3.9: Empty-string Finite Automata

In this and the following two sections, we will study three progressively more restricted kinds of finite automata:

- empty-string finite automata (EFAs);
- nondeterministic finite automata (NFAs); and
- deterministic finite automata (DFAs).

Every DFA will be an NFA; every NFA will be an EFA; and every EFA will be an FA. Thus, $L(M)$ will be well-defined, if M is a DFA, NFA or EFA.

Introduction

The more restricted kinds of automata will be easier to process on the computer than the more general kinds; they will also have nicer reasoning principles than the more general kinds.

Introduction

The more restricted kinds of automata will be easier to process on the computer than the more general kinds; they will also have nicer reasoning principles than the more general kinds.

We will give algorithms for converting the more general kinds of automata into the more restricted kinds. Thus even the deterministic finite automata will accept the same set of languages as the finite automata.

Introduction

The more restricted kinds of automata will be easier to process on the computer than the more general kinds; they will also have nicer reasoning principles than the more general kinds.

We will give algorithms for converting the more general kinds of automata into the more restricted kinds. Thus even the deterministic finite automata will accept the same set of languages as the finite automata.

On the other hand, it will sometimes be easier to find one of the more general kinds of automata that accepts a given language rather than one of the more restricted kinds accepting the language.

Introduction

The more restricted kinds of automata will be easier to process on the computer than the more general kinds; they will also have nicer reasoning principles than the more general kinds.

We will give algorithms for converting the more general kinds of automata into the more restricted kinds. Thus even the deterministic finite automata will accept the same set of languages as the finite automata.

On the other hand, it will sometimes be easier to find one of the more general kinds of automata that accepts a given language rather than one of the more restricted kinds accepting the language.

And, there are languages where the smallest DFA accepting the language is exponentially bigger than the smallest FA accepting the language.

Definition of EFAs

An *empty-string finite automaton* (EFA) M is a finite automaton such that

$$T_M \subseteq \{\, q, x \to r \mid q, r \in \textbf{Sym} \text{ and } x \in \textbf{Str} \text{ and } |x| \leq 1 \,\}.$$

For example, $\mathsf{A}, \% \to \mathsf{B}$ and $\mathsf{A}, 1 \to \mathsf{B}$ are legal EFA transitions, but $\mathsf{A}, 11 \to \mathsf{B}$ is not legal.

We write **EFA** for the set of all empty-string finite automata. Thus **EFA** $\subsetneq$ **FA**.

The following proposition obviously holds.

Proposition 3.9.1

Suppose M is an EFA.

- *For all $N \in$ **FA**, if M **iso** N, then N is an EFA.*
- *For all bijections f from Q_M to some set of symbols,* **renameStates**(M, f) *is an EFA.*
- **renameStatesCanonically** M *is an EFA.*
- **simplify** M *is an EFA.*

Converting FAs to EFAs

If we want to convert an FA into an equivalent EFA, we can proceed as follows. Every state of the FA will be a state of the EFA, the start and accepting states are unchanged, and every transition of the FA that is a legal EFA transition will be a transition of the EFA.

Converting FAs to EFAs

If we want to convert an FA into an equivalent EFA, we can proceed as follows. Every state of the FA will be a state of the EFA, the start and accepting states are unchanged, and every transition of the FA that is a legal EFA transition will be a transition of the EFA.

If our FA has a transition

$$p, b_1 b_2 \cdots b_n \to r,$$

where $n \geq 2$ and the b_i are symbols, then we replace this transition with

Converting FAs to EFAs

If we want to convert an FA into an equivalent EFA, we can proceed as follows. Every state of the FA will be a state of the EFA, the start and accepting states are unchanged, and every transition of the FA that is a legal EFA transition will be a transition of the EFA.

If our FA has a transition

$$p, b_1 b_2 \cdots b_n \to r,$$

where $n \geq 2$ and the b_i are symbols, then we replace this transition with the n transitions

$$p \xrightarrow{b_1} q_1, q_1 \xrightarrow{b_2} q_2, \ldots, q_{n-1} \xrightarrow{b_n} r,$$

where $q_1, \ldots, q_{n-1}$ are $n - 1$ new, states.

Converting FAs to EFAs

If we want to convert an FA into an equivalent EFA, we can proceed as follows. Every state of the FA will be a state of the EFA, the start and accepting states are unchanged, and every transition of the FA that is a legal EFA transition will be a transition of the EFA.

If our FA has a transition

$$p, b_1 b_2 \cdots b_n \to r,$$

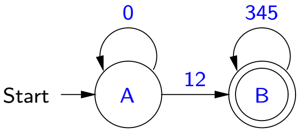
where $n \geq 2$ and the b_i are symbols, then we replace this transition with the n transitions

$$p \stackrel{b_1}{\to} q_1, q_1 \stackrel{b_2}{\to} q_2, \ldots, q_{n-1} \stackrel{b_n}{\to} r,$$

where $q_1, \ldots, q_{n-1}$ are $n - 1$ new, non-accepting, states.

Example FA to EFA Conversion

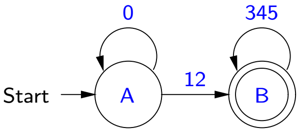
For example, we can convert the FA



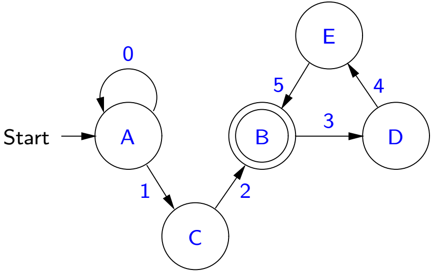
into the EFA

Example FA to EFA Conversion

For example, we can convert the FA



into the EFA



An FA to EFA Conversion Algorithm

How should be go about choosing the new states?

