the graphs as follows:

```
mutual
  g1 : Node
  g1 .cur = 1
  g1 .adj = g0 :: []

  g0 : Node
  g0 .cur = 0
  g0 .adj = g1 :: []
```

```
g2 : Node
g2 .cur = 5
g2 .adj = ( g 3 [] ) :: ( g 7 ((g 4 []) :: []) ) :: []
```

```
g3 : N → Node
g3 n .cur = n
g3 n .adj = g3 ( suc n ) :: []
```

The nodes g0 and g1 have each other in their adjacency list. This creates a graph with cycle containing g0 and g1. Another way to create a graph is to use the constructor, but this can only be used on the right hand side of the definition, as shown in g2. You can also make a function that takes arguments to create a graph. g3 is a an infinite graph starting from a specified label counting up.

4.3 Musical encoding

After finishing the guarded variant of the graph encoding, I shifted my focus to the musical option. The way I approached this problem was by simply translating the guarded version to the equivalent musical notation. At first I ended up with a mix of musical and guarded notation, since I had a coinductive record that used the musical operators in its fields. After revising this intermediate solution, I refined it to only contain the musical notation and got the following result.

```
data Node : Set where
  g : N → (List (∞ Node)) → Node
```

Comparing the musical notation and guarded variant, it is abundantly clear that the musical option is way more concise. It is merely a data type with one constructor that takes a label of Nat type and a List of suspended Nodes and returns a Node. Now I needed to evaluate if this encoding was suitable for the proof assistant, which led to the more complicated part of the conversion. Converting the mutual data type for the isPresent property.

```
mutual
  data isPresent : N → Node → Set where
    node-present : {n m : N} {adj : (List (∞ Node))} →
                                  n ≡ m → isPresent n (g m adj)
    adj-present : {n m : N} {adj : (List (∞ Node))} →
                                  isPresentInList n (adj) → isPresent n (g m adj)

  data isPresentInList : N → (List (∞ Node)) → Set where
    here : {n : N} {x : (∞ Node)} {xs : (List (∞ Node))} →
                                  isPresent n (♭ x) → isPresentInList n (x :: xs)
    there : {n : N} {x : (∞ Node)} {xs : (List (∞ Node))} →
                                  isPresentInList n xs → isPresentInList n (x :: xs)
```

Since I wanted to use the members of the Node type to use them in the evaluation, I could no longer just give the node as is to the constructors of the data type. Like how I did in the guarded case, because in contrast to that case I do not have access to functions that destruct the node. So in order to still be able to use to adjacency list and the label, I indexed on them in both constructors of the isPresent type. This allowed me to use the members of the Node type. The only thing I had to keep in mind was that I had to reconstruct the node with its constructor when returning the isPresent data types. The second thing I had to handle was the suspended Node types and the sharp and flat operators. I needed to make all the nodes in the adjacency lists suspended and wherever I wanted to evaluate a node from these lists I had to use the force operator. It turned out the only place I needed this was in the here clause of the isPresentInList type. After finishing this it was time to check if I could reason about the graph using this data type. This turned out to be successful as well, I give an overview of the results of the experiments in section 5.

4.3.1 Example graphs

For this encoding the constructor of the data type can be used to create a graph. I created the same graphs mentioned in section 4.2.1. Since the adjacency list contains delayed nodes all of the nodes in the adjacency lists must be preceded with a ♯ symbol.

```
mutual
  g1 : Node
  g1 = g 1 ((♯ g0) :: [])

  g0 : Node
  g0 = g 0 ((♯ g1) :: [])
```

```
g2 : Node
g2 = g 5 ( (♯ ( g 3 [] ))
        :: (♯ ( g 7 ((♯ (g 4 [])) :: []) )) :: [] )
```

```
g3 : N → Node
g3 n = g n ((♯ (g3 (suc n))) :: [])
```

