Encoding Finite State Automata in Agda using coinduction

Evaluating the support for coinduction in Agda

Noky Soekarman[1]
Supervisor(s): Jesper Cockx[1]**, Bohdan Liesnikov**[1]
[1]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: Noky Soekarman
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Bohdan Liesnikov, Diomidis Spinellis

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

The proof assistant Agda supports coinduction, which can be used to reason about infinite and cyclic structures. The possibilities and limitations of using coinduction in Agda are not well known. To better understand these, I will implement Finite State Automata and their equivalence in Agda. Finite State Automata (FSA) is an example of a cyclic structure. FSA are an introductory model in computation theory, and can be used text processing and hardware design. Equivalence of two FSA is used in software and hardware verification. I created various encodings for FSA and prove equivalence between two deterministic FSA for each of them. At the end, I compared them and see whether they are limited by the support for coinduction in Agda.

1 Introduction

Proof assistants, such as Agda, are used to "define properties and do logical reasoning"[Geu09] on mathematical theories. They do not automatically create proofs, "... user input is always required. But, depending on the assistant, certain tasks are automated." [Sch14]. Agda achieves the creation of formal proofs by using dependent types and strong typing [Tea24a]. Agda also has support for coinduction, the dual to well known principle of induction.

In computer science, coinduction is used to reason about infinite structures such as infinite lists and trees [KS16], but also cyclic structures such as Finite State Automata (FSA) [San11]. FSA are an introductory model to the theory of computation. They make use of concepts that are also used in the more complex models in this field. FSA themselves can be used in various applications, from text processing to hardware design [Sip13]. An example of an FSA would be one of an automatic door system, where the doors open when a person is in front of it on either the front or rear side. This can be represented with the FSA in figure 1. The door stays closed when nothing is in front or behind it, and it opens when there is. It stays open while there is something in front or behind it, and closes again when that is no longer the case.

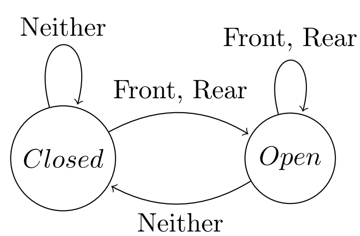


Figure 1: An example FSA for an automatic sliding door

A lot of work has already been put in the theory of Finite State Automata, but not as much into implementing them, and little to no work in implementing them in proof assistants such as Agda. I was able to find two implementations for Finite State Automata in Agda, one by Tejiščák [Tej12] and another by Ivanov [Iva20]. These two implementations are not coinductive however. These two things together lead us to the following research question: What are different ways to model Finite State Automata in Agda in a coinductive way and how to prove equivalence for them? Equivalence is a very relevant problem, as it is used in hardware and software verification [HR15]. I will explain the definition in chapter 3.

I will start by introducing every relevant topic that is used to answer this research question in chapter 3. After the topic introduction, I will present the different encodings of Finite State Automata I created in chapter 4. In the next chapter, I will show the most important experiments run on these encodings. During the development and experiments, some problems with the Agda language have presented themselves, and will be discussed in chapter 6. The final chapter will present the conclusions and suggestions for future work. Below is a list of contributions that are contained within the above mentioned chapters:

- Various encodings for FSA (chapter 4).
- Notion of equivalence for deterministic FSA.
- A set of experiments and proofs performed with the created encodings (chapter 5)
- Description of limitations with coinduction in Agda (chapter 6).

The methodology used to achieve these contributions is as follows. First, research what FSA are and what exactly has already been done, start implementing different encodings and do some testing on them to verify whether they work, implement equivalence for these various created encodings, compare them, and finally document why these encodings are valid and suitable for Agda.

2 Responsible Research

This chapter is about the practice of responsible research, how reproducible the research is, and some ethical aspects within this research.

The results of this research are fully reproducible since the full source code will be available and the specific version of Agda will be provided as well. The source code can be found on this[1] GitHub repository. The Agda version used for this research is 2.7.0.1.

For the creation of all of the code, GitHub Copilot[2] has been used to autocomplete lines for me. It did not generate full solutions. An attempt was made to use ChatGPT[3] for code generation when I was stuck or encountered some issues. ChatGPT gave suggestions on how to potentially fix the problem as an output. These suggestions were not useful however. The suggestions the AI model gave, looked like valid Agda code, but when putting it in your solution it would not compile, not solve the original problem, or create a new problem. As a results of this, no code presented in this paper is fully written by AI, only line autocompletion has been used to finish a already partially written line. The autocompletion was done to speed up the workflow.

The report writing has been solely done by myself and no AI has been involved in the writing of the text. The only time where AI was involved was in getting the highlighted Agda code, since it was easier to feed this to ChatGPT, than to compile it myself. It was easier since the Agda to LaTeX compilation sometimes said that the Agda code was not valid, while the normal type checker said that it was valid. One trick that I needed to use, was to first give it a valid example. Without this, the accuracy of the generated LaTeX was lower than with this trick. Some prompts to generate LaTeX from Agda code can be found in appendix B.

