Formalising automata theory in Agda

Mark Armstrong
Department of Computing And Software, McMaster University
Supervisors: Jeffery Zucker & Wolfram Kahl

Problem: Formalising automata theory

By *formalise* we mean to *mechanise* in an automated proof assistant.

Sub-problems:

- Create representations of automata
- Code their semantics
- Complete formal proofs of known properties; e.g.
  - computational (in)equivalence of various machines
  - closure properties (of languages accepted)
  - existence of minimal state machines

End goal is to formalise *generalised computability theory*, meaning computation on *any* ring, e.g. the real numbers.

Methodology: Proofs ≈ Programs

- Agda is a dependently typed programming language.
- By the Curry-Howard correspondence, *propositions* correspond to *types*.
- So an Agda type can be viewed as a proposition.

Examples:

- `n : ℕ`
  - Proof there is a natural number
- `(n : ℕ) → n < suc n`
  - Proof every natural is less than its successor
- `p? : Decidable P`
  - Proof that a predicate `P` is decidable; i.e. `p?` is a decision procedure for `P`

References

1 Asperti, Andrea and Ricciotti, Wilmer. Formalizing Turing Machines.
2 Asperti, Andrea and Ricciotti, Wilmer. A formalization of multi-tape Turing machines.
3 Constable, Robert L. and Jackson, Paul B. and Naumov, Pavel and Uribe, Juan. Constructively Formalizing Automata Theory.
4 Ulf Norell. Towards a practical programming language based on dependent type theory.
5 Xu, Jian and Zhang, Xingyuan and Urban, Christian. Mechanising Turing Machines and Computability Theory in Isabelle/HOL.

A representation of deterministic finite automata

```
record DFA (|Σ| : ℕ) : Set₁ where
  constructor ⟨_,_,_,_⟩
  field
    |Q| : ℕ

  Q : Set
  Q = Fin |Q|

  Σ : Set
  Σ = Fin |Σ|

  field
    s : Q
    {F} : PredicateOn Q
    F? : Decidable F
    δ : Q → Σ → Q
```

Encoding of semantics

- Transitive closure of the transition function:

```
δ* : (M : DFA |Σ|) → Q → String Σ → Q
δ* M q [] = q
δ* M q (x :: xs) = δ* M (δ q x) xs
```

- Acceptance relation:

```
_Accepts_ : (M : DFA |Σ|) → String Σ → Set
M Accepts xs = F (δ* M s xs)
```

A "general" notion of (deterministic) automata

We want to be as general as possible — while still being reasonable.

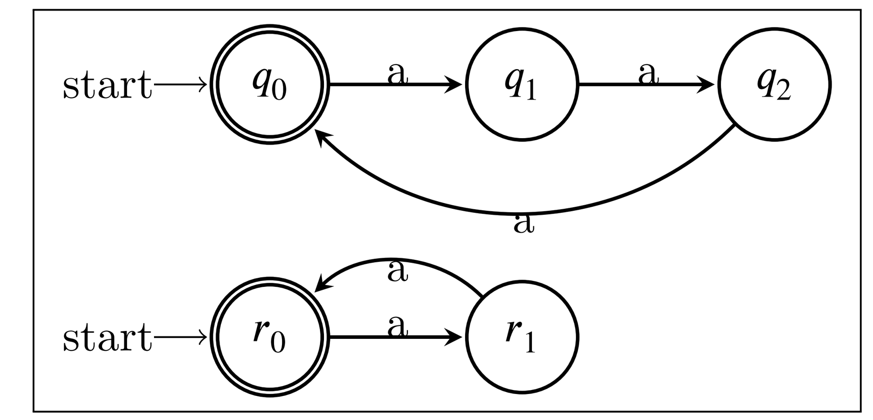
```
record Automaton : Set₁ where
  constructor ⟨_,_,_,_⟩
  field
    i : StateIndex

  Q : Set
  Q = StateSpace i

  field
    s : Q
    {F} : PredicateOn Q
    F? : Decidable F
    δ : Q → Σ → Q × Action
```

For instance, a Turing machine is an Automaton with a finite Q and actions which manipulate its tape(s).

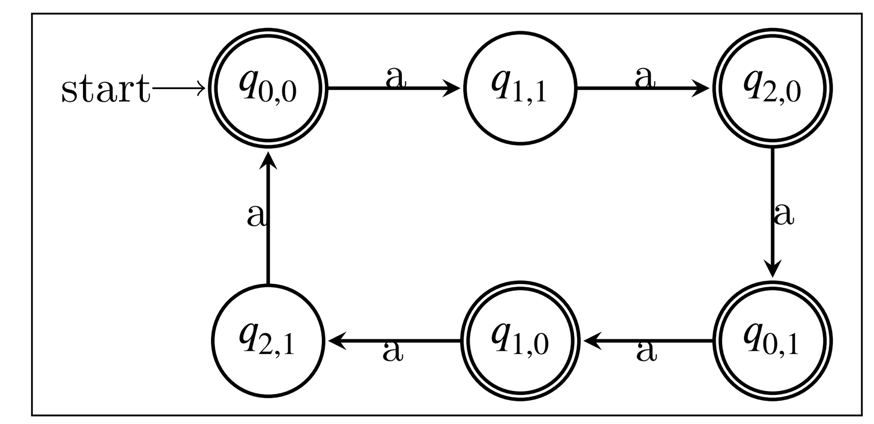
Example proposition: equivalence of DFAs and NFAs

We know *deterministic* and *non-deterministic* finite automata are *computationally equivalent* - i.e., accept the same languages.

For instance, an NFA (with two start states) which accepts strings of a's whose length is a multiple of 2 or 3.



And DFA for the same language.



Formalising equivalence of DFAs and NFAs

In Agda, we formalise (one direction of) this proposition by

- Constructing an NFA for each DFA:

```
DFA-to-NFA : DFA Σ → NFA Σ
```

- Proving it accepts all strings the DFA accepts:

```
accepts-all-strings : (M : DFA |Σ|)
                    → (xs : String Σ)
                    → M Accepts xs
                    → (DFA-to-NFA M) Accepts xs
```

- Proving it accepts *only* the strings the DFA accepts:

```
accepts-only-strings : (M : DFA |Σ|)
                     → (xs : String Σ)
                     → (DFA-to-NFA M) Accepts xs
                     → M Accepts xs
```