**Please read this!**

Many years ago — I don’t want to say how many, it’s depressing — I taught a course on the automata-theoretic approach to model checking at the Technical University of Munich, basing it on lectures notes for another course on the same topic that Moshe Vardi had recently taught in Israel. Between my lectures I extended and polished the notes, and sent them to Moshe. At that time he and Orna Kupferman were thinking of writing a book, and the idea came up of doing it together. We made some progress, but life and other work got in the way, and the project has been postponed so many times that it I don’t dare to predict a completion date.

Some of the work that got in the way was the standard course on automata theory in Munich, which I had to teach several times. The syllabus contained both automata on finite and infinite words, and for the latter I used our notes. Each time I had to teach the course again, I took the opportunity to add some new material about automata on finite words, which also required to reshape the chapters on infinite words, and the notes kept growing and evolving. Now they’ve reached the point where they are in sufficiently good shape to be shown not only to my students, but to a larger audience. So, after getting Orna and Moshe’s very kind permission, I’ve decided to make them available here.

Despite several attempts I haven’t yet convinced Orna and Moshe to appear as co-authors of the notes. But I don’t give up: apart from the material we wrote together, their influence on the rest is much larger than they think. Actually, my secret hope is that after they see this material in my home page we’ll finally manage to gather some morsels of time here and there and finish our joint project. If you think we should do so, tell us! Send an email to: vardi@cs.rice.edu, orna@cs.huji.ac.il, and esparza@in.tum.de.

**Sources**

I haven’t yet compiled a careful list of the sources I’ve used, but I’m listing here the main ones. I apologize in advance for any omissions.

- The chapter on automata for fixed-length languages (‘Finite Universes’)’ was very influenced by Henrik Reif Andersen’s beautiful introduction to Binary Decision Diagrams, available at [www.itu.dk/courses/AVA/E2005/bdd-eap.pdf](http://www.itu.dk/courses/AVA/E2005/bdd-eap.pdf).

- The short chapter on pattern matching is influenced by David Eppstein’s lecture notes for his course on Design and Analysis of Algorithms, see [http://www.ics.uci.edu/~eppstein/teach.html](http://www.ics.uci.edu/~eppstein/teach.html).

- As mentioned above, the chapters on operations for Büchi automata and applications to verification are heavily based on notes by Orna Kupferman and Moshe Vardi.

- The chapter on the emptiness problem for Büchi automata is based on several research papers:

• The chapter on Linear Arithmetic is heavily based on the work of Bernard Boigelot, Pierre Wolper, and their co-authors, in particular the paper “An effective decision procedure for linear arithmetic over the integers and reals”, published in ACM. Trans. Comput. Logic 6(3) in 2005.

Acknowledgments

First of all, thanks to Orna Kupferman and Moshe Vardi for all the reasons explained above (if you haven’t read the section “Please read this” yet, please do it now!). Many thanks to Jörg Kreiker, Jan Kretinsky, and Michael Luttenberger for many discussions on the topic of this notes, and for their contributions to several chapters. All three of them helped me to teach the automata course of different occasions. In particular, Jan contributed a lot to the chapter on pattern matching. Breno Faria helped to draw many figures. He was funded by a program of the Computer Science Department Technical University of Munich. Thanks also to Birgit Engelmann, Moritz Fuchs, Stefan Krusche, Philipp Müller, Martin Perzl, Marcel Ruegenberg, Franz Saller, Hayk Shoukourian, and Daniel Weißauer, all of whom attended the automata course, and provided very helpful comments.
# Contents

1 Introduction and Goal  

I Automata on Finite Words  

2 Automata Classes and Conversions  

2.1 Regular expressions: a language to describe languages  
2.2 Automata classes  

2.2.1 Using DFAs as data structures  
2.3 Conversion Algorithms between Finite Automata  

2.3.1 From NFA to DFA  
2.3.2 From NFA-\(\varepsilon\) to NFA  
2.4 Conversion algorithms between regular expressions and automata  

2.4.1 From regular expressions to NFA-\(\varepsilon\)’s  
2.4.2 From NFA-\(\varepsilon\)’s to regular expressions  
2.5 A Tour of Conversions  

3 Minimization and Reduction  

3.1 Minimal DFAs  
3.2 Minimizing DFAs  

3.2.1 Computing the language partition  
3.2.2 Quotienting  
3.3 Reducing NFAs  

3.3.1 The reduction algorithm  
3.4 A Characterization of the Regular Languages  

4 Operations on Sets: Implementations  

4.1 Implementation on DFAs  
4.1.1 Membership  
4.1.2 Complement  
4.1.3 Binary Boolean Operations
4.1.4 Emptiness. .................................................. 66
4.1.5 Universality. ............................................... 66
4.1.6 Inclusion. .................................................. 67
4.1.7 Equality. ................................................... 67

4.2 Implementation on NFAs ...................................... 67
4.2.1 Membership. ............................................... 68
4.2.2 Complement. ............................................... 69
4.2.3 Union and intersection. ................................... 69
4.2.4 Emptiness and Universality. ............................. 72
4.2.5 Inclusion and Equality. ................................... 75

5 Applications I: Pattern matching 85
5.1 The general case .............................................. 85
5.2 The word case ................................................ 87
5.2.1 Lazy DFAs .................................................. 90

6 Operations on Relations: Implementations 95
6.1 Encodings ...................................................... 96
6.2 Transducers and Regular Relations .......................... 97
6.3 Implementing Operations on Relations ....................... 98
6.3.1 Projection .................................................. 99
6.3.2 Join, Post, and Pre ....................................... 100
6.4 Relations of Higher Arity ..................................... 106

7 Finite Universes 111
7.1 Fixed-length Languages and the Master Automaton .......... 111
7.2 A Data Structure for Fixed-length Languages ............... 113
7.3 Operations on fixed-length languages ........................ 115
7.4 Determinization and Minimization ............................. 121
7.5 Operations on Fixed-length Relations ........................ 122
7.6 Decision Diagrams ............................................ 127
7.6.1 Zip-automata and Kernels ................................ 128
7.6.2 Operations on Kernels .................................... 131

8 Applications II: Verification 139
8.1 The Automata-Theoretic Approach to Verification .......... 139
8.2 Networks of Automata. ....................................... 142
8.2.1 Checking Properties ...................................... 146
8.3 The State-Explosion Problem .................................. 148
8.3.1 Symbolic State-space Exploration ........................ 149
8.4 Safety and Liveness Properties .............................. 153
9 Automata and Logic 159
  9.1 First-Order Logic on Words 159
      9.1.1 Expressive power of \( FO(\Sigma) \) 162
  9.2 Monadic Second-Order Logic on Words 163
      9.2.1 Expressive power of \( MSO(\Sigma) \) 164

10 Applications III: Presburger Arithmetic 179
  10.1 Syntax and Semantics 179
  10.2 An NFA for the Solutions over the Naturals.
      10.2.1 Equations 185
  10.3 An NFA for the Solutions over the Integers.
      10.3.1 Equations 189
      10.3.2 Algorithms 190

II Automata on Infinite Words 193

11 Classes of \( \omega \)-Automata and Conversions 195
  11.1 \( \omega \)-languages and \( \omega \)-regular expressions 195
  11.2 B"uchi automata 196
      11.2.1 From \( \omega \)-regular expressions to NBAs and back 198
      11.2.2 Non-equivalence of NBA and DBA 200
  11.3 Generalized B"uchi automata 201
  11.4 Other classes of \( \omega \)-automata 203
      11.4.1 Co-B"uchi Automata 203
      11.4.2 Muller automata 206
      11.4.3 Rabin automata 209

12 Boolean operations: Implementations 215
  12.1 Union and intersection 215
  12.2 Complement 218
      12.2.1 The problems of complement 218
      12.2.2 Rankings and ranking levels 220
      12.2.3 A (possible infinite) complement automaton 221
      12.2.4 The size of \( \overline{A} \) 226

13 Emptiness check: Implementations 229
  13.1 Algorithms based on depth-first search 229
      13.1.1 The nested-DFS algorithm 232
      13.1.2 The two-stack algorithm 238
  13.2 Algorithms based on breadth-first search 250
      13.2.1 Emerson-Lei’s algorithm 251
13.2.2 A Modified Emerson-Lei’s algorithm ........................................... 253
13.2.3 Comparing the algorithms ......................................................... 255

14 Applications I: Verification and Temporal Logic .................................. 259
14.1 Automata-Based Verification of Liveness Properties .............................. 259
14.1.1 Checking Liveness Properties ...................................................... 260
14.2 Linear Temporal Logic ..................................................................... 263
14.3 From LTL formulas to generalized Büchi automata .............................. 266
14.3.1 Satisfaction sequences and Hintikka sequences ............................... 266
14.3.2 Constructing the NGA ............................................................... 269
14.3.3 Reducing the NGA ................................................................. 271
14.3.4 Size of the NGA ..................................................................... 272
14.4 Automatic Verification of LTL Formulas .......................................... 273

15 Applications II: Monadic Second-Order Logic and Linear Arithmetic .......... 277
15.1 Monadic Second-Order Logic on $\omega$-Words ..................................... 277
15.1.1 Expressive power of $MSO(\Sigma)$ on $\omega$-words ............................ 278
15.2 Linear Arithmetic ....................................................................... 279
15.2.1 Encoding Real Numbers .......................................................... 279
15.3 Constructing an NBA for the Real Solutions ...................................... 280
15.3.1 A NBA for the Solutions of $a \cdot x_F \leq \beta$ .................................. 282
Why this book?

There are excellent textbooks on automata theory, ranging from course books for undergraduates to research monographs for specialists. Why another one?

During the late 1960s and early 1970s the main application of automata theory was the development of lexicographic analyzers, parsers, and compilers, and so automata were viewed as abstract machines that process inputs and accept or reject them. This view deeply influenced the textbook presentation of the theory. While results about the expressive power of machines, like pumping lemmas, equivalences between models, and closure properties, were devoted much attention, constructions on automata, like the powerset or product construction, often played a subordinate role as proof tools. To give a simple example, in many textbooks of the time—and in later textbooks written in the same style—the powerset construction is not introduced as an algorithm that constructs a DFA equivalent to a given NFA; instead, the equivalent DFA is mathematically defined by describing its set of states, transition relation, etc.. Sometimes, the simple but computationally important fact that only states reachable from the initial state need be constructed is not mentioned.

Since the middle 1980s program verification has progressively replaced compilers as the driving application of automata theory. Automata started to be used to describe the behaviours—and intended behaviour—of hardware and software systems, not their syntactic descriptions. This shift from syntax to semantics had important consequences. While automata for lexical or syntactical analysis typically have at most some thousands of states, automata for semantic descriptions can easily be many orders of magnitude larger. Efficient constructions and algorithmic issues became imperative, and research in this direction made great progress. At the same time, automata on infinite words and the connection between logic and automata became more relevant. While these research developments certainly influenced textbooks, and algorithmic issues gradually received more attention, undergraduate texts still retain many concepts from the past.

This book intends to reflect the evolution of automata theory. Automata on infinite words can hardly be seen as machines accepting or rejecting an input: they could only do so after infinite time! And the view automata as abstract machines puts an emphasis expressibility that no longer corresponds to its relevance for modern applications. We think that modern automata theory is better captured by the slogan automata as data structures. Just as hash tables and Fibonacci heaps are both adequate data structures for representing sets depending when the operations one needs are those of a dictionary or a priority queue, automata are the right data structure for represent sets and relations when the required operations are union, intersection, complement, projections and
joins. In this view the *algorithmic implementation* of the operations must get the limelight, and, as a consequence, they constitute the spine of this book.

A second goal of this book is concerned with presentation. Experience tells that automata-theoretic constructions are best explained by means of examples, and that examples are best presented with the help of pictures. Automata on words are blessed with a graphical representation of instantaneous appeal. In this book we have invested much effort into finding illustrative, non-trivial examples whose graphical representation still fits in one page.
Chapter 1

Introduction and Goal

Courses on data structures show how to represent sets of objects in a computer so that operations like insertion, deletion, lookup, and many others can be efficiently implemented. Typical representations are hash tables, search trees, or heaps.

These lecture notes also deal with the problem of representing and manipulating sets, but with respect to a different set of operations: the boolean operations of set theory (union, intersection, and complement with respect to some universe set), plus some operations that check basic properties (if a set is empty, if it contains all elements of the universe, or if it is contained in another one), and operations on relations. Here is a more precise definition of the operations to be supported, where \( U \) denotes some universe of objects, \( X, Y \) are subsets of \( U \), \( x \) is an element of \( U \), and \( R, S \subseteq U \times U \) are binary relations on \( U \):

\[
\begin{align*}
\text{Member}(x, X) & : \text{ returns } \text{true} \text{ if } x \in X, \text{false otherwise.} \\
\text{Complement}(X) & : \text{ returns } U \setminus X. \\
\text{Intersection}(X, Y) & : \text{ returns } X \cap Y. \\
\text{Union}(X, Y) & : \text{ returns } X \cup Y. \\
\text{Empty}(X) & : \text{ returns } \text{true} \text{ if } X = \emptyset, \text{false otherwise.} \\
\text{Universal}(X) & : \text{ returns } \text{true} \text{ if } X = U, \text{false otherwise.} \\
\text{Included}(X, Y) & : \text{ returns } \text{true} \text{ if } X \subseteq Y, \text{false otherwise.} \\
\text{Equal}(X, Y) & : \text{ returns } \text{true} \text{ if } X = Y, \text{false otherwise.} \\
\text{Projection}_1(R) & : \text{ returns the set } \pi_1(R) = \{ x | \exists y \ (x, y) \in R \}. \\
\text{Projection}_2(R) & : \text{ returns the set } \pi_2(R) = \{ y | \exists x \ (x, y) \in R \}. \\
\text{Join}(R, S) & : \text{ returns } R \circ S = \{(x, z) | \exists y \in X (x, y) \in R \land (y, z) \in S \} \\
\text{Post}(X, R) & : \text{ returns } \text{post}_R(X) = \{ y \in U | \exists x \in X (x, y) \in R \}. \\
\text{Pre}(X, R) & : \text{ returns } \text{pre}_R(X) = \{ y \in U | \exists x \in X (y, x) \in R \}.
\end{align*}
\]

Observe that many other operations, e.g. set difference, can be reduced to the ones above. Similarly, operations on \( n \)-ary relations for \( n \geq 3 \) can be reduced to operations on binary relations.

An important point is that we are not only interested on finite sets, we wish to have a data
structure able to deal with infinite sets over some infinite universe. However, a simple cardinality argument shows that no data structure can provide finite representations of all infinite sets: an infinite universe has uncountably many subsets, but every data structure mapping sets to finite representations only has countably many instances. (Loosely speaking, there are more sets to be represented than representations available.) Because of this limitation every good data structure for infinite sets must find a reasonable compromise between expressibility (how large is the set of representable sets) and manipulability (which operations can be carried out, and at which cost). These notes present the compromise offered by word automata, which, as shown by 50 years of research on the theory of formal languages, is the best one available for most purposes. Word automata, or just automata, represent and manipulate sets whose elements are encoded as words, i.e., as sequences of letters over an alphabet\(^1\).

Any kind of object can be represented by a word, at least in principle. Natural numbers, for instance, are represented in computer science as sequences of digits, i.e., as words over the alphabet of digits. Vectors and lists can also be represented as words by concatenating the word representations of their elements. As a matter of fact, whenever a computer stores an object in a file, the computer is representing it as a word over some alphabet, like ASCII or Unicode. So word automata are a very general data structure. However, while any object can be represented by a word, not every object can be represented by a finite word, that is, a word of finite length. Typical examples are real numbers and non-terminating executions of a program. When objects cannot be represented by finite words, computers usually only represent some approximation: a float instead of a real number, or a finite prefix instead of a non-terminating computation. In the second part of the notes we show how to represent sets of infinite objects exactly using automata on infinite words. While the theory of automata on finite words is often considered a “gold standard” of theoretical computer science—a powerful and beautiful theory with lots of important applications in many fields—automata on infinite words are harder, and their theory does not achieve the same degree of “perfection”. This gives us a structure for Part II of the notes: we follow the steps of Part I, always comparing the solutions for infinite words with the “gold standard”.

\(^1\)There are generalizations of word automata in which objects are encoded as trees. The theory of tree automata is also very well developed, but not the subject of these notes. So we shorten word automaton to just automaton.
Part I

Automata on Finite Words
Chapter 2

Automata Classes and Conversions

In Section 2.1 we introduce basic definitions about words and languages, and then introduce regular expressions, a textual notation for defining languages of finite words. Like any other formal notation, it cannot be used to define each possible language. However, the next chapter shows that they are an adequate notation when dealing with automata, since they define exactly the languages that can be represented by automata on words.

2.1 Regular expressions: a language to describe languages

An alphabet is a finite, nonempty set. The elements of an alphabet are called letters. A finite, possibly empty sequence of letters is a word. A word $a_1a_2 \ldots a_n$ has length $n$. The empty word is the only word of length 0 and it is written $\epsilon$. The concatenation of two words $w_1 = a_1 \ldots a_n$ and $w_2 = b_1 \ldots b_m$ is the word $w_1w_2 = a_1 \ldots a_nb_1 \ldots b_m$, sometimes also denoted by $w_1 \cdot w_2$. Notice that $\epsilon \cdot w = w = w \cdot \epsilon = w$. For every word $w$, we define $w^0 = \epsilon$ and $w^{k+1} = w^k \cdot w$.

Given an alphabet $\Sigma$, we denote by $\Sigma^*$ the set of all words over $\Sigma$. A set $L \subseteq \Sigma^*$ of words is a language over $\Sigma$.

The complement of a language $L$ is the language $\Sigma^* \setminus L$, which we often denote by $\overline{L}$ (notice that this notation implicitly assumes the alphabet $\Sigma$ is fixed). The concatenation of two languages $L_1$ and $L_2$ is $L_1 \cdot L_2 = \{w_1w_2 \in \Sigma^* \mid w_1 \in L_1, w_2 \in L_2\}$. The iteration of a language $L \subseteq \Sigma^*$ is the language $L^* = \bigcup_{i \geq 0} L^i$, where $L_0 = \{\epsilon\}$ and $L^{i+1} = L^i \cdot L$ for every $i \geq 0$.

**Definition 2.1** Regular expressions $r$ over an alphabet $\Sigma$ are defined by the following grammar, where $a \in \Sigma$

$$r ::= \emptyset | \epsilon | a | r_1r_2 | r_1 + r_2 | r^*$$

The set of all regular expressions over $\Sigma$ is written $\mathcal{RE}(\Sigma)$. The language $L(r) \subseteq \Sigma^*$ of a regular expression $r \in \mathcal{RE}(\Sigma)$ is defined inductively by
CHAPTER 2. AUTOMATA CLASSES AND CONVERSIONS

\[ L(\emptyset) = \emptyset \quad L(r_1 r_2) = L(r_1) \cdot L(r_2) \quad L(r^*) = L(r) \]

A language \( L \) is regular if there is a regular expression \( r \) such that \( L = L(r) \).

We often abuse language, and write “the language \( r \)” instead of “the language \( L(r) \).”

2.2 Automata classes

We briefly recapitulate the definitions of deterministic and nondeterministic finite automata, as well as nondeterministic automata with \( \varepsilon \)-transitions and regular expressions.

**Definition 2.2** A deterministic automaton (DA) is a tuple \( A = (Q, \Sigma, \delta, q_0, F) \), where

- \( Q \) is a set of states,
- \( \Sigma \) is an alphabet,
- \( \delta : Q \times \Sigma \to Q \) is a transition function,
- \( q_0 \in Q \) is the initial state, and
- \( F \subseteq Q \) is the set of final states.

A run of \( A \) on input \( a_0 a_1 \ldots a_n \) is a sequence \( p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \ldots \xrightarrow{a_{n-1}} p_n \), such that \( p_i \in Q \) for \( 0 \leq i \leq n \), \( p_0 = q_0 \), and \( \delta(p_i, a_i) = p_{i+1} \) for \( 0 \leq i < n - 1 \). A run is accepting if \( p_n \in F \). A accepts a word \( w \in \Sigma^* \), if there is an accepting run on input \( w \). The language recognized by \( A \) is the set \( L(A) = \{ w \in \Sigma^* \mid w \text{ is accepted by } A \} \).

A deterministic finite automaton (DFA) is a DA with a finite set of states.

Notice that a DA has exactly one run on a given word. Given a DA, we often say “the word \( w \) leads from \( q_0 \) to \( q \)”, meaning that the unique run of the DA on the word \( w \) ends at the state \( q \).

Graphically, non-final states of a DFA are represented by circles, and final states by double circles (see the example below). The transition function is represented by labeled directed edges: if \( \delta(q, a) = q' \) then we draw an edge from \( q \) to \( q' \) labeled by \( a \). We also draw an edge into the initial state.

**Example 2.3** Figure 2.3 shows the graphical representation of the DFA \( A = (Q, \Sigma, \delta, q_0, F) \), where \( Q = \{ q_0, q_1, q_2, q_3 \} \), \( \Sigma = \{ a, b \} \), \( F = \{ q_0 \} \), and \( \delta \) is given by the following table

\[
\begin{align*}
\delta(q_0, a) &= q_1 \\
\delta(q_1, a) &= q_0 \\
\delta(q_2, a) &= q_3 \\
\delta(q_3, a) &= q_2 \\
\delta(q_0, b) &= q_3 \\
\delta(q_1, b) &= q_2 \\
\delta(q_2, b) &= q_1 \\
\delta(q_3, b) &= q_0
\end{align*}
\]

\[ \square \]
2.2. AUTOMATA CLASSES

2.2.1 Using DFAs as data structures

Here are four examples of how DFAs can be used to represent interesting sets of objects.

Example 2.4 The DFA of Figure 2.5 recognizes the strings over the alphabet \{-, \cdot, 0, 1, \ldots, 9\} that encode real numbers with a finite decimal part. We wish to exclude 002, −0, or 3.10000000, but accept 37, 10.503, or −0.234 as correct encodings. A description of the strings in English is rather long: a string encoding a number consists of an integer part, followed by a possibly empty fractional part; the integer part consists of an optional minus sign, followed by a nonempty sequence of digits; if the first digit of this sequence is 0, then the sequence itself is 0; if the fractional part is nonempty, then it starts with \cdot, followed by a nonempty sequence of digits that does not end with 0; if the integer part is −0, then the fractional part is nonempty.

Example 2.5 The DFA of Figure recognizes the binary encodings of all the multiples of 3. For
instance, it recognizes 11, 110, 1001, and 1100, which are the binary encodings of 3, 6, 9, and 12, respectively, but not, say, 10 or 111.

![DFA for the multiples of 3 in binary](image)

Figure 2.3: A DFA for the multiples of 3 in binary

**Example 2.6** The DFA of Figure 2.4 recognizes all the nonnegative integer solutions of the inequation $2x - y \leq 2$, using the following encoding. The alphabet of the DFA has four letters, namely

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

A word like

$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

encodes a pair of numbers, given by the top and bottom rows, 101100 and 010011. The binary encodings start with the least significant bit, that is

- 101100 encodes $2^0 + 2^2 + 2^3 = 13$, and
- 010011 encodes $2^2 + 2^5 + 2^6 = 50$.

We see this as an encoding of the valuation $(x, y) := (13, 50)$. This valuation satisfies the inequation, and indeed the word is accepted by the DFA.

**Example 2.7** Consider the following program with two boolean variables $x, y$:

```
1   while $x = 1$ do
2     if $y = 1$ then
3       $x \leftarrow 0$
4     $y \leftarrow 1 - x$
5   end
```

A configuration of the program is a triple $[\ell, n_x, n_y]$, where $\ell \in \{1, 2, 3, 4, 5\}$ is the current value of the program counter, and $n_x, n_y \in \{0, 1\}$ are the current values of $x$ and $y$. The initial configurations are $[1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]$, i.e., all configurations in which control is at line 1. The DFA of Figure 2.5 recognizes all reachable configurations of the program. For instance, the DFA accepts $[5, 0, 1]$, indicating that it is possible to reach the last line of the program with values $x = 0, y = 1$. 
2.2. AUTOMATA CLASSES

Definition 2.8 A non-deterministic automaton (NA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where $Q$, $\Sigma$, $q_0$, and $F$ are as for DAs and

- $\delta: Q \times \Sigma \to \mathcal{P}(Q)$ is a transition relation.

The runs of NAs are defined as for DAs, but substituting $p_{i+1} \in \delta(p_i, a_i)$ for $\delta(p_i, a_i) = p_{i+1}$.

Acceptance and the language recognized by a NA are defined as for DAs. A nondeterministic finite automaton (NFA) is a NA with a finite set of states.

We often identify the transition function $\delta$ of a DA with the set of triples $(q, a, q')$ such that $q' = \delta(q, a)$, and the transition relation $\delta$ of a NFA with the set of triples $(q, a, q')$ such that $q' \in \delta(q, a)$; so we often write $(q, a, q') \in \delta$, meaning $q' = \delta(q, a)$ for a DA, or $q' \in \delta(q, a)$ for a NA.

Definition 2.9 A non-deterministic automaton with $\varepsilon$-transitions (NA-\$\varepsilon\$) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where $Q$, $\Sigma$, $q_0$, and $F$ are as for NAs and...
\begin{itemize}
  \item \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q) \) is a transition relation.
\end{itemize}

The runs and accepting runs of NA-\(\varepsilon\) are defined as for NAs. A accepts a word \(a_1 \ldots a_n \in \Sigma^*\) if \(A\) has an accepting run on \(\varepsilon^{k_0} a_1 \varepsilon^{k_1} \ldots \varepsilon^{k_{n-1}} a_n \varepsilon^{k_n} \in (\Sigma \cup \{\varepsilon\})^*\) for some \(k_0, k_1, \ldots, k_n \geq 0\).

A nondeterministic finite automaton with \(\varepsilon\)-transitions (NFA-\(\varepsilon\)) is a NA-\(\varepsilon\) with a finite set of states.

**Definition 2.10** Let \(A = (Q, \Sigma, \delta, q_0, F)\) be an automaton. A state \(q \in Q\) is reachable from \(q' \in Q\) if \(q = q'\) or if there exists a run \(q' \xrightarrow{a_1} \ldots \xrightarrow{a_n} q\) on some input \(a_1 \ldots a_n \in \Sigma^*\). \(A\) is in normal form if every state is reachable from the initial state.

**Convention:** Unless otherwise stated, we assume that automata are in normal form.

All our algorithms preserve normal forms, i.e., when the output is an automaton, the automaton is in normal form.

We extend NAs to allow regular expressions on transitions. Such automata are called \(NA-reg\) and they are obviously a generalization of both regular expressions and NA-\(\varepsilon\)s. They will be useful to provide a uniform conversion between automata and regular expressions.

**Definition 2.11** A non-deterministic automaton with regular expression transitions (NA-reg) is a tuple \(A = (Q, \Sigma, \delta, q_0, F)\), where \(Q, \Sigma, q_0,\) and \(F\) are as for NAs, and where

\(\delta : Q \times \mathbb{RE}(\Sigma) \to \mathcal{P}(Q)\) is a relation such that \(\delta(q, r) = \emptyset\) for all but a finite number of pairs \((q, r) \in Q \times \mathbb{RE}(\Sigma)\).

Accepting runs are defined as for NFAs. A accepts a word \(w \in \Sigma^*\) if \(A\) has an accepting run on \(r_1 \ldots r_k\) such that \(w = L(r_1) \cdot \ldots \cdot L(r_k)\).

A nondeterministic finite automaton with regular expression transitions (NFA-reg) is a NA-reg with a finite set of states.

### 2.3 Conversion Algorithms between Finite Automata

We recall that all our data structures can represent exactly the same languages. Since DFAs are a special case of NFA, which are a special case of NFA-\(\varepsilon\), it suffices to show that every language recognized by an NFA-\(\varepsilon\) can also be recognized by an NFA, and every language recognized by an NFA can also be recognized by a DFA.

#### 2.3.1 From NFA to DFA.

The powerset construction transforms an NFA \(A\) into a DFA \(B\) recognizing the same language. We first give an informal idea of the construction. Recall that a NFA may have many different runs on a word \(w\), possibly leading to different states, while a DFA has exactly one run on \(w\). Denote by
2.3. CONVERSION ALGORITHMS BETWEEN FINITE AUTOMATA

$Q_w$ the set of states $q$ such that some run of $A$ on $w$ leads from its initial state $q_0$ to $q$. Intuitively, $B$ “keeps track” of the set $Q_w$: its states are sets of states of $A$, with $\{q_0\}$ as initial state, and its transition function is defined to ensure that the run of $B$ on $w$ leads from $\{q_0\}$ to $Q_w$ (see below). It is then easy to ensure that $A$ and $B$ recognize the same language: it suffices to choose the final states of $B$ as the sets of states of $A$ containing at least one final state, because for every word $w$:

- $B$ accepts $w$ iff $Q_w$ is a final state of $B$
- $B$ accepts $w$ iff $Q_w$ contains at least a final state of $A$
- $B$ accepts $w$ iff some run of $A$ on $w$ leads to a final state of $A$
- $B$ accepts $w$ iff $A$ accepts $w$.

Let us now define the transition function of $B$, say $\Delta$. “Keeping track of the set $Q_w$” amounts to satisfying $\Delta(Q_w, a) = Q_{wa}$ for every word $w$. But we have $Q_{wa} = \bigcup_{q \in Q_w} \delta(q, a)$, and so we define

$$\Delta(Q', a) = \bigcup_{q \in Q'} \delta(q, a)$$

for every $Q' \subseteq Q$. Notice that we may have $Q' = \emptyset$; in this case, $\emptyset$ is a state of $B$, and, since $\Delta(\emptyset, a) = \emptyset$ for every $a \in \Delta$, a “trap” state.

Summarizing, given $A = (Q, \Sigma, \delta, q_0, F)$ we define the DFA $B = (Q, \Sigma, \Delta, Q_0, F)$ as follows:

- $Q = \mathcal{P}(Q)$;
- $\Delta(Q', a) = \bigcup_{q \in Q'} \delta(q, a)$ for every $Q' \subseteq Q$ and every $a \in \Sigma$;
- $Q_0 = \{q_0\}$; and
- $F = \{Q' \in Q | Q' \cap F \neq \emptyset\}$.

Notice, however, that $B$ may not be in normal form: it may have many states non-reachable from $Q_0$. For instance, assume $A$ happens to be a DFA with states $\{q_0, \ldots, q_{n-1}\}$. Then $B$ has $2^n$ states, but only the singletons $\{q_0\}, \ldots, \{q_{n-1}\}$ are reachable. The conversion algorithm shown in Table 2.3.1 constructs only the reachable states. It is written in pseudocode, with abstract sets as data structure. Like nearly all the algorithms presented in the next chapters, it is a worklist algorithm. Worklist algorithms maintain a worklist of objects waiting to be processed. The name “worklist” is actually misleading, because the worklist is in fact a workset, it does not impose any order onto its elements, and it contains at most one copy of an element (i.e., if an element already in the workset is added to it again, the contents of the workset do not change). We however keep the name “worklist” for historical reasons.

In $\text{NFAtoDFA()}$ the worklist is called $W$, in other algorithms just $W$ (we use a calligraphic font to emphasize that in this case the objects of the worklist are sets). Worklist algorithms repeatedly pick an object from the worklist (instruction $\text{pick } Q \text{ from } W$), and process it; notice that picking
NFAtoDFA(A)
Input: NFA A = (Q, Σ, δ, q₀, F)
Output: DFA B = (Q, Σ, Δ, Q₀, F) with L(B) = L(A)
1. Q, Δ, F ← ∅; Q₀ ← {q₀}
2. W = {Q₀}
3. while W ≠ ∅ do
4. pick Q' from W
5. add Q' to Q
6. if Q' ∩ F ≠ ∅ then add Q' to F
7. for all a ∈ Σ do
8. Q'' ← ∪_{q ∈ Q'} δ(q, a)
9. if Q'' ∉ Q then add Q'' to W
10. add (Q', a, Q'') to Δ

Table 2.1: NFAtoDFA(A)

an object removes it from the worklist. Processing an object may generate new objects that are added to the list. The algorithm terminates when the worklist is empty. Since objects removed from the list may generate new objects, worklist algorithms may potentially fail to terminate, even if the set of all objects is finite, because the same object might be added to and removed from the worklist infinitely many times. Termination is guaranteed by making sure that no object that has been removed from the list once is ever added to it again. For this, objects picked from the worklist are stored (in NFAtoDFA() they are stored in Q), and objects are added to the worklist only if they have not been stored yet.

Figure 2.6 shows an NFA at the top, and some snapshots of the run of NFAtoDFA() on it. The states of the DFA are labelled with the corresponding sets of states of the NFA. The algorithm picks states from the worklist in order {1}, {1, 2}, {1, 3}, {1, 4}, {1, 2, 4}. Snapshots (a)-(d) are taken right after it picks the states {1, 2}, {1, 3}, {1, 4}, and {1, 2, 4}, respectively. Snapshot (e) is taken at the end. Notice that out of the \(2^4 = 16\) subsets of states of the NFA only 5 are constructed, because the rest are not reachable from {1}.

Complexity. If A has n states, then the output of NFAtoDFA(A) can have up to \(2^n\) states. To show that this bound is essentially reachable, consider the family \(\{L_n\}_{n≥1}\) of languages over \(\Sigma = \{a, b\}\) given by \(L_n = (a + b)^*a(a + b)^{(n−1)}\). That is, \(L_n\) contains the words of length at least \(n\) whose \(n\)-th letter starting from the end is an \(a\). The language \(L_n\) is accepted by the NFA with \(n + 1\) states shown in Figure 2.7(a): intuitively, the automaton chooses one of the \(a\)'s in the input word, and checks that it is followed by exactly \(n − 1\) letters before the word ends. Applying the subset construction, however, yields a DFA with \(2^n\) states. The DFA for \(L_3\) is shown on the left of Figure 2.7(b). The
2.3. CONVERSION ALGORITHMS BETWEEN FINITE AUTOMATA

Figure 2.6: Conversion of a NFA into a DFA.
CHAPTER 2. AUTOMATA CLASSES AND CONVERSIONS

(a) NFA for $L_n$

(b) DFA for $L_3$ and interpretation.

Figure 2.7: NFA for $L_n$, and DFA for $L_3$. 
2.3. CONVERSION ALGORITHMS BETWEEN FINITE AUTOMATA

states of the DFA have a natural interpretation: they “store” the last $n$ letters read by the automaton. If the DFA is in the state storing $a_1a_2\ldots a_n$ and it reads the letter $a_{n+1}$, then it moves to the state storing $a_2\ldots a_{n+1}$. States are final if the first letter they store is an $a$. The interpreted version of the DFA is shown on right of Figure 2.7(b).

We can also easily prove that any DFA recognizing $L_n$ must have at least $2^n$ states. Assume there is a DFA $A_n = (Q, \Sigma, \delta, q_0, F)$ such that $|Q| < 2^n$ and $L(A_n) = L_n$. We can extend $\delta$ to a mapping $\hat{\delta} : Q \times \{a, b\} \to Q$, where $\hat{\delta}(q, e) = q$ and $\hat{\delta}(q, w \cdot \sigma) = \delta(\hat{\delta}(q, w), \sigma)$ for all $w \in \Sigma^*$ and for all $\sigma \in \Sigma$. Since $|Q| < 2^n$, there must be two words $u \cdot a \cdot v_1$ and $u \cdot b \cdot v_2$ of length $n$ for which $\hat{\delta}(q_0, u \cdot a \cdot v_1) = \hat{\delta}(q_0, u \cdot b \cdot v_2)$. But then we would have that $\hat{\delta}(q_0, u \cdot a \cdot v_1) = \hat{\delta}(q_0, u \cdot b \cdot v_2 \cdot u)$; that is, either both $u \cdot a \cdot v_1 \cdot u$ and $u \cdot b \cdot v_2 \cdot u$ are accepted by $A_n$ or neither do. Since, however, $|a \cdot v_1 \cdot u| = |b \cdot v_2 \cdot u| = n$, this contradicts the assumption that $A_n$ consists of exactly the words with an $a$ at the $n$-th position from the end.

2.3.2 From NFA-ε to NFA.

Let $A$ be a NFA-ε over an alphabet $\Sigma$. In this section we use $a$ to denote an element of $\Sigma$, and $\alpha, \beta$ to denote elements of $\Sigma \cup \{\varepsilon\}$.

Loosely speaking, the conversion first adds to $A$ new transitions that make all ε-transitions redundant, without changing the recognized language: every word accepted by $A$ before adding the new transitions is accepted after adding them by a run without ε-transitions. The conversion then removes all ε-transitions, delivering an NFA that recognizes the same language as $A$.

The new transitions are shortcuts: If $A$ has transitions $(q, \alpha, q')$ and $(q', \beta, q'')$ such that $\alpha = \varepsilon$ or $\beta = \varepsilon$, then the shortcut $(q, \alpha\beta, q'')$ is added. (Notice that either $\alpha\beta = a$ for some $a \in \Sigma$, or $\alpha\beta = \varepsilon$.) Shortcuts may generate further shortcuts: for instance, if $\alpha\beta = a$ and $A$ has a further transition $(q'', a, q'''')$, then a new shortcut $(q, a, q''')$ is added. We call the process of adding all possible shortcuts saturation. Obviously, saturation does not change the language of $A$, and if before saturation $A$ has a run accepting a nonempty word, for example

$$
q_0 \overset{e}{\rightarrow} q_1 \overset{e}{\rightarrow} q_2 \overset{a}{\rightarrow} q_3 \overset{e}{\rightarrow} q_4 \overset{b}{\rightarrow} q_5 \overset{e}{\rightarrow} q_6
$$

then after saturation it has a run accepting the same word, and visiting no ε-transitions, namely

$$
q_0 \overset{a}{\rightarrow} q_4 \overset{b}{\rightarrow} q_6
$$

However, we cannot yet remove ε-transitions. The NFA-ε of Figure 2.8(a) accepts ε. After saturation we get the NFA-ε of Figure 2.8(b). However, removing all ε-transitions yields an NFA that no longer accepts ε. To solve this problem, if $A$ accepts ε, then we mark its initial state as final, which clearly does not change the language. To decide whether $A$ accepts ε, we check if some state reachable from $A$ by a sequence of ε-transitions is final. Figure 2.8(c) shows the final result. Notice that, in general, after removing ε-transitions the automaton may not be in normal form, because some states may no longer be reachable. So the naïve procedure runs in three phases: saturation, ε-check, and normalization.
CHAPTER 2. AUTOMATA CLASSES AND CONVERSIONS

(a) NFA-\(\varepsilon\) accepting \(L(0^*1^*2^*)\)

(b) After saturation

(c) After marking the initial state and final and removing all \(\varepsilon\)-transitions.

Figure 2.8: Conversion of an NFA-\(\varepsilon\) into an NFA by shortcutting \(\varepsilon\)-transitions.

However, it is possible to carry all three steps in a single pass. We give a worklist implementation of this procedure in which the check is done while saturating, and only the reachable states are generated (in the pseudocode \(\alpha\) and \(\beta\) stand for either a letter of \(\Sigma\) or \(\varepsilon\), and \(a\) stands for a letter of \(\Sigma\)). Furthermore, the algorithm avoids constructing some redundant shortcuts. For instance, for the NFA-\(\varepsilon\) of Figure 2.8(a) the algorithm does not construct the transition labeled by 2 leading from the state in the middle to the state on the right.
2.3. CONVERSION ALGORITHMS BETWEEN FINITE AUTOMATA

\( \text{NFA} \rightarrow \text{NFA}(A) \)

**Input:** NFA-ε \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** NFA \( B = (Q', \Sigma, \delta', q'_0, F') \) with \( L(B) = L(A) \)

1. \( q'_0 \leftarrow q_0 \)
2. \( Q' \leftarrow \{q_0\}; \delta' \leftarrow \emptyset; F' \leftarrow F \cap \{q_0\} \)
3. \( \delta'' \leftarrow \emptyset; W \leftarrow \{(q, \alpha, q') \in \delta \mid q = q_0\} \)
4. **while** \( W \neq \emptyset \) **do**
   
   **pick** \((q_1, \alpha, q_2)\) from \( W \)
   
   **if** \( \alpha \neq \varepsilon \) **then**
   
   **add** \( q_2 \) **to** \( Q' \); **add** \((q_1, \alpha, q_2)\) **to** \( \delta' \); **if** \( q_2 \in F \) **then** **add** \( q_2 \) **to** \( F' \)
   
   **for all** \( q_3 \in \delta(q_2, \varepsilon) \) **do**
   
   **if** \((q_1, \alpha, q_3) \notin \delta' \) **then** **add** \((q_1, \alpha, q_3)\) **to** \( W \)
   
   **for all** \( \alpha \in \Sigma, q_3 \in \delta(q_2, \alpha) \) **do**
   
   **if** \((q_2, \alpha, q_3) \notin \delta' \) **then** **add** \((q_2, \alpha, q_3)\) **to** \( W \)
   
   **else** / \( * \alpha = \varepsilon * / \)
   
   **add** \((q_1, \alpha, q_2)\) **to** \( \delta'' \); **if** \( q_2 \in F \) **then** **add** \( q_0 \) **to** \( F' \)
   
   **for all** \( \beta \in \Sigma \cup \{\varepsilon\}, q_3 \in \delta(q_2, \beta) \) **do**
   
   **if** \((q_1, \beta, q_3) \notin \delta' \cup \delta'' \) **then** **add** \((q_1, \beta, q_3)\) **to** \( W \)

The correctness proof is conceptually easy, but the different cases require some care, and so we devote it a proposition.

**Proposition 2.12** Let \( A \) be a NFA-ε, and let \( B = \text{NFA} \rightarrow \text{NFA}(A) \). Then \( B \) is a NFA and \( L(A) = L(B) \).

**Proof:** Notice first that every transition that leaves \( W \) is never added to \( W \) again: when a transition \((q_1, \alpha, q_2)\) leaves \( W \) it is added to either \( \delta' \) or \( \delta'' \), and a transition enters \( W \) only if it does not belong to either \( \delta' \) or \( \delta'' \). Since every execution of the while loop removes a transition from the worklist, the algorithm eventually exits the loop and terminates.

To show that \( B \) is a NFA we have to prove that it only has non-ε transitions, and that it is in normal form, i.e., that every state of \( Q' \) is reachable from \( q_0 \) in \( B \). For the first part, observe that transitions are only added to \( \delta' \) in line 7, and none of them is an ε-transition because of the guard in line 6. For the second part, we need the following invariant, which can be easily proved by inspection: for every transition \((q_1, \alpha, q_2)\) added to \( W \), if \( \alpha = \varepsilon \) then \( q_1 = q_0 \), and if \( \alpha \neq \varepsilon \), then \( q_2 \) is reachable in \( B \) (after termination). Now, since new states are added to \( Q' \) only at line 7, applying the invariant we get that every state of \( Q' \) is reachable from \( q_0 \) in \( B \). It remains to prove \( L(A) = L(B) \). The inclusion \( L(A) \supseteq L(B) \) follows from the fact that every transition added to \( \delta' \) is a shortcut, which can be proved by inspection. For the inclusion \( L(A) \subseteq L(B) \), we first prove that \( \varepsilon \in L(A) \) implies \( \varepsilon \in L(B) \). Let \( q_0 \xrightarrow{\varepsilon} q_1 \ldots q_{n-1} \xrightarrow{\varepsilon} q_n \) be a run of \( A \) such that \( q_n \in F \). If \( n = 0 \) (i.e., \( q_n = q_0 \)), then we are done. If \( n > 0 \), then we prove by induction on \( n \) that a transition
(q₀, ε, qₙ) is eventually added to W (and so eventually picked from it), which implies that q₀ is eventually added to F' at line 13. If n = 1, then (q₀, ε, qₙ) is added to W at line 3. If n > 1, then by hypothesis (q₀, ε, qₙ₋₁) is eventually added to W, picked from it at some later point, and so (q₀, ε, qₙ) is added to W at line 15. We now prove that for every w ∈ Σ⁺, if w ∈ L(A) then w ∈ L(B).

Let w = a₁a₂...aₙ with n ≥ 1. Then A has a run

\[ q₀ \xrightarrow{ε} \ldots \xrightarrow{ε} q_{m₁} \xrightarrow{a₁} q_{m₁+1} \xrightarrow{ε} \ldots \xrightarrow{ε} q_{mₙ} \rightarrow q_{mₙ+1} \rightarrow \ldots \rightarrow q_{m} \]

such that qₘ ∈ F. We have just proved that a transition (q₀, ε, qₘ₁) is eventually added to W. So (q₀, a₁, qₘ₁⁺₁) is eventually added at line 15, (q₀, a₁, qₘ⁺₂),..., (q₀, a₁, qₘₙ) are eventually added at line 9, and (qₘ₂, a₂, qₘ₂⁺₁) is eventually added at line 11. Iterating this argument, we obtain that

\[ q₀ \xrightarrow{a₁} q_{m₂} \xrightarrow{a₂} q_{m₃} \ldots q_{mₙ} \xrightarrow{aₙ} q_{m} \]

is a run of B. Moreover, qₘ is added to F' at line 7, and so w ∈ L(B).

**Complexity.** Observe that the algorithm processes pairs of transitions (q₁, α, q₂), (q₂, β, q₃), where (q₁, α, q₂) comes from W and (q₂, β, q₃) from δ (lines 8, 10, 14). Since every transition is removed from W at most once, the algorithm processes at most |Q| · |Σ| · |δ| pairs (because for a fixed transition (q₂, β, q₃) ∈ δ there are |Q| possibilities for q₁ and |Σ| possibilities for α). The runtime is dominated by the processing of the pairs, and so it is \( O(|Q| \cdot |Σ| \cdot |δ|) \).

### 2.4 Conversion algorithms between regular expressions and automata

To convert regular expressions to automata and vice versa we use NFA-regs as introduced in Definition 2.11. Both NFA-ε’s and regular expressions can be seen as subclasses of NFA-regs: an NFA-ε is an NFA-reg whose transitions are labeled by letters or by ε, and a regular expression r “is” the NFA-reg A, having two states, the one initial and the other final, and a single transition labeled r leading from the initial to the final state.

We present algorithms that, given an NFA-reg belonging to one of this subclasses, produces a sequence of NFA-regs, each one recognizing the same language as its predecessor in the sequence, and ending in an NFA-reg of the other subclass.

#### 2.4.1 From regular expressions to NFA-ε’s

Given a regular expression s over alphabet Σ, it is convenient do some preprocessing by exhaustively applying the following rewrite rules:

\[ \emptyset \cdot r \leadsto \emptyset \quad r \cdot \emptyset \leadsto \emptyset \]
\[ r + \emptyset \leadsto r \quad \emptyset + r \leadsto r \]
\[ \emptyset^* \leadsto \varepsilon \]
Since the left- and right-hand-sides of each rule denote the same language, the result of the pre-
processing is a regular expression for the same language as the original one. Moreover, if \( r \) is the
resulting regular expression, then either \( r = \emptyset \), or \( r \) does not contain any occurrence of the \( \emptyset \) symbol.
In the former case, we can directly produce an NFA-\( \varepsilon \). In the second, we transform the NFA-reg
\( A_r \) into an equivalent NFA-\( \varepsilon \) by exhaustively applying the transformation rules of Figure 2.9.

\[
\begin{array}{c}
\text{Automaton for the regular expression } a, \text{ where } a \in \Sigma \cup \{\varepsilon\} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Rule for concatenation} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Rule for choice} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Rule for Kleene iteration} \\
\end{array}
\]

Figure 2.9: Rules converting a regular expression given as NFA-reg into an NFA-\( \varepsilon \).

It is easy to see that each rule preserves the recognized language (i.e., the NFA-regs before and
after the application of the rule recognize the same language). Moreover, since each rule splits a
regular expression into its constituents, we eventually reach an NFA-reg to which no rule can be
applied. Furthermore, since the initial regular expression does not contain any occurrence of the \( \emptyset \) symbol, this NFA-reg is necessarily an NFA-\( \varepsilon \).

Example 2.13 Consider the regular expression \((a^*b^* + c)^*d\). The result of applying the transfor-
mation rules is shown in Figure 2.10 on page 30.

Complexity. It follows immediately from the rules that the final NFA-\( \varepsilon \) has the two states of \( A_r \),
plus one state for each occurrence of the concatenation or the Kleene iteration operators in \( r \). The
number of transitions is linear in the number of symbols of \( r \). The conversion runs in linear time.
Figure 2.10: The result of converting \((a^*b^* + c)^*d\) into an NFA-\(\varepsilon\).
2.4. CONVERSION ALGORITHMS BETWEEN REGULAR EXPRESSIONS AND AUTOMATA

2.4.2 From NFA-ε’s to regular expressions

Given an NFA-ε A, we transform it into the NFA-reg A_r for some regular expression r. It is again convenient to apply some preprocessing to guarantee that the NFA-ε has one single final state, and that this final state has no outgoing transition:

- Add a new state q_f and ε-transitions leading from each final state to q_f, and replace the set of final states by \{q_f\}. Graphically:

Rule 1

After preprocessing, the algorithm runs in phases. Each phase consist of two steps. The first step yields an automaton with at most one transition between any two given states:

- Repeat exhaustively: replace a pair of transitions \((q, r_1, q'), (q, r_2, q')\) by a single transition \((q, r_1 + r_2, q')\).

Rule 2

The second step reduces the number of states by one, unless the only states left are the initial state and the final state q_f.

- Pick a non-final and non-initial state q, and shortcut it: If q has a self-loop \((q, r, q)^1\), replace each pair of transitions \((q', s, q), (q, t, q'')\) (where \(q' \neq q \neq q''\), but possibly \(q' = q''\)) by a shortcut \((q', sr^*t, q'')\); otherwise, replace it by the shortcut \((q', st, q'')\). After shortcutting all pairs, remove q. Graphically:

Rule 3

---

^1Notice that it can have at most one, because otherwise we would have two parallel edges.
At the end of the last phase we have now an NFA-reg with exactly two states, the initial state \( q_0 \) (which is non-final) and the final state \( q_f \). Moreover, \( q_f \) has no outgoing transitions, because it was initially so and the application of the rules cannot change it. After apply Rule 2 exhaustively to guarantee that there is at most one transition from \( q_0 \) to itself, and from \( q_0 \) to \( q_f \), we are left with an NFA-reg having at most two transitions left: a transition \((q_0, r_1, q_f)\), plus possibly a self-loop \((q_0, r_2, q_0)\). If there is no self-loop, then we are done, the NFA-reg is \( A_{r_1} \). Otherwise we apply some post-processing:

- If the automaton has a self-loop \((q_0, r_2, q_0)\), remove it, and replace the transition \((q_0, r_1, q_f)\) by \((q_0, r^*_2 r_1, q_f)\). The resulting automaton is \( A_{r^*_2 r_1} \).

The complete algorithm is:

\[
\text{NFA}\text{to}\text{RE}(A)
\]
**Input:** NFA-\( \varepsilon \) \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** regular expression \( r \) with \( L(r) = L(A) \)

1. apply Rule 1
2. while \( Q' \setminus (F \cup \{q_0\}) \neq \emptyset \) do
3. pick \( q \) from \( Q \setminus (F \cup \{q_0\}) \)
4. apply exhaustively Rule 2
5. apply Rule 3 to \( q \)
6. apply exhaustively Rule 2
7. if there is a transition from \( q_0 \) to itself then apply Rule 4
8. return the label of the (unique) transition

**Example 2.14** Consider the automaton of Figure 2.11(a) on page 33. It is a DFA for the language of all words over the alphabet \( \{a, b\} \) that contain an even number of \( a \)'s and an even number of \( b \)'s. (The DFA is in the states on the left, respectively on the right, if it has read an even, respectively an odd, number of \( a \)'s. Similarly, it is in the states at the top, respectively at the bottom, if it has read an even, respectively an odd, number of \( b \)'s.) The rest of the figure shows some snapshots of the run of \( \text{NFA}\text{to}\text{RE}() \) on this automaton. Snapshot (b) is taken right after applying rule 1. Snapshots (c) to (e) are taken after each execution of the body of the while loop. Snapshot (f) shows the final result.
2.4. CONVERSION ALGORITHMS BETWEEN REGULAR EXPRESSIONS AND AUTOMATA

Figure 2.11: Run of $NFA\rightarrow RE()$ on a DFA
Complexity. The complexity of this algorithm depends on the data structure used to store regular expressions. If regular expresions are stored as strings or trees (following the syntax tree of the expression), then the complexity can be exponential. To see this, consider for each \( n \geq 1 \) the NFA \( A = (Q, \Sigma, \delta, q_0, F) \) where \( Q = \{q_0, \ldots, q_{n-1}\} \), \( \Sigma = \{a_{ij} \mid 0 \leq i, j \leq n-1\} \), \( \delta = \{(q_i, a_{ij}, q_j) \mid 0 \leq i, j \leq n-1\} \), and \( F = \{q_0\} \). By symmetry, the runtime of the algorithm is independent of the order in which states are eliminated. Consider the order \( q_1, q_2, \ldots, q_{n-1} \). It is easy to see that after eliminating the state \( q_i \) the NFA-reg contains some transitions labeled by regular expressions with \( 3^i \) occurrences of letters. The exponential blowup cannot be avoided: It can be shown that every regular expression recognizing the same language as \( A \) contains at least \( 2^{(n-1)} \) occurrences of letters.

If regular expressions are stored as acyclic directed graphs (the result of sharing common subexpressions in the syntax tree), then the algorithm works in polynomial time, because the label for a new transition is obtained by concatenating aor starring already computed labels.

2.5 A Tour of Conversions

We present an example illustrating all conversions of this chapter. We start with the DFA of Figure 2.11(a) recognizing the words over \( \{a, b\} \) with an even number of \( a \)’s and an even number of \( b \)’s. The figure converts it into a regular expression. Now we convert this expression into a NFA-\( \varepsilon \): Figure 2.12 on page 35 shows four snapshots of the process of applying rules 1 to 4. In the next step we convert the NFA-\( \varepsilon \) into an NFA. The result is shown in Figure 2.13 on page 36. Finally, we transform the NFA into a DFA by means of the subset construction. The result is shown in Figure 2.14. Observe that we do not recover the DFA we started with, but another one recognizing the same language. A last step allowing us to close the circle is presented in the next chapter.

Exercises

Exercise 1 Transform this DFA

\[
\begin{array}{ccc}
& a & \\
& b & \rightarrow \\
\text{start} & a & b \\
& b & \\
& b & \\
& a & b
\end{array}
\]

into an equivalent regular expression, then transform this expression into an NFA (with \( \varepsilon \)-transitions), remove the \( \varepsilon \)-transitions, and determinize the automaton.

Exercise 2 Prove or disprove: the languages of the regular expressions \( (1 + 10)^* \) and \( 1*(101^*)^* \) are equal.
2.5. A TOUR OF CONVERSIONS

Figure 2.12: Constructing a NFA-$\varepsilon$ for $(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$
Exercise 3  Give a regular expression for the words over \{0, 1\} that do not contain 010 as subword.

Exercise 4  Which regular expressions \( r \) satisfy the implication \((rr = r \Rightarrow r = r^*)\) ?

Exercise 5  (R. Majumdar) Consider a deck of cards (with arbitrary many cards) in which black and red cards alternate, and the top card is black. Cut the deck at any point into two piles, and then perform a rifflle (also called a dovetail shuffle) to yield a new deck. E.g., we can cut a deck with six cards 123456 into two piles 12 and 3456, and the rifflle yields 132456 or 314256, depending on the pile we start the rifflle with. Now, take the cards from the new deck two at a time (if the rifflle yields 132456, then this exposes the cards 3, 4, and 6; if it yields 314256, then the cards 1, 2, and 6). Prove with the help of regular expressions that all the exposed cards have the same color.

Hint: The expression \((BR)^*(\epsilon + B)\) represents the possible initial decks. Give a regular expression for the set of possible decks after the rifflle.

Exercise 6  Extend the syntax and semantics of regular expressions as follows: If \( r \) and \( r' \) are regular expressions over \( \Sigma \), then \( \overline{r} \) and \( r \cap r' \) are also regular expressions, where \( L(r) = L(r) \) and \( L(r \cap r') = L(r) \cap L(r') \). A language \( L \subseteq \Sigma^* \) is called star-free if there exists an extended regular expression \( r \) without any occurrence of a Kleene star operation such that \( L = L(r) \). For example,
2.5. A TOUR OF CONVERSIONS

\[\Sigma^*\] is star-free, because \(\Sigma^* = L(\emptyset)\). Show that \(L((01 + 10)^*)\) is star-free.

**Exercise 7** Prove or disprove: Every regular language is recognized by a NFA . . .

- . . . having one single final state.
- . . . whose states are all final.
- . . . whose initial state has no incoming transitions.
- . . . whose final state has no outgoing transitions.

Which of the above hold for DFAs? Which ones for NFA-\(\epsilon\)?

**Exercise 8** Given a language \(L\), let \(L_{\text{pref}}\) and \(L_{\text{suf}}\) denote the languages of all prefixes and all suffixes, respectively, of words in \(L\). E.g. for \(L = \{abc, d\}\), we have \(L_{\text{pref}} = \{abc, ab, a, \epsilon, d\}\) and \(L_{\text{suf}} = \{abc, bc, c, \epsilon, d\}\).

1. Given an automaton \(A\), construct automata \(A_{\text{pref}}\) and \(A_{\text{suf}}\) so that \(L(A_{\text{pref}}) = L(A)_{\text{pref}}\) and \(L(A_{\text{suf}}) = L(A)_{\text{suf}}\).

2. Consider the regular expression \(r = (ab + b)^*c\). Give a regular expression \(r_{\text{pref}}\) so that \(L(r_{\text{pref}}) = L(r)_{\text{pref}}\).

3. More generally, give an algorithm that receives a regular expression \(r\) as input and returns a regular expression \(r_{\text{pref}}\) so that \(L(r_{\text{pref}}) = L(r)_{\text{pref}}\).
Exercise 9 Recall that a nondeterministic automaton $A$ accepts a word $w$ if at least one of the runs of $A$ on $w$ is accepting. This is sometimes called the existential accepting condition. Consider the variant in which $A$ accepts $w$ if all runs of $A$ on $w$ are accepting (in particular, if $A$ has no run on $w$ then it accepts $w$). This is called the universal accepting condition. Notice that a DFA accepts the same language with both the existential and the universal accepting conditions.

Give an algorithm that transforms an automaton with universal accepting condition into a DFA recognizing the same language.

Exercise 10 Consider the family $L_n$ of languages over the alphabet $\{0, 1\}$ given by $L_n = \{ww \in \Sigma^{2n} | w \in \Sigma^n\}$.

- Give an automaton of size $O(n)$ with universal accepting condition that recognizes $L_n$.
- Prove that every NFA (and so in particular every DFA) recognizing $L_n$ has at least $O(2^n)$ states.

Exercise 11 The existential and universal accepting conditions can be combined, yielding alternating automata. The states of an alternating automaton are partitioned into existential and universal states. An existential state $q$ accepts a word $w$ (i.e., $w \in L(q)$) if $w = \varepsilon$ and $q \in F$ or $w = aw'$ and there exists a transition $(q, a, q')$ such that $q'$ accepts $w'$. A universal state $q$ accepts a word $w$ if $w = \varepsilon$ and $q \in F$ or $w = aw'$ and for every transition $(q, a, q')$ the state $q'$ accepts $w'$. The language recognized by an alternating automaton is the set of words accepted by its initial state.

Give an algorithm that transforms an alternating automaton into a DFA recognizing the same language.

Exercise 12 Let $L$ be an arbitrary language over a 1-letter alphabet. Prove that $L'$ is regular.

Exercise 13 In algorithm $NFAeNFA$ for the removal of $\varepsilon$-transitions, no transition that has been added to the worklist, processed and removed from the worklist is ever added to the worklist again. However, transitions may be added to the worklist more than once. Give a NFA-\$e\$ $A$ and a run of $NFAeNFA(A)$ in which this happens.

Exercise 14 We say that $u = a_1 \cdots a_n$ is a scattered subword of $w$, denoted by $u \triangleleft w$, if there are words $w_0, \ldots, w_n \in \Sigma^*$ such that $w = w_0a_1w_1a_2 \cdots a_nw_n$. The upward closure of a language $L$ is the language $L \uparrow = \{u \in \Sigma^* | \exists w \in L : w \triangleleft u\}$. The downward closure of $L$ is the language $L \downarrow = \{u \in \Sigma^* | \exists w \in L : u \triangleleft w\}$. Give algorithms that take a NFA $A$ as input and return NFAs for $L(A) \uparrow$ and $L(A) \downarrow$, respectively.
Exercise 15  Algorithm NFAtoRE transforms a finite automaton into a regular expression representing the same language by iteratively eliminating states of the automaton. In this exercise we see that eliminating states corresponds to eliminating variables from a system of language equations.

1. Arden’s Lemma states that given two languages $A, B \subseteq \Sigma^*$ with $\varepsilon \notin A$, the smallest language $X \subseteq \Sigma^*$ satisfying $X = AX \cup B$ (i.e., the smallest language $L$ such that $AL \cup B$ is again $L$, where $AL$ is the concatenation of $A$ and $L$) is the language $A^*B$. Prove Arden’s Lemma.

2. Consider the following system of equations, where the variables $X, Y$ represent languages over the alphabet $\Sigma = \{a, b, c, d, e, f\}$, the operator $\cdot$ represents language concatenation, and $\cup$ represents set union.

\[
X = \{a\}Y \cup \{b\} \cdot Y \cup \{c\}
\]

\[
Y = \{d\}X \cup \{e\} \cdot Y \cup \{f\}.
\]

(Here $\{a\}X$ denotes the concatenation of the language $\{a\}$ containing only the word $a$ and $X$.) This system has many solutions. For example, $X, Y = \Sigma^*$ is a solution. But there is a unique minimal solution, i.e., a solution contained in every other solution. Find the smallest solution with the help of Arden’s Lemma.

**Hint:** In a first step, consider $X$ not as a variable, but as a constant language, and solve the equation for $Y$ using Arden’s Lemma.

We can associate to any NFA $A = (Q, \Sigma, \delta, q_I, F)$ a system of linear equations as follows. We take as variables the states of the automaton, which we call here $X, Y, Z, \ldots$, with $X$ as initial state. The system has an equation for each state of the form

\[
X = \bigcup_{(X, a, Y) \in \delta} \{a\}Y
\]

if $X \notin F$, or

\[
X = \left( \bigcup_{(X, a, Y) \in \delta} \{a\} \cdot Y \right) \cup \{\varepsilon\}
\]

if $X \in F$.

3. Consider the DFA of Figure 2.11(a). Let $X, Y, Z, W$ be the states of the automaton, read from top to bottom and from left to right. The associated system of linear equations is

\[
X = \{a\}Y \cup \{b\}Z \cup \{\varepsilon\}
\]

\[
Y = \{a\}X \cup \{b\}W
\]

\[
Z = \{b\}X \cup \{a\}W
\]

\[
W = \{b\}Y \cup \{a\}Z
\]

Calculate the solution of this linear system by iteratively eliminating variables. Start with $Y$, then eliminate $Z$, and finally $W$. Compare with the elimination procedure shown in Figure 2.11.
Exercise 16  Given \( n \in \mathbb{N}_0 \), let \( \text{msbf}(n) \) be the set of *most-significant-bit-first* encodings of \( n \), i.e., the words that start with an arbitrary number of leading zeros, followed by \( n \) written in binary. For example:

\[
\text{msbf}(3) = 0^*11 \quad \text{and} \quad \text{msbf}(9) = 0^*1001 \quad \text{msbf}(0) = 0^*.
\]

Similarly, let \( \text{lsbf}(n) \) denote the set of *least-significant-bit-first* encodings of \( n \), i.e., the set containing for each word \( w \in \text{msbf}(n) \) its reverse. For example:

\[
\text{lsbf}(6) = L(0110^*) \quad \text{and} \quad \text{lsbf}(0) = L(0^*).
\]

1. Construct and compare DFAs recognizing the encodings of the even numbers \( n \in \mathbb{N}_0 \) w.r.t. the unary encoding, where \( n \) is encoded by the word \( 1^n \), the msbf-encoding, and the lsbf-encoding.

2. Same for the set of numbers divisible by 3.

3. Give regular expressions corresponding to the languages in (a) and (b).

Exercise 17

- Give a regular expression of size \( O(n) \) such that the smallest DFA equivalent to it has \( O(2^n) \) states.

- Give a regular expression of size \( O(n) \) without + such that the smallest DFA equivalent to it has \( O(2^n) \) states.

- Give a regular expression of size \( O(n) \) without + and of start-height 1 such that the smallest DFA equivalent to it has \( O(2^n) \) states. (Paper by Shallit at STACS 2008).
Chapter 3

Minimization and Reduction

In the previous chapter we showed through a chain of conversions that the two DFAs of Figure 3.1 recognize the same language. Obviously, the automaton on the left of the figure is better as a data structure for this language, since it has smaller size. A DFA (respectively, NFA) is minimal if no other DFA (respectively, NFA) recognizing the same language has fewer states. We show that every regular language has a unique minimal DFA up to isomorphism (i.e., up to renaming of the states). and present an efficient algorithm that “minimizes” a given DFA, i.e., converts it into the unique minimal DFA. In particular, the algorithm converts the DFA on the right of Figure 3.1 into the one on the left.

From a data structure point of view, the existence of a unique minimal DFA has two important consequences. First, as mentioned above, the minimal DFA is the one that can be stored with a minimal amount of memory. Second, the uniqueness of the minimal DFA makes it a canonical representation of a regular language. As we shall see, canonicity leads to a fast equality check: In order to decide if two regular languages are equal, we can construct their minimal DFAs, and check

Figure 3.1: Two DFAs for the same language
if they are isomorphic.

In the second part of the chapter we show that, unfortunately, computing a minimal NFA is a PSPACE complete problem, for which no efficient algorithm is likely to exist. Moreover, the minimal NFA is not unique. However, we show that a generalization of the minimization algorithm for DFAs can be used to at least reduce the size of an NFA while preserving its language.

3.1 Minimal DFAs

We start with a simple but very useful definition.

Definition 3.1 Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, the $w$-residual of $L$ is the language $L^w = \{ u \in \Sigma^* \mid Wu \in L \}$. A language $L' \subseteq \Sigma^*$ is a residual of $L$ if $L' = L^w$ for at least one $w \in \Sigma^*$.

A language may be infinite but have a finite number of residuals. An example is the language of the DFAs in Figure 3.1. Recall it is the language of all words over \{a, b\} with an even number of a’s and an even number of b’s, which we denote in what follows by EE. The language has four residuals, namely EE, EO, OE, OO, where EO contains the words with an even number of a’s and an odd number of b’s, etc. For example, we have $EE^e = EE, EE^a = EE^{aaa} = OE$, and $EE^{ab} = OO$.

There is a close connection between the states of a DA (not necessarily finite) and the residuals of its language. In order to formulate it we introduce the following definition:

Definition 3.2 Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DA and let $q \in Q$. The language recognized by $q$, denoted by $L_A(q)$, or just $L(q)$ if there is no risk of confusion, is the language recognized by $A$ with $q$ as initial state, i.e., the language recognized by the DA $(Q, \Sigma, \delta, q, F)$.

Figure 3.2 shows the result of labeling the states of the two DFAs of Figure 3.1 with the languages they recognize, which are residuals of EE.

Lemma 3.3 Let $L$ be a language and let $A = (Q, \Sigma, \delta, q_0, F)$ be a DA recognizing $L$.

1. Every residual of $L$ is recognized by some state of $A$. Formally: for every $w \in \Sigma^*$ there is $q \in Q$ such that $L_A(q) = L^w$.

2. Every state of $A$ recognizes a residual of $L$. Formally: for every $q \in Q$ there is $w \in \Sigma^*$ such that $L_A(q) = L^w$.