[1] https://github.com/Soek02/FSA-in-Agda
[2] https://github.com/features/copilot
[3] https://chatgpt.com/

There are no other ethical issues since this research does not involve any participants, and the results of the research does not have any ethical implications either.

3 Background

In this chapter, the two necessary pieces of background information will be presented. Firstly a description and definition will be given for Finite State Automata. Secondly, there will be a description on how to use coinduction in Agda with some examples.

3.1 Finite State Automata

The following section summarizes the concept of Finite State Automata described by Sipser [Sip13, chapter 1.1-1.2].

Finite State Automata are used to decide regular languages; they do this by consuming one element at a time from the input. A Finite State Automaton consists of states and transitions between those states. Each state can either be accepting or rejecting, and every state needs to have transitions for each element in the alphabet. When consuming an element, the automaton follows the corresponding transition to end up in a new state. If there are no more elements to be consumed, the automaton accepts if the current state is accepting, else it rejects. A more formal definition would be:

A finite (state) automaton is a 5-tuple (Q, Σ, δ, q_0, F), where

- Q is a finite set of states.
- Σ is a finite set of symbols called the alphabet.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept states. In a visual representation, these states will have a double border.

There are two types of Finite State Automata, a deterministic one and a nondeterministic one. A Deterministic Finite State Automaton always needs to have exactly one transition for each element of the alphabet to a state for every state. A nondeterministic Finite State Automaton can have multiple transitions for one element; the automaton nondeterministically chooses the transition which leads to the accept state. It can also have no transitions for the element. A way to model this, is whenever an element does not have a transition, it goes to a so called trap state, where all transitions originating from it lead back to itself. Finally, it can also have transitions which do not consume any elements, it nondeterministically chooses if this transition should be taken to reach the accepting state. Such a transition is denoted with an ϵ and is called an epsilon transition.

The formal definition of a nondeterministic Finite State Automaton is almost the same the deterministic one, except that the transition function is different. The transition function now looks as follows:

- $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition function. Where Σ_ϵ is $\Sigma \cup \{\epsilon\}$ and where $P(Q)$ is the powerset of Q.

Equality of two Finite State Automata is defined as them having the same language. This means that for every possible combination of elements of the alphabet, they need to have the same acceptance.

3.2 Coinduction In Agda

This section will explain how coinduction can be used in Agda. The Agda proof assistant allows you to create coinductive structures such as infinite lists or streams. There are three ways to construct such structures in Agda, namely using Musical, Guarded or Sized style. I will not discuss Sized style, as this deemed itself beyond the scope.

To understand what coinduction is and how it works in Agda, we first need to know the very basics of Agda and coinduction. Agda is a functional programming language which makes use of dependent types, a simple example would be the $Vec\ n$ type. This type has an extra parameter n to specify the length of the vector. This n can now be used to verify whether the given vector actually has the specified length. Agda is also a total language, this means that "... a program e of type T will always terminate with a value in T. No runtime error can occur, and no nonterminating programs can be written (unless explicitly requested by the programmer)." [Tea24a]. Coinduction is used to reason about infinite structures, such as infinite lists for example. To get values from these infinite structures, these datatypes need to have an observer or destructor present. The simplest example of these are the head and tail operators for streams. You can either get the value at the start of the stream with the head operator, or get the whole stream except the first element with the tail operator. A code example of this can be seen in figure 2. To get a notion of equality, bisimilarity can be used. Bisimilarity for infinite lists means that the values at the head of the two lists must be equal, and that their tails must again be bisimilar. [KS16]

I will now discuss the different styles of coinduction in Agda. The Musical style has three important Unicode characters to define certain behaviour, which are the infinity symbol (∞), the sharp symbol (#), and the flat symbol ($\flat$). The ∞ symbol can be used to create a new type that indicates that the value has not been computed yet, "The type ∞A can be seen as a suspended computation of type A." [Tea24b]. To create such a value of type you can use the # operator, and to force the computation of such a type you can use the $\flat$ operator. When using the Musical style, you need to manually think about where you need to delay and force the types to satisfy Agda. The official signature of both of those operators from the Agda docs is the following:

```
#_ : ∀ {a} {A : Set a} → A → ∞ A
♭  : ∀ {a} {A : Set a} → ∞ A → A
```

An example using the Musical style is the Streams data type from the standard Agda library is below in figure 2.

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

head : Stream A → A
head (x : xs) = x

tail : Stream A → Stream A
tail (x : xs) = ♭ xs

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

Figure 2: Streams in Agda using the Musical style

In this example, you can see the ∞ symbol in the definition, the ♭ symbol for computing the tail of a stream, and the # symbol in the repeat function to make sure the function does not go on until eternity. This style is not recommended any more, the next style, Guarded, is the recommended way to do coinduction in Agda.

