Modelling Finite State Automata in Agda

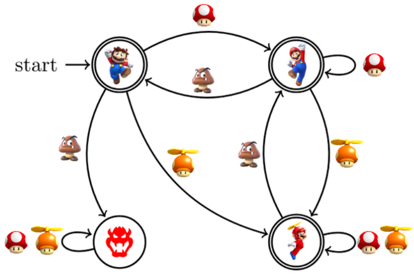
Noky Soekarman, Supervisor: Bohdan Liesnikov, Responsible Professor: Jesper Cockx

Contact: N.P.Soekarman@student.tudelft.nl

1. Introduction

Finite State Automata (FSA)

- Input Alphabet
- Set of states
- For every element in the alphabet there exists a transition to a next state (DFA).
- For a nondeterministic version there are zero or more transitions for each element in the alphabet.
- See the example with the Mario, whenever he takes a powerup or bumps into a goomba, he moves to another state.



Agda

- Functional programming language
  - Programs are constructed by applying and composing functions.
- Dependently typed
  - Dependent types are introduced by having families of types indexed by objects in another type.
- Total language
  - Program e of type T will always terminate with a value in T.

Coinduction

- Coinduction can be used to represent infinite or cyclic structures.
- To get values from these structures, observers and destructors are used.
- Two coinductive types are bisimilar if they have the same behaviour.

2. Research Question:

What are different ways to model Finite State Automata in Agda in a coinductive way and how to prove equivalence for them?

3. Methods

- Research what FSA are and when they are equivalent.
- Look into past work to see what has been done already.
- Implement different encodings of FSA.
- Try to prove equivalence for the various encodings.
- Compare the encodings.
- Suggest possible improvements for Agda to make it easier to use or implement FSA using coinduction.

4. Results

- Two fully working DFA encodings, both in Guarded and Musical style.
- A notion of equivalence for all the DFA encodings.
- An NFA encoding which can run input.

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

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

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

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

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

5. Agda Issues

- Incompleteness of documentation.
  - This made learning and understanding coinduction in Agda take more time.
- Error messages are not clear enough in Agda.
  - As a results, you needed to spend quite some time figuring out what was actually wrong.
- Termination checker only checks for structural recursion.
  - I needed to cheat the termination checker to be able to run input on the NFA.

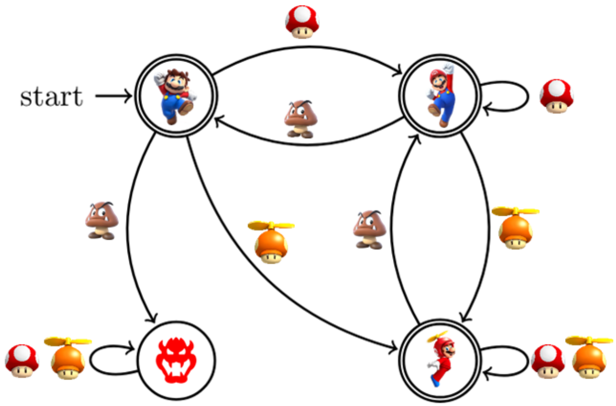
6. Limitations and future work

- The encodings encode a single state, one needs to be mindful of this when using the encodings.
- Some encodings do not force you to create a set of states exactly according to the official definition.
- Future work could investigate making use of sized types for encoding FSA, creating some notion of equivalence for an NFA, or adding operations to these encodings such as unifications, concatenation or conversion.

1. Introduction

Finite State Automata (FSA)

- Input Alphabet
- Set of states
- For every element in the alphabet there exists a transition to a next state (DFA).
- For a nondeterministic version there are zero or more transitions for each element in the alphabet.
- See the example with the Mario, whenever he takes a powerup or bumps into a goomba, he moves to another state.



Agda

- Functional programming language
  - Programs are constructed by applying and composing functions.
- Dependently typed
  - Dependent types are introduced by having families of types indexed by objects in another type.
- Total language
  - Program e of type T will always terminate with a value in T.

Coinduction

- Coinduction can be used to represent infinite or cyclic structures.
- To get values from these structures, observers and destructors are used.
- Two coinductive types are bisimilar if they have the same behaviour.

Modellin

Noky Soekarma

Contact: N.P.Soekarm

4. Results

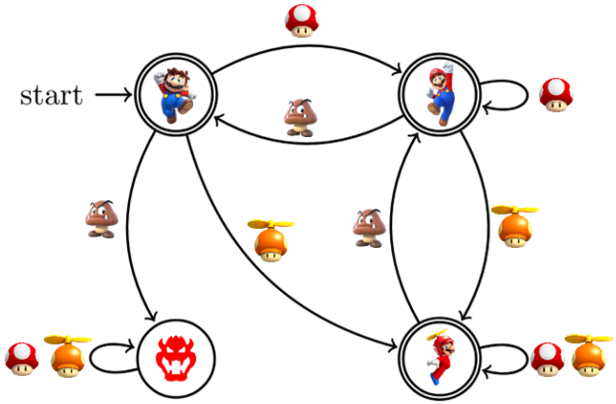
- Two fully working DFA encodings
- A notion of equivalence for all the
- An NFA encoding which can run i

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

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

```
record _≈₁_ (d1 d2 : DFAState) :
  coinductive
```

state.