Proof: (1) Let $w \in \Sigma^*$, and let $q$ be the state reached by the unique run of $A$ on $w$. Then a word $wu \in \Sigma^*$ is recognized by $A$ iff $w$ is recognized by $A$ with $q$ as initial state. So $L_A(q) = L^w$.

(2) Since $A$ is in normal form, $q$ can be reached from $q_0$ by at least a word $w$. So $L_A(q) = L^w$, and we are done.
Figure 3.2: Languages recognized by the states of the DFAs of Figure 3.1.

The notion of residuals allows to define the canonical deterministic automaton for a language $L \subseteq \Sigma^*$.

**Definition 3.4** Let $L \subseteq \Sigma^*$ be a language. The canonical DA for $L$ is the DA $C_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$, where:

- $Q_L$ is the set of residuals of $L$; i.e., $Q_L = \{ L^w | w \in \Sigma^* \}$;
- $\delta(K, a) = K^a$ for every $K \in Q_L$ and $a \in \Sigma$;
- $q_{0L} = L$; and
- $F_L = \{ K \in Q_L | \varepsilon \in K \}$.

Notice that the number of states of $C_L$ is equal to the number of residuals of $L$, and both may be infinite.

**Example 3.5** The canonical DA for the language $EE$ is shown on the left of Figure 3.2. It has four states, corresponding to the four residuals of $EE$. Since, for instance, we have $EE^a = OE$, we have a transition labeled by $a$ leading from $EE$ to $OE$.  

It is intuitively clear, and easy to show, that the canonical DA for a language $L$ recognizes $L$.

**Proposition 3.6** For every language $L \subseteq \Sigma^*$, the canonical DA for $L$ recognizes $L$.

**Proof:** Let $C_L$ be the canonical DA for $L$. We prove $L(C_L) = L$.

Let $w \in \Sigma^*$. We prove by induction on $|w|$ that $w \in L$ iff $w \in L(C_L)$.  

If $|w| = 0$ then $w = \varepsilon$, and we have

\[
\begin{align*}
\varepsilon &\in L \quad (w = \varepsilon) \\
\iff 
L &\in F_L \quad (\text{definition of } F_L) \\
\iff 
q_{0L} &\in F_L \quad (q_{0L} = L) \\
\iff 
\varepsilon &\in L(C_L) \quad (q_{0L} \text{ is the initial state of } C_L)
\end{align*}
\]

If $|w| > 0$, then $w = aw'$ for some $a \in \Sigma$ and $w' \in \Sigma^*$, and we have

\[
\begin{align*}
aw' &\in L \\
\iff 
w' &\in L^a \quad (\text{definition of } L^a) \\
\iff 
w' &\in L(C_L^a) \quad (\text{induction hypothesis}) \\
\iff 
aw' &\in L(C_L) \quad (\delta_L(L, \alpha) = L^a)
\end{align*}
\]

We now prove that $C_L$ is the unique minimal DFA recognizing a regular language $L$ (up to isomorphism). The informal argument goes as follows. Since every DFA for $L$ has at least one state for each residual and $C_L$ has exactly one state for each residual, $C_L$ is minimal, and every other minimal DFA for $L$ also has exactly one state for each residual. But the transitions, initial and final states of a minimal DFA are completely determined by the residuals of the states: if state $q$ recognizes residual $R$, then the $a$-transition leaving $q$ necessarily leads to the state recognizing $R^a$; $q$ is final iff $\varepsilon \in R$, and $q$ is initial iff $R = L$. So all minimal DFAs are isomorphic. A more formal proof looks as follows:

**Theorem 3.7** If $L$ is regular, then $C_L$ is the unique minimal DFA up to isomorphism recognizing $L$.

**Proof:** Let $L$ be a regular language, and let $A = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary DFA recognizing $L$. By Lemma 3.3 the number the number of states of $A$ is greater than or equal to the number of states of $C_L$, and so $C_L$ is a minimal automaton for $L$. To prove uniqueness of the minimal automaton up to isomorphism, assume $A$ is minimal. By Lemma 3.3(2), $\mathcal{L}_A$ is a mapping that assigns to each state of $A$ a residual of $L$, and so $\mathcal{L}_A : Q \rightarrow Q_L$. We prove that $\mathcal{L}_A$ is an isomorphism. $\mathcal{L}_A$ is bijective because it is surjective (Lemma 3.3(2)), and $|Q| = |Q_L|$ ($A$ is minimal by assumption). Moreover, if $\delta(q, a) = q'$, then $L_A(q') = (L_A(q))^a$, and so $\delta_L(L_A(q), a) = L_A(q')$. Also, $\mathcal{L}_A$ maps the initial state of $A$ to the initial state of $C_L$: $L_A(q_0) = L = q_{0L}$. Finally, $\mathcal{L}_A$ maps final to final and non-final to non-final states: $q \in F$ iff $\varepsilon \in L_A(q)$ iff $L_A(q) \in F_L$. 

The following simple corollary is often useful to establish that a given DFA is minimal:

**Corollary 3.8** A DFA is minimal if and only if $L(q) \neq L(q')$ for every two distinct states $q$ and $q'$. 

3.2. MINIMIZING DFAS

Proof: $(\Rightarrow)$: By Theorem 3.7, the number of states of a minimal DFA is equal to the number of residuals of its language. Since every state of recognizes some residual, each state must recognize a different residual.

$(\Leftarrow)$: If all states of a DFA $A$ recognize different languages, then, since every state recognizes some residual, the number of states of $A$ is less than or equal to the number of residuals. So $A$ has at most as many states as $C_{L(A)}$, and so it is minimal.

3.2 Minimizing DFAs

We present an algorithm that converts a given DFA into (a DFA isomorphic to) the unique minimal DFA recognizing the same language. The algorithm first partitions the states of the DFA into blocks, where a block contains all states recognizing the same residual. We call this partition the language partition. Then, the algorithm “merges” the states of each block into one single state, an operation usually called quotienting with respect to the partition. Intuitively, this yields a DFA in which every state recognizes a different residual. These two steps are described in Section 3.2.1 and Section 3.2.2.

For the rest of the section we fix a DFA $A = (Q, \Sigma, \delta, q_0, F)$ recognizing a regular language $L$.

3.2.1 Computing the language partition

We need some basic notions on partitions. A partition of $Q$ is a finite set $P = \{B_1, \ldots, B_n\}$ of nonempty subsets of $Q$, called blocks, such that $Q = B_1 \cup \ldots \cup B_n$, and $B_i \cap B_j = \emptyset$ for every $1 \leq i \neq j \leq n$. The block containing a state $q$ is denoted by $[q]_P$. A partition $P'$ refines or is a refinement of another partition $P$ if every block of $P'$ is contained in some block of $P$. If $P'$ refines $P$ and $P' \neq P$, then $P$ is coarser than $P'$.

The language partition, denoted by $P_{\ell}$, puts two states in the same block if and only if they recognize the same language (i.e, the same residual). Formally, $[q]_{P_{\ell}} = [q']_{P_{\ell}}$ if and only if $L(q) = L(q')$. To compute $P_{\ell}$ we iteratively refine an initial partition $P_0$ while maintaining the following invariant:

Invariant: States in different blocks recognize different languages.

$P_0$ consists of two blocks containing the final and the non-final states, respectively (or just one of the two if all states are final or all states are nonfinal). That is, $P_0 = \{F, Q \setminus F\}$ if $F$ and $Q \setminus F$ are nonempty, $P_0 = \{F\}$ if $Q \setminus F$ is empty, and $P_0 = \{Q \setminus F\}$ if $F$ is empty. Notice that $P_0$ satisfies the invariant, because every state of $F$ accepts the empty word, but no state of $Q \setminus F$ does.

A partition is refined by splitting a block into two blocks. To find a block to split, we first observe the following:

Fact 3.9 If $L(q_1) = L(q_2)$, then $L(\delta(q_1, a)) = L(\delta(q_2, a))$ for every $a \in \Sigma$. 

Now, by contraposition, if \( L(\delta(q_1, a)) \neq L(\delta(q_2, a)) \), then \( L(q_1) \neq L(q_2) \), or, rephrasing in terms of blocks: if \( \delta(q_1, a) \) and \( \delta(q_2, a) \) belong to different blocks, but \( q_1 \) and \( q_2 \) belong to the same block \( B \), then \( B \) can be split, because \( q_1 \) and \( q_2 \) can be put in different blocks while respecting the invariant.

**Definition 3.10** Let \( B, B' \) be (not necessarily distinct) blocks of a partition \( P \), and let \( a \in \Sigma \). The pair \( (a, B') \) splits \( B \) if there are \( q_1, q_2 \in B \) such that \( \delta(q_1, a) \in B' \) and \( \delta(q_2, a) \notin B' \). The result of the split is the partition \( \text{Ref}_P(B, a, B') = (P \setminus \{B\}) \cup \{B_0, B_1\} \), where

\[
B_0 = \{ q \in B \mid \delta(q, a) \notin B' \} \quad \text{and} \quad B_1 = \{ q \in B \mid \delta(q, a) \in B' \}.
\]

A partition is unstable if it contains blocks \( B, B' \) such that \( (a, B') \) splits \( B \) for some \( a \in \Sigma \), and stable otherwise.

The partition refinement algorithm \( \text{LanPar}(A) \) iteratively refines the initial partition of \( A \) until it becomes stable. The algorithm terminates because every iteration increases the number of blocks by one, and a partition can have at most \(|Q| \) blocks.

\[
\begin{align*}
\text{LanPar}(A) \\
\text{Input:} & \text{ DFA } A = (Q, \Sigma, \delta, q_0, F) \\
\text{Output:} & \text{ The language partition } P_\ell.
\end{align*}
\]

1. if \( F = \emptyset \) or \( Q \setminus F = \emptyset \) then return \( \{Q\} \)
2. else \( P \leftarrow \{F, Q \setminus F\} \)
3. while \( P \) is unstable do
4. pick \( B, B' \in P \) and \( a \in \Sigma \) such that \( (a, B') \) splits \( B \)
5. \( P \leftarrow \text{Ref}_P(B, a, B') \)
6. return \( P \)

Notice that if all states of a DFA are nonfinal then every state recognizes \( \emptyset \), and if all are final then every state recognizes \( \Sigma^* \). In both cases all states recognize the same language, and the language partition is \( \{Q\} \).

**Example 3.11** Figure 3.3 shows a run of \( \text{LanPar}(\cdot) \) on the DFA on the right of Figure 3.1. States that belong to the same block are given the same color. The initial partition, shown at the top, consists of the yellow and the pink states. The yellow block and the letter \( a \) split the pink block into the green block (pink states with an \( a \)-transition to the yellow block) and the rest (pink states with an \( a \)-transition to other blocks), which stay pink. In the final step, the green block and the letter \( b \) split the pink block into the magenta block (pink states with a \( b \) transition into the green block) and the rest, which stay pink.

We prove in two steps that \( \text{LanPar}(A) \) computes the language partition. First, we show that it computes the coarsest stable refinement of \( P_0 \), denoted by \( \text{CSR} \); in other words we show that after termination the partition \( P \) is coarser than every other stable refinement of \( P_0 \). Then we prove that \( \text{CSR} \) is equal to \( P_\ell \).
Figure 3.3: Computing the language partition for the DFA on the left of Figure 3.1
Lemma 3.12 \textit{LanPar}(A) computes CSR.

\textbf{Proof:} \textit{LanPar}(A) clearly computes a stable refinement of \( P_0 \). We prove that after termination \( P \) is coarser than any other stable refinement of \( P_0 \), or, equivalently, that every stable refinement of \( P_0 \) refines \( P \). Actually, we prove more: we show that this is in fact an invariant of the program: it holds not only after termination, but at any time.

Let \( P' \) be an arbitrary stable refinement of \( P_0 \). Initially \( P = P_0 \), and so \( P' \) refines \( P \). Now, we show that if \( P' \) refines \( P \), then \( P' \) also refines \( \text{Ref}_P[B, a, B'] \). For this, let \( q_1, q_2 \) be two states belonging to the same block of \( P' \). We show that they belong to the same block of \( \text{Ref}_P[B, a, B'] \). Assume the contrary. Since the only difference between \( P \) and \( \text{Ref}_P[B, a, B'] \) is the splitting of \( B \) into \( B_0 \) and \( B_1 \), exactly one of \( q_1 \) and \( q_2 \), say \( q_1 \), belongs to \( B_0 \), and the other belongs to \( B_1 \). So there exists a transition \((q_2, a, q'_2) \in \delta \) such that \( q'_2 \in B' \). Since \( P' \) is stable and \( q_1, q_2 \) belong to the same block of \( P' \), there is also a transition \((q_1, a, q'_1) \in \delta \) such that \( q'_1 \in B' \). But this contradicts \( q_1 \in B_0 \).

\begin{proof}
\end{proof}

Theorem 3.13 CSR is equal to \( P_\ell \).

\textbf{Proof:} The proof has three parts:

(a) \( P_\ell \) refines \( P_0 \). Obvious.

(b) \( P_\ell \) is stable. By Fact 3.9, if two states \( q_1, q_2 \) belong to the same block of \( P_\ell \), then \( \delta(q_1, a), \delta(q_2, a) \) also belong to the same block, for every \( a \). So no block can be split.

(c) Every stable refinement \( P \) of \( P_0 \) refines \( P_\ell \). Let \( q_1, q_2 \) be states belonging to the same block \( B \) of \( P \). We prove that they belong to the same block of \( P_\ell \), i.e., that \( L(q_1) = L(q_2) \). By symmetry, it suffices to prove that, for every word \( w \), if \( w \in L(q_1) \) then \( w \in L(q_2) \). We proceed by induction on the length of \( w \). If \( w = \varepsilon \) then \( q_1 \in F \), and since \( P \) refines \( P_0 \), we have \( q_2 \in F \), and so \( w \in L(q_2) \). If \( w = aw' \), then there is \((q_1, a, q'_1) \in \delta \) such that \( w' \in L(q'_1) \). Let \( B' \) be the block containing \( q'_1 \). Since \( P \) is stable, \( B' \) does not split \( B \), and so there is \((q_2, a, q'_2) \in \delta \) such that \( q'_2 \in B' \). By induction hypothesis, \( w' \in L(q'_1) \) iff \( w' \in L(q'_2) \). So \( w' \in L(q'_2) \), which implies \( w \in L(q_2) \).

\begin{proof}
\end{proof}

3.2.2 Quotienting

It remains to define the quotient of \( A \) with respect to a partition. It is convenient to define it not only for DFAs, but more generally for NFAs. The states of the quotient are the blocks of the partition, and there is a transition \((B, a, B') \) from block \( B \) to block \( B' \) if \( A \) contains some transition \((q, a, q') \) for states \( q \) and \( Q' \) belonging to \( B \) and \( B' \), respectively. Formally:

\begin{definition}
The quotient of a NFA \( A \) with respect to a partition \( P \) is the NFA \( A/P = (Q_P, \Sigma, \delta_P, q_{0P}, F_P) \) where
\end{definition}
3.2. MINIMIZING DFAS

- $Q_P$ is the set of blocks of $P$;
- $(B, a, B') \in \delta_P$ if $(q, a, q') \in \delta$ for some $q \in B$, $q' \in B'$;
- $q_0 P$ is the block of $P$ containing $q_0$; and
- $F_P$ is the set of blocks of $P$ that contain at least one state of $F$.

**Example 3.15** Figure 3.4 shows on the right the result of quotienting the DFA on the left with respect to its language partition. The quotient has as many states as colors, and it has a transition between two colors (say, an $a$-transition from pink to magenta) if the DFA on the left has such a transition.

![Figure 3.4: Quotient of a DFA with respect to its language partition](image)

We show that $A/P_\ell$, the quotient of a DFA $A$ with respect to the language partition, is the minimal DFA for $L$. The main part of the argument is contained in the following lemma. Loosely speaking, it says that any refinement of the language partition, i.e., any partition in which states of the same block recognize the same language, “is good” for quotienting, because the quotient recognizes the same language as the original automaton. Moreover, if the partition not only refines but is equal to the language partition, then the quotient is a DFA.

**Lemma 3.16** Let $A$ be a NFA, and let $P$ be a partition of the state of $A$. If $P$ refines $P_\ell$, then $L_A(q) = L_{A/P}(B)$ for every state $q$ of $A$, where $B$ is the block of $P$ containing $q$; in particular $L(A/P) = L(A)$. Moreover, if $A$ is a DFA and $P = P_\ell$, then $A/P$ is a DFA.

**Proof:** Let $P$ be any refinement of $P_\ell$. We prove that for every $w \in \Sigma^*$ we have $w \in L_A(q)$ iff $w \in L_{A/P}(B)$. The proof is by induction on $|w|$.
- $|w| = 0$. Then $w = \varepsilon$ and we have
\( e \in L_A(q) \)
iff \( q \in F \)
iff \( B \subseteq F \) \( (P \text{ refines } P_\ell, \text{ and so also } P_0) \)
iff \( B \in F_P \)
iff \( e \in L_{A/P}(B) \)

\(|w| > 0\). Then \( w = aw' \) for some \( a \in \Sigma \). So \( w \in L_A(q) \) iff there is a transition \((q, a, q') \in \delta\) such that \( w' \in L_A(q') \). Let \( B' \) be the block containing \( q' \). By the definition of \( A/P \) we have \((B, a, B') \in \delta_P\), and so:

\[
aw' \in L_A(q) \quad \text{(definition of } q')
\]
\[
w' \in L_{A/P}(B') \quad \text{(induction hypothesis)}
\]
\[
aw' \in L_{A/P}(B) \quad ( (B, a, B') \in \delta_P)
\]

For the second part, show that \((B, a, B_1), (B, a, B_2) \in \delta_P\) implies \(B_1 = B_2\). By definition there exist \((q, a, q_1), (q', a, q_2) \in \delta\) for some \(q, q' \in B, q_1 \in B_1, \text{ and } q_2 \in B_2\). Since \(q, q' \) belong to the same block of the language partition, we have \(L_A(q) = L_A(q')\). Since \(A\) is a DFA, we get \(L_A(q_1) = L_A(q_2)\). Since \(P = P_\ell\), the states \(q_1\) and \(q_2\) belong to the same block, and so \(B_1 = B_2\).

**Proposition 3.17** The quotient \(A/P_\ell\) is the minimal DFA for \(L\).

**Proof:** By Lemma 3.16, \(A/P_\ell\) is a DFA, and its states recognize residuals of \(L\). Moreover, two states of \(A/P_\ell\) recognize different residuals by definition of the language partition. So \(A/P_\ell\) has as many states as residuals, and we are done.

### 3.3 Reducing NFAs

There is no canonical minimal NFA for a given regular language. The simplest witness of this fact is the language \(aa^*\), which is recognized by the two non-isomorphic, minimal NFAs of Figure 3.5.

![Figure 3.5: Two minimal NFAs for \(aa^*\).](image)

Moreover, computing any of the minimal NFAs equivalent to a given NFA is computationally hard. Recall that the universality problem is PSPACE-complete: given a NFA \(A\) over an alphabet \(\Sigma\), decide whether \(L(A) = \Sigma^*\). Using this result, we can easily prove that deciding the existence of a small NFA for a language is PSPACE-complete.
3.3. REDUCING NFAS

Theorem 3.18 The following problem is PSPACE-complete: given a NFA $A$ and a number $k \geq 1$, decide if there exists an NFA equivalent to $A$ having at most $k$ states.

Proof: To prove membership in PSPACE, observe first that if $A$ has at most $k$ states, then we can answer $A$. So assume that $A$ has more than $k$ states. We use $\text{NPSPACE} = \text{PSPACE} = \text{co-PSPACE}$. Since $\text{PSPACE} = \text{co-PSPACE}$, it suffices to give a procedure to decide if no NFA with at most $k$ states is equivalent to $A$. For this we construct all NFAs with at most $k$ states (over the same alphabet as $A$), reusing the same space for each of them, and check that none of them is equivalent to $A$. Now, since $\text{NPSPACE} = \text{PSPACE}$, it suffices to exhibit a nondeterministic algorithm that, given a NFA $B$ with at most $k$ states, checks that $B$ is not equivalent to $A$ (and runs in polynomial space). The algorithm nondeterministically guesses a word, one letter at a time, while maintaining the sets of states in both $A$ and $B$ reached from the initial states by the word guessed so far. The algorithm stops when it observes that the current word is accepted by exactly one of $A$ and $B$.

PSPACE-hardness is easily proved by reduction from the universality problem. If an NFA is universal, then it is equivalent to an NFA with one state, and so, to decide if a given NFA $A$ is universal we can proceed as follows: Check first if $A$ accepts all words of length 1. If not, then $A$ is not universal. Otherwise, check if some NFA with one state is equivalent to $A$. If not, then $A$ is not universal. Otherwise, if such a NFA, say $B$, exists, then, since $A$ accepts all words of length 1, $B$ is the NFA with one final state and a loop for each alphabet letter. So $A$ is universal.

However, we can reuse part of the theory for the DFA case to obtain an efficient algorithm to possibly reduce the size of a given NFA.

3.3.1 The reduction algorithm

We fix for the rest of the section an NFA $A = (Q, \Sigma, \delta, q_0, F)$ recognizing a language $L$. Recall that Definition 3.14 and the first part of Lemma 3.16 were defined for NFA. So $L(A) = L(A/P)$ holds for every refinement $P$ of $P_\ell$, and so any refinement of $P_\ell$ can be used to reduce $A$. The largest reduction is obtained for $P = P_\ell$, but $P_\ell$ is hard to compute for NFA. On the other extreme, the partition that puts each state in a separate block is always a refinement of $P_\ell$, but it does not provide any reduction.

To find a reasonable trade-off we examine again Lemma 3.12, which proves that $\text{LanPar}(A)$ computes $\text{CSR}$ for deterministic automata. Its proof only uses the following property of stable partitions: if $q_1, q_2$ belong to the same block of a stable partition and there is a transition $(q_2, a, q'_2) \in \delta$ such that $q'_2 \in B'$ for some block $B'$, then there is also a transition $(q_1, a, q'_1) \in \delta$ such that $q'_1 \in B'$. We extend the definition of stability to NFAs so that stable partitions still satisfy this property: we just replace condition

$\delta(q_1, a) \in B'$ and $\delta(q_2, a) \notin B'$

of Definition 3.10 by

$\delta(q_1, a) \cap B' \neq \emptyset$ and $\delta(q_2, a) \cap B' = \emptyset$. 
Definition 3.19 (Refinement and stability for NFAs) Let $B, B'$ be (not necessarily distinct) blocks of a partition $P$, and let $a \in \Sigma$. The pair $(a, B')$ splits $B$ if there are $q_1, q_2 \in B$ such that $\delta(q_1, a) \cap B' \neq \emptyset$ and $\delta(q_2, a) \cap B' = \emptyset$. The result of the split is the partition $\text{Ref}^\text{NFA}_P [B, a, B'] = (P \setminus \{B\}) \cup \{B_0, B_1\}$, where

$$B_0 = \{q \in B \mid \delta(q, a) \cap B' = \emptyset\} \text{ and } B_1 = \{q \in B \mid \delta(q, a) \cap B' \neq \emptyset\}.$$  

A partition is unstable if it contains blocks $B, B'$ such that $B'$ splits $B$, and stable otherwise.

Using this definition we generalize LanPar$(A)$ to NFAs in the obvious way: allow NFAs as inputs, and replace Ref$_p$ by Ref$_p^\text{NFA}$ as new notion of refinement. Lemma 3.12 still holds: the algorithm still computes CSR, but with respect to the new notion of refinement. Notice that in the special case of DFAs it reduces to LanPar$(A)$, because Ref$_p$ and Ref$_p^\text{NFA}$ coincide for DFAs.

\begin{verbatim}
CSR(A)
Input: NFA A = (Q, \Sigma, \delta, q_0, F)
Output: The partition CSR of A.
1 if F = \emptyset or Q \setminus F = \emptyset then P ← \{Q\}
2 else P ← \{F, Q \setminus F\}
3 while P is unstable do
4     pick B, B' ∈ P and a ∈ \Sigma such that (a, B') splits B
5     P ← Ref^\text{NFA}_P \{B, a, B'\}
6 return P
\end{verbatim}

Notice that line 1 is different from line 1 in algorithm LanPar. If all states of a NFA are nonfinal then every state recognizes \emptyset, but if all are final we can no longer conclude that every state recognizes $\Sigma^*$, as was the case for DFAs. In fact, all states might recognize different languages.

In the case of DFAs we had Theorem 3.13, stating that CSR is equal to $P_\ell$. The theorem does not hold anymore for NFAs, as we will see later. However, part (c) of the proof, which showed that CSR refines $P_\ell$, still holds, with exactly the same proof. So we get:

Theorem 3.20 Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFA. The partition CSR refines $P_\ell$.

Now, Lemma 3.16 and Theorem 3.13 lead to the final result:

Corollary 3.21 Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFA. Then $L(A/CSR) = L(A)$.

Example 3.22 Consider the NFA at the top of Figure 3.6. CSR is the partition indicated by the colors. A possible run of CSR(A) is graphically represented at the bottom of the figure as a tree. Initially we have the partition with two blocks shown at the top of the figure: the block $\{1, \ldots, 14\}$ of non-final states and the block $\{15\}$ of final states. The first refinement uses $(a, \{15\})$ to split the block of non-final states, yielding the blocks $\{1, \ldots, 8, 11, 12, 13\}$ (no $a$-transition to $\{15\}$) and $\{9, 10, 14\}$ (an $a$-transition to $\{15\}$). The leaves of the tree are the blocks of CSR.
Figure 3.6: An NFA and a run of CSR() on it.
Figure 3.7: The quotient of the NFA of Figure 3.6.
In this example we have $CSR \neq P_\ell$. For instance, states 3 and 5 recognize the same language, namely $(a + b)^* aa(a + b)^*$, but they belong to different blocks of $CSR$.

The quotient automaton is shown in Figure 3.7.

We finish the section with a remark.

**Remark 3.23** If $A$ is an NFA, then $A/P_\ell$ may not be a minimal NFA for $L$. The NFA of Figure 3.8 is an example: all states accept different languages, and so $A/P_\ell = A$, but the NFA is not minimal, since, for instance, the state at the bottom can be removed without changing the language.

![Figure 3.8: An NFA $A$ such that $A/P_\ell$ is not minimal.](image)

It is not difficult to show that if two states $q_1, q_2$ belong to the same block of $CSR$, then they not only recognize the same language, but also satisfy the following far stronger property: for every $a \in \Sigma$ and for every $q'_1 \in \delta(q_1, a)$, there exists $q'_2 \in \delta(q_2, a)$ such that $L(q'_1) = L(q'_2)$. This can be used to show that two states belong to different blocks of $CSR$. For instance, consider states 2 and 3 of the NFA on the left of Figure 3.9. They recognize the same language, but state 2 has a $c$-successor, namely state 4, that recognizes $\{d\}$, while state 3 has no such successor. So states 2 and 3 belong to different blocks of $CSR$. A possible run of the CSR algorithm on this NFA is shown on the right of the figure. For this NFA, $CSR$ has as many blocks as states.

### 3.4 A Characterization of the Regular Languages

We present a useful byproduct of the results of Section 3.1.

**Theorem 3.24** A language $L$ is regular iff it has finitely many residuals.

**Proof:** If $L$ is not regular, then no DFA recognizes it. Since, by Proposition 3.6, the canonical automaton $C_L$ recognizes $L$, then $C_L$ necessarily has infinitely many states, and so $L$ has infinitely many residuals.

If $L$ is regular, then some DFA $A$ recognizes it. By Lemma 3.3, the number of states of $A$ is greater than or equal to the number of residuals of $L$, and so $L$ has finitely many residuals.
CHAPTER 3. MINIMIZATION AND REDUCTION

This theorem provides a useful technique for proving that a given language \( L \subseteq \Sigma^* \) is not regular: exhibit an infinite set of words \( W \subseteq \Sigma^* \) with pairwise different residuals, i.e., \( W \) must satisfy \( L^w \neq L^v \) for every two distinct words \( w, v \in W \). Let us apply the technique to some typical examples.

- \( \{a^n b^n \mid n \geq 0\} \) is not regular. Let \( W = \{a^k \mid k \geq 0\} \). For every two distinct words \( a^i, a^j \in W \) (i.e., \( i \neq j \)), we have \( b^j \in L^{a^i} \) but \( b^j \notin L^{a^i} \).

- \( \{w w \mid w \in \Sigma^*\} \) is not regular. Let \( W = \Sigma^* \). For every two distinct words \( w, v \in W \) (i.e., \( w \neq v \)), we have \( w \in L^w \) but \( w \notin L^v \).

- \( \{a^{n^2} \mid n \geq 0\} \). Let \( W = \{a^{n^2} \mid n \geq 0\} \) (\( W = L \) in this case). For every two distinct words \( a^{i^2}, a^{j^2} \in W \) (i.e., \( i \neq j \)), we have that \( a^{2i+1} \) belongs to the \( a^{i^2} \)-residual of \( L \), because \( a^{i^2+2i+1} = a^{(i+1)^2} \), but not to the \( a^{j^2} \)-residual, because \( a^{j^2+2i+1} \) is only a square number for \( i = j \).

Exercises

Exercise 18  Consider the most-significant-bit-first encoding (msbf encoding) of natural numbers over the alphabet \( \Sigma = \{0, 1\} \). Recall that every number has infinitely many encodings, because all the words of \( 0 * w \) encode the same number as \( w \). Construct the minimal DFAs accepting the following languages.

- \( \{w \mid \text{msbf}^{-1}(w) \mod 3 = 0\} \cap \Sigma^4 \).
- \( \{w \mid \text{msbf}^{-1}(w) \text{ is a prime}\} \cap \Sigma^4 \).
Exercise 19  Consider the family of languages \( L_k = \{ww \mid w \in \Sigma^k\} \), where \( k \geq 2 \).
- Construct the minimal DFA for \( L_4 \).
- How many states has the minimal DFA accepting \( L_k \)?

Exercise 20  Consider the following DFA over the alphabet \( \Sigma = \{00, 01, 10, 11\} \):

We interpret a word \( w \Sigma^* \) as a pair of natural numbers \( (X(w), Y(w)) \in \mathbb{N}_0 \times \mathbb{N}_0 \) by reading the bits at odd positions as the msbf encoding encoding of \( X(w) \), and the bits as even positions as the mbsf encoding of \( Y(w) \). For example, if \( w = 00001011 \) then \( X(w) = 3 \) abd \( Y(w) = 1 \), because

\[
00001011 \rightarrow 00001011 \rightarrow (0011, 0001) \rightarrow (3, 1)
\]

1. Show that a word \( w \) is accepted by the DFA iff \( X(w) = 3 \cdot Y(w) \).
2. Construct the minimal DFA representing the language \( \{w \in \{0, 1\}^* \mid \text{msbf}^{-1}(w) \text{ is divisible by 3}\} \).

Exercise 21  Consider the language partition algorithm \( \text{LanPar} \). Since every execution of its while loop increases the number of blocks by one, the loop can be executed at most \( |Q| - 1 \) times. Show that this bound is tight, i.e. give a family of DFAs for which the loops is executed \( |Q| - 1 \) times. 

*Hint*: You can take a one-letter alphabet.

Exercise 22  Describe in words the language of the following NFA, and compute CSR, i.e., the coarsest stable refinement of \( P_0 \).
1. Describe $L(A)$.

2. Determine the CSR of $A$ using the algorithm presented in the lecture.

**Exercise 23** Determine the residuals of the following languages over $\Sigma = \{a, b\}$:

- $(ab + ba)^*$
- $(aa)^*$
- $\{w \in \{a, b\}^* | w$ contains the same number of occurrences of $ab$ and $ba$\}$
- $\{a^n b^n c^n | n \geq 0\}$

**Exercise 24** Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, we denote $wL = \{u \in \Sigma^* | uw \in L\}$. A language $L' \subseteq \Sigma^*$ is an inverse residual of $L$ if $L' = wL$ for some $w \in \Sigma^*$.

- Determine the inverse residuals of the first two languages in Exercise 23.
- Show that a language is regular iff it has finitely many inverse residuals.
- Does a language always have as many residuals as inverse residuals?

**Exercise 25** Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, the $w$-context of $L$ is the set of pairs $\{(u, v) \in \Sigma^* | uwv \in L\}$. A language is a context of $L$ if it is a $w$-context for at least one $w \in \Sigma^*$.

- Determine the contexts of $L_2$ in Exercise 23.
- Can a language have more residuals than contexts? And more contexts than residuals?

**Exercise 26** A DFA with negative transitions (DFA-n) is a DFA whose transitions are partitioned into positive and negative transitions. A run of a DFA-n is accepting if:

- it ends in a final state and the number of occurrences of negative transitions is even, or
- it ends in a non-final state and the number of occurrences of negative transitions is odd.

The intuition is that taking a negative transition “inverts the polarity” of the acceptance condition: after taking the transition we accept iff we would not accept were the transition positive.

- Prove that the languages recognized by DFAs with negative transitions are regular.
- Give a DFA-n for a regular language having fewer states than the minimal DFA for the language.
- Show that the minimal DFA-n for a language is not unique (even for languages whose minimal DFA-n’s have fewer states than their minimal DFAs).
Exercise 27  A residual of a regular language $L$ is composite if it is the union of other residuals of $L$. A residual of $L$ is prime if it is not composite. Show that every regular language $L$ is recognized by an NFA whose number of states is equal to the number of prime residuals of $L$.

Exercise 28  Show that the following languages over \{0, 1, 2\} are not regular:

- $\{0^n1^n2^m \mid n, m \geq 0\}$
- $\{0^{2n}12^{3n} \mid n \geq 0\}$
- $\{w \mid |w|_0 = |w|_2\}$, where $|w|_\sigma$ denotes the number of occurrences of the letter $\sigma$ in word $w$.

Exercise 29  Prove or disprove:

- A subset of a regular language is regular.
- A superset of a regular language is regular.
- If $L_1$ and $L_1L_2$ are regular, then $L_2$ is regular.
- If $L_2$ and $L_1L_2$ are regular, then $L_1$ is regular.

Exercise 30  (T. Henzinger) Which of these languages over the alphabet \{0, 1\} are regular?

- The set of words containing the same number of 0’s and 1’s.
- The set of words containing the same number of occurrences of the strings 01 and 10. (E.g., 01010001 contains three occurrences of 01 and two occurrences of 10.)
- Same for the pairs of strings 00 and 11, the pair 001 and 110, and the pair 001 and 100.

Exercise 31  A word $w = a_1 \ldots a_n$ is a subword of $v = b_1 \ldots b_m$, denoted by $w \preceq v$, if there are indices $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_{i_j} = b_j$ for every $j \in \{1, \ldots, n\}$. Higman’s lemma states that every infinite set of words over a finite alphabet contains two words $w_1, w_2$ such that $w_1 \preceq w_2$.

A language $L \subseteq \Sigma^*$ is upward-closed if for every two words $w, v \in \Sigma^*$, if $w \in L$ and $w \preceq v$, then $v \in L$. The upward-closure of a language $L$ is the upward-closed language obtained by adding to $L$ all words $v$ such that $w \preceq v$ for some $v \in L$.

- Prove using Higman’s lemma that every upward-closed language is regular.
- Give regular expressions for the upward-closures of the languages in Exercise 28.
- Give an algorithm that transforms a regular expression for a language into a regular expression for its upward-closure.
Exercise 32 A word of a language $L$ is *minimal* if it is not a proper subword of another word of $L$. We denote the set of minimal words of $L$ by $\text{min}(L)$. Give a family of regular languages $\{L_n\}_{n=1}^{\infty}$ such that every $L_n$ is recognized by a NFA with $O(n)$ states, but the smallest NFA recognizing $\text{min}(L)$ has $O(2^n)$ states.

Exercise 33 Consider the alphabet $\Sigma = \{\text{up, down, left, right}\}$. A word over $\Sigma$ corresponds to a line in a grid consisting of concatenated segments drawn in the direction specified by the letters. In the same way, a language corresponds to a set of lines.

The set of all *staircases* can be specified as the set of lines given by the regular language $(\text{upright})^*$. It is a regular language.

- Specify the set of all *skylines* as a regular language.
- Show that the set of all *rectangles* is not regular.
Chapter 4

Operations on Sets: Implementations

Recall the list of operations on sets that should be supported by our data structures, where \( U \) is the universe of objects, \( X, Y \) are subsets of \( U \), \( x \) is an element of \( U \):

- **Member**\((x, X)\) : returns **true** if \( x \in X \), **false** otherwise.
- **Complement**\((X)\) : returns \( U \setminus X \).
- **Intersection**\((X, Y)\) : returns \( X \cap Y \).
- **Union**\((X, Y)\) : returns \( X \cup Y \).
- **Empty**\((X)\) : returns **true** if \( X = \emptyset \), **false** otherwise.
- **Universal**\((X)\) : returns **true** if \( X = U \), **false** otherwise.
- **Included**\((X, Y)\) : returns **true** if \( X \subseteq Y \), **false** otherwise.
- **Equal**\((X, Y)\) : returns **true** if \( X = Y \), **false** otherwise.

We fix an alphabet \( \Sigma \), and assume that there exists a bijection between \( U \) and \( \Sigma^* \), i.e., we assume that each object of the universe is encoded by a word, and each word is the encoding of some object. Under this assumption, the operations on sets and elements become operations on languages and words. For instance, the first two operations become

- **Member**\((w, L)\) : returns **true** if \( w \in L \), **false** otherwise.
- **Complement**\((L)\) : returns \( \overline{L} \).

The assumption that each word encodes some object may seem too strong. Indeed, the language \( E \) of encodings is usually only a subset of \( \Sigma^* \). However, once we have implemented the operations above under this strong assumption, we can easily modify them so that they work under a much weaker assumption, that almost always holds: the assumption that the language \( E \) of encodings is regular. Assume, for instance, that \( E \) is a regular subset of \( \Sigma^* \) and \( L \) is the language of encodings of a set \( X \). Then, we implement **Complement**\((X)\) so that it returns, not \( \overline{L} \), but \( \text{Intersection}(\overline{L}, E) \).

For each operation we present an implementation that, given automata representations of the
operands, returns an automaton representing the result (or a boolean, when that is the return type). Sections 4.1 and 4.2 consider the cases in which the representation is a DFA and a NFA, respectively.

4.1 Implementation on DFAs

In order to evaluate the complexity of the operations we must first make explicit our assumptions on the complexity of basic operations on a DFA $A = (Q, \Sigma, \delta, q_0, F)$. We assume that dictionary operations (lookup, add, remove) on $Q$ and $\delta$ can be performed in constant time using hashing. We assume further that, given a state $q$, we can decide in constant time if $q = q_0$, and if $q \in F$, and that given a state $q$ and a letter $a \in \Sigma$, we can find in constant time the unique state $\delta(q, a)$.

4.1.1 Membership.

To check membership for a word $w$ we just execute the run of the DFA on $w$. It is convenient for future use to have an algorithm $\text{Member}[A](w, q)$ that takes as parameter a DFA $A$, a state $q$ of $A$, and a word $w$, and checks if $w$ is accepted with $q$ as initial state. $\text{Member}(w, L)$ can then be implemented by $\text{Mem}[A](w, q_0)$, where $A$ is the automaton representing $L$.

Writing $\text{head}(aw) = a$ and $\text{tail}(aw) = w$ for $a \in \Sigma$ and $w \in \Sigma^*$, the algorithm looks as follows:

$$\text{MemDFA}[A](w, q)$$

Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$, state $q \in Q$, word $w \in \Sigma^*$,
Output: true if $w \in \mathcal{L}(q)$, false otherwise

1. if $w = \epsilon$ then return $q \in F$
2. else return $\text{Member}[A](\delta(q, \text{head}(w)), \text{tail}(w))$

The complexity of the algorithm is $O(|w|)$.

4.1.2 Complement.

Implementing the complement operations on DFAs is easy. Recall that a DFA has exactly one run for each word, and the run is accepting iff it reaches a final state. Therefore, if we swap final and non-final states, the run on a word becomes accepting iff it was non-accepting, and so the new DFA accepts the word iff the new one did not accept it. So we get the following linear-time algorithm:

$$\text{CompDFA}(A)$$

Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$,
Output: DFA $B = (Q', \Sigma, \delta', q'_0, F')$ with $L(B) = \overline{L(A)}$

1. $Q' \leftarrow Q$; $\delta' \leftarrow \delta$; $q'_0 \leftarrow q_0$; $F' = \emptyset$
2. for all $q \in Q$ do
3. if $q \notin F$ then add $q$ to $F'$
Observe that complementation of DFAs preserves minimality. By construction, each state of $\text{Comp}(A)$ recognizes the complement of the language recognized by the same state in $A$. Therefore, if the states of $A$ recognize pairwise different languages, so do the states of $\text{Comp}(A)$. Apply now Corollary 3.8, stating that a DFA is minimal iff their states recognize different languages.

### 4.1.3 Binary Boolean Operations

Instead of specific implementations for union and intersection, we give a generic implementation for all binary boolean operations. Given two DFAs $A_1$ and $A_2$ and a binary boolean operation like union, intersection, or difference, the implementation returns a DFA recognizing the result of applying the operation to $L(A_1)$ and $L(A_2)$. The DFAs for different boolean operations always have the same states and transitions, they differ only in the set of final states. We call this DFA with a yet unspecified set of final states the *pairing* of $A_1$ and $A_2$, denoted by $[A_1, A_2]$. Formally:

**Definition 4.1** Let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be DFAs. The pairing $[A_1, A_2]$ of $A_1$ and $A_2$ is the tuple $(Q, \Sigma, \delta, q_0)$ where:

- $Q = \{ [q_1, q_2] \mid q_1 \in Q_1, q_2 \in Q_2 \}$;
- $\delta = \{ ([q_1, q_2], a, [q_1', q_2']) \mid (q_1, a, q_1') \in \delta_1, (q_2, a, q_2') \in \delta_2 \}$;
- $q_0 = [q_{01}, q_{02}]$.

The run of $[A_1, A_2]$ on a word of $\Sigma^*$ is defined as for DFAs.

It follows immediately from this definition that the run of $[A_1, A_2]$ over a word $w = a_1a_2 \ldots a_n$ is also a “pairing” of the runs of $A_1$ and $A_2$ over $w$. Formally,

$$
\begin{align*}
q_{01} &\xrightarrow{a_1} q_{11} & q_{11} &\xrightarrow{a_2} q_{21} & \ldots & q_{(n-1)1} &\xrightarrow{a_n} q_{n1} \\
q_{02} &\xrightarrow{a_1} q_{12} & q_{12} &\xrightarrow{a_2} q_{22} & \ldots & q_{(n-1)2} &\xrightarrow{a_n} q_{n2}
\end{align*}
$$

are the runs of $A_1$ and $A_2$ on $w$ if and only if

$$
\begin{bmatrix}
q_{01} \\
q_{02}
\end{bmatrix} \xrightarrow{a_1} \begin{bmatrix}
q_{11} \\
q_{12}
\end{bmatrix} \xrightarrow{a_2} \begin{bmatrix}
q_{21} \\
q_{22}
\end{bmatrix} \ldots \begin{bmatrix}
q_{(n-1)1} \\
q_{(n-1)2}
\end{bmatrix} \xrightarrow{a_n} \begin{bmatrix}
q_{n1} \\
q_{n2}
\end{bmatrix}
$$

is the run of $[A_1, A_2]$ on $w$.

DFAs for different boolean operations are obtained by adding an adequate set of final states to $[A_1, A_2]$. Let $L_1, L_2$ be the languages. For intersection, $[A_1, A_2]$ must accept $w$ if and only if $A_1$ accepts $w$ and $A_2$ accepts $w$. This is achieved by declaring a state $[q_1, q_2]$ final if and only if $q_1 \in F_1$ and $q_2 \in F_2$. For union, we just replace and by or. For difference, $[A_1, A_2]$ must accept $w$ if and only if $A_1$ accepts $w$ and $A_2$ does not accept $w$, and so we declare $[q_1, q_2]$ final if and only if $q_1 \in F_1$ and not $q_2 \in F_2$. 

Example 4.2 Figure 4.2 shows at the top two DFAs over the alphabet $\Sigma = \{a\}$. They recognize the words whose length is a multiple of 2 and a multiple of three, respectively. We denote these languages by $\text{Mult}_2$ and $\text{Mult}_3$. The Figure then shows the pairing of the two DFAs (for clarity the states carry labels $x, y$ instead of $[x, y]$), and three DFAs recognizing $\text{Mult}_2 \cap \text{Mult}_3$, $\text{Mult}_2 \cup \text{Mult}_3$, and $\text{Mult}_2 \setminus \text{Mult}_3$, respectively.

![Diagram of DFAs](image)

Figure 4.1: Two DFAs, their pairing, and DFAs for the intersection, union, and difference of their languages.

Example 4.3 The tour of conversions of Chapter 2 started with a DFA for the language of all words over $\{a, b\}$ containing an even number of $a$’s and an even number of $b$’s. This language is the intersection of the language of all words containing an even number of $a$’s, and the language of all words containing an even number of $b$’s. Figure 4.2 shows DFAs for these two languages, and the DFA for their intersection.

![Diagram of DFAs](image)

Figure 4.2: Two DFAs, their pairing, and DFAs for the intersection, union, and difference of their languages.

We can now formulate a generic algorithm that, given two DFAs recognizing languages $L_1, L_2$ and a binary boolean operation, returns a DFA recognizing the result of “applying” the boolean operation to $L_1, L_2$. First we formally define what this means. Given an alphabet $\Sigma$ and a binary
4.1. IMPLEMENTATION ON DFAS

![DFAs and DFA for their intersection](image)

Figure 4.2: Two DFAs and a DFA for their intersection.

boolean operator $\odot$: \{true, false\} $\times$ \{true, false\} $\rightarrow$ \{true, false\}, we lift $\odot$ to a function $\widehat{\odot}$: $2^\Sigma \times 2^\Sigma \rightarrow 2^\Sigma$ on languages as follows

$L_1 \widehat{\bigodot} L_2 = \{w \in \Sigma^* \mid (w \in L_1) \odot (w \in L_2)\}$

That is, in order to decide if $w$ belongs to $L_1 \widehat{\bigodot} L_2$, we first evaluate $(w \in L_1)$ and $(w \in L_2)$ to true or false, and then apply $\widehat{\odot}$ to the results. For instance we have $L_1 \cap L_2 = L_1 \land L_2$. The generic algorithm, parameterized by $\odot$, looks as follows:

BinOp[$\odot$](A$_1$, A$_2$)

**Input:** DFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$

**Output:** DFA $A = (Q, \delta, q_0, F)$ with $L(A) = L(A_1) \odot L(A_2)$

1. $Q, \delta, F \leftarrow \emptyset$
2. $q_0 \leftarrow [q_{01}, q_{02}]$
3. $W \leftarrow \{q_0\}$
4. while $W \neq \emptyset$ do
5.   pick $[q_1, q_2]$ from $W$
6.   add $[q_1, q_2]$ to $Q$
7.   if $q_1 \in F_1 \odot q_2 \in F_2$ then add $[q_1, q_2]$ to $F$
8.   for all $a \in \Sigma$ do
9.     $q'_1 \leftarrow \delta_1(q_1, a)$; $q'_2 \leftarrow \delta_2(q_2, a)$
10.   if $[q'_1, q'_2] \notin Q$ then add $[q'_1, q'_2]$ to $W$
11.   add $([q_1, q_2], a, [q'_1, q'_2])$ to $\delta$

Popular choices of boolean language operations are summarized in the left column below, while the right column shows the corresponding boolean operation needed to instantiate BinOp[$\odot$].
## 4.1.4 Emptiness.

A DFA accepts the empty language if and only if it has no final states (recall our normal form, where all states must be reachable!).

\[ \text{Empty}(A) \]

**Input:** DFA \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** true if \( L(A) = \emptyset \), false otherwise

1. return \( F = \emptyset \)

The runtime depends on the implementation. If we keep a boolean indicating whether the DFA has some final state, then the complexity of \( \text{Empty}() \) is \( O(1) \). If checking \( F = \emptyset \) requires a linear scan over \( Q \), then the complexity is \( O(|Q|) \).

## 4.1.5 Universality.

A DFA accepts \( \Sigma^* \) iff all its states are final, again an algorithm with complexity \( O(1) \) given normal form, and \( O(|Q|) \) otherwise.

\[ \text{UnivDFA}(A) \]

**Input:** DFA \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** true if \( L(A) = \Sigma^* \), false otherwise

1. return \( F = Q \)
4.1.6 Inclusion.

Given two regular languages \( L_1, L_2 \), the following lemma characterizes when \( L_1 \subseteq L_2 \) holds.

**Lemma 4.4** Let \( A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \) and \( A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2) \) be DFAs. \( L(A_1) \subseteq L(A_2) \) if and only if every state \([q_1, q_2]\) of the pairing \([A_1, A_2]\) satisfying \( q_1 \in F_1 \) also satisfies \( q_2 \in F_2 \).

**Proof:** Let \( L_1 = L(A_1) \) and \( L_2 = L(A_1) \). We have \( L_1 \not\subseteq L_2 \) iff \( L_1 \not\cap L_2 \not= \emptyset \) iff at least one state \([q_1, q_2]\) of the DFA for \( L_1 \not\cap L_2 \) is final iff \( q_1 \in F_1 \) and \( q_2 \not\in F_2 \).

The condition of the lemma can be checked by slightly modifying BinOp. The resulting algorithm checks inclusion on the fly:

\[
\text{InclDFA}(A_1, A_2)
\]

**Input:** DFAs \( A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \), \( A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2) \)

**Output:** true if \( L(A_1) \subseteq L(A_2) \), false otherwise

1. \( Q \leftarrow \emptyset; \)
2. \( W \leftarrow \{(q_{01}, q_{02})\} \)
3. while \( W \not= \emptyset \) do
4. pick \([q_1, q_2]\) from \( W \)
5. add \([q_1, q_2]\) to \( Q \)
6. if \((q_1 \in F_1) \text{ and } (q_2 \not\in F_2)\) then return false
7. for all \( a \in \Sigma \) do
8. \( q'_1 \leftarrow \delta_1(q_1, a); q'_2 \leftarrow \delta_2(q_2, a) \)
9. if \([q'_1, q'_2]\) \not\in Q \text{ then add } \[q'_1, q'_2\] \text{ to } W \)
10. return true

4.1.7 Equality.

For equality, just observe that \( L(A_1) = L(A_2) \) holds if and only if the symmetric difference of \( L(A_1) \) and \( L(A_2) \) is empty. The algorithm is obtained by replacing Line 7 of InclDFA\((A_1, A_2)\) by

\[
\text{if } ((q_1 \in F_1) \text{ and } q_2 \not\in F_2) \text{ or } ((q_1 \not\in F_1) \text{ and } q_2 \in F_2) \text{ then return false} .
\]

4.2 Implementation on NFAs

For NFAs we make the same assumptions on the complexity of basic operations as for DFAs.

For DFAs, however, we had the assumption that, given a state \( q \) and a letter \( a \in \Sigma \), we can find in constant time the unique state \( \delta(q, a) \). This assumption no longer makes sense for NFA, since \( \delta(q, a) \) is a set.
4.2.1 Membership.

Membership testing is slightly more involved for NFAs than for DFAs. An NFA may have many runs on the same word, and examining all of them one after the other in order to see if at least one is accepting is a bad idea: the number of runs may be exponential in the length of the word. The algorithm below does better. For each prefix of the word it computes the set of states in which the automaton may be after having read the prefix.

\[ \text{MemNFA} \left[ A \right] (w) \]

**Input:** NFA \( A = (Q, \Sigma, \delta, q_0, F) \), word \( w \in \Sigma^* \),

**Output:** true if \( w \in \mathcal{L}(A) \), false otherwise

1. \( W \leftarrow \{q_0\} \);
2. while \( w \neq \varepsilon \) do
3. \( U \leftarrow \emptyset \)
4. for all \( q \in W \) do
5. \( \text{add } \delta(q, \text{head}(w)) \text{ to } U \)
6. \( W \leftarrow U \)
7. \( w \leftarrow \text{tail}(w) \)
8. return \((W \cap F \neq \emptyset)\)

**Example 4.5** Consider the NFA of Figure 4.3, and the word \( w = aaabba \). The successive values of \( W \), that is, the sets of states \( A \) can reach after reading the prefixes of \( w \), are shown on the right. Since the final set contains final states, the algorithm returns true.

Figure 4.3: An NFA \( A \) and the run of \( \text{Mem}[A](aaabba) \).

<table>
<thead>
<tr>
<th>Prefix read</th>
<th>( W )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>{1}</td>
</tr>
<tr>
<td>( a )</td>
<td>{2}</td>
</tr>
<tr>
<td>( aa )</td>
<td>{2, 3}</td>
</tr>
<tr>
<td>( aaa )</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>( aaab )</td>
<td>{2, 3, 4}</td>
</tr>
<tr>
<td>( aaabb )</td>
<td>{2, 3, 4}</td>
</tr>
<tr>
<td>( aaabba )</td>
<td>{1, 2, 3, 4}</td>
</tr>
</tbody>
</table>

For the complexity, observe first that the while loop is executed \( |w| \) times. The for loop is executed at most \( |Q| \) times. Each execution takes at most time \( O(|Q|) \), because \( \delta(q, \text{head}(w)) \) contains at most \( |Q| \) states. So the overall runtime is \( O(|w| \cdot |Q|^2) \).
4.2 IMPLEMENTATION ON NFAS

4.2.2 Complement.

Recall that an NFA $A$ may have multiple runs on a word $w$, and it accepts $w$ if at least one is accepting. In particular, an NFA can accept $w$ because of an accepting run $\rho_1$, but have another non-accepting run $\rho_2$ on $w$. It follows that the complementation operation for DFAs cannot be extended to NFAs: after exchanging final and non-final states the run $\rho_1$ becomes non-accepting, but $\rho_2$ becomes accepting. So the new NFA still accepts $w$ (at least $\rho_2$ accepts), and so it does not recognize the complement of $L(A)$.

For this reason, complementation for NFAs is carried out by converting to a DFA, and complementing the result.

\[ CompNFA(A) \]
\[ \text{Input: NFA } A, \]
\[ \text{Output: DFA } \overline{A} \text{ with } L(\overline{A}) = \overline{L(A)} \]

1. $\overline{A} \leftarrow CompDFA(NFAtoDFA(A))$

Since making the NFA deterministic may cause an exponential blow-up in the number of states, the number of states of $\overline{A}$ may be $O(2^{|Q|})$.

4.2.3 Union and intersection.

On NFAs it is no longer possible to uniformly implement binary boolean operations. The pairing operation can be defined exactly as in Definition 4.1. The runs of a pairing $[A_1, A_2]$ of NFAs on a given word are defined as for NFAs. The difference with respect to the DFA case is that the pairing may have multiple runs or no run at all on a word. But we still have that

\[ q_{01} \xrightarrow{a_1} q_{11} \xrightarrow{a_2} q_{21} \ldots q_{(n-1)1} \xrightarrow{a_n} q_{n1} \]
\[ q_{02} \xrightarrow{a_1} q_{12} \xrightarrow{a_2} q_{22} \ldots q_{(n-1)2} \xrightarrow{a_n} q_{n2} \]

are runs of $A_1$ and $A_2$ on $w$ if and only if

\[ \begin{bmatrix} q_{01} \\ q_{02} \end{bmatrix} \xrightarrow{a_1} \begin{bmatrix} q_{11} \\ q_{12} \end{bmatrix} \xrightarrow{a_2} \begin{bmatrix} q_{21} \\ q_{22} \end{bmatrix} \ldots \begin{bmatrix} q_{(n-1)1} \\ q_{(n-1)2} \end{bmatrix} \xrightarrow{a_n} \begin{bmatrix} q_{n1} \\ q_{n2} \end{bmatrix} \]

is a run of $[A_1, A_2]$ on $w$.

Let us now discuss separately the cases of intersection, union, and set difference.

**Intersection.** Let $[q_1, q_2]$ be a final state of $[A_1, A_2]$ if $q_1$ is a final state of $A_1$ and $q_2$ is a final state of $q_2$. Then it is still the case that $[A_1, A_2]$ has an accepting run on $w$ if and only if $A_1$ has an accepting run on $w$ and $A_2$ has an accepting run on $w$. So, with this choice of final states, $[A_1, A_2]$ recognizes $L(A_1) \cap L(A_2)$. So we get the following algorithm:
However, there is an algorithm that produces a smaller NFA. Observe first that a NFA-together with \( \varepsilon \)-applying the reduction algorithm for NFAs we only obtain the 10-state automaton of Figure 3.7.

Intersection, which means that the output of

\[ \text{IntersNFA} \]

a recognizes the words containing at least two blocks with two consecutive \( t \)'s each, the second one those containing at least one block. The result of applying \( \text{IntersNFA} \) is the NFA of Figure 3.6 in page 53. Observe that the NFA has 15 states, i.e., all pairs of states are reachable.

Observe that in this example the intersection of the languages recognized by the two NFAs is equal to the language of the first NFA. So there is an NFA with 5 states that recognizes the intersection, which means that the output of \( \text{IntersNFA} \) is far from optimal in this case. Even after applying the reduction algorithm for NFAs we only obtain the 10-state automaton of Figure 3.7.

Example 4.6 Consider the two NFAs of Figure 4.4 over the alphabet \( \{a, b\} \). The first one recognizes the words containing at least two blocks with two consecutive \( a \)'s each, the second one those containing at least one block. The result of applying \( \text{IntersNFA} \) is the NFA of Figure 3.6 in page 53. Observe that the NFA has 15 states, i.e., all pairs of states are reachable.

Observe that in this example the intersection of the languages recognized by the two NFAs is equal to the language of the first NFA. So there is an NFA with 5 states that recognizes the intersection, which means that the output of \( \text{IntersNFA} \) is far from optimal in this case. Even after applying the reduction algorithm for NFAs we only obtain the 10-state automaton of Figure 3.7.

Union. The argumentation for intersection still holds if we replace \( \text{and} \) by \( \text{or} \), and so an algorithm obtained from \( \text{IntersNFA} \) by substituting \( \text{or} \) for \( \text{and} \) correctly computes a NFA for \( L(A_1) \cup L(A_2) \). However, there is an algorithm that produces a smaller NFA. Observe first that a NFA-\( \varepsilon \) for \( L(A_1) \cup L(A_2) \) can be constructed by putting \( A_1 \) and \( A_2 \) “side by side”, and adding a new initial state \( q_0 \), together with \( \varepsilon \)-transitions from \( q_0 \) to \( q_{01} \) and \( q_{02} \). To get an NFA, we then apply the algorithm for removing \( \varepsilon \)-transitions.

\[
\text{IntersNFA}(A_1, A_2)
\]

**Input:** NFA \( A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \), \( A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2) \)

**Output:** NFA \( A_1 \cap A_2 = (Q, \Sigma, \delta, q_0, F) \) with \( L(A_1 \cap A_2) = L(A_1) \cap L(A_2) \)

1. \( Q, \delta, F \leftarrow \emptyset \)
2. \( q_0 \leftarrow [q_{01}, q_{02}] \)
3. \( W \leftarrow \{ [q_{01}, q_{02}] \} \)
4. while \( W \neq \emptyset \) do
5.   pick \( [q_1, q_2] \) from \( W \)
6.   add \( [q_1, q_2] \) to \( Q \)
7. if \( (q_1 \in F_1) \) and \( (q_2 \in F_2) \) then add \( [q_1, q_2] \) to \( F \)
8.   for all \( a \in \Sigma \) do
9.   for all \( q'_1 \in \delta_1(q_1, a) \), \( q'_2 \in \delta_2(q_2, a) \) do
10.  if \( [q'_1, q'_2] \notin Q \) then add \( [q'_1, q'_2] \) to \( W \)
11.  add \( ([q_1, q_2], a, [q'_1, q'_2]) \) to \( \delta \)

Notice that we overload the symbol \( \cap \), and denote the output by \( A_1 \cap A_2 \). The automaton \( A_1 \cap A_2 \) is often called the product of \( A_1 \) and \( A_2 \). It is easy to see that, as operation on NFAs, \( \cap \) is associative and commutative in the following sense:

\[
L((A_1 \cap A_2) \cap A_3) = L(A_1) \cap L(A_2) \cap L(A_3) = L(A_1 \cap (A_2 \cap A_3))
\]

\[
L(A_1 \cap A_2) = L(A_1) \cap L(A_2) = L(A_2 \cap A_1)
\]

For the complexity, observe that in the worst case the algorithm must examine all pairs \( t_1, t_2 \) of transitions of \( \delta_1 \times \delta_2 \), but every pair is examined at most once. So the runtime is \( \mathcal{O}(||\delta_1|||\delta_2|) \).
4.2. IMPLEMENTATION ON NFAS

The algorithm below directly construct the final result, with the assumption that \( q_0 \) is a fresh state, not belonging to \( Q_1 \) or \( Q_2 \). Removing the \( \epsilon \)-transitions may make the old initial states \( q_{01}, q_{02} \) unreachable from the new initial state \( q_0 \), and in this case we remove them. Notice that the states become unreachable if and only if they had no incoming transitions in \( A_1 \) and \( A_2 \), respectively. We denote by \( \delta_i^{-1}(q_{0i}) \) the set of states \( q \) such that \((q, a, q_{0i})\) for some \( a \in \Sigma \).

**UnionNFA**\((A_1, A_2)\)

**Input:** NFA \( A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \), \( A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2) \)

**Output:** NFA \( A_1 \cup A_2 \) with \( L(A_1 \cup A_2) = L(A_1) \cup L(A_2) \)

1. \( Q \leftarrow Q_1 \cup Q_2 \cup \{q_0\} \)
2. \( \delta \leftarrow \delta_1 \cup \delta_2 \)
3. \( F \leftarrow F_1 \cup F_2 \)
4. **for all** \( i = 1, 2 \) **do**
5.   **if** \( q_{0i} \in F_i \) **then** add \( q_0 \) to \( F \)
6.   **for all** \((q_{0i}, a, q) \in \delta_i \) **do**
7.     add \((q_{0i}, a, q)\) to \( \delta \)
8.   **if** \( \delta_i^{-1}(q_{0i}) = \emptyset \) **then**
9.     remove \( q_{0i} \) from \( Q \)
10.   **for all** \( a \in \Sigma, q \in \delta_i(q_{0i}, a) \) **do**
11.     remove \((q_{0i}, a, q)\) from \( \delta_i \)
12. return \((Q, \Sigma, \delta, q_0, F)\)

Figure 4.5 is a graphical representation of how the algorithm works (up to removal of the old initial states). If emptiness of \( \delta_i^{-1}(q_{0i}) \) can be checked in constant time, then the complexity is \( O(m_1 + m_2) \), where \( m_i \) is the number of transitions of \( A_i \) starting at \( q_{0i} \).

**Set difference.** The generalization of the procedure for DFAs fails. Let \([q_1, q_2]\) be a final state of \([A_1, A_2]\) if \( q_1 \) is a final state of \( A_1 \) and \( q_2 \) is not a final state of \( A_2 \). Then \([A_1, A_2]\) has an accepting
CHAPTER 4. OPERATIONS ON SETS: IMPLEMENTATIONS

4.2.4 Emptiness and Universality.

Emptiness for NFAs is decided using the same algorithm as for DFAs: just check if the NFA has at least one final state.

Universality requires a new algorithm. Since an NFA may have multiple runs on a word, an NFA may be universal even if some states are non-final: for every word having a run that leads to a non-final state there may be another run leading to a final state. An example is the NFA of Figure 4.3, which, as we shall show in this section, is universal.

A language $L$ is universal if and only if $\overline{L}$ is empty, and so universality of an NFA $A$ can be checked by applying the emptiness test to $\overline{A}$. Since complementation, however, involves a worst-case exponential blowup in the size of $A$, the algorithm requires exponential time and space.

We show that the universality problem is PSPACE-complete. That is, the superpolynomial blowup cannot be avoided unless $P = PSPACE$, which is unlikely.

**Theorem 4.7**  The universality problem for NFAs is PSPACE-complete

**Proof:** We only sketch the proof. To prove that the problem is in PSPACE, we show that it belongs to NPSPACE and apply Savitch’s theorem. The polynomial-space nondeterministic algorithm for universality looks as follows. Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$, the algorithm guesses a run of $B = NFAtoDFA(A)$ leading from $\{q_0\}$ to a non-final state, i.e., to a set of states of $A$ containing no

![Figure 4.5: Union for NFAs](image-url)
4.2. IMPLEMENTATION ON NFAS

final state (if such a run exists). The algorithm only does not store the whole run, only the current
state, and so it only needs linear space in the size of $A$.

We prove PSPACE-hardness by reduction from the acceptance problem for linearly bounded
automata. A linearly bounded automaton is a deterministic Turing machine that always halts and
only uses the part of the tape containing the input. A configuration of the Turing machine on an
input of length $k$ is coded as a word of length $k$. The run of the machine on an input can be encoded
as a word $c_0\#c_1\ldots\#c_n$, where the $c_i$’s are the encodings of the configurations.

Let $\Sigma$ be the alphabet used to encode the run of the machine. Given an input $x$, $M$ accepts if
there exists a word $w$ of $\Sigma^*$ satisfying the following properties:

(a) $w$ has the form $c_0\#c_1\ldots\#c_n$, where the $c_i$’s are configurations;
(b) $c_0$ is the initial configuration;
(c) $c_n$ is an accepting configuration; and
(d) for every $0 \leq i \leq n - 1$: $c_{i+1}$ is the successor configuration of $c_i$ according to the transition
relation of $M$.

The reduction shows how to construct in polynomial time, given a linearly bounded automaton
$M$ and an input $x$, an NFA $A(M, x)$ accepting all the words of $\Sigma^*$ that do not satisfy at least one of
the conditions (a)-(d) above. We then have

- If $M$ accepts $x$, then there is a word $w(M, x)$ encoding the accepting run of $M$ on $x$, and so
$L(A(M, x)) = \Sigma^* \setminus \{w(M, x)\}$.
- If $M$ rejects $x$, then no word encodes an accepting run of $M$ on $x$, and so $L(A(M, x)) = \Sigma^*$.

So $M$ accepts $x$ if and only if $L(A(M, x)) = \Sigma^*$, and we are done.

A Subsumption Test. We show that it is not necessary to completely construct $\overline{A}$. First, the
universality check for DFA only examines the states of the DFA, not the transitions. So instead
of $\text{NFAtoDFA}(A)$ we can apply a modified version that only stores the states of $\overline{A}$, but not its
transitions. Second, it is not necessary to store all states.

Definition 4.8 Let $A$ be a NFA, and let $B = \text{NFAtoDFA}(A)$. A state $Q'$ of $B$ is minimal if no other
state $Q''$ satisfies $Q'' \subset Q'$.

Proposition 4.9 Let $A$ be a NFA, and let $B = \text{NFAtoDFA}(A)$. $A$ is universal iff every minimal state
of $B$ is final.

Proof: Since $A$ and $B$ recognize the same language, $A$ is universal iff $B$ is universal. So $A$ is
universal iff every state of $B$ is final. But a state of $B$ is final iff it contains some final state of $A$,
and so every state of $B$ is final iff every minimal state of $B$ is final.
Example 4.10 Figure 4.6 shows a NFA on the left, and the equivalent DFA obtained through the application of \textit{NFAtoDFA()} on the right. Since all states of the DFA are final, the NFA is universal. Only the states \{1\}, \{2\}, and \{3, 4\} (shaded in the picture), are minimal.

Figure 4.6: An NFA, and the result of converting it into a DFA, with the minimal states shaded.