The Guarded style works by creating a record with the "coinductive" keyword added to it. Lets first look at the streams example again, but now in Guarded style.

```
record Stream (A : Set a) : Set a where
  coinductive
  constructor _::_
  field
    head : A
    tail : Stream A
```

```
repeat : {A : Set} → A → Stream A
repeat x .head = x
repeat x .tail = repeat x
```

As you might see, the creation of a stream is different from the Musical style. Instead of directly concatenating the head to the tail of the stream, we now define them separately. The example results in an infinite stream of the given value. It starts by defining the head, which is just x, and then it defines the tail to be itself again. This method of defining the head and tail is called copattern matching [Tea24b].

3.3 Properties to prove

In this section, I will mention the properties I will be proving with the created encodings for FSA and why they are important or necessary. There will also be a mention of other considered properties.

3.3.1 Running input

The first property that I will be implementing will be running input on the automaton. The automaton then should be able to receive input and give a boolean output whether the input is in the language of the automaton or not. This is a necessary property to have to be able to prove equivalence, as the definition for equivalence is based on whether the input is accepted or not.

3.3.2 Equivalence

The second property I will be proving and implementing is equivalence for two automata. I will be doing this with bisimulation. The bisimulation will be performed on the individual states, rather than the full automaton. For two states to be bisimilar, they need to have the same acceptance, and for every transition, the resulting states should be bisimilar again. This property is chosen as equivalence is a relevant problem as described in section 1, the introduction.

3.3.3 Other considered properties

Other properties that were considered were operations such as concatenation and unification. This was left out since this was not the main goal of the research, and because of time constraints with the already existing properties. Another property that was considered, was conversion from an NFA to a DFA. However, this presented some complications and was therefore left out. The complications are presented in section 5.3.

4 Encodings

This chapter contains all the different encodings I came up with for Finite State Automata. The first section will be about Deterministic Finite State Automata, and the second section will be about the nondeterministic variant.

4.1 Deterministic Finite State Automata

This section will contain the different encodings for Deterministic Finite State Automata in different styles of representing coinductive structures in Agda. All the encodings will encode a single state, and some way to move from the current state to a next state.

4.1.1 Function transition

This version of a DFA has a state encoded with a name, a boolean if the state is accepting, and a function that receives a Symbol as an input and gives the next state as an output. The guarded variant encapsulates this in a coinductive record type. It is made coinductive using the built-in keyword 'coinductive'. Below you will find the code for this variant.

```
data Symbol : Set where
  a : Symbol
  b : Symbol
```

```
record DFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : Symbol → DFAState
```

The variant that makes use of the musical style for coinduction in Agda works slightly different. This makes use of a custom data type instead of a record. The data type only has a single constructor which takes in three parameters, a string type for the name, a boolean whether the state is accepting, and the transition function.

```
data DFAState : Set where
  state : String → Bool → (Symbol → ∞ DFAState) → DFAState
```

As you might notice, there now is an ∞ in the transition function. This is to denote that the next state has a delayed type, so it is not computed yet. This is to make sure that there will be no infinite recursion when creating new states.

Note that the Symbol datatype only has two constructors in this example. This can be expanded to any finite amount to represent the alphabet of the DFA.

4.1.2 List transition

A DFA can also be encoded with a lookup list as the transition, instead of using a function. The list consists of pairs where the first element is a Symbol, and the second element is a state. Below is the Guarded style of this version.

```
record DFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List (Symbol × DFAState)
```

And here is the Musical style of this version.

```
data DFAState : Set where
  state : Bool → List (Symbol × (∞ DFAState)) → DFAState
```

4.2 Nondeterministic Finite State Automata

This section will contain an encoding for an NFA, it will have the same structure as the DFA with the list as the transition. The encoding is in the Guarded style and can be found below.

```
record NFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List ((Maybe Symbol) × NFAState)
```

The change that makes this suitable for an NFA, is the addition of the Maybe datatype. This allows epsilon transitions to exists. This list also allows for multiple transitions for a single element of the alphabet.

4.3 Equivalence

To get a notion of equivalence for these encodings, I made use of bisimulation. For two states to be bisimilar, they need to have the same acceptence, and for every transition the resulting states should be bisimilar again. Below is a notion of bismilarity for the first DFA encoding that uses the Guarded style. For the other encodings it looks very similar, the code for them can be found in the github repository mentioned in chapter 2.

```
record _~_ (d1 : DFAState) (d2 : DFAState) : Set where
  coinductive
  field
    accept : d1 .isAccepting ≡ d2 .isAccepting
    transition : ∀ (c : Symbol) → _~_ (d1 .transition c) (d2 .transition c)
```

Using this notion of bisimilarity, I also created the following record type:

```
record _≈l_ (d1 d2 : DFAState) : Set where
  coinductive
  field
    equivLanguage : ∀ (s : List Symbol) → accepts s d1 ≡ accepts s d2
```