4.4 Sized type encoding

Lastly, I looked into creating a sized type translation of the encoding I presented. I approached this by reading one of the referenced papers [Abe16] from the documentation [Tea24c] and using the intuition I developed from it. However, when thinking about how to

convert the encoding and where to include the size parameter, I came to the conclusion that implementing sized types does not buy us any more power than we already have. I came up with three possible positions to use the size parameter. The first being inside the definition.

```
record Node (i : Size) : Set where
  coinductive
  constructor g
  field
    cur : N
    adj : ∀{j : Size} → List (Node j)
```

Now the node has a size i and the nodes in the adjacency list have a size j. I used different letters because you can not make the assumption that they are of the same depth. However, you can also not make the assumption that the nodes in the adjacency list are of a lesser size compared to size i. This is because of the possibility of a cycle occurring somewhere down the line. So, this does not particularly provide any new information to the termination checker in any way. A node has some size and the nodes in the adjacency list also have a size that is not related to the first size. When converting the rest of the code to sized types where necessary the program still type checks, but in terms of power we did not gain anything.

Another way was to add the size to the List type of the adjacency list, but you might notice quickly that this does also not change anything for the node itself. Now, what if we introduce a wrapper around the graph and have this contain a size. This wrapper contains all the nodes of a graph. The only thing we gained with this, is being able to explicitly define that a graph is infinite or finite. With the regular guarded encoding all graphs are potentially infinite. We could make finiteness explicit in the guarded case by having a node that contains all other nodes and all the other nodes can not refer to this "parent" node. So, in principle we still have not found anything new we can do when including size.

In conclusion, I do not believe that sized types give any extra benefits as opposed to musical or guarded coinduction, considering the adjacency list model, based on my intuition.

5 Proving graph properties on the encodings

In this section I will discuss the creation of the other data types for the properties in order of guarded and musical. I will also give an overview of the results from the experiments. Lastly, I will go into the creation of an automatic proving algorithm.

5.1 Guarded

Aside from the isPresent property there are two other properties I used to reason about the graph encodings. The first being hasCycle and the second being hasPath. While coming up with the design of these data types I noticed that both data types can be described in terms of the isPresent data type. For hasCycle it was rather straightforward. The data type takes a node as input and checks if that same node is present in the adjacency list of the same node. In the case that the provided node is not part of a cycle, the nodes in the adjacency list can be checked for a cycle with the CycleInList data type through adj-cycle. In a similar way the isPresent data type checked for presence, this mutual data type checks for the presence of a cycle.

```
mutual
  data CycleInList : List Node → Set where
    here : {g : Node} {gs : List Node} → hasCycle g → CycleInList (g :: gs)
    there : {g : Node} {gs : List Node} → CycleInList gs → CycleInList (g :: gs)

  data hasCycle : Node → Set where
    node-cycle : {g : Node} → isPresentInList (cur g) (adj g) → hasCycle g
    adj-cycle : {g : Node} → CycleInList (adj g) → hasCycle g
```

For the hasPath type it could be defined as a type with a constructor that takes two labels one for the source and one for the target and a node that represents the graph. Now with these inputs it checks if label t is present in the graph that has label s. If this is the case it means that a path from s to t exists in the initial graph so the data type hasPath can be returned. However there is one complication in this description. How do we get the graph that has s as a label?

5.1.1 Graph functions on the encodings

In order to answer the question, I needed to find a way to get the node with the correct label from a graph. In non-functional languages this would have been almost trivial as we could use any search algorithm and return the graph when found. However, in functional languages this is a bit more complicated due to the immutability of structures. So, the question boils down to creating a search algorithm that returns the path to a node.

I created a fuelled depth first search algorithm that returns a pair type of Boolean and List of Nat, which indicate whether the target is in the graph and if so what the path is to this target from the initial graph. The dfs algorithm takes a maximum depth of how deep it is allowed to search. This is necessary for the termination checker, because without this depth you can not be sure if the dfs reaches an end or loops forever. It takes a target, the intermediate result of the path and a graph. This algorithm will return a path if the target is found in the graph within a certain depth.

So now we have our path we needed for finding the specific graph. However we are not finished yet. We still need to retrieve the node itself. So for this I defined a function findGraph that given a path and a node it returns the node at the end of following this path.