Proposition 4.9 shows that it suffices to construct and store the minimal states of \(B\). Algorithm \textit{UnivNFA(A)} below constructs the states of \(B\) as in \textit{NFAtoDFA(A)}, but introduces at line 8 a subsumption test: it checks if some state \(Q'' \subseteq \delta(Q', a)\) has already been constructed. In this case either \(\delta(Q', a)\) has already been constructed (case \(Q'' = \delta(Q', a)\)) or is non-minimal (case \(Q'' \subset \delta(Q', a)\)). In both cases, the state is not added to the worklist.

\textit{UnivNFA(A)}

\textbf{Input:} NFA \(A = (Q, \Sigma, \delta, q_0, F)\)

\textbf{Output:} true if \(L(A) = \Sigma^*\), false otherwise

1 \(Q \leftarrow \emptyset;\)
2 \(W \leftarrow \{ q_0 \} \)
3 \textbf{while} \(W \neq \emptyset\) \textbf{do}
4 \textbf{pick} \(Q'\) \textbf{from} \(W\)
5 \textbf{if} \(Q' \cap F = \emptyset\) \textbf{then return} \text{false}\n6 \textbf{add} \(Q'\) \textbf{to} \(Q\)
7 \textbf{for all} \(a \in \Sigma\) \textbf{do}
8 \textbf{if} \(W \cup Q\) \textbf{contains no} \(Q'' \subseteq \delta(Q', a)\) \textbf{then add} \(\delta(Q', a)\) \textbf{to} \(W\)
9 \textbf{return} \text{true}
The next proposition shows that UnivNFA(A) constructs all minimal states of B. If UnivNFA(A) would first generate all states of A and then would remove all non-minimal states, the proof would be trivial. But the algorithm removes non-minimal states whenever they appear, and we must show that this does not prevent the future generation of other minimal states.

**Proposition 4.11** Let A = (Q, Σ, δ, q₀, F) be a NFA, and let B = NFAtoDFA(A). After termination of UnivNFA(A), the set Ω contains all minimal states of B.

**Proof:** Let Q₁ be the value of Q after termination of UnivNFA(A). We show that no path of B leads from a state of Q₁ to a minimal state of B not in Q₁. Since {q₀} ∈ Q₁ and all states of B are reachable from {q₀}, it follows that every minimal state of B belongs to Q₁.

Assume there is a path π = Q₁ → Q₂ → Q₃ → ... → Qₙ → Qₙ₊₁ of B such that Q₁ ∈ Q₁, Qₙ₊₁ ∈ Q₁, and Qₙ is minimal. Assume further that π is as short as possible. This implies Q₂ ∉ Q₁ (otherwise Q₂ → Qₙ₊₁ is a shorter path satisfying the same properties), and so Q₂ is never added to the worklist. On the other hand, since Q₁ ∈ Q₁, the state Q₁ is eventually added to and picked from the worklist. When Q₁ is processed at line 7 the algorithm considers Q₂ = δ(Q₁, a₁), but does not add it to the worklist in line 8. So at that moment either the worklist or Q contains a state Q₂′ ⊆ Q₂. This state is eventually added to Q (if it is not already there), and so Q₂′ ∈ Q₁. So B has a path π′ = Q₂′ → Q₃ → ... → Qₙ₊₁ for some states Q₃′, ..., Qₙ′. Since Q₂′ ⊆ Q₂, Q₃′ ⊆ Q₁, ..., Qₙ′ ⊆ Qₙ (notice that we may have Q₃′ = Q₂). By the minimality of Qₙ, we get Qₙ′ = Qₙ, and so π′ leads from Q₂′, which belongs to Q₁, to Qₙ, which is minimal and not in to Q₁. This contradicts the assumption that π is as short as possible.

Notice that the complexity of the subsumption test may be considerable, because the new set δ(Q′, a) must be compared with every set in W ∪ Q. Good use of data structures (hash tables or radix trees) is advisable.

### 4.2.5 Inclusion and Equality.

Recall Lemma 4.4: given two DFAs A₁, A₂, the inclusion L(A₁) ⊆ L(A₂) holds if and only if every state [q₁, q₂] of [A₁, A₂] satisfying q₁ ∈ F₁ also satisfies q₂ ∈ F₂. This lemma no longer holds for NFAs. To see why, let A be any NFA having two runs for some word w, one of them leading to a final state q₁, the other to a non-final state q₂. We have L(A) ⊆ L(A), but the pairing [A, A] has a run on w leading to [q₁, q₂].

To obtain an algorithm for checking inclusion, we observe that L₁ ⊆ L₂ holds if and only if L₁ ∩ L₂ = ∅. This condition can be checked using the constructions for intersection and for the emptiness check. However, as in the case of universality, we can apply a subsumption check.

**Definition 4.12** Let A₁, A₂ be NFAs, and let B₂ = NFAtoDFA(A₂). A state [q₁, Q₂] of [A₁, B₂] is minimal if no other state [q′₁, Q′₂] satisfies q′₁ = q₁ and Q′₂ ⊂ Q₂.
Proposition 4.13 Let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1), A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be NFAs, and let $B_2 = \text{NFAtoDFA}(A_2)$. $L(A_1) \subseteq L(A_2)$ iff every minimal state $[q_1, Q_2]$ of $[A_1, B_2]$ satisfying $q_1 \in F_1$ also satisfies $Q_2 \cap F_2 \neq \emptyset$.

Proof: Since $A_2$ and $B_2$ recognize the same language, $L(A_1) \subseteq L(A_2)$ iff $L(A_1) \cap \overline{L(A_2)} = \emptyset$ iff $L(A_1) \cap L(B_2) = \emptyset$ iff $[A_1, B_2]$ has a state $[q_1, Q_2]$ such that $q_1 \in F_1$ and $Q_2 \cap F_2 = \emptyset$. But $[A_1, B_2]$ has some state satisfying this condition iff it has some minimal state satisfying it.

So we get the following algorithm to check inclusion:

\[
\text{InclNFA}(A_1, A_2)
\]

**Input:** NFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$

**Output:** true if $L(A_1) \subseteq L(A_2)$, false otherwise

1. $Q \leftarrow \emptyset$
2. $W \leftarrow \{ [q_{01}, q_{02}] \}$
3. while $W \neq \emptyset$ do
4. pick $[q_1, Q_2]$ from $W$
5. if $(q_1 \in F_1)$ and $(Q_2 \cap F_2 = \emptyset)$ then return false
6. add $[q_1, Q_2]$ to $Q$
7. for all $a \in \Sigma, q'_1 \in \delta_1(q_1, a)$ do
8. $Q'_2 \leftarrow \delta_2(Q_2, a)$
9. if $W \cup Q$ contains no $[q''_1, Q''_2]$ s.t. $q''_1 = q'_1$ and $Q''_2 \subseteq Q'_2$ then
10. add $[q'_1, Q'_2]$ to $W$
11. return true

Notice that in unfavorable cases the overhead of the subsumption test may not be compensated by a reduction in the number of states. Without the test, the number of pairs that can be added to the worklist is at most $|Q_2|^2 |Q_1|$. For each of them we have to execute the for loop $\mathcal{O}(|Q_1|)$ times, each of them taking $\mathcal{O}(|Q_2|^2)$ time. So the algorithm runs in $|Q_1|^2 2^{\mathcal{O}(|Q_1|)}$ time and space.

As was the case for universality, the inclusion problem is PSPACE-complete, and so the exponential cannot be avoided unless $P = \text{PSPACE}$.

Proposition 4.14 The inclusion problem for NFAs is PSPACE-complete

Proof: We first prove membership in PSPACE. Since PSPACE=co-PSPACE=NPSPACE, it suffices to give a polynomial space nondeterministic algorithm that decides non-inclusion. Given NFAs $A_1$ and $A_2$, the algorithm guesses a word $w \in L(A_1) \setminus L(A_2)$ letter by letter, maintaining the sets $Q'_1, Q'_2$ of states that $A_1$ and $A_2$ can reach by the word guessed so far. When the guessing ends, the algorithm checks that $Q'_1$ contains some final state of $A_1$, but $Q'_2$ does not.

Hardness follows from the fact that $A$ is universal iff $\Sigma \subseteq L(A)$, and so the universality problem, which is PSPACE-complete, is a subproblem of the inclusion problem.
4.2. IMPLEMENTATION ON NFAS

There is however an important case with polynomial complexity, namely when $A_2$ is a DFA. The number of pairs that can be added to the worklist is then at most $|Q_1||Q_2|$. The for loop is still executed $O(|Q_1|)$ times, but each of them takes $O(1)$ time. So the algorithm runs in $O(|Q_1|^2|Q_2|)$ time and space.

Equality. Equality of two languages is decided by checking that each of them is included in the other. The equality problem is again PSPACE-complete. The only point worth observing is that, unlike the inclusion case, we do not get a polynomial algorithm when $A_2$ is a DFA.

Exercises

Exercise 34 Consider the following languages over the alphabet $\Sigma = \{a, b\}$:

- $L_1$ is the set of all words where between any two occurrences of $b$’s there is at least one $a$.
- $L_2$ is the set of all words where every non-empty maximal sequence of consecutive $a$’s has odd length.
- $L_3$ is the set of all words where $a$ occurs only at even positions.
- $L_4$ is the set of all words where $a$ occurs only at odd positions.
- $L_5$ is the set of all words of odd length.
- $L_6$ is the set of all words with an even number of $a$’s.

Remark: For this exercise we assume that the first letter of a nonempty word is at position 1, e.g., $a \in L_4$, $a \notin L_3$.

Construct an NFA for the language

$$ (L_1 \setminus L_2) \cup (L_3 \triangle L_4) \cap L_5 \cap L_6 $$

where $L \triangle L'$ denotes the symmetric difference of $L$ and $L'$, while sticking to the following rules:

- Start from NFAs for $L_1, \ldots, L_6$.
- Any further automaton must be constructed from already existing automata via an algorithm introduced in the chapter, e.g. Comp, BinOp, UnionNFA, NFAtoDFA, etc.

Try to find an order on the construction steps such that the intermediate automata and the final result have as few states as possible.

Exercise 35 Prove or disprove: the minimal DFAs recognizing a language and its complement have the same number of states.
Exercise 36  Let $r$ be the regular expression $((0 + 1)(0 + 1))^*$ over $\Sigma = \{0, 1\}$

- Give a regular expression $r'$ such that $L(r') = \overline{L(r)}$.
- Construct a regular expression $r'$ such that $L(r') = \overline{L(r)}$ using the algorithms of this and the preceding chapters.

Exercise 37  Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Show that with the universal accepting condition of Exercise 9 the automaton $A' = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognizes the complement of the language recognized by $A$.

Exercise 38  Recall the alternating automata introduced in Exercise ??.

- Let $A = (Q_1, Q_2, \Sigma, \delta, q_0, F)$ be an alternating automaton, where $Q_1$ and $Q_2$ are the sets of existential and universal states, respectively, and $\delta: (Q_1 \cup Q_2) \times \Sigma \rightarrow \mathcal{P}(Q_1 \cup Q_2)$. Show that the alternating automaton $A = (Q_2, Q_1, \Sigma, \delta, q_0, Q \setminus F)$ recognizes the complement of the language recognized by $A$. I.e., show that alternating automata can be complemented by exchanging existential and universal states, and final and non-final states.

- Give linear time algorithms that take two alternating automata recognizing languages $L_1, L_2$ and deliver a third alternating automaton recognizing $L_1 \cup L_2$ and $L_1 \cap L_2$.
  \textit{Hint:} The algorithms are very similar to \textit{UnionDFA}.

- Show that the emptiness problem for alternating automata is PSPACE-complete.
  \textit{Hint:} Using Exercise 41, prove that the emptiness problem for alternating automata is PSPACE-complete.

Exercise 39  Find a family $\{A_n\}_{n=1}^{\infty}$ of NFAs with $O(n)$ states such that every NFA recognizing the complement of $L(A_n)$ has at least $2^n$ states.

\textit{Hint:} See Exercise 10.

Exercise 40  Consider again the regular expressions $(1 + 10)^*$ and $1^* (101^*)^*$ of Exercise 2.

- Construct NFAs for the expressions and use \textit{InclNFA} to check if their languages are equal.
- Construct DFAs for the expressions and use \textit{InclDFA} to check if their languages are equal.
- Construct minimal DFAs for the expressions and check whether they are isomorphic.
Exercise 41

- Prove that the following problem is PSPACE-complete:

Given: DFAs $A_1, \ldots, A_n$ over the same alphabet $\Sigma$.
Decide: Is $\bigcap_{i=1}^n L(A_i) = \emptyset$?

*Hint:* Reduction from the acceptance problem for deterministic, linearly bounded automata.

- Prove that if $\Sigma$ is a 1-letter alphabet then the problem is “only” NP-complete.

*Hint:* Reduction from 3-SAT.

- Prove that if the DFAs are acyclic (but the alphabet arbitrary) then the problem is again NP-complete.

Exercise 42

Consider the variant of IntersNFA in which line 7

\[
\text{if } (q_1 \in F_1) \text{ and } (q_2 \in F_2) \text{ then add } [q_1, q_2] \text{ to } F
\]

is replaced by

\[
\text{if } (q_1 \in F_1) \text{ or } (q_2 \in F_2) \text{ then add } [q_1, q_2] \text{ to } F
\]

Let $A_1 \otimes A_2$ be the result of applying this variant to two NFAs $A_1, A_2$. We call $A_1 \otimes A_2$ the \textit{or-product} of $A_1$ and $A_2$.

An NFA $A = (Q, \Sigma, \delta, q_0, F)$ is \textit{complete} if $\delta(q, a) \neq \emptyset$ for every $q \in Q$ and every $a \in \Sigma$.

- Prove: If $A_1$ and $A_2$ are complete NFAs, then $L(A_1 \otimes A_2) = L(A_1) \cup L(A_2)$.

- Give NFAs $A_1, A_2$ such that $L(A_1 \otimes A_2) = L(A_1) \cup L(A_2)$ but neither $A_1$ nor $A_2$ are complete.

Exercise 43

Given a word $w = a_1a_2 \ldots a_n$ over an alphabet $\Sigma$, we define the \textit{even part} of $w$ as the word $a_2a_4 \ldots a_{\lfloor n/2 \rfloor}$. Given an NFA for a language $L$, construct an NFA recognizing the even parts of the words of $L$.

Exercise 44

Given regular languages $L_1, L_2$ over an alphabet $\Sigma$, the \textit{left quotient} of $L_1$ by $L_2$ is the language

\[
L_2 \setminus L_1 := \{v \in \Sigma^* \mid \exists u \in L_2 : uv \in L_1\}
\]

(Note that $L_2 \setminus L_1$ is different from the set difference $L_2 \setminus L_1$.)

1. Given NFA $A_1, A_2$, construct an NFA $A$ such that $L(A) = L(A_1) \setminus L(A_2)$.

2. Do the same for the \textit{right quotient} of $L_1$ by $L_2$, defined as $L_1 / L_2 := \{u \in \Sigma^* \mid \exists v \in L_2 : uv \in L_1\}$.

3. Determine the inclusion relations between the following languages: $L_1$, $(L_1 / L_2)L_2$, and $(L_1L_2) / L_2$.
Exercise 45  We have shown in Exercise 31 that every upward-closed language is regular. A language \( L \subseteq \Sigma^* \) is downward-closed if for every two words \( w, v \in \Sigma^* \), if \( w \in L \) and \( v \preceq w \), then \( v \in L \).

- Prove that every downward-closed language is regular.
- Give an algorithm that transforms a regular expression for a language into a regular expression for its downward-closure.

Exercise 46  (Abdulla, Bouajjani, and Jonsson) An atomic expression over an alphabet \( \Sigma^* \) is an expression of the form \( \emptyset, \epsilon, (a + \epsilon) \) or \( (a_1 + \ldots + a_n)^* \), where \( a, a_1, \ldots, a_n \in \Sigma \). A product is a concatenation \( e_1 e_2 \ldots e_n \) of atomic expressions. A simple regular expression is a sum \( p_1 + \ldots + p_n \) of products.

- Prove that the language of a simple regular expression is downward-closed.
- Prove that every downward-closed language can be represented by a simple regular expression.
  Hint: since every downward-closed language is regular, it is represented by a regular expression. Prove that this expression is equivalent to a simple regular expression.

Exercise 47  Let \( L_i = \{ w \in \{a\}^* \mid \text{the length of } w \text{ is divisible by } i \} \).

1. Construct an NFA for \( L := L_4 \cup L_6 \) with at most 11 states.
2. Construct the minimal DFA for \( L \).

Exercise 48

- Modify algorithm \textit{Empty} so that when the DFA or NFA is nonempty it returns a witness, i.e., a word accepted by the automaton.
- Same for a shortest witness.

Exercise 49  Check by means of \textit{UnivNFA} whether the following NFA is universal.
4.2. IMPLEMENTATION ON NFAS

Exercise 50  Let $\Sigma$ be an alphabet, and define the shuffle operator $\parallel : \Sigma^* \times \Sigma^* \rightarrow 2^{\Sigma^*}$ as follows, where $a, b \in \Sigma$ and $w, v \in \Sigma^*$:

- $w \parallel \varepsilon = \{w\}$
- $\varepsilon \parallel w = \{w\}$
- $aw \parallel bv = \{a\} (w \parallel bv) \cup \{b\} (aw \parallel v) \cup \{bw \mid w \in au\parallel v\}$

For example we have:

- $b \parallel d = \{bd, db\}$
- $ab \parallel d = \{abd, adb, dab\}$
- $ab \parallel cd = \{cabd, acbd, abcd, cadb, acdb, cdab\}$

Given DFAs recognizing languages $L_1, L_2 \subseteq \Sigma^*$ construct an NFA recognizing their shuffle

$L_1 \parallel L_2 := \bigcup_{u \in L_1, v \in L_2} u \parallel v$.

Exercise 51  Let $\Sigma_1, \Sigma_2$ be two alphabets. A homomorphism is a map $h : \Sigma_1^* \rightarrow \Sigma_2^*$ such that $h(\varepsilon) = \varepsilon$ and $h(w_1w_2) = h(w_1)h(w_2)$ for every $w_1, w_2 \in \Sigma_1^*$. Observe that if $\Sigma_1 = \{a_1, \ldots, a_n\}$ then $h$ is completely determined by the values $h(a_1), \ldots, h(a_n)$.

1. Let $h : \Sigma_1^* \rightarrow \Sigma_2^*$ be a homomorphism and let $A$ be a NFA over $\Sigma_1$. Describe how to construct a NFA for the language $h(L(A)) := \{h(w) \mid w \in L(A)\}$.

2. Let $h : \Sigma_1^* \rightarrow \Sigma_2^*$ be a homomorphism and let $A$ be a NFA over $\Sigma_2$. Describe how to construct a NFA for the language $h^{-1}(L(A)) := \{w \in \Sigma_1^* \mid h(w) \in L(A)\}$.

3. Recall that the language $\{0^n1^n \mid n \in \mathbb{N}\}$ is not regular. Use the preceding results to show that $\{(01^k2)^n3^n \mid k, n \in \mathbb{N}\}$ is also not regular.
Exercise 52  Given alphabets $\Sigma$ and $\Delta$, a substitution is a map $f : \Sigma \rightarrow 2^{\Delta^*}$ assigning to each letter $a \in \Sigma$ a language $L_a \subseteq \Delta^*$. A substitution $f$ can be canonically extended to a map $2^\Sigma \rightarrow 2^{\Delta^*}$ by defining $f(\varepsilon) = \varepsilon$, $f(wa) = f(w)f(a)$, and $f(L) = \bigcup_{w \in L} f(w)$. Note that a homomorphism can be seen as the special case of a substitution in which all $L_a$'s are singletons.

Let $\Sigma = \{\text{Name}, \text{Tel}, :, \#\}$, let $\Delta = \{A, \ldots, Z, 0, 1, \ldots, 9, :, \#\}$, and let $f$ be the substitution $f$ given by

$$
\begin{align*}
    f(\text{Name}) &= (A + \cdots + Z)^* \\
    f(,:) &= \{:\} \\
    f(\text{Tel}) &= 0049(1 + \ldots + 9)(0 + 1 + \ldots + 9)^{10} + 00420(1 + \ldots + 9)(0 + 1 + \ldots + 9)^8 \\
    f(\#) &= \{\#\}
\end{align*}
$$

1. Draw a DFA recognizing $L = \text{Name} : \text{Tel} (#\text{Telephone})^*$.
2. Sketch an NFA-reg recognizing $f(L)$.
3. Give an algorithm that takes as input an NFA $A$, a substitution $f$, and for every $a \in \Sigma$ an NFA recognizing $f(a)$, and returns an NFA recognizing $f(L(A))$.

Exercise 53  Given two NFAs $A_1$ and $A_2$, let $B = \text{NFAtoDFA} (\text{IntersNFA}(A_1, A_2))$ and $C = \text{IntersDFA} (\text{NFAtoDFA}(A_1), \text{NFAtoDFA}(A_2))$. Show that $B$ and $C$ are isomorphic, and so in particular have the same number of states.  
(A superficial analysis gives that for NFAs with $n_1$ and $n_2$ states $B$ and $C$ have $O(2^{n_1\cdot n_2})$ and $O(2^{n_1 + n_2})$ states, respectively, wrongly suggesting that $C$ might be more compact.)

Exercise 54  A DFA is synchronizing if there is a word $w$ and a state $q$ such that after reading $w$ from any state the DFA is always in state $q$.

1. Show that the following DFA is synchronizing.

```
Exercise 52  Given alphabets $\Sigma$ and $\Delta$, a substitution is a map $f : \Sigma \rightarrow 2^{\Delta^*}$ assigning to each letter $a \in \Sigma$ a language $L_a \subseteq \Delta^*$. A substitution $f$ can be canonically extended to a map $2^\Sigma \rightarrow 2^{\Delta^*}$ by defining $f(\varepsilon) = \varepsilon$, $f(wa) = f(w)f(a)$, and $f(L) = \bigcup_{w \in L} f(w)$. Note that a homomorphism can be seen as the special case of a substitution in which all $L_a$'s are singletons.

Let $\Sigma = \{\text{Name}, \text{Tel}, :, \#\}$, let $\Delta = \{A, \ldots, Z, 0, 1, \ldots, 9, :, \#\}$, and let $f$ be the substitution $f$ given by

$$
\begin{align*}
    f(\text{Name}) &= (A + \cdots + Z)^* \\
    f(,:) &= \{:\} \\
    f(\text{Tel}) &= 0049(1 + \ldots + 9)(0 + 1 + \ldots + 9)^{10} + 00420(1 + \ldots + 9)(0 + 1 + \ldots + 9)^8 \\
    f(\#) &= \{\#\}
\end{align*}
$$

1. Draw a DFA recognizing $L = \text{Name} : \text{Tel} (#\text{Telephone})^*$.
2. Sketch an NFA-reg recognizing $f(L)$.
3. Give an algorithm that takes as input an NFA $A$, a substitution $f$, and for every $a \in \Sigma$ an NFA recognizing $f(a)$, and returns an NFA recognizing $f(L(A))$.

Exercise 53  Given two NFAs $A_1$ and $A_2$, let $B = \text{NFAtoDFA} (\text{IntersNFA}(A_1, A_2))$ and $C = \text{IntersDFA} (\text{NFAtoDFA}(A_1), \text{NFAtoDFA}(A_2))$. Show that $B$ and $C$ are isomorphic, and so in particular have the same number of states.  
(A superficial analysis gives that for NFAs with $n_1$ and $n_2$ states $B$ and $C$ have $O(2^{n_1\cdot n_2})$ and $O(2^{n_1 + n_2})$ states, respectively, wrongly suggesting that $C$ might be more compact.)

Exercise 54  A DFA is synchronizing if there is a word $w$ and a state $q$ such that after reading $w$ from any state the DFA is always in state $q$.

1. Show that the following DFA is synchronizing.

```

```
```
2. Give an algorithm to decide if a given DFA is synchronizing.

3. Give a polynomial time algorithm to decide if a given DFA is synchronizing.
Chapter 5

Applications I: Pattern matching

As a first example of a practical application of automata, we consider the pattern matching problem. Given \( w, w' \in \Sigma^* \), we say that \( w' \) is a factor of \( w \) if there are words \( w_1, w_2 \in \Sigma^* \) such that \( w = w_1w'w_2 \). If \( w_1 \) and \( w_1w' \) have lengths \( k \) and \( k' \), respectively, we say that \( w' \) is the \([k, k']\)-factor of \( w \). The pattern matching problem is defined as follows: Given a word \( t \in \Sigma^+ \) (called the text), and a regular expression \( p \) over \( \Sigma \) (called the pattern), determine the smallest \( k \geq 0 \) such that some \([k', k]\)-factor of \( t \) belongs to \( L(p) \). We call \( k \) the first occurrence of \( p \) in \( t \).

Example 5.1 Let \( p = a(ab^*a)b \). Since \( ab, aabab \in L(p) \), the \([1, 3]\), \([3, 5]\), and \([0, 5]\)-factors of \( aabab \) belong to \( L(p) \). So the first occurrence of \( p \) in \( aabab \) is 3.

Usually one is interested not only in finding the ending position \( k \) of the \([k', k]\)-factor, but also in the starting position \( k' \). Adapting the algorithms to also provide this information is left as an exercise.

5.1 The general case

We present two different solutions to the pattern matching problem, using nondeterministic and deterministic automata, respectively.

Solution 1. Clearly, some word of \( L(p) \) occurs in \( t \) if and only if some prefix of \( t \) belongs to \( L(\Sigma^*p) \). So we construct an NFA \( A \) for the regular expression \( \Sigma^*p \) using the rules of Figure 2.9 on page 29, and then removing the \( \varepsilon \) transitions by means of \( \text{NFAtoNFA} \) on page 26. Let us call the resulting algorithm \( \text{RegtoNFA} \). Once \( A \) is constructed, we simulate \( A \) on \( t \) as in \( \text{MemNFA}[A](q_0, t) \) on page 62. Recall that the simulation algorithm reads the text letter by letter, maintaining the set of states reachable from the initial state by the prefix read so far. So the simulation reaches a set of states \( S \) containing a final state if and only if the prefix read so far belongs to \( L(\Sigma^*p) \). Here is the pseudocode for this algorithm:
PatternMatchingNFA(t, p)

Input: text \( t = a_1 \ldots a_n \in \Sigma^+ \), pattern \( p \in \Sigma^* \)

Output: the first occurrence of \( p \) in \( t \), or \( \bot \) if no such occurrence exists.

1. \( A \leftarrow \text{RegtoNFA}(\Sigma^* p) \)
2. \( S \leftarrow \{q_0\} \)
3. for all \( k = 0 \) to \( n - 1 \) do
   4. if \( S \cap F \neq \emptyset \) then return \( k \)
   5. \( S \leftarrow \delta(S, a_{k+1}) \)
4. return \( \bot \)

If we assume that the alphabet \( \Sigma \) has fixed size, then a rough estimate of the complexity of PatternMatchingNFA for a word of length \( n \) and a pattern of length \( m \) can be obtained as follows. \text{RegtoNFA} is the concatenation of \text{RegtoNFA}_e and \text{NFAtoNFA}. For a pattern of size \( m + 2 \), \text{RegtoNFA}_e takes \( \mathcal{O}(m) \) time, and outputs a NFA with \( \mathcal{O}(m) \) states and \( \mathcal{O}(m) \) transitions. So, when applied to this automaton, \text{NFAtoNFA} takes \( \mathcal{O}(m \cdot n \cdot m) \) time and outputs a NFA with \( \mathcal{O}(m) \) states and \( \mathcal{O}(nm^2) \) transitions (see page 28 for the complexity of \text{NFAtoNFA}). The loop is executed at most \( n \) times, and, for an automaton with \( \mathcal{O}(m) \) states, each line of the loop’s body takes at most \( \mathcal{O}(m^2) \) time. So the loop runs in \( \mathcal{O}(nm^2) \) time. The overall runtime is thus \( \mathcal{O}(nm^2) \).

Solution 2. We proceed as in the previous case, but constructing a DFA for \( \Sigma^* p \) instead of a NFA:

PatternMatchingDFA(t, p)

Input: text \( t = a_1 \ldots a_n \in \Sigma^+ \), pattern \( p \)

Output: the first occurrence of \( p \) in \( t \), or \( \bot \) if no such occurrence exists.

1. \( A \leftarrow \text{NFAtoDFA}(\text{RegtoNFA}(\Sigma^* p)) \)
2. \( q \leftarrow q_0 \)
3. for all \( k = 0 \) to \( n - 1 \) do
   4. if \( q \in F \) then return \( k \)
   5. \( q \leftarrow \delta(q, a_{k+1}) \)
4. return \( \bot \)

Notice that there is trade-off: while the conversion to a DFA can take (much) longer than the conversion to a NFA, the membership check for a DFA is faster. The complexity of PatternMatchingDFA for a word of length \( n \) and a pattern of length \( m \) can be easily estimated: \text{RegtoNFA}_e(p) runs in \( \mathcal{O}(m^3) \) time, but it outputs a NFA with only \( \mathcal{O}(m) \) states, and so the call to \text{NFAtoDFA} (see Table 2.3.1) takes \( 2^{\mathcal{O}(m)} \) time and space. Since the loop is executed at most \( n \) times, and each line of the body takes constant time, the overall runtime is \( \mathcal{O}(n) + 2^{\mathcal{O}(m)} \).
5.2 The word case

We study the special but very common special case of the pattern-matching problem in which we wish to know if a given word appears in a text. In this case the pattern \( p \) is the word itself. For the rest of the section we consider an arbitrary but fixed text \( t = a_1 \ldots a_n \) and an arbitrary but fixed word pattern \( p = b_1 \ldots b_m \).

It is easy to find a faster algorithm for this special case, without any use of automata theory: just move a “window” of length \( m \) over the text \( t \), one letter at a time, and check after each move whether the content of the window is \( p \). The number of moves is \( n - m + 1 \), and a check requires \( \mathcal{O}(m) \) letter comparisons, giving a runtime of \( \mathcal{O}(nm) \). In the rest of the section we present a faster algorithm with time complexity \( \mathcal{O}(m+n) \) independently of the size of the alphabet (i.e., the alphabet could have as many letters as the text). Notice that in some applications both \( n \) and \( m \) can be large, and the difference between \( \mathcal{O}(nm) \) and \( \mathcal{O}(m+n) \) very significant.

Figure 5.1(a) shows the obvious NFA \( A_p \) recognizing \( \Sigma^* p \) for \( p = \text{nano} \). In general, \( A_p = (Q, \Sigma, \delta, q_0, F) \), where \( Q = \{0, 1, \ldots, m\}, q_0 = 0, F = \{m\} \), and

\[
\delta = \{(i, b_{i+1}, i + 1) | i \in [0, m-1] \} \cup \{(0, a, 0) | a \in \Sigma\}.
\]

Clearly, \( A_p \) can reach state \( k \) whenever the word read so far ends with \( b_0 \ldots b_k \). Given a state \( i \in Q \) and a letter \( a \in \Sigma \), we say that \( a \) is a hit for \( i \) if it is the letter that leads from state \( i \) to state \( i+1 \); otherwise we say that \( a \) is a miss for \( i \). Intuitively, the hit is the letter that makes \( A_p \) “progress” towards reading \( p \) when it is at state \( i \).

Figure 5.1(b) shows the DFA \( B_p \) obtained by applying \NFAToDFA\ to \( A_p \). It has as many states as \( A_p \), and there is a natural correspondence between the states of \( A_p \) and \( B_p \); each state of \( A_p \) is the largest element of exactly one state of \( B_p \). For instance, 3 is the largest element of \( \{3, 1, 0\} \), and 4 is the largest element of \( \{4, 0\} \). We call this largest element the head of the state. The hit for a state of \( B_p \) is define as the hit of its head in \( A_p \). If we label a state with head \( k \) by the string \( b_1 \ldots b_k \), as shown in Figure 5.1(c), then we see that \( B_p \) keeps track of how close it is to finding \text{nano}. For instance:

- if \( B_p \) is in state \( n \) and reads an \( a \) (a hit), then it “makes progress”, and moves to state \( na \);
- if \( B_p \) is in state \( nan \) and reads an \( a \) (a miss), then it is “thrown back” to state \( na \). Not to \( \varepsilon \), because if the next two letters are \( n \) and \( o \), then \( B_p \) should accept!

\( B_p \) has another property that will be very important later on: for each state \( S \) of \( B_p \) (with the exception of \( S = \{0\} \)) the tail of \( S \), i.e., the result of removing the head of \( S \), is again a state of \( B_p \). For instance, the tail of \( \{3, 1, 0\} \) is \( \{1, 0\} \), which is also a state of \( B_p \). We show that this property and the ones above hold in general, and not only in the spacial case \( p = \text{nano} \). Formally, we prove the following invariant of \NFAToDFA\ (which is shown again in Table 5.2 for convenience) when applied to a word pattern \( p \). We use the notation \( \delta(S, a) = \cup_{q \in S} \delta(q, a) \), and given a set \( S \subseteq \{0, 1, \ldots, m\} \), denote the head and tail of \( S \) by \( h(S) \) and \( t(S) \), respectively.
Proposition 5.2 Let \( p = b_1 \ldots b_m \), and let \( S_k \) be the \( k \)-th set picked from the worklist during the execution of \( \text{NFAtoDFA}(A_p) \). Then

1. \( h(S_k) = k \) (and therefore \( k \leq m \)), and

2. either \( k = 0 \) and \( t(S_k) = \emptyset \), or \( k > 0 \) and \( t(S_k) \in \Omega \).

Proof: We prove that (1) and (2) hold for all \( 0 \leq k \leq m - 1 \), and moreover that right before the \( k \)-th iteration of the while loop the worklist only contains \( S_k \) (we call this (3)).

Clearly, (1), (2), and (3) hold for \( k = 0 \). For \( k > 0 \), consider the \( k \)-th iteration of the while loop. By induction hypothesis, we have \( h(S_k) = k \) by (1), \( t(S_k) = S_l \) for some \( l < k \) by (2), and by (3) we know that before the \( k \)-th iteration the worklist only contains \( S_k \). So the algorithm picks \( S_k \) from
the worklist and examines the sets $\delta(S_k, a)$ for every action $a$. We consider two cases: $a$ is a hit for $S_k$, and $a$ is a miss for $S_k$.

If $a$ is a miss for $S_k$, then by definition it is also a miss for its head $h(S_k) = k$. So we have $\delta(k, a) = \emptyset$, and hence $\delta(S_k, a) = \delta(t(S_k), a) = \delta(S_l, a)$. So $\delta(S_k, a)$ was already explored by the algorithm during the $l$-th iteration of the loop, and $\delta(S_k, a)$ is not added to the worklist at line 9.

If $a$ is a hit for $S_k$, then $\delta(k, a) = \{k + 1\}$. Since $\delta(S_k, a) = \delta(h(S_k), a) \cup \delta(t(S_k), a)$, we get $\delta(S_k, a) = \{k + 1\} \cup \delta(S_l, a)$. Since state $k + 1$ has not been explored before, the set $\{k + 1\} \cup \delta(S_l, a)$ is new, and so it becomes the $(k + 1)$-st state added to the worklist, i.e., $S_{k+1} = \{k + 1\} \cup \delta(S_l, a)$. So we get $h(S_{k+1}) = k + 1$, and, since $t(S_{k+1}) = t(S_l)$, also $t(S_{k+1}) \in \mathcal{Q}$; in other words, $S_{k+1}$ satisfies (1) and (2). Moreover, $S_{k+1}$ is the only state added to the worklist during the $k$-th iteration of the loop, and so we also have (3).

Finally, consider the $m$-th iteration of the loop. Since no letter is a hit for state $m$, the algorithm does not add any new state to the worklist, and so at the end of the iteration the worklist is empty and the algorithm terminates. □

By Proposition 5.2, $B_p$ has at most $m + 1$ states for a pattern of length $m$. So $NFAtoDFA$ does not incur in any exponential blowup for word patterns. Even more: any DFA for $\Sigma^* p$ must have at least $m + 1$ states, because for any two distinct prefixes $p_1, p_2$ of $p$ the residuals $(\Sigma^* p)^{p_1}$ and $(\Sigma^* p)^{p_2}$ are also distinct. So we get

**Corollary 5.3** $B_p$ is the minimal DFA recognizing $\Sigma^* p$.

$B_p$ has $m + 1$ states and $m|\Sigma|$ transitions. Since $\Sigma$ has size $\Theta(n)$, $B_p$ has size $\Theta(nm)$, and so its construction requires at least $\Theta(nm)$ time. We can do better by storing $B_p$ more compactly.
Transitions of $B_p$ labeled by letters that do not appear in $p$ always lead to state 0, and so they do not need to be explicitly stored. The remaining $O(m)$ transitions for each state can be constructed and stored using $O(m^2)$ space and time. However, this improvement only leads to a $O(n + m^2)$ algorithm. To achieve $O(n + m)$ time we introduce an even more compact data structure: the lazy DFA for $\Sigma^*p$, that, as we shall see, can be constructed in $O(m)$ space and time.

5.2.1 Lazy DFAs

A DFA can be seen as the control unit of a machine that reads input from a tape divided into cells by means of a reading head (see Figure 5.2). Initially, the tape contains the word to be read, and the reading head is positioned on the first cell of the tape. At each step, the machine reads the content of the cell occupied by the reading head, updates the current state according to the transition function, and advances the head one cell to the right. The machine accepts a word if the state reached after reading it completely is final.

In lazy DFAs the machine may advance the head one cell to the right or keep it on the same cell. Which of the two takes place is a function of the current state and the current letter read by the head. Formally, a lazy DFA only differs from a DFA in the transition function, which has the form $\delta: Q \times \Sigma \rightarrow Q \times \{R, N\}$, where $R$ stands for move Right and $N$ stands for No move. A transition of a lazy DFA is a fourtuple $(q, a, q', d)$, where $d \in \{R, N\}$ is the move of the head. Intuitively, a transition $(q, a, q', N)$ means that state $q$ is lazy and delegates processing the letter $a$ to the state $q'$.

In order to construct a lazy DFA for $\Sigma^*p$, recall that every state $S_k$ of $B_p$ has a hit letter, namely the letter $a$ that makes $B_p$ move from $S_k$ to $S_{k+1}$, and all other letters are misses. In particular, if $a$ is a miss, then $\delta_B(S_k, a) = \delta(t(S_k), a)$, and so $B_p$ moves to the same state it would move to if it were in state $t(S_k)$.

Using this, we construct a lazy DFA $C_p$ with the same states as $B_p$, and transition function $\delta_C(S_k, a)$ for a state $S_k$ and a letter $a$ given by:

- If $a$ is a hit, then $C_p$ behaves as $B_p$, i.e., $\delta_C(S_k, a) = (S_{k+1}, R)$.
- If $a$ is a miss, then $S_k$ delegates to the state $t(S_k)$, i.e., $\delta_C(S_k, a) = (t(S_k), N)$. However, if $k = 0$ then $t(S_k)$ is not a state, and $C_p$ moves as $B_p$, i.e., $\delta_C(S_0, a) = (S_0, R)$.

In the case of a miss $C_p$ always delegates to the same state, independently of the letter being read, and so we can “summarize” the transitions for all misses into a single transition $\delta_C(S_k, miss) = (t(S_k), N)$. Figure 5.3 shows again the DFA and the lazy DFA with summarized transitions for $p = nano$. (We also write just $k$ instead of $S_k$ in the states.) Consider the behaviours of $B_p$ and $C_p$.
from state $S_3$ when the next letter is a $n$. $B_p$ moves to $S_1$ (because that is what it would do if it were in state $S_1$), while $C_p$ delegates to $S_1$, which delegates to $S_0$, which moves to state $S_1$: the move of $B_p$ is simulated in $C_p$ by a chain of delegations, followed by a last move that advances the reading head to the right (in the worst case the chain of delegations reaches $S_0$, who cannot delegate to anybody). The final destination is clearly the same in both cases.

![Diagram](image)

**Figure 5.3: DFA and lazy DFA for $p = nano$**

Notice that $C_p$ may require more steps than $B_p$ to read the text. But the number of steps is at most $2n + m$. As mentioned above, $C_p$ performs for every letter of the text a number of $N$-steps, followed by one $R$-step. Call this sequence a macrostep, and let $S_{j_i}$ be the state reached after the $i$-th macrostep. Since the $i$-th macrostep leads from $S_{j_{i-1}}$ to $S_{j_i}$, and $N$-steps never move forward along the spine, the number of steps of the $i$-th macrostep is bounded by $j_{i-1} - j_i + 2$. So the total number of steps is bounded by

$$\sum_{i=1}^{n} (j_{i-1} - j_i + 2) = j_n - j_0 + 2n \leq m + 0 + 2n = m + 2n .$$

It remains to show that $C_p$ can be constructed in $O(m)$ time. Let $Miss(i)$ be the head of the target state of the miss transition for $S_i$. For instance, for $p = nano$ we have $Miss(3) = 1$ and $Miss(i) = 0$ for $i = 0, 1, 2, 4$. Clearly, if we can compute all of $Miss(0), \ldots, Miss(m)$ together in $O(m)$ time, then we can construct $C_p$ in $O(m)$ time.
Let $n$. We prove it remains to show that all calls to Table, which returns the head of the state of a miss the state $S$ can then be easily transformed into an algorithm for Miss. By definition, for every $i > 0$ in the case of a miss the state $S_i$ delegates to $t(S_i)$, i.e., $miss(S_i) = t(S_i)$. Since $t(S_1) = \{0\} = S_0$, this already gives $miss(S_1) = S_0$. For $i > 1$ we have $S_i = [i] \cup t(S_i)$, and so

$$t(S_i) = t(\delta_B(S_{i-1}, b_i)) = t(\delta(i-1, b_i) \cup \delta(t(S_{i-1}, b_i))) = t([i] \cup \delta(t(S_{i-1}, b_i))) = \delta_B(t(S_{i-1}), b_i).$$

Hence we get

$$miss(S_i) = \delta_B(miss(S_{i-1}), b_i)$$

Moreover, we have

$$\delta_B(S_j, b) = \begin{cases} S_{j+1} & \text{if } b = b_{j+1} \text{ (hit)} \\ S_0 & \text{if } b \neq b_{j+1} \text{ (miss) and } j = 0 \\ \delta_B(t(S_j), b) & \text{if } b \neq b_{j+1} \text{ (miss) and } j \neq 0 \end{cases}$$

Putting these two last equations together, and recalling that $miss(S_0) = S_0$, we finally obtain

$$miss(S_i) = \begin{cases} S_0 & \text{if } i = 0 \text{ or } i = 1 \\ \delta_B(miss(S_{i-1}), b_i) & \text{if } i > 1 \end{cases}$$  \hspace{1cm} (5.1)

$$\delta_B(S_j, b) = \begin{cases} S_{j+1} & \text{if } b = b_{j+1} \text{ (hit)} \\ S_0 & \text{if } b \neq b_{j+1} \text{ (miss) and } j = 0 \\ \delta_B(miss(S_j), b) & \text{if } b \neq b_{j+1} \text{ (miss) and } j \neq 0 \end{cases}$$ \hspace{1cm} (5.2)

Equations 5.1 and 5.2 lead to the algorithms shown in Table 5.2. CompMiss($p$) computes Miss($i$) for each index $i \in \{0, \ldots, m\}$. CompMiss($p$) calls DeltaB($j, b$), shown on the right of the Table, which returns the head of the state $\delta_B(S_j, b)$, i.e., DeltaB($j, b$) = $k$ iff $\delta_B(S_j, b) = S_k$.

**CompMiss($p$)**

**Input:** pattern $p = b_1 \cdots b_m$.

**Output:** heads of targets of miss transitions.

1. Miss(0) $\leftarrow$ 0; Miss(1) $\leftarrow$ 0
2. for $i \leftarrow 2, \ldots, m$ do
   1. Miss($i$) $\leftarrow$ DeltaB(Miss($i-1$), $b_i$)
3. **end for**

**DeltaB($j, b$)**

**Input:** head $j \in \{0, \ldots, m\}$, letter $b$.

**Output:** head of the state $\delta_B(S_j, b)$.

1. while $b \neq b_{j+1}$ and $j \neq 0$ do $j \leftarrow$ Miss($j$)
2. if $b = b_{j+1}$ then return $j + 1$
3. else return 0

**Table 5.2: Algorithm CompMiss($p$)**

It remains to show that all calls to DeltaB during the execution of Miss($p$) take together $\Theta(m)$ time. Let $n_i$ be the number of iterations of the while loop at line 1 during the call DeltaB(Miss($i-1$), $b_i$). We prove $\sum_{i=2}^{m} n_i \leq m$. To this end, observe that $n_i \leq$ Miss($i-1$) – (Miss($i-1$) – 1), because initially $j$
is set to Miss(i − 1), each iteration of the loop decreases j by at least 1 (line 2), and the return value assigned to Miss(i) is at most the final value of j plus 1. So we get
\[
\sum_{i=2}^{m} n_i \leq \sum_{i=2}^{m} (Miss(i - 1) - Miss(i) + 1) \leq Miss(1) - Miss(m) + m \leq Miss(1) + m = m
\]

**Exercises**

**Exercise 55** Design an algorithm that solves the following problem for an alphabet Σ: Given a text \( t \in \Sigma^* \) and a regular expression \( p \) over Σ, find a shortest prefix \( w_1 \in \Sigma^* \) of \( t \) such that some prefix \( w_1w_2 \) of \( w \) satisfies \( w_2 \in L(p) \). Discuss the complexity of your solution.

**Exercise 56** Construct NFAs, DFAs, and lazy DFAs for the patterns mammamia and abracadabra.

**Exercise 57** We have shown that lazy DFAs for a word pattern may need more than \( n \) steps to read a text of length \( n \), but not more than \( 2n + m \), where \( m \) is the length of the pattern. Find a text \( t \) and a word pattern \( p \) such that the run of \( B_p \) on \( t \) takes at most \( n \) steps and the run of \( C_p \) takes at least \( 2n - 1 \) steps.

*Hint:* a simple pattern of the form \( a^k \) for some \( k \geq 0 \) is sufficient.

**Exercise 58** Two-way finite automata are an extension of lazy automata in which the reading head may not only move right or stay put, but also move left. The tape extends infinitely long to both the left and to the right of the input with all cells empty. A word is accepted if the control state is a final state at the moment the head reaches an empty cell to the right of the input for the first time.

- Give a two-way DFA with \( O(n) \) states for the language \( (0 + 1)^*1(0 + 1)^n \).
- Give algorithms for membership and emptiness of two-way automata.
- (Difficult!) Prove that the languages recognized by two-way DFA are regular.
Chapter 6

Operations on Relations: Implementations

We show how to implement operations on relations over a (possibly infinite) universe $U$. Even though we will encode the elements of $U$ as words, when implementing relations it is convenient to think of $U$ as an abstract universe, and not as the set $\Sigma^*$ of words over some alphabet $\Sigma$. The reason, as we shall see, is that for some operations we encode an element of $X$ not by one word, but by many, actually by infinitely many. In the case of operations on sets this is not necessary, and one can safely identify the object and its encoding as word.

We are interested in a number of operations. A first group contains the operations we already studied for sets, but lifted to relations. For instance, we consider the operation $\text{Membership}((x, y), R)$ that returns $true$ if $(x, y) \in R$, and $false$ otherwise, or $\text{Complement}(R)$, that returns $\overline{R} = (X \times X) \setminus R$. Their implementations will be very similar to those of the language case. A second group contains three fundamental operations proper to relations. Given $R, R_1, R_2 \subseteq U \times U$:

- $\text{Projection}_1(R) :$ returns the set $\pi_1(R) = \{x \mid \exists y (x, y) \in R\}$.
- $\text{Projection}_2(R) :$ returns the set $\pi_2(R) = \{y \mid \exists x (x, y) \in R\}$.
- $\text{Join}(R_1, R_2) :$ returns $R_1 \circ R_2 = \{(x, z) \mid \exists y (x, y) \in R_1 \land (y, z) \in R_2\}$

Finally, given $X \subseteq U$ we are interested in two derived operations:

- $\text{Post}(X, R) :$ returns $\text{post}_R(X) = \{y \in U \mid \exists x \in X : (x, y) \in R\}$.
- $\text{Pre}(X, R) :$ returns $\text{pre}_R(X) = \{y \mid \exists x \in X : (y, x) \in R\}$.

Observe that $\text{Post}(X, R) = \text{Projection}_2(\text{Join}(\text{Id}_X, R))$, and $\text{Pre}(X, R) = \text{Projection}_1(\text{Join}(R, \text{Id}_x))$, where $\text{Id}_X = \{(x, x) \mid x \in X\}$. 

95
### 6.1 Encodings

We encode elements of $U$ as words over an alphabet $\Sigma$. It is convenient to assume that $\Sigma$ contains a padding letter $\#$, and that an element $x$ is encoded not only by a word $s_x \in \Sigma^*$, but by all the words $s_x\#^n$ with $n \geq 0$. That is, an element $x$ has a shortest encoding $s_x$, and other encodings are obtained by appending to the shortest encoding an arbitrary number of padding letters. We assume that the shortest encodings of two distinct elements are also distinct, and that for every $x \in U$ the last letter of $s_x$ is different from $. It follows that the sets of encodings of two distinct elements are disjoint.

The advantage is that for any two elements $x, y$ there is a number $n$ (in fact infinitely many) such that both $x$ and $y$ have encodings of length $n$. We say that $(w_x, w_y)$ encodes the pair $(x, y)$ if $w_x$ encodes $x$, $w_y$ encodes $y$, and $w_x, w_y$ have the same length. Notice that if $(w_x, w_y)$ encodes $(x, y)$, then so does $(w_x\#^k, w_y\#^k)$ for every $k \geq 0$. If $s_x, s_y$ are the shortest encodings of $x$ and $y$, and $|s_x| \leq |s_y|$, then the shortest encoding of $(x, y)$ is $(s_x\#^{|s_y|-|s_x|}, s_y)$.

**Example 6.1** We encode the number 6 not only by its small end binary representation 011, but by any word of $L(0110^*)$. In this case we have $\Sigma = \{0, 1\}$ with 0 as padding letter. Notice, however, that taking 0 as padding letter requires to take the empty word as the shortest encoding of the number 0 (otherwise the last letter of the encoding of 0 is the padding letter).

In the rest of this chapter, we will use this particular encoding of natural numbers without further notice. We call it the least significant bit first encoding and write $\text{lsbf}(6)$ to denote the language $L(0110^*)$.

If we encode an element of $U$ by more than one word, then we have to define when is an element accepted or rejected by an automaton. Does it suffice that the automaton accepts(rejects) some encoding, or does it have to accept (reject) all of them? Several definitions are possible, leading to different implementations of the operations. We choose the following option:

**Definition 6.2** Assume an encoding of the universe $U$ over $\Sigma^*$ has been fixed. Let $A$ be an NFA.

- $A$ accepts $x \in U$ if it accepts all encodings of $x$.
- $A$ rejects $x \in U$ if it accepts no encoding of $x$.
- $A$ recognizes a set $X \subseteq U$ if

$$L(A) = \{w \in \Sigma^* \mid w \text{ encodes some element of } X\}.$$ 

A set is regular (with respect to the fixed encoding) if it is recognized by some NFA.

Notice that if $A$ recognizes $X \subseteq U$ then, as one would expect, $A$ accepts every $x \in X$ and rejects every $x \notin X$. Observe further that with this definition a NFA may neither accept nor reject a given $x$. An NFA is well-formed if it recognizes some set of objects, and ill-formed otherwise.
6.2 Transducers and Regular Relations

We assume that an encoding of the universe $U$ over the alphabet $\Sigma$ has been fixed.

**Definition 6.3** A transducer over $\Sigma$ is an NFA over the alphabet $\Sigma \times \Sigma$.

Transducers are also called Mealy machines. According to this definition a transducer accepts sequences of pairs of letters, but it is convenient to look at it as a machine accepting pairs of words:

**Definition 6.4** Let $T$ be a transducer over $\Sigma$. Given words $w_1 = a_1a_2\ldots a_n$ and $w_2 = b_1b_2\ldots b_n$, we say that $T$ accepts the pair $(w_1, w_2)$ if it accepts the word $(a_1, b_1)\ldots(a_n, b_n) \in (\Sigma \times \Sigma)^*$. In other words, we identify the sets $\bigcup_{i\geq 0} (\Sigma^i \times \Sigma^i)$ and $(\Sigma \times \Sigma)^* = \bigcup_{i\geq 0} (\Sigma \times \Sigma)^i$.

We now define when a transducer accepts a pair $(x, y) \in X \times X$, which allows us to define the relation recognized by a transducer. The definition is completely analogous to Definition 6.2

**Definition 6.5** Let $T$ be a transducer.

- $T$ accepts a pair $(x, y) \in X \times X$ if it accepts all encodings of $(x, y)$.
- $T$ rejects a pair $(x, y) \in X \times X$ if it accepts no encoding of $(x, y)$.
- $T$ recognizes a relation $R \subseteq X \times X$ if

$$L(T) = \{(w_x, w_y) \in (\Sigma \times \Sigma)^* \mid (w_x, w_y) \text{ encodes some pair of } R\}.$$ 

A relation is regular if it is recognized by some transducer.

It is important to emphasize that not every transducer recognizes a relation, because it may recognize only some, but not all, the encodings of a pair $(x, y)$. As for NFAs, we say a transducer if well-formed if it recognizes some relation, and ill-formed otherwise.

**Example 6.6** The Collatz function is the function $f : \mathbb{N} \to \mathbb{N}$ defined as follows:

$$f(n) = \begin{cases} 
3n + 1 & \text{if } n \text{ is odd} \\
n/2 & \text{if } n \text{ is even}
\end{cases}$$

Figure 6.1 shows a transducer that recognizes the relation $\{(n, f(n)) \mid n \in \mathbb{N}\}$ with $\text{lsbf}$ encoding and with $\Sigma = \{0, 1\}$. The elements of $\Sigma \times \Sigma$ are drawn as column vectors with two components. The transducer accepts for instance the pair $(7, 22)$ because it accepts the pairs $(111000^k, 011010^k)$, that is, it accepts

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \end{bmatrix}^k$$

for every $k \geq 0$, and we have $\text{lsbf}(7) = L(1110^* +)$ and $\text{lsbf}(22) = L(011010^*)$. \qed
Determinism  A transducer is deterministic if it is a DFA. In particular, a state of a deterministic transducer over the alphabet $\Sigma \times \Sigma$ has exactly $|\Sigma|^2$ outgoing transitions. The transducer of Figure 6.1 is deterministic in this sense, when an appropriate sink state is added.

There is another possibility to define determinism of transducers, in which the letter $(a, b)$ is interpreted as “the transducer receives the input $a$ and produces the output $b$”. In this view, a transducer is called deterministic if for every state $q$ and every letter $a$ there is exactly one transition of the form $(q, (a, b), q')$. Observe that these two definitions of determinism are not equivalent.

We do not give separate implementations of the operations for deterministic and nondeterministic transducers. The new operations (projection and join) can only be reasonably implemented on nondeterministic transducers, and so the deterministic case does not add anything new to the discussion of Chapter 4.

6.3 Implementing Operations on Relations

In Chapter 4 we made two assumptions on the encoding of objects from the universe $U$ as words:

- every word is the encoding of some object, and
- every object is encoded by exactly one word.

We have relaxed the second assumption, and allowed for multiple encodings (in fact, infinitely many), of an object. Fortunately, as long as the first assumption still holds, the implementations of the boolean operations remain correct, in the following sense: If the input automata are well formed then the output automaton is also well-formed. Consider for instance the complementation operation on DFAs. Since every word encodes some object, the set of all words can be partitioned in
equivalence classes, each of them containing all the encodings of an object. If the input automaton
A is well-formed, then for every object x from the universe, A either accepts all the words in an
equivalence class, or none of them. The complement automaton then satisfies the same property,
but accepting a class iff the original automaton does not accept it.

Notice further that membership of an object x in a set represented by a well-formed automaton
can be checked by taking any encoding wx of x, and checking if the automaton accepts wx.

### 6.3.1 Projection

Given a transducer T recognizing a relation \( R \subseteq X \times X \), we construct an automaton over \( \Sigma \) recognizing the set \( \pi_1(R) \). The initial idea is very simple: loosely speaking, we go through all transitions,
and replace their labels \((a, b)\) by \(a\). This transformation yields a NFA, and this NFA has an accepting
run on a word \(a_1 \ldots a_n\) iff the transducer has an accepting run on some pair \((w, w')\). Formally,
this step is carried out in lines 1-4 of the following algorithm (line 5 is explained below):

\[
\begin{align*}
Proj_1(T) \\
\text{Input:} & \text{ transducer } T = (Q, \Sigma \times \Sigma, \delta, q_0, F) \\
\text{Output:} & \text{ NFA } A = (Q', \Sigma, \delta', q'_0, F') \text{ with } L(A) = \pi_1(L(T)) \\
1 & Q' \leftarrow Q; q'_0 \leftarrow q_0; F'' \leftarrow F \\
2 & \delta' \leftarrow \emptyset; \\
3 & \text{for all } ((q, (a, b), q')) \in \delta \text{ do} \\
4 & \quad \text{add } (q, a, q') \text{ to } \delta' \\
5 & F' \leftarrow \text{PadClosure}((Q', \Sigma, \delta', q'_0, F''), \#)
\end{align*}
\]

However, this initial idea is not fully correct. Consider the relation \( R = \{(1, 4)\} \) over \( \mathbb{N} \). A
transducer T recognizing R recognizes the language

\[\{(10^{n+2}, 0010^n) \mid n \geq 0\}\]

and so the NFA constructed after lines 1-4 recognizes \(\{10^{n+2} \mid n \geq 0\}\). However, it does not
recognize the number 1, because it does not accept all its encodings: the encodings 1 and 10 are
rejected.

This problem can be easily repaired. We introduce an auxiliary construction that “completes”
a given NFA: the \textit{padding closure} of an NFA is another NFA \(A'\) that accepts a word \(w\) if and only
if the first NFA accepts \(w\#^n\) for some \(n \geq 0\) and a padding symbol \#. Formally, the padding closure
augments the set of final states and return a new such set. Here is the algorithm constructing the
padding closure:
Chapter 6. Operations on Relations: Implementations

PadClosure(A, #)

**Input:** NFA A = (Σ × Σ, Q, δ, q₀, F)

**Output:** new set F' of final states

1. W ← F; F' ← ∅;

2. while W ≠ ∅ do

3. pick q from W

4. add q to F'

5. for all (q', #, q) ∈ δ do

6. if q' ≠ F' then add q' to W

7. return F'

Projection onto the second component is implemented analogously. The complexity of Proj_i() is clearly $O(|δ| + |Q|)$, since every transition is examined at most twice, once in line 3, and possibly a second time at line 5 of PadClosure().

Observe that projection does not preserve determinism, because two transitions leaving a state and labeled by two different (pairs of) letters (a, b) and (a, c), become after projection two transitions labeled with the same letter a: In practice the projection of a transducer is hardly ever deterministic. Since, typically, a sequence of operations manipulating transitions contains at least one projection, deterministic transducers are relatively uninteresting.

Example 6.7 Figure 6.2 shows the NFA obtained by projecting the transducer for the Collatz function onto the first and second components. States 4 and 5 of the NFA at the top (first component) are made final by PadClosure(), because they can both reach the final state 6 through a chain of 0s (recall that 0 is the padding symbol in this case). The same happens to state 3 for the NFA at the bottom (second component), which can reach the final state 2 with 0.

Recall that the transducer recognizes the relation $R = \{(n, f(n)) \mid n \in \mathbb{N}\}$, where f denotes the Collatz function. So we have $π_1(R) = \{n \mid n \in \mathbb{N}\} = \mathbb{N}$ and $π_2(R) = \{f(n) \mid n \in \mathbb{N}\}$, and a moment of thought shows that $π_2(R) = \mathbb{N}$ as well. So both NFAs should be universal, and the reader can easily check that this is indeed the case. Observe that both projections are nondeterministic, although the transducer is deterministic.

6.3.2 Join, Post, and Pre

We give an implementation of the Join operation, and then show how to modify it to obtain implementations of Pre and Post.

Given transducers $T_1, T_2$ recognizing relations $R_1$ and $R_2$, we construct a transducer $T_1 \circ T_2$ recognizing $R_1 \circ R_2$. We first construct a transducer $T$ with the following property: $T$ accepts (w, w') iff there is a word $w''$ such that $T_1$ accepts (w, w'') and $T_2$ accepts (w'', w'). The intuitive idea is to slightly modify the product construction. Recall that the pairing $[A_1, A_2]$ of two NFA $A_1, A_2$ has a transition $[q, r] \xrightarrow{a} [q', r']$ if and only if $A_1$ has a transition $q \xrightarrow{a} r$ and $A_2$ has a transition $q' \xrightarrow{a} r'$. Similarly, $T_1 \circ T_2$ has a transition $[q, r] \xrightarrow{(a, b)} [q', r']$ if there is a letter c such that $T_1$ has a transition
6.3. IMPLEMENTING OPERATIONS ON RELATIONS

Figure 6.2: Projections of the transducer for the Collatz function onto the first and second components.

$q \xrightarrow{(a,c)} r$ and $A_2$ has a transition $q' \xrightarrow{(c,b)} r'$. Intuitively, $T$ can output $b$ on input $a$ if there is a letter $c$ such that $T_1$ can output $c$ on input $a$, and $T_2$ can output $b$ on input $c$. The transducer $T$ has a run

$$
\begin{bmatrix}
q_{01} \\ q_{02}
\end{bmatrix}
\xrightarrow{egin{bmatrix}a_1 \\ b_1
\end{bmatrix}}
\begin{bmatrix}
q_{11} \\ q_{12}
\end{bmatrix}
\xrightarrow{egin{bmatrix}a_2 \\ b_2
\end{bmatrix}}
\begin{bmatrix}
q_{21} \\ q_{22}
\end{bmatrix}
\cdots
\begin{bmatrix}
q_{(n-1)1} \\ q_{(n-1)2}
\end{bmatrix}
\xrightarrow{egin{bmatrix}a_n \\ b_n
\end{bmatrix}}
\begin{bmatrix}
q_{n1} \\ q_{n2}
\end{bmatrix}
$$
iff \( T_1 \) and \( T_2 \) have runs

\[
\begin{array}{cccccc}
q_{01} & [a_1] & \rightarrow & q_{11} & \rightarrow & q_{21} & \rightarrow & \cdots & q_{(n-1)1} & \rightarrow & q_{n1} \\
q_{02} & [c_1] & \rightarrow & q_{12} & \rightarrow & q_{22} & \rightarrow & \cdots & q_{(n-1)2} & \rightarrow & q_{n2}
\end{array}
\]

Formally, if \( T_1 = (Q_1, \Sigma \times \delta_1, q_{01}, F_1) \) and \( T_2 = (Q_2, \Sigma \times \delta_2, q_{02}, F_2) \), then \( T = (Q, \Sigma \times \delta, q_0, F') \) is the transducer generated by lines 1–9 of the algorithm below:

\[
\text{Join}(T_1, T_2)
\]

Input: transducers \( T_1 = (Q_1, \Sigma \times \delta_1, q_{01}, F_1) \), \( T_2 = (Q_2, \Sigma \times \delta_2, q_{02}, F_2) \)

Output: transducer \( T_1 \circ T_2 = (Q, \Sigma \times \delta, q_0, F) \)

1. \( Q, \delta, F' \leftarrow \emptyset; \ q_0 \leftarrow [q_{01}, q_{02}] \)
2. \( W \leftarrow \{[q_{01}, q_{02}]\} \)
3. while \( W \neq \emptyset \) do
4. \( \text{pick} [q_1, q_2] \text{ from } W \)
5. \( \text{add } [q_1, q_2] \text{ to } Q \)
6. if \( q_1 \in F_1 \) and \( q_2 \in F_2 \) then \( \text{add } [q_1, q_2] \text{ to } F' \)
7. for all \( (q_1, (a, c), q'_1) \in \delta_1, (q_2, (c, b), q'_2) \in \delta_2 \) do
8. \( \text{add } ([q_1, q_2], (a, b), [q'_1, q'_2]) \text{ to } \delta \)
9. if \( [q'_1, q'_2] \notin Q \) then \( \text{add } [q'_1, q'_2] \text{ to } W \)
10. \( F \leftarrow \text{PadClosure}((Q, \Sigma \times \delta, q_0, F'), (\#, \#)) \)

However, \( T \) is not yet the transducer we are looking for. The problem is similar to the one of the projection operation. Consider the relations on numbers \( R_1 = \{(2, 4)\} \) and \( R_2 = \{(4, 2)\} \). Then \( T_1 \) and \( T_2 \) recognize the languages \( \{(010^{n+1}, 0010^n) \mid n \geq 0\} \) and \( \{(0010^n, 010^{n+1}) \mid n \geq 0\} \) of word pairs. So \( T \) recognizes \( \{(010^{n+1}, 010^{n+1}) \mid n \geq 0\} \). But then, according to our definition, \( T \) does not accept the pair \((2, 2) \in \mathbb{N} \times \mathbb{N}\), because it does not accept all its encodings: the encoding \((01, 01)\) is missing. So we add a padding closure again at line 10, this time using \( [\#, \#] \) as padding symbol.

The number of states of \( \text{Join}(T_1, T_2) \) is \( 0(\{|Q_1| \cdot |Q_2|\}) \), as for the standard product construction.

Example 6.8 Recall that the transducer of Figure 6.1, shown again at the top of Figure 6.3, recognizes the relation \( \{(n, f(n)) \mid n \in \mathbb{N}\} \), where \( f \) is the Collatz function. Let \( T \) be this transducer. The bottom part of Figure 6.3 shows the transducer \( T \circ T \) as computed by \( \text{Join}(T, T) \). For example, the transition leading from \([2, 3]\) to \([3, 2]\), labeled by \((0, 0)\), is the result of “pairing” the transition from 2 to 3 labeled by \((0, 1)\), and the one from 3 to 2 labeled by \((1, 0)\). Observe that \( T \circ T \) is not deterministic, because for instance \([1, 1]\) is the source of two transitions labeled by \((0, 0)\), even
Figure 6.3: A transducer for $f(f(n))$. 
though \( T \) is deterministic. This transducer recognizes the relation \( \{(n, f(f(n))) \mid n \in \mathbb{N}\} \). A little calculation gives

\[
f(f(n)) = \begin{cases} 
  n/4 & \text{if } n \equiv 0 \mod 4 \\
  3n/2 + 1 & \text{if } n \equiv 2 \mod 4 \\
  3n/2 + 1/2 & \text{if } n \equiv 1 \mod 4 \text{ or } n \equiv 3 \mod 4
\end{cases}
\]

The three components of the transducer reachable from the state \([1, 1]\) correspond to these three cases.

**Post**\((X, R)\) and **Pre**\((X, R)\) Given an NFA \( A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \) recognizing a regular set \( X \subseteq U \) and a transducer \( T_2 = (Q_2, \Sigma \times \Sigma, \delta_2, q_{02}, F_2) \) recognizing a regular relation \( R \subseteq U \times U \), we construct an NFA \( B \) recognizing the set \( \text{post}(R(U)) \). It suffices to slightly modify the join operation. The algorithm \( \text{Post}(A_1, T_2) \) is the result of replacing lines 7-8 of \( \text{Join}() \) by

\[
7 \quad \text{for all } (q_1, c, q_1') \in \delta_1, (q_2, (c, b), q_2') \in \delta_2 \text{ do} \\
8 \quad \text{add } ((q_1, q_2), b, [q_1', q_2']) \text{ to } \delta
\]

As for the join operation, the resulting NFA has to be postprocessed, closing it with respect to the padding symbol.