This record states that two (input) states have the same language, iff for all inputs, they have the same acceptance. This should only be used with the start state of an FSA, since this is where the initial input is "inserted". I also created a method that transforms a notion of bisimilarity, to this new record. So if two start states are bisimilar, then they will have the same language. This works because the states are bisimilar if they have the same acceptance, and if for all characters, the transitions lead to bisimilar states again. So for every element of the input, the acceptance of the states is goes through, will be the same.

```
bisim_to_equiv : ∀{d1 d2} → d1 ~ d2 → d1 ≈l d2
bisim_to_equiv {d1} {d2} bisim = record { equivLanguage = λ s → bisim-eq d1 d2 s bisim}
  where
    bisim-eq : ∀ (d1 d2: DFAState) (s: List Symbol) → d1 ~ d2 → accepts s d1 ≡ accepts s d2
    bisim-eq d1 d2 [ ] bisim = bisim .accept
    bisim-eq d1 d2 (x :: xs) bisim = bisim-eq (d1 .transition x) (d2 .transition x) xs (bisim .transition x)
```

5 Experiments

This chapter is about the experiments I performed using the encodings described in the previous chapter. The first set of experiments is running some input on the Finite State Automata, the second set of experiments is to try and prove equivalence using bisimilarity. After the experiments I will explain the limitations of the encodings and shortly compare them.

5.1 Running input

This subsection describes the steps that were needed to run input on each encoding. The general overview of the required steps is below.

- Define a method that takes in a state and the current input
- Define a method that correctly transitions from the current state to the next state based on the given input
- Define a method that combines these two, it receives a starting state and the input, and it outputs whether the input is accepted or not
- Create the states together with their transitions

5.1.1 Function transition

For the first encoding that has the Symbol datatype and the function with the ($Symbol \to DFAState$) signature, these were all relatively straightforward. To start with the Guarded style, there are only two methods needed for this.

```
runDFA : DFAState → List Symbol → DFAState
runDFA s [ ] = s
runDFA s (x :: xs) = runDFA (s .transition x) xs

accepts : List Symbol → DFAState → Bool
accepts input startDFAState = (runDFA startDFAState input) .isAccepting
```

The Musical style is very similar in terms of methods, the exact code can be found in appendix A.

The $runDFA$ method goes over the input symbols one by one, gives the first symbol to the current state's transition function and gets the resulting state. It recursively calls the method again with that state and the remaining input.

To create states for the Guarded style, use the copattern notation with suffix. To create states for the Musical style, make use of the special operators to get the right types according to the definition.

5.1.2 List transition

For the other encoding with the transitions in a list, the methods are slightly different to be able to lookup the correct state.

Guarded:

```
getFromList : Symbol → List (Symbol × DFAState) → DFAState
getFromList e [ ] = q_reject
getFromList e ((x , y) :: xs) = if x == e then y else getFromList e xs

runDFA : DFAState → List Symbol → DFAState
runDFA s [ ] = s
runDFA s (x :: xs) = runDFA (getFromList x (s .transition )) xs

accepts : List Symbol → DFAState → Bool
accepts input startState = (runDFA startState input) .isAccepting
```

The methods to achieve the same in Musical style are very similar, if you wan to look at them, you can find them in appendix A.

The $runDFA$ and *accepts* method still have the same function as before, there are some extra methods to accommodate for the difference in how the transitions are stored. If the definition of the transition list is not according to the definition of a DFA given earlier, so the list does not contain a transition for every element of the alphabet, it goes to a reject state. This state is rejecting and can only transition to itself regardless of the current input.

5.1.3 NFA

This subsection describes how input can be run on the encoding for the NFA. This was not as straight forward as the DFA encodings, this is due to the nondeterminism. Below is the most important code that was needed to run input on an NFA. The other methods that are needed are:

- stateInList: Check whether a given state is in the given list.
- getEpsilonStates: Gets the states which are reachable from the given state by only using a single epsilon transition.

```
{-# TERMINATING #-}
getReachableStates : NFAState → List NFAState → List NFAState
getReachableStates currentState visitedStates = if stateInList currentState visitedStates then [ ]
  else
    let
      newVisitedStates = currentState :: visitedStates
      epsilonStates = getEpsilonStates (currentState .transition)
    in
      currentState :: concatMap (λ s → getReachableStates s newVisitedStates) epsilonStates

getUniqueStates : List NFAState → List NFAState
getUniqueStates [ ] = [ ]
getUniqueStates (x :: xs) = if stateInList x xs then getUniqueStates xs else x :: getUniqueStates xs

runNFA : NFAState → List Char → Bool
runNFA currentState [ ] = currentState .isAccepting
runNFA currentState (c :: cs) =
  let
    reachable = getUniqueStates (getReachableStates currentState [ ])
    nextStates = getUniqueStates (concatMap (λ s → getFromListWithoutEpsilon c (s .transition)) reachable)
    finalStates = getUniqueStates (concatMap (λ s → getReachableStates s [ ]) nextStates)
  in
    any (λ s → runNFA s cs) finalStates
```

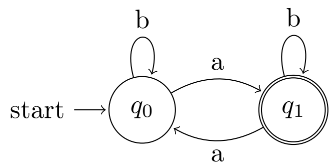
5.2 Equality

This section is about showing equivalence for the created DFA encodings using bisimilarity. To start, I present an example where I show equivalence for a DFA to itself, and after that a more complicated example where the DFA's have the same language, but do not have the same structure.