- Program e of type T will always terminate with a value in T.

Coinduction

- Coinduction can be used to represent infinite or cyclic structures.
- To get values from these structures, observers and destructors are used.
- Two coinductive types are bisimilar if they have the same behaviour.

2. Research Question:

What are different ways to model Finite State Automata in Agda in a coinductive way and how to prove equivalence for them?

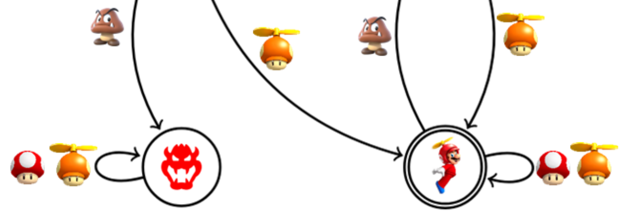
3. Methods

- Research what FSA are and when they are equivalent.
- Look into past work to see what has been done already.
- Implement different encodings of FSA.
- Try to prove equivalence for the various encodings.
- Compare the encodings.
- Suggest possible improvements for Agda to make it easier to use or implement FSA using coinduction.

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

```
record _≈l_ (d1 d2 : DFAState) : Se
  coinductive
  field
    equivLanguage : ∀ (s : List Sy
```

```
record NFAState : Set wh
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List ((
```



represent infinite or cyclic structures.

- To get values from these structures, observers and destructors are used.
- Two coinductive types are bisimilar if they have the same behaviour.

2. Research Question:

What are different ways to model Finite State Automata in Agda in a coinductive way and how to prove equivalence for them?

3. Methods

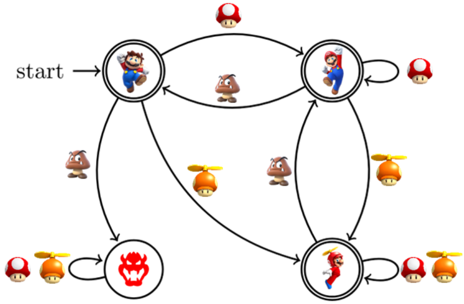
- Research what FSA are and when they are equivalent.
- Look into past work to see what has been done already.
- Implement different encodings of FSA.
- Try to prove equivalence for the various encodings.
- Compare the encodings.
- Suggest possible improvements for Agda to make it easier to use or implement FSA using coinduction.

```
  coinductive
  field
    accept : d1 .isAccepting ≡ d2
    transition : ∀ (c : Symbol) →
```

```
record _≈l_ (d1 d2 : DFAState) :
  coinductive
  field
    equivLanguage : ∀ (s : List
```

```
record NFAState : Set
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List
```

- Input Alphabet
- Set of states
- For every element in the alphabet there exists a transition to a next state (DFA).
- For a nondeterministic version there are zero or more transitions for each element in the alphabet.
- See the example with the Mario, whenever he takes a powerup or bumps into a goomba, he moves to another state.



- Functional programming language
  - Programs are constructed by applying and composing functions.
- Dependently typed
  - Dependent types are introduced by having families of types indexed by objects in another type.
- Total language
  - Program e of type T will always terminate with a value in T.

Coinduction

- Coinduction can be used to represent infinite or cyclic structures.
- To get values from these structures, observers and destructors are used.
- Two coinductive types are bisimilar if they have the same behaviour.

2. Research Question:

What are different ways to model Finite State Automata in Agda in a coinductive way and how to prove equivalence for them?

3. Methods

- Research what FSA are and when they are equivalent.
- Look into past work to see what has been done already.
- Implement different encodings of FSA.
- Try to prove equivalence for the various encodings.
- Compare the encodings.
- Suggest possible improvements for Agda to make it easier to use or implement FSA using coinduction.

Contact: N.P.Soekarman@student.tudelft.nl

4. Results

- Two fully working DFA encodings, both in Guarded and Musical style.
- A notion of equivalence for all the DFA encodings.
- An NFA encoding which can run input.

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

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

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

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

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

5. Agda Issues

- Incompleteness of documentati
  - This made learning and understanding coinducti take more time.
- Error messages are not clear eno Agda.
  - As a results, you needed quite some time figuring was actually wrong.
- Termination checker only checks structural recursion.
  - I needed to cheat the ter checker to be able to run the NFA.