In order to construct an NFA recognizing \( \text{pre}(R(X)) \), we replace lines 7-8 by

\[
7 \quad \text{for all } (q_1, (a, c), q_1') \in \delta_1, (q_2, c, q_2') \in \delta_2 \text{ do} \\
8 \quad \text{add } \delta \text{ to } ([q_1, q_2], a, [q_1', q_2'])
\]

Notice that both post and pre are computed with the same complexity as the pairing construction, namely, the product of the number of states of transducer and NFA.

**Example 6.9** We construct an NFA recognizing the image under the Collatz function of all multiples of 3, i.e., the set \( \{f(3n) \mid n \in \mathbb{N}\} \). For this, we first need an automaton recognizing the set \( Y \) of all \( \text{lsbf} \) encodings of the multiples of 3. The following DFA does the job:

![Diagram of DFA for Example 6.9](image)

For instance, this DFA recognizes 0011 (encoding of 12) and 01001 (encoding of 18), but not 0101 (encoding of 10). We now compute \( \text{post}(Y) \), where, as usual, \( R = \{(n, f(n)) \mid n \in \mathbb{N}\} \). The result is the NFA shown in Figure 6.4. For instance, the transition \([1, 1] \rightarrow [1, 3]\) is generated by the transitions \(1 \rightarrow 1\) of the DFA and \(1 \rightarrow ([0, 1])\) of the transducer for the Collatz function. State \([2, 3]\) becomes final due to the closure with respect to the padding symbol 0.
Figure 6.4: Computing \( f(n) \) for all multiples of 3.
6.4 Relations of Higher Arity

The implementations described in the previous sections can be easily extended to relations of higher arity over the universe \( U \). We briefly describe the generalization.

Fix an encoding of the universe \( U \) over the alphabet \( \Sigma \) with padding symbol \( \# \). A tuple \((w_1, \ldots, w_k)\) of words over \( \Sigma \) encodes the tuple \((x_1, \ldots, x_k)\) of length \( k \) if \( w_i \) encodes \( x_i \) for every \( 1 \leq i \leq k \), and \( w_1, \ldots, w_k \) have the same length. A \( k \)-transducer over \( \Sigma \) is an NFA over the alphabet \( \Sigma^k \). Acceptance of a \( k \)-transducer is defined as for normal transducers.

Boolean operations are defined as for NFAs. The projection operation can be generalized to projection over an arbitrary subset of components. For this, given an index set \( I = \{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\} \), let \( \bar{x}_I \) denote the projection of a tuple \( \bar{x} = (x_1, \ldots, x_k) \in U^k \) over \( I \), defined as the tuple \((x_{i_1}, \ldots, x_{i_n}) \in U^n \). Given a relation \( R \subseteq U \), we define

\[
\text{Projection}_I(R) := \{ \bar{x}_I \mid \bar{x} \in R \}.
\]

The operation is implemented analogously to the case of a binary relation. Given a \( k \)-transducer \( T \) recognizing \( R \), the \( n \)-transducer recognizing \( \text{Projection}_I(R) \) is computed as follows:

- Replace every transition \((q, (a_1, \ldots, a_k), q')\) of \( T \) by the transition \((q, (a_{i_1}, \ldots, a_{i_n}), q')\).

- Compute the PAD-closure of the result: for every transition \((q, (\#, \ldots, \#), q')\), if \( q' \) is a final state, then add \( q \) to the set of final states.

The join operation can also be generalized. Given two tuples \( \bar{x} = (x_1, \ldots, x_n) \) and \( \bar{y} = (y_1, \ldots, y_m) \) of arities \( n \) and \( m \), respectively, we denote the tuple \((x_1, \ldots, x_n, y_1, \ldots, y_m)\) of dimension \( n + m \) by \( \bar{x} \cdot \bar{y} \). Given relations \( R_1 \subseteq U^{k_1} \) and \( R_2 \subseteq U^{k_2} \) of arities \( k_1 \) and \( k_2 \), respectively, and index sets \( I_1 \subseteq \{1, \ldots, k_1\} \), \( I_2 \subseteq \{1, \ldots, k_2\} \) of the same cardinality, we define

\[
\text{Join}_I(R_1, R_2) := \{ \bar{x}_{K_1 \setminus I_1} \bar{x}_{K_2 \setminus I_2} \mid \exists \bar{x} \in R_1, \bar{y} \in R_2: \bar{x}_{I_1} = \bar{y}_{I_2} \}
\]
The arity of $\text{Join}_I(R_1, R_2)$ is $k_1 + k_2 - |I_1|$. The operation is implemented analogously to the case of binary relations. We proceed in two steps. The first step constructs a transducer according to the following rule:

If the transducer recognizing $R_1$ has a transition $(q, \vec{a}, q')$, the transducer recognizing $R_2$ has a transition $(r, \vec{b}, r')$, and $\vec{a}_I = \vec{b}_I$, then add to the transducer for $\text{Join}_I(R_1, R_2)$ a transition $([q, r], \vec{a}_K \cdot \vec{b}_K, [q', r'])$.

In the second step, we compute the PAD-closure of the result. The generalization of the $\text{Pre}$ and $\text{Post}$ operations is analogous.

**Exercises**

**Exercise 59** As we have seen, the application of the $\text{Post}$, $\text{Pre}$ operations to transducers requires to compute the padding closure in order to guarantee that the resulting automaton accepts either all or none of the encodings of a object. The padding closure has been defined for encodings where padding occurs on the right, i.e., if $w$ encodes an object, then so does $w \#^k$ for every $k \geq 0$. However, in some natural encodings, like the most-significant-bit-first encoding of natural numbers, padding occurs on the left. Give an algorithm for calculating the padding closure of a transducer when padding occurs on the left.

**Exercise 60** Let $A$ be an NFA over the alphabet $\Sigma$.

- Show how to construct a transducer $T$ over the alphabet $\Sigma \times \Sigma$ such that $(w, v) \in L(T)$ iff $wv \in L(A)$ and $|w| = |v|$.

- Give an algorithm that accepts an NFA $A$ as input and returns an NFA $A/2$ such that $L(A/2) = \{w \in \Sigma^* | \exists v \in \Sigma^* : wv \in L(A) \land |w| = |v|\}$.

**Exercise 61** In phone dials letters are mapped into digits as follows:

- ABC $\mapsto$ 2
- DEF $\mapsto$ 3
- GHI $\mapsto$ 4
- JKL $\mapsto$ 5
- MNO $\mapsto$ 6
- PQRS $\mapsto$ 7
- TUV $\mapsto$ 8
- WXYZ $\mapsto$ 9

This map can be used to assign a telephone number to a given word. For instance, the number for AUTOMATON is 288662866.

Consider the problem of, given a telephone number, finding the set of English words that are mapped into it. For instance, the set of words mapping to 233 contains at least ADD, BED, and BEE. Assume a DFA $N$ over the alphabet $\{A, \ldots, Z\}$ recognizing the set of all English words is given. Show how to construct a nondeterministic transducer over $\Sigma \times \Sigma$ for $\Sigma = \{A, \ldots, Z, 0, \ldots, 9\}$ recognizing the set of pairs $(n, w)$ such that $n \in \{0, \ldots, 9\}^*$ and $w \in \{A, \ldots, Z\}^*$ and $w$ is mapped to $n$. 
Exercise 62  We have defined transducers as NFAs whose transitions are labeled by pairs of symbols \((a, b) \in \Sigma \times \Sigma\). With this definition transducers can only accept pairs of words \((a_0a_1 \ldots a_l, b_0b_1 \ldots b_l)\) of the same length.

An \(\varepsilon\)-transducer is a NFA whose transitions are labeled by elements of \((\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})\). An \(\varepsilon\)-transducer accepts a pair \((w, w')\) of words if it has a run \(q_0 \xrightarrow{(a_0,b_0)} q_1 \xrightarrow{(a_1,b_1)} \ldots \xrightarrow{(a_n,b_n)} q_n\) with \(a_i, b_i \in \Sigma \cup \{\varepsilon\}\) such that \(w = a_0a_1 \ldots a_n\) and \(w' = b_0b_1 \ldots b_n\). Note that \(|w| \leq n\) and \(|w'| \leq n\). The relation accepted by the \(\varepsilon\)-transducer \(T\) is denoted by \(L(T)\).

1. Construct \(\varepsilon\)-transducers \(T_1, T_2\) recognizing the relations \(R_1 = \{(a^n b^m, c^{2n}) \mid n, m \geq 0\}\), and \(R_2 = \{(a^n b^m, c^{2m}) \mid n, m \geq 0\}\).

2. Apply \(\text{IntersNFA}\) to \(T_1\) and \(T_2\). Which is the language recognized by the resulting \(\varepsilon\)-transducer?

3. Show that no \(\varepsilon\)-transducer recognizes \(R_1 \cap R_2\).

Exercise 63  Transducers can be used to capture the behaviour of simple programs. Let \(P\) be the following program:

\begin{verbatim}
1  x ← ?
2  write x
3  while true do
4    do
5      read y
6    until x = y
7  if eof then
8    write y
9  end
10  do
11    x ← x − 1
12    or
13    y ← y + 1
14  until x ≠ y
\end{verbatim}

\(P\) communicates with the environment through the boolean variables \(x\) and \(y\), both with 0 as initial value. Let \([i, x, y]\) denote the state of \(P\) in which the next instruction to be executed is the one at line \(i\), and the values of \(x\) and \(y\) are \(x\) and \(y\), respectively. The initial state of \(P\) is \([1, 0, 0]\). By executing the first instruction \(P\) moves nondeterministically to one of the states \([2, 0, 0]\) and \([2, 1, 0]\); no input symbol is read and no output symbol is written, and so the transition relation
6.4. RELATIONS OF HIGHER ARITY

\( \delta \) for \( P \) contains the transitions \(([1, 0, 0], (e, e), [2, 0, 0]) \) and \(([1, 0, 0], (e, e), [2, 1, 0]) \). Similarly, by executing its second instruction, the program \( P \) moves from \([2, 1, 0]\) to \([3, 1, 0]\) while reading nothing and writing 1. Hence, \( \delta \) also contains the transition rule \(([2, 1, 0], (e, 1), [3, 1, 0]) \).

1. Draw an \( \varepsilon \)-transducer modelling the behaviour of \( P \).

2. Can an overflow error occur?

3. What are the possible values of \( x \) and \( y \) upon termination, i.e. upon reaching \textbf{end}?

4. Is there an execution during which \( P \) reads 101 and writes 01?

5. Let \( I \) and \( O \) be regular sets of inputs and outputs, respectively. Think of \( O \) as a set of dangerous outputs that we want to avoid. We wish to prove that the inputs from \( I \) are safe, i.e. that none of the dangerous outputs can occur. Describe an algorithm that decides, given \( I \) and \( O \), whether there are \( i \in I \) and \( o \in O \) such that \((i, o)\) is accepted.

**Exercise 64** (Inspired by a paper by Galwani at POPL’11.) Consider transducers whose transitions are labeled by elements of \((\Sigma \cup \{\varepsilon\}) \times (\Sigma^* \cup \{\varepsilon\})\). Intuitively, at each transition these transducers read one letter or no letter, an write a string of arbitrary length. These transducers can be used to perform operations on strings like, for instance, capitalizing all the words in the string: if the transducer reads, say, "singing in the rain", it writes "Singing In The Rain". Give transducers for the following operations, each of which is informally defined by means of two examples. In each example, if the transducer reads the string on the left then it writes the string on the right.
CHAPTER 6. OPERATIONS ON RELATIONS: IMPLEMENTATIONS
Chapter 7

Finite Universes

In Chapter 3 we proved that every regular language has a unique minimal DFA. A natural question is whether the operations on languages and relations described in Chapters 4 and 6 can be implemented using minimal DFAs and minimal deterministic transducers as data structure.

The implementations of (the first part of) Chapter 4 accept and return DFAs, but do not preserve minimality: even if the arguments are minimal DFAs, the result may be non-minimal (the only exception was complementation). So, in order to return the minimal DFA for the result an extra minimization operation must be applied. The situation is worse for the projection and join operations of Chapter 6, because they do not even preserve determinacy: the result of projecting a deterministic transducer or joining two of them may be nondeterministic. In order to return a minimal DFA it is necessary to first determinize, at exponential cost in the worst case, and then minimize.

In this chapter we present implementations that directly yield the minimal DFA, with no need for an extra minimization step, for the case in which the universe \( U \) of objects is finite.

When the universe is finite, all objects can be encoded by words of the same length, and this common length is known a priori. For instance, if the universe consists of the numbers in the range \([0..2^{32} - 1]\), its objects can be encoded by words over \( \{0, 1\} \) of length 32. Since all encodings have the same length, padding is not required to represent tuples of objects, and we can assume that each object is encoded by exactly one word. As in Chapter ??, we also assume that each word encodes some object. Operations on objects correspond to operations on languages, but complementation requires some care. If \( X \subset U \) is encoded as a language \( L \subseteq \Sigma^k \) for some number \( k \geq 0 \), then the complement set \( U \setminus X \) is not encoded by \( \overline{L} \) (which contains words of any length) but by \( \overline{L} \cap \Sigma^k \).

7.1 Fixed-length Languages and the Master Automaton

**Definition 7.1** A language \( L \subseteq \Sigma^* \) has length \( n \geq 0 \) if every word of \( L \) has length \( n \). If \( L \) has length \( n \) for some \( n \geq 0 \), then we say that \( L \) is a fixed-length language, or that it has fixed-length.

Some remarks are in order:
According to this definition, the empty language has length \( n \) for every \( n \geq 0 \) (the assertion “every word of \( L \) has length \( n \)” is vacuously true). This is useful, because then the complement of a language of length \( n \) has also length \( n \).

- There are exactly two languages of length 0: the empty language \( \emptyset \), and the language \( \{ \varepsilon \} \) containing only the empty word.

- Every fixed-length language contains only finitely many words, and so it is regular.

The master automaton over an alphabet \( \Sigma \) is a deterministic automaton with an infinite number of states, but no initial state. As in the case of canonical DAs, the states are languages.

For the definition, recall the notion of residual with respect to a letter: given a language \( L \subseteq \Sigma^* \) and \( a \in \Sigma \), its residual with respect to \( a \) is the language \( L^a = \{ w \in \Sigma^* \mid aw \in L \} \). Recall that, in particular, we have \( \emptyset^a = \{ \varepsilon \}^a = \emptyset \). A simple but important observation is that if \( L \) has fixed-length, then so does \( L^a \).

**Definition 7.2** The master automaton over the alphabet \( \Sigma \) is the tuple \( M = (Q_M, \Sigma, \delta_M, F_M) \), where

- \( Q_M \) is is the set of all fixed-length languages over \( \Sigma \);
- \( \delta : Q_M \times \Sigma \to Q_M \) is given by \( \delta(q, a) = L^a \) for every \( q \in Q_M \) and \( a \in \Sigma \);
- \( F_M \) is the singleton set containing the language \( \{ \varepsilon \} \) as only element.

**Example 7.3** Figure 7.1 shows a small fragment of the master automaton for the alphabet \( \Sigma = \{ a, b \} \). Notice that \( M \) is almost acyclic. More precisely, the only cycles of \( M \) are the self-loops corresponding to \( \delta_M(\emptyset, a) = \emptyset \) for every \( a \in \Sigma \).

The following proposition was already proved in Chapter 3, but with slightly different terminology.

**Proposition 7.4** Let \( L \) be a fixed-length language. The language recognized from the state \( L \) of the master automaton is \( L \).

**Proof:** By induction on the length \( n \) of \( L \). If \( n = 0 \), then \( L = \{ \varepsilon \} \) or \( L = \emptyset \), and the result is proved by direct inspection of the master automaton. For \( n > 0 \) we observe that the successors of the initial state \( L \) are the languages \( L^a \) for every \( a \in \Sigma \). Since, by induction hypothesis, the state \( L^a \) recognizes the language \( L^a \), the state \( L \) recognizes the language \( L \).

By this proposition, we can look at the master automaton as a structure containing DFAs recognizing all the fixed-length languages. To make this precise, each fixed-length language \( L \) determines a DFA \( A_L = (Q_L, \Sigma, \delta_L, q_0, F_L) \) as follows: \( Q_L \) is the set of states of the master automaton reachable from the state \( L \); \( q_0 \) is the state \( L \); \( \delta_L \) is the projection of \( \delta_M \) onto \( Q_L \); and \( F_L = F_M \). It is easy to show that \( A_L \) is the minimal DFAs recognizing \( L \).
Proposition 7.5  For every fixed-language $L$, the automaton $A_L$ is the minimal DFA recognizing $L$.

Proof: By definition, distinct states of the master automaton are distinct languages. By Proposition 7.4, distinct states of $A_L$ recognize distinct languages. By Corollary 3.8 (a DFA is minimal if and only if distinct states recognized different languages) $A_L$ is minimal.

7.2  A Data Structure for Fixed-length Languages

Proposition 7.5 allows us to define a data structure for representing finite sets of fixed-length languages, all of them of the same length. Loosely speaking, the structure representing the languages $\mathcal{L} = \{L_1, \ldots, L_n\}$ is the fragment of the master automaton containing the states recognizing $L_1, \ldots, L_n$ and their descendants. It is a DFA with multiple initial states, and for this reason we call it the multi-DFA for $\mathcal{L}$. Formally:

Definition 7.6  Let $\mathcal{L} = \{L_1, \ldots, L_n\}$ be a set of languages of the same length over the same alphabet $\Sigma$. The multi-DFA $A_\mathcal{L}$ is the tuple $A_\mathcal{L} = (Q_\mathcal{L}, \Sigma, \delta_\mathcal{L}, Q_{0_\mathcal{L}}, F_\mathcal{L})$, where $Q_\mathcal{L}$ is the set of states of the master automaton reachable from at least one of the states $L_1, \ldots, L_n$; $Q_{0_\mathcal{L}} = \{L_1, \ldots, L_n\}$; $\delta_\mathcal{L}$ is the projection of $\delta_M$ onto $Q_\mathcal{L}$; and $F_\mathcal{L} = F_M$. 

Figure 7.1: A fragment of the master automaton for the alphabet $\{a, b\}$
CHAPTER 7. FINITE UNIVERSES

Figure 7.2: The multi-DFA for \( \{L_1, L_2, L_3\} \) with \( L_1 = \{aa, ba\} \), \( L_2 = \{aa, ba, bb\} \), and \( L_3 = \{ab, bb\} \).

Example 7.7 Figure 7.2 shows (a DFA isomorphic to) the multi-DFA for the set \( \{L_1, L_2, L_3\} \), where \( L_1 = \{aa, ba\} \), \( L_2 = \{aa, ba, bb\} \), and \( L_3 = \{ab, bb\} \). For clarity the state for the empty language has been omitted, as well as the transitions leading to it.

In order to manipulate multi-DFA we represent them as a table of nodes. Assume \( \Sigma = \{a_1, \ldots, a_m\} \). A node is a pair \( \langle q, s \rangle \), where \( q \) is a state identifier and \( s = (q_1, \ldots, q_m) \) is the successor tuple of the node. Along the chapter we denote the state identifiers of the state for the languages \( \emptyset \) and \( \{\epsilon\} \) by \( q_\emptyset \) and \( q_\epsilon \), respectively.

The multi-DFA is represented by a table containing a node for each state, with the exception of the nodes \( q_\emptyset \) and \( q_\epsilon \). The table for the multi-DFA of Figure 7.2, where state identifiers are numbers, is

<table>
<thead>
<tr>
<th>Ident.</th>
<th>a-succ</th>
<th>b-succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The procedure \( \text{make}(s) \). The algorithms on multi-DFA use a procedure \( \text{make}(s) \) that returns the state of \( T \) having \( s \) as successor tuple, if such a state exists; otherwise, it adds a new node \( \langle q, s \rangle \) to \( T \), where \( q \) is a fresh state identifier (different from all other state identifiers in \( T \)) and returns \( q \). If \( s \) is the tuple all whose components are \( q_\emptyset \), then \( \text{make}(s) \) returns \( q_\emptyset \). The procedure assumes that all the states of the tuple \( s \) (with the exception of \( q_\emptyset \) and \( q_\epsilon \)) appear in \( T \).\(^1\) For instance, if \( T \) is the

\(^1\)Notice that the procedure makes use of the fact that no two states of the table have the same successor tuple.
7.3. OPERATIONS ON FIXED-LENGTH LANGUAGES

All operations assume that the input fixed-length language(s) is (are) given as multi-DFAs represented as a table of nodes. Nodes are pairs of state identifier and successor tuple.

The key to all implementations is the fact that if $L$ is a language of length $n \geq 1$, then $L^a$ is a language of length $n - 1$. This allows to design recursive algorithms that directly compute the result when the inputs are languages of length 0, and reduce the problem of computing the result for languages of length $n \geq 1$ to the same problem for languages of smaller length.

**Fixed-length membership.** The operation is implemented as for DFAs. The complexity is linear in the size of the input.

**Fixed-length union and intersection.** Implementing a boolean operation on multi-DFAs corresponds to possibly extending the multi-DFA, and returning the state corresponding to the result of the operation. This is best explained by means of an example. Consider again the multi-DFA of Figure 7.2. An operation like $\text{Union}(L_1, L_2)$ gets the initial states 5 and 6 as input, and returns the state recognizing $L_1 \cup L_2$; since $L_1 \cup L_2 = L_2$, the operation returns state 6. However, if we take $\text{Intersection}(L_2, L_3)$, then the multi-DFA does not contain any state recognizing it. In this case the operation extends the multi-DFA for $\{L_1, L_2, L_3\}$ to the multi-DFA for $\{L_1, L_2, L_3, L_2 \cup L_3\}$, shown in Figure 7.3, and returns state 8. So $\text{Intersection}(L_2, L_3)$ not only returns a state, but also has a side effect on the multi-DFA underlying the operations.
Given two fixed-length languages $L_1, L_2$ of the same length and a boolean operation $\odot$, we present an algorithm that returns the state of the master automaton recognizing $L_1 \cap L_2$ (the algorithm for $L_1 \cup L_2$ is analogous). The properties

- if $L_1 = \emptyset$ or $L_2 = \emptyset$, then $L_1 \cap L_2 = \emptyset$;
- if $L_1 = \{e\}$ and $L_2 = \{e\}$, then $L_1 \cap L_2 = \{e\}$;
- if $L_1, L_2 \notin \emptyset$, then $(L_1 \cap L_2)^a = L_1^a \cap L_2^a$ for every $a \in \Sigma$;

lead to the recursive algorithm $\text{inter}(q_1, q_2)$ shown in Table 7.1. Assume the states $q_1, q_2$ recognize the languages $L_1, L_2$. The algorithm returns the state identifier $q_{L_1 \cap L_2}$. If $q_1 = q_0$, then $L_1 = \emptyset$, which implies $L_1 \cap L_2 = \emptyset$. So the algorithm returns the state identifier $q_0$. If $q_2 = q_0$, the algorithm also returns $q_0$. If $q_1 = q_e = q_2$, the algorithm returns $q_e$. This deals with all the cases in which $q_1, q_2 \in \{q_0, q_e\}$ (and some more, which does no harm). If $q_1, q_2 \notin \{q_0, q_e\}$, then the algorithm computes the state identifiers $r_1, \ldots, r_m$ recognizing the languages $(L_1 \cap L_2)^{a_1}, \ldots, (L_1 \cap L_2)^{a_m}$, and returns $\text{make}(r_1, \ldots, r_m)$ (creating a new node if no node of $T$ has $(r_1, \ldots, r_m)$ as successor tuple). But how does the algorithm compute the state identifier of $(L_1 \cap L_2)^{a_i}$? By equation (3) above, we have $(L_1 \cap L_2)^{a_i} = L_1^{a_i} \cap L_2^{a_i}$, and so the algorithm computes the state identifier of $L_1^{a_i} \cap L_2^{a_i}$ by a recursive call $\text{inter}(q^{a_i}_1, q^{a_i}_2)$.

The only remaining point is the rôle of the table $G$. The algorithm uses memoization to avoid recomputing the same object. The table $G$ is initially empty. When $\text{inter}(q_1, q_2)$ is computed for the first time, the result is memoized in $G(q_1, q_2)$. In any subsequent call the result is not recomputed, but just read from $G$. For the complexity, let $n_1, n_2$ be the number of states of $T$ reachable from the state $q_1, q_2$. It is easy to see that every call to $\text{inter}$ receives as arguments states reachable from $q_1$ and $q_2$, respectively. So $\text{inter}$ is called with at most $n_1 \cdot n_2$ possible arguments, and the complexity is $O(n_1 \cdot n_2)$.

$\text{inter}(q_1, q_2)$

**Input:** states $q_1, q_2$ recognizing languages of the same length

**Output:** state recognizing $L(q_1) \cap L(q_2)$

1. if $G(q_1, q_2)$ is not empty then return $G(q_1, q_2)$
2. if $q_1 = q_0$ or $q_2 = q_0$ then return $q_0$
3. else if $q_1 = q_e$ and $q_2 = q_e$ then return $q_e$
4. else /* $q_1, q_2 \notin \{q_0, q_e\}$ */
5. for all $i = 1, \ldots, m$ do $r_i \leftarrow \text{inter}(q^{a_i}_1, q^{a_i}_2)$
6. $G(q_1, q_2) \leftarrow \text{make}(r_1, \ldots, r_m)$
7. return $G(q_1, q_2)$

Table 7.1: Algorithm $\text{inter}$

Algorithm $\text{inter}$ is generic: in order to obtain an algorithm for another binary operator it suffices to change lines 2 and 3. If we are only interested in intersection, then we can easily gain a more
efficient version. For instance, we know that $\text{inter}(q_1, q_2)$ and $\text{inter}(q_2, q_1)$ return the same state, and so we can improve line 1 by checking not only if $G(q_1, q_2)$ is nonempty, but also if $G(q_2, q_1)$ is. Also, $\text{inter}(q, q)$ always returns $q$, no need to compute anything either.

**Example 7.8** Consider the multi-DFA at the top of Figure 7.4, but without the blue states. State 0, accepting the empty language, is again not shown. The tree at the bottom of the figure graphically describes the run of $\text{inter}(12, 13)$ (that is, we compute the node for the intersection of the languages recognized from states 12 and 13). A node $q, q' \mapsto q''$ of the tree stands for a recursive call to $\text{inter}$ with arguments $q$ and $q'$ that returns $q''$. For instance, the node 2,4 $\mapsto$ 2 indicates that $\text{inter}$ is called with arguments 2 and 4 and the call returns state 2. Let us see why is this so. The call $\text{inter}(2, 4)$ produces two recursive calls, first $\text{inter}(1, 1)$ (the $a$-successors of 2 and 4), and then $\text{inter}(0, 1)$. The first call returns 1, and the second 0. Therefore $\text{inter}(2, 4)$ returns a state with 1 as $a$-successor and 0 as $b$-successor. Since this state already exists (it is state 2), $\text{inter}(2, 4)$ returns 2. On the other hand, $\text{inter}(9,10)$ creates and returns a new state: its two “children calls” return 5 and 6, and so a new state with state 5 and 6 as $a$- and $b$-successors must be created.

Pink nodes correspond to calls that have already been computed, and for which $\text{inter}$ just looks up the result in $G$. Green nodes correspond to calls that are computed by $\text{inter}$, but not by the more efficient version. For instance, the result of $\text{inter}(4,4)$ at the bottom right can be returned immediately.

**Fixed-length complement.** Recall that if a set $X \subseteq U$ is encoded by a language $L$ of length $n$, then the set $U \setminus X$ is encoded by the fixed-length complement $\Sigma_n \setminus L$, which we denote by $L^n$. Since the empty language has all lengths, we have e.g. $\emptyset^2 = \Sigma^2$, but $\emptyset^3 = \Sigma^3$ and $\emptyset^0 = \Sigma^0 = \{\epsilon\}$.

Given the state of the master automaton recognizing $L$, we compute the state recognizing $L^n$ with the help of these properties:

- If $L$ has length 0 and $L = \emptyset$ then $L^n = \{\epsilon\}$.
- If $L$ has length 0 and $L = \{\epsilon\}$, then $L^n = \emptyset$.
- If $L$ has length $n \geq 1$, then $(L^n)^a = L_{(a-1)}^n$.

(Observe that $w \in (L)^a$ iff $aw \notin L$ iff $w \notin L^a$ iff $w \in L^n$.)

We obtain the algorithm of Table 7.9. If the master automaton has $n$ states reachable from $q$, then the operation has complexity $O(n)$.

**Example 7.9** Consider again the multi-DFA at the top of Figure 7.5 without the blue states. The tree of recursive calls at the bottom of the figure graphically describes the run of $\text{comp}(4, 12)$ (that is, we compute the node for the complement of the language recognized from state 12, which has length 4). For instance, $\text{comp}(1,2)$ generates two recursive calls, first $\text{comp}(0,1)$ (the $a$-successor of 2), and then $\text{comp}(0,0)$. The calls returns 0 and 1, respectively, and so $\text{comp}(1,2)$ returns 3. Observe
Figure 7.4: An execution of $\text{inter}$. 
7.3. OPERATIONS ON FIXED-LENGTH LANGUAGES

\( \text{comp}(n, q) \)

**Input:** length \( n \), state \( q \) of length \( n \)

**Output:** state recognizing \( L(q) \)

1. if \( G(n, q) \) is not empty then return \( G(q) \)
2. if \( n = 0 \) and \( q = q_0 \) then return \( q_\epsilon \)
3. else if \( n = 0 \) and \( q = q_\epsilon \) then return \( q_0 \)
4. else /* \( n \geq 1 \) */
5. for all \( i = 1, \ldots, m \) do \( r_i \leftarrow \text{comp}(n - 1, q^a) \)
6. \( G(q) \leftarrow \text{make}(r_1, \ldots, r_m) \)
7. return \( G(n, q) \)

Table 7.2: Algorithm \( \text{comp} \)

how the call \( \text{comp}(2, 0) \) returns 7, the state accepting \( \{a, b\}^2 \). Pink nodes correspond again to calls for which \( \text{comp} \) just looks up the result in \( G \). Green nodes correspond to calls whose result is directly computed by a more efficient version of \( \text{comp} \) that applies the following rule: if \( \text{comp}(i, j) \) returns \( k \), then \( \text{comp}(i, k) \) returns \( j \).

**Fixed-length emptiness.** A fixed-language language is empty if and only if the node representing it has \( q_\emptyset \) as state identifier, and so emptiness can be checked in constant time.

**Fixed-length universality.** A language \( L \) of length \( n \) is fixed-length universal if \( L = \Sigma^n \). The universality of a language of length \( n \) recognized by a state \( q \) can be checked in time \( O(n) \). It suffices to check for all states reachable from \( q \), with the exception of \( q_\emptyset \), that no transition leaving them leads to \( q_\emptyset \). More systematically, we use the properties

- if \( L = \emptyset \), then \( L \) is not universal;
- if \( L = \{\epsilon\} \), then \( L \) is universal;
- if \( \emptyset \neq L \neq \{\epsilon\} \), then \( L \) is universal iff \( L^a \) is universal for every \( a \in \Sigma \);

that lead to the algorithm of Table 7.3.

**Fixed-length inclusion.** Given two languages \( L_1, L_2 \subseteq \Sigma^n \), in order to check \( L_1 \subseteq L_2 \) we compute \( L_1 \cap L_2 \) and check whether it is equal to \( L_1 \) using the equality check shown next. The complexity is dominated by the complexity of computing the intersection.
Figure 7.5: An execution of $\text{comp}$.
7.4. DETERMINIZATION AND MINIMIZATION

univ(q)
Input: state q
Output: true if L(q) is fixed-length universal,
false otherwise
1 if G(q) is not empty then return G(q)
2 if q = q₀ then return false
3 else if q = q₁ then return true
4 else /
∗ q₁, q₀ and q₁, q₂ *
/ 
5 G(q) ← and(univ(q₁₁), . . . , univ(q₁w))
6 return G(q)

Table 7.3: Algorithm univ

Fixed-length equality. Since the minimal DFA recognizing a language is unique, two languages are equal if and only if the nodes representing them have the same state identifier, leading to a constant time algorithm. This solution, however, assumes that the two input nodes come from the same table. If they come from two different tables T₁, T₂, then, since state identifiers can be assigned in both tables in different ways, it is necessary to check if the DFA rooted at the states q₁ and q₂ are isomorphic. This is done by algorithm eq2 of Table 7.4, which assumes that q₁ belongs to a table T₁, and that both tables assign state identifiers q₀₁ and q₀₂ to the empty language.

eq2(q₁, q₂)
Input: states q₁, q₂ of different tables
Output: true if L(q₁) = L(q₂), false otherwise
1 if G(q₁, q₂) is not empty then return G(q₁, q₂)
2 if q₁ = q₀₁ and q₂ = q₀₂ then G(q₁, q₂) ← true
3 else if q₁ = q₀₁ and q₂ ≠ q₀₂ then G(q₁, q₂) ← false
4 else if q₁ ≠ q₀₁ and q₂ = q₀₂ then G(q₁, q₂) ← false
5 else /
∗ q₁ ≠ q₀₁ and q₂ ≠ q₀₂ *
/ 
6 G(q₁, q₂) ← and(eq(q₁₁, q₂₁), . . . , eq(q₁w, q₂w))
7 return G(q₁, q₂)

Table 7.4: Algorithm eq2

7.4 Determinization and Minimization

Let L be a fixed-length language, and let A = (Q, Σ, δ, q₀, F) be a NFA recognizing L. The algorithm det&min(A) shown in Table 7.5 returns the state of the master automaton recognizing L. In other
words, \( \text{det}\&\text{min}(A) \) simultaneously determinizes and minimizes \( A \).

The algorithm actually solves a more general problem. Given a set \( S \) of states of \( A \), all recognizing languages of the same length, the language \( L(S) = \bigcup_{q \in S} L(q) \) has also this common length. The heart of the algorithm is a procedure \( \text{state}(S) \) that returns the state recognizing \( L(S) \). Since \( L = L(\{q_0\}) \), \( \text{det}\&\text{Min}(A) \) just calls \( \text{state}(\{q_0\}) \).

We make the assumption that for every state \( q \) of \( A \) there is a path leading from \( q \) to some final state. This assumption can be enforced by suitable preprocessing, but usually it is not necessary; in applications, NFAs for fixed-length languages usually satisfy the property by construction. With this assumption, \( L(S) \) satisfies:

- if \( S = \emptyset \) then \( L(S) = \emptyset \);
- if \( S \cap F \neq \emptyset \) then \( L(S) = \{\varepsilon\} \)
  (since the states of \( S \) recognize fixed-length languages, the states of \( F \) necessarily recognize \( \{\varepsilon\} \); since all the states of \( S \) recognize languages of the same length and \( S \cap F \neq \emptyset \), we have \( L(S) = \{\varepsilon\} \));
- if \( S \neq \emptyset \) and \( S \cap F = \emptyset \), then \( L(S) = \bigcup_{i=1}^{n} a_i \cdot L(S_i) \), where \( S_i = \delta(S, a_i) \).

These properties lead to the recursive algorithm of Table 7.5. The procedure \( \text{state}(S) \) uses a table \( G \) of results, initially empty. When \( \text{state}(S) \) is computed for the first time, the result is memoized in \( G(S) \), and any subsequent call directly reads the result from \( G \). The algorithm has exponential complexity, because, in the worst case, it may call \( \text{state}(S) \) for every set \( S \subseteq Q \). To show that an exponential blowup is unavoidable, consider the family \( \{L_n\}_{n \geq 0} \) of languages, where \( L_n = \{ww' | w, w' \in \{0, 1\}^n \text{ and } w \neq w'\} \). While \( L_n \) can be recognized by an NFA of size \( O(n^2) \), its minimal DFA has \( O(2^n) \) states: for every \( u, v \in \Sigma^n \) if \( u \neq v \) then \( L_n^u \neq L_n^v \), because \( v \in L_n^u \) but \( v \notin L_n^v \).

**Example 7.10** Figure 7.6 shows a NFA (upper left) and the result of applying \( \text{det}\&\text{min} \) to it. The run of \( \text{det}\&\text{min} \) is shown at the bottom of the figure, where, for the sake of readability, sets of states are written without the usual parenthesis (e.g. \( \{\beta, \gamma\} \) instead of \( \{\beta\}, \{\gamma\} \)). Observe, for instance, that the algorithm assigns to \( \{\gamma\} \) the same node as to \( \{\beta, \gamma\} \), because both have the states 2 and 3 as \( a \)-successor and \( b \)-successor, respectively.

### 7.5 Operations on Fixed-length Relations

Fixed-length relations can be manipulated very similarly to fixed-length languages. Boolean operations are as for fixed-length languages. The projection, join, \( \text{pre} \), and \( \text{post} \) operations can be however implemented more efficiently as in Chapter 6.

We start with an observation on encodings. In Chapter 6 we assumed that if an element of \( X \) is encoded by \( w \in \Sigma^* \), then it is also encoded by \( w\# \), where \# is the padding letter. This ensures that every pair \( (x, y) \in X \times X \) has an encoding \( (w_x, w_y) \) such that \( w_x \) and \( w_y \) have the same length.
Figure 7.6: Run of $det\&min$ on an NFA for a fixed-length language
det&min(A)

**Input:** NFA $A = (Q, \Sigma, \delta, q_0, F)$

**Output:** master state recognizing $L(A)$

1. return state($q_0$)

$state(S)$

**Input:** set $S \subseteq Q$ recognizing languages of the same length

**Output:** state recognizing $L(S)$

1. if $G(S)$ is not empty then return $G(S)$
2. else if $S = \emptyset$ then return $q_0$
3. else if $S \cap F \neq \emptyset$ then return $q_e$
4. else /* $S \neq \emptyset$ and $S \cap F = \emptyset$ */
5. for all $i = 1, \ldots, m$ do $S_i \leftarrow \delta(S, a_i)$
6. $G(S) \leftarrow$ make($state(S_1), \ldots, state(S_m)$);
7. return $G(S)$

Table 7.5: Algorithm det&min.

Since in the fixed-length case all shortest encodings have the same length, the padding symbol is no longer necessary. So in this section we assume that each word or pair has exactly one encoding.

The basic definitions on fixed-length languages extend easily to fixed-length relations. A word relation $R \subseteq \Sigma^* \times \Sigma^*$ has length $n \geq 0$ if for all pairs $(w_1, w_2) \in R$ the words $w_1$ and $w_2$ have length $n$. If $R$ has length $n$ for some $n \geq 0$, then we say that $R$ has fixed-length.

Recall that a transducer $T$ accepts a pair $(w_1, w_2) \in \Sigma^* \times \Sigma^*$ if $w_1 = a_1 \ldots a_n$, $w_2 = b_1 \ldots b_n$, and $T$ accepts the word $(a_1, b_1) \ldots (a_n, b_n) \in \Sigma^* \times \Sigma^*$. A fixed-length transducer accepts a relation $R \subseteq X \times X$ if it recognizes the word relation $\{(w_x, w_y) \mid (x, y) \in R\}$.

Given a language $R \subseteq \Sigma^* \times \Sigma^*$ and $a, b \in \Sigma$, we define $R^{[a,b]} = \{(w_1, w_2) \in \Sigma^* \times \Sigma^* \mid (aw_1, bw_2) \in R\}$. Notice that in particular, $\emptyset^{[a,b]} = \emptyset$, and that if $R$ has fixed-length, then so does $R^{[a,b]}$. The master transducer over the alphabet $\Sigma$ is the tuple $MT = (Q_M, \Sigma \times \Sigma, \delta_M, F_M)$, where $Q_M$ is is the set of all fixed-length relations, $F_M = \{(e, e)\}$, and $\delta_M: Q_M \times (\Sigma \times \Sigma) \rightarrow Q_M$ is given by $\delta_M(R, [a, b]) = R^{[a,b]}$ for every $q \in Q_M$ and $a, b \in \Sigma$. As in the language case, the minimal deterministic transducer recognizing a fixed-length relation $R$ is the fragment of the master transducer containing the states reachable from $R$.

Like minimal DFA, minimal deterministic transducers are represented as tables of nodes. However, a remark is in order: since a state of a deterministic transducer has $|\Sigma|^2$ successors, one for each letter of $\Sigma \times \Sigma$, a row of the table has $|\Sigma|^2$ entries, too large when the table is only sparsely filled. Sparse transducers over $\Sigma \times \Sigma$ are better encoded as NFAs over $\Sigma$ by introducing auxiliary states: a transition $q \rightarrow^{[a,b]} q'$ of the transducer is “simulated” by two transitions $q \rightarrow^a r \rightarrow^b q'$, where $r$ is an auxiliary state with exactly one input and one output transition.
7.5. OPERATIONS ON FIXED-LENGTH RELATIONS

Fixed-length projection  The implementation of the projection operation of Chapter 6 may yield a nondeterministic transducer, even if the initial transducer is deterministic. So we need a different implementation. We observe that projection can be reduced to \textit{pre} or \textit{post}: the projection of a binary relation \( R \) onto its first component is equal to \( \text{pre}_R(\Sigma^*) \), and the projection onto the second component to \( \text{post}_R(\Sigma^*) \). So we defer dealing with projection until the implementation of \( \text{pre} \) and \( \text{post} \) have been discussed.

Fixed-length join.  We give a recursive definition of \( R_1 \circ R_2 \). Let \([a, b] \in R \Longleftrightarrow (aw_1, bw_2) \in R \). We have the following properties:

\[
\begin{align*}
\bullet & \quad \emptyset \circ R = R \circ \emptyset = \emptyset; \\
\bullet & \quad \{ [\varepsilon, \varepsilon] \} \circ \{ [\varepsilon, \varepsilon] \} = \{ [\varepsilon, \varepsilon] \}; \\
\bullet & \quad R_1 \circ R_2 = \bigcup_{a, b, c \in \Sigma} [a, b] \cdot \left( R_1^{[a, c]} \circ R_2^{[c, b]} \right);
\end{align*}
\]

which lead to the algorithm of Figure 7.7, where \textit{union} is defined similarly to \textit{inter}. The complexity is exponential in the worst case: if \( t(n) \) denotes the worst-case complexity for two states of length \( n \), then we have \( t(n) = O(t(n - 1)^2) \), because \textit{union} has quadratic worst-case complexity. This exponential blowup is unavoidable. We prove it later for the projection operation (see Example 7.11), which is a special case of \textit{pre} and \textit{post}, which in turn can be seen as variants of join.

\textit{join}(r_1, r_2)

\textbf{Input:} states \( r_1, r_2 \) of transducer table

\textbf{Output:} state recognizing \( L(r_1) \circ L(r_2) \)

1. \text{if} \( G(r_1, r_2) \) is not empty \text{then return} \( G(r_1, r_2) \)
2. \text{if} \( r_1 = q_0 \) \text{or} \( r_2 = q_0 \) \text{then return} \( q_0 \)
3. \text{else if} \( r_1 = q_\varepsilon \) \text{and} \( r_2 = q_\varepsilon \) \text{then return} \( q_\varepsilon \)
4. \text{else} /* q_0 \neq r_1 \neq q_\varepsilon \text{and} q_0 \neq r_2 \neq q_\varepsilon */
5. \text{for all} \( (a_i, a_j) \in \Sigma \times \Sigma \) \text{do}
6. \quad \text{let} \( r_{i, j} \leftarrow \text{union} \left( \text{join} \left( r_1^{[a_i, a_1]}, r_2^{[a_1, a_j]} \right), \ldots, \text{join} \left( r_1^{[a_i, a_m]}, r_2^{[a_m, a_j]} \right) \right) \)
7. \quad G(r_1, r_2) = \text{make}(r_{1, 1}, \ldots, r_{m, m})
8. \text{return} \( G(r_1, r_2) \)

Figure 7.7: Algorithm \textit{join}

Fixed-length \textit{pre} and \textit{post}.  Recall that in the fixed-length case we do not need any padding symbol. Then, given a fixed-length language \( L \) and a relation \( R \), \textit{pre}_R(L) \) admits an inductive definition that we now derive. We have the following properties:

• if $R = \emptyset$ or $L = \emptyset$, then $\text{pre}_R(L) = \emptyset$;

• if $R = \{[\epsilon, \epsilon]\}$ and $L = \{\epsilon\}$, then $\text{pre}_R(L) = \{\epsilon\}$;

• if $\emptyset \neq R \neq \{[\epsilon, \epsilon]\}$ and $\emptyset \neq L \neq \{\epsilon\}$, then $\text{pre}_R(L) = \bigcup_{a,b \in \Sigma} a \cdot \text{pre}_{R[a,b]}(L^b)$,

where $R[a,b] = \{w \in (\Sigma \times \Sigma)^* | [a, b]w \in R\}$.

The first two properties are obvious. For the last one, observe that all pairs of $R$ have length at least one, and so every word of $\text{pre}_R(L)$ also has length at least one. Now, given an arbitrary word $aw_1 \in \Sigma^*$, we have

$$aw_1 \in \text{pre}_R(L) \iff \exists b w_2 \in L: [aw_1, bw_2] \in R \iff \exists b \in \Sigma \exists w_2 \in L^b: [w_1, w_2] \in R[a,b] \iff \exists b \in \Sigma: w_1 \in \text{pre}_{R[a,b]}(L^b) \iff aw_1 \in \bigcup_{b \in \Sigma} a \cdot \text{pre}_{R[a,b]}(L^b)$$

and so $\text{pre}_R(L) = \bigcup_{a,b \in \Sigma} a \cdot \text{pre}_{R[a,b]}(L^b)$ These properties lead to the recursive algorithm of Table 7.5, which accepts as inputs a state of the transducer table for a relation $R$ and a state of the automaton table for a language $L$, and returns the state of the automaton table recognizing $\text{pre}_R(L)$. The transducer table is not changed by the algorithm.

\begin{verbatim}
pre(r, q)
\textbf{Input:} state $r$ of transducer table, state $q$ of automaton table
\textbf{Output:} state recognizing $\text{pre}_{L(r)}(L(q))$
1 \hspace{1em} if $G(r, q)$ is not empty then return $G(r, q)$
2 \hspace{1em} if $r = r_0$ or $q = q_0$ then return $q_0$
3 \hspace{1em} else if $r = r_\epsilon$ and $q = q_\epsilon$ then return $q_\epsilon$
4 \hspace{1em} else
5 \hspace{1em} for all $a_i \in \Sigma$
6 \hspace{2em} $q'_i \leftarrow \text{union}(\text{pre}(r[a_i, a_1], q^{a_1}), \ldots, \text{pre}(r[a_i, a_m], q^{a_m}))$
7 \hspace{1em} $G(q, r) \leftarrow \text{make}(q'_1, \ldots, q'_m)$
8 \hspace{1em} return $G(q, r)$
\end{verbatim}

Table 7.6: Algorithm $pre$.

As promised, we can now give an implementation of the operation that projects a relation $R$ onto its first component. It suffices to give a dedicated algorithm for $\text{pre}_R(\Sigma^*)$, shown in Table 7.5. Algorithm $pro_1$ has exponential worst-case complexity. As in the case of $\text{join}$, the reason is the quadratic blowup introduced by $\text{union}$ when the recursion depth increases by one. The next example shows that projection is inherently exponential.
**7.6. DECISION DIAGRAMS**

**Table 7.7: Algorithm pro₁.**

Example 7.11 Consider the relation $R \subseteq \Sigma^{2n} \times \Sigma^{2n}$ given by

$$R = \{(w_1 x w_2 y w_3, 0^{w_1} 1^{w_2} 10^{w_3}) \mid x \neq y, |w_2| = n \text{ and } |w_1 w_3| = n - 2\}.$$  

That is, $R$ contains all pairs of words of length $2n$ whose first word has a position $i \leq n$ such that the letters at positions $i$ and $i + n$ are distinct, and whose second word contains only 0’s but for two 1’s at the same two positions. It is easy to see that the minimal deterministic transducer for $R$ has $\mathcal{O}(n^2)$ states (intuitively, it memorizes the letter $x$ above the first 1, reads $n - 1$ letters of the form $(z, 0)$, and then reads $(y, 1)$, where $y \neq x$). On the other hand, we have

$$pro₁(R) = \{ww' \mid w, w' \in \Sigma^n \text{ and } w \neq w'\},$$

whose minimal DFA, as shown when discussing $det \& min$, has $\mathcal{O}(2^n)$ states. So any algorithm for projection has $\Omega(2^{\sqrt{n}})$ complexity.  

Slight modifications of this example show that $join$, $pre$, and $post$ are inherently exponential as well.

### 7.6 Decision Diagrams

**Binary Decision Diagrams**, BDDs for short, are a very popular data structure for the representation and manipulation of boolean functions. In this section we show that they can be seen as minimal automata of a certain kind.

Given a boolean function $f(x_1, \ldots, x_n) : \{0, 1\}^n \rightarrow \{0, 1\}$, let $L_f$ denote the set of strings $b_1 b_2 \ldots b_n \in \{0, 1\}^n$ such that $f(b_1, \ldots, b_n) = 1$. The minimal DFA recognizing $L_f$ is very similar to the BDD representing $f$, but not completely equal. We modify the constructions of the last section to obtain an exact match.

Consider the following minimal DFA for a language of length four:
Its language can be described as follows: after reading an $a$, accept any word of length three; after reading $ba$, accept any word of length 2; after reading $bb$, accept any two-letter word whose last letter is a $b$. Following this description, the language can also be more compactly described by an automaton with regular expressions as transitions:

The converse direction also works: the minimal DFA can be recovered from the zip-automaton by “reversing” the rule. This already allows us to use zip-automata as a data structure for fixed-length languages, but only through conversion to minimal DFAs: to compute an operation using zip-automata, expand them to minimal DFAs, conduct the operation, and convert the result back. Section 7.6.2 shows how to do better by directly defining the operations on zip-automata, bypassing the minimal DFAs.

### 7.6.1 Zip-automata and Kernels

A zip over an alphabet $\Sigma$ is a regular expression of the form

$$a\Sigma^n = \underbrace{\Sigma\Sigma\ldots\Sigma}_n$$
(aΣΣΣ...ΣΣ looks a bit like a zip). A zip-NFA is a NFA whose transitions are labelled by zips. A zip-NFA $A \in (Q, \Sigma, \delta, q_0, F)$ is deterministic if for every $q \in Q$ and $a \in \Sigma$ there is exactly one $k \in \mathbb{N}$ such that $\delta(q, a\Sigma^k) \neq \emptyset$.

It is easy to see that a fixed-length language is recognized by a zip-DFA if and only it is a kernel:

**Definition 7.12** A fixed-language $L$ is a kernel if it is empty or if there are $a, b \in \Sigma$ such that $L^a \neq L^b$. The kernel of a fixed-language $L$ is the unique kernel $\langle L \rangle$ satisfying $L = \Sigma^k \langle L \rangle$ for some $k \geq 0$.

The master zip-automaton (we call it just “the master”) has the set of all kernels as states, the kernel $\{\epsilon\}$ as unique final state, and a transition $(K, a\Sigma^k, \langle K^a \rangle)$ for every kernel $K$ and $a \in \Sigma$, where $k$ is equal to the length of $K^a$ minus the length of $\langle K^a \rangle$.

**Example 7.13** Figure 7.8 shows a fragment of the master (compare with Figure 7.1). The languages $\{a, b\}$, $\{aa, ab, ba, bb\}$, and $\{ab, bb\}$ of Figure 7.1 are not kernels, and so they are not states of the master either.

The zip-DFA $A_K$ for a kernel $K$ is the fragment of the master containing the states reachable from $K$. It is easy to see that $A_K$ recognizes $K$. A zip-DFA is minimal if no other zip-DFA for
the same language has fewer states. The following proposition shows that minimal zip-DFAs have very similar properties to minimal DFAs:

**Proposition 7.14** (1) A zip-DFA $A$ that recognizes a kernel is minimal if and only if (i) every state of $A$ recognizes a kernel, and (ii) distinct states of $A$ recognize distinct kernels.

(2) $A_K$ is the unique minimal zip-DFA recognizing $K$.

(3) The result of exhaustively applying the reduction rule to the minimal DFA recognizing a fixed-length language $L$ is the minimal zip-DFA recognizing $\langle L \rangle$.

**Proof:** (1 $\Rightarrow$): For (i), assume $A$ contains a state $q$ such that $L(q)$ is not a kernel. Since $L(A)$ is a kernel, $q$ is neither initial nor final. Let $k$ be the smallest number such that $A$ contains a transition $(q, a^k, q')$ for some letter $a$ and some state $q'$. Then $L(q)^a = \Sigma^k L(q')$, and, since $L(q)$ is not a kernel, $L(q)^a = L(q)b$ for every $b \in \Sigma$. So we have $L(q) = \bigcup_{a \in \Sigma} a^k L(q') = \Sigma^{k+1} L(q')$. Now we perform the following two operations: first, we replace every transition $(q'', b^j, q)$ of $A$ by a transition $(q'', b^{j+k+1}, q')$; then, we remove $q$ and any other state no longer reachable form the initial state (recall that $q$ is neither initial nor final). The resulting zip-DFA recognizes the same language as $A$, and has at least one state less. So $A$ is not minimal.

For (ii), observe that the quotienting operation can be defined for zip-DFAs as for DFAs, allowing us to merge states that recognize the same kernel without changing the language. If two distinct states of $A$ recognize the same kernel then the quotient has fewer states than $A$, and so $A$ is not minimal.

(1 $\Leftarrow$): We show that two zip-DFAs $A$ and $A'$ that satisfy (i) and (ii) and recognize the same language are isomorphic, which, together with (1 $\Rightarrow$), proves that they are minimal. It suffices to prove that if two states $q, q'$ of $A$ and $A'$ satisfy $L(q) = L(q')$, then for every $a \in \Sigma$ the (unique) transitions $(q, a^k, r)$ and $(q', a^k, r')$ satisfy $k = k'$ and $L(r) = L(r')$. Let $L(q) = K = L(q')$. By (1 $\Rightarrow$), both $L(r)$ and $L(r')$ are kernels. But then we necessarily have $L(r) = \langle K' \rangle = L(q')$, because the only solution to the equation $K = a^l K'$, where $l$ and $K'$ are unknowns and $K'$ must be a kernel, is $K' = \langle l \rangle K''$.

(2) $A_K$ recognizes $K$ and it satisfies conditions (a) and (b) of part (1) by definition. So it is a minimal zip-DFA. Uniqueness follows from the proof of (1 $\Leftarrow$).

(3) Let $B$ be a zip-DFA obtained by exhaustively applying the reduction rule to $A$. By (1), it suffices to prove that $B$ satisfies (i) and (ii). For (ii) observe that, since every state of $A$ recognizes a different language, so does every state of $B$ (the reduction rule preserves the recognized languages).

For (i), assume that some state $q$ does not recognize a kernel. Without loss of generality, we can choose $L(q)$ of minimal length, and therefore the target states of all outgoing transitions of $q$ recognize kernels. It follows that all of them necessarily recognize $\langle L(q) \rangle$. Since $B$ contains at most one state recognizing $\langle L(q) \rangle$, all outgoing transitions of $q$ have the same target, and so the reduction rule can be applied to $q$, contradicting the hypothesis that it has been applied exhaustively. 

7.6. DECISION DIAGRAMS

131

Figure 7.9: The multi-zDFA for \(\{L_1, L_2, L_3\}\) with \(L_1 = \{aa\}\), \(L_2 = \{aa, bb\}\), and \(L_3 = \{aa, ab\}\).

7.6.2 Operations on Kernels

We use multi-zip-DFAs to represent sets of fixed-length languages of the same length. A set \(\mathcal{L} = \{L_1, \ldots, L_m\}\) is represented by the states of the master recognizing \(\langle L_1 \rangle, \ldots, \langle L_m \rangle\) and by the common length of \(L_1, \ldots, L_m\). Observe that the states and the length completely determine \(\mathcal{L}\).

Example 7.15 Figure 7.9 shows the multi-zip-DFA for the set \(\{L_1, L_2, L_3\}\) of Example 7.7. Recall that \(L_1 = \{aa, ba\}\), \(L_2 = \{aa, ba, bb\}\), and \(L_3 = \{ab, bb\}\). The multi-zip-DFA is the result of applying the reduction rule to the multi-automaton of Figure 7.2. We represent the set by the multi-zip-DFA and the number 2, the length of \(L_1, L_2, L_3\). Observe that, while \(L_1, L_2\) and \(L_3\) have the same length, \(\langle L_2 \rangle\) has a different length than \(\langle L_1 \rangle\) and \(\langle L_3 \rangle\).

Multi-zip-DFAs are represented as a table of kernodes. A kernode is a triple \(\langle q, l, s \rangle\), where \(q\) is a state identifier, \(l\) is a length, and \(s = (q_1, \ldots, q_m)\) is the successor tuple of the kernode. The table for the multi-zip-DFA of Figure 7.9 is:

<table>
<thead>
<tr>
<th>Ident.</th>
<th>Length</th>
<th>a-succ</th>
<th>b-succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

This example explains the role of the new length field. If we only now that the \(a\)- and \(b\)-successors of, say, state 6 are states 2 and 1, we cannot infer which zips label the transitions from 6 to 2 and from 6 to 1: they could be \(a\) and \(b\), or \(a\Sigma\) and \(b\Sigma\), or \(a\Sigma^n\) and \(b\Sigma^{n+1}\) for any \(n \geq 0\). However, once we know that state 6 accepts a language of length 2, we can deduce the correct labels: since states 2 and 1 accept languages of length 1 and 0, respectively, the labels are \(a\) and \(b\Sigma\).
The procedure \texttt{kmake}(l, s). All algorithms call a procedure \texttt{kmake}(l, s) with the following specification. Let $K_i$ be the kernel recognized by the $i$-th component of $s$. Then \texttt{kmake}(l, s) returns the kernode for $\langle L \rangle$, where $L$ is the unique language of length $l$ such that $\langle L^a \rangle = K_i$ for every $a_i \in \Sigma$.

If $K_i \neq K_j$ for some $i, j$, then \texttt{kmake}(l, s) behaves like \texttt{make}(s): if the current table already contains a kernode $\langle q, l, s \rangle$, then \texttt{kmake}(l, s) returns $q$; and, if no such kernode exists, then \texttt{kmake}(l, s) creates a new kernode $\langle q, l, s \rangle$ with a fresh identifier $q$, and returns $q$.

If $K_1, \ldots, K_m$ are all equal to some kernel $K$, then we have $L = \bigcup_{i=1}^m a_i \Sigma^k K$ for some $k$, and therefore $\langle L \rangle = (\Sigma^{l+1} K) = K$. So \texttt{kmake}(l, i) returns the kernode for $K$. For instance, if $T$ is the table above, then \texttt{kmake}(3, (2, 2)) returns 3, while \texttt{make}(2, 2) creates a new node having 2 as $a$-successor and $b$-successor.

Algorithms. The algorithms for operations on kernels are modifications of the algorithms of the previous section. We show how to modify the algorithms for intersection, complement, and for simultaneous determinization and minimization. In the previous section, the state of the master automaton for a language $L$ was the language $L$ itself, and was obtained by recursively computing the states for $L^a_1, \ldots, L^a_n$ and then applying make. Now, the state of the zip-master automaton for $L$ is $\langle L \rangle$, and can be obtained by recursively computing states for $\langle L^a_1 \rangle, \ldots, \langle L^a_n \rangle$ and applying \texttt{kmake}.

Fixed-length intersection. Given kernels $K_1, K_2$ of languages $L_1, L_2$, we compute the state recognizing $K_1 \cap K_2 \overset{\text{def}}{=} \langle L_1 \cap L_2 \rangle$. We have the obvious property

- if $K_1 = \emptyset$ or $K_2 = \emptyset$, then $K_1 \cap K_2 = \emptyset$.

Assume now $K_1 \neq \emptyset \neq K_2$. If the lengths of $K_1$ and $K_2$ are $l_1, l_2$, then since $\langle \Sigma^k L \rangle = \langle L \rangle$ holds for every $k, L$ we have

$$K_1 \cap K_2 = \begin{cases} \langle \Sigma^{l_1-k} K_1 \cap K_2 \rangle & \text{if } l_1 < l_2 \\ \langle K_1 \cap \Sigma^{l_2-k} K_2 \rangle & \text{if } l_1 > l_2 \\ \langle K_1 \cap K_2 \rangle & \text{if } l_1 = l_2 \end{cases}$$

which allows us to obtain the state for $K_1 \cap K_2$ by computing states for

$$\langle (\Sigma^{l_2-k} K_1 \cap K_2)^a \rangle \quad \langle (K_1 \cap \Sigma^{l_2-k} K_2)^a \rangle \quad \text{or} \quad \langle (K_1 \cap K_2)^a \rangle$$

for every $a \in \Sigma$, and applying \texttt{kmake}. These states can be computed recursively by means of:

- if $l_1 < l_2$ then $\langle (\Sigma^{l_2-k} K_1 \cap K_2)^a \rangle = \langle \Sigma^{l_2-k-1} K_1 \cap K_2^a \rangle = K_1 \cap \langle K_2^a \rangle$;

- if $l_1 > l_2$ then $\langle (K_1 \cap \Sigma^{l_2-k} K_2)^a \rangle = \langle K_1^a \cap \Sigma^{l_1-k-1} K_2 \rangle = \langle K_1^a \rangle \cap K_2$;

- if $l_1 = l_2$ then $\langle (K_1 \cap K_2)^a \rangle = \langle K_1^a \cap K_2^a \rangle = \langle K_1^a \rangle \cap \langle K_2^a \rangle$;

which leads to the algorithm of Table 7.8.

---

\footnote{$\cap$ is well defined because $\langle L_i \rangle = \langle L'_i \rangle$ and $\langle L_2 \rangle = \langle L'_2 \rangle$ implies $\langle L_1 \cap L_2 \rangle = \langle L'_1 \cap L'_2 \rangle$.}
**7.6. DECISION DIAGRAMS**

7.6. DECISION DIAGRAMS

\[ kinter(q_1, q_2) \]

**Input:** states \( q_1, q_2 \) recognizing \( \langle L_1 \rangle, \langle L_2 \rangle \)

**Output:** state recognizing \( \langle L_1 \cap L_2 \rangle \)

1. if \( G(q_1, q_2) \) is not empty then return \( G(q_1, q_2) \)
2. if \( q_1 = q_0 \) or \( q_2 = q_0 \) then return \( q_0 \)
3. if \( q_1 \neq q_0 \) and \( q_2 \neq q_0 \) then
   4. if \( l_1 < l_2 \) /* lengths of the kernodes for \( q_1, q_2 \) */ then
      5. for all \( i = 1, \ldots, m \) do \( r_i \leftarrow kinter(q_1, q_2^a) \)
      6. \( G(q_1, q_2) \leftarrow \text{kmake}(l_2, r_1, \ldots, r_m) \)
   7. else if \( l_1 = l_2 \) then
      8. for all \( i = 1, \ldots, m \) do \( r_i \leftarrow kinter(q_1^a, q_2) \)
      9. \( G(q_1, q_2) \leftarrow \text{kmake}(l_1, r_1, \ldots, r_m) \)
   10. else /* \( l_1 = l_2 \) */
      11. for all \( i = 1, \ldots, m \) do \( r_i \leftarrow kinter(q_1^a, q_2^a) \)
      12. \( G(q_1, q_2) \leftarrow \text{kmake}(l_1, r_1, \ldots, r_m) \)
13. return \( G(q_1, q_2) \)

**Table 7.8: Algorithm kinter**

**Example 7.16** Example 7.8 shows a run of \( \text{inter} \) on the two languages represented by the multi-DFA at the top of Figure 7.4. The multi-zip-DFA for the same languages is shown at the top of Figure 7.10, and the rest of the figure describes the run of \( kinter \) on it. Recall that pink nodes correspond to calls whose result has already been memoized, and need not be executed. The meaning of the green nodes is explained below.

The algorithm can be improved by observing that two further properties hold:

- if \( K_1 = \{ \varepsilon \} \) then \( L_1 \cap L_2 = L_1 \), and so \( K_1 \cap K_2 = K_1 \), and if \( K_2 = \{ \varepsilon \} \) then \( L_1 \cap L_2 = L_2 \), and so \( K_1 \cap K_2 = K_2 \).

These properties imply that \( kinter(q_\varepsilon, q) = q = kinter(q, q_\varepsilon) \) for every state \( q \). So we can improve \( kinter \) by explicitly checking if one of the arguments is \( q_\varepsilon \). The green nodes in Figure 7.10 correspond to calls whose result is immediately returned with the help of this check. Observe how this improvement has a substantial effect, reducing the number of calls from 19 to only 5.

**Fixed-length complement.** Given the kernel \( K \) of a fixed-language languages \( L \) of length \( n \), we wish to compute the master state recognizing \( \langle L^n \rangle \), where \( n \) is the length of \( L \). The subscript \( n \) is only necessary because \( \emptyset \) has all possible lengths, and so \( \emptyset^n = \Sigma^n \neq \Sigma^m = \overline{L}^m \) for \( n \neq m \). Now we have \( \langle \emptyset^n \rangle = \{ \varepsilon \} \) for every \( n \geq 0 \), and so the subscript is not needed anymore. We define the
Figure 7.10: An execution of \textit{kinter}.
operator \( \sim \) on kernels by \( \bar{K} = \langle L \rangle \).\(^3\) We obtain the state for \( \bar{K} \) by recursively computing states for \( \langle \bar{K}^a \rangle \) by means of the properties

- if \( K = \emptyset \) then \( \bar{K} = \{e\} \), and if \( K = \{e\} \), then \( \bar{K} = \emptyset \);
- if \( \emptyset \neq K \neq \{e\} \) then \( \langle \bar{K}^a \rangle = \bar{K}^a \);

which lead to the algorithm of Table 7.9.

\[
\begin{align*}
kcomp(q) \\
\textbf{Input:} & \text{ state } q \text{ recognizing a kernel } K \\
\textbf{Output:} & \text{ state recognizing } \bar{K} \\
1 & \text{ if } G(q) \text{ is not empty then return } G(q) \\
2 & \text{ if } q = q_\emptyset \text{ then return } q_e \\
3 & \text{ else if } q = q_e \text{ then return } q_0 \\
4 & \text{ else } \\
5 & \text{ for all } i = 1, \ldots, m \text{ do } r_i \leftarrow kcomp(q^a_i) \\
6 & G(q) \leftarrow kmake(r_1, \ldots, r_m) \\
7 & \text{ return } G(q)
\end{align*}
\]

Table 7.9: Algorithm \( kcomp \)

**Determinization and Minimization.**

The algorithm \( kdet\&\min \) that converts a NFA recognizing a fixed-language \( L \) into the minimal zip-DFA recognizing \( \langle L \rangle \) differs from \( det\&min() \) essentially in one letter: it uses \( kmake \) instead of \( \text{make} \). It is shown in Table 7.10.

**Example 7.17** Figure 7.11 shows again the NFA of Figure 7.6, and the minimal zip-DFA for the kernel of its language. The run of \( kdet\&\min(A) \) is shown at the bottom of the figure. For the difference with \( det\&min(A) \), consider the call \( kstate((\delta, e, \zeta)) \). Since the two recursive calls \( kstate(\{\eta\}) \) and \( kstate(\{\eta, \theta\}) \) return both state 1 with length 1, \( kmake(1, 1) \) does not create a new state, as \( make(1, 1) \) would do it returns state 1. The same occurs at the top call \( kstate(\{\alpha\}) \). \( \Box \)

**Exercises**

**Exercise 65** Prove that the minimal DFAs for the languages of length 3 have at most 8 states.

---

\(^3\)The operator is well defined because \( \langle L \rangle = \langle L' \rangle \) implies \( \langle L \rangle = \langle L' \rangle \).
Figure 7.11:
Exercise 66  Give an efficient algorithm that receives as input the minimal DFA of a fixed-length language and returns the number of words it contains.

Exercise 67  Let $\Sigma = \{0, 1\}$. Given $a, b \in \Sigma$, let $a \cdot b$ be the usual multiplication (an analog of boolean and) and let $a \oplus b$ be 0 if $a = b = 0$ and 1 otherwise (an analog of boolean or).