5.2.1 DFA

This subsection shows how bisimilarity is used to show equivalence of two DFA's. First the function version will be presented, and afterwards the list version.

To start, I am going to show bisimilarity for a DFA to itself. The DFA I am going to use is the following:

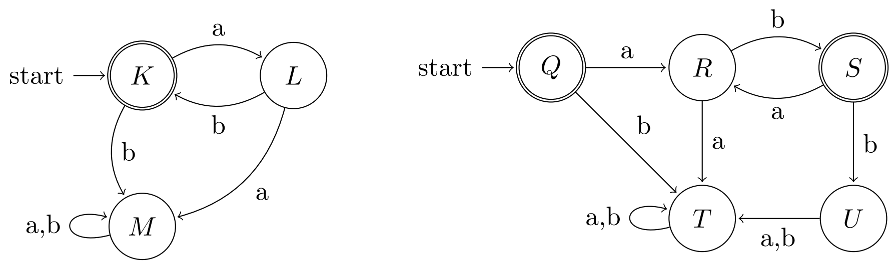


The proof for this in Agda:

```
mutual
  q0_bisim_q0 : q0 ~ q0
  q0_bisim_q0 .accept = refl
  q0_bisim_q0 .transition a = q1_bisim_q1
  q0_bisim_q0 .transition b = q0_bisim_q0

  q1_bisim_q1 : q1 ~ q1
  q1_bisim_q1 .accept = refl
  q1_bisim_q1 .transition a = q0_bisim_q0
  q1_bisim_q1 .transition b = q1_bisim_q1
```

A second, and more interesting example from the course slides of the "Languages and Automata" course from the Wellesley College is as follows [Tur10].



(a) DFA D_1 (b) DFA D_2

Figure 5: Two DFA's that accept the same language

To show that these two DFA's are bisimilar, we need to show that every state in the first one, is bisimilar to a state in the other one. The pairings used for this are presented in the table below:

Table 1: Pairings for bisimilar states in D_1 and D_2

State from D_1	Bisimilar states in D_2
K	Q, S
L	R
M	T, U

To prove this in Agda, we need to create a bisimilarity record for each pair presented in table 1. Similar to the earlier example, first define the acceptance in the record, and then for every symbol specify the next bisimilarity record. The code for the proof that K is bisimilar to Q is bellow, this will give an intuition on how to this for other states.

```
k_bisim_q .accept = refl
k_bisim_q .transition a = l_bisim_r
k_bisim_q .transition b = m_bisim_t
```

The transition field of the bisimilarity should only be defined for a and b in this case, since those are the only two symbols that occur in the DFA's. For the symbol a it is defined as the bisimilarity between L and R, since those are the states that are next when reading an a from K and Q.

Initially the list version had the $Character$ datatype, this caused problems when trying to prove equivalence for two DFA's. That is why it now has the *Symbol* datatype. A more detailed explanation will be given in section 6.2. After this change, I was able to prove equivalence for two DFA's just like the other version with the function as transition.

5.3 Limitations

This section will be about the limitations that these encodings have, and how to work with these limitations.

5.3.1 DFA

Both of the encodings for the DFA only encode a single state. When actually using these encodings, you need to be mindful of this, since running the input from the wrong state most likely influences the output. Therefore, the $runDFA$ method should only be called on the start state. For proving equivalence, you should keep the same limitation in mind. It does not make sense to prove equivalence for anything else but the start state.

A problem exists with the first encoding, that uses a function as the transition, when trying to compare two DFA's with different alphabets. You then need to alter them such that their transition functions are defined for every element of the union of their alphabets. Only then can you try and prove bisimilarity for them.

The second encoding with the list as the transition, has the limitation that there needs to exist a predefined reject state. This is needed for Agda to verify that the method to get the next state, always returns a state, even when the symbol that you are looking for might not be in the transition list. Another issue with this one is that the transition list is not forcing you to explicitly define a next state for every symbol. In that case the symbol would go the reject state. One might argue that this is still a valid DFA, since it still has a transition for every symbol. A second issue is that the transition list can be ill defined in the way that there can exist multiple entries for the same symbol. This does not lead to nondeterminism however, because of the way the next state is picked. It will always get the first entry from the list.