6. Limitations and future

- The encodings encode a single st needs to be mindful of this when encodings.
- Some encodings do not force yo a set of states exactly according official definition.
- Future work could investigate ma sized types for encoding FSA, cre notion of equivalence for an NFA operations to these encodings su unifications, concatenation or co

Modelling Finite State Automata in Agda

Noky Soekarman, Supervisor: Bohdan Liesnikov, Responsible Professor: Jesper Cockx

Contact: N.P.Soekarman@student.tudelft.nl

g

y

nctions.

es are
having
es
ects in

ype T will
ate with a

d to
lic

e
nd

re
e same

ctive way

4. Results

- Two fully working DFA encodings, both in Guarded and Musical style.
- A notion of equivalence for all the DFA encodings.
- An NFA encoding which can run input.

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

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

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

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

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

5. Agda Issues

- Incompleteness of documentation.
  - This made learning and understanding coinduction in Agda take more time.
- Error messages are not clear enough in Agda.
  - As a results, you needed to spend quite some time figuring out what was actually wrong.
- Termination checker only checks for structural recursion.
  - I needed to cheat the termination checker to be able to run input on the NFA.

6. Limitations and future work

- The encodings encode a single state, one needs to be mindful of this when using the encodings.
- Some encodings do not force you to create a set of states exactly according to the official definition.
- Future work could investigate making use of sized types for encoding FSA, creating some notion of equivalence for an NFA, or adding operations to these encodings such as unifications, concatenation or conversion

T will
with a

o

ame

e way

```
  a : Symbol
  b : Symbol
```

```
  field
    name : String
    isAccepting : Bool
    transition : Symbol → DFAState
```

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

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

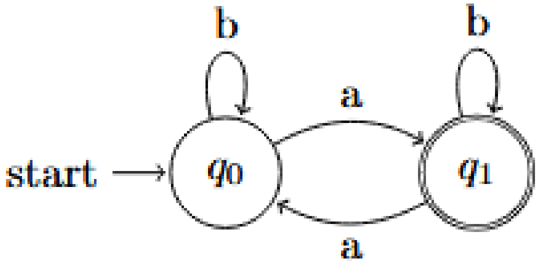
```
record NFAState : Set where
  coinductive
  field
    name : String
    isAccepting : Bool
    transition : List ((Maybe Symbol) × NFAState)
```

- Incompleteness of documentation.
  - This made learning and understanding coinduction in Agda take more time.
- Error messages are not clear enough in Agda.
  - As a results, you needed to spend quite some time figuring out what was actually wrong.
- Termination checker only checks for structural recursion.
  - I needed to cheat the termination checker to be able to run input on the NFA.

6. Limitations and future work

- The encodings encode a single state, one needs to be mindful of this when using the encodings.
- Some encodings do not force you to create a set of states exactly according to the official definition.
- Future work could investigate making use of sized types for encoding FSA, creating some notion of equivalence for an NFA, or adding operations to these encodings such as unifications, concatenation or conversion.

Example proof

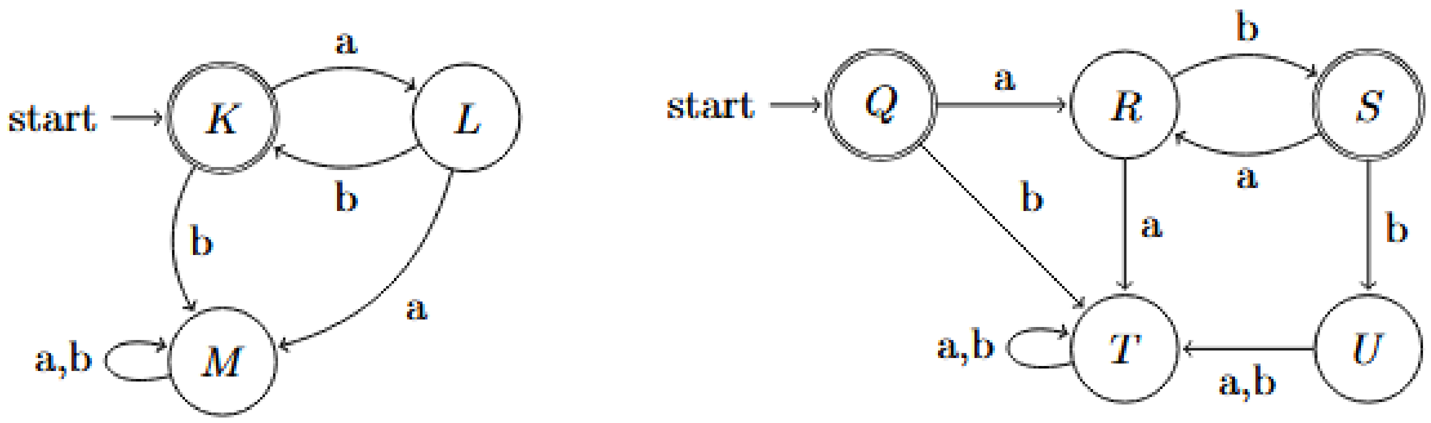


The proof for this in Agda:

```
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
```

Example proof



(a) DFA D_1

(b) DFA D_2

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

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

Bad error message

```
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)
```

Termination checker cheat

```
{-# 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
```