An FA to EFA Conversion Algorithm

How should be go about choosing the new states?

The symbols we choose can't be states of the original machine, and we can't choose the same symbol twice.

A Conversion Algorithm

First, we rename each old state q to $\langle 1, q\rangle$. Then we can replace a transition

$$p \stackrel{b_1 b_2 \cdots b_n}{\rightarrow} r,$$

where $n \geq 2$ and the b_i are symbols, with the transitions

$$\langle 1, p\rangle \stackrel{b_1}{\rightarrow} \langle 2, \langle p, b_1, b_2 \cdots b_n, r\rangle\rangle,$$

$$\langle 2, \langle p, b_1, b_2 \cdots b_n, r\rangle\rangle \stackrel{b_2}{\rightarrow} \langle 2, \langle p, b_1 b_2, b_3 \cdots b_n, r\rangle\rangle,$$

$$\ldots,$$

$$\langle 2, \langle p, b_1 b_2 \cdots b_{n-1}, b_n, r\rangle\rangle \stackrel{b_n}{\rightarrow}, \langle 1, r\rangle.$$

A Conversion Algorithm

We define a function **faToEFA** $\in$ **FA** $\to$ **EFA** that converts FAs into EFAs by saying that **faToEFA** M is the result of running the above algorithm on input M.

Theorem 3.9.2

For all $M \in$ **FA**:

- **faToEFA** $M \approx M$; *and*
- **alphabet**(**faToEFA** M) $=$ **alphabet** M.

Processing EFAs in Forlan

The Forlan module `EFA` defines an abstract type `efa` (in the top-level environment) of empty-string finite automata, along with various functions for processing EFAs.

Values of type `efa` are implemented as values of type `fa`, and the module EFA provides functions:

```
val injToFA    : efa -> fa
val projFromFA : fa -> efa
val input      : string -> efa
val fromFA     : fa -> efa
```

`injToFA` is an "injection" function. `projFromFA` is a "projection" function, which raises an exception if its argument isn't a legal EFA. The last of these is in the top-level environment as:

```
val faToEFA : fa -> efa
```

Processing EFAs in Forlan

Finally, most of the functions for processing FAs that were introduced in previous sections are inherited by `EFA`:

```
val output                    : string * efa -> unit
val numStates                 : efa -> int
val numTransitions            : efa -> int
val equal                     : efa * efa -> bool
val alphabet                  : efa -> sym set
val checkLP                   : efa -> lp -> unit
val validLP                   : efa -> lp -> bool
val isomorphism               : efa * efa * sym_rel -> bool
val findIsomorphism           : efa * efa -> sym_rel
val isomorphic                : efa * efa -> bool
val renameStates              : efa * sym_rel -> efa
val renameStatesCanonically : efa -> efa
```

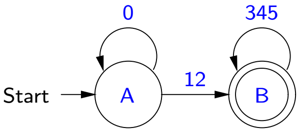
Processing EFAs in Forlan

More inherited functions:

```
val processStr       : efa -> sym set * str -> sym set
val accepted         : efa -> str -> bool
val findLP           : efa -> sym set * str * sym set -> lp
val findAcceptingLP : efa -> str -> lp
val simplified       : efa -> bool
val simplify         : efa -> efa
```

Forlan Examples

Suppose that `fa` is the finite automaton



Here are some example uses of a few of the above functions:

```
- projFAToEFA fa;
invalid label in transition: "12"

uncaught exception Error
- val efa = faToEFA fa;
val efa = - : efa
```

Forlan Examples

```
- EFA.output("", efa);
{states}
<1,A>, <1,B>, <2,<A,1,2,B>>, <2,<B,3,45,B>>,
<2,<B,34,5,B>>
{start state} <1,A> {accepting states} <1,B>
{transitions}
<1,A>, 0 -> <1,A>; <1,A>, 1 -> <2,<A,1,2,B>>;
<1,B>, 3 -> <2,<B,3,45,B>>; <2,<A,1,2,B>>, 2 -> <1,B>;
<2,<B,3,45,B>>, 4 -> <2,<B,34,5,B>>;
<2,<B,34,5,B>>, 5 -> <1,B>
val it = () : unit
```

Forlan Examples

```
- val efa' = EFA.renameStatesCanonically efa;
val efa' = - : efa
- EFA.output("", efa');
{states} A, B, C, D, E {start state} A
{accepting states} B
{transitions}
A, 0 -> A; A, 1 -> C; B, 3 -> D; C, 2 -> B; D, 4 -> E;
E, 5 -> B
val it = () : unit
```

```
- val rel = EFA.findIsomorphism(efa, efa');
val rel = - : sym_rel
- SymRel.output("", rel);
(<1,A>, A), (<1,B>, B), (<2,<A,1,2,B>>, C),
(<2,<B,3,45,B>>, D), (<2,<B,34,5,B>>, E)
val it = () : unit
- LP.output("", FA.findAcceptingLP fa (Str.input ""));
@ 012345
@ .
A, 0 => A, 12 => B, 345 => B
val it = () : unit
- LP.output
= ("", EFA.findAcceptingLP efa' (Str.input ""));
@ 012345
@ .
A, 0 => A, 1 => C, 2 => B, 3 => D, 4 => E, 5 => B
val it = () : unit
```