Consider the boolean function $f: \Sigma^6 \rightarrow \Sigma$ defined by

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 \cdot x_2) \oplus (x_3 \cdot x_4) \oplus (x_5 \cdot x_6)$$

1. Construct the minimal DFA recognizing $\{x_1, x_2, x_3, x_4, x_5, x_6 \mid f(x_1, x_2, x_3, x_4, x_5, x_6) = 1\}$.
   (For instance, the DFA accepts 111000 because $f(1, 1, 1, 0, 0, 0) = 1$, but not 101010, because $f(1, 0, 1, 0, 1, 0) = 0$.)

2. Construct the minimal DFA recognizing $\{x_1, x_2, x_3, x_4, x_5, x_6 \mid f(x_1, x_2, x_3, x_4, x_5, x_6) = 1\}$.
   (Notice the different order! Now the DFA accepts neither 111000, because $f(1, 0, 1, 0, 1, 0) = 0$, nor 101010, because $f(1, 0, 1, 1, 0) = 0$.)

3. More generally, consider the function

$$f(x_1, \ldots, x_{2n}) = \bigoplus_{1 \leq k \leq n} (x_{2k-1} \cdot x_{2k})$$

and the languages $\{x_1, x_2, \ldots, x_{2n-1}, x_{2n} \mid f(x_1, \ldots, x_{2n}) = 1\}$ and $\{x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n} \mid f(x_1, \ldots, x_{2n}) = 1\}$. 

Table 7.10: The algorithm $kdet&\min(A)$.
Show that the size of the minimal DFA grows linearly in $n$ for the first language, and exponentially in $n$ for the second language.

**Exercise 68** Given a language $L$ of length $n$ and a permutation $\pi$ of $\{1, \ldots, n\}$, we define $\pi(L) = \{a_{\pi(1)} \ldots a_{\pi(n)} \mid a_1 a_2 \ldots a_n \in L\}$. Prove that the following problem is NP-complete:

*Given:* A minimal DFA $A$ recognizing a fixed-length language, a number $k \geq 1$.

*Decide:* Is there a permutation $\pi$ of $\{1, \ldots, n\}$, where $n$ is the length of $L(A)$, such that the minimal DFA for $L(\pi(A))$ has at most $k$ states?

**Exercise 69** Given $X \subset \{0, 1, \ldots, 2^k - 1\}$, let $A_X$ be the minimal DFA recognizing the *lsbf* encodings of length $k$ of the elements of $X$.

- Define $X + 1$ by $X + 1 = \{x + 1 \mod 2^k \mid x \in X\}$. Give an algorithm that on input $A_X$ produces $A_{X+1}$ as output.
- Let $A_X = (Q, \{0, 1\}, \delta, q_0, F)$. Which is the set of numbers recognized by the automaton $(Q, \{0, 1\}, \delta', q_0, F)$, where $\delta'(q, b) = \delta(q, 1 - b)$?

**Exercise 70** Recall the definition of DFAs with negative transitions (DFA-nt’s) introduced in Exercise 26, and consider the alphabet $\{0, 1\}$. Show that if only transitions labelled by 1 can be negative, then every regular language over $\{0, 1\}$ has a *unique* minimal DFA-nt.

**Exercise 71** Define

$$emb(L) = \{(v_1, v_2) \in (\Sigma \times \Sigma)^n \mid v_2 \in L\}$$

and define $pre_S(L)$, where $S \in (\Sigma \times \Sigma)^*$ and $L \in \Sigma^*$, as follows:

$$pre_S(L) = \{w_1 \in \Sigma^n \mid \exists (v_1, v_2) \in S : v_1 = w_1 \text{ and } v_2 \in L\}$$

Show that $(pre_S(L))^a = \bigcup_{b \in \Sigma} pre_S[a, b](L^b)$.

**Exercise 72** Exhibit a family of relations $R_n \subseteq \{0, 1\}^n \times \{0, 1\}^n$ such that

- the minimal deterministic transducer for $R_n$ has $O(n)$ states, and
- the minimal DFA for $proj_1(R_n)$ has $2^{O(n)}$ states.
Chapter 8

Applications II: Verification

One of the main applications of automata theory is the automatic verification or falsification of correctness properties of hardware or software systems. Given a system (like a hardware circuit, a program, or a communication protocol), and a property (like “after termination the values of the variables $x$ and $y$ are equal” or “every sent message is eventually received”), we wish to automatically determine whether the system satisfies the property or not.

8.1 The Automata-Theoretic Approach to Verification

We consider discrete systems for which a notion of configuration can be defined. The system is always at a certain configuration, with instantaneous moves from one configuration to the next determined by the system dynamics. If the semantics allows a move from a configuration $c$ to another one $c'$, then we say that $c'$ is a legal successor of $c$. A configuration may have several successors, in which case the system is nondeterministic. There is a distinguished set of initial configurations. An execution is a sequence of configurations (finite or infinite) starting at some initial configuration, and in which every other configuration is a legal successor of its predecessor in the sequence. A full execution is either an infinite execution, or an execution whose last configuration has no successors.

In this chapter we are only interested in finite executions. The set of executions can then be seen as a language $E \subseteq C^*$, where the alphabet $C$ is the set of possible configurations of the system. We call $C^*$ the potential executions of the system.

Example 8.1 As an example of a system, consider the following program with two boolean variables $x, y$:

---

1We speak of the configurations of a system, and not of its states, in order to avoid confusion with the states of automata.
while $x = 1$ do
  if $y = 1$ then
    $x \leftarrow 0$
    $y \leftarrow 1 - x$
  end

A configuration of the program is a triple $[\ell, n_x, n_y]$, where $\ell \in \{1, 2, 3, 4, 5\}$ is the current value of the program counter, and $n_x, n_y \in \{0, 1\}$ are the current values of $x$ and $y$. So the set $C$ of configurations contains in this case $5 \times 2 \times 2 = 20$ elements. The initial configurations are $[1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]$, i.e., all configurations in which control is at line 1. The sequence

$[1, 1, 1] [2, 1, 1] [3, 1, 1] [4, 0, 1] [1, 0, 1] [5, 0, 1]$

is a full execution, while

$[1, 1, 0] [2, 1, 0] [4, 1, 0] [1, 1, 0]$

is also an execution, but not a full one. In fact, all the words of

$( [1, 1, 0] [2, 1, 0] [4, 1, 0] )^*$

are executions, and so the language $E$ of all executions is infinite.

Assume we wish to determine whether the system has an execution satisfying some property of interest. If both the language $E \subseteq C^*$ of executions and the language $P \subseteq C^*$ of potential executions that satisfy the property are regular, and we can construct automata recognizing them, then we can solve the problem by checking whether the language $E \cap P$ is empty, which can be decided using the algorithms of Chapter 4. This is the main insight behind the automata-theoretic approach to verification.

The requirement that the language $E$ of executions is regular is satisfied by all systems with finitely many reachable configurations (i.e., finitely many configurations $c$ such that some execution leads from some initial configuration to $c$). A system NFA recognizing the executions of the system can be easily obtained from the configuration graph: the graph having the reachable configurations as nodes, and arcs from each configuration to its successors. The construction is very simple: The states of the system NFA are the reachable configurations of the program plus a new state, which is also the initial state. For every transition $c \rightarrow c'$ of the graph there is a transition $c \xrightarrow{c'}$ in the NFA. All states are final.

Example 8.2 Figure 8.1 shows the configuration graph of the program of Example 8.1, and below it its system NFA. Notice that the labels of the transitions of the NFA carry no information, because they are just the name of the target state.

We wish to automatically determine if the system has a full execution such that initially $y = 1$, finally $y = 0$, and $y$ never increases. Let $[\ell, x, 0], [\ell, x, 1]$ stand for the sets of configurations where $y = 0$ and $y = 1$, respectively, but the values of $\ell$ and $x$ are arbitrary. Similarly, let $[5, x, 0]$ stand...
for the set of configurations with $\ell = 5$ and $y = 0$, but $x$ arbitrary. The set of potential executions satisfying the property is given by the regular expression

$$[\ell, x, 1][\ell, x, 1]^* [\ell, x, 0]^* [5, x, 0]$$

which is recognized by the property NFA at the top of Figure 8.2. Its intersection with the system NFA of Figure 8.1 is shown at the bottom of Figure 8.2. A light pink state of the pairing labeled by $[\ell, x, y]$ is the result of pairing the light pink state of the property NFA and the state $[\ell, x, y]$ of the system NFA. Since labels of the transitions of the pairing are always equal to the target state, they are omitted for the sake of readability.

Since no state of the intersection has a dark pink color, the intersection is empty, and so the program has no execution satisfying the property.

**Example 8.3** We wish now to automatically determine whether the assignment $y \leftarrow 1 - x$ in line 4 of the program of Example 8.1 is redundant and can be safely removed. This is the case if
the assignment never changes the value of \( y \). The potential executions of the program in which the assignment changes the value of \( y \) at some point correspond to the regular expression
\[
[\ell, x, y]^* \left( [4, x, 0] [1, x, 1] + [4, x, 1] [1, x, 0] \right) [\ell, x, y]^*.
\]
A property NFA for this expression can be easily constructed, and its intersection with the system NFA is again empty. So the property holds, and the assignment is indeed redundant.

8.2 Networks of Automata.

Concurrent systems are particularly difficult to design correctly, which makes them a suitable target for automatic verification techniques. These systems usually consist of a number of communicating sequential components. In this section we exploit this structure to define configuration graphs in a more systematic way. We assign an NFA to each sequential component, yielding a network of automata and then define the graph by means of an automata theoretic construction.

A network of automata is a tuple \( A = \langle A_1, \ldots, A_n \rangle \) of NFAs with pairwise disjoint sets of states. Each NFA has its own alphabet \( \Sigma_i \) (the alphabets \( \Sigma_1, \ldots, \Sigma_n \) are not necessarily pairwise disjoint). Alphabet letters are called actions. Given an action \( a \), we say that the \( i \)-th NFA participates in \( a \) if \( a \in \Sigma_i \).

A configuration of a network is a tuple \( \langle q_1, \ldots, q_n \rangle \) of states, where \( q_i \in Q_i \) for every \( i \in \{1, \ldots, n\} \). The initial configuration is the configuration \( \langle q_01, \ldots, q_0n \rangle \), where \( q_0i \) is the initial state of \( A_i \). An action \( a \) is enabled at a configuration \( \langle q_1, \ldots, q_n \rangle \) if for every \( i \in \{1, \ldots, n\} \) such that \( A_i \) participates in \( a \) there is a transition \( (q_i, a, q'_i) \in \delta_i \). If \( a \) is enabled, then it can occur, and its occurrence makes all participating NFAs \( A_i \) move to the state \( q'_i \), while the non-participating...
NFAs do not change their state. The configuration reached by the occurrence of \( a \) is a successor of \((q_1, \ldots, q_n)\).

**Example 8.4** The upper part of Figure 8.3 shows a network of three NFAs modeling a 3-bit counter.

We call the NFAs \( A_0, A_1, A_2 \) instead of \( A_1, A_2, A_3 \) to better reflect their meaning: \( A_i \) stands for the \( i \)-th bit. Each NFA but the last one has three states, two of which are marked with 0 and 1. The alphabets are

\[
\Sigma_0 = \{\text{inc, inc}_1, 0, \ldots, 7\} \quad \Sigma_1 = \{\text{inc}_1, \text{inc}_2, 0, \ldots, 7\} \quad \Sigma_2 = \{\text{inc}_2, 0, \ldots, 7\}
\]

Intuitively, the system interacts with its environment by means of the actions \( \text{inc}, 0, 1, \ldots, 7 \). More precisely, \( \text{inc} \) models a request of the environment to increase the counter by 1, and \( i \in \{0, 1, \ldots, 7\} \) models a query of the environment asking if \( i \) is the current value of the counter. A configuration of the form \([b_2, b_1, b_0]\), where \( b_2, b_1, b_0 \in \{0, 1\} \), indicates that the current value of the counter is \( 4b_2 + 2b_1 + b_0 \) (configurations are represented as triples of states of \( A_2, A_1, A_0 \), in that order).

**Example 8.5** The bottom part of Figure 8.3 shows the asynchronous product of the network modeling the 3-bit counter, shown at the top of the figure. (Actually, all states are final, but have been drawn as simple instead of double ellipses for simplicity.) Observe that at the configurations \([1, \text{aux}, 0]\) and \([0, \text{aux}, 0]\) the actions \( \text{inc} \) and \( \text{inc}_2 \) are concurrent: they are both enabled, and the sets of automata participating in them are disjoint. This means that they can occur independently of each other.

Given a network of automata we define its *asynchronous product* as the output of algorithm Async in Table 8.1. Starting at the initial configuration, the algorithm repeatedly picks a configuration from the worklist and constructs its successors; adding \( nw \) successors to the worklist.

**Example 8.5** The bottom part of Figure 8.3 shows the asynchronous product of the network modeling the 3-bit counter, shown at the top of the figure. (Actually, all states are final, but have been drawn as simple instead of double ellipses for simplicity.) Observe that at the configurations \([1, \text{aux}, 0]\) and \([0, \text{aux}, 0]\) the actions \( \text{inc} \) and \( \text{inc}_2 \) are concurrent: they are both enabled, and the sets of automata participating in them are disjoint. This means that they can occur independently of each other.

This proposition, whose proof is an easy consequence of the definitions, characterizes the executions of the network in terms of the asynchronous product:

**Proposition 8.6** A sequence \( c_0, c_1 \ldots c_n \) of configurations is an execution of the network \( A_1 \otimes \cdots \otimes A_n \) if and only if there is a word \( a_1 \ldots a_n \in \Sigma^* \) such that \( c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} c_n \) is an accepting run of \( A_1 \otimes \cdots \otimes A_n \).

Notice that \( A_1 \otimes \cdots \otimes A_n \) can be easily transformed into the system NFA accepting the executions of the network by means of a transformation similar to the one described in Figure 8.1.

The next example shows how to model a (possibly concurrent) program with finite-domain variables (like booleans or finite ranges) as a network of automata. The key idea is to have a network component for each process and for each variable. A process-component has a state for each control point of the process, and a variable component has one state for each possible value of the variable. Reading or writing of a variable by a process is modeled as a *joint action* with the process-component and the variable-component as participants.
Figure 8.3: A network modeling a 3-bit counter and its asynchronous product.
AsyncProduct\((A_1, \ldots, A_n)\)

**Input:** a network of automata \(A = A_1, \ldots, A_n\), where 
\(A_1 = (Q_1, \Sigma_1, \delta_1, q_01, Q_1), \ldots, A_n = (Q_n, \Sigma_n, \delta_n, q_0n, Q_n)\)

**Output:** the asynchronous product \(A_1 \otimes \cdots \otimes A_n = (Q, \Sigma, q_0, F)\)

1. \(Q, \delta, F \leftarrow \emptyset\)
2. \(q_0 \leftarrow [q_01, \ldots, q_0n]\)
3. \(W \leftarrow \{[q_01, \ldots, q_0n]\}\)
4. **while** \(W \neq \emptyset\) **do**
5.   **pick** \([q_1, \ldots, q_n]\) from \(W\)
6.   **add** \([q_1, \ldots, q_n]\) to \(Q\)
7.   **add** \([q_1, \ldots, q_n]\) to \(F\)
8.   **for all** \(a \in \Sigma_1 \cup \ldots \cup \Sigma_n\) **do**
9.     **for all** \(i \in [1..n]\) **do**
10.       **if** \(a \in \Sigma_i\) **then** \(Q_i' \leftarrow \delta_i(q_i, a)\) **else** \(Q_i' = \{q_i\}\)
11.     **for all** \([q_1', \ldots, q_n'] \in Q_1' \times \ldots \times Q_n'\) **do**
12.       **if** \([q_1', \ldots, q_n'] \notin Q\) **then** **add** \([q_1', \ldots, q_n']\) to \(W\)
13.       **add** \(([q_1, \ldots, q_n], a, [q_1', \ldots, q_n'])\) to \(\delta\)
14. **return** \((Q, \Sigma, \delta, q_0, F)\)

Table 8.1: Asynchronous product of a tuple of automata.

**Example 8.7** The network of Figure 8.4 models a version of Lamport and Burns’ 1-bit mutual exclusion algorithm for two processes\(^2\). In the algorithm, process 0 and process 1 communicate through two shared boolean variables, \(b_0\) and \(b_1\), which initially have the value 0. Process \(i\) reads and writes variable \(b_i\) and reads variable \(b_{(1-i)}\). The algorithm should guarantee that the processes 0 and 1 never are simultaneously in their critical sections. Other properties the algorithm should satisfy are discussed later.

The NFAs for the processes are shown in Figure 8.4. Initially, process 0 is in its non-critical section (local state \(nc_0\)); it can also be trying to enter its critical section (\(t_0\)), or be already in its critical section (\(c_0\)). It can move from \(nc_0\) to \(t_0\) at any time by setting \(b_0\) to 1; it can move from \(t_0\) to \(c_0\) if the current value of \(b_1\) is 0; finally, it can move from \(c_0\) to \(nc_0\) at any time by setting \(b_0\) to 0.

Process 1 is a bit more complicated. While \(nc_1\), \(t_1\), and \(c_1\) play the same rôle as in process 0, the local states \(q_1\) and \(q_1'\) model a “polite” behavior: Intuitively, if process 1 sees that process 0 is either trying to enter or in the critical section, it moves to an “after you” local state \(q_1\), and then sets \(b_1\) to 0 to signal that it is no longer trying to enter its critical section (local state \(q_1'\)). It can then return to the non-critical section if the value of \(b_0\) is 0.

In principle, in the non-critical sections the processes can also do some unspecified work, which

should be represented by transition from $nc_0$ and $nc_1$ to themselves. Since for the purposes of this chapter these transitions are not important, we have omitted them.

A configuration of this network is a four tuple. Transitions of the asynchronous product correspond to the execution of a read or a write statement by one of the processes. The asynchronous product is shown in Figure 8.5, where $\langle v_0, v_1, s_0, s_1 \rangle$ represents the configuration in which the NFA modelling processes 0 and 1 are in states $s_0, s_1$, and in which $b_0 = v_0$ and $b_1 = v_1$.

8.2.1 Checking Properties

We use Lamport’s algorithm to present some more examples of properties and how to check them automatically.

The mutual exclusion property can be easily formalized: it holds if the asynchronous product does not contain any configuration of the form $[v_0, v_1, c_0, c_1]$, where $v_0, v_1 \in \{0, 1\}$. The property
8.2. NETWORKS OF AUTOMATA.

can be easily checked on-the-fly while constructing the asynchronous product, and a quick inspection of Figure 8.5 shows that it holds. Notice that in this case we do not need to construct the NFA for the executions of the program. This is always the case if we only wish to check the reachability of a configuration or set of configurations. Other properties of interest for the algorithm are:

- Deadlock freedom. The algorithm is deadlock-free if every configuration of the asynchronous product has at least one successor. Again, the property can be checked on the fly, and it holds for Lamport’s algorithm.

- Bounded overtaking. After process 0 signals its interest in accessing the critical section (by moving to state \( t_0 \)), process 1 can enter the critical section at most once before process 0 enters the critical section.

This property can be checked using the NFA \( E \) recognizing the executions of the network, obtained as explained above by renaming the labels of the transitions of the asynchronous product. Let \( NC_i, T_i, C_i \) be the sets of configurations in which process \( i \) is in its non-critical

![Figure 8.5: Asynchronous product of the network of Figure 8.4.](image-url)
section, is trying to access its critical section, or is in its critical section, respectively. Let \( \Sigma \) stand for the set of all configurations. The regular expression
\[
r = \Sigma^* \, T_0 \, (\Sigma \setminus C_0)^* \, C_1 \, (\Sigma \setminus C_1)^* \, N \, C_1 \, (\Sigma \setminus C_0)^* \, C_1 \, \Sigma^*
\]
represents all the possible executions that violate the property. Now, if \( E \) is the NFA recognizing the language of executions, the property holds if and only if \( L(E) \cap L(r) = \emptyset \), which can be checked using the algorithms of Chapter 4.

The straightforward method to check if \( L(E) \cap L(r) = \emptyset \) has four steps: (1) construct the NFA \( E \) for the network \( A_1, \ldots, A_n \) using \texttt{AsyncProduct()}; (2) transform \( r \) into an NFA \( V \) (\( V \) for “violations”) using the algorithm of Section 2.4.1; (3) construct the NFA \( E \cap V \) using \texttt{intersNFA()}; (4) check emptiness of \( E \cap V \) using \texttt{empty()}. The key problem of this approach is that the number of states of \( E \) can be as high as the product of the number of states of the \( \theta \) components \( A_1, \ldots, A_n \), which can easily exceed the available memory. This is called the \textit{state-explosion} problem, and the literature contains a wealth of proposals to deal with it. We conclude the section with a first easy step towards palliating this problem. A more in depth discussion can be found in the next section.

Observe that the NFA \( E \) may have \textit{more} states than \( E \cap V \): if a state of \( E \) is not reachable by any word on which \( V \) has a run, then it does not appear in \( E \cap V \). Since the transitions of \( V \) are often labeled with only a small fraction of the letters in the alphabet of \( E \), the difference in size between \( E \) and \( E \cap V \) can be considerable, and so it is better to \textit{directly} construct \( E \cap V \), bypassing the construction of \( E \). This is easily achieved by observing that the intersection \( A_1 \cap A_2 \) of two NFAs \( A_1, A_2 \) corresponds to the particular case of the asynchronous product in which \( \Sigma_1 \subseteq \Sigma_2 \) (or vice versa): if \( \Sigma_1 \subseteq \Sigma_2 \), then \( A_2 \) participates in every action, and the NFAs \( A_1 \otimes A_2 \) and \( A_1 \cap A_2 \) coincide. More generally, if \( \Sigma_1 \cup \cdots \cup \Sigma_n \subseteq \Sigma_{n+1} \), then \( A_1 \otimes \cdots \otimes A_n \otimes A_{n+1} = (A_1 \otimes \cdots \otimes A_n) \cap A_{n+1} \).

Since the alphabet of \( V \) is the union of the alphabets of \( A_1, \ldots, A_n \), we can then check emptiness of \( A_1 \otimes \cdots \otimes A_n \otimes V \) by means of the algorithm of Table 8.2.

### 8.3 The State-Explosion Problem

The automata-theoretic approach constructs an NFA \( V \) recognizing the potential executions of the system that violate the property one is interested in, and checks whether the automaton \( E \cap V \) is empty, where \( E \) is an NFA recognizing the executions of the system. This is done by constructing the set of states of \( E \cap V \), while simultaneously checking if any of them is final.

The number of states of \( E \) can be very high. If we model \( E \) as a network of automata, the number can be as high as the product of the number of states of all the components of the network. So the approach has exponential worst-case complexity. The following result shows that this cannot be avoided unless \( \text{P=PSpace} \).

**Theorem 8.8** The following problem is \( \text{PSPACE-complete} \).

\[ \text{Given: A network of automata } A_1, \ldots, A_n \text{ over alphabets } \Sigma_1, \ldots, \Sigma_n, \text{ a NFA } V \text{ over } \Sigma_1 \cup \ldots \cup \Sigma_n. \]

\[ \text{Decide: if } L(A_1 \otimes \cdots \otimes A_n \otimes V) \neq \emptyset. \]
The state-explosion problem

CheckViol(A₁,…,Aₙ, V)
Input: a network ⟨A₁,…,Aₙ⟩, where Aᵢ = (Qᵢ, Σᵢ, δᵢ, q₀ᵢ, Qᵢ); an NFA V = (Qᵥ, Σ₁ ∪ … ∪ Σₙ, δᵥ, q₀ᵥ, Fᵥ).
Output: true if A₁ ⊗⋅⋅⋅⊗Aₙ ⊗ V is nonempty, false otherwise.

1. Q ← ∅; q₀ ← [q₀₁,…,q₀ₙ, q₀ᵥ]
2. W ← {q₀}
3. while W ≠ ∅ do
4.   pick [q₁,…,qₙ, q] from W
5.   add [q₁,…,qₙ, q] to Q
6.   for all a ∈ Σ₁ ∪ … ∪ Σₙ do
7.     for all i ∈ [1..n] do
8.       if a ∈ Σᵢ then Qᵢ′ ← δᵢ(qᵢ, a) else Qᵢ′ = {qᵢ}
9.       Q′ ← δᵥ(q, a)
10.     for all [q₁′,…,qₙ′, q′] ∈ Q₁′ × … × Qₙ′ × Q′ do
11.       if \( \land_{i=1}^{n} q_i' \in F_i \) and q ∈ Fᵥ then return true
12.       if [q₁′,…,qₙ′, q′] ∉ Q then add [q₁′,…,qₙ′, q′] to W
13. return false

Table 8.2: Algorithm to check violation of a property.

Proof: We only give a high-level sketch of the proof. To prove that the problem is in PSPACE, we show that it belongs to NPSpace and apply Savitch’s theorem. The polynomial-space nondeterministic algorithm just guesses an execution of the product, one configuration at a time, leading to a final configuration. Notice that storing a configuration requires linear space.

PSPACE-hardness is proven by reduction from the acceptance problem for linearly bounded automata. A linearly bounded automaton (LBA) is a deterministic Turing machine that always halts and only uses the part of the tape containing the input. Given an LBA A, we construct in linear time a network of automata that “simulates” A. The network has one component modeling the control of A (notice that the control is essentially a DFA), and one component for each tape cell used by the input. The states of the control component are pairs (q, k), where q is a control state of A, and k is a head position. The states of a cell-component are the possible tape symbols. The transitions correspond to the possible moves of A according to its transition table. Acceptance of A corresponds to reachability of certain configurations in the network, which can be easily encoded as an emptiness problem.

8.3.1 Symbolic State-space Exploration

Figure 8.6 shows again the program of Example 8.1, and its flowgraph. An edge of the flowgraph
leading from node \( \ell \) to node \( \ell' \) can be associated a step relation \( S_{\ell,\ell'} \) containing all pairs of configurations \(((\ell, x_0, y_0), (\ell', x'_0, y'_0))\) such that if at control point \( \ell \) the current values of the variables are \( x_0, y_0 \), then the program can take a step after which the new control point is \( \ell' \), and the new values are \( x'_0, y'_0 \). For instance, for the edge leading from node 4 to node 1 we have

\[
S_{4,1} = \{ (4, x_0, y_0), (1, x'_0, y'_0) \mid x'_0 = x_0, y'_0 = 1 - x_0 \}
\]

and for the edge leading from 1 to 2

\[
S_{1,2} = \{ (1, x_0, y_0), (2, x'_0, y'_0) \mid x_0 = 1 = x'_0, y'_0 = y_0 \}
\]

It will be convenient to assign a relation to every pair of nodes of the control graph, even to those not connected by any edge. If no edge leads from \( a \) to \( b \), then we define \( R_{a,b} = \emptyset \). The complete program is then described by the global step relation

\[
S = \bigcup_{a,b \in C} S_{a,b}
\]

where \( C \) is the set of control points.

Given a set \( I \) of initial configurations, the set of configurations reachable from \( I \) can be computed by the following algorithm, which repeatedly applies the Post operation:

*Reach\((I, R)\)*

**Input:** set \( I \) of initial configurations; relation \( R \)

**Output:** set of configurations reachable from \( I \)

1. \( \text{Old}P \leftarrow \emptyset; P \leftarrow I \)
2. while \( P \neq \text{Old}P \) do
3. \( \text{Old}P \leftarrow P \)
4. \( P \leftarrow \text{Union}(P, \text{Post}(P, S)) \)
5. return \( P \)
The algorithm can be implemented using different data structures. The verification community distinguishes between explicit and symbolic data structures. Explicit data structures store separately each of the configurations of \( P \), and the pairs of configurations of \( S \); typical examples are lists and hash tables. Their distinctive feature is that the memory needed to store a set is proportional to the number of its elements. Symbolic data structures, on the contrary, do not store a set by storing each of its elements; they store a representation of the set itself. A prominent example of a symbolic data structure are finite automata and transducers: given an encoding of configurations as words over some alphabet \( \Sigma \), the set \( P \) and the step relation \( S \) are represented by an automaton and a transducer, respectively, recognizing the encodings of its elements. Their sizes can be much smaller than the sizes of \( P \) or \( S \). For instance, if \( P \) is the set of all possible configurations then its encoding is usually \( \Sigma^* \), which is encoded by a very small automaton.

Symbolic data structures are only useful if all the operations required by the algorithm can be implemented without having to switch to an explicit data structure. This is the case of automata and transducers: \textbf{Union}, \textbf{Post}, and the equality check in the condition of the while loop operation are implemented by the algorithms of Chapters 4 and 6, or, if they are fixed-length, by the algorithms of Chapter 7.

Symbolic data structures are interesting when the set of reachable configurations can be very large, or even infinite. When the set is small, the overhead of symbolic data structures usually offsets the advantage of a compact representation. Despite this, and in order to illustrate the method, we apply it to the five-line program of Figure 8.6. The fixed-length transducer for the step relation \( S \) is shown in Figure 8.7; a configuration \([\ell,x_0,y_0]\) is encoded by the word \( \ell x_0 y_0 \) of length 3. Consider for instance the transition labeled by \([4,1]\). Using it the transducer can recognize four pairs, which describe the action of the instruction \( y \leftarrow 1 - x \), namely

\[
\begin{pmatrix}
400 \\
101
\end{pmatrix}
\begin{pmatrix}
401 \\
101
\end{pmatrix}
\begin{pmatrix}
410 \\
110
\end{pmatrix}
\begin{pmatrix}
411 \\
110
\end{pmatrix}
\]

Figure 8.8 shows minimal DFAs for the set \( I \) and for the sets obtained after each iteration of the while loop.

\textbf{Variable orders.}

We have defined a configuration of the program of Example 8.1 as a triple \([\ell,n_x,n_y]\), and we have encoded it as the word \( \ell n_x n_y \). We could have also encoded it as the word \( n_x \ell n_y, n_y \ell n_x \), or as any other permutation, since in all cases the information content is the same. Of course, when encoding a set of configurations all the configurations must be encoded using the same \textit{variable order}.

While the information content is independent of the variable order, the \textit{size} of the automaton encoding a set is not. An extreme case is given by the following example.

\textbf{Example 8.9} Consider the set of tuples \( X = \{(x_1, x_2, \ldots, x_{2k}) \mid x_1, \ldots, x_{2k} \in \{0, 1\}\} \), and the subset \( Y \subseteq X \) of tuples satisfying \( x_1 = x_k, x_2 = x_{k+1}, \ldots, x_k = x_{2k} \). Consider two possible encodings of a
tuple $[x_1, x_2, \ldots, x_{2k}]$: by the word $x_1x_2 \ldots x_{2k}$, and by the word $x_1x_{k+1}x_2x_{k+2} \ldots x_kx_{2k}$. In the first case, the encoding of $Y$ for $k = 3$ is the language

$$L_1 = \{000000, 001001, 010010, 011011, 100100, 101101, 110110, 111111\}$$

and in the second the language

$$L_2 = \{000000, 000011, 001100, 001111, 110000, 110011, 111011, 111100, 111111\}$$

Figure 8.9 shows the minimal DFAs for the languages $L_1$ and $L_2$. It is easy to see that the minimal DFA for $L_1$ has at least $2^k$ states: since for every word $w \in \{0, 1\}^k$ the residual $L_1^w$ is equal to $\{w\}$, the language $L_1$ has a different residual for each word of length $k$, and so the minimal DFA has at least $2^k$ states (the exact number is $2^{k+1} + 2^k - 2$). On the other hand, it is easy to see that the minimal DFA for $L_2$ has only $3k + 1$ states. So a good variable order can lead to a exponentially more compact representation.

We can also appreciate the effect of the variable order in Lamport’s algorithm. The set of reachable configurations, sorted according to the state of the first process and then to the state of
the second process, is

\[
\begin{align*}
\langle nc_0, nc_1, 0, 0 \rangle & \quad \langle t_0, nc_1, 1, 0 \rangle & \quad \langle c_0, nc_1, 1, 0 \rangle \\
\langle nc_0, t_1, 0, 1 \rangle & \quad \langle t_0, t_1, 1, 1 \rangle & \quad \langle c_0, t_1, 1, 1 \rangle \\
\langle nc_0, c_1, 0, 1 \rangle & \quad \langle t_0, c_1, 1, 1 \rangle & \\
\langle nc_0, q_1, 0, 1 \rangle & \quad \langle t_0, q_1, 1, 1 \rangle & \quad \langle c_0, q_1, 1, 1 \rangle \\
\langle nc_0, q'_1, 0, 0 \rangle & \quad \langle t_0, q'_1, 1, 0 \rangle & \quad \langle c_0, q'_1, 1, 0 \rangle
\end{align*}
\]

If we encode a tuple \(\langle s_0, s_1, v_0, v_1 \rangle\) by the word \(v_0s_0s_1v_1\), the set of reachable configurations is recognized by the minimal DFA on the left of Figure 8.10. However, if we encode by the word \(v_1s_1s_0v_0\) we get the minimal DFA on the right. The same example can be used to visualize how by adding configurations to a set the size of its minimal DFA can decrease. If we add the “missing” configuration \(\langle c_0, c_1, 1, 1 \rangle\) to the set of reachable configurations (filling the “hole” in the list above), two states of the DFAs of Figure 8.10 can be merged, yielding the minimal DFAs of Figure 8.11. Observe also that the set of all configurations, reachable or not, contains 120 elements, but is recognized by a five-state DFA.

8.4 Safety and Liveness Properties

Apart from the state-explosion problem, the automata-theoretic approach to automatic verification described in this chapter has a second limitation: it assumes that the violations of the property can be described by a set of finite executions. In other words, and loosely speaking, it assumes that if the property is violated, then it is already violated after a finite amount of time, independently of the rest of the execution. Not all properties are of this form. A typical example is the property “if a process requests access to the critical section, it eventually enters the critical section” (without specifying how long it may take). After finite time we can only tell that the process has not entered the critical section yet, but it might enter it in the future. A violation of the property can only be witnessed by an infinite execution, in which we observe that the process requests access, but the access is never granted.

Properties which are violated by finite executions are called safety properties. Intuitively, they correspond to properties of the form “nothing bad ever happens”. Typical examples are “the system never deadlocks”, or, more generally, “the system never enters a set of bad states”. Clearly, every interesting system must also satisfy properties of the form “something good eventually happens”, because otherwise the system that does nothing would already satisfy all properties. Properties of this kind are called liveness properties, and can only be witnessed by infinite executions. Fortunately, the automata-theoretic approach can be extended to liveness properties. This requires to develop a theory of automata on infinite words, which is the subject of the second part of this book.

The application of this theory to the verification of liveness properties is presented in Chapter 14.

Exercises
Exercise 73  Let $\Sigma = \{\text{request}, \text{answer}, \text{working}, \text{idle}\}$.

1. Build an automaton recognizing all words with the property $P_1$: for every occurrence of request there is a later occurrence of answer.

2. $P_1$ does not imply that every occurrence of request has “its own” answer: for instance, the sequence request request answer satisfies $P_1$, but both requests must necessarily be mapped to the same answer. But, if words were infinite and there were infinitely many requests, would $P_1$ guarantee that every request has its own answer?

   More precisely, let $w = w_1w_2\cdots$ satisfying $P_1$ and containing infinitely many occurrences of request, and define $f : \mathbb{N} \to \mathbb{N}$ such that $w_{f(i)}$ is the $i$th request in $w$. Is there always an injective function $g : \mathbb{N} \to \mathbb{N}$ satisfying $w_{g(i)} = \text{answer}$ and $f(i) < g(i)$ for all $i \in \{1, \ldots, k\}$?

3. Build an automaton recognizing all words with the property $P_2$: there is an occurrence of answer, and before that only working and request occur.

4. Using the intersection construction, prove that all accepting runs of the automaton below satisfy $P_1$. Find all accepting runs violating $P_2$.

\[
\begin{array}{c}
\text{start} \\
\longrightarrow \Sigma \\
\longrightarrow \text{answer} \\
\longrightarrow \text{answer} \\
\end{array}
\]

Exercise 74  This exercise focuses on modelling and verification of mutual exclusion (mutex) algorithms. Consider two processes running the following mutex algorithm, where $id$ is an identifier, a local variable having value 0 for one of the processes and value 1 for the other.

\[
\textbf{while true do} \\
\quad \text{loop-arbitrarily-many-times} \\
\quad \text{non-critical-command} \\
\quad \textbf{enter}(id) \\
\quad \text{critical-command} \\
\quad \textbf{leave}(id)
\]

The procedures $\text{enter}()$ and $\text{leave}()$ are specified below. They use a global variable $\text{turn}$, initially set to 0.

\[
\begin{array}{l}
\text{proc enter}(i) \\
\quad \textbf{while turn} = 1 - i \textbf{ do skip}
\end{array}
\quad
\begin{array}{l}
\text{proc leave}(i) \\
\quad \textbf{turn} \leftarrow 1 - i
\end{array}
\]
Design an asynchronous network of automata capturing this algorithm. Furthermore, build an automaton recognizing all runs reaching a configuration with both agents in the critical section. Using the intersection algorithm, prove that there are no such runs of this system, i.e. it is a mutex algorithm. Do all infinite runs satisfy that if a process wants to enter the critical section then it eventually enters it?

Consider now a different definition of enter and leave. Now the processes use two boolean variables \( \text{flag}[0] \) and \( \text{flag}[1] \) initially set to false.

\[
\begin{align*}
\text{proc enter}(i) \\
\text{flag}[i] &\leftarrow \text{true} \\
\text{while } \text{flag}[1 - i] \text{ do skip}
\end{align*}
\]

\[
\begin{align*}
\text{proc leave}(i) \\
\text{flag}[i] &\leftarrow \text{false}
\end{align*}
\]

Design an asynchronous network of automata capturing this behaviour. Can a deadlock occur?

Finally, Peterson’s algorithm combines both approaches. The processes use variables \( \text{turn} \), initially set to 0, and \( \text{flag}[0], \text{flag}[1] \), initially set to \( \text{false} \). The procedures \( \text{enter} \) and \( \text{leave} \) are defined as follows:

\[
\begin{align*}
\text{proc enter}(i) \\
\text{turn} &\leftarrow 1 - i \\
\text{flag}[i] &\leftarrow \text{true} \\
\text{while } \text{flag}[1 - i] \text{ and } \text{turn} = 1 - i \text{ do skip}
\end{align*}
\]

Can a deadlock occur? What kind of starvation can occur?

Exercise 75 Consider a circular railway divided into 8 tracks: \( 0 \rightarrow 1 \rightarrow \ldots \rightarrow 7 \rightarrow 0 \). In the railway circulate three trains, modeled by three automata \( T_1, T_2, \) and \( T_3 \). Each automaton \( T_j \) has states \( \{q_{i,0}, \ldots , q_{i,7}\} \), alphabet \( \{\text{enter}[i, j] \mid 0 \leq j \leq 7\} \) (where \( \text{enter}[i, j] \) models that train \( i \) enters track \( j \)), transition relation \( \{(q_{i,j}, \text{enter}[i, j \oplus 1], q_{i,j \oplus 1}) \mid 0 \leq j \leq 7\} \), and initial state \( q_{1,2} \), where \( \oplus \) denotes addition modulo 8.

Define automata \( C_0, \ldots , C_7 \) (the local controllers) to make sure that two trains can never be on the same or adjacent tracks (i.e., there must always be at least one empty track between two trains). Each controller \( C_j \) can only have knowledge of the state of the tracks \( j \oplus 1, j, \) and \( j \oplus 1, \). There must be no deadlocks, and every train must eventually visit every track. More formally, the asynchronous product \( R = C_0 \otimes \ldots \otimes C_7 \otimes T_1 \otimes T_2 \otimes T_3 \) must satisfy the following specification:

- For \( j = 0, \ldots , 7 \): \( C_j \) has alphabet \( \{\text{enter}[i, j \oplus 1], \text{enter}[i, j], \text{enter}[i, j \oplus 1] \mid 1 \leq i \leq 3\} \).
  (\( C_j \) only knows the state of tracks \( j \oplus 1, j, \) and \( j \oplus 1, \).)

- For \( i = 1, 2, 3 \): \( L(R) |_{L} = (\text{enter}[i, 2i] \text{enter}[i, 2i \oplus 1] \ldots \text{enter}[i, 2i \oplus 7])^* \).
  (No deadlocks, and every train eventually visits every segment.)

- For every word \( w \in L(R) \): if \( w = w_1 \text{enter}[i, j] \text{enter}[i', j'] w_2 \) and \( i' \neq i \), then \( |j - j'| \neq \{0, 1, 7\} \).
  (No two trains on the same or adjacent tracks.)
Figure 8.8: Minimal DFAs for the reachable configurations of the program of Figure 8.6
Figure 8.9: Minimal DFAs for the languages $L_1$ and $L_2$
Figure 8.10: Minimal DFAs for the reachable configurations of Lamport’s algorithm. On the left a configuration \( \langle s_0, s_1, v_0, v_1, q \rangle \) is encoded by the word \( s_0 s_1 v_0 v_1 q \), on the right by \( v_1 s_1 s_0 v_0 \).

Figure 8.11: Minimal DFAs for the reachable configurations of Lamport’s algorithm plus \( \langle c_0, c_1, 1, 1 \rangle \).
Chapter 9

Automata and Logic

A regular expression can be seen as a set of instructions (a ‘recipe’) for generating the words of a language. For instance, the expression $aa(a + b)^*b$ can be interpreted as “write two $a$’s, repeatedly write $a$ or $b$ an arbitrary number of times, and then write a $b$”. We say that regular expressions are an operational description language.

Languages can also be described in declarative style, as the set of words that satisfy a property. For instance, “the words over $\{a, b\}$ containing an even number of $a$’s and an even number of $b$’s” is a declarative description. A language may have a simple declarative description and a complicated operational description as a regular expression. For instance, the regular expression

$$(aa + bb + (ab + ba)(aa + bb)^*(ba + ab))^*$$

is a natural operational description of the language above, and it is arguably less intuitive than the declarative one. This becomes even more clear if we consider the language of the words over $\{a, b, c\}$ containing an even number of $a$’s, of $b$’s, and of $c$’s.

In this chapter we present a logical formalism for the declarative description of regular languages. We use logical formulas to describe properties of words, and logical operators to construct complex properties out of simpler ones. We then show how to automatically translate a formula describing a property of words into an automaton recognizing the words satisfying the property. As a consequence, we obtain an algorithm to convert declarative into operational descriptions, and vice versa.

9.1 First-Order Logic on Words

In declarative style, a language is defined by its membership predicate, i.e., the property that words must satisfy in order to belong to it. Predicate logic is the standard language to express membership predicates. Starting from some natural, “atomic” predicates, more complex ones can be constructed through boolean combinations and quantification. We introduce atomic predicates $Q_a(x)$, where $a$ is a letter, and $x$ ranges over the positions of the word. The intended meaning is “the letter at
position \( x \) is an \( a \)." For instance, the property "all letters are \( a \)" is formalized by the formula 
\[ \forall x \ Q_a(x). \]

In order to express relations between positions we add to the syntax the predicate \( x < y \), with intended meaning "position \( x \) is smaller than (i.e., lies to the left of) position \( y \)." For example, the property "if the letter at a position is an \( a \), then all letters to the right of this position are also \( a \)" is formalized by the formula

\[ \forall x \forall y ((Q_a(x) \land x < y) \rightarrow Q_a(y)). \]

**Definition 9.1** Let \( V = \{x, y, z, \ldots\} \) be an infinite set of variables, and let \( \Sigma = \{a, b, c, \ldots\} \) be a finite alphabet. The set \( \text{FO}(\Sigma) \) of first-order formulas over \( \Sigma \) is the set of expressions generated by the grammar:

\[ \varphi ::= Q_a(x) \mid x < y \mid \neg \varphi \mid (\varphi \lor \varphi) \mid \exists x \varphi. \]

As usual, variables within the scope of an existential quantifier are "bounded," and otherwise "free." A formula without free variables is a "sentence." Sentences of \( \text{FO}(\Sigma) \) are interpreted on words over \( \Sigma \). For instance, \( \forall x Q_a(x) \) is true for the word \( aa \), but false for word \( ab \). Formulas with free variables cannot be interpreted on words alone: it does not make sense to ask whether \( Q_a(x) \) holds for the word \( ab \) or not. A formula with free variables is interpreted over a pair \((w, I)\), where \( I \) assigns to each free variable (and perhaps to others) a position in the word. For instance, \( Q_a(x) \) is true for the pair \((ab, x \mapsto 1)\), because the letter at position 1 of \( ab \) is \( a \), but false for \((ab, x \mapsto 2)\).

**Definition 9.2** An interpretation of a formula \( \varphi \) of \( \text{FO}(\Sigma) \) is a pair \((w, I)\) where \( w \in \Sigma^* \) and \( I \) is a mapping that assigns to every free variable \( x \) a position \( I(x) \in \{1, \ldots, |w|\} \) (the mapping may also assign positions to other variables).

Notice that if \( \varphi \) is a sentence then a pair \((w, \bar{I})\), where \( \bar{I} \) is the empty mapping that does not assign any position to any variable, is an interpretation of \( \varphi \). Instead of \((w, \bar{I})\) we write simply \( w \).

We now formally define when an interpretation satisfies a formula. Given a word \( w \) and a number \( k \), let \( w[k] \) denote the letter of \( w \) at position \( k \).

**Definition 9.3** The satisfaction relation \((w, I) \models \varphi\) between a formula \( \varphi \) of \( \text{FO}(\Sigma) \) and an interpretation \((w, I)\) of \( \varphi \) is defined by:

\[
\begin{align*}
(w, I) &\models Q_a(x) \quad \text{iff} \quad w[I(x)] = a \\
(w, I) &\models x < y \quad \text{iff} \quad I(x) < I(y) \\
(w, I) &\models \neg \varphi \quad \text{iff} \quad (w, I) \not\models \varphi \\
(w, I) &\models \varphi_1 \lor \varphi_2 \quad \text{iff} \quad (w, I) \models \varphi_1 \text{ or } (w, I) \models \varphi_2 \\
(w, I) &\models \exists x \varphi \quad \text{iff} \quad |w| \geq 1 \text{ and some } i \in \{1, \ldots, |w|\} \text{ satisfies } (w, I[i/x]) \models \varphi
\end{align*}
\]

where \( w[i] \) is the letter of \( w \) at position \( i \), and \( I[i/x] \) is the mapping that assigns \( i \) to \( x \) and otherwise coincides with \( I \). (Notice that \( I \) may not assign any value to \( x \).) If \((w, I) \models \varphi\) we say that \((w, I)\) is a model of \( \varphi \). Two formulas are equivalent if they have the same models.
It follows easily from this definition that if two interpretations \((w, J_1)\) and \((w, J_2)\) of \(\varphi\) differ only in the positions assigned by \(J_1\) and \(J_2\) to bounded variables, then either both interpretations are models of \(\varphi\), or none of them is. In particular, whether an interpretation \((w, J)\) of a sentence is a model or not depends only on \(w\), not on \(J\).

We use some standard abbreviations:

\[
\forall x \varphi := \neg \exists x \neg \varphi \\
\varphi_1 \land \varphi_2 := \neg (\neg \varphi_1 \lor \neg \varphi_2) \\
\varphi_1 \rightarrow \varphi_2 := \neg \varphi_1 \lor \varphi_2
\]

Notice that according to the definition of the satisfaction relation the empty word \(\epsilon\) satisfies no formulas of the form \(\exists x \varphi\), and all formulas of the form \(\forall x \varphi\). While this causes no problems for our purposes, it is worth noticing that in other contexts it may lead to complications. For instance, the formulas \(\exists x Q_a(x)\) and \(\forall y \exists x Q_a(x)\) do not hold for exactly the same words, because the empty word satisfies the second, but not the first. Further useful abbreviations are:

\[
\begin{align*}
\text{first}(x) & := \neg \exists y \; y < x & \text{“}x\text{ is the first position}\text{”} \\
\text{last}(x) & := \neg \exists y \; x < y & \text{“}x\text{ is the last position}\text{”} \\
y = x + 1 & := x < y \land \neg \exists z (x < z \land z < y) & \text{“}y\text{ is the successor position of }x\text{“} \\
y = x + 2 & := \exists z (z = x + 1 \land y = z + 1) \\
y = x + (k + 1) & := \exists z (z = x + k \land y = z + 1)
\end{align*}
\]

Example 9.4 Some examples of properties expressible in the logic:

- “The last letter is a \(b\) and before it there are only \(a\)’s.”
  \[
  \exists x Q_b(x) \land \forall x (\text{last}(x) \rightarrow Q_b(x) \land \neg \text{last}(x) \rightarrow Q_a(x))
  \]

- “Every \(a\) is immediately followed by a \(b\).”
  \[
  \forall x (Q_a(x) \rightarrow \exists y (y = x + 1 \land Q_b(y)))
  \]

- “Every \(a\) is immediately followed by a \(b\), unless it is the last letter.”
  \[
  \forall x (Q_a(x) \rightarrow \forall y (y = x + 1 \rightarrow Q_b(y)))
  \]

- “Between every \(a\) and every later \(b\) there is a \(c\).”
  \[
  \forall x \forall y (Q_a(x) \land Q_b(y) \land x < y \rightarrow \exists z (x < z \land z < y \land Q_c(z)))
  \]

\[
\square
\]
9.1.1 Expressive power of $FO(\Sigma)$

Once we have defined which words satisfy a sentence, we can associate to a sentence the set of words satisfying it.

**Definition 9.5** The language $L(\varphi)$ of a sentence $\varphi \in FO(\Sigma)$ is the set $L(\varphi) = \{w \in \Sigma^* \mid w \models \varphi\}$. We also say that $\varphi$ expresses $L(\varphi)$. A language $L \subseteq \Sigma^*$ is $FO$-definable if $L = L(\varphi)$ for some formula $\varphi$ of $FO(\Sigma)$.

The languages of the properties in the example are $FO$-definable by definition. To get an idea of the expressive power of $FO(\Sigma)$, we prove a theorem characterizing the $FO$-definable languages in the case of a 1-letter alphabet $\Sigma = \{a\}$. In this simple case we only have one predicate $Q_a(x)$, which is always true in every interpretation. So every formula is equivalent to a formula without any occurrence of $Q_a(x)$. For example, the formula $\exists y (Q_a(x) \land y < x)$ is equivalent to $\exists y y < x$.

We prove that a language over a one-letter alphabet is $FO$-definable if and only if it is finite or co-finite; finally, we prove that 1-letter languages are $QF$-definable iff they are finite or co-finite; finally, we prove that 1-letter languages are $FO$-definable iff they are $QF$-definable.

For the definition of $QF$ we need some more macros whose intended meaning should be easy to guess:

$$
\begin{align*}
&x + k < y := \exists z \,(z = x + k \land z < y) \\
&x < y + k := \exists z \,(z = y + k \land x < z) \\
&k \leq \text{last} := \forall x \,(\text{last}(x) \rightarrow x \geq k)
\end{align*}
$$

In these macros $k$ is a constant, that is, $k \leq \text{last}$ stands for the infinite family of macros $1 \leq \text{last}, 2 \leq \text{last}, 3 \leq \text{last} \ldots$. Macros like $k > x$ or $x + k > y$ are defined similarly.

**Definition 9.6** The logic $QF$ (for quantifier-free) is the fragment of $FO(\{a\})$ with syntax

$$
\begin{align*}
f := x \approx k \mid x \approx y + k \mid k \approx \text{last} \mid f_1 \lor f_2 \mid f_1 \land f_2
\end{align*}
$$

where $\approx \in \{<,>\}$ and $k \in \mathbb{N}$.

**Proposition 9.7** A language over a 1-letter alphabet is $QF$-definable iff it is finite or co-finite.

**Proof:** ($\Rightarrow$): Let $f$ be a sentence of $QF$. Since $QF$ does not have quantifiers, $f$ does not contain any occurrence of a variable, and so it is a positive (i.e., negation-free) boolean combination of formulas of the form $k < \text{last}$ or $k \geq \text{last}$. We proceed by induction on the structure of $f$. If $f = k < \text{last}$, then $L(\varphi)$ is co-finite, and if $f = k \geq \text{last}$, then $L(\varphi)$ is finite. If $f = f_1 \lor f_2$, then by induction hypothesis $L(f_1)$ and $L(f_2)$ are finite or co-finite; if $L(f_1)$ and $L(f_2)$ are finite, then so is $L(f)$, and otherwise $L(f)$ is co-finite. The case $f = f_1 \land f_2$ is similar.
(⇐): A finite language \{a^k, \ldots, a^n\} is expressed by the formula \((last > k_1 - 1 \land last < k_1 + 1) \lor \ldots \lor (last > k_1 - 1 \land last < k_1 + 1)\). To express a co-finite language, it suffices to show that for every formula \(f\) of QF expressing a language \(L\), there is another formula \(\bar{f}\) expressing the language \(\overline{L}\). This is easily proved by induction on the structure of the formula.

**Theorem 9.8** Every formula \(\varphi\) of FO(\{a\}) is equivalent to a formula \(f\) of QF.

**Proof:** Sketch. By induction on the structure of \(\varphi\). If \(\varphi(x, y) = x < y\), then \(\varphi \equiv y < x + 0\). If \(\varphi = \neg \psi\), the result follows from the induction hypothesis and the fact that negations can be removed using De Morgan’s rules and equivalences like \(\neg(x < y + k) \equiv x \geq y + k\). If \(\varphi = \varphi_1 \lor \varphi_2\), the result follows directly from the induction hypothesis. Consider now the case \(\varphi = \exists x \psi\). By induction hypothesis, \(\psi\) is equivalent to a formula \(f\) of QF, and we can assume that \(f\) is in disjunctive normal form, say \(f = D_1 \lor \ldots \lor D_n\). Then \(\varphi \equiv \exists x D_1 \lor \exists x D_2 \lor \ldots \lor \exists x D_n\), and so it suffices to find a formula \(f_i\) of QF equivalent to \(\exists x D_i\).

The formula \(f_i\) is a conjunction of formulas containing all conjuncts of \(D_i\) with no occurrence of \(x\), plus other conjuncts obtained as follows. For every lower bound \(x < t_1\) of \(D_i\), where \(t_1 = k_1\) or \(t_1 = x_1 + k_1\), and every upper bound of the form \(x > t_2\), where \(t_2 = k_1\) or \(t_2 = x_1 + k_1\) we add to \(f_i\) a conjunct equivalent to \(t_2 + 1 < t_1\). For instance, \(y + 7 < x\) and \(x < z + 3\) we add \(y + 5 < z\). It is easy to see that \(f_i \equiv \exists x D_i\).

**Corollary 9.9** The language \(\text{Even} = \{a^{2n} \mid n \geq 0\}\) is not first-order expressible.

These results show that first-order logic cannot express all regular languages, not even over a 1-letter alphabet. For this reason we now introduce monadic second-order logic.

### 9.2 Monadic Second-Order Logic on Words

Monadic second-order logic extends first-order logic with variables \(X, Y, Z, \ldots\) ranging over sets of positions, and with predicates \(x \in X\), meaning “position \(x\) belongs to the set \(X\).” \(^1\) It is allowed to quantify over both kinds of variables. Before giving a formal definition, let us informally see how this extension allows to describe the language \(\text{Even}\). The formula states that the last position belongs to the set of even positions. A position belongs to this set iff it is the second position, or the second successor of another position in the set.

The following formula states that \(X\) is the set of even positions:

\[
\text{second}(x) := \exists y \ (\text{first}(y) \land x = y + 1) \\
\text{Even}(X) := \forall x \ (x \in X \leftrightarrow (\text{second}(x) \lor \exists y \ (x = y + 2 \land y \in X)))
\]

For the complete formula, we observe that the word has even length if its last position is even:

\[
\text{EvenLength} := \exists X \ (\text{Even}(X) \land \forall x \ (\text{last}(x) \rightarrow x \in X))
\]

\(^1\)More generally, second-order logic allows for variables ranging over relations of arbitrary arity. The monadic fragment only allows arity 1, which corresponds to sets.
We now define the formal syntax and semantics of the logic.

**Definition 9.10** Let $X_1 = \{x, y, z, \ldots\}$ and $X_2 = \{X, Y, Z, \ldots\}$ be two infinite sets of first-order and second-order variables. Let $\Sigma = \{a, b, c, \ldots\}$ be a finite alphabet. The set $\text{MSO}(\Sigma)$ of monadic second-order formulas over $\Sigma$ is the set of expressions generated by the grammar:

$$\varphi ::= Q_a(x) \mid x < y \mid x \in X \mid \neg \varphi \mid \varphi \lor \varphi \mid \exists x \varphi \mid \exists X \varphi$$

An interpretation of a formula $\varphi$ is a pair $(w, I)$ where $w \in \Sigma^*$, and $I$ is a mapping that assigns every free first-order variable $x$ a position $I(x) \in \{1, \ldots, |w|\}$ and every free second-order variable $X$ a set of positions $I(X) \subseteq \{1, \ldots, |w|\}$. (The mapping may also assign positions to other variables.)

The satisfaction relation $(w, I) \models \varphi$ between a formula $\varphi$ of $\text{MSO}(\Sigma)$ and an interpretation $(w, I)$ of $\varphi$ is defined as for $\text{FO}(\Sigma)$, with the following additions:

$$(w, I) \models x \in X \iff I(x) \in I(X)$$

$$(w, I) \models \exists X \varphi \iff |w| > 0 \text{ and some } S \subseteq \{1, \ldots, |w|\} \text{ satisfies } (w, I[S/X]) \models \varphi$$

where $I[S/X]$ is the interpretation that assigns $S$ to $X$ and otherwise coincides with $I$ — whether $I$ is defined for $X$ or not. If $(w, I) \models \varphi$ we say that $(w, I)$ is a model of $\varphi$. Two formulas are equivalent if they have the same models. The language $L(\varphi)$ of a sentence $\varphi \in \text{MSO}(\Sigma)$ is the set $L(\varphi) = \{w \in \Sigma^* \mid w \models \varphi\}$. A language $L \subseteq \Sigma^*$ is $\text{MSO}$-definable if $L = L(\varphi)$ for some formula $\varphi \in \text{MSO}(\Sigma)$.

Notice that in this definition the set $S$ may be empty. So, for instance, any interpretation that assigns the empty set to $X$ is a model of the formula $\exists X \forall x \neg (x \in X)$.

We use the standard abbreviations

$$\forall x \in X \varphi ::= \forall x(x \in X \rightarrow \varphi) \quad \exists x \in X \varphi ::= \exists x(x \in X \land \varphi)$$

### 9.2.1 Expressive power of $\text{MSO}(\Sigma)$

We show that the languages expressible in monadic second-order logic are exactly the regular languages. We start with an example.

**Example 9.11** Let $\Sigma = \{a, b, c, d\}$. We construct a formula of $\text{MSO}(\Sigma)$ expressing the regular language $c^*(ab)^*d^*$. The membership predicate of the language can be informally formulated as follows:

There is a block of consecutive positions $X$ such that: before $X$ there are only $c$’s; after $X$ there are only $d$’s; in $X$ $b$’s and $a$’s alternate; the first letter in $X$ is an $a$ and the last letter is a $b$.

The predicate is a conjunction of predicates. We give formulas for each of them.
• “X is a block of consecutive positions.”

\[
\text{Cons}(X) := \forall x \in X \forall y \in X \ (x < y \rightarrow (\forall z \ (x < z \land z < y) \rightarrow z \in X))
\]

• “x lies before/after X.”

\[
\text{Before}(x, X) := \forall y \in X \ x < y \quad \text{After}(x, X) := \forall y \in X \ y < x
\]

• “Before X there are only c’s.”

\[
\text{Before\_only\_c}(X) := \forall x \ \text{Before}(x, X) \rightarrow Q_c(x)
\]

• “After X there are only d’s.”

\[
\text{After\_only\_d}(X) := \forall x \ \text{After}(x, X) \rightarrow Q_d(x)
\]

• “a’s and b’s alternate in X.”

\[
\text{Alternate}(X) := \forall x \in X \ (Q_a(x) \rightarrow \forall y \in X \ (y = x + 1 \rightarrow Q_b(y)) \land \ Q_b(x) \rightarrow \forall y \in X \ (y = x + 1 \rightarrow Q_a(y)))
\]

• ”The first letter in X is an a and the last is a b.”

\[
\text{First\_a}(X) := \forall x \in X \ \forall y \ (y < x \rightarrow \neg y \in X) \rightarrow Q_a(x)
\]

\[
\text{Last\_b}(X) := \forall x \in X \ \forall y \ (y > x \rightarrow \neg y \in X) \rightarrow Q_a(x)
\]

Putting everything together, we get the formula

\[
\exists X ( \text{Cons}(X) \land \text{Before\_only\_c}(X) \land \text{After\_only\_d}(X) \land \\
\text{Alternate}(X) \land \text{First\_a}(X) \land \text{Last\_b}(X))
\]

Notice that the empty word is a model of the formula. because the empty set of positions satisfies all the conjuncts.

Let us now directly prove one direction of the result.

**Proposition 9.12** If \( L \subseteq \Sigma^* \) is regular, then \( L \) is expressible in MSO(\( \Sigma \)).
We express these properties through formulas. For every \( a \in \Sigma \), we construct a formula \( \varphi_A \) such that for every \( w \neq \epsilon \), \( w \models \varphi_A \) iff \( w \in L(A) \). If \( \epsilon \in L(A) \), then we can extend the formula to \( \varphi_A \lor \varphi'_A \), where \( \varphi'_A \) is only satisfied by the empty word (e.g. \( \varphi'_A = \forall x \ x < x \)).

We start with some notations. Let \( w = a_1 \ldots a_m \) be a word over \( \Sigma \), and let

\[
P_q = \{ i \in [1, \ldots, m] \mid \hat{\delta}(q_0, a_0 \ldots a_i) = q \} .
\]

In words, \( i \in P_q \) iff \( A \) is in state \( q \) immediately after reading the letter \( a_i \). Then \( A \) accepts \( w \) iff \( m \in \bigcup_{q \in F} P_q \).

Assume we were able to construct a formula \( \text{Visits}(X_0, \ldots X_n) \) with free variables \( X_0, \ldots X_n \) such that \( \mathfrak{I}(X_i) = P_{q_i} \) holds for every model \( (w, \mathfrak{I}) \) and for every \( 0 \leq i \leq n \). In words, \( \text{Visits}(X_0, \ldots X_n) \) is only true when \( X_i \) takes the value \( P_{q_i} \) for every \( 0 \leq i \leq n \). Then \( (w, \mathfrak{I}) \) would be a model of

\[
\psi_A := \exists X_0 \ldots \exists X_n \text{Visits}(X_0, \ldots X_n) \land \exists x \left( \text{last}(x) \land \bigvee_{q \in F} x \in X_i \right)
\]

iff \( w \) has a last letter, and \( w \in L \). So we could take

\[
\varphi_A := \left\{ \begin{array}{ll}
\psi_A & \text{if } q_0 \notin F \\
\psi_A \lor \forall x \ x < x & \text{if } q_0 \in F
\end{array} \right.
\]

Let us now construct the formula \( \text{Visits}(X_0, \ldots X_n) \). The sets \( P_q \) are the unique sets satisfying the following properties:

(a) \( 1 \in P_{\hat{\delta}(q_0, a_1)} \), i.e., after reading the letter at position 1 the DFA is in state \( \hat{\delta}(q_0, a_1) \);

(b) every position \( i \) belongs to exactly one \( P_q \), i.e., the \( P_q \)'s build a partition of the set positions; and

(c) if \( i \in P_q \) and \( \hat{\delta}(q, a_{i+1}) = q' \) then \( i + 1 \in P_{q'} \), i.e., the \( P_q \)'s “respect” the transition function \( \hat{\delta} \).

We express these properties through formulas. For every \( a \in \Sigma \), let \( q_{i_a} = \hat{\delta}(q_0, a) \). The formula for (a) is:

\[
\text{Init}(X_0, \ldots , X_n) = \exists x \left( \text{first}(x) \land \bigvee_{a \in \Sigma} (Q_a(x) \land x \in X_{i_a}) \right)
\]

(in words: if the letter at position 1 is \( a \), then the position belongs to \( X_{i_a} \)).

Formula for (b):

\[
\text{Partition}(X_0, \ldots , X_n) = \forall x \left( \bigvee_{i=0}^n x \in X_i \land \bigwedge_{i,j=0}^n (x \in X_i \rightarrow x \notin X_j) \right)
\]
9.2. MONADIC SECOND-ORDER LOGIC ON WORDS

Formula for (c):

\[ \text{Respect}(X_0, \ldots, X_n) = \forall x \forall y \left( \begin{array}{c} y = x + 1 \rightarrow \\ \bigvee_{a \in \Sigma} \bigwedge_{i, j \in [0, \ldots, n]} \left( x \in X_i \land Q_a(x) \land y \in X_j \right) \end{array} \right) \]

Altogether we get

\[ \text{Visits}(X_0, \ldots X_n) := \text{Init}(X_0, \ldots, X_n) \land \text{Partition}(X_0, \ldots, X_n) \land \text{Respect}(X_0, \ldots, X_n) \]

It remains to prove that MSO-definable languages are regular. Given a sentence \( \varphi \in MSO(\Sigma) \) show that \( L(\varphi) \) is regular by induction on the structure of \( \varphi \). However, since the subformulas of a sentence are not necessarily sentences, the language defined by the subformulas of \( \varphi \) is not defined. We correct this. Recall that the interpretations of a formula are pairs \((w, I)\) where \( I \) assigns positions to the free first-order variables and sets of positions to the free second-order variables. For example, if \( \Sigma = \{a, b\} \) and if the free first-order and second-order variables of the formula are \( x, y \) and \( X, Y \), respectively, then two possible interpretations are

\[
\begin{pmatrix}
    x & \mapsto & 1 \\
    y & \mapsto & 3 \\
    X & \mapsto & \{2, 3\} \\
    Y & \mapsto & \{1, 2\}
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
    x & \mapsto & 2 \\
    y & \mapsto & 1 \\
    X & \mapsto & 0 \\
    Y & \mapsto & \{1\}
\end{pmatrix}
\]

Given an interpretation \((w, I)\), we can encode each assignment \( x \mapsto k \) or \( X \mapsto \{k_1, \ldots, k_l\} \) as a bitstring of the same length as \( w \): the string for \( x \mapsto k \) contains exactly a 1 at position \( k \), and 0’s everywhere else; the string for \( X \mapsto \{k_1, \ldots, k_l\} \) contains 1’s at positions \( k_1, \ldots, k_l \), and 0’s everywhere else. After fixing an order on the variables, an interpretation \((w, I)\) can then be encoded as a tuple \((w, w_1, \ldots, w_n)\), where \( n \) is the number of variables, \( w \in \Sigma^* \), and \( w_1, \ldots, w_n \in \{0, 1\}^* \). Since all of \( w, w_1, \ldots, w_n \) have the same length, we can as in the case of transducers look at \((w, w_1, \ldots, w_n)\) as a word over the alphabet \( \Sigma \times \{0, 1\}^n \). For the two interpretations above we get the encodings

\[
\begin{array}{cccc}
    a & a & b & b & a \\
    x & 1 & 0 & 0 & x & 0 & 1 \\
    y & 0 & 0 & 1 & y & 1 & 0 \\
    X & 0 & 1 & 1 & X & 0 & 0 \\
    Y & 1 & 1 & 0 & Y & 1 & 0 \\
\end{array}
\]

corresponding to the words.
CHAPTER 9. AUTOMATA AND LOGIC

\[
\begin{bmatrix}
 a & a & b \\
 1 & 0 & 0 \\
 0 & 0 & 1 \\
 0 & 1 & 1 \\
 1 & 1 & 0 \\
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
 b & a \\
 0 & 1 \\
 0 & 0 \\
 1 & 0 \\
\end{bmatrix}
\over \Sigma \times \{0, 1\}^4
\]

**Definition 9.13** Let \( \varphi \) be a formula with \( n \) free variables, and let \((w, I)\) be an interpretation of \( \varphi \). We denote by \( \text{enc}(w, I) \) the word over the alphabet \( \Sigma \times \{0, 1\}^n \) described above. The language of \( \varphi \) is \( L(\varphi) = \{ \text{enc}(w, I) \mid (w, I) \models \varphi \} \).

