

3.10: Nondeterministic Finite Automata

In this section, we study the second of our more restricted kinds of finite automata: nondeterministic finite automata.

Definition of NFAs

A *nondeterministic finite automaton* (NFA) M is a finite automaton such that

$$T_M \subseteq \{q, x \rightarrow r \mid q, r \in \mathbf{Sym} \text{ and } x \in \mathbf{Str} \text{ and } |x| = 1\}.$$

For example, $A, 1 \rightarrow B$ is a legal NFA transition, but $A, \% \rightarrow B$ and $A, 11 \rightarrow B$ are not legal.

We write **NFA** for the set of all nondeterministic finite automata.
Thus **NFA** \subsetneq **EFA** \subsetneq **FA**.

Properties of NFAs

The following proposition obviously holds.

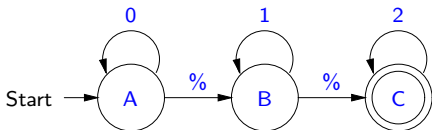
Proposition 3.10.1

Suppose M is an NFA.

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

Converting EFAs to NFAs

Suppose M is the EFA



To convert M into an equivalent NFA, we will have to:

- replace the transitions $A, \% \rightarrow B$ and $B, \% \rightarrow C$ with legal transitions (for example, because of the valid labeled path

$$A \xRightarrow{\%} B \xRightarrow{1} B \xRightarrow{\%} C,$$

we will add the transition $A, 1 \rightarrow C$);

- make (at least) A be an accepting state (so that $\%$ is accepted by the NFA).

The Empty-closure of a Set of States

Suppose M is a finite automaton and $P \subseteq Q_M$. The *empty-closure* of P ($\mathbf{emptyClose}_M P$) is the least subset X of Q_M such that

- $P \subseteq X$;
- for all $q, r \in Q_M$, if $q \in X$ and $q, \% \rightarrow r \in T_M$, then $r \in X$.

For example, if M is our example EFA and $P = \{A\}$, then:

- $A \in X$;
- $B \in X$, since $A \in X$ and $A, \% \rightarrow B \in T_M$;
- $C \in X$, since $B \in X$ and $B, \% \rightarrow C \in T_M$.

Thus $\mathbf{emptyClose} P = \{A, B, C\}$.

Backwards Empty-closure

Suppose M is a finite automaton and $P \subseteq Q_M$. The *backwards empty-closure* of P ($\mathbf{emptyCloseBackwards}_M P$) is the least subset X of Q_M such that

- $P \subseteq X$;
- for all $q, r \in Q_M$, if $r \in X$ and $q, \% \rightarrow r \in T_M$, then $q \in X$.

For example, if M is our example EFA and $P = \{C\}$, then:

- $C \in X$;
- $B \in X$, since $C \in X$ and $B, \% \rightarrow C \in T_M$;
- $A \in X$, since $B \in X$ and $A, \% \rightarrow B \in T_M$.

Thus $\mathbf{emptyCloseBackwards} P = \{A, B, C\}$.

Properties of Empty-closure and Backwards Empty-closure

Proposition 3.10.2

Suppose M is a finite automaton. For all $P \subseteq Q_M$,
emptyClose $_M P = \Delta_M(P, \%)$.

Proposition 3.10.3

Suppose M is a finite automaton. For all $P \subseteq Q_M$,
emptyCloseBackwards $_M P = \{q \in Q_M \mid \Delta_M(\{q\}, \%) \cap P \neq \emptyset\}$.

Conversion Algorithm

We define a function/algorithm **efaToNFA** \in **EFA** \rightarrow **NFA** that converts EFAs into NFAs by saying that **efaToNFA** M is the NFA N such that:

- $Q_N = Q_M$;
- $s_N = s_M$;
- $A_N = \mathbf{emptyCloseBackwards} A_M$;
- T_N is the set of all transitions $q', a \rightarrow r'$ such that $q', r' \in Q_M$, $a \in \mathbf{Sym}$, and there are $q, r \in Q_M$ such that:
 - $q, a \rightarrow r \in T_M$;
 - $q' \in \mathbf{emptyCloseBackwards} \{q\}$; and
 - $r' \in \mathbf{emptyClose} \{r\}$.

Conversion Algorithm

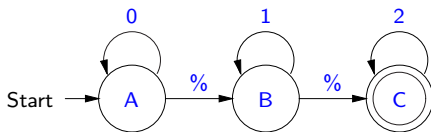
To compute the set T_N , we process each transition $q, x \rightarrow r$ of M as follows. If $x = \%$, then we generate no transitions. Otherwise, our transition is $q, a \rightarrow r$ for some symbol a . We then compute the backwards empty-closure of $\{q\}$, and call the result X , and compute the (forwards) empty-closure of $\{r\}$, and call the result Y . We then add all of the elements of

$$\{q', a \rightarrow r' \mid q' \in X \text{ and } r' \in Y\}$$

to T_N .

Conversion Example

Let M be our example EFA



and let $N = \mathbf{efaToNFA} M$. Then

- $Q_N = Q_M = \{A, B, C\}$;
- $s_N = s_M = A$;
- $A_N = \mathbf{emptyCloseBackwards} A_M = \mathbf{emptyCloseBackwards} \{C\} = \{A, B, C\}$.