Some of these limitations could potentially be solved by creating a wrapper structure for the whole DFA, which would include all states, the starting state and maybe even the alphabet of the DFA. This however, would not add any extra functionality over the current solution. With the current solution, the user needs to be a bit more careful when creating and using the states.

5.3.2 NFA

The NFA encoding does not have a notion of bisimilarity at this point, this is harder than initially expected due to the presence of epsilon transition. At first I wanted to avoid these transitions by converting the NFA to a DFA and use the notion of bismilarity for DFAs. The implementation of this conversion is not working due to the fact that Agda only has immutable variables, and following the procedure described by Sipser [Sip13], it needs some mutability to my current understanding. There is however, a paper by Bonchi and Pous [BP13] to check for NFA equivalence with bisimulation.

Another limitation is that you need to define the states such that they all have a unique name. This is because the name of the state is used to check whether the state has already been visited when checking for reachable states with epsilon transitions.

5.4 Comparing versions

The two biggest differences between these two encodings is how they transition from one function to another and what input they accept. When using a function to transition from one state to another, you ensure that every transition leads to exactly one state. If you use the transition list, you can have multiple entries in the list for the same symbol which may point to different states.

A benefit of using a list over a function, is that you can iterate over the list to get the alphabet of the DFA if you don't know it beforehand. This is assuming the state is properly defined and thus has a transition for every element of the input alphabet.

There is no difference in functionality between the Guarded and Musical style for both encodings, the only difference is how you should create instances.

6 Encountered problems in Agda

This chapter describes the problems that I encountered when experimenting that are related to Agda and not the encodings itself.

6.1 Problems with NFA

The biggest problem that I encountered was to get all the reachable states by just using epsilon transitions. To be able to do this I need to keep track of the already visited nodes so that there would not be any loops when visiting the states. This did not satisfy the Agda termination checker, that is why the method is annotated with the TERMINATING flag. This forces the termination checker to skip this method and mark it as terminating. The flag is needed, because Agda does not know there are only finite amount of states. This method is however terminating if you define your NFA properly, such that it has a finite amount of states. Worst case scenario all the states will be reachable and also loop back to state where you initially called the method on. Then it will terminate because the state is already in the visited list. The resulting list most likely has duplicates in them, this is why there is method to filter out the duplicates. It does this by checking if the state at the start of the list also occurs in the tail of the list. It compares states by their name, so it is really important that you define states with unique names!

A more general statement of this problem is that the termination checker does not "see" the use of an extra data structure to keep track of termination, in this case the visited list. Agda only checks for structural recursion, "This means that we require recursive calls to be on a (strict) subexpression of the argument ..." [Tea24c]. In other words, the input to the recursive calls needs to be smaller than the input of the function which makes the recursive call.

6.2 General Agda problems

The problem that was most obvious when researching Agda itself, creating the encodings, and experimenting with the encodings, was the incompleteness of official documentation and examples. Especially for coinduction this was a problem, since the page about it only contained some basic examples. The page is mainly about the Guarded style, as this is the recommended way according to the same page. However, it lacks and explanation on why this the recommended style. There is a short section at the bottom for the Musical style, but this again lacks why this is not recommended anymore.

Another problem was the usefulness of the error messages. These would tell you the location where in your code the error occured, but the actual description was not always clear to what the exact problem was. An example of this would be the problem that occurred when proving equivalence for the list DFA. The error message was as follows:

```
q_reject !=
if
Relation.Nullary.Decidable.Core.isYes
(Relation.Nullary.Decidable.Core.map' Data.Char.Properties.≈⇒≡
 Data.Char.Properties.≈-reflexive
 (Relation.Nullary.Decidable.Core.map'
  (Data.Nat.Properties.≡ᵇ⇒≡ 97 (toℕ c))
```

```
  (Data.Nat.Properties.≡⇒≡ᵇ 97 (toN c))
  ((97 Agda.Builtin.Nat.== toN c)
   Relation.Nullary.Decidable.Core.because
   Relation.Nullary.Reflects.T-reflects
   (97 Agda.Builtin.Nat.== toN c))))
then q1 else getFromList c (('b' , q0) :: [])
of type DFAState
when checking that the expression q_reject_bisim_q_reject has type
getFromList c (q0 .transition) ~ getFromList c (q0 .transition)
```

This only said that the transition did not ultimately lead to the $q\_reject \sim q\_reject$ type. It did not say that the bisimilarity should be defined for every character.