Now that we have associated to every formula \( \varphi \) a language (whose alphabet depends on the free variables), we prove by induction on the structure of \( \varphi \) that \( L(\varphi) \) is regular. We do so by exhibiting automata (actually, transducers) accepting \( L(\varphi) \). For simplicity we assume \( \Sigma = \{a, b\} \), and denote by \( \text{free}(\varphi) \) the set of free variables of \( \varphi \).

- \( \varphi = Q_a(x) \). Then \( \text{free}(\varphi) = x \), and the interpretations of \( \varphi \) are encoded as words over \( \Sigma \times \{0, 1\} \). The language \( L(\varphi) \) is given by
  \[
  L(\varphi) = \left\{ \begin{bmatrix} a_1 & \cdots & a_k \\ b_1 & \cdots & b_k \end{bmatrix} \mid k \geq 0, \ a_i \in \Sigma \text{ and } b_i \in \{0, 1\} \text{ for every } i \in \{1, \ldots, k\}, \text{ and } b_i = 1 \text{ for exactly one index } i \in \{1, \ldots, k\} \text{ such that } a_i = a \right\}
  \]
  and is recognized by
  \[
  \begin{array}{c}
  \text{\includegraphics{automaton1.png}}
  \end{array}
  \]

- \( \varphi = x < y \). Then \( \text{free}(\varphi) = \{x, y\} \), and the interpretations of \( \phi \) are encoded as words over \( \Sigma \times \{0, 1\}^2 \). The language \( L(\varphi) \) is given by
  \[
  L(\varphi) = \left\{ \begin{bmatrix} a_1 & \cdots & a_k \\ b_1 & \cdots & b_k \\ c_1 & \cdots & c_k \end{bmatrix} \mid k \geq 0, \ a_i \in \Sigma \text{ and } b_i, c_i \in \{0, 1\} \text{ for every } i \in \{1, \ldots, k\}, \ b_i = 1 \text{ for exactly one index } i \in \{1, \ldots, k\}, \ c_j = 1 \text{ for exactly one index } j \in \{1, \ldots, k\}, \text{ and } i < j \right\}
  \]
  and is recognized by
  \[
  \begin{array}{c}
  \text{\includegraphics{automaton2.png}}
  \end{array}
  \]
9.2. MONADIC SECOND-ORDER LOGIC ON WORDS

\[ \phi = x \in X. \] Then \( \text{free}(\phi) = \{x, X\} \), and interpretations are encoded as words over \( \Sigma \times \{0, 1\}^2 \).

The language \( L(\phi) \) is given by

\[
L(\phi) = \left\{ \begin{bmatrix} a_1 & \ldots & a_k \\ b_1 & \ldots & b_k \\ c_1 & \ldots & c_k \end{bmatrix} \mid k \geq 0, \ a_i \in \Sigma \text{ and } b_i, c_i \in \{0, 1\} \text{ for every } i \in \{1, \ldots, k\}, \ b_i = 1 \text{ for exactly one index } i \in \{1, \ldots, k\}, \text{ and for every } i \in \{1, \ldots, k\}, \text{ if } b_i = 1 \text{ then } c_i = 1 \right\}
\]

and is recognized by

\[
\begin{bmatrix} a_1 & b_1 & a_2 & b_2 & a_3 \end{bmatrix}
\begin{bmatrix} a_1 & b_1 & a_2 & b_2 & a_3 \end{bmatrix}
\begin{bmatrix} a_1 & b_1 & a_2 & b_2 & a_3 \end{bmatrix}
\begin{bmatrix} a_1 & b_1 & a_2 & b_2 & a_3 \end{bmatrix}
\begin{bmatrix} a_1 & b_1 & a_2 & b_2 & a_3 \end{bmatrix}
\begin{bmatrix} a_1 & b_1 & a_2 & b_2 & a_3 \end{bmatrix}
\]

• \( \phi = \neg \psi \). Then \( \text{free}(\phi) = \text{free}(\psi) \), and by induction hypothesis there exists an automaton \( A_\psi \) s.t. \( L(A_\psi) = L(\psi) \).

Observe that \( L(\phi) \) is not in general equal to \( L(\psi) \). To see why, consider for example the case \( \psi = Q_a(x) \) and \( \phi = \neg Q_a(x) \). The word

\[
\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}
\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}
\]

belongs neither to \( L(\psi) \) nor \( L(\phi) \), because it is not the encoding of any interpretation: the bitstring for \( x \) contains more than one 1. What holds is \( L(\phi) = L(\neg \psi) \cap \text{Enc}(\psi) \), where \( \text{Enc}(\psi) \) is the language of the encodings of all the interpretations of \( \psi \) (whether they are models of \( \psi \) or not). We construct an automaton \( A^\text{enc}_\psi \) recognizing \( \text{Enc}(\psi) \), and so we can take \( A_\phi = A_\psi \cap A^\text{enc}_\psi \).

Assume \( \psi \) has \( k \) first-order variables. Then a word belongs to \( \text{Enc}(\psi) \) iff each of its projections onto the 2nd, 3rd, \ldots, \((k+1)\)-th component is a bitstring containing exactly one 1. As states of \( A^\text{enc}_\psi \) we take all the strings \( \{0, 1\}^k \). The intended meaning of a state, say state 101 for the case \( k = 3 \), is “the automaton has already read the 1’s in the bitstrings of the first and third variables, but not yet read the 1 in the second.” The initial and final states are \( 0^k \) and \( 1^k \), respectively. The transitions are defined according to the intended meaning of the states. For instance, the automaton \( A^\text{enc}_{x < y} \) is

- \( \phi = x \in X \). Then \( \text{free}(\phi) = \{x, X\} \), and interpretations are encoded as words over \( \Sigma \times \{0, 1\}^2 \).
Observe that the number of states of $A^\text{enc}_\psi$ grows exponentially in the number of free variables. This makes the negation operation expensive, even when the automaton $A_\phi$ is deterministic.

- $\varphi = \varphi_1 \lor \varphi_2$. Then $\text{free}(\varphi) = \text{free}(\varphi_1) \cup \text{free}(\varphi_2)$, and by induction hypothesis there are automata $A_{\varphi_1}, A_{\varphi_2}$ such that $L(A_{\varphi_1}) = L(\varphi_1)$ and $L(A_{\varphi_2}) = L(\varphi_2)$.

If $\text{free}(\varphi_1) = \text{free}(\varphi_2)$, then we can take $A_\varphi = A_{\varphi_1} \cup A_{\varphi_2}$. But this need not be the case. If $\text{free}(\varphi_1) \neq \text{free}(\varphi_2)$, then $L(\varphi_1)$ and $L(\varphi_2)$ are languages over different alphabets $\Sigma_1, \Sigma_2$, or over the same alphabet, but with different intended meaning, and we cannot just compute their intersection. For example, if $\varphi_1 = Q_a(x)$ and $\varphi_2 = Q_b(y)$, then both $L(\varphi_1)$ and $L(\varphi_2)$ are languages over $\Sigma \times \{0, 1\}$, but the second component indicates in the first case the value of $x$, in the second the value of $y$.

This problem is solved by extending $L(\varphi_1)$ and $L(A_{\varphi_2})$ to languages $L_1$ and $L_2$ over $\Sigma \times \{0, 1\}^2$. In our example, the language $L_1$ contains the encodings of all interpretations $(w, \{x \mapsto n_1, y \mapsto n_2\})$ such that the projection $(w, \{x \mapsto n_1\})$ belongs to $L(Q_a(x))$, while $L_2$ contains the interpretations such that $(w, \{y \mapsto n_2\})$ belongs to $L(Q_b(y))$. Now, given the automaton $A_{Q_a(x)}$ recognizing $L(Q_a(x))$

we transform it into an automaton $A_1$ recognizing $L_1$.
After constructing $A_2$ similarly, take $A_\varphi = A_1 \cup A_2$.

- $\varphi = \exists x \psi$. Then $\text{free}(\varphi) = \text{free}(\psi) \setminus \{x\}$, and by induction hypothesis there is an automaton $A_\psi$ s.t. $L(A_\psi) = L(\psi)$. Define $A_{\exists x \psi}$ as the result of the projection operation, where we project onto all variables but $x$. The operation simply corresponds to removing in each letter of each transition of $A_\sigma$ the component for variable $x$. For example, the automaton $A_{\exists x \psi}$ is obtained by removing the second components in the automaton for $A_{\psi}$ shown above, yielding

![Automaton Diagram]

Observe that the automaton for $\exists x \psi$ can be nondeterministic even if the one for $\psi$ is deterministic, since the projection operation may map different letters into the same one.

- $\varphi = \exists X \varphi$. We proceed as in the previous case.

**Size of $A_\varphi$.** The procedure for constructing $A_\varphi$ proceeds bottom-up on the syntax tree of $\varphi$. We first construct automata for the atomic formulas in the leaves of the tree, and then proceed upwards: given automata for the children of a node in the tree, we construct an automaton for the node itself.

Whenever a node is labeled by a negation, the automaton for it can be exponentially bigger than the automaton for its only child. This yields an upper bound for the size of $A_\varphi$ equal to a tower of exponentials, where the height of the tower is equal to the largest number of negations in any path from the root of the tree to one of its leaves.

It can be shown that this very large upper bound is essentially tight: there are formulas for which the smallest automaton recognizing the same language as the formula reaches the upper bound. This means that MSO-logic allows to describe some regular languages in an extremely succinct form.

**Example 9.14** Consider the alphabet $\Sigma = \{a, b\}$ and the language $a^*b \subseteq \Sigma^*$, recognized by the NFA

![NFA Diagram]
We derive this NFA by giving a formula $\varphi$ such that $L(\varphi) = a^*b$, and then using the procedure described above. We shall see that the procedure is quite laborious. The formula states that the last letter is $b$, and all other letters are $a$’s.

$$\varphi = \exists x (\text{last}(x) \land Q_b(x)) \land \forall x (\neg \text{last}(x) \rightarrow Q_a(x))$$

We first bring $\varphi$ into the equivalent form

$$\psi = \exists x (\text{last}(x) \land Q_b(x)) \land \neg \exists x (\neg \text{last}(x) \land \neg Q_a(x))$$

We transform $\psi$ into an NFA. First, we compute an automaton for $\text{last}(x) = \neg \exists y x < y$. Recall that the automaton for $x < y$ is

![Automaton for $x < y$]

Applying the projection operation, we get following automaton for $\exists y x < y$

![Automaton for $\exists y x < y$]

Recall that computing the automaton for the negation of a formula requires more than complementing the automaton. First, we need an automaton recognizing $\text{Enc}(\exists y x < y)$.

![Automaton for $\text{Enc}(\exists y x < y)$]

Second, we determinize and complement the automaton for $\exists y x < y$:

![Automaton for $\exists y x < y$ determinized and complemented]
And finally, we compute the intersection of the last two automata, getting

\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]
\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
1 & 1 \\
\end{array}
\]
\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]

whose last state is useless and can be removed, yielding the following NFA for \text{last}(x):

\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]
\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
1 & 1 \\
\end{array}
\]

Next we compute an automaton for \(\exists x (\text{last}(x) \land Q_b(x))\), the first conjunct of \(\psi\). We start with an NFA for \(Q_b(x)\)

\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]
\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]

The automaton for \(\exists x (\text{last}(x) \land Q_b(x))\) is the result of intersecting this automaton with the NFA for \(\text{last}(x)\) and projecting onto the first component. We get

\[
\begin{array}{c|c}
a, b \\
\end{array}
\]

Now we compute an automaton for \(\neg \exists x (\neg \text{last}(x) \land \neg Q_a(x))\), the second conjunct of \(\psi\). We first obtain an automaton for \(\neg Q_a(x)\) by intersecting the complement of the automaton for \(Q_a(x)\) and the automaton for \(Enc(Q_a(x))\). The automaton for \(Q_a(x)\) is

\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]
\[
\begin{array}{c|c}
\text{[a]} & \text{[b]} \\
0 & 0 \\
\end{array}
\]
and after determinization and complementation we get

For the automaton recognizing $\text{Enc}(Q_a(x))$, notice that $\text{Enc}(Q_a(x)) = \text{Enc}(\exists y \ x < y)$, because both formulas have the same free variables, and so the same interpretations. But we have already computed an automaton for $\text{Enc}(\exists y \ x < y)$, namely

The intersection of the last two automata yields a three-state automaton for $\neg Q_a(x)$, but after eliminating a useless state we get

Notice that this is the same automaton we obtained for $Q_b(x)$, which is fine, because over the alphabet \{a, b\} the formulas $Q_b(x)$ and $\neg Q_a(x)$ are equivalent.

To compute an automaton for $\neg \text{last}(x)$ we just observe that $\neg \text{last}(x)$ is equivalent to $\exists y \ x < y$, for which we have already compute an NFA, namely
Intersecting the automata for $\neg$last$(x)$ and $\neg Q_a(x)$, and subsequently projecting onto the first component, we get an automaton for $\exists x (\neg$last$(x) \land \neg Q_a(x))$:

Determinizing, complementing, and removing a useless state yields the following NFA for $\neg \exists x (\neg$last$(x) \land \neg Q_a(x))$:

Summarizing, the automata for the two conjuncts of $\psi$ are

whose intersection yields a 3-state automaton, which after removal of a useless state becomes

ending the derivation.

Exercises

**Exercise 76** Characterize the languages described by the following formulas and give a corresponding automaton:

1. $\exists x \text{first}(x)$
2. $\forall x \text{first}(x)$
3. $\neg \exists x \exists y (x < y \land Q_a(x) \land Q_b(y)) \land \forall x (Q_b(x) \rightarrow \exists y x < y \land Q_a(y)) \land \exists x \neg \exists y (x < y \land Q_a(x))$
Exercise 77 Give a defining MSO-formula, an automaton, and a regular expression for the following languages over \{a, b\}.

- The set of words of even length and containing only a’s or only b’s.
- The set of words, where between each two b’s with no other b in between there is a block of an odd number of letters a.
- The set of words with odd length and an odd number of occurrences of a.

Exercise 78 For every \(n \geq 1\), give a FO-formula of polynomial length in \(n\) abbreviating \(y = x + 2^n\). (Notice that the abbreviation \(y = x + k\) of page 9.1 has length \(O(k)\), and so it cannot be directly used.) Use it to give another FO-formula \(\varphi_n\), also of of polynomial length in \(n\), for the language \(L_n = \{ww \in \{a, b\}^* \mid |w| = 2^n\}\).

Remark: Since the minimal DFA for \(L_n\) has \(2^{2^n}\) states (Exercise 10), this shows that the number of states of a minimal automaton equivalent to a given FO-formula may be double exponential in the length of the formula.

Exercise 79 MSO over a unary alphabet can be used to automatically prove some simple properties of the natural numbers. Consider for instance the following property: every finite set of natural numbers has a minimal element\(^2\). It is easy to see that this property holds iff the formula

\[
\forall Z \exists x \forall y \ (y \in Z \rightarrow (x \leq y \wedge x \in Z))
\]

is a tautology, i.e., if it is satisfied by every word. Construct an automaton for the formula, and check that it is universal.

Exercise 80 Give formulas \(\varphi_1, \ldots, \varphi_4\) for the following abbreviations:

\[
\begin{align*}
sing(X) &:= \varphi_1 \quad \text{“}X\text{ is a singleton, i.e., } X\text{ contains one element}\text{”} \\
X \subseteq Y &:= \varphi_2 \quad \text{“}X\text{ is a subset of } Y\text{”} \\
X \subseteq Q_a &:= \varphi_3 \quad \text{“every position of } X\text{ contains an } a\text{”} \\
X < Y &:= \varphi_4 \quad \text{“}X\text{ and } Y\text{ are singletons } X = \{x\} \text{ and } Y = \{y\} \text{ satisfying } x < y\text{”}
\end{align*}
\]

Exercise 81 Express addition in MSO(\(\{a\}\)). More precisely, find a formula \(+(X, Y, Z)\) of MSO(\(\{a\}\)) that is true iff \(x + y = z\), where \(x, y, z\) are the numbers encoded by the sets \(X, Y, Z\), respectively, in \(lsbf\) encoding. You may use any abbreviation defined in the chapter.

\(^2\)Of course, this also holds for every infinite set, but we cannot prove it using MSO over finite words.
Exercise 82  The nesting depth $d(\varphi)$ of a formula $\varphi$ of $FO(\{a\})$ is defined inductively as follows:

- $d(Q_a(x)) = d(x < y) = 0$;
- $d(\neg \varphi) = d(\varphi)$, $d(\varphi_1 \lor \varphi_2) = \max\{d(\varphi_1), d(\varphi_2)\}$; and
- $d(\exists x \varphi) = 1 + d(\varphi)$.

Prove that every formula $\varphi$ of $FO(\{a\})$ of nesting depth $n$ is equivalent to a formula $f$ of $QF$ having the same free variables as $\varphi$, and such that every constant $k$ appearing in $f$ satisfies $k \geq 2^n$.

*Hint:* The proof is similar to that of Theorem 9.8. The difficult case is the one where $\varphi$ has the form $\exists x \psi$ and $\psi$ is a conjunction. Define $f$ as the following conjunction. All conjuncts of $\psi$ not containing $x$ are also conjuncts of $f$; for every conjunct of $D$ of the form $x \geq k$ or $x \geq y + k$, $f$ contains a conjunct $\text{last} \geq k$; for every two conjuncts of $D$ containing $x$, $f$ contains a conjunct obtained by “quantifying $x$ away”: for example, if the conjuncts are $x \geq k_1$ and $y \geq x + k_2$, then $f$ has the conjunct $y \geq k_1 + k_2$. Since the constants in the new conjuncts are the sum of two old constants, the new constants are bounded by $2 \cdot 2^d = 2^{d+1}$. 
Chapter 10

Applications III: Presburger Arithmetic

Presburger arithmetic is a logical language for expressing properties of numbers by means of addition and comparison. A typical example of such a property is “$x + 2y > 2z$ and $2x - 3z = 4y$”. The property is satisfied by some triples $(n_x,n_y,n_z)$ of natural numbers, like $(4, 2, 0)$ and $(8, 1, 4)$, but not by others, like $(6, 0, 4)$ or $(2, 2, 4)$. Valuations satisfying the property are called solutions or models. We show how to construct for a given formula $\varphi$ of Presburger arithmetic an NFA $A_\varphi$ recognizing the solutions of $\varphi$. In Section 10.1 we introduce the syntax and semantics of Presburger arithmetic. Section 10.2 constructs a NFA recognizing all solutions over the natural numbers, and Section 10.3 a NFA recognizing all solutions over the integers.

10.1 Syntax and Semantics

Formulas of Presburger arithmetic are constructed out of an infinite set of variables $V = \{x, y, z, \ldots\}$ and the constants 0 and 1. The syntax of formulas is defined in three steps. First, the set of terms is inductively defined as follows:

- the symbols 0 and 1 are terms;
- every variable is a term;
- if $t$ and $u$ are terms, then $t + u$ is a term.

An atomic formula is an expression $t \leq u$, where $t$ and $u$ are terms. The set of Presburger formulas is inductively defined as follows:

- every atomic formula is a formula;
- if $\varphi_1, \varphi_2$ are formulas, then so are $\neg \varphi_1, \varphi_1 \lor \varphi_2$, and $\exists x \varphi_1$. 

As usual, variables within the scope of an existential quantifier are bounded, and otherwise free. Besides standard abbreviations like $\forall$, $\land$, $\to$, we also introduce:

$$n := \frac{1 + 1 + \ldots + 1}{n \text{ times}} \quad t \geq t' := t' \leq t$$

$$nx := \frac{x + x + \ldots + x}{n \text{ times}} \quad t = t' := t \leq t' \land t \geq t'$$

$$t < t' := t \leq t' \land \neg(t = t')$$

$$t > t' := t' < t$$

An interpretation $I$ is a function $I : V \rightarrow \mathbb{N}$. An interpretation $I$ is extended to terms in the natural way: $I(0) = 0$, $I(1) = 1$, and $I(t + u) = I(t) + I(u)$. The satisfaction relation $I \models \varphi$ for an interpretation $I$ and a formula $\varphi$ is inductively defined as follows, where $I[n/x]$ denotes the interpretation that assigns the number $n$ to the variable $x$, and the same numbers as $I$ to all other variables:

$$I \models t \leq u \iff I(t) \leq I(u)$$

$$I \models \neg \varphi_1 \iff \lnot I \models \varphi_1$$

$$I \models \varphi_1 \lor \varphi_2 \iff I \models \varphi_1 \text{ or } I \models \varphi_2$$

$$I \models \exists x \varphi \iff \text{there exists } n \geq 0 \text{ such that } I[n/x] \models \varphi$$

It is easy to see that whether $I$ satisfies $\varphi$ or not depends only on the values $I$ assigns to the free variables of $\varphi$ (i.e., if two interpretations assign the the same values to the free variables, then either both satisfy the formula, or none does). The solutions of $\varphi$ are the projection onto the free variables of $\varphi$ of the interpretations that satisfy $\varphi$. If we fix a total order on the set $V$ of variables and a formula $\varphi$ has $k$ free variables, then its set of solutions can be represented as a subset of $\mathbb{N}^k$, or as relation of arity $k$ over the universe $\mathbb{N}$. We call this subset the solution space of $\varphi$, and denote it by $\text{Sol}(\varphi)$.

**Example 10.1** The solution space of the formula $x - 2 \geq 0$ is the set $\{2, 3, 4, \ldots\}$. The free variables of the formula $\exists x (2x = y \land 2y = z)$ are $y$ and $z$. The solutions of the formula are the pairs $\{(2n, 4n) \mid n \geq 0\}$, where we assume that the first and second components correspond to the values of $y$ and $z$, respectively.

**Automata encoding natural numbers.** We use transducers to represent, compute and manipulate solution spaces of formulas. As in Section 6.1 of Chapter 6, we encode natural numbers as strings over $\{0, 1\}$ using the least-significant-bit-first encoding $\text{lsbf}$. If a formula has free variables $x_1, \ldots, x_k$, then its solutions are encoded as words over $\{0, 1\}^k$. For instance, the word

$$\begin{array}{c|c|c|c|}
  x_1 & 1 & 0 & 1 \\
  x_2 & 0 & 1 & 0 \\
  x_3 & 0 & 0 & 0 \\
\end{array}$$

encodes the solution $(3, 10, 0)$. The language of a formula $\phi$ is defined as

$$L(\phi) = \bigcup_{s \in \text{Sol}(\phi)} \text{lsbf}(s)$$
where \( lsbf(s) \) denotes the set of all encodings of the tuple \( s \) of natural numbers. In other words, \( L(\varphi) \) is the encoding of the relation \( Sol(\varphi) \).

## 10.2 An NFA for the Solutions over the Naturals.

Given a Presburger formula \( \varphi \), we construct a transducer \( A_\varphi \) such that \( L(A_\varphi) = L(\varphi) \). Recall that \( Sol(\varphi) \) is a relation over \( \mathbb{N} \) whose arity is given by the number of free variables of \( \varphi \). The last section of Chapter 6 implements operations relations of arbitrary arity. These operations can be used to compute the solution space of the negation of a formula, the disjunction of two formulas, and the existential quantification of two formulas:

- The solution space of the negation of a formula with \( k \) free variables is the complement of its solution space with respect to the universe \( U^k \). In general, when computing the complement of a relation we have to worry about ensuring that the NFAs we obtain only accept words that encode some tuple of elements (i.e., some clean-up maybe necessary to ensure that automata do not accept words encoding nothing). For Presburger arithmetic this is not necessary, because in the \( lsbf \) encoding every word encodes some tuple of numbers.

- The solution space of a disjunction \( \varphi_1 \lor \varphi_2 \) where \( \varphi_1 \) and \( \varphi_2 \) have the same free variables is clearly the union of their solution spaces, and can be computed as \( \text{Union}(Sol(\varphi_1), Sol(\varphi_2)) \). If \( \varphi_1 \) and \( \varphi_2 \) have different sets \( V_1 \) and \( V_2 \) of free variables, then some preprocessing is necessary. Define \( Sol_{V_{1\cup V_2}}(\varphi_i) \) as the set of valuations of \( V_{1\cup V_2} \) whose projection onto \( V_1 \) belongs to \( Sol(\varphi_i) \). Transducers recognizing \( Sol_{V_{1\cup V_2}}(\varphi_i) \) for \( i = 1, 2 \) are easy to compute from transducers recognizing \( Sol(\varphi_i) \), and the solution space is \( \text{Union}(Sol_{V_{1\cup V_2}}(\varphi_1), Sol_{V_{1\cup V_2}}(\varphi_2)) \).

- The solution space of a formula \( \exists x \varphi \), where \( x \) is a free variable of \( \varphi \), is \( \text{Projection}_I(Sol(\varphi)) \), where \( I \) contains the indices of all variables with the exception of the index of \( x \).

It only remains to construct automata recognizing the solution space of atomic formulas. Consider an expression of the form

\[
\varphi = a_1x_1 + \ldots + a_nx_n \leq b
\]

where \( a_1, \ldots, a_n, b \in \mathbb{Z} \) (not \( \mathbb{N} \)). Since we allow negative integers as coefficients, for every atomic formula there is an equivalent expression in this form (i.e., an expression with the same solution space). For example, \( x \geq y + 4 \) is equivalent to \( -x + y \leq -4 \). Letting \( a = (a_1, \ldots, a_n) \), \( x = (x_1, \ldots, x_n) \), and denoting the scalar product of \( a \) and \( x \) by \( a \cdot x \) we write \( \varphi = a \cdot x \leq b \).

We construct a DFA for \( Sol(\varphi) \). The states of the DFA are integers. We choose transitions and final states of the DFA so that the following property holds:

\[
\text{State } q \in \mathbb{Z} \text{ recognizes the encodings of the tuples } c \in \mathbb{N}^n \text{ such that } a \cdot c \leq q. \tag{10.1}
\]

Given a state \( q \in \mathbb{Z} \) and a letter \( \zeta \in \{0, 1\}^n \), let us determine the target state \( q' \) of the transition \( q \xrightarrow{\zeta} q' \) of the DFA, where \( \zeta \in \{0, 1\}^n \). A word \( w' \in (\{0, 1\}^n)^* \) is accepted from \( q' \) iff the word \( \zeta w' \)
is accepted from \( q \). Since we use the \( \text{lsbf} \) encoding, the tuple of natural numbers encoded by \( \zeta w' \) is \( 2c' + \zeta \), where \( c' \in \mathbb{N}^n \) is the tuple encoded by \( w' \). So \( c' \in \mathbb{N}^n \) is accepted from \( q' \) iff \( 2c' + c_\zeta \) is accepted from \( q \). Therefore, in order to satisfy property 10.1 we must choose \( q' \) so that \( a \cdot c \leq q \) iff \( a \cdot (2c + c_\zeta) \leq q' \). A little arithmetic yields

\[
q' = \left\lfloor \frac{1}{2}(q - a \cdot \zeta) \right\rfloor
\]

and so we define the transition function of the DFA by

\[
\delta(q, \zeta) = \frac{1}{2}(q - a \cdot \zeta).
\]

For the final states we observe that a state is final iff it accepts the empty word iff it accepts the tuple \( (0, \ldots, 0) \in \mathbb{N}^n \). So in order to satisfy property 10.1 we must make state \( q \) final iff \( q \geq 0 \). As initial state we choose \( b \). This leads to the algorithm \( \text{AFtoDFA}(\varphi) \) of Table 10.1, where for clarity the state corresponding to an integer \( k \in \mathbb{Z} \) is denoted by \( s_k \).

\textbf{AFtoDFA}(\varphi)
\begin{itemize}
  \item \textbf{Input:} Atomic formula \( \varphi = a \cdot x \leq b \)
  \item \textbf{Output:} DFA \( A_\varphi = (Q, \Sigma, \delta, q_0, F) \) such that \( L(A_\varphi) = L(\varphi) \)
\end{itemize}

\begin{algorithmic}
  \STATE \( Q, \delta, F \leftarrow \emptyset; q_0 \leftarrow s_b \)
  \STATE \( W \leftarrow \{s_b\} \)
  \WHILE{\( W \neq \emptyset \)}
    \STATE \textbf{pick} \( s_k \) \textbf{from} \( W \)
    \STATE \textbf{add} \( s_k \) \textbf{to} \( Q \)
    \IF{\( k \geq 0 \)} \textbf{add} \( s_k \) \textbf{to} \( F \)
    \ENDIF
    \FOR{all \( \zeta \in \{0, 1\}^n \)}
      \STATE \( j \leftarrow \left\lfloor \frac{1}{2}(k - a \cdot \zeta) \right\rfloor \)
      \IF{\( s_j \notin Q \)} \textbf{add} \( s_j \) \textbf{to} \( W \)
      \ENDIF
    \ENDFOR
  \STATE \textbf{add} \( (s_k, \zeta, s_j) \) \textbf{to} \( \delta \)
\ENDWHILE
\end{algorithmic}

Table 10.1: Converting an atomic formula into a DFA recognizing the \( \text{lsbf} \) encoding of its solutions.

\textbf{Example 10.2} Consider the atomic formula \( 2x - y \leq 2 \). The DFA obtained by applying \( \text{AFtoDFA} \) to it is shown in Figure 10.1. The initial state is 2. Transitions leaving state 2 are given by

\[
\delta(2, \zeta) = \left\lfloor \frac{1}{2}(2 - 2 \cdot (2, -1) \cdot (\zeta_x, \zeta_y)) \right\rfloor = \left\lfloor \frac{1}{2}(2 - 2\zeta_x + \zeta_y) \right\rfloor
\]

and so we have \( 2 \xrightarrow{[0,0]} 1 \), \( 2 \xrightarrow{[0,1]} 1 \), \( 2 \xrightarrow{[1,0]} 0 \) and \( 2 \xrightarrow{[1,1]} 0 \). States 2, 1, and 0 are final. The DFA
10.2. AN NFA FOR THE SOLUTIONS OVER THE NATURALS.

Figure 10.1: DFA for the formula \(2x - y \leq 2\).

accepts, for example, the word
\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
\end{bmatrix}
\]
which encodes \(x = 12\) and \(y = 50\) and, indeed \(24 - 50 \leq 2\). If we remove the last letter then the word encodes \(x = 12\) and \(y = 18\), and is not accepted, which indeed corresponds to \(24 - 18 \not\leq 2\).

Consider now the formula \(x + y \geq 4\). We rewrite it as \(-x - y \leq -4\,\), and apply the algorithm. The resulting DFA is shown in Figure 10.2. The initial state is \(-4\). Transitions leaving \(-4\) are given by
\[
\delta(-4, \zeta) = \left\lfloor \frac{1}{2}(-4 - (-1, -1) \cdot (\zeta_x, \zeta_y)) \right\rfloor = \left\lfloor \frac{1}{2}(-4 + \zeta_x + \zeta_y) \right\rfloor
\]
and so we have \(-4 \xrightarrow{[0,0]} -2\), \(-4 \xrightarrow{[0,1]} -2\), \(-4 \xrightarrow{[1,0]} -2\) and \(-4 \xrightarrow{[1,1]} -1\). Notice that the DFA is not minimal, since states 0 and 1 can be merged.

Partial correctness of \(AFtoDFA\) is easily proved by showing that for every \(q \in \mathbb{Z}\) and every word \(w \in \{(0, 1)^n\}^*\), the state \(q\) accepts \(w\) if \(w\) encodes \(c \in \mathbb{N}^n\) satisfying \(a \cdot c \leq q\). The proof proceeds by induction of \(|w|\). For \(|w| = 0\) the result follows immediately from the definition of the final states, and for \(|w| > 0\) from the fact that \(\delta\) satisfies property 10.1 and from the induction hypothesis. Details are left to the reader. Termination of \(AFtoDFA\) also requires a proof: in principle the algorithm could keep generating new states forever. We show that this is not the case.

**Lemma 10.3** Let \(\varphi = a \cdot x \leq b\) and let \(s = \sum_{i=1}^{k} |a_i|\). All states \(s_j\) added to the worklist during the execution of \(AFtoDFA(\varphi)\) satisfy
\[
-|b| - s \leq j \leq |b| + s.
\]
**CHAPTER 10. APPLICATIONS III: PRESBURGER ARITHMETIC**

Figure 10.2: DFA for the formula $x + y \geq 4$.

**Proof:** The property holds for $s_b$, the first state added to the worklist. We show that, at any point in time, if all the states added to the worklist so far satisfy the property, then so does the next one. Let $s_j$ be this next state. Then there exists a state $s_k$ in the worklist and $\zeta \in \{0, 1\}^n$ such that $j = \lfloor \frac{1}{2}(k - a \cdot \zeta) \rfloor$. Since by assumption $s_k$ satisfies the property we have

$$-|b| - s \leq k \leq |b| + s$$

and so

$$\frac{-|b| - s - a \cdot \zeta}{2} \leq j \leq \frac{|b| + s - a \cdot \zeta}{2}$$

(10.2)

Now we manipulate the right and left ends of (10.2). A little arithmetic yields

$$\frac{-|b| - s - 2s}{2} \leq \frac{-|b| - s - a \cdot \zeta}{2} \leq \frac{|b| + s - a \cdot \zeta}{2} \leq \frac{|b| + 2s}{2} \leq |b| + s$$

which together with (10.2) leads to

$$-|b| - s \leq j \leq |b| + s$$

and we are done.

**Example 10.4** We compute all natural solutions of the system of linear inequations

$$2x - y \leq 2$$

$$x + y \geq 4$$
such that both $x$ and $y$ are multiples of 4. This corresponds to computing a DFA for the Presburger formula
\[ \exists z \ x = 4z \land \exists w \ y = 4w \land 2x - y \leq 2 \land x + y \geq 4 \]

The minimal DFA for the first two conjuncts can be computed using projections and intersections, but the result is also easy to guess: it is the DFA of Figure 10.3 (where a trap state has been omitted).

The solutions are then represented by the intersection of the DFAs shown in Figures 10.1, 10.2 (after merging states 0 and 1), and 10.3. The result is shown in Figure 10.4. (Some states from which no final state can be reached are omitted.)

A slight modification of AFtoNFA directly constructs a DFA for the solutions of $a \cdot x = b$, without having to intersect DFAs for $a \cdot x \leq b$ and $-a \cdot x \leq -b$. The states of the DFA are a trap state $q_t$.
accepting the empty language, plus integers satisfying:

\[ q \in \mathbb{Z} \text{ recognizes the encodings of the tuples } c \in \mathbb{N}^n \text{ such that } a \cdot c = q. \quad (10.3) \]

For the trap state \( q_t \), we take \( \delta(q_t, \zeta) = q_t \) for every \( \zeta \in \{0, 1\}^n \). For a state \( q \in \mathbb{Z} \) and a letter \( \zeta \in \{0, 1\}^n \) we determine the target state \( q' \) of transition \( q \xrightarrow{\zeta} q' \). Given a tuple \( c' \in \mathbb{N}^n \), property 10.3 requires \( c' \in L(q') \) iff \( a \cdot c' = q' \). As in the case of inequations, we have

\[
\begin{align*}
\delta(q, \zeta) &= \begin{cases}
q_t & \text{if } q = q_t \text{ or } q - a \cdot \zeta \text{ is odd} \\
\frac{1}{2}(q - a \cdot \zeta) & \text{if } q - a \cdot \zeta \text{ is even}
\end{cases} \\
\text{iff } a \cdot c' &= \begin{cases}
q & \text{property } 10.3 \text{ for } q' \\
\frac{1}{2}(q - a \cdot \zeta) & \text{iff } a \cdot c' \in L(q')
\end{cases}
\end{align*}
\]

If \( q - a \cdot \zeta \) is odd, then, since \( a \cdot c' \) is an integer, the equation \( a \cdot c' = \frac{1}{2}(q - a \cdot \zeta) \) has no solution. So in order to satisfy property 10.3 we must choose \( q' \) satisfying \( L(q') = \emptyset \), and so we take \( q' = q_t \). If \( q - a \cdot \zeta \) is even then we must choose \( q' \) satisfying \( a \cdot c' = q' \), and so we take \( q' = \frac{1}{2}(q - a \cdot \zeta) \). Therefore, the transition function of the DFA is given by:

For the final states, recall that a state is final iff it accepts the tuple \((0, \ldots, 0)\). So \( q_t \) is nonfinal and, by property 10.3, \( q \in \mathbb{Z} \) is final iff \( a \cdot (0, \ldots, 0) = q \). So the only final state is \( q = 0 \). The resulting algorithm is shown in Table 10.2. The algorithm does not construct the trap state.

**Example 10.5** Consider the formulas \( x + y \leq 4 \) and \( x + y = 4 \). The result of applying \( \text{AFtoDFA} \) to \( x + y \leq 4 \) is shown at the top of Figure 10.5. Notice the similarities and differences with the DFA for \( x + y \geq 4 \) in Figure 10.2. The bottom part of the Figure shows the result of applying \( \text{EqtoDFA} \) to \( x + y = 4 \). Observe that the transitions are a subset of the transitions of the DFA for \( x + y \leq 4 \). This example shows that the DFA is not necessarily minimal, since state \(-1\) can be deleted.

Partial correctness and termination of \( \text{EqtoDFA} \) are easily proved following similar steps to the case of inequations.

### 10.3 An NFA for the Solutions over the Integers.

We construct an NFA recognizing the encodings of the integer solutions (positive or negative) of a formula. In order to deal with negative numbers we use 2-complements. A 2-complement encoding of an integer \( x \in \mathbb{Z} \) is any word \( a_0a_1 \ldots a_n \), where \( n \geq 1 \), satisfying

\[
x = \sum_{i=0}^{n-1} a_i \cdot 2^i - a_n \cdot 2^n \quad (10.4)
\]
10.3. AN NFA FOR THE SOLUTIONS OVER THE INTEGERS.

EqtoDFA(ϕ)

Input: Equation ϕ = a · x = b
Output: DFA A = (Q, Σ, δ, q₀, F) such that L(A) = L(ϕ)
(without trap state)

1. Q, δ, F ← ∅; q₀ ← sb
2. W ← {sb}
3. while W ≠ ∅ do
   4. pick sk from W
   5. add sk to Q
   6. if k = 0 then add sk to F
   7. for all ζ ∈ {0, 1}ⁿ do
      8. if (k − a · ζ) is even then
         9. j ← ½(k − a · ζ)
      10. if sj ∉ Q then add sj to W
   11. add (sk, ζ, sj) to δ

Table 10.2: Converting an equation into a DFA recognizing the lsbf encodings of its solutions.

We call an the sign bit. For example, 110 encodes 1 + 2 − 0 = 3, and 111 encodes 1 + 2 − 4 = −1. If the word has length 1 then its only bit is the sign bit; in particular, the word 0 encodes the number 0, and the word 1 encodes the number −1. The empty word encodes no number. Observe that all of 110, 1100, 11000, . . . encode 3, and all of 1, 11, 111, . . . encode −1. In general, it is easy to see that all words of the regular expression a₀ . . . a₀−₁a₀a₀⁻¹ encode the same number: for a₀ = 0 this is obvious, and for a₀ = 1 both a₀ . . . a₀⁻¹1 and a₀ . . . a₀⁻¹11 encode the same number because

\[-2^m+n + 2^{m−1+n} + \ldots + 2^{n+1} = 2^n.\]

This property allows us to encode tuples of numbers using padding. Instead of padding with 0, we pad with the sign bit.

Example 10.6 The triple (12, −3, −14) is encoded by all the words of

\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}^* \]

For example, the triples (x, y, z) and (x', y', z') encoded by

\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1
\end{bmatrix}
\quad \text{and} \quad \begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}
\]
are given by

\[
\begin{align*}
  x &= 0 + 0 + 4 + 8 + 0 = 12 &
  x' &= 0 + 0 + 4 + 8 + 0 + 0 + 0 + 0 = 12 \\
  y &= 1 + 0 + 4 + 8 - 16 = -3 &
  y' &= 1 + 0 + 4 + 8 + 16 + 32 - 64 = -3 \\
  z &= 0 + 2 + 0 + 0 - 16 = -14 &
  z' &= 0 + 2 + 0 + 0 + 16 + 32 - 64 = -14
\end{align*}
\]

We construct a NFA (no longer a DFA!) recognizing the integer solutions of an atomic formula
\(a \cdot x \leq b\). As usual we take integers for the states, and the NFA should satisfy:

State \(q \in \mathbb{Z}\) recognizes the encodings of the tuples \(c \in \mathbb{Z}^n\) such that \(a \cdot c \leq q\). \hspace{1cm} (10.5)

However, integer states are no longer enough, because no state \(q \in \mathbb{Z}\) can be final: in the 2-complement encoding the empty word encodes no number, and so, since \(q\) cannot accept the empty word by property 10.5, \(q\) must be nonfinal. But we need at least one final state, and so we add to the NFA a unique final state \(q_f\) without any outgoing transitions, accepting only the empty word.

Given a state \(q \in \mathbb{Z}\) and a letter \(\zeta \in \{0, 1\}^n\), we determine the targets \(q'\) of the transitions \(q \xrightarrow{\zeta} q'\) of the NFA, where \(\zeta \in \{0, 1\}^n\). (As we will see, the NFA may have one or two such transitions.)
A word \( w' \in (\{0, 1\}^n)^* \) is accepted from some target state \( q' \) iff \( \zeta w' \) is accepted from \( q \). In the 2-complement encoding there are two cases:

(1) If \( w' \neq \epsilon \), then \( \zeta w' \) encodes the tuple of integers \( 2c' + \zeta \), where \( c' \in \mathbb{Z}^n \) is the tuple encoded by \( w' \). (This follows easily from the definition of 2-complements.)

(2) If \( w' = \epsilon \), then \( \zeta w' \) encodes the tuple of integers \( -\zeta \), because in this case \( \zeta \) is the sign bit.

In case (1), property 10.5 requires a target state \( q' \) such that \( a \cdot c \leq q \) iff \( a \cdot (2c + c\zeta) \leq q' \). So we take

\[
q' = \left\lfloor \frac{1}{2} (q - a \cdot \zeta) \right\rfloor
\]

For case (2) property 10.5 only requires a target state \( q' \) if \( a \cdot (-\zeta) \leq q \), and if so then it requires \( q' \) to be a final state. So if \( q + a \cdot \zeta \geq 0 \) then we add \( q \xrightarrow{\zeta} q_f \) to the set of transitions, and in this case the automaton has two transitions leaving state \( q \) and labeled by \( \zeta \). Summarizing, we define the transition function of the NFA by

\[
\delta(q, \zeta) = \begin{cases} 
\left\lfloor \frac{1}{2} (q - a \cdot \zeta) \right\rfloor, q_f & \text{if } q + a \cdot \zeta \geq 0 \\
\frac{1}{2} (q - a \cdot \zeta) & \text{otherwise}
\end{cases}
\]

Observe that the NFA contains all the states and transitions of the DFA for the natural solutions of \( a \cdot x \leq b \), plus possibly other transitions. All integer states are now nonfinal, the only final state is \( q_f \).

**Example 10.7** Figure 10.6 shows at the top the NFA recognizing all integer solutions of \( 2x - y \leq 2 \). It has all states and transitions of the DFA for the natural solutions, plus some more (compare with Figure 10.1). The final state \( q_f \) and the transitions leading to it are drawn in red. Consider for instance state \(-1\). In order to determine the letters \( \zeta \in \{0, 1\} \) for which \( q_f \in \delta(-1, \zeta) \), we compute \( q + a \cdot \zeta = -1 + 2\zeta_x - \zeta_y \) for each \((\zeta_x, \zeta_y) \in \{0, 1\}^2\), and compare the result to 0. We obtain that the letters leading to \( q_f \) are \((1, 0)\) and \((1, 1)\).

**10.3.1 Equations**

If order to construct an NFA for the integer solutions of an equation \( a \cdot x = b \) we can proceed as for inequations. The result is again an NFA containing all states and transitions of the DFA for the natural solutions computed in Section 10.2.1, plus possible some more. The automaton has an additional final state \( q_f \), and a transition \( q \xrightarrow{\zeta} q_f \) iff \( q + a \cdot \zeta = 0 \). Graphically, we can also obtain the NFA by starting with the NFA for \( a \cdot x \leq b \), and removing all transitions \( q \xrightarrow{\zeta} q' \) such that \( q' \neq \frac{1}{2} (q - a \cdot \zeta) \), and all transitions \( q \xrightarrow{\zeta} q_f \) such that \( q + a \cdot \zeta \neq 0 \).

**Example 10.8** The NFA for the integer solutions of \( 2x - y = 2 \) is shown in the middle of Figure 10.6. Its transitions are a subset of those of the NFA for \( 2x - y \leq 2 \).
The NFA for the integer solutions of an equation has an interesting property. Since $q + a \cdot \zeta = 0$ holds iff $\frac{1}{2}(q + a \cdot \zeta) = \frac{1}{2}(2q) = q$, the NFA has a transition $q \xrightarrow{\zeta} q_f$ iff it also has a self-loop $q \xrightarrow{\zeta} q$. (For instance, state 1 of the DFA in the middle of Figure 10.6 has a red transition labeled by $(0, 1)$ and a self-loop labeled by $(0, 1)$.) Using this property it is not difficult to see that the powerset construction does not cause a blowup in the number of states: it only adds one extra state for each predecessor of the final state.

**Example 10.9** The DFA obtained by applying the powerset construction to the NFA for $2x - y = 2$ is shown at the bottom of Figure 10.6 (the trap state has been omitted). Each of the three predecessors of $q_f$ gets “duplicated”.

Moreover, the DFA obtained by means of the powerset construction is minimal. This can be proved by showing that any two states recognize different languages. If exactly one of the states is final, we are done. If both states are nonfinal, say, $k$ and $k'$, then they recognize the solutions of $a \cdot x = k$ and $a \cdot x = k'$, and so their languages are not only distinct but even disjoint. If both states are final, then they are the “duplicates” of two nonfinal states $k$ and $k'$, and their languages are those of $k$ and $k'$, plus the empty word. So, again, their languages are distinct.

### 10.3.2 Algorithms

The algorithms for the construction of the NFAs are shown in Table 10.3. Additions to the previous algorithms are shown in blue.

#### Exercises

**Exercise 83** Let $k_1, k_2 \in \mathbb{N}_0$ be constants. Find a Presburger arithmetic formula, $\varphi(x, y)$, with free variables $x$ and $y$ such that $I \models \varphi(x, y)$ iff $I(x) \geq I(y)$ and $I(x) - I(y) \equiv k_1 \pmod{k_2}$. Find a corresponding automaton for the case $k_1 = 0$ and $k_2 = 2$.

**Exercise 84** Construct a finite automaton for the Presburger formula $\exists y \ x = 3y$ using the algorithms of the chapter.

**Exercise 85** $AFtoDFA$ returns a DFA recognizing all solutions of a given linear inequation

$$a_1 x_1 + a_2 x_2 + \ldots + a_k x_k \leq b \text{ with } a_1, a_2, \ldots, a_k, b \in \mathbb{Z}$$

encoded using the $lsbf$ encoding of $\mathbb{N}^k$. We may also use the most-significant-bit-first ($msbf$) encoding, e.g.,

$$msbf\left(\begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) = L\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix} \ \begin{bmatrix} 1 \\ 1 \end{bmatrix} \ \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right)$$
10.3. AN NFA FOR THE SOLUTIONS OVER THE INTEGERS.

**IneqZtoNFA(φ)**

**Input:** Inequation $φ = ax \leq b$ over $\mathbb{Z}$

**Output:** NFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $L(A) = L(φ)$

1. $Q, \delta, F \leftarrow \emptyset$; $q_0 \leftarrow s_b$
2. $W \leftarrow \{s_b\}$
3. While $W \neq \emptyset$
   4. Pick $s_k$ from $W$
   5. Add $s_k$ to $Q$
   6. For all $ζ \in \{0, 1\}^n$
      7. $j \leftarrow \frac{1}{2}(k - a \cdot ζ)$
      8. If $s_j \notin Q$ then add $s_j$ to $W$
      9. Add $(s_k, ζ, s_j)$ to $δ$
     10. $j' \leftarrow k + a \cdot ζ$
    11. If $j' \geq 0$ then
       12. Add $q_f$ to $Q$ and $F$
       13. Add $(s_k, ζ, q_f)$ to $δ$

**EqZtoNFA(φ)**

**Input:** Equation $φ = ax = b$ over $\mathbb{Z}$

**Output:** NFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $L(A) = L(φ)$

1. $Q, \delta, F \leftarrow \emptyset$; $q_0 \leftarrow s_b$
2. $W \leftarrow \{s_b\}$
3. While $W \neq \emptyset$
   4. Pick $s_k$ from $W$
   5. Add $s_k$ to $Q$
   6. For all $ζ \in \{0, 1\}^n$
      7. If $(k - a \cdot ζ)$ is even then
         8. If $(k + a \cdot ζ) = 0$ then add $k$ to $F$
         9. $j \leftarrow \frac{1}{2}(k - a \cdot ζ)$
      10. If $s_j \notin Q$ then add $s_j$ to $W$
     11. Add $(s_k, ζ, s_j)$ to $δ$
     12. $j' \leftarrow k + a \cdot ζ$
    13. If $j' \geq 0$ then
       14. Add $q_f$ to $Q$ and $F$
       15. Add $(s_k, ζ, q_f)$ to $δ$

Table 10.3: Converting an inequality into a NFA accepting the 2-complement encoding of the solution space.

1. Construct a finite automaton for the inequation $2x - y \leq 2$ w.r.t. $msbf$ encoding.
2. Adapt $AFtoDFA$ to the $msbf$ encoding.
Figure 10.6: NFAs for the solutions of $2x - y \leq 2$ and $2x - y = 2$ over $\mathbb{Z}$, and minimal DFA for the solutions of $2x - y = 2$. 
Part II

Automata on Infinite Words
Chapter 11

Classes of ω-Automata and Conversions

In Section 11.1 we introduce ω-regular expressions, a textual notation for defining languages of infinite words. The other sections introduce different classes of automata on infinite words, most of them with the same expressive power as ω-regular expressions, and conversion algorithms between them.

11.1 ω-languages and ω-regular expressions

Let Σ be an alphabet. An infinite word, also called an ω-word, is an infinite sequence \(a_0a_1a_2\ldots\) of letters of Σ. The concatenation of a finite word \(w_1 = a_1\ldots a_n\) and an ω-word \(w_2 = b_1b_2\ldots\) is the ω-word \(w_1w_2 = a_1\ldots a_nb_1b_2\ldots\), sometimes also denoted by \(w_1 \cdot w_2\). We denote by \(Σ^ω\) the set of all ω-words over Σ. A set \(L \subseteq Σ^ω\) of ω-words is an infinitary language or ω-language over Σ.

The concatenation of a language \(L_1\) and an ω-language \(L_2\) is \(L_1 \cdot L_2 = \{w_1w_2 \in Σ^ω \mid w_1 \in L_1\text{ and } w_2 \in L_2\}\). The ω-iteration of a language \(L \subseteq Σ^*\) is the ω-language \(L^ω = \{w_1w_2w_3\ldots \mid w_i \in L \setminus \{\epsilon\}\}\). Observe that \(\{\emptyset\}^ω = \emptyset\), in contrast to the case of finite words, where \(\{\emptyset\}^* = \{\epsilon\}\).

We extend regular expressions to ω-regular expressions, a formalism to define ω-languages.

**Definition 11.1** The ω-regular expressions \(s\) over an alphabet \(Σ\) are defined by the following grammar, where \(r \in RE(Σ)\) is a regular expression

\[
s ::= r^ω \mid rs_1 \mid s_1 + s_2
\]

Sometimes we write \(r \cdot s_1\) instead of \(rs_1\). The set of all ω-regular expressions over \(Σ\) is denoted by \(RE_ω(Σ)\). The language \(L_ω(s) \subseteq Σ\) of an ω-regular expression \(s \in RE_ω(Σ)\) is defined inductively as

- \(L_ω(r^ω) = (L(r))^ω\);
- \(L_ω(rs_1) = L(r) \cdot L_ω(s_1)\); and
- \(L_ω(s_1 + s_2) = L_ω(s_1) \cup L_ω(s_2)\).
A language \( L \) is \( \omega \)-regular if there is an \( \omega \)-regular expression \( s \) such that \( L = L_\omega(s) \).

Observe that the empty \( \omega \)-language is \( \omega \)-regular because \( L_\omega(\emptyset^\omega) = \emptyset \).

### 11.2 Büchi automata

Büchi automata have the same syntax as NFAs, but a different definition of acceptance. Suppose that an NFA \( A = (Q, \Sigma, \delta, q_0, F) \) is given as input an infinite word \( w = a_0a_1a_2 \ldots \) over \( \Sigma \). Intuitively, a run of \( A \) on \( w \) never terminates, and so we cannot define acceptance in terms of the state reached at the end of the run. In fact, even the name “final state” is no longer appropriate for Büchi automata. So from now on we speak of “accepting states”, although we still denote by \( F \) the set of accepting states. We say that a run of a Büchi automaton is accepting if some accepting state is visited along the run infinitely often. Since the set of accepting states is finite, “some accepting state is visited infinitely often” is equivalent to “the set of accepting states is infinitely often”.

**Definition 11.2** A nondeterministic Büchi automaton (NBA) is a tuple \( A = (Q, \Sigma, \delta, q_0, F) \), where \( Q, \Sigma, \delta, q_0, \) and \( F \) are defined as for NFAs. A run of \( A \) on an \( \omega \)-word \( a_0a_1a_2 \ldots \) is an infinite sequence \( \rho = p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \ldots \) such that \( p_i \in Q \) for \( 0 \leq i \leq n \) and \( p_{i+1} \in \delta(p_i, a_i) \) for every \( 0 \leq i \). Let \( \inf(\rho) \) be the set \( \{ q \in Q \mid q = p_i \text{ for infinitely many } i \}'s \), i.e., the set of states that occur in \( \rho \) infinitely often. A run \( \rho \) is accepting if \( \inf(\rho) \cap F \neq \emptyset \). An NBA accepts an \( \omega \)-word \( w \in \Sigma^\omega \) if it has an accepting run on \( w \). The language recognized by and NBA \( A \) is the set \( L_\omega(A) = \{ w \in \Sigma^\omega \mid w \text{ is accepted by } A \} \).

We refer to the condition \( \inf(\rho) \cap F \neq \emptyset \) as the Büchi condition \( F \).

A Büchi automaton is deterministic if it is deterministic when seen as an automaton on finite words. One can also define NBAs with \( \epsilon \)-transitions, but we will not need them.\(^1\)

**Example 11.3** Figure 11.1 shows two Büchi automata. The automaton on the left is deterministic, and recognizes all \( \omega \)-words over the alphabet \( \{a, b\} \) that contain infinitely many occurrences of \( a \). So, for instance, \( A \) accepts \( a^\omega, ba^\omega, (ab)^\omega \), or \( (ab^{100})^\omega \), but not \( b^\omega \) or \( a^{100}b^\omega \). To prove that this is indeed the language we show that every \( \omega \)-word containing infinitely many \( as \) is accepted by \( A \), and that every word accepted by \( A \) contains infinitely many \( as \). For the first part, observe that immediately after reading any \( a \) the automaton \( A \) always visits its (only) accepting state (because all transitions labeled by \( a \) lead to it); therefore, when \( A \) reads an \( \omega \)-word containing infinitely many \( as \) it visits its accepting state infinitely often, and so it accepts. For the second part, if \( w \) is accepted by \( A \), then there is a run of \( A \) on \( w \) that visits the accepting state infinitely often. Since all transitions leading to the accepting state are labeled by \( a \), the automaton must read infinitely many \( as \) during the run, and so \( w \) contains infinitely many \( as \).

\(^1\)Notice that the definition of NBA-\( \epsilon \) requires some care, because infinite runs containing only finitely many non-\( \epsilon \) transitions are never accepting, even if they visit some accepting state infinitely often.
11.2. Büchi Automata

The automaton on the right of the figure is not deterministic, and recognizes all \( \omega \)-words over the alphabet \{a, b\} that contain finitely many occurrences of \( a \). The proof is similar.

**Example 11.4** Figure 11.2 shows three further Büchi automata over the alphabet \{a, b, c\}.

The top-left automaton recognizes the \( \omega \)-words in which for every occurrence of \( a \) there is a later occurrence of \( b \). So, for instance, the automaton accepts \((ab)^\omega\), \(c^\omega\), or \((bc)^\omega\), but not \(ac^\omega\) or \(ab(ac)^\omega\).

The top right automaton recognizes the \( \omega \)-words that contain finitely many occurrences of \( a \), or infinitely many occurrences of \( a \) and infinitely many occurrences of \( b \). Finally, the automaton at the bottom recognizes the \( \omega \)-words in which between every occurrence of \( a \) and the next occurrence
of \( c \) there is at most one occurrence of \( b \); more precisely, for every two numbers \( i < j \), if the letter at position \( i \) is an \( a \) and the first occurrence of \( c \) after \( i \) is at position \( j \), then there is at most one number \( i < k < j \) such that the letter at position \( k \) is a \( b \).

11.2.1 From \( \omega \)-regular expressions to NBAs and back

We present algorithms for converting an \( \omega \)-regular expression into a NBA, and vice versa. This provides a first “sanity check” for NBAs as data structure: they can represent exactly the \( \omega \)-regular languages.

From \( \omega \)-regular expressions to NBAs. We proceed by induction on the structure of the \( \omega \)-regular expression. Recall that for every regular expression \( r \) we can construct an NFA-\( \epsilon \) \( A_r \) with a unique final state, no transition leading to the initial state, and no transition leaving the final state. In fact, if \( \epsilon \not\in L(r) \) then \( A_r \) has exactly one \( \epsilon \)-transition, otherwise none. An NBA for \( r \) is obtained by adding to the NFA-\( \epsilon \) for \( r \) transitions leading from the final state to the targets of the transitions leaving the initial state, as shown at the top of Figure ??; the \( \epsilon \)-transition, if present, can be easily removed. An NBA for \( r \cdot s \) is obtained by merging states as shown in the middle of the figure. Finally, an NBA for \( s_1 + s_2 \) is obtained by merging the initial states of the NBAs for \( s_1 \) and \( s_2 \) as shown at the bottom. Notice that we always obtain a NBA with no transitions leading into the initial state.

From NBAs to \( \omega \)-regular expressions. Let \( A = (Q, \Sigma, \delta, q_0, F) \) be a NBA. For every two states \( q, q' \in Q \), let \( A_q^{q'} \) be the NFA (not the NBA!) obtained from \( A \) by changing the initial state to \( q \) and the set of final states to \( \{q'\} \). Using algorithm NFAtoRE we can construct a regular expression \( s_q^{q'} \) such that \( L(A_q^{q'}) = L(s_q^{q'}) \). In fact, it is even better to construct a regular expression \( r_q^{q'} \) whose language contains the word accepted by \( L(A_q^{q'}) \) by means of runs that visit \( q' \) exactly once (how to use NFAtoRE for this is left as a little exercise). We use these expressions to compute an \( \omega \)-regular expression for \( L_\omega(A) \). For every accepting state \( q \in F \), let \( L_q \subseteq L_\omega(A) \) be the set of \( \omega \)-words \( w \) such that some run of \( A \) on \( w \) visits the state \( q \) infinitely often. We have \( L_\omega(A) = \bigcup_{q \in F} L_q \).

Every word \( w \in L_q \) can be split into an infinite sequence \( w_1 w_2 w_3 \ldots \) of finite, nonempty words, where \( w_1 \) is the word read by \( A \) until it visits \( q \) for the first time, and for every \( i > 1 \) \( w_i \) is the word read by the automaton between the \( i \)-th and the \( (i+1) \)-th visits to \( q \). It follows \( w_1 \in L(r_q^{q0}) \), and \( w_i \in L(r_q^{q_i}) \) for every \( i > 1 \). So we have \( L_q = L_\omega \left( r_q^{q0} \right)^\omega \), and therefore

\[
\sum_{q \in F} r_q^{q0} \left( r_q^{q} \right)^\omega
\]

is the \( \omega \)-regular expression we are looking for.

Example 11.5 Consider the top right NBA of Figure 11.2, shown again in Figure 11.4.
11.2. BÜCHI AUTOMATA

Figure 11.3: From ω-regular expressions to Büchi automata
We have to compute \( r_1^1 \left( r_1^1 \right)^\omega + r_0^2 \left( r_2^2 \right)^\omega \). Using \textsc{NFAtoRE} and simplifying we get

\[
\begin{align*}
\hat{r}_0^1 &= (a + b + c)^*(b + c) \\
\hat{r}_0^2 &= (a + b + c)^*b \\
\hat{r}_1^1 &= (b + c)^* \\
\hat{r}_2^1 &= b + (a + c)(a + b + c)^*b
\end{align*}
\]

and (after some simplifications) we obtain the \( \omega \)-regular expression

\[
(a + b + c)^*(b + c)^\omega + (a + b + c)^*b(b + (a + c)(a + b + c)^*b)^\omega
\]

\[ \blacksquare \]

### 11.2.2 Non-equivalence of NBA and DBA

Unfortunately, DBAs do not recognize all \( \omega \)-regular languages, and so they do not have the same expressive power as NBAs. We show that the language of \( \omega \)-words containing finitely many \( a \)s is not recognized by any DBA. Intuitively, the NBA for this language “guesses” the last occurrence of \( a \), and this guess cannot be determinized using only a finite number of states.

**Proposition 11.6** The language \( L = (a+b)^* b^\omega \), (i.e., the language of all \( \omega \)-words in which \( a \) occurs only finitely often) is not recognized by any DBA.

**Proof:** Assume that \( L = L_\omega(A) \), for some DBA \( A = (\{a, b\}, Q, q_0, \delta, F) \), and extend \( \delta \) to \( \hat{\delta} : Q \times \{a, b\}^* \to Q \) by \( \hat{\delta}(q, \epsilon) = q \) and \( \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a) \). Consider the \( \omega \)-word \( w_0 = b^\omega \). Since \( w_0 \in L \), the automaton \( A \) has an accepting run on \( w_0 \), and so \( \hat{\delta}(q_0, u_0) \in F \) for some finite prefix \( u_0 \) of \( w_0 \). Consider now \( w_1 = u_0 a b^\omega \). We have \( w_1 \in L \), and so \( A \) has an accepting run on \( w_1 \), which
implies $\hat{\delta}(q_0, u_0 a u_1) \in F$ for some finite prefix $u_0 a u_1$ of $w_1$. In a similar fashion we continue constructing finite words $u_i$ such that $\hat{\delta}(q_0, u_0 a u_1 a \ldots a u_i) \in F$. Since $Q$ is finite, there are indices $0 \leq i < j$ such that $\hat{\delta}(q_0, u_0 a u_1 a \ldots a u_i) = \hat{\delta}(q_0, u_0 a u_1 a \ldots a u_j)$. It follows that $A$ has an accepting run on $u_0 a u_1 a \ldots a u_i (a u_{i+1} \ldots u_{j-1} a u_j)$. But this word has infinitely many occurrences of $a$, so it does not belong to $L$.

Note that the $L = ((a + b)^* a)^\omega$ (the set of infinite words in which $a$ occurs infinitely often) is accepted by the DBA on the left of Figure 11.1.

### 11.3 Generalized Büchi automata

Generalized Büchi automata are an extension of Büchi automata that is convenient for some constructions (for instance, intersection). A generalized Büchi automaton (NGA) differs from a Büchi automaton in its accepting condition. A NGA has a collection of sets of accepting states $F = \{F_0, \ldots, F_{m-1}\}$, and a run $\rho$ is accepting if for every set $F_i \in F$ some state of $F_i$ is visited by $\rho$ infinitely often. Formally, $\rho$ is accepting if $\inf(\rho) \cap F_i \neq \emptyset$ for every $i \in \{0, \ldots, m - 1\}$. Abusing language, we speak of the generalized Büchi condition $F$. Ordinary Büchi automata correspond to the special case $m = 1$.

A NGA with $n$ states and $m$ sets of accepting states can be translated into an NBA with $mn$ states. The construction is based on the following observation: a run $\rho$ visits each set of $F$ infinitely if and only if the following two conditions hold:

1. $\rho$ eventually visits $F_0$; and

2. for every $i \in \{0, \ldots, m - 1\}$, every visit of $\rho$ to $F_i$ is eventually followed by a later visit to $F_{i \oplus 1}$, where $\oplus$ denotes addition modulo $m$. (Between the visits to $F_i$ and $F_{i \oplus 1}$ there can be arbitrarily many visits to other sets of $F$.)

whose proof is obvious. This suggests to take for the NBA $m$ “copies” of the NGA, but with a modification: the NBA moves from the $i$-th to the $i \oplus 1$-th copy whenever it visits a state of $F_i$ (i.e., the transitions of the $i$-th copy that leave a state of $F_i$ are redirected from the $i$-th copy to the $(i \oplus 1)$-th copy). This way, visiting the accepting states of the first copy infinitely often is equivalent to visiting the accepting states of each copy infinitely often.

More formally, the states of the NBA are pairs $[q, i]$ where $q$ is a state of the NGA and $i \in \{0, \ldots, m - 1\}$. Intuitively, $[q, i]$ is the $i$-th copy of $q$. If $q \notin F_i$ then the successors of $[q, i]$ are states of the $i$-th copy, and otherwise states of the $(i \oplus 1)$-th copy.
NGAtoNBA(A)

Input: NGA A = (Q, Σ, q₀, δ, ℱ), where ℱ = {F₁, ..., Fₘ}

Output: NBA A′ = (Q′, Σ, δ′, q₀′, F′)

1. Q′, δ′, F′ ← ∅; q₀′ ← [q₀, 0]
2. W ← {[q₀, 0]}
3. while W ≠ ∅ do
   4. pick [q, i] from W
   5. add [q, i] to Q′
   6. if q ∈ F₀ and i = 0 then add [q, i] to F′
   7. for all a ∈ Σ, q′ ∈ δ(q, a) do
      8. if q ∉ Fᵢ then
         9. if [q′, i] ∉ Q′ then add [q′, i] to W
      10. add ([q, i], a, [q′, i]) to δ′
      else /* q ∈ Fᵢ */
         12. if [q′, i ⊕ 1] ∉ Q′ then add [q′, i ⊕ 1] to W
      13. add ([q, i], a, [q′, i ⊕ 1]) to δ′
   14. return (Q′, Σ, δ′, q₀′, F′)

Example 11.7

Figure 11.5 shows a NGA over the alphabet {a, b} on the left, and the NBA obtained by applying NGAtoNBA to it on the right. The NGA has two sets of accepting states, F₀ = {q} and F₁ = {r}, and so its accepting runs are those that visit both q and r infinitely often. It is easy to see that the automaton recognizes the ω-words containing infinitely many occurrences of a and infinitely many occurrences of b.

The NBA on the right consists of two copies of the NGA: the 0-th copy (pink) and the 1-st copy (blue). Transitions leaving [q, 0] are redirected to the blue copy, and transitions leaving [r, 1] are redirected to the pink copy. The only accepting state is [q, 0].
11.4 Other classes of $\omega$-automata

We have seen that not every NBA is equivalent to a DBA; i.e., there is determinization procedure for Büchi automata. This raises the question whether there exists other classes of automata for which a determinization procedure exists. As we shall see, the answer is yes, but the first determinizable classes we find will have other problems, and so this section can be seen as a quest for automata classes satisfying more and more properties.