Conversion Example

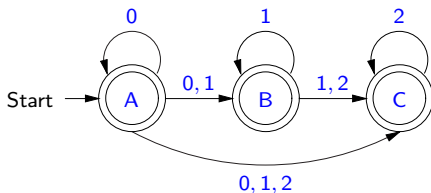
Now, let's work out what T_N is, by processing each of M 's transitions.

- From the transitions $A, \% \rightarrow B$ and $B, \% \rightarrow C$, we get no elements of T_N .
- Consider the transition $A, 0 \rightarrow A$. Since **emptyCloseBackwards** $\{A\} = \{A\}$ and **emptyClose** $\{A\} = \{A, B, C\}$, we add $A, 0 \rightarrow A$, $A, 0 \rightarrow B$ and $A, 0 \rightarrow C$ to T_N .
- Consider the transition $B, 1 \rightarrow B$. Since **emptyCloseBackwards** $\{B\} = \{A, B\}$ and **emptyClose** $\{B\} = \{B, C\}$, we add $A, 1 \rightarrow B$, $A, 1 \rightarrow C$, $B, 1 \rightarrow B$ and $B, 1 \rightarrow C$ to T_N .

Conversion Example

- Consider the transition $C, 2 \rightarrow C$. Since **emptyCloseBackwards** $\{C\} = \{A, B, C\}$ and **emptyClose** $\{C\} = \{C\}$, we add $A, 2 \rightarrow C$, $B, 2 \rightarrow C$ and $C, 2 \rightarrow C$ to T_N .

Thus our NFA N is



Specification of Conversion Function

Theorem 3.10.7

For all $M \in \mathbf{EFA}$:

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

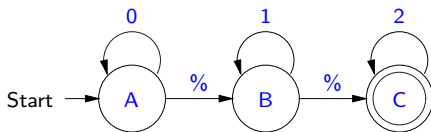
Empty-closure in Forlan

The Forlan module **FA** defines the following functions for computing forwards and backwards empty-closures:

```
val emptyClose          : fa -> sym set -> sym set
val emptyCloseBackwards : fa -> sym set -> sym set
```

Empty-closure in Forlan

For example, if `fa` is bound to the finite automaton



then we can compute the empty-closure of `{A}` as follows:

```
- SymSet.output  
= ("",  
  = FA.emptyClose fa (SymSet.input ""));  
@ A  
@ .  
A, B, C  
val it = () : unit
```

Processing NFAs in Forlan

The Forlan module `NFA` defines an abstract type `nfa` (in the top-level environment) of nondeterministic finite automata, along with various functions for processing NFAs.

Values of type `nfa` are implemented as values of type `fa`, and the module `NFA` provides the following injection and projection functions:

```
val injToFA      : nfa -> fa
val injToEFA     : nfa -> efa
val projFromFA   : fa -> nfa
val projFromEFA  : efa -> nfa
```

The functions `injToFA`, `injToEFA`, `projFromFA` and `projFromEFA` are available in the top-level environment as `injNFAToFA`, `injNFAToEFA`, `projFAToNFA` and `projEFAToNFA`, respectively.

Processing NFAs in Forlan

The module `NFA` also defines the functions:

```
val input    : string -> nfa
val fromEFA : efa -> nfa
```

The function `input` is used to input an NFA, and the function `fromEFA` corresponds to our conversion function `efaToNFA`, and is available in the top-level environment with that name:

```
val efaToNFA : efa -> nfa
```

Processing NFAs in Forlan

Most of the functions for processing FAs that were introduced in previous sections are inherited by **NFA**:

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

Processing NFAs in Forlan

More inherited functions:

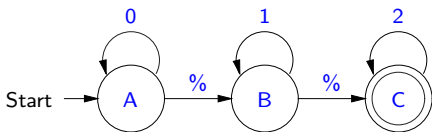
```
val processStr      : nfa -> sym set * str -> sym set
val accepted       : nfa -> str -> bool
val findLP         : nfa -> sym set * str * sym set -> lp
val findAcceptingLP : nfa -> str -> lp
val simplified      : nfa -> bool
val simplify       : nfa -> nfa
```

Finally, the functions for computing forwards and backwards empty-closures are inherited by the EFA module

```
val emptyClose      : efa -> sym set -> sym set
val emptyCloseBackwards : efa -> sym set -> sym set
```

Forlan Examples

Suppose that `efa` is the `efa`



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

```
- projEFAToNFA efa;  
invalid label in transition: "%"
```

```
uncaught exception Error  
- val nfa = efaToNFA efa;  
val nfa = - : nfa
```

Forlan Examples

```
- NFA.output("", nfa);  
{states} A, B, C {start state} A  
{accepting states} A, B, C  
{transitions}  
A, 0 -> A | B | C; A, 1 -> B | C; A, 2 -> C;  
B, 1 -> B | C; B, 2 -> C; C, 2 -> C  
val it = () : unit
```

Forlan Examples

```
- LP.output
= ("", EFA.findAcceptingLP efa (Str.input ""));
@ 012
@ .
A, 0 => A, % => B, 1 => B, % => C, 2 => C
val it = () : unit

- LP.output
= ("", NFA.findAcceptingLP nfa (Str.input ""));
@ 012
@ .
A, 0 => A, 1 => B, 2 => C
val it = () : unit
```