I encountered this specific error message when trying to prove bisimilarity for two DFA's that are encoded using the list transition. Initially, I had the list transition as a list of pairs with Characters and States. Since the error message did not help me in resolving the problem, I tried to narrow it down. So I changed from the *Char* datatype to the *Symbol* datatype I created myself. This resolved the issue since I could now define the transition for the bisimilarity record for every symbol. This also learned me that you need to cannot match more than one character with the underscore in these types of proof. An example would example the left example in the code below, where the alphabet is still just a and b, but they both go to the same state. However, the example on the right is not allowed. Since only one character is matched with the underscore, namely the b.

```
q0_bisim_q0 .accept = refl
q0_bisim_q0 .transition _ = q1_bisim_q1
```

(a) Not allowed

```
q0_bisim_q0 .accept = refl
q0_bisim_q0 .transition a = q1_bisim_q1
q0_bisim_q0 .transition _ = q1_bisim_q1
```

(b) Allowed

7 Conclusions and Future Work

The main research question was to implement FSA in Agda using coinduction and prove their equivalence. Various encodings for DFA's have been made that accomplishes this goal, they can all receive input and run them, and they are able to be used to prove equivalence between them. An encoding for an NFA is also presented, this one however, can only run input and cannot be used to prove equivalence between two NFA.

The encodings I created are both in the Guarded and Musical style. I found no difference in functionality between those two. Since Guarded is the recommended way to do coinduction in Agda, I also recommend to use the Guarded versions of the encodings. Since they are easier to work with, because you do not need to think about when to delay and force certain computations.

Future work could look into the third method of coinduction in Agda, Sized types, to see whether or not this opens more possibilities. Future work in terms of functionality, could be to prove equivalence for the NFA encoding, but also operations or conversions on the created encodings. Examples of operations are: Unification, intersection, concatenation etc. Conversion would mainly be from an NFA to a DFA. A starting point for NFA equivalence could be the paper from Bonchi and Pous about NFA equivalence with bisimulations up to congruence [BP13].

References

[BP13] Filippo Bonchi and Damien Pous. "Checking NFA equivalence with bisimulations up to congruence". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 2013), pp. 457–468. DOI: `10.1145/2429069.2429124`.

[Geu09] H. Geuvers. "Proof assistants: History, ideas and future". In: *Sadhana* 34.1 (Feb. 2009), pp. 3–25. DOI: `10.1007/s12046-009-0001-5`.

[HR15] Thomas A. Henzinger and Jean-François Raskin. "The equivalence problem for finite automata". In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 86–86. DOI: `10.1145/2701001`.

[Iva20] Desislav Nikolaev Ivanov. "Formalization of Regular Languages in Agda". PhD thesis. UNIVERSIT 'A DEGLI STUDI DI TORINO, 2020.

[KS16] Dexter Kozen and Alexandra Silva. "Practical coinduction". In: *Mathematical Structures in Computer Science* 27.7 (Feb. 2016), pp. 1132–1152. DOI: `10.1017/s0960129515000493`.

[San11] Davide Sangiorgi. *Introduction to bisimulation and Coinduction*. Cambridge University Press, 2011.

[Sch14] Patrick Schnider. 2014. URL: `https://people.inf.ethz.ch/~fukudak/lect/mssemi/reports/09_rep_PatrickSchnider.pdf`.

[Sip13] Michael Sipser. *Introduction to the theory of computation Michael Sipser*. Cengagae Learning, 2013.

[Tea24a] The Agda Team. 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/getting-started/what-is-agda.html`.

[Tea24b] The Agda Team. 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/coinduction.html`.

[Tea24c] The Agda Team. 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/termination-checking.html`.

[Tej12] Matúš Tejiščák. *DFA*. 2012. URL: `https://github.com/ziman/agda-stuff/blob/master/dfa.agda`.

[Tur10] Franklyn Turbak. *DFA Operations (Revised)*. Course slides, CS235: Computation, Wellesley College. 2010. URL: `https://cs.wellesley.edu/~cs235/fall10/lectures/14_DFA_operations_revised_2.pdf`.

A Agda code

Musical style for the function transition version:

```
getFromList : Symbol → List (Symbol × (∞ DFAState)) → ∞ DFAState
getFromList e [ ] = q_reject
getFromList e ((x , y) :: xs) = if x == e then y else getFromList e xs

transition : DFAState → Symbol → DFAState
transition (state _ xs) c = ♭ (getFromList c xs)

runDFA : DFAState → List Symbol → DFAState
runDFA s [ ] = s
runDFA s (x :: xs) = runDFA (transition s x) xs

stateAccepts : DFAState → Bool
stateAccepts (state accept _) = accept

accepts : List Symbol → DFAState → Bool
accepts input startState = stateAccepts (runDFA startState input)
```

Musical style for list transition version:

```
getFromList : Symbol → List (Symbol × (∞ DFAState)) → ∞ DFAState
getFromList e [ ] = q_reject
getFromList e ((x , y) :: xs) = if x == e then y else getFromList e xs

transition : DFAState → Symbol → DFAState
transition (state _ xs) c = ♭ (getFromList c xs)

runDFA : DFAState → List Symbol → DFAState
runDFA s [ ] = s
runDFA s (x :: xs) = runDFA (transition s x) xs

stateAccepts : DFAState → Bool
stateAccepts (state accept _) = accept

accepts : List Symbol → DFAState → Bool
accepts input startState = stateAccepts (runDFA startState input)
```

B LLM prompts

In this appendix I will present some of the LLM prompts used during this project.