Now we can use these functions in the definition of the hasPath type.

```
data hasPath : N → N → Node → Set where
  path : {s t : N} {g : Node} → isPresent t (findGraph (proj₂ (dfs 10 s [] g)) g)
                              → hasPath s t g
```

5.2 Musical

After finishing the guarded case I once again shifted my focus to translating it to the musical variant. The two data type definitions stayed almost the same except for the indexing change that was mentioned in the section explaining the conversion of the isPresent type and the addition of a suspended node where necessary.

Something interesting to note is that the previously mentioned mapping function, that did not work for the guarded encoding, does work for this encoding. I believe this to be

the case because the sharp guards the co-recursive call, allowing it to pass the termination checker. However, I was not able to apply my initial intuition of mapping over the adjacency list with this function and the node-present function to check for presence.

5.3 Results

I tried creating manual proofs, which have been added to appendix B, for each property in each encoding and the result of this can be summarised as follows:

	Guarded	Musical	Sized Types
isPresent	✓	✓	✗
hasCycle	✓	✓	✗
hasPath	✓	✓	✗

Table 1: Results of suitability of different encodings in proving properties about graphs.

5.4 Automatic proofs

In Agda you can also create functions that for the appropriate input return a proof. I tried implementing this for the isPresent property, but was not able to do this in the sense that the function proves the presence of a label in a graph. While coding I could not come up with an algorithm that could traverse the graph and search for the label while also keeping track of the intermediate results. As mentioned I already had a dfs algorithm that returns a path to the target, if it is present. With this input I was able to create a function that returns an isPresent data type. However, it is important to note that this function does not constitute to a function that proves the presence of a label in a given graph. It proves that for a certain path and a certain node from a graph, the path reaches the target node.

6 Improvements to Agda

In this section I will go over some difficulties I encountered while programming in Agda and the potential corresponding improvements to improve the support for coinduction in Agda and coding in general.

6.1 Documentation

Agda has an online documentation that contains definitions and some explanations of concepts that can be used in Agda. However, the documentation does not provide enough information in my opinion. An example would be the page for sized types. This page has a hyperlink to the different operators used for sized types, but these operators are not explained. This requires a user to find different resources to figure out how to use sized types.

6.2 Error messages

While programming in Agda I encountered various error messages. More often then not, reading these error messages did not help me resolve the problems within my code. They would usually confuse me more. This made debugging the code a cumbersome task. I believe user would greatly benefit from error messages that help the user identify the issue more concretely and clearly.

6.3 Coinduction

As for using coinduction, I did not encounter any real limitations in the context of creating the graph models. I mentioned that my first intuition of using the map function was blocked, because the termination checker did not allow the co-recursive call to be within the map function. However, I do not think I can list this a limitation, because the call itself indeed could loop forever. The only other difficulty I encountered with coinduction is using sized types. This is mainly because I could not find suitable resources to help me use this concept.

6.4 Coverage checking

When I created the algorithm that returns a proof of type isPresent, there were certain cases that would never be reached due to how the input was formatted. However, the coverage checker of Agda complained that I did not include these cases in the definition of the functions. I think that it would be nice to have a pragma like the non-terminating pragma to make explicit that the coverage checker does not need to worry about a particular function. The current solution for this in my case would be to return a Maybe type.

7 Conclusions and Future Work

In my research I tried to encode a graph using the different flavours of coinduction (guarded coinduction, musical coinduction, and sized types). I used the adjacency list representation of graphs as a basis for all the encodings. These encodings were used in experiments, in which I proved properties of graphs. In doing so I encountered various struggles concerning programming in Agda and using coinduction.

Both guarded coinduction and musical coinduction were successful in all of the experiments. Even though musical coinduction is deprecated, it seems to provide the most power. The musical notation opposed to the guarded variant was less strict with its guardedness conditions. The sized type implementation did not yield any results, which could be partly due to the lack of resources available concerning this method.

While programming in Agda I encountered various struggles mainly concerning the understandability of the programming language. The support for coinduction and Agda in general would be greatly benefitted by a more extensive documentation of the language. Another point of improvement my research identified is increasing the clarity of the error messages. These improvements would help to flatten the learning curve and ease of debugging for new users.