11.4.1 Co-Büchi Automata

Like a Büchi automaton, a (nondeterministic) co-Büchi automaton (NCA) has a set $F$ of accepting states. However, a run $\rho$ of a NCA is accepting if it only visits states of $F$ finitely often. Formally, $\rho$ is accepting if $\inf(\rho) \cap F = \emptyset$. So a run of a NCA is accepting iff it is not accepting as run of a NBA (this is the reason for the name “co-Büchi”).

We show that co-Büchi automata can be determinized. We fix an NCA $A = (Q, \Sigma, \delta, q_0, F)$ with $n$ states, and use Figure 11.6 as running example. We construct a DCA $B$ such that $L_\omega(B) = L_\omega(A)$ in three steps:

1. We define a mapping $\text{dag}$ that assigns to each $w \in \Sigma^\omega$ a directed acyclic graph $\text{dag}(w)$.

2. We prove that $w$ is rejected by $A$ iff $\text{dag}(w)$ contains only finitely many breakpoints.

3. We construct a DCA $B$ which accepts $w$ if and only if $\text{dag}(w)$ contains finitely many breakpoints.

Intuitively, $\text{dag}(w)$ is the result of “bundling together” all the runs of $A$ on the word $w$. Figure 11.7 shows the initial parts of $\text{dag}(ab\omega)$ and $\text{dag}((ab)\omega)$. Formally, let $w = \sigma_1\sigma_2 \ldots$ be a word of $\Sigma^\omega$. The directed acyclic graph $\text{dag}(w)$ has nodes in $Q \times \mathbb{N}$ and edges labelled by letters of $\Sigma$, and is inductively defined as follows:

- $\text{dag}(w)$ contains a node $\langle q, 0 \rangle$ for every initial state $q \in Q_0$.

- If $\text{dag}(w)$ contains a node $\langle q, i \rangle$ and $q' \in \delta(q_{i+1}, q)$, then $\text{dag}(w)$ also contains a node $\langle q', i+1 \rangle$ and an edge $\langle q, i \rangle \xrightarrow{\sigma_{i+1}} \langle q', i+1 \rangle$. 

Figure 11.6: Running example for the determinization procedure
• \( \text{dag}(w) \) contains no other nodes or edges.

Clearly, \( q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \cdots \) is a run of \( A \) if and only if \( \langle q_0, 0 \rangle \xrightarrow{\sigma_1} \langle q_1, 1 \rangle \xrightarrow{\sigma_2} \langle q_2, 2 \rangle \cdots \) is a path of \( \text{dag}(w) \). Moreover, \( A \) accepts \( w \) if and only if no path of \( \text{dag}(w) \) visits accepting states infinitely often. We partition the nodes of \( \text{dag}(w) \) into levels: the \( i \)-th level contains all the nodes of \( \text{dag}(w) \) of the form \( \langle q, i \rangle \).

One could be tempted to think that the accepting condition “some path of \( \text{dag}(w) \) only visits accepting states finitely often” is equivalent to “only finitely many levels of \( \text{dag}(w) \) contain accepting states”, but \( \text{dag}(aba^\omega) \) shows that this is not true: even though all its paths visit accepting states only finitely often, infinitely many levels (in fact, all all levels \( i \geq 3 \)) contain accepting states. For this reason we introduce the set of breakpoint levels of the graph \( \text{dag}(w) \), inductively defined as follows:

• The 0-th level of \( \text{dag}(w) \) is a breakpoint.

• If level \( l \) is a breakpoint, then the next level \( l' > l \) such that every path between nodes of \( l \) and \( l' \) visits an accepting state is also a breakpoint.

We claim that “some path of \( \text{dag}(w) \) only visits accepting states finitely often” is equivalent to “the set of breakpoint levels of \( \text{dag}(w) \) is finite”. If the breakpoint set is infinite, then clearly \( \text{dag}(w) \) contains at least an infinite path, and moreover all infinite paths visit accepting states infinitely often. If the breakpoint set is finite, let \( i \) be the largest breakpoint. If \( \text{dag}(w) \) is finite, we are done. If \( \text{dag}(w) \) is infinite, then for every \( j > i \) there is a path \( \pi_j \) from level \( i \) to level \( j \) that does not visit any accepting state. The paths \( \{\pi_j\}_{j>i} \) build an acyclic graph of bounded degree, which therefore must contain an infinite path \( \pi \) by Kőnigs lemma. But obviously \( \pi \) never visits any accepting state, and we are done.

Now, if we were able to tell that a level is a breakpoint just by examining it, we would be done: we would take the set of all possible levels as states of the DCA (i.e., the powerset of \( Q \), as

![Figure 11.7: The (initial parts of) \( \text{dag}(aba^\omega) \) and \( \text{dag}((ab)^\omega) \)]
in the powerset construction for determination of NFAs), the possible transitions between levels as transitions, and the breakpoints as accepting states. The run of this automaton on \( w \) would be nothing but an encoding of \( \text{dag}(w) \), and it would be accepting iff it contains only finitely many breakpoints, as required by the co-Büchi acceptance condition. However, the level does not contain enough information for that. The solution to this problem is to add information to the states: we take for the states of the DCA pairs \([P, O]\), where \( O \subseteq P \subseteq Q \), with the following intended meaning: \( P \) is the set of states of a level, and \( q \in O \) iff \( q \) is the endpoint of a path that starts at the last breakpoint and has not yet visited any accepting state. We call \( O \) the set of owing states (states that “owe” a visit to the accepting states). To guarantee that \( O \) indeed has this intended meaning, we define the DCA \( B = (\tilde{Q}, \Sigma, \tilde{\delta}, \tilde{q}_0, \tilde{F}) \) as follows:

- The initial state is the pair \([\{q_0\}, \emptyset]\) if \( q_0 \in F \), and the pair \([\{q_0\}, \{q_0\}]\) otherwise.
- The transition relation is given by \( \tilde{\delta}([P, O], a) = [P', O'] \), where \( P' = \delta(P, a) \), and:
  - if \( O \neq \emptyset \), i.e., if the current set of owing states is nonempty, then \( O' = \delta(O, a) \setminus F \);
  - if \( O = \emptyset \), (i.e., if the current level is a breakpoint, and the automaton must start searching for the next one) then \( O' = \delta(P, a) \setminus F \), i.e., all non-final states of the next level become owing.
- The accepting states are those at which a breakpoint is reached, i.e. \([P, O] \in \tilde{F}\) is accepting iff \( O = \emptyset \).

With this definition, a run is accepting iff it contains infinitely many breakpoints. The algorithm for the construction is:

\[
\text{NCAtoDCA}(A)
\]

**Input:** NCA \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** DCA \( B = (\tilde{Q}, \Sigma, \tilde{\delta}, \tilde{q}_0, \tilde{F}) \) with \( L_\omega(A) = \tilde{B} \)

1. \( \tilde{Q}, \tilde{\delta}, \tilde{F} \leftarrow \emptyset \); if \( q_0 \in F \) then \( \tilde{q}_0 \leftarrow [q_0, \emptyset] \) else \( \tilde{q}_0 \leftarrow [\{q_0\}, \{q_0\}] \)
2. \( W \leftarrow \{ \tilde{q}_0 \} \)
3. while \( W \neq \emptyset \) do
4. \( \text{pick} \ [P, O] \text{ from } W; \text{ add} \ [P, O] \text{ to } \tilde{Q} \)
5. if \( P = \emptyset \) then \( \text{add} \ [P, O] \text{ to } \tilde{F} \)
6. for all \( a \in \Sigma \) do
7. \( P' = \delta(P, a) \)
8. if \( O \neq \emptyset \) then \( O' \leftarrow \delta(O, a) \setminus F \) else \( O' \leftarrow \delta(P, a) \setminus F \)
9. add \( ([P, O], a, [P', O']) \) to \( \tilde{\delta} \)
10. if \( [P', O'] \notin \tilde{Q} \) then \( \text{add} \ [P', Q'] \text{ to } W \)

Figure 11.8 shows the result of applying the algorithm to our running example. The NCA is at the top, and the DCA below it on the left. On the right we show the DFA obtained by applying the
powerset construction. Observe that it does not yield a correct result: the resulting NCA accepts for instance the word $b^\omega$, which is not accepted by the original NCA. For the complexity, observe that the number of states of the DCA is bounded by the number of pairs $[P, O]$ such that $O \subseteq P \subseteq Q$. For every state $q \in Q$ there are three mutually exclusive possibilities: $q \in O$, $q \in P \setminus O$, and $q \in Q \setminus P$. So if $A$ has $n$ states then $B$ has at most $3^n$ states.

Unfortunately, co-Büchi automata do not recognize all $\omega$-regular languages. In particular, we claim that no NCA recognizes the language $L$ of $\omega$-words over $\{a, b\}$ containing infinitely many $a$’s. To see why, assume the contrary. Then there would also be a DCA for the same language. Now we swap accepting and non-accepting states, and interpret the result as a DBA. It is easy to see that this DBA accepts the complement of $L$. But the complement of $L$ is $(a + b)^*b^\omega$, which by Proposition 11.6 is not accepted by any DBA, and we reach a contradiction, proving the claim. So we now ask whether there is a class of $\omega$-automata that (1) recognizes all $\omega$-regular languages and (2) has a determinization procedure.

### 11.4.2 Muller automata

A (nondeterministic) Muller automaton (NMA) has a collection $\{F_0, \ldots, F_{m-1}\}$ of sets of accepting states. A run $\rho$ is accepting if the set of states $\rho$ visits infinitely often is equal to one of the $F_i$’s. Formally, $\rho$ is accepting if $\inf(\rho) = F_i$ for some $i \in \{0, \ldots, m-1\}$. We speak of the Muller condition $\{F_0, \ldots, F_{m-1}\}$.
11.4. OTHER CLASSES OF ω-AUTOMATA

NMA$s$ have the nice feature that any boolean combination of predicates of the form “state $q$ is visited infinitely often” can be formulated as a Muller condition. It suffices to put in the collection all sets of states for which the predicate holds. For instance, the condition $(q \in \inf(\rho)) \land \neg(q' \in \inf(\rho))$ corresponds to the Muller condition containing all sets of states $F$ such that $q \in F$ and $q' \notin F$. In particular, the Büchi and generalized Büchi conditions are special cases of the Muller condition (as well as the Rabin and Street conditions introduced in the next sections). The obvious disadvantage is that the translation of a Büchi condition into a Muller condition involves an exponential blow-up: a Büchi automaton with states $Q = \{q_0, \ldots, q_n\}$ and Büchi condition $\{q_n\}$ is transformed into an NMA with the same states and transitions, but with a Muller condition $\{F \subseteq Q | q_n \in F\}$, a collection containing $2^n$ sets of states.

Deterministic Muller automata recognize all $\omega$-regular languages. The proof of this result is complicated, and we omit it here.

**Theorem 11.8 (Safra)** Any NBA with $n$ states can be effectively transformed into a DMA of size $n^{O(n)}$.

We can easily give a deterministic Muller automaton for the language $L = (a + b)^* b^\omega$, which, as shown in Proposition 11.6, is not recognized by any DBA. The automaton is

![Diagram](attachment:image.png)

with Muller condition $\{\{q_1\}\}$. The accepting runs are the runs $\rho$ such that $\inf(\rho) = \{q_1\}$, i.e., the runs visiting state 1 infinitely often and state $q_0$ finitely often. Those are the runs that initially move between states $q_0$ and $q_1$, but eventually jump to $q_1$ and never visit $q_0$ again. It is easy to see that these runs accept exactly the words containing finitely many occurrences of $a$.

We finally show that an NMA can be translated into a NBA, and so that Muller and Büchi automata have the same expressive power. We start with a simple observation. In Chapter 4 we have presented an algorithm that given NFAs $A_1$ and $A_2$ computes an NFA $A_1 \cup A_2$ recognizing $L(A_1) \cup L(A_2)$. The algorithm puts $A_1$ and $A_2$ “side by side”, and adds a new initial state and transitions. It is graphically represented in Figure 11.9. It is easy to see that the same algorithm works for NBAs.

Equipped with this observation we proceed as follows. Given a Muller automaton $A = (Q, \Sigma, q_0, \delta, \{F_0, \ldots, F_{m-1}\})$, it is easy to see that $L_\omega(A) = \bigcup_{i=0}^{m-1} L_\omega(A_i)$, where $A_i = (Q, \Sigma, q_0, \delta, \{F_i\})$. So we proceed in three steps: first, we convert the NMA $A_i$ into a NGA $A'_i$; then we convert $A'_i$ into a NBA $A''_i$ using $NGAtoNBA()$ and, finally, we construct $A'' = \bigcup_{i=0}^{m-1} A''_i$ using the algorithm for union of NFAs.

For the first step, we observe that, since an accepting run $\rho$ of $A_i$ satisfies $\inf(\rho) = F_i$, from some point on the run only visits states of $F_i$. In other words, $\rho$ consists of an initial finite part, say $\rho_0$, that may visit all states, and an infinite part, say $\rho_1$, that only visits states of $F_i$. The idea for
the construction for $A'_i$ is to take two copies of $A_i$. The first one is a “full” copy, while the second one only contains copies of the states of $F_i$. For every transition $[q, 0] \xrightarrow{a} [q', 0]$ of the first copy such that $q' \in F_i$ there is another transition $[q, 0] \xrightarrow{a} [q', 1]$ leading to the “twin brother” $[q', 1]$. Intuitively, $A'_i$ simulates $\rho$ by executing $\rho_0$ in the first copy, and $\rho_1$ in the second. The condition that $\rho_1$ must visit each state of $F_i$ infinitely often is enforced as follows: if $F_i = \{q_1, \ldots, q_k\}$, then we take for $A_i$ the generalized Büchi condition $\{[q_1, 1], \ldots, [q_k, 1]\}$.

**Example 11.9** Figure 11.10 shows a NMA $A = (Q, \Sigma, \delta, q_0, F)$ where $F = \{\{q\}, \{r\}\}$. While $A$ is syntactically identical to the NGA of Figure 11.5, we now interpret $F$ as a Muller condition: a run $\rho$ is accepting if $\inf(\rho) = \{q\}$ or $\inf(\rho) = \{r\}$. In other words, an accepting run $\rho$ eventually moves to $q$ and stays there forever, or eventually moves to $r$ and stays there forever. It follows that $A$ accepts the $\omega$-words that contain finitely many $a$s or finitely many $b$s. The top-right part of the figure shows the two NGAs $A'_0, A'_1$ defined above. Since in this particular case $F'_0$ and $F'_1$ only contain singleton sets, $A'_0$ and $A'_1$ are in fact NBAs, i.e., we have $A''_0 = A'_0$ and $A''_1 = A'_1$. The bottom-right part shows the final NBA $A'' = A''_0 \cup A''_1$.

Formally, the algorithm carrying out the first step of the construction looks as follows:
11.4. OTHER CLASSES OF $\omega$-AUTOMATA

$\text{NMA1toNGA}(A)$

Input: NMA $A = (Q, \Sigma, q_0, \delta, \{F\})$

Output: NGA $A = (Q', \Sigma, q'_0, \delta', \mathcal{F}')$

1. $Q', \delta', \mathcal{F}' \leftarrow \emptyset$
2. $q'_0 \leftarrow [q_0, 0]$
3. $W \leftarrow \{(q_0, 0)\}$
4. while $W \neq \emptyset$ do
   5. pick $[q, i]$ from $W$; add $[q, i]$ to $Q'$
   6. if $q \in F$ and $i = 1$ then add $\{(q, 1)\}$ to $\mathcal{F}'$
   7. for all $a \in \Sigma$, $q' \in \delta(q, a)$ do
      8. if $i = 0$ then
          9. add $([q, 0], a, [q', 0])$ to $\delta'$
         10. if $[q', 0] \notin Q'$ then add $[q', 0]$ to $W$
      11. if $q' \in F$ then
          12. add $([q, 0], a, [q', 1])$ to $\delta'$
         13. if $[q', 1] \notin Q'$ then add $[q', 1]$ to $W$
     14. else /* $i = 1$ */
      15. if $q' \in F$ then
          16. add $([q, 1], a, [q', 1])$ to $\delta'$
         17. if $[q', 1] \notin Q'$ then add $[q', 1]$ to $W$
   18. return $(Q', \Sigma, q'_0, \delta', \mathcal{F}')$

Complexity. Assume $Q$ contains $n$ states and $\mathcal{F}$ contains $m$ accepting sets. Each of the NGAs $A'_0, \ldots, A'_{m-1}$ has at most $2n$ states, and an acceptance condition containing at most $m$ acceptance sets. So each of the NBAs $A'_0, \ldots, A'_{m-1}$ has at most $2n^2$ states, and the final NBA has at most $2n^2m + 1$ states. Observe in particular that while the conversion from NBA to NMA involves a possibly exponential blow-up, the conversion NMA to NBA does not.

It can be shown that the exponential blow-up in the conversion from NBA to NMA cannot be avoided, which leads to the next step in our quest: is there a class of $\omega$-automata that (1) recognizes all $\omega$-regular languages, (2) has a determinization procedure, and (3) has polynomial conversion algorithms to and from NBA.

11.4.3 Rabin automata

The acceptance condition of a nondeterministic Rabin automaton (NRA) is a set of pairs $\mathcal{F} = \langle F_0, G_0 \rangle, \ldots, \langle F_m, G_m \rangle$, where the $F_i$'s and $G_i$'s are sets of states. A run $\rho$ is accepting if there is a pair $\langle F_i, G_i \rangle$ such that $\rho$ visits some state of $F_i$ infinitely often and all states of $G_i$ finitely often. Formally, $\rho$ is accepting if there is $i \in \{1, \ldots, m\}$ such that $\text{inf}(\rho) \cap F_i \neq \emptyset$ and $\text{inf}(\rho) \cap G_i = \emptyset$.

A Büchi automaton can be easily transformed into a Rabin automaton and vice versa, without any exponential blow-up.
A Büchi condition $\{q_1, \ldots, q_k\}$ corresponds to the Rabin condition $[(q_1, \emptyset), \ldots, (q_k, \emptyset)]$.

**NRA → NBA.** Given a Rabin automaton $A = (Q, \Sigma, q_0, \delta, \{(F_0, G_0), \ldots, (F_m, G_m)\})$, it is easy to see that, as for Muller automata, we have $L_{\omega}(A) = \bigcup_{i=0}^{m-1} L_{\omega}(A_i)$, where $A_i = (Q, \Sigma, q_0, \delta, \{(F_i, G_i)\})$. In this case we directly translate each $A_i$ into an NBA. Since an accepting run $\rho$ of $A_i$ satisfies $\inf(\rho) \cap G_i = \emptyset$, from some point on the run only visits states of $Q \setminus G_i$. So $\rho$ consists of an initial finite part, say $\rho_0$, that may visit all states, and an infinite part, say $\rho_1$, that only visits states of $Q \setminus G_i$. Again, we take two copies of $A_i$. Intuitively, $A'_i$ simulates $\rho$ by executing $\rho_0$ in the first copy, and $\rho_1$ in the second. The condition that $\rho_1$ must visit some state of $F_i$ infinitely often is enforced by taking $F_i$ as Büchi condition.

**Example 11.10** Figure 11.10 can be reused to illustrate the conversion of a Rabin into a Büchi automaton. Consider the automaton on the left, but this time with Rabin accepting condition $\{(F_0, G_0), (F_1, G_1)\}$, where $F_0 = \{q\} = G_1$, and $G_0 = \{r\} = F_1$. Then the automaton accepts the $\omega$-words that contain finitely many $a$s or finitely many $b$s. The Büchi automata $A'_0, A'_1$ are as shown at the top-right part, but now instead of NGAs they are NBAs with accepting states $\{q, 1\}$ and $\{r, 1\}$, respectively. The final NBA is exactly the same one.

For the complexity, observe that each of the $A'_i$ has at most $2n$ states, and so the final Büchi automaton has at most $2nm + 1$ states.

The proof that NRAs are as expressive as DRAs is complicated. Since NRAs and NBAs are equally expressive by the conversions above, it suffices to show that DRAs are as expressive as NBAs. We only present the result, without proof.

**Theorem 11.11 (Safra)** Any NBA with $n$ states can be effectively transformed into a DRA of size $n^{O(n)}$ and $O(n)$ accepting pairs.

The accepting condition of Rabin automata is not “closed under negation”. Recall the condition:

there is $i \in \{1, \ldots, m\}$ such that $\inf(\rho) \cap F_i \neq \emptyset$ and $\inf(\rho) \cap G_i = \emptyset$

The negation is of the form

for every $i \in \{1, \ldots, m\}$: $\inf(\rho) \cap F_i = \emptyset$ or $\inf(\rho) \cap G_i \neq \emptyset$

This is called the **Streett condition**. We finish the chapter with a short discussion of Street automata.

**Streett automata**

The final class of $\omega$-automata we consider are Streett automata. In a Streett automaton, the acceptance condition is again a set of pairs $\{(F_1, G_1), \ldots, (F_m, G_m)\}$, where $F_i, G_i$ are sets of states. A run $\rho$ is accepting if for every pair $(F_i, G_i)$, if $\rho$ visits some state of $F_i$ infinitely often, then it also
visits some state of $G_i$ infinitely often. Formally, a run $\rho$ is accepting if for every $i \in \{1, \ldots, m\}$ \(\inf(\rho) \cap F_i \neq \emptyset\) implies \(\inf(\rho) \cap G_i \neq \emptyset\) (equivalently, if \(\inf(\rho) \cap F_i = \emptyset\) or \(\inf(\rho) \cap G_i \neq \emptyset\) for every \(i \in \{1, \ldots, m\}\)). The reader has probably observed that this is the dual of Rabin’s condition: a run satisfies the Streett condition \(\{\langle F_1, G_1 \rangle, \ldots, \langle F_m, G_m \rangle\}\) if and only if it does not satisfy the Rabin condition \(\{\langle F_1, G_1 \rangle, \ldots, \langle F_m, G_m \rangle\}\).

A Büchi automaton can be easily transformed into a Streett automaton and vice versa. However, the conversion from Streett to Büchi is exponential.

#### NBA → NSA.
A Büchi condition \(\{q_1, \ldots, q_k\}\) corresponds to the Streett condition \(\{\langle Q, \{q_1, \ldots, q_k\}\rangle\}\).

#### NSA → NBA.
We can transform into an NBA by following the path NSA → NMA → NBA. This yields for a NSA with \(n\) states an NBA with \(2^{2n}\) states. It can be shown that the exponential blow-up is unavoidable; in other words, Streett automata can be exponentially more succinct than Büchi automata.

**Example 11.12** Let \(\Sigma = \{0, 1, 2\}\). For \(n \geq 1\), we represent an infinite sequence \(x_1, x_2, \ldots\) of vectors of dimension \(n\) with components in \(\Sigma\) by the \(\omega\)-word \(x_1x_2\ldots\) over \(\Sigma^n\). Let \(L_n\) be the language in which, for each component \(i \in \{1, \ldots, n\}\), \(x_j(i) = 1\) for infinitely many \(j\)’s if and only if \(x_k(i) = 2\) for infinitely many \(k\)’s. It is easy to see that \(L_n\) can be accepted by a NSA with \(3n\) states and \(2n\) accepting pairs, but cannot be accepted by any NBA with less than \(2^n\) states. 

**Exercises**

**Exercise 86** A finite set of finite words is always a regular language, but a finite set of \(\omega\)-words is not always an \(\omega\)-regular language: find an \(\omega\)-word \(w \in \{a, b\}^\omega\) such that no Büchi automaton recognizes the language \(\{w\}\).

**Exercise 87** Construct Büchi automata and \(\omega\)-regular expressions recognizing the following languages over the alphabet \(\{a, b, c\}\).

1. \(\{w \in \{a, b, c\}^\omega \mid \{a, b\} \supseteq \inf(w)\}\)

2. \(\{w \in \{a, b, c\}^\omega \mid \{a, b\} = \inf(w)\}\)

3. \(\{w \in \{a, b, c\}^\omega \mid \{a, b\} \subseteq \inf(w)\}\)

4. \(\{w \in \{a, b, c\}^\omega \mid \{a, b, c\} = \inf(w)\}\)

5. \(\{w \in \{a, b, c\}^\omega \mid \text{if } a \in \inf(w) \text{ then } \{b, c\} \subseteq \inf(w)\}\)

*Hint:* It may be easier to construct a generalized Büchi automaton first and then transform it into a Büchi automaton.
Exercise 88 Find algorithms for the following decision problems:

- Given finite words $u, v, x, y \in \Sigma^*$, decide whether the $\omega$-words $u v^\omega$ and $x y^\omega$ are equal.
- Given a Büchi automaton $A$ and finite words $u, v$, decide whether $A$ accepts the $\omega$-word $u v^\omega$.

Exercise 89 Find $\omega$-regular expressions for the following languages:

1. $\{w \in \{a, b\}^\omega | k \text{ is even for each substring } ba^kB \text{ of } w\}$
2. $\{w \in \{a, b\}^\omega | w \text{ has no occurrence of } bab\}$

Exercise 90 A Büchi automaton is deterministic in the limit if all its accepting states and their descendants are deterministic states. Formally, $A = (\Sigma, Q, Q^0, \delta, \alpha)$ is deterministic in the limit if $|\delta(q, \sigma)| \leq 1$ for every state $q \in Q$ that is reachable from some state of $\alpha$, and for every $\sigma \in \Sigma$. Prove that every language recognized by nondeterministic Büchi automata is also accepted by Büchi automata deterministic in the limit.

Exercise 91 The parity acceptance condition for $\omega$-automata is defined as follows. Every state $q$ of the automaton is assigned a natural number $n_q$. A run $\rho$ is accepting if the number $\max\{n_s | s \in \text{inf}(\rho)\}$ is even.

- Find a parity automaton accepting the language $L = \{w \in \{a, b\}^\omega | w \text{ has exactly two occurrences of } ab\}$.
- Show that each language accepted by a parity automaton is also accepted by a Rabin automaton and vice versa.
11.4. OTHER CLASSES OF $\omega$-AUTOMATA

$F = \{ F_0, F_1 \}$

$F_0 = \{ q \}$

$F_1 = \{ r \}$

$\mathcal{F}_0 = \{ [q, 1] \}$

$\mathcal{F}_1 = \{ [r, 1] \}$

Figure 11.10: A Muller automaton and its conversion into a NBA
Chapter 12

Boolean operations: Implementations

The list of operations of Chapter 4 can be split into two parts, with the the boolean operations union, intersection, and complement in the first part, and the emptiness, inclusion, and equality tests in the second. This chapter deals with the boolean operations, while the tests are discussed in the next one. Observe that we now leave the membership test out. Observe that a test for arbitrary \( \omega \)-words does not make sense, because no description formalism can represent arbitrary \( \omega \)-words. For \( \omega \)-words of the form \( w_1(w_2)^\omega \), where \( w_1, w_2 \) are finite words, membership in an \( \omega \)-regular language \( L \) can be implemented by checking if the intersection of \( L \) and \( \{w_1(w_2)^\omega\} \) is empty.

We provide implementations for \( \omega \)-languages represented by NBAs and NGAs. We do not discuss implementations on DBAs, because they cannot represent all \( \omega \)-regular languages.

In Section 12.1 we show that union and intersection can be easily implemented using constructions already presented in Chapter 2. The rest of the chapter is devoted to the complement operation, which is more involved.

12.1 Union and intersection

As already observed in Chapter 2, the algorithm for union of regular languages represented as NFAs also works for NBAs and for NGAs.

One might be tempted to think that, similarly, the intersection algorithm for NFAs also works for NBAs. However, this is not the case. Consider the two Büchi automata \( A_1 \) and \( A_2 \) of Figure 12.1. The Büchi automaton \( A_1 \cap A_2 \) obtained by applying algorithm \textit{IntersNFA}(\( A_1, A_2 \)) in page

![Figure 12.1: Two Büchi automata accepting the language \( a^\omega \)](image-url)
more precisely, by interpreting the output of the algorithm as a Büchi automaton) is shown in Figure 12.2. It has no accepting states, and so $L_\omega(A_1) = L_\omega(A_2) = \{a^\omega\}$, but $L_\omega(A_1 \cap A_2) = \emptyset$.

![Figure 12.2: The automaton $A_1 \cap A_2$](image)

What happened? A run $\rho$ of $A_1 \cap A_2$ on an $\omega$-word $w$ is the result of pairing runs $\rho_1$ and $\rho_2$ of $A_1$ and $A_2$ on $w$. Since the accepting set of $A_1 \cap A_2$ is the cartesian product of the accepting sets of $A_1$ and $A_2$, $\rho$ is accepting if $\rho_1$ and $\rho_2$ simultaneously visit accepting states infinitely often. This condition is too strong, and as a result $L_\omega(A_1 \cap A_2)$ can be a strict subset of $L_\omega(A_1) \cap L_\omega(A_2)$.

This problem is solved by means of the observation we already made when dealing with NGAs: the run $\rho$ visits states of $F_1$ and $F_2$ infinitely often if and only if the following two conditions hold:

1. $\rho$ eventually visits $F_1$; and
2. every visit of $\rho$ to $F_1$ is eventually followed by a visit to $F_2$ (with possibly further visits to $F_1$ in-between), and every visit to $F_2$ is eventually followed by a visit to $F_1$ (with possibly further visits to $F_1$ in-between).

We proceed as in the translation NGA $\rightarrow$ NBA. Intuitively, we take two “copies” of the pairing $[A_1, A_2]$, and place them one of top of the other. The first and second copies of a state $[q_1, q_2]$ are called $[q_1, q_2, 1]$ and $[q_1, q_2, 2]$, respectively. The transitions leaving the states $[q_1, q_2, 1]$ such that $q_1 \in F_1$ are redirected to the corresponding states of the second copy, i.e., every transition of the form $[q_1, q_2, 1] \xrightarrow{a} [q_1', q_2, 1]$ is replaced by $[q_1, q_2, 1] \xrightarrow{a} [q_1', q_2', 2]$. Similarly, the transitions leaving the states $[q_1, q_2, 2]$ such that $q_2 \in F_2$ are redirected to the first copy. We choose $[q_{01}, q_{02}, 1]$, as initial state, and declare the states $[q_1, q_2, 1]$ such that $q_1 \in F_1$ as accepting.

**Example 12.1** Figure 12.3 shows the result of the construction for the NBAs $A_1$ and $A_2$ of Figure 12.1, after removing the states that are not reachable from the initial state. Since $q_0$ is not an accepting state of $A_1$, the transition $[q_0, r_0, 1] \xrightarrow{a} [q_1, r_1, 1]$ is not redirected. However, since $q_1$ is

![Figure 12.3: The NBA $A_1 \cap_\omega A_2$ for the automata $A_1$ and $A_2$ of Figure 12.1](image)
12.1. UNION AND INTERSECTION

an accepting state, transitions leaving \([q_1, r_1, 1]\) must jump to the second copy, and so we replace \([q_1, r_1, 1] \xrightarrow{a} [q_0, r_0, 1]\) by \([q_1, r_1, 1] \xrightarrow{a} [q_0, r_0, 2]\). Finally, since \(r_0\) is an accepting state of \(A_2\), transitions leaving \([q_0, r_0, 2]\) must return to the first copy, and so we replace \([q_0, r_0, 2] \xrightarrow{a} [q_1, r_1, 2]\) by \([q_0, r_0, 2] \xrightarrow{a} [q_1, r_1, 1]\). The only accepting state is \([q_1, r_1, 1]\), and the language accepted by the NBA is \(a^ω\).

To see that the construction works, observe first that a run \(\rho\) of this new NBA still corresponds to the pairing of two runs \(\rho_1\) and \(\rho_2\) of \(A_1\) and \(A_2\), respectively. Since all transitions leaving the accepting states jump to the second copy, \(\rho\) is accepting iff it visits both copies infinitely often, which is the case iff \(\rho_1\) and \(\rho_2\) visit states of \(F_1\) and \(F_2\), infinitely often, respectively.

Algorithm \(\text{IntersNBA}\), shown below, returns an NBA \(A_1 \cap_ω A_2\). As usual, the algorithm only constructs states reachable from the initial state.

\[
\text{IntersNBA}(A_1, A_2)
\]

Input: NBAs \(A_1 = (Q_1, \Sigma, δ_1, q_{01}, F_1)\), \(A_2 = (Q_2, \Sigma, δ_2, q_{02}, F_2)\)

Output: NBA \(A_1 \cap_ω A_2 = (Q, \Sigma, δ, q_0, F)\) with \(L_ω(A_1 \cap_ω A_2) = L_ω(A_1) \cap L_ω(A_2)\)

1. \(Q, δ, F \leftarrow \emptyset\)
2. \(q_0 \leftarrow [q_{01}, q_{02}, 1]\)
3. \(W \leftarrow \{[q_{01}, q_{02}, 1]\}\)
4. while \(W \neq \emptyset\) do
5. \quad pick \([q_1, q_2, i]\) from \(W\)
6. \quad add \([q_1, q_2, i]\) to \(Q'\)
7. \quad if \(q_1 \in F_1\) and \(i = 1\) then add \([q_1, q_2, 1]\) to \(F'\)
8. \quad for all \(a \in \Sigma\) do
9. \quad \quad for all \(q'_1 \in δ_1(q_1, a), q'_2 \in δ_2(q_2, a)\) do
10. \quad \quad \quad if \(i = 1\) and \(q_1 \notin F_1\) then
11. \quad \quad \quad \quad add \([(q_1, q_2, 1), a, [q'_1, q'_2, 1]]\) to \(δ\)
12. \quad \quad \quad \quad if \([q'_1, q'_2, 1] \notin Q'\) then add \([q'_1, q'_2, 1]\) to \(W\)
13. \quad \quad \quad if \(i = 1\) and \(q_1 \in F_1\) then
14. \quad \quad \quad \quad add \([(q_1, q_2, 1), a, [q'_1, q'_2, 2]]\) to \(δ\)
15. \quad \quad \quad \quad if \([q'_1, q'_2, 2] \notin Q'\) then add \([q'_1, q'_2, 2]\) to \(W\)
16. \quad \quad \quad if \(i = 2\) and \(q_2 \notin F_2\) then
17. \quad \quad \quad \quad add \([(q_1, q_2, 2), a, [q'_1, q'_2, 2]]\) to \(δ\)
18. \quad \quad \quad \quad if \([q'_1, q'_2, 2] \notin Q'\) then add \([q'_1, q'_2, 2]\) to \(W\)
19. \quad \quad \quad if \(i = 2\) and \(q_2 \in F_2\) then
20. \quad \quad \quad \quad add \([(q_1, q_2, 2), a, [q'_1, q'_2, 1]]\) to \(δ\)
21. \quad \quad \quad \quad if \([q'_1, q'_2, 1] \notin Q'\) then add \([q'_1, q'_2, 1]\) to \(W\)
22. \quad return \((Q, \Sigma, δ, q_0, F)\)

There is an important case in which the construction for NFAs can also be applied to NBAs,
namely when all the states of one of the two NBAs, say $A_1$, are accepting. In this case, the condition that two runs $\rho_1$ and $\rho_2$ on an $\omega$-word $w$ simultaneously visit accepting states infinitely often is equivalent to the weaker condition that does not require simultaneity: any visit of $\rho_2$ to an accepting state is a simultaneous visit of $\rho_1$ and $\rho_2$ to accepting states.

It is also important to observe a difference with the intersection for NFAs. In the finite word case, given NFAs $A_1, \ldots, A_k$ with $n_1, \ldots, n_k$ states, we can compute an NFA for $L(A_1) \cap \ldots \cap L(A_n)$ with at most $\prod_{i=1}^k n_i$ states by repeatedly applying the intersection operation, and this construction is optimal (i.e., there is a family of instances of arbitrary size such that the smallest NFA for the intersection of the languages has the same size). In the NBA case, however, the repeated application of $\text{IntersNBA}()$ is not optimal. Since $\text{IntersNBA}()$ introduces an additional factor of 2 in the number of states, for $L_\omega(A_1) \cap \ldots \cap L_\omega(A_k)$ it yields an NBA with $2^{k-1} \cdot n_1 \cdot \ldots \cdot n_k$ states. We obtain a better construction proceeding as in the translation NGA $\rightarrow$ NBA: we produce $k$ copies of $A_1 \times \ldots \times A_k$, and move from the $i$-th copy to the $(i+1)$-th copy when we hit an accepting state of $A_i$. This construction yields an NBA with $k \cdot n_1 \cdot \ldots \cdot n_k$ states.

### 12.2 Complement

So far we have been able to adapt the constructions for NFAs to NBAs. The situation is considerably more involved for complement.

#### 12.2.1 The problems of complement

Recall that for NFAs a complement automaton is constructed by first converting the NFA into a DFA, and then exchanging the final and non-final states of the DFA. For NBAs this approach breaks down completely:

(a) The subset construction does not preserve $\omega$-languages; i.e, a NBA and the result of applying the subset construction to it do not necessarily accept the same $\omega$-language.

The NBA on the left of Figure 12.4 accepts the empty language. However, the result of applying the subset construction to it, shown on the right, accepts $a^\omega$. Notice that both automata accept the same finite words.

![Figure 12.4: The subset construction does not preserve $\omega$-languages](image-url)
(b) The subset construction cannot be replaced by another determinization procedure, because no such procedure exists: As we have seen in Proposition 11.6, some languages are accepted by NBAs, but not by DBAs.

(c) The automaton obtained by exchanging accepting and non-accepting states in a given DBA does not necessarily recognize the complement of the language.

In Figure 12.1, \(A_2\) is obtained by exchanging final and non-final states in \(A_1\). However, both \(A_1\) and \(A_2\) accept the language \(a^\omega\). Observe that as automata for finite words they accept the words over the letter \(a\) of even and odd length, respectively.

Despite these discouraging observations, NBAs turn out to be closed under complement. For the rest of the chapter we fix an NBA \(A = (Q, \Sigma, \delta, q_0, F)\) with \(n\) states, and use Figure 12.5 as running example. Further, we abbreviate “infinitely often” to “i.o.”. We wish to build an automaton

![Diagram of automaton](attachment:diagram.png)

Figure 12.5: Running example for the complementation procedure

\(\overline{A}\) satisfying:

no path of \(\text{dag}(w)\) visits accepting states of \(A\) i.o.
if and only if
some run of \(w\) in \(\overline{A}\) visits accepting states of \(\overline{A}\) i.o.

We give a summary of the procedure. First, we define the notion of ranking. For the moment it suffices to say that a ranking of \(w\) is the result of decorating the nodes of \(\text{dag}(w)\) with numbers. This can be done in different ways, and so, while a word \(w\) has one single \(\text{dag} \ \text{dag}(w)\), it may have many rankings. The essential property of rankings will be:

no path of \(\text{dag}(w)\) visits accepting states of \(A\) i.o.
if and only if for some ranking \(R(w)\)
every path of \(\text{dag}(w)\) visits nodes of odd rank i.o.

In the second step we profit from the determinization construction for co-Büchi automata. Recall that the construction maps \(\text{dag}(w)\) to a run \(\rho\) of a new automaton such that: every path of \(\text{dag}(w)\) visits accepting states of \(A\) i.o. if and only if \(\rho\) visits accepting states of the new automaton i.o. We apply the same construction to map every ranking \(R(w)\) to a run \(\rho\) of a new automaton \(B\) such that
every path of \( \text{dag}(w) \) visits nodes of odd rank i.o. (in \( R(w) \))

if and only if

the run \( \rho \) visits states of \( B \) i.o.

This immediately implies \( L_\omega(B) = \overline{L_\omega(A)} \). However, the automaton \( B \) may in principle have an infinite number of states! In the final step, we show that a finite subautomaton \( \overline{A} \) of \( B \) already recognizes the same language as \( B \), and we are done.

### 12.2.2 Rankings and ranking levels

Recall that, given \( w \in a^\omega \), the directly acyclic graph \( \text{dag}(w) \) is the result of bundling together the runs of \( A \) on \( w \). A ranking of \( \text{dag}(w) \) is a mapping \( R(w) \) that associates to each node of \( \text{dag}(w) \) a natural number, called a rank, satisfying the following two properties:

(a) the rank of a node is greater than or equal to the rank of its children, and

(b) the rank of an accepting node is even.

![Graphs](image)

Figure 12.6: Rankings for \( \text{dag}(aba^\omega) \) and \( \text{dag}((ab)^\omega) \)

The ranks of the nodes in an infinite path form a non-increasing sequence, and so there is a node such that all its (infinitely many) successors have the same rank; we call this number the stable rank of the path. Figure 12.6 shows rankings for \( \text{dag}(aba^\omega) \) and \( \text{dag}((ab)^\omega) \). Both have one single infinite path with stable rank 1 and 0, respectively. We now prove the fundamental property of rankings:

**Proposition 12.2** No path of \( \text{dag}(w) \) visits accepting nodes of \( A \) i.o. if and only if for some ranking \( R(w) \) every infinite path of \( \text{dag}(w) \) visits nodes of odd rank i.o.
12.2. COMPLEMENT

**Proof:** If all infinite paths of a ranking \( R \) have odd stable rank, then each of them contains only finitely many nodes with even rank. Since accepting nodes have even ranks, no path visits accepting nodes i.o.

For the other direction, assume that no path of \( \text{dag}(w) \) visits accepting nodes of \( A \) i.o. Give each accepting node \( \langle q, l \rangle \) the rank \( 2k \), where \( k \) is the maximal number of accepting nodes in the paths starting at \( \langle q, l \rangle \), and give a non-accepting nodes rank \( 2k + 1 \), where \( 2k \) is the maximal rank of its descendants with even rank. In the ranking so obtained every infinite path visits nodes of even rank only finitely often, and therefore it visits nodes of odd rank i.o. 

Recall that the \( i \)-th level of \( \text{dag}(w) \) is defined as the set of nodes of \( \text{dag}(w) \) of the form \( \langle q, i \rangle \). Let \( R \) be the set of all ranking levels. Any ranking \( r \) of \( \text{dag}(w) \) can be decomposed into an infinite sequence \( lr_1, lr_2, \ldots \) of level rankings by defining \( lr_i(q) = r(\langle q, i \rangle) \) if \( \langle q, i \rangle \) is a node of \( \text{dag}(w) \), and \( lr_i(q) = \perp \) otherwise. For example, if we represent a level ranking \( lr \) of our running example by the column vector 

\[
\begin{bmatrix}
lr(q_0) \\
lr(q_1)
\end{bmatrix},
\]

then the rankings of Figure 12.6 correspond to the sequences 

\[
\begin{bmatrix}
2 \\
\perp
\end{bmatrix}, \begin{bmatrix}
\perp \\
2
\end{bmatrix}, \begin{bmatrix}
1 \\
\perp
\end{bmatrix}, \begin{bmatrix}
1 \\
0
\end{bmatrix}, \begin{bmatrix}
1 \\
\perp
\end{bmatrix}
\]

For two level rankings \( lr \) and \( lr' \) and a letter \( a \in \Sigma \), we write \( lr \xrightarrow{a} lr' \) if for every \( q' \in Q \):

- \( lr'(q') = \perp \) iff no \( q \in Q \) satisfies \( q \xrightarrow{a} q' \), and
- \( lr(q) \geq lr'(q') \) for every \( q \in Q \) satisfying \( q \xrightarrow{a} q' \).

12.2.3 A (possible infinite) complement automaton

We construct an NBA \( B \) an infinite number of states (and many initial states) whose runs on an \( \omega \)-word \( w \) are the rankings of \( \text{dag}(w) \). The automaton accepts a ranking \( R \) iff every infinite path of \( R \) visits nodes of odd rank i.o.

We start with an automaton without any accepting condition:

- The states are all the possible ranking levels.
- The initial states are the levels \( rl_n \) defined by: \( rl_n(q_0) = n \), and \( rl_n(q) = \perp \) for every \( q \neq q_0 \).
- The transitions are the triples \( (rl, a, rl') \), where \( rl \) and \( rl' \) are ranking levels, \( a \in \Sigma \), and \( rl \xrightarrow{a} rl' \) holds.
The runs of this automaton on \( w \) clearly correspond to the rankings of \( \text{dag}(w) \). Now we apply the same construction we used for determinization of co-Büchi automata. We decorate the ranking levels with a set of ‘owing’ states, namely those that owe a visit to a state of odd rank, and take as accepting states the breakpoints i.e., the levels with an empty set of ‘owing’ states. We get the Büchi automaton \( B \):

- The states are all pairs \([rl, O]\), where \( rl \) is a ranking level and \( O \) is a subset of the states \( q \) for which \( rl(q) \in \mathbb{N} \).

- The initial states are all pairs of the form \([lr, \{q_0\}]\) where \( lr(q_0) \) is an even number and \( lr_0(q) = \bot \) for every \( q \neq q_0 \), and of the form \([lr, \emptyset]\), where \( lr(q_0) \) is an odd number and \( lr_0(q) = \bot \) for every \( q \neq q_0 \).

- The transitions are the triples \([rl, O] \delta a [rl', O']\) if \( lr \xrightarrow{a} rl' \) and
  - \( O \neq \emptyset \) and \( O' = \{q' \in \delta(O, a) \mid lr'(q') \text{ is even}\} \), or
  - \( O = \emptyset \) and \( O' = \{q' \in Q \mid lr'(q') \text{ is even}\} \).

- The accepting states (breakpoints) are the pairs \([rl, \emptyset]\).

\( B \) accepts a ranking iff it contains infinitely many breakpoints. As we saw in the construction for co-Büchi automata, this is the case iff every infinite path of \( \text{dag}(w) \) visits nodes of odd rank i.o., and so iff \( A \) does not accept \( w \).

The remaining problems with this automaton are that its number of states is infinite, and that it has many initial states. Both can be solved by proving the following assertion: there exists a number \( k \) such that for every word \( w \), if \( \text{dag}(w) \) admits an odd ranking, then it admits an odd ranking whose initial node \( \langle q_0, 0 \rangle \) has rank \( k \). (Notice that, since ranks cannot increase along paths, every node has rank at most \( k \).) If we are able to prove this, then we can eliminate all states corresponding to ranking levels in which some node is mapped to a number larger than \( k \): they are redundant. Moreover, the initial state is now fixed: it is the level ranking that maps \( q_0 \) to \( k \) and all other states to \( \bot \).

**Proposition 12.3** Let \( n \) be the number of states of \( A \). For every word \( w \in \Sigma^\omega \), if \( w \) is rejected by \( A \) then \( \text{dag}(w) \) has a ranking such that

(a) every infinite path of \( \text{dag}(w) \) visits nodes of odd rank i.o., and

(b) the initial node \( \langle q_0, 0 \rangle \) has rank \( 2n \).

**Proof:** In the proof we call a ranking satisfying (a) an odd ranking. Assume \( w \) is rejected by \( A \). We construct an odd ranking in which \( \langle q_0, 0 \rangle \) has rank at most \( 2n \). Then we can just change the rank of the initial node to \( 2n \), since the change preserves the properties of a ranking.

In the sequel, given two DAGS \( D, D' \), we denote by \( D' \subseteq D \) the fact that \( D' \) can be obtained from \( D \) through deletion of some nodes and their adjacent edges.
Assume that \( A \) rejects \( w \). We describe an odd ranking for \( \text{dag}(w) \). We say that a node \( \langle q, l \rangle \) is \textit{red} in a (possibly finite) \( \text{dag} D \subseteq \text{dag}(w) \) iff only finitely many nodes of \( D \) are reachable from \( \langle q, l \rangle \). The node \( \langle q, l \rangle \) is \textit{yellow} in \( D \) iff all the nodes reachable from \( \langle q, l \rangle \) (including itself) are not accepting. In particular, yellow nodes are not accepting. Observe also that the children of a red node are red, and the children of a yellow node are red or yellow. We inductively define an infinite sequence \( D_0 \supseteq D_1 \supseteq D_2 \supseteq \ldots \) of \textsc{dags} as follows:

- \( D_0 = \text{dag}(w); \)
- \( D_{2i+1} = D_{2i} \setminus \{ \langle q, l \rangle \mid \langle q, l \rangle \text{ is red in } D_{2i} \}; \)
- \( D_{2i+2} = D_{2i+1} \setminus \{ \langle q, l \rangle \mid \langle q, l \rangle \text{ is yellow in } D_{2i+1} \}. \)

Figure 12.7 shows \( D_0, D_1, \) and \( D_2 \) for \( \text{dag}(aba^\omega) \). \( D_3 \) is the empty dag.

Consider the function \( f \) that assigns to each node of \( \text{dag}(w) \) a natural number as follows:

\[
f(\langle q, l \rangle) = \begin{cases} 
2i & \text{if } \langle q, l \rangle \text{ is red in } D_{2i} \\
2i + 1 & \text{if } \langle q, l \rangle \text{ is yellow in } D_{2i+1}
\end{cases}
\]

We prove that \( f \) is an odd ranking. The proof is divided into three parts:

(1) \( f \) assigns all nodes a number in the range \([0 \ldots 2n]\).
(2) If \( \langle q', l' \rangle \) is a child of \( \langle q, l \rangle \), then \( f(\langle q', l' \rangle) \leq f(\langle q, l \rangle) \).

(3) If \( \langle q, l \rangle \) is an accepting node, then \( f(\langle q, l \rangle) \) is even.

**Part (1).** We show that for every \( i \geq 0 \) there exists a number \( l_i \) such that for all \( l \geq l_i \), the dag \( D_{2i} \) contains at most \( n - i \) nodes of the form \( \langle q, l \rangle \). This implies that \( D_{2n} \) is finite, and so that \( D_{2n+1} \) is empty, which in turn implies that \( f \) assigns all nodes a number in the range \([0 \ldots 2n]\).

The proof is by an induction on \( i \). The case where \( i = 0 \) follows from the definition of \( G_0 \): indeed, in \( \text{dag}(w) \) all levels \( l \geq 0 \) have at most \( n \) nodes of the form \( \langle q, l \rangle \). Assume now that the hypothesis holds for \( i \); we prove it for \( i + 1 \). Consider the dag \( D_{2i} \). If \( D_{2i} \) is finite, then \( D_{2i+1} \) is empty; \( D_{2i+2} \) is empty as well, and we are done. So assume that \( D_{2i} \) is infinite. We claim that \( D_{2i+1} \) contains some yellow node. Assume, by way of contradiction, that no node in \( D_{2i+1} \) is yellow. Since \( D_{2i} \) is infinite, \( D_{2i+1} \) is also infinite. Moreover, since \( D_{2i+1} \) is obtained by removing all red nodes from \( D_{2i} \), every node of \( D_{2i+1} \) has at least one child. Let \( \langle q_0, l_0 \rangle \) be an arbitrary node of \( D_{2i+1} \). Since, by the assumption, it is not yellow, there exists an accepting node \( \langle q'_0, l'_0 \rangle \) reachable from \( \langle q_0, l_0 \rangle \). Let \( \langle q_1, l_1 \rangle \) be a child of \( \langle q'_0, l'_0 \rangle \). By the assumption, \( \langle q_1, l_1 \rangle \) is also not yellow, and so there exists an accepting node \( \langle q'_1, l'_1 \rangle \) reachable from \( \langle q_1, l_1 \rangle \). We can thus construct an infinite sequence of nodes \( \langle q_j, l_j \rangle, \langle q'_j, l'_j \rangle \) such that for all \( i \) the node \( \langle q'_i, l'_i \rangle \) is accepting, reachable from \( \langle q_i, l_i \rangle \), and \( \langle q_{j+1}, l_{j+1} \rangle \) is a child of \( \langle q'_j, l'_j \rangle \). Such a sequence, however, corresponds to a path in \( \text{dag}(w) \) visiting infinitely many accepting nodes, which contradicts the assumption that \( A \) rejects \( w \), and the claim is proved.

So, let \( \langle q, l \rangle \) be a yellow node in \( D_{2i+1} \). We claim that we can take \( l_{i+1} = l \), that is, we claim that for every \( j \geq l \) the dag \( D_{2i+2} \) contains at most \( n - (i + 1) \) nodes of the form \( \langle q, j \rangle \). Since \( \langle q, l \rangle \) is in \( D_{2i+1} \), it is not red in \( D_{2i} \). Thus, infinitely many nodes of \( D_{2i} \) are reachable from \( \langle q, l \rangle \). By König’s Lemma, \( D_{2i} \) contains an infinite path \( \langle q, l \rangle, \langle q_1, l + 1 \rangle, \langle q_2, l + 2 \rangle, \ldots \). For all \( k \geq 1 \), infinitely many nodes of \( D_{2i} \) are reachable from \( \langle q_k, l + k \rangle \), and so \( \langle q_k, l + k \rangle \) is not red in \( D_{2i} \). Therefore, the path \( \langle q, l \rangle, \langle q_1, l + 1 \rangle, \langle q_2, l + 2 \rangle, \ldots \) exists also in \( D_{2i+1} \). Recall that \( \langle q, l \rangle \) is yellow. Hence, being reachable from \( \langle q, l \rangle \), all the nodes \( \langle q_k, l + k \rangle \) in the path are yellow as well. Therefore, they are not in \( D_{2i+2} \). It follows that for all \( j \geq l \) the number of nodes of the form \( \langle q, j \rangle \) in \( D_{2i+2} \) is strictly smaller than their number in \( D_{2i} \), which, by the induction hypothesis, is \( n - i \). So there are at most \( n - (i + 1) \) nodes of the form \( \langle q, j \rangle \) in \( D_{2i+2} \), and the claim is proved.

**Part (2).** Follows from the fact that the children of a red node in \( D_{2i} \) are red, and the children of a yellow node in \( D_{2i+1} \) are yellow. Therefore, if a node has rank \( i \), all its successors have rank at most \( i \) or lower.

**Part (3).** Nodes that get an odd rank are yellow at \( D_{2i+1} \) for some \( i \), and so not accepting. \( \square \)

**Example 12.4** We construct the complements \( \overline{A_1} \) and \( \overline{A_2} \) of the two possible NBAs over the alphabet \( \{a\} \) having one state and one transition: \( B_1 = (\{q\}, \{a\}, \delta, \{q\}) \) and \( B_2 = (\{q\}, \{a\}, \delta, \{q\}, \emptyset) \),
where $\delta(q,a) = \{q\}$. The only difference between $B_1$ and $B_2$ is that the state $q$ is accepting in $B_1$, but not in $B_2$. We have $L_{\omega_1}(A_1) = a^\omega$ and $L_{\omega_2}(A_2) = \emptyset$.

We begin with $B_1$. A state of $B_1$ is a pair $\langle lr, O \rangle$, where $lr$ is the rank of node $q$ (since there is only one state, we can identify $lr$ and $lr(q)$). The initial state is $\langle 2, \{q\} \rangle$, because $q$ has even rank and so it “owes” a visit to a node of odd rank. Let us compute the successors of $\langle 2, \{q\} \rangle$ under the letter $a$. Let $\langle lr', O' \rangle$ be a successor. Since $\delta(q,a) = \{q\}$, we have $lr' \neq \bot$, and since $q$ is accepting, we have $lr' \neq 1$. So either $lr' = 0$ or $lr' = 2$. In both cases the visit to a node of odd rank is still “owed”, which implies $O' = \{q\}$. So the successors of $\langle 2, \{q\} \rangle$ are $\langle 2, \{q\} \rangle$ and $\langle 0, \{q\} \rangle$. Consider now the successors $\langle lr'', O'' \rangle$ of $\langle 0, \{q\} \rangle$. We have $lr'' \neq \bot$ and $lr'' \neq 1$ as before, but now, since ranks cannot increase a long a path, we also have $lr'' \neq 2$. So $lr'' = 0$, and, since the visit to the node of odd rank is still ‘owed”, the only successor of $\langle 0, \{q\} \rangle$ is $\langle 0, \{q\} \rangle$. Since the set of owing states is never empty, $B_1$ has no accepting states, and so it recognizes the empty language. $B_1$ is shown on the left of Figure 12.8. Let us now construct $B_2$. The difference with $B_2$ is that, since $q$ is no longer accepting, it can also have odd rank 1. So $\langle 2, \{q\} \rangle$ has three successors: $\langle 2, \{q\} \rangle$, $\langle 1, \emptyset \rangle$, and $\langle 0, \{q\} \rangle$. The successors of $\langle 1, \emptyset \rangle$ are $\langle 1, \emptyset \rangle$ and $\langle 0, \{q\} \rangle$, and the only successor of $\langle 0, \{q\} \rangle$ is $\langle 0, \{q\} \rangle$. The only accepting state is $\langle 1, \emptyset \rangle$, and $B_2$ recognizes $a^\omega$. □

The pseudocode for the complementation algorithm is shown below. In the code, $\mathcal{R}$ denotes the set of all level rankings, and $lr_0$ denotes the level ranking given by $lr(q_0) = 2|Q|$ and $lr(q) = \bot$ for every $q \neq q_0$. Recall also that $lr \xrightarrow{a} lr'$ holds if for every $q' \in Q$: $lr'(q') = \bot$ iff no $q \in Q$ satisfies $q \xrightarrow{a} q'$, and $lr(q) \geq lr'(q')$ for every $q \in Q$ satisfying $q \xrightarrow{a} q'$.
We exhibit a family \( \{ L_n \}_{n \geq 1} \) of infinitary languages such that \( L_n \) is accepted by an automaton with \( n + 2 \) states and any Büchi automaton accepting the complement of \( L_n \) has at least \( n! \in 2^{\Theta(n \log n)} \) states.

Let \( \Sigma_n = \{ 1, \ldots, n, \# \} \). We associate to a word \( w \in \Sigma_n^\omega \) the following directed graph \( G(w) \): the nodes of \( G(w) \) are \( 1, \ldots, n \) and there is an edge from \( i \) to \( j \) if \( w \) contains infinitely many occurrences of the word \( ij \). Define \( L_n \) as the language of infinite words \( w \in \Sigma_n^\omega \) for which \( G(w) \) has a cycle and define \( \overline{L}_n \) as the complement of \( L_n \).

We first show that for all \( n \geq 1 \), \( L_n \) is recognized by a Büchi automaton with \( n + 2 \) states. Let \( A_n \) be the automaton shown in Figure 12.9. We show that \( A_n \) accepts \( L_n \).

1. If \( w \in L_n \), then \( A_n \) accepts \( w \).

Choose a cycle \( a_{i_1} a_{i_2} \ldots a_{i_k} a_{i_1} \) of \( G(w) \). We construct an accepting run of \( A_n \) by picking \( q_{i_1} \) as initial state and iteratively applying the following rule:

If the current state is \( q_{i_j} \), stay there until the next occurrence of the word \( a_j a_{i_{j+1}} \) in \( w \), then use \( a_j \) to move to \( r \), and use \( a_{i_{j+1}} \) to move to \( q_{i_{j+1}} \).

By the definition of \( G(w) \), \( r \) is visited infinitely often, and so \( w \) is accepted.

2. If \( A_n \) accepts \( w \), then \( w \in L_n \).
12.2. COMPLEMENT

Let \( \rho \) be a run of \( A_n \) accepting \( w \), and let \( Q_\rho = \inf(\rho) \cap \{q_1, \ldots, q_n\} \). Since \( \rho \) is accepting, it cannot stay in any of the \( q_i \) forever, and so for each \( q_i \in Q_\rho \) there is \( q_j \in Q_\rho \) such that the sequence \( q_irq_j \) appears infinitely often in \( \rho \). Therefore, for every \( q_i \in Q_\rho \) there is \( q_j \in Q_\rho \) such that \( a_i a_j \) appears infinitely often in \( w \), or, in other words, such that \( (a_i, a_j) \in G(w) \). Since \( Q_\rho \) is finite, \( G(w) \) contains a cycle, and so \( w \in L_n \).

Proposition 12.5 For all \( n \geq 1 \), every NBA recognizing \( L_n \) has at least \( n! \) states.

Proof: We need some preliminaries. Given a permutation \( \tau = \langle \tau(1), \ldots, \tau(n) \rangle \) of \( \langle 1, \ldots, n \rangle \), we identify \( \tau \) and the word \( \tau(1) \ldots \tau(n) \). We make two observations:

(a) \( (\tau\#)^\omega \in \bar{L}_n \) for every permutation \( \tau \).

The edges of \( G((\tau\#)^\omega) \) are \( \langle \tau(1), \tau(a_2) \rangle, \langle \tau(a_2), \tau(a_3) \rangle, \ldots, \langle \tau(a_{n-1}), \tau(a_n) \rangle \), and so \( G((\tau\#)^\omega) \) is acyclic.

(b) If a word \( w \) contains infinitely many occurrences of two different permutations \( \tau \) and \( \tau' \) of \( 1 \ldots n \), then \( w \in L_n \).

Since \( \tau \) and \( \tau' \) are different, there are \( i \) and \( j \) in \( \{1, \ldots, n\} \) such that \( i \) precedes \( j \) in \( \tau \) and \( j \) precedes \( i \) in \( \tau' \). Since \( w \) contains infinitely many occurrences of \( \tau \), \( G(w) \) has a path from \( i \) to \( j \). Since it contains infinitely many occurrences of \( \tau' \), \( G(w) \) has a path from \( j \) to \( i \). So \( G(w) \) contains a cycle, and so \( w \in L_n \).

Now, let \( A \) be a Büchi automaton recognizing \( \bar{L}_n \), and let \( \tau, \tau' \) be two arbitrary permutations of \( \langle 1, \ldots, n \rangle \). By (a), there exist runs \( \rho \) and \( \rho' \) of \( A \) accepting \( (\tau\#)^\omega \) and \( (\tau'\#)^\omega \), respectively. We prove that the intersection of \( \inf(\rho) \) and \( \inf(\rho') \) is empty. This implies that \( A \) contains at least as many final states as permutations of \( \langle 1, \ldots, n \rangle \), which proves the Proposition.

We proceed by contradiction. Assume \( q \in \inf(\rho) \cap \inf(\rho') \). We build an accepting run \( \rho'' \) by “combining” \( \rho \) and \( \rho' \) as follows:
Starting from the initial state of $\rho$, go to $q$ following the run $\rho$.

(1) Starting from $q$, follow $\rho'$ until having gone through a final state, and having read at least once the word $\tau'$; then go back to $q$ (always following $\rho'$).

(2) Starting from $q$, follow $\rho$ until having gone through a final state, and having read at least once the word $\tau$; then go back to $q$ (always following $\rho$).

(3) Go to (1).

The word accepted by $\rho''$ contains infinitely many occurrences of both $\tau$ and $\tau'$. By (b), this word belongs to $L_n$, contradicting the assumption that $A$ recognizes $\overline{L}_n$.

Exercises

**Exercise 92** Show that for every DBA $A$ with $n$ states there is an NBA $B$ with $2n$ states such that $L_\omega(B) = \overline{L}_\omega(A)$.

**Exercise 93** Give algorithms that directly complement deterministic Muller and parity automata, without going through Büchi automata.

**Exercise 94** Let $A = (Q, \Sigma, q_0, \delta, \{\langle F_0, G_0 \rangle, \ldots, \langle F_{m-1}, G_{m-1} \rangle\})$ be deterministic. Which is the relation between the languages recognized by $A$ as a deterministic Rabin automaton and as a deterministic Streett automaton?

**Exercise 95** Consider Büchi automata with universal accepting condition (UBA): an $\omega$-word $w$ is accepted if every run of the automaton on $w$ is accepting, i.e., if every run of the automaton on $w$ visits final states infinitely often.

Recall that automata on finite words with existential and universal accepting conditions recognize the same languages. Prove that is no longer the case for automata on $\omega$-words by showing that for every UBA there is a DBA automaton that recognizes the same language. (This implies that the $\omega$-languages recognized by UBAs are a proper subset of the $\omega$-regular languages.)

**Hint:** On input $w$, the DBA checks that every path of $\text{dag}(w)$ visits some final state infinitely often. The states of the DBA are pairs $(Q', O)$ of sets of the UBA where $O \subseteq Q'$ is a set of “owing” states (see below). Loosely speaking, the transition relation is defined to satisfy the following property: after reading a prefix $w'$ of $w$, the DBA is at the state $(Q', O)$ given by:

- $Q'$ is the set of states reached by the runs of the UBA on $w'$.
- $O$ is the subset of states of $Q'$ that “owe” a visit to a final state of the UBA. (See the construction for the complement of a Büchi automaton.)
Chapter 13

Emptiness check: Implementations

We present efficient algorithms for the emptiness check. We fix an NBA $A = (Q, \Sigma, \delta, q_0, F)$. Since transition labels are irrelevant for checking emptiness, in this Chapter we redefine $\delta$ from a subset of $Q \times \Sigma \times Q$ into a subset of $Q \times Q$ as follows:

$$\delta := \{(q, q') \in Q \times Q \mid (q, a, q') \in \delta \text{ for some } a \in \Sigma\}$$

Since in many applications we have to deal with very large Büchi automata, we are interested in on-the-fly algorithms that do not require to know the Büchi automaton in advance, but check for emptiness while constructing it. More precisely, we assume the existence of an oracle that, provided with a state $q$, returns the set $\delta(q)$.

We need a few graph-theoretical notions. If $(q, r) \in \delta$, then $r$ is a successor of $q$ and $q$ is a predecessor of $r$. A path is a sequence $q_0, q_1, \ldots, q_n$ of states such that $q_{i+1}$ is a successor of $q_i$ for every $i \in \{0, \ldots, n-1\}$; we say that the path leads from $q_0$ to $q_n$. Notice that a path may consist of only one state; in this case, the path is empty, and leads from a state to itself. A cycle is a path that leads from a state to itself. We write $q \sim r$ to denote that there is a path from $q$ to $r$.

Clearly, $A$ is nonempty if it has an accepting lasso, i.e., a path $q_0q_1 \ldots q_{n-1}q_n$ such that $q_n = q_i$ for some $i \in \{0, \ldots, n-1\}$, and at least one of $\{q_i, q_{i+1}, \ldots, q_{n-1}\}$ is accepting. The lasso consists of a path $q_0 \ldots q_i$, followed by a nonempty cycle $q_{i+1} \ldots q_{n-1}q_i$. We are interested in emptiness checks that on input $A$ report EMPTY or NONEMPTY, and in the latter case return an accepting lasso, as a witness of nonemptiness.

13.1 Algorithms based on depth-first search

We present two emptiness algorithms that explore $A$ using depth-first search (DFS). We start with a brief description of depth-first search and some of its properties.

A depth-first search (DFS) of $A$ starts at the initial state $q_0$. If the current state $q$ still has unexplored outgoing transitions, then one of them is selected. If the transition leads to a not yet discovered state $r$, then $r$ becomes the current state. If all of $q$’s outgoing transitions have been
explored, then the search “backtracks” to the state from which \( q \) was discovered, i.e., this state becomes the current state. The process continues until \( q_0 \) becomes the current state again and all its outgoing transitions have been explored. Here is a pseudocode implementation (ignore the algorithm \( \text{DFS.Tree} \) for the moment).

### DFS(A)

**Input:** NBA \( A = (Q, \Sigma, \delta, q_0, F) \)

1. \( S \leftarrow \emptyset \)
2. \( \text{dfs}(q_0) \)
3. proc \( \text{dfs}(q) \)
4. add \( q \) to \( S \)
5. for all \( r \in \delta(q) \) do
6. if \( r \notin S \) then \( \text{dfs}(r) \)
7. return

### DFS.Tree(A)

**Input:** NBA \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** Time-stamped tree \( (S, T, d, f) \)

1. \( S \leftarrow \emptyset \)
2. \( T \leftarrow \emptyset; t \leftarrow 0 \)
3. \( \text{dfs}(q_0) \)
4. proc \( \text{dfs}(q) \)
5. \( t \leftarrow t + 1; \ d[q] \leftarrow t \)
6. add \( q \) to \( S \)
7. for all \( r \in \delta(q) \) do
8. if \( r \notin S \) then
9. add \( (q, r) \) to \( T \); \( \text{dfs}(r) \)
10. \( t \leftarrow t + 1; \ f[q] \leftarrow t \)
11. return

Observe that \( \text{DFS} \) is nondeterministic, because we do not fix the order in which the states of \( \delta(q) \) are examined by the for-loop. Since, by hypothesis, every state of an automaton is reachable from the initial state, we always have \( S = Q \) after termination. Moreover, after termination every state \( q \neq q_0 \) has a distinguished input transition, namely the one that, when explored by the search, led to the discovery of \( q \). It is well-known that the graph with states as nodes and these transitions as edges is a tree with root \( q_0 \), called a **DFS-tree**. If some path of the DFS-tree leads from \( q \) to \( r \), then we say that \( q \) is an **ascendant** of \( r \), and \( r \) is a **descendant** of \( q \) (in the tree).