Prompt 1:

If I have this piece of Agda Code:

```
data Symbol : Set where
  a : Symbol
  b : Symbol

record DFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : Symbol → DFAState
```

It gets translated to latex in the following way:

```
\begin{code}%
\>[0]\AgdaKeyword{data}\AgdaSpace{}%
\AgdaDatatype{Symbol}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPrimitive{Set}\AgdaSpace{}%
\AgdaKeyword{where}\<%
\\
\>[0][@{}l@{\AgdaIndent{0}}]%
\>[2]\AgdaInductiveConstructor{a}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{Symbol}\<%
\\
%
\>[2]\AgdaInductiveConstructor{b}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{Symbol}\<%
\\
%
\\[\AgdaEmptyExtraSkip]%
\>[0]\AgdaKeyword{record}\AgdaSpace{}%
\AgdaRecord{DFAState}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPrimitive{Set}\AgdaSpace{}%
\AgdaKeyword{where}\<%
\\
\>[0][@{}l@{\AgdaIndent{0}}]%
\>[2]\AgdaKeyword{coinductive}\<%
\\
%
\>[2]\AgdaKeyword{field}\<%
\\
\>[2][@{}l@{\AgdaIndent{0}}]%
\>[4]\AgdaField{name}\AgdaSpace{}%
```

```
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPostulate{String}\<%
\\
%
\>[4]\AgdaField{isAccepting}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{Bool}\<%
\\
%
\>[4]\AgdaField{transition}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{Symbol}\AgdaSpace{}%
\AgdaSymbol{→}\AgdaSpace{}%
\AgdaRecord{DFAState}\<%
\\
\>[0]\<%
\end{code}
```

```
Can you do the same for this next piece of agda code?

record DFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List (Char × DFAState)
```

Answer to prompt 1:

```
Here's how your updated Agda code would be translated into LaTeX using the agda —lat
Agda Code

record DFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List (Char × DFAState)

LaTeX Translation

\begin{code}%
\>[0]\AgdaKeyword{record}\AgdaSpace{}%
\AgdaRecord{DFAState}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPrimitive{Set}\AgdaSpace{}%
\AgdaKeyword{where}\<%
\\
```

```
\>[0][@{}l@{\AgdaIndent{0}}]%
\>[2]\AgdaKeyword{coinductive}\<%
\\
%
\>[2]\AgdaKeyword{field}\<%
\\
\>[2][@{}l@{\AgdaIndent{0}}]%
\>[4]\AgdaField{name}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPostulate{String}\<%
\\
%
\>[4]\AgdaField{isAccepting}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{Bool}\<%
\\
%
\>[4]\AgdaField{transition}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{List}\AgdaSpace{}%
\AgdaSymbol{(}\AgdaDatatype{Char}\AgdaSpace{}%
\AgdaSymbol{×}\AgdaSpace{}%
\AgdaRecord{DFAState}\AgdaSymbol{)}\<%
\\
\>[0]\<%
\end{code}

Let me know if you'd like the same translation for a different formatting style (e.g.
```

Prompt 2:

```
Can you do it again for this piece of code please?

record NFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List ((Maybe Char) × NFAState)
```

Anser to prompt 2:

```
Certainly! Here's the LaTeX translation of your Agda code for the NFAState record usi
Agda Code

record NFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List ((Maybe Char) × NFAState)
```

LaTeX Translation

```
\begin{code}%
\>[0]\AgdaKeyword{record}\AgdaSpace{}%
\AgdaRecord{NFAState}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPrimitive{Set}\AgdaSpace{}%
\AgdaKeyword{where}\<%
\\
\>[0][@{}l@{\AgdaIndent{0}}]%
\>[2]\AgdaKeyword{coinductive}\<%
\\
%
\>[2]\AgdaKeyword{field}\<%
\\
\>[2][@{}l@{\AgdaIndent{0}}]%
\>[4]\AgdaField{name}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaPostulate{String}\<%
\\
%
\>[4]\AgdaField{isAccepting}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{Bool}\<%
\\
%
\>[4]\AgdaField{transition}\AgdaSpace{}%
\AgdaSymbol{:}\AgdaSpace{}%
\AgdaDatatype{List}\AgdaSpace{}%
\AgdaSymbol{((}\AgdaDatatype{Maybe}\AgdaSpace{}%
\AgdaDatatype{Char}\AgdaSymbol{)}\AgdaSpace{}%
\AgdaSymbol{×}\AgdaSpace{}%
\AgdaRecord{NFAState}\AgdaSymbol{)}\<%
\\
\>[0]\<%
\end{code}
```

Let me know if you'd like this rendered as a full document snippet or adjusted for s

I used more prompts just like prompt 2, but just with different code. I'll not be putting those in since it would mostly be duplication, and it is clear how the Agda to LaTeX conversion was done.