Future work regarding coinduction in Agda could use my research as a starting point for improving coinduction support. I would recommend exploring the capabilities of the musical variant compared to the guarded version. Another proposal would be to further investigate the role sized types can play in coinduction.

References

[1] Andreas Abel. *Equational Reasoning about Formal Languages in Coalgebraic Style*. 2016.

[2] Abel Andreas et al. "Copatterns Programming Infinite Structures by Observations". In: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. 2013.

[3] Jesper Cockx and Abel Andreas. "Elaborating dependent (co)pattern matching". In: *Proc. ACM Program. Lang.* (2018).

[4] Martin Erwig. "Inductive graphs and functional graph algorithms". In: *Journal of Functional Programming* (2001).

[5] Makoto Hamana et al. "Representing Cyclic Structures as Nested Datatypes". In: *Presented at Trends in Functional Programming* (2006).

[6] Donnacha Oisín Kidney and Nicolas Wu. "Formalising Graph Algorithms with Coinduction". In: *Proc. ACM Program. Lang.* (2025).

[7] David Jonathan King. "Functional programming and graph algorithms". PhD thesis. University of Glascow, 1996.

[8] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[9] Celia Picard and Ralph Matthes. "Coinductive Graph Representation: the Problem of Embedded Lists". In: *Electronic Communications of the EASST* (2011).

[10] S. G. Shrinivas, Santhakumaran Vetrivel, and N. Elango. "APPLICATIONS OF GRAPH THEORY IN COMPUTER SCIENCE AN OVERVIEW". In: *International Journal of Engineering Science and Technology* (2010).

[11] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan Claypool, 2016.

[12] The Agda Team. 2024. URL: `https://agda.readthedocs.io/en/v2.6.2.1/getting-started/a-taste-of-agda.html#agda-as-a-proof-assistant-proving-associativity-of-addition`.

[13] The Agda Team. 2024. URL: `https://agda.readthedocs.io/en/v2.6.2.1/language/coinduction.html`.

[14] The Agda Team. 2024. URL: `https://agda.readthedocs.io/en/v2.6.2.1/language/sized-types.html`.

A Use of Generative AI

This section contains a list of prompts I used to interact with LLMs:

Prompts
How can I convert a function definition from coq to Agda?
How does the with syntax work in Agda?
What does this error message mean [insert error message]?
hi i want to make a table in latex with 4 columns and 4 rows where the first row and first column is empty and the rest of the first row is filled with "guarded" "musical" and "sized types" and the rest of the first column needs to be filled with "isPresent" "hasCycle" and "hasPath". The inner cells need to be able to be filled with the colour green and a check mark for succes and a colour red and an x for failure
How to import the standard library of agda?

B Manual proofs

This section contains the manual proofs used in the experiments for each of the properties. The proofs itself are exactly the same for both the guarded variant and the musical variant.

```
- prove g2 has 5
proof1 : isPresent 5 g2
proof1 = node-present refl

- prove g2 has 3
proof2 : isPresent 3 g2
proof2 = adj-present (here (node-present refl))

- prove g2 has 7
proof3 : isPresent 7 g2
proof3 = adj-present (there (here (node-present refl)))

- prove g2 has 4
proof4 : isPresent 4 g2
proof4 = adj-present (there (here (adj-present (here (node-present refl)))))

- prove g0 has a cycle
proof5 : hasCycle g0
proof5 = node-cycle (here (adj-present (here (node-present refl))))

- prove g4 has a cycle
proof6 : hasCycle g4
proof6 = adj-cycle
  (there
    (here
      (adj-cycle
        (here
          (adj-cycle
            (here
              (node-cycle (here (adj-present (here (node-present refl))))))))))))

- prove g2 has a path from 5 to 4
proof7 : hasPath 5 4 g2
proof7 = path
  (adj-present
    (there (here (adj-present (here (node-present refl))))))

- prove g2 has a path from 7 to 4
proof8 : hasPath 7 4 g2
proof8 = path (adj-present (here (node-present refl)))
```