It is easy to modify \( \text{DFS} \) so that it returns a DFS-tree, together with **timestamps** for the states. The algorithm, which we call \( \text{DFS.Tree} \) is shown above. While timestamps are not necessary for conducting a search itself, many algorithms based on depth-first search use them for other purposes\(^1\). Each state \( q \) is assigned two timestamps. The first one, \( d[q] \), records when \( q \) is first discovered, and the second, \( f[q] \), records when the search finishes examining the outgoing transitions of \( q \). Since we are only interested in the relative order in which states are discovered and finished, we can assume that the timestamps are integers ranging between 1 and \( 2|Q| \). Figure 13.1 shows an example.

---

\(^1\)In the rest of the chapter, and in order to present the algorithms is more compact form, we omit the instructions for computing the timestamps, and just assume they are there.
In our analyses we also assume that at every time point a state is white, grey, or black. A state \( q \) is white during the interval \([0, d[q]]\), grey during the interval \((d[q], f[q])\), and black during the interval \((f[q], 2|Q|]\). So, loosely speaking, \( q \) is white, if it has not been yet discovered, grey if it has already been discovered but still has unexplored outgoing edges, or black if all its outgoing edges have been explored. It is easy to see that at all times the grey states form a path (the grey path) starting at \( q_0 \) and ending at the state being currently explored, i.e., at the state \( q \) such that \( \text{dfs}(q) \) is being currently executed; moreover, this path is always part of the DFS-tree.

![Diagram of an NBA and a possible run of DFS Tree](image.png)

Figure 13.1: An NBA (the labels of the transitions have been omitted), and a possible run of DFS Tree on it. The numeric intervals are the discovery and finishing times of the states, shown in the format \([d[q], f[q]]\).

We recall two important properties of depth-first search. Both follow easily from the fact that a procedure call suspends the execution of the caller, which is only resumed after the execution of the callee terminates.

**Theorem 13.1 (Parenthesis Theorem)** In a DFS-tree, for any two states \( q \) and \( r \), exactly one of the following four conditions holds, where \( I(q) \) denotes the interval \((d[q], f[q])\), and \( I(q) < I(r) \) denotes that \( f[q] < d[r] \) holds.

- \( I(q) \subseteq I(r) \) and \( q \) is a descendant of \( r \), or
- \( I(r) \subseteq I(q) \) and \( r \) is a descendant of \( q \), or
- \( I(q) < I(r) \), and neither \( q \) is a descendant of \( r \), nor \( r \) is a descendant of \( q \), or
- \( I(r) < I(q) \), and neither \( q \) is a descendant of \( r \), nor \( r \) is a descendant of \( q \).
Theorem 13.2 (White-path Theorem) In a DFS-tree, \( r \) is a descendant of \( q \) (and so \( I(r) \subseteq I(q) \)) if and only if at time \( d[q] \) state \( r \) can be reached from \( q \) in \( A \) along a path of white states.

13.1.1 The nested-DFS algorithm

To determine if \( A \) is empty we can search for the accepting states of \( A \), and check if at least one of them belongs to a cycle. A naïve implementation proceeds in two phases, searching for accepting states in the first, and for cycles in the second. The runtime is quadratic: since an automaton with \( n \) states and \( m \) transitions has \( \Theta(n) \) accepting states, and since searching for a cycle containing a given state takes \( O(n + m) \) time, we obtain a \( O(n^2 + nm) \) bound.

The nested-DFS algorithm runs in time \( O(n + m) \) by using the first phase not only to discover the reachable accepting states, but also to sort them. The searches of the second phase are conducted according to the order determined by the sorting. As we shall see, conducting the search in this order avoids repeated visits to the same state.

The first phase is carried out by a DFS, and the accepting states are sorted by increasing finishing (not discovery!) time. This is known as the postorder induced by the DFS. Assume that in the second phase we have already performed a search starting from the state \( q \) that has failed, i.e., no cycle of \( A \) contains \( q \). Suppose we proceed with a search from another state \( r \) (which implies \( f[q] < f[r] \)), and this search discovers some state \( s \) that had already been discovered by the search starting at \( q \). We claim that it is not necessary to explore the successors of \( s \) again. More precisely, we claim that \( s \not\rightarrow r \), and so it is useless to explore the successors of \( s \), because the exploration cannot return any cycle containing \( r \). The proof of the claim is based on the following lemma:

Lemma 13.3 If \( q \not\rightarrow r \) and \( f[q] < f[r] \) in some DFS-tree, then some cycle of \( A \) contains \( q \).

Proof: Let \( \pi \) be a path leading from \( q \) to \( r \), and let \( s \) be the first node of \( \pi \) that is discovered by the DFS. By definition we have \( d[s] \leq d[q] \). We prove that \( s \neq q \), \( q \not\rightarrow s \) and \( s \not\rightarrow q \) hold, which implies that some cycle of \( A \) contains \( q \).

- \( q \neq s \). If \( s = q \), then at time \( d[q] \) the path \( \pi \) is white, and so \( I(r) \subseteq I(q) \), contradicting \( f[q] < f[r] \).

- \( q \not\rightarrow s \). Obvious, because \( s \) belongs to \( \pi \).

- \( s \not\rightarrow q \). By the definition of \( s \), and since \( s \neq q \), we have \( d[s] \leq d[q] \). So either \( I(q) \subseteq I(s) \) or \( I(s) < I(q) \). We claim that \( I(s) < I(q) \) is not possible. Since at time \( d[s] \) the subpath of \( \pi \) leading from \( s \) to \( r \) is white, we have \( I(r) \subseteq I(s) \). But \( I(r) \subseteq I(s) \) and \( I(s) < I(q) \) contradict \( f[q] < f[r] \), which proves the claim. Since \( I(s) < I(q) \) is not possible, we have \( I(q) \subseteq I(s) \), and hence \( q \) is a descendant of \( s \), which implies \( s \not\rightarrow q \).

\( \square \)
Example 13.4 The NBA of Figure 13.1 contains a path from $q_1$ to $q_0$, and the DFS-tree displayed satisfied $f[q_1] = 11 < 12 = f[q_0]$. As guaranteed by lemma 13.3, some cycle contains $q_1$, namely the cycle $q_1q_6q_0$.  

To prove the claim, we assume that $s \rightsquigarrow r$ holds and derive a contradiction. Since $s$ was already discovered by the search starting at $q$, we have $q \rightsquigarrow s$, and so $q \rightsquigarrow r$. Since $f[q] < f[r]$, by Lemma 13.3 some cycle of $A$ contains $q$, contradicting the assumption that the search from $q$ failed.

Hence, during the second phase we only need to explore a transition at most once, namely when its source state is discovered for the first time. This guarantees the correctness of the following algorithm:

- Perform a DFS on $A$ from $q_0$, and output the accepting states of $A$ in postorder\(^2\). Let $q_1, \ldots, q_k$ be the output of the search, i.e., $f[q_1] < \cdots < f[q_k]$.

- For $i = 1$ to $k$, perform a DFS from the state $q_i$, with the following changes:
  - If the search visits a state $q$ that was already discovered by any of the searches starting at $q_1, \ldots, q_{i-1}$, then the search backtracks.
  - If the search visits $q_i$, it stops and returns NONEMPTY.

- If none of the searches from $q_1, \ldots, q_k$ returns NONEMPTY, return EMPTY.

Example 13.5 We apply the algorithm to the example of Figure 13.1. Assume that the first DFS runs as in Figure 13.1. The search outputs the accepting states in postorder, i.e., in the order $q_2, q_1, q_0$. Figure 13.2 shows the transitions explored during the searches of the second phase. The search from $q_2$ explores the transitions labelled by 2.1, 2.2, 2.3. The search from $q_1$ explores the transitions 1.1, \ldots, 1.5. Notice that the search backtracks after exploring 1.1, because the state $q_2$ was already visited by the previous search. This search is successful, because transition 1.5 reaches state $q_1$, and so a cycle containing $q_1$ has been found.

The running time of the algorithm can be easily determined. The first DFS requires $O(|Q| + |\delta|)$ time. During the searches of the second phase each transition is explored at most once, and so they can be executed together in $O(|Q| + |\delta|)$ time.

Nesting the two searches

Recall that we are looking for algorithms that return an accepting lasso when $A$ is nonempty. The algorithm we have described is not good for this purpose. Define the DFS-path of a state as the unique path of the DFS-tree leading from the initial state to it. When the second phase answers NONEMPTY, the DFS-path of the state being currently explored, say $q$, is an accepting cycle, but

\(^2\)Notice that this does not require to apply any sorting algorithm, it suffices to output an accepting state immediately after blackening it.
Figure 13.2: The transitions explored during the search starting at $q_i$ are labelled by the index $i$. The search starting at $q_1$ stops with NONEMPTY.

usually not an accepting lasso. For an accepting lasso we can prefix this path with the DFS-path of $q$ obtained during the first phase. However, since the first phase cannot foresee the future, it does not know which accepting state, if any, will be identified by the second phase as belonging to an accepting lasso. So either the first search must store the DFS-paths of all the accepting states it discovers, or a third phase is necessary, in which a new DFS-path is recomputed.

This problem can be solved by nesting the first and the second phases: Whenever the first DFS blackens an accepting state $q$, we immediately launch a second DFS to check if $q$ is reachable from itself. We obtain the nested-DFS algorithm, due to Courcoubetis, Vardi, Wolper, and Yannakakis:

- Perform a DFS from $q_0$.

- Whenever the search blackens an accepting state $q$, launch a new DFS from $q$. If this second DFS visits $q$ again (i.e., if it explores some transition leading to $q$), stop with NONEMPTY. Otherwise, when the second DFS terminates, continue with the first DFS.

- If the first DFS terminates, output EMPTY.

A pseudocode implementation is shown below; for clarity, the program on the left does not include the instructions for returning an accepting lasso. A variable seed is used to store the state from which the second DFS is launched. The instruction report $X$ produces the output $X$ and stops the execution. The set $S$ is usually implemented by means of a hash-table. Notice that it is not necessary to store states $[q, 1]$ and $[q, 2]$ separately. Instead, when a state $q$ is discovered, either during the first or the second searches, then it is stored at the hash address, and two extra bits are used to store which of the following three possibilities hold: only $[q, 1]$ has been discovered so far, only $[q, 2]$, or both. So, if a state is encoded by a bitstring of length $c$, then the algorithm needs $c + 2$ bits of memory per state.
NestedDFS(A)

Input: NBA A = (Q, Σ, δ, q₀, F)
Output: EMP if L_ω(A) = ∅
         NEMP otherwise
1   S ← ∅
2   dfs1(q₀)
3   report EMP
4   proc dfs1(q)
5      add [q, 1] to S
6      for all r ∈ δ(q) do
7      if [r, 1] ∉ S then dfs1(r)
8      if q ∈ F then { seed ← q; dfs2(q) }
9      return
10  proc dfs2(q)
11    add [q, 2] to S
12    for all r ∈ δ(q) do
13    if r = seed then report NEMP
14    if [r, 2] ∉ S then dfs2(r)
15    return

NestedDFSwithWitness(A)

Input: NBA A = (Q, Σ, δ, q₀, F)
Output: EMP if L_ω(A) = ∅
         NEMP otherwise
1   S ← ∅; succ ← false
2   dfs1(q₀)
3   report EMP
4   proc dfs1(q)
5      add [q, 1] to S
6      for all r ∈ δ(q) do
7      if [r, 1] ∉ S then dfs1(r)
8      if succ = true then return [q, 1]
9      if q ∈ F then
10     seed ← q; dfs2(q)
11     if succ = true then return [q, 1]
12    return
13  proc dfs2(q)
14    add [q, 2] to S
15    for all r ∈ δ(q) do
16    if [r, 2] ∉ S then dfs2(r)
17    if succ = true then return [q, 2]
18    if r = seed then
19    report NEMP; succ ← true
20   return

The algorithm on the right shows how to modify NestedDFS so that it returns an accepting lasso. It uses a global boolean variable succ (for success), initially set to false. If at line 11 the algorithm finds that r = seed holds, it sets success to true. This causes procedure calls in dfs1(q) and dfs2(q) to be replaced by return[q, 1] and return[q, 2], respectively. The lasso is produced in reverse order, i.e., with the initial state at the end.

A small improvement

We show that dfs2(q) can already return NONEMPTY if it discovers a state that belongs to the DFS-path of q in dfs1. Let qₖ be an accepting state. Assume that dfs1(q₀) discovers qₖ, and that the
DFS-path of $q_k$ in $dfs1$ is $q_0q_1 \ldots q_{k-1}q_k$. Assume further that $dfs2(q_k)$ discovers $q_i$ for some $0 \leq i \leq k-1$, and that the DFS-path of $q_i$ in $dfs2$ is $q_k q_{k+1} \ldots q_k+1q_i$. Then the path $q_0q_1 \ldots q_{k-1}q_k \ldots q_k+1q_i$ is a lasso, and, since $q_k$ is accepting, it is an accepting lasso. So stopping with NONEMPTY is correct. Implementing this modification requires to keep track during $dfs1$ of the states that belong to the DFS-path of the state being currently explored. Notice, however, that we do not need information about their order. So we can use a set $P$ to store the states of the path, and implement $P$ as e.g. a hash table. We do not need the variable $seed$ anymore, because the case $r = seed$ is subsumed by the more general $r \in P$.

**ImprovedNestedDFS(A)**

**Input:** NBA $A = (Q, \Sigma, \delta, q_0, F)$

**Output:** EMP if $L_{\omega}(A) = \emptyset$, NEMP otherwise

1. $S \leftarrow \emptyset$; $P \leftarrow \emptyset$
2. $dfs1(q_0)$
3. report EMP
4. proc $dfs1(q)$
   5. add $[q, 1]$ to $S$; add $q$ to $P$
   6. for all $r \in \delta(q)$ do
      7. if $[r, 1] \not< S$ then $dfs1(r)$
   8. if $q \in F$ then $dfs2(q)$
   9. remove $q$ from $P$
10. return

11. proc $dfs2(q)$
12. add $[q, 2]$ to $S$
13. for all $r \in \delta(q)$ do
   14. if $r \in P$ then report NEMP
      15. if $[r, 2] \not< S$ then $dfs2(r)$
16. return

**Evaluation**

The strong point of the nested-DFS algorithm are its very modest space requirements. Apart from the space needed to store the stack of calls to the recursive $dfs$ procedure, the algorithm just needs two extra bits for each state of $A$. In many practical applications, $A$ can easily have millions or tens of millions of states, and each state may require many bytes of storage. In these cases, the two extra bits per state are negligible.

The algorithm, however, also has two important weak points: It cannot be extended to NGAs, and it is not optimal, in a formal sense defined below. We discuss these two points separately.
The nested-DFS algorithm works by identifying the accepting states first, and then checking if they belong to some cycle. This principle no longer works for the acceptance condition of NGAs, where we look for cycles containing at least one state of each family of accepting states. No better procedure than translating the NGA into an NBA has been described so far. For NGAs having a large number of accepting families, the translation may involve a substantial penalty in performance.

A search-based algorithm for emptiness checking explores the automaton \( A \) starting from the initial state. At each point \( t \) in time, the algorithm has explored a subset of the states and the transitions of the algorithm, which form a sub-NBA \( A_t = (Q_t, \Sigma, \delta_t, q_0, F_t) \) of \( A \) (i.e., \( Q_t \subseteq Q, \delta_t \subseteq \delta, \) and \( F_t \subseteq F \)). Clearly, a search-based algorithm can have only reported NONEMPTY at a time \( t \) if \( A_t \) contains an accepting lasso. A search-based algorithm is optimal if the converse holds, i.e., if it reports NONEMPTY at the earliest time \( t \) such that \( A_t \) contains an accepting lasso. It is easy to see that NestedDFS is not optimal. Consider the automaton on top of Figure 13.5. Initially,

![Figure 13.3: Two bad examples for NestedDFS](image)

the algorithm chooses between the transitions \((q_0, q_1)\) and \((q_0, q_2)\). Assume it chooses \((q_0, q_1)\) (the algorithm does not know that there is a long tail behind \( q_2 \)). The algorithm explores \((q_0, q_1)\) and then \((q_1, q_0)\) at some time \( t \). The automaton \( A_t \) already contains an accepting lasso, but, since \( q_0 \) has not been blackened yet, \( dfs1 \) continues its execution with \((q_0, q_2)\), and explores all transitions before \( dfs2 \) is called for the first time, and reports NONEMPTY. So the time elapsed between the first moment at which the algorithm has enough information to report NONEMPTY, and the moment at which the report occurs, can be arbitrarily large.

The automaton at the bottom of Figure 13.5 shows another problem of NestedDFS related to non-optimality. If it selects \((q_0, q_1)\) as first transition, then, since \( q_n \) precedes \( q_0 \) in postorder, \( dfs2(q_n) \) is executed before \( dfs2(q_0) \), and it succeeds, reporting the lasso \( q_0q_2\ldots q_nq_{n+1}q_n \), instead of the much shorter lasso \( q_0q_1q_0 \).
In the next section we describe an optimal algorithm that can be easily extended to NGAs. The price to pay is a higher memory consumption. As we shall see, the new algorithm needs to assign a number to each state, and store it (apart from maintaining other data structures).

### 13.1.2 The two-stack algorithm

Recall that the nested-DFS algorithm searches for accepting states of $A$, and then checks if they belong to some cycle. The two-stack algorithm proceeds the other way round: It searches for states that belong to some cycle of $A$ by means of a single DFS, and checks whether they are accepting.

A first observation is that by the time the DFS blackens a state, it already has explored enough to decide whether it belongs to a cycle:

**Lemma 13.6** Let $A_t$ be the sub-NBA of $A$ containing the states and transitions explored by the DFS up to (and including) time $t$. If a state $q$ belongs to some cycle of $A$, then it already belongs to some cycle of $A_{f[q]}$.

**Proof:** Let $\pi$ be a cycle containing $q$, and consider the snapshot of the DFS at time $f[q]$. Let $r$ be the last state of $\pi$ after $q$ such that all states in the subpath from $q$ to $r$ are black. We have $f[r] \leq f[q]$. If $r = q$, then $\pi$ is a cycle of $A_{f[q]}$, and we are done. If $r \neq q$, let $s$ be the successor of $r$ in $\pi$ (see Figure 13.4). We have $f[r] < f[q] < f[s]$. Moreover, since all successors of $r$ have necessarily been discovered at time $f[r]$, we have $d[s] < f[r] < f[q] < f[s]$. By the Parenthesis theorem, $s$ is a DFS-ascendant of $q$. Let $\pi'$ be the cycle obtained by concatenating the DFS-path from $s$ to $q$, the prefix of $\pi$ from $q$ to $r$, and the transition $(r, s)$. By the Parenthesis Theorem, all the transitions in this path have been explored at time $f[q]$, and so the cycle belongs to $A_{f[q]}$.

![Figure 13.4: Illustration of the proof of Lemma 13.6](image)

This lemma suggests to maintain during the DFS a set $C$ of candidates, containing the states for which it is not yet known whether they belong to some cycle or not. A state is added to $C$ when it is discovered. While the state is grey, the algorithm tries to find a cycle containing it. If it succeeds, then the state is removed from $C$. If not, then the state is removed from $C$ when it is blackened. At any time $t$, the candidates are the currently grey states that do not belong to any cycle of $A_t$. 
Assume that at time $t$ the set $C$ indeed contains the current set of candidates, and that the DFS explores a new transition $(q, r)$. We need to update $C$. If $r$ has not been discovered yet (i.e., if it does not belong to $A_t$), the addition of $r$ and $(q, r)$ to $A_t$ does not create any new cycle, and the update just adds $r$ to $C$. If $r$ belongs to $A_t$ but no path of $A_t$ leads from $r$ to $q$, again no new cycle is created, and the set $C$ does not change. But if $r$ belongs to $A_t$, and $r \sim q$ then the addition of $(q, r)$ does create new cycles. Let us assume we can ask an oracle whether $r \sim q$ holds, and the oracle answers ‘yes’. Then we have already learnt that both $q$ and $r$ belong to some cycle of $A_t$, and so both of them must be removed from $C$. However, we may have to remove other states as well. Consider the DFS of Figure 13.5: after adding $(q_4, q_1)$ to the set of explored transitions at time 5, all of $q_1, q_2, q_3, q_4$ belong to a cycle. The fact that these are the states discovered by the DFS between the discoveries of $q_1$ and $q_4$ suggests to implement $C$ using a stack: By pushing states into $C$ when they are discovered, and removing when they are blackened or earlier (if some cycle contains them), the update to $C$ after exploring a transition $(q, r)$ can be performed very easily: it suffices to pop from $C$ until $r$ is hit (in Figure 13.5 we pop $q_4$, $q_3$, $q_2$, and $q_1$). Observe also that removing $q$ from $C$ when it is blackened does not require to inspect the complete stack; since every state is removed at the latest when it is blackened, if $q$ has not been removed yet, then it is necessarily at the top of $C$. (This is the case of state $q_0$ in Figure 13.5). So it suffices to inspect the top of the stack: if $q$ is at the top, we pop it; otherwise $q$ is not in the stack, and we do nothing.

This leads to our first attempt at an algorithm, shown on the top left corner of Figure 13.6. When a state $q$ is discovered it is pushed into $C$ (line 5), and its successors explored (lines 6-12). When exploring a successor $r$, if $r$ has not been discovered yet then $dfs(r)$ is called (line 7). Otherwise the oracle is consulted (line 8), and if $r \sim q$ holds at the time (i.e., in the part of $A_t$ explored so far), then states are popped from $C$ until $r$ is hit (lines 9-12). Then the algorithm checks if $q$ has already been removed by inspecting the top of the stack (line 13), and removes $q$ if that is the case.

The NBA below FirstAttempt shows, however, that the algorithm needs to be patched. After exploring $(q_4, q_1)$ the states $q_4, q_3, q_2, q_1$ are popped from $C$, in that order, and $C$ contains only $q_0$. Now the DFS backtracks to state $q_3$, and explores $(q_3, q_5)$, pushing $q_5$ into the stack. Then the DFS explores $(q_5, q_1)$, and pops from $C$ until it hits $q_1$. But this leads to an incorrect result, because, since $q_1$ no longer belongs to $C$, the algorithm pops all states from $C$, and when it pops $q_0$ it reports
**First Attempt**

**Input:** NBA $A = (Q, \Sigma, \delta, q_0, F)$

**Output:** EMP if $L_\omega(A) = \emptyset$

NEMP otherwise

1. $S, C \leftarrow \emptyset$;
2. dfs($q_0$)
3. report EMP

**proc dfs($q$)**
4. add $q$ to $S$; push($q, C$)
5. for all $r \in \delta(q)$ do
6. if $r \notin S$ then dfs($r$)
7. else if $r \leadsto q$ then
8. repeat
9. $s \leftarrow$ pop($C$)
10. if $s \in F$ then report NEMP
11. until $s = r$
12. if top($C$) = $q$ then pop($C$)

**Second Attempt**

**Input:** NBA $A = (Q, \Sigma, \delta, q_0, F)$

**Output:** EMP if $L_\omega(A) = \emptyset$

NEMP otherwise

1. $S, C \leftarrow \emptyset$;
2. dfs1($q_0$)
3. report EMP

**proc dfs($q$)**
4. add $q$ to $S$; push($q, C$)
5. for all $r \in \delta(q)$ do
6. if $r \notin S$ then dfs($r$)
7. else if $r \leadsto q$ then
8. repeat
9. $s \leftarrow$ pop($C$)
10. if $s \in F$ then report NEMP
11. until $s = r$
12. if top($C$) = $q$ then pop($C$)

![Figure 13.6: Two incorrect attempts at an emptiness checking algorithm](image-url)
13.1. ALGORITHMS BASED ON DEPTH-FIRST SEARCH

NONEMPTY.

This problem can be solved as follows: If the DFS explores \((q, r)\) and \(r \sim q\) holds, then it pops from \(C\) until \(r\) is popped, and pushes \(r\) back into the stack. This second attempt is shown at the top right of the figure. However, the NBA below SecondAttempt shows that it is again incorrect. After exploring \((q_4, q_1)\) (with stack content \(q_4q_3q_2q_1q_0\)), the states \(q_4, q_3, q_2, q_1\) are popped from \(C\), in that order, and \(q_1\) is pushed again. \(C\) contains now \(q_1q_0\). The DFS explores \((q_3, q_5)\) next, pushing \(q_5\), followed by \((q_5, q_4)\). Since \(q_4 \sim q_5\) holds, the algorithm pops from \(C\) until \(q_4\) is found. But, again, since \(q_4\) does not belong to \(C\), the result is incorrect.

A patch for this problem is to change the condition of the repeat loop: If the DFS explores \((q, r)\) and \(r \sim q\) holds, we cannot be sure that \(r\) is still in the stack. So we pop until either \(r\) or some state discovered before \(r\) is hit, and then we push this state back again. In the example, after exploring \((q_5, q_4)\) with stack content \(q_5q_1q_0\), the algorithm pops \(q_5\) and \(q_1\), and then pushes \(q_1\) back again. This new patch leads to the OneStack algorithm:

\[
\text{OneStack}(A)
\]

**Input:** NBA \(A = (Q, \Sigma, \delta, q_0, F)\)

**Output:** EMP if \(L_\omega(A) = \emptyset\), NEMP otherwise

1. \(S, C \leftarrow \emptyset\);
2. \(\text{dfs}(q_0)\)
3. \(\text{report EMP}\)
4. \(\text{dfs}(q)\)
5. \(\text{add } q \text{ to } S; \text{ push}(q, C)\)
6. \(\text{for all } r \in \delta(q) \text{ do}\)
7. \(\text{if } r \not\in S \text{ then } \text{dfs}(r)\)
8. \(\text{else if } r \sim q \text{ then}\)
9. \(\text{repeat}\)
10. \(s \leftarrow \text{pop}(C); \text{ if } s \in F \text{ then report NEMP}\)
11. \(\text{until } d[s] \leq d[r]\)
12. \(\text{push}(s, C)\)
13. \(\text{if top}(C) = q \text{ then pop}(C)\)

**Example 13.7** Figure 13.7 shows a run of OneStack on the NBA shown at the top. The NBA has no accepting states, and so it is empty. However, during the run we will see how the algorithm answers NONEMPTY (resp. EMPTY) when \(f\) (res. \(h\)) is the only accepting state. The discovery and finishing times are shown at the top. Observe that the NBA has three sccs: \(\{a, b, e, f, g\}\), \(\{h\}\), and \(\{c, d, i, j\}\), with roots \(a, h, \text{ and } i\), respectively.

Below the NBA at the top, the figure shows different snapshots of the run of OneStack. At each snapshot, the current grey path is shown in red/pink. The dotted states and transitions have not been discovered yet, while dark red states have already been blackened. The current content of stack \(C\) is shown on the right (ignore stack \(L\) for the moment).
• The first snapshot is taken immediately before state \( j \) is blackened. The algorithm has just explored the transition \((j, i)\), has popped the states \( c, d, j, i \) from \( C \), and has pushed state \( i \) back. The states \( c, d, j, i \) have been identified as belonging to some cycle.

• The second snapshot is taken immediately after state \( i \) is blackened. State \( i \) has been popped from \( C \) at line 13. Observe that after this the algorithm backtracks to \( dfs(h) \), and, since state \( h \) is at the top of the stack, it pops \( h \) from \( C \) at line 13. So \( h \) is never popped by the repeat loop, and so even if \( h \) is accepting the algorithm does not report NONEMPTY.

• The third snapshot is taken immediately before state \( f \) is blackened. The algorithm has just explored the transition \((f, a)\), has popped the states \( f, g, b, a \) from \( C \), and has pushed state \( a \) back. The states \( f, g, b, a \) have been identified as belonging to some cycle. If state \( f \) is accepting, at this point the algorithm reports EMPTY and stops.

• The fourth snapshot is taken immediately after state \( e \) is discovered. The state has been pushed into \( C \).

• The final snapshot is taken immediately before \( a \) is blackened and the run terminates. State \( a \) is going to be removed from \( C \) at line 13, and the run terminates.

Observe that when the algorithm explores transition \((b, c)\) it calls the oracle, which answers \( c \not\rightarrow b \), and no states are popped from \( C \).

The algorithm looks now plausible, but we still must prove it correct. We have two proof obligations:

• If \( A \) is nonempty, then OneStack reports NONEMPTY. This is equivalent to: every state that belongs to some cycle is eventually popped during the repeat loop.

• If OneStack reports NONEMPTY, then \( A \) is nonempty. This is equivalent to: every state popped during the repeat loop belongs to some cycle.

These properties are shown in Propositions 13.8 and 13.11 below.

**Proposition 13.8** If \( q \) belongs to a cycle, then \( q \) is eventually popped by the repeat loop.

**Proof:** Let \( \pi \) be a cycle containing \( q \), let \( q' \) be the last successor of \( q \) along \( \pi \) such that at time \( d[q] \) there is a white path from \( q \) to \( q' \), and let \( r \) be the successor of \( q' \) in \( \pi \). Since \( r \) is grey or black at time \( d[q] \), we have \( d[r] \leq d[q] \leq d[q'] \). By the White-path Theorem, \( q' \) is a descendant of \( q \), and so the transition \((q', r)\) is explored before \( q \) is blackened. So when \((q', r)\) is explored, \( q \) has not been popped at line 13. Since \( r \sim q' \), either \( q \) has already been popped by at some former execution of the repeat loop, or it is popped now, because \( d[r] \leq d[q'] \).
13.1. ALGORITHMS BASED ON DEPTH-FIRST SEARCH

Figure 13.7: A run of OneStack
Actually, the proof of Proposition 13.8 proves not only that $q$ is eventually popped by the repeat loop, but also that for every cycle $\pi$ containing $q$, the repeat loop pops $q$ immediately after all transitions of $\pi$ have been explored, or earlier. But this is precisely the optimality property, which leads to:

**Corollary 13.9** *OneStack is optimal.*

The property that every state popped during the repeat loop belongs to some cycle has a more involved proof. A strongly connected component (scc) of $A$ is a maximal set of states $S \subseteq Q$ such that $q \leadsto r$ for every $q, r \in S$.\(^3\) The first state of a reachable scc that is discovered by the DFS is called the root of the scc (with respect to this DFS). In other words, if $q$ is the root of an scc, then $d[q] \leq d[r]$ for every state $r$ of the scc. The following lemma states an important invariant of the algorithm. If a root belongs to $C$ at line 9, before the repeat loop is executed, then it still belongs to $C$ after the loop finishes and the last popped state is pushed back. So, loosely speaking, the repeat loop cannot remove a root from the stack; more precisely, if the loop removes a root, then the push instruction at line 12 reintroduces it again.

**Lemma 13.10** Let $\rho$ be a root. If $\rho$ belongs to $C$ before an execution of the repeat loop at lines 9-11, then all states $s$ popped during the execution of the loop satisfy $d[\rho] \leq d[s]$, and $\rho$ still belongs to $C$ after the repeat loop has terminated and line 12 has been executed.

**Proof:** Let $t$ be the time right before an execution of the repeat loop starts at line 9, and assume $\rho$ belongs to $C$ at time $t$. Since states are removed from $C$ when they are blackened or earlier, $\rho$ is still grey at time $t$. Since $r \in \delta(q)$ (line 6) and $r \leadsto q$ (line 8), both $q$ and $r$ belong to the same scc. Let $\rho'$ be the root of this scc. Since $\rho'$ is also grey at time $t$, and grey states always are a path of the DFS-tree, either $\rho$ is a DFS-ancestor of $\rho'$, or $\rho' \neq \rho$ and $\rho'$ is a DFS-ancestor of $\rho$. In the latter case we have $\rho' \leadsto \rho \leadsto q$, contradicting that $\rho'$ is the root of $q$'s scc. So $\rho$ is a DFS-ancestor of $\rho'$, and so in particular we have $d[\rho] \leq d[\rho'] \leq d[r]$. Since states are added to $C$ when they are discovered, the states of $C$ are always ordered by decreasing discovery time, starting from the top, and so every state $s$ popped before $\rho$ satisfies $d[\rho] \leq d[s]$. If $\rho$ is popped, then, since $d[\rho] \leq d[r]$, the execution of the repeat loop terminates, and $\rho$ is pushed again at line 12.

**Proposition 13.11** Any state popped during the repeat loop (at line 10) belongs to some cycle.

**Proof:** Consider the time $t$ right before a state $s$ is about to be popped at line 10 while the for-loop (lines 6-12) is exploring a transition $(q, r)$. (Notice that the body of the for-loop may have already been executed for other transitions $(q, r')$). Since the algorithm has reached line 10, we have $r \leadsto q$ (line 8), and so both $q$ and $r$ belong to the same scc of $A$. Let $\rho$ be the root of this scc. We show that $s$ belongs to a cycle.

---

\(^3\)Notice that a path consisting of just a state $q$ and no transitions is a path leading from $q$ to $q$. 
13.1. ALGORITHMS BASED ON DEPTH-FIRST SEARCH

(1) \( s \) is a DFS-ascendant of \( q \).
   Since \( s \) belongs to \( C \) at time \( t \), both \( s \) and \( q \) are grey at time \( t \), and so they belong to the current path of grey states. Moreover, since \( dfs(q) \) is being currently executed, \( q \) is the last state in the path. So \( s \) is a DFS-ascendant of \( q \).

(2) \( \rho \) is a DFS-ascendant of \( s \).
   Since \( \rho \) is a root of the scc of \( q \), at time \( d[\rho] \) there is a white path from \( \rho \) to \( q \). By the White-path and Parenthesis Theorem, \( \rho \) is a DFS-ascendant of \( q \). Together with (1), this implies that either \( \rho \) is a DFS-ascendant of \( s \) or \( s \) is a DFS-ascendant of \( \rho \). By Lemma 13.10 we have \( d[\rho] \leq d[s] \), and so by the Parenthesis Theorem \( \rho \) is a DFS-ascendant of \( s \).

By (1), (2), and \( r \{ q \), we have \( \rho \{ s \{ q \{ r \{ \rho \), and so \( s \) belongs to a cycle. \( \square \)

Implementing the oracle

Recall that \( OneStack \) calls an oracle to decide at time \( t \) if \( r \{ q \) holds. At first sight the oracle seems difficult to implement. We show that this is not the case.

Assume that \( OneStack \) calls the oracle at line 8 at some time \( t \). We look for a condition that holds at time \( t \) if and only if \( r \sim q \), and is easy to check.

Lemma 13.12 Assume that \( OneStack(A) \) is currently exploring a transition \( (q, r) \), and the state \( r \) has already been discovered. Let \( R \) be the scc of \( A \) satisfying \( r \in R \). Then \( r \sim q \) iff some state of \( R \) is not black.

Proof: Assume \( r \sim q \). Then \( r \) and \( q \) belong to \( R \), and since \( q \) is not black because \( (q, r) \) is being explored, \( R \) is not black.
   Assume every state of \( R \) is black. Not all states of \( R \) are white because \( r \) has already been discovered, and so at least one state \( s \in R \) is grey. Since grey states form a path ending at the state whose output transitions are being currently explored, the grey path contains \( s \) and ends at \( q \). So \( s \sim q \), and, since \( s \) and \( r \) belong to \( R \), we have \( r \sim q \). \( \square \)

By Lemma 13.12, checking if \( r \sim q \) holds amounts to checking if all states of \( R \) are black or not. This can be done as follows: we maintain a set \( V \) of active states, where a state is active if its scc has not yet been completely explored, i.e., if some state of the scc is not black. Then, checking \( r \sim q \) reduces to checking whether \( r \) is active. The set \( V \) can be maintained by adding a state to it whenever it is discovered, and removing all the states of a scc right after the last of them is blackened. The next lemma shows that the last of them is always the root:

Lemma 13.13 Let \( \rho \) be a root, and let \( q \) be a state such that \( \rho \sim q \sim \rho \). Then \( I(q) \subseteq I(r) \). (In words: The root is the first state of a scc to be grayed, and the last to be blackened)

Proof: By the definition of a root, at time \( d[\rho] \) there is a white path from \( \rho \) to \( q \). By the White-path and the Parenthesis Theorems, \( I(q) \subseteq I(r) \). \( \square \)
CHAPTER 13. EMPTINESS CHECK: IMPLEMENTATIONS

By this lemma, in order to maintain $V$ it suffices to remove all the states of an scc whenever its root is blackened. So whenever the DFS blackens a state $q$, we have to perform two tasks: (1) check if $q$ is a root, and (2) if $q$ is a root, remove all the states of $q$’s scc. Checking if $q$ is a root is surprisingly simple:

Lemma 13.14 When OneStack executes line 13, $q$ is a root if and only if $\text{top}(C) = q$.

Proof: Assume $q$ is a root. By Lemma 13.10, $q$ still belongs to $C$ after the for loop at lines 6-12 is executed, and so $\text{top}(C) = q$ at line 13.

Assume now that $q$ is not a root. Then there is a path from $q$ to the root of $q$’s scc. Let $r$ be the first state in the path satisfying $d[r] < d[q]$, and let $q'$ be the predecessor of $r$ in the path. By the White-path theorem, $q'$ is a descendant of $q$, and so when transition $(q, r)$ is explored, $q$ is not yet black. When OneStack explores $(q', r)$, it pops all states $s$ from $C$ satisfying $d[s] > d[r]$, and none of these states is pushed back at line 12. In particular, either OneStack has already removed $q$ from $C$, or it removes it now. Since $q$ has not been blackened yet, when OneStack executes line 14 for $dfs(q)$, the state $q$ does not belong to $C$ and in particular $q \neq \text{top}(C)$.

Tasks (2) can be performed very elegantly by implementing $V$ as a second stack, and maintaining it as follows:

- when a state is discovered (greyed), it is pushed into the stack (so states are always ordered in $V$ by increasing discovery time); and
- when a root is blackened, all states of $V$ above it (including the root itself) are popped.

Example 13.15 Figure 13.7 shows the content of $L$ at different times when the policy above is followed. Right before state $j$ is blackened, $L$ contains all states in the grey path. When the root $i$ is blackened, all states above it (including $i$ itself), are popped; these are the states $c, d, j, i$, which form a scc. State $h$ is also a root, and it is also popped. Then, states $f$ and $e$ are discovered, and pushed into $V$. Finally, when the root $a$ is blackened, states $c, f, g, b, a$ are popped, which correspond to the third and last scc.

We show that the states popped when $\rho$ is blackened are exactly those that belong to $\rho$’s scc.

Lemma 13.16 The states popped from $V$ right after blackening a root $\rho$ are exactly those belonging to $\rho$’s scc.

Proof: Let $q$ be a state of $\rho$’s scc. Since $\rho$ is a root, we have $d[\rho] \leq d[q]$, and so $q$ lies above $\rho$ in $L$. So $q$ is popped when $\rho$ is blackened, unless it has been popped before. We show that this cannot be the case. Assume $q$ is popped before $\rho$ is blackened, i.e., when some other root $\rho' \neq \rho$ is blackened at time $f[\rho']$. We collect some facts: (a) $d[\rho] \leq d[q] \leq d[q] \leq f[\rho]$ by Lemma 13.13; (b) $d[\rho'] \leq d[q] < f[\rho']$, because $q$ is in the stack at time $f[\rho']$ (implying $d[q] < f[\rho']$), and it is
above \( \rho' \) in the stack (implying \( d[\rho'] \leq d[q] \)); (c) \( f[\rho'] < f[\rho] \), because \( q \) has not been popped yet at time \( f[\rho'] \), and so \( \rho \) cannot have been blackened yet. From (a)-(c) and the Parenthesis Theorem we get \( I(q) \subseteq I(\rho') \subseteq I(\rho) \), and so in particular \( \rho \sim \rho' \sim q \). But then, since \( q \sim \rho \), we get \( \rho \sim \rho' \sim \rho \), contradicting that \( \rho \) and \( \rho' \) are different roots, and so belong to different sccs.

This finally leads to the two-stack algorithm

\[
\text{TwoStack}(A)
\]

**Input:** NBA \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** EMP if \( L_\omega(A) = \emptyset \), NEMP otherwise

1. \( S, C, V \leftarrow \emptyset; \)
2. \( \text{dfs}(q_0) \)
3. \( \text{report EMP} \)
4. \( \text{proc } \text{dfs}(q) \)
5. \( \text{add } q \text{ to } S; \text{ push}(q, C); \text{ push}(q, V) \)
6. \( \text{for all } r \in \delta(q) \) do
7. \( \text{if } r \notin S \text{ then } \text{dfs}(r) \)
8. \( \text{else if } r \in V \text{ then} \)
9. \( \text{repeat} \)
10. \( s \leftarrow \text{pop}(C); \text{ if } s \in F \text{ then } \text{report NEMP} \)
11. \( \text{until } d[s] \leq d[r] \)
12. \( \text{push}(s, C) \)
13. \( \text{if } \text{top}(C) = q \text{ then} \)
14. \( \text{pop}(C) \)
15. \( \text{repeat } s \leftarrow \text{pop}(V) \text{ until } s = q \)

The changes with respect to OneStack are shown in blue. The oracle \( r \rightarrow q \) is replaced by \( r \in V \) (line 8). When the algorithm blackens a root (line 13), it pops from \( V \) all elements above \( q \), and \( q \) itself (line 15).

Observe that \( V \) cannot be implemented only as a stack, because at line 8 we have to check if a state belongs to the stack or not. The solution is to implement \( V \) both as a stack and use an additional bit in the hash table for \( S \) to store whether the state belongs to \( V \) or not, which is possible because \( V \subseteq S \) holds at all times. The check at line 8 is performed by checking the value of the bit.

**Extension to GBAs**

We show that TwoStack can be easily transformed into an emptiness check for generalized Büchi automata that does not require to construct an equivalent NBA. Recall that a NGA has in general several sets \( \{F_0, \ldots, F_{k-1}\} \) of accepting states, and that a run \( \rho \) is accepting if \( \inf \rho \cap F_i \neq \emptyset \) for every \( i \in \{0, \ldots, k - 1\} \). So we have the following characterization of nonemptiness, where \( K = \{0, \ldots, k - 1\} \):
Fact 13.17 Let $A$ be a NGA with accepting condition $\{F_0, \ldots, F_{k-1}\}$. $A$ is nonempty iff some scc $S$ of $A$ satisfies $S \cap F_i \neq \emptyset$ for every $i \in K$.

Since every time the repeat loop at line 15 is executed, it pops from $L$ one scc, we can easily check this condition by modifying line 15 accordingly. However, the resulting algorithm would not be optimal, because the condition is not checked until the scc has been completely explored. To solve this problem, we have a closer look at Proposition 13.11. The proof shows that a state $s$ popped at line 10 belongs to the scc of state $r$. So, in particular, all the states popped during an execution of the repeat loop at lines 9-11 belong to the same scc. So we collect the set $I$ of indices of the sets of accepting states they belong to, and keep checking whether $I = K$. If so, then we report NONEMPTY. Otherwise, we attach the set $I$ to the state $s$ that is pushed back into the $C$ at line 12. This yields algorithm TwoStackNGA, where $F(q)$ denotes the set of all indices $i \in K$ such that $q \in F_i$:

TwoStackNGA($A$)

**Input:** NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{k-1}\})$

**Output:** EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

1. $S, C, V \leftarrow \emptyset$
2. $dfs(q_0)$
3. report EMP
4. proc dfs($q$)
5. add $[q, F(q)]$ to $S$; push($[q, F(q)], C$); push($q, V$)
6. for all $r \in \delta(q)$ do
7. if $r \notin S$ then dfs($r$)
8. else if $r \in V$ then
9. $I \leftarrow \emptyset$
10. repeat
11. $[s, J] \leftarrow \text{pop}(C)$;
12. $I \leftarrow I \cup J$; if $I = K$ then report NEMP
13. until $d[s] \leq d[r]$
14. push($[s, I], C$)
15. if top($C$) = ($q, I$) for some $I$ then
16. pop($C$)
17. repeat $s \leftarrow \text{pop}(V)$ until $s = q$

For the correctness of the algorithm, observe that, at every time $t$, the states of the subautomaton $A_t$ can be partitioned into strongly connected components, and each of these components has a root. The key invariant for the correctness proof is the following:
13.1. ALGORITHMS BASED ON DEPTH-FIRST SEARCH

**Lemma 13.18** At every time \( t \), the stack \( C \) contains a pair \( [q, I] \) iff \( q \) is a root of \( A_t \), and \( I \) is the subset of indices \( i \in K \) such that some state of \( F_i \) belongs to \( q \)'s scc.

**Proof:** Initially the invariant holds because both \( A_t \) and \( C \) are empty. We show that whenever a new transition \( (q, r) \) is explored, TwoStackNGA carries out the necessary changes to keep the invariant. Let \( t \) be the time immediately after \( (q, r) \) is explored. If \( r \) is a new state, then it has no successors in \( A_t \), and so it builds an scc of \( A_t \) by itself, with root \( r \). Moreover, all roots before the exploration of \( (q, r) \) are also roots of \( A_t \). So a new pair \( [r, F(r)] \) must be added to \( C \), and that is what dfs does. If \( r \not\in L \), then \( r \not\rightarrow \rho \), the addition of \( (q, r) \) has not changed the sccs of the automaton explored so far, and nothing must be done. If \( r \in L \), then the addition of \( (q, r) \) creates new cycles, and some states stop being roots. More precisely, let \( NR \) be the set of states of \( A_t \) that belong to a cycle containing both \( q \) and \( r \), and let \( \rho \) be the state of \( NR \) with minimal discovery time. Then the algorithm must remove from \( C \) all states of \( NR \) except for \( \rho \)(and no others). We show that this is exactly what the execution of lines 9-14 achieves. By Proposition 13.8 and Corollary 13.9, all the states of \( NR \setminus \{\rho\} \) have already been removed at some former execution of the loop, or are removed now at lines 9-14, because they have discovery time smaller than or equal to \( d[r] \). It remains to show that all states popped at line 11 belong to \( NR \) (that \( \rho \) is not removed follows then from the fact that the state with the lowest discovery time is pushed again at line 14, and that state is \( \rho \)). For this, we have a closer look at the proof of Proposition 13.11. The proposition shows not only that the states popped by the repeat loop belong to some cycle, but also that they all belong to cycles that containing \( q \) and \( r \) (see the last line of the proof), and we are done.

We can now easily prove:

**Proposition 13.19** TwoStackNGA(\( A \)) reports NONEMPTY iff \( A \) is nonempty. Moreover, TwoStack-NGA is optimal.

**Proof:** If TwoStackNGA(\( A \)) reports NONEMPTY, then the repeat loop at lines 10-13 pops some pair \( [q, K] \). By Lemma 13.18, \( q \) belongs to a cycle of \( A \) containing some state of \( F_i \) for every \( i \in K \).

If \( A \) is nonempty, then some scc \( S \) of \( A \) satisfies \( S \cap F_i \neq \emptyset \) for every \( i \in K \). So there is an earliest time \( t \) such that \( A_t \) contains an scc \( S_t \subseteq S \) satisfying the same property. By Lemma 13.18, TwoStackNGA(\( A \)) reports NONEMPTY at time \( t \) or earlier, and so it is optimal.

**Evaluation**

Recall that the two weak points of the nested-DFS algorithm were that it cannot be directly extended to NGAs, and it is not optimal. Both are strong points of the two-stack algorithm.

The strong point of the the nested-DFS algorithm were its very modest space requirements: just two extra bits for each state of \( A \). Let us examine the space needed by the two-stack algorithm. It is convenient to compute it for empty automata, because in this case both the nested-DFS and the two-stack algorithms must visit all states.
Because of the check $d[s] \leq d[r]$, the algorithm needs to store the discovery time of each state. This is done by extending the hash table $S$. If a state $q$ can be stored using $c$ bits, then $\log n$ bits are needed to store $d[q]$; however, in practice $d[q]$ is stored using a word of memory, because if the number states of $A$ exceeds $2^w$, where $w$ is the number of bits of a word, then $A$ cannot be stored in main memory anyway. So the hash table $S$ requires $c + w + 1$ bits per state (the extra bit being the one used to check membership in $L$ at line 8).

The stacks $C$ and $L$ do not store the states themselves, but the memory addresses at which they are stored. Ignoring hashing collisions, this requires $2w$ additional bits per state. For generalized Büchi automata, we must also add the $k$ bits needed to store the subset of $K$ in the second component of the elements of $C$. So the two-stack algorithm uses a total of $c + 3w + 1$ bits per state ($c + 3w + k + 1$ in the version for NGA), compared to the $c + 2$ bits required by the nested-DFS algorithm. In most cases $w << c$, and so the influence of the additional memory requirements on the performance is small.

### 13.2 Algorithms based on breadth-first search

In this section we describe algorithms based on breadth-first search (BFS). No linear BFS-based emptiness check is known, and so this section may look at first sight superfluous. However, BFS-based algorithms can be suitably described using operations and checks on sets, which allows us to implement them using automata as data structures. In many cases, the gain obtained by the use of the data structure more than compensates for the quadratic worse-case behaviour, making the algorithms competitive.

Breadth-first search (BFS) maintains the set of states that have been discovered but not yet explored, often called the frontier or boundary. A BFS from a set $Q_0$ of states (in this section we consider searches from an arbitrary set of states of $A$) initializes both the set of discovered states and its frontier to $Q_0$, and then proceeds in rounds. In a forward search, a round explores the outgoing transitions of the states in the current frontier; the new states found during the round are added to the set of discovered states, and they become the next frontier. A backward BFS proceeds similarly, but explores the incoming instead of the outgoing transitions. The pseudocode implementations of both BFS variants shown below use two variables $S$ and $B$ to store the set of discovered states and the boundary, respectively. We assume the existence of oracles that, given the current boundary $B$, return either $\delta(B) = \bigcup_{q \in B} \delta(q)$ or $\delta^{-1}(B) = \bigcup_{q \in B} \delta^{-1}(q)$.

\[
\begin{array}{ll}
\text{ForwardBFS}[A](Q_0) & \text{Input:} \quad \text{NBA} \ A = (Q, \Sigma, \delta, q_0, F), \\
Q_0 \subseteq Q & Q_0 \subseteq Q \\
\begin{array}{ll}
1 & S, B \leftarrow Q_0; \\
2 & \textbf{repeat} \\
3 & B \leftarrow \delta(B) \setminus S \\
4 & S \leftarrow S \cup B \\
5 & \textbf{until} \ B = \emptyset
\end{array}
\end{array}
\]

\[
\begin{array}{ll}
\text{BackwardBFS}[A](Q_0) & \text{Input:} \quad \text{NBA} \ A = (Q, \Sigma, \delta, q_0, F), \\
Q_0 \subseteq Q & Q_0 \subseteq Q \\
\begin{array}{ll}
1 & S, B \leftarrow Q_0; \\
2 & \textbf{repeat} \\
3 & B \leftarrow \delta^{-1}(B) \setminus S \\
4 & S \leftarrow S \cup B \\
5 & \textbf{until} \ B = \emptyset
\end{array}
\end{array}
\]
Both BFS variants compute the successors or predecessors of a state exactly once, i.e., if in the course of the algorithm the oracle is called twice with arguments \(B_i\) and \(B_j\), respectively, then \(B_i \cap B_j = \emptyset\). To prove this in the forward case (the backward case is analogous), observe that \(B \subseteq S\) is an invariant of the repeat loop, and that the value of \(S\) never decreases. Now, let \(B_1, S_1, B_2, S_2, \ldots\) be the sequence of values of the variables \(B\) and \(S\) right before the \(i\)-th execution of line 3. We have \(B_i \subseteq S_i\) by the invariant, \(S_i \subseteq S_j\) for every \(j \geq i\), and and \(B_{j+1} \cap S_j = \emptyset\) by line 3. So \(B_j \cap B_i = \emptyset\) for every \(j > i\).

As data structures for the sets \(S\) and \(B\) we can use a hash table and a queue, respectively. But we can also take the set \(Q\) of states of \(A\) as finite universe, and use automata for fixed-length languages to represent both \(S\) and \(B\). Moreover, we can represent \(\delta \subseteq Q \times Q\) by a finite transducer \(T_\delta\), and reduce the computation of \(\delta(B)\) and \(\delta^{-1}(B)\) in line 3 to computing \(\text{Post}(B, \delta)\) and \(\text{Pre}(B, \delta)\), respectively.

### 13.2.1 Emerson-Lei’s algorithm

A state \(q\) of \(A\) is live if some infinite path starting at \(q\) visits accepting states infinitely often. Clearly, \(A\) is nonempty if and only if its initial state is live. We describe an algorithm due to Emerson and Lei for computing the set of live states. For every \(n \geq 0\), the \(n\)-live states of \(A\) are inductively defined as follows:

- every state is 0-live;
- a state \(q\) is \((n+1)\)-live if some path containing at least one transition leads from \(q\) to an accepting \(n\)-live state.

Loosely speaking, a state is \(n\)-live if starting at it it is possible to visit accepting states \(n\)-times. Let \(L[n]\) denote the set of \(n\)-live states of \(A\). We have:

**Lemma 13.20**  
(a) \(L[n] \supseteq L[n+1]\) for every \(n \geq 0\).

(b) The sequence \(L[0] \supseteq L[1] \supseteq L[2] \ldots\) reaches a fixpoint \(L[i]\) (i.e., there is a least index \(i \geq 0\) such that \(L[i+1] = L[i]\)), and \(L[i]\) is the set of live states.

**Proof:** We prove (a) by induction on \(n\). The case \(n = 0\) is trivial. Assume \(n > 0\), and let \(q \in L[n+1]\). There is a path containing at least one transition that leads from \(q\) to an accepting state \(r \in L[n]\). By induction hypothesis, \(r \in L[n-1]\), and so \(q \in L[n]\).

To prove (b), first notice that, since \(Q\) is finite, the fixpoint \(L[i]\) exists. Let \(L\) be the set of live states. Clearly, \(L \subseteq L[i]\) for every \(i \geq 0\). Moreover, since \(L[i] = L[i+1]\), every state of \(L[i]\) has a proper descendant that is accepting and belongs to \(L[i]\). So \(L[i] \subseteq L\).

Emerson-Lei’s algorithm computes the fixpoint \(L[i]\) of the sequence \(L[0] \supseteq L[1] \supseteq L[2] \ldots\). To compute \(L[n+1]\) given \(L[n]\) we observe that a state is \(n+1\)-live if some nonempty path leads from it to an \(n\)-live accepting state, and so

\[ L[n+1] = \text{BackwardBFS}(\text{Pre}(L[n] \cap F)) \]
The pseudocode for the algorithm is shown below on the left-hand-side; the variable $L$ is used to store the elements of the sequence $L[0], L[1], L[2], \ldots$.

**EmersonLei(A)**

- **Input:** NBA $A = (Q, \Sigma, \delta, q_0, F)$
- **Output:** EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

1. $L \leftarrow Q$
2. repeat
3. $OldL \leftarrow L$
4. $L \leftarrow \text{Pre}(OldL \cap F)$
5. $L \leftarrow \text{BackwardBFS}(L)$
6. until $L = OldL$
7. if $q_0 \in L$ then report NEMP
8. else report NEMP

The repeat loop is executed at most $|Q| + 1$-times, because each iteration but the last one removes at least one state from $L$. Since each iteration takes $O(|Q| + |\delta|)$ time, the algorithm runs in $O(|Q| \cdot (|Q| + |\delta|))$ time.

The algorithm may compute the predecessors of a state twice. For instance, if $q \in F$ and there is a transition $(q, q)$, then after line 4 is executed the state still belongs to $L$. The version on the right avoids this problem.

Emerson-Lei’s algorithm can be easily generalized to NGAs (we give only the generalization of the first version):

**GenEmersonLei(A)**

- **Input:** NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{m-1}\})$
- **Output:** EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

1. $L \leftarrow Q$
2. repeat
3. $OldL \leftarrow L$
4. for $i=0$ to $m-1$
5.   $L \leftarrow \text{Pre}(OldL \cap F_i)$
6.   $L \leftarrow \text{BackwardBFS}(L)$
7. until $L = OldL$
8. if $q_0 \in L$ then report NEMP
9. else report NEMP

**Proposition 13.21** GenEmersonLei(A) reports NEMP iff $A$ is nonempty.
Proof: For every $k \geq 0$, redefine the $n$-live states of $A$ as follows: every state is 0-live, and $q$ is $(n+1)$-live if some path having at least one transition leads from $q$ to a $n$-live state of $F_{(n \mod m)}$. Let $L[n]$ denote the set of $n$-live states. Proceeding as in Lemma 13.20, we can easily show that $L[(n+1) \cdot m] \supseteq L[n \cdot m]$ holds for every $n \geq 0$.

We claim that the sequence $L[0] \supseteq L[m] \supseteq L[2 \cdot m] \ldots$ reaches a fixpoint $L[i \cdot m]$ (i.e., there is a least index $i \geq 0$ such that $L[(i+1) \cdot m] = L[i \cdot m]$), and $L[i \cdot m]$ is the set of live states. Since $Q$ is finite, the fixpoint $L[i \cdot m]$ exists. Let $q$ be a live state. There is a path starting at $q$ that visits $F_j$ infinitely often for every $j \in \{0, \ldots, m-1\}$. In this path, every occurrence of a state of $F_j$ is always followed by some later occurrence of a state of $F_{(j+1) \mod m}$, for every $i \in \{0, \ldots, m-1\}$. So $q \in L[i \cdot m]$. We now show that every state of $L[i \cdot m]$ is live. For every state $q \in L[(i+1) \cdot m]$ there is a path $\pi = \pi_{m-1} \pi_{m-2} \pi_0$ such that for every $j \in \{0, \ldots, m-1\}$ the segment $\pi_j$ contains at least one transition and leads to a state of $L[i \cdot m + j] \cap F_j$. In particular, $\pi$ visits states of $F_0, \ldots, F_{m-1}$, and, since $L[(i+1) \cdot m] = L[i \cdot m]$, it leads from a state of $L[(i+1) \cdot m]$ to another state of $L[(i+1) \cdot m]$. So every state of $L[(i+1) \cdot m] = L[i \cdot m]$ is live, which proves the claim.

Since $GenEmersonLei(A)$ computes the sequence $L[0] \supseteq L[m] \supseteq L[2 \cdot m] \ldots$, after termination $L$ contains the set of live states.

13.2.2 A Modified Emerson-Lei’s algorithm

There exist many variants of Emerson-Lei’s algorithm that have the same worst-case complexity, but try to improve the efficiency, at least in some cases, by means of heuristics. We present here one of these variants, which we call the Modified Emerson-Lei’s algorithm (MEL).

Given a set $S \subseteq Q$ of states, let $inf(S)$ denote the states $q \in S$ such that some infinite path starting at $q$ contains only states of $S$. Instead of computing $Pre(OldL \cap F)$ at each iteration step, MEL computes $Pre(inf(OldL) \cap F_i)$.
MEL(A)

Input: NGA A = (Q, Σ, δ, q₀, {F₀, ..., Fₖ₋₁})

Output: EMP if \( Lω(A) = ∅ \), NEMP otherwise

1. \( L \leftarrow Q; \)
2. repeat
3. \( \text{OldL} \leftarrow L \)
4. \( L \leftarrow \text{inf}(\text{OldL}) \)
5. \( L \leftarrow \text{Pre}(L \cap F) \)
6. \( L \leftarrow \text{BackwardBFS}(L) \)
7. until \( L = \text{OldL} \)
8. if \( q₀ \in L \) then report NEMP
9. else report NEMP

function \( \text{inf}(S) \)

10. repeat
11. \( \text{OldS} \leftarrow S \)
12. \( S \leftarrow S \cap \text{Pre}(S) \)
13. until \( S = \text{OldS} \)
14. return \( S \)

In the following we show that MEL is correct, and then compare it with Emerson-Lei’s algorithm. As we shall see, while MEL introduces the overhead of repeatedly computing \( \text{inf} \)-operations, it still makes sense in many cases because it reduces the number of executions of the repeat loop.

To prove correctness we claim that after termination \( L \) contains the set of live states. Recall that the set of live states is the fixpoint \( L[i] \) of the sequence \( L[0] \supseteq L[1] \supseteq L[2] \ldots \). By the definition of liveness we have \( \text{inf}(L[i]) = L[i] \). Define now \( L'[0] = Q \), and \( L'[n + 1] = \text{inf}(\text{pre}⁺(L'[i] \cap α)) \). Clearly, MEL computes the sequence \( L'[0] \supseteq L'[1] \supseteq L'[2] \ldots \). Since \( L[n] \supseteq L'[n] \supseteq L[i] \) for every \( n > 0 \), we have that \( L[i] \) is also the fixpoint of the sequence \( L'[0] \supseteq L'[1] \supseteq L'[2] \ldots \), and so MEL computes \( L[i] \). Since \( \text{inf}(S) \) can be computed in time \( O(|Q| + |δ|) \) for any set \( S \), MEL runs in \( O(|Q| \cdot (|Q| + |δ|)) \) time.

Interestingly, we have already met Emerson-Lei’s algorithm in Chapter ???. In the proof of Proposition 12.3 we defined a sequence \( D₀ \supseteq D₁ \supseteq D₂ \supseteq \ldots \) of infinite acyclic graphs. In the terminology of this chapter, \( D_{2i+1} \) was obtained from \( D_{2i} \) by removing all nodes having only finitely many descendants, and \( D_{2i+2} \) was obtained from \( D_{2i+1} \) by removing all nodes having only non-accepting descendants. This corresponds to \( D_{2i+1} := \text{inf}(D_{2i}) \) and \( D_{2i+2} := \text{pre}⁺(D_{2i+1} \cap α) \). So, in fact, we can look at this procedure as the computation of the live states of \( D₀ \) using MEL.
13.2.3 Comparing the algorithms

We give two families of examples showing that MEL may outperform Emerson-lei’s algorithm, but not always.

**A good case for MEL.** Consider the automaton of Figure 13.8. The \(i\)-th iteration of Emerson-Lei’s algorithm removes \(q_{n-i+1}\). The number of calls to BackwardBFS is \((n + 1)\), although a simple modification allowing the algorithm to stop if \(L = \emptyset\) spares the \((n + 1)\)-th operation. On the other hand, the first \(inf\)-operation of MEL already sets the variable \(L\) to the empty set of states, and so, with the same simple modification, the algorithm stops after one iteration.

![Figure 13.8: An example in which the MEL-algorithm outperforms the Emerson-Lei algorithm](image)

**A good case for Emerson-Lei’s algorithm.** Consider the automaton of Figure 13.9. The \(i\)-th iteration of Emerson-Lei’s algorithm removes \(q_{(n-i+1),1}\) and \(q_{(n-i+1),2}\), and so the algorithm calls BackwardBFS \((n + 1)\) times. The \(i\)-th iteration of MEL-algorithm removes no state as result of the \(inf\)-operation, and states \(q_{(n-i+1),1}\) and \(q_{(n-i+1),2}\) as result of the call to BackwardBFS. So in this case the \(inf\)-operations are all redundant.

![Figure 13.9: An example in which the EL-algorithm outperforms the MEL-algorithm](image)

**Exercises**

**Exercise 96** Which lassos of the following NBA can be found by a run of NestedDFS? (Recall that NestedDFS is a non-deterministic algorithm, and so different runs on the same input may return different lassos.)
Exercise 97  A Büchi automaton is weak if no strongly connected component contains both accepting and non-accepting states. Show that the following algorithm correctly decides emptiness of weak Büchi automata.

*Hint.* Consider the root of a scc containing only accepting states.

```
SingleDFS(A)
Input: weak NBA A = (Q, Σ, δ, q₀, F)
Output: EMP if Lω(A) = ∅, NEMP otherwise
1  S ← ∅
2  dfs(q₀)
3  report EMP
4  proc dfs(q)
5    add q to S
6    for all r ∈ δ(q) do
7      if r ∉ S then dfs(r)
8      if r ∈ F then report NEMP
9  return
```

Exercise 98  Consider Muller automata whose accepting condition contains one single set of states F, i.e., a run ρ is accepting if inf(ρ) = F. Transform TwoStack into a linear algorithm for checking emptiness of these automata.

*Hint:* Consider the version of TwoStack for NGAs.

Exercise 99

(1) Given R, S ⊆ Q, define pre⁺(R, S) as the set of ascendants q of R such that there is a path from q to R that contains only states of S. Give an algorithm to compute pre⁺(R, S).
(2) Consider the following modification of Emerson-Lei’s algorithm:

\[ MEL2(A) \]

**Input:** NBA \( A = (Q, \Sigma, \delta, q_0, F) \)

**Output:** EMP if \( L_\omega(A) = \emptyset \), NEMP otherwise

1. \( L \leftarrow Q \)
2. repeat
3. \( \text{OldL} \leftarrow L \)
4. \( L \leftarrow \text{pre}^+(L \cap F, L) \)
5. until \( L = \text{OldL} \)
6. if \( q_0 \in L \) then report NEMP
7. else report NEMP

Is \( MEL2 \) correct? What is the difference between the sequences of sets computed by \( MEL \) and \( MEL2 \)?
Chapter 14

Applications I: Verification and Temporal Logic

Recall that, intuitively, liveness properties are those stating that the system will eventually do something good. More formally, they are properties that are only violated by infinite executions of the systems, i.e., by examining only a finite prefix of an infinite execution it is not possible to determine whether the infinite execution violates the property or not. In this chapter we apply the theory of Büchi automata to the problem of automatically verifying liveness properties.

14.1 Automata-Based Verification of Liveness Properties

In Chapter 8 we introduced some basic concepts about systems: configuration, possible execution, and execution. We extend these notions to the infinite case. An $\omega$-execution of a system is an infinite sequence $c_0c_1c_2\ldots$ of configurations where $c_0$ is some initial configuration, and for every $i \geq 1$ the configuration $c_i$ is a legal successor according to the semantics of the system of the configuration $c_{i-1}$. Notice that according to this definition, if a configuration has no legal successors then it does not belong to any $\omega$-execution. Usually this is undesirable, and it is more convenient to assume that such a configuration $c$ has exactly one legal successor, namely $c$ itself. In this way, every reachable configuration of the system belongs to some $\omega$-execution. The terminating executions are then the $\omega$-executions of the form $c_0\ldots c_{n-1}c_n^\omega$ for some terminating configuration $c_n$. The set of terminating configurations can usually be identified syntactically. For instance, in a program the terminating configurations are usually those in which control is at some particular program line.

In Chapter 8 we showed how to construct a system NFA recognizing all the executions of a given system. The same construction can be used to define a system NBA recognizing all the $\omega$-executions.

Example 14.1 Consider the little program of Chapter 8.
Figure 14.1: System NBA for the program

1. while \( x = 1 \) do
   2. if \( y = 1 \) then
   3. \( x \leftarrow 0 \)
   4. \( y \leftarrow 1 - x \)
5. end

Its system NFA is the automaton of 14.1, but without the red self-loops at states \([5, 0, 0]\) and \([5, 0, 1]\). The system NBA is the result of adding the self-loops.

### 14.1.1 Checking Liveness Properties

In Chapter 8 we used Lamport’s algorithm to present examples of safety properties, and how they can be automatically checked. We do the same now for liveness properties. Figure 14.2 shows again the network of automata modelling the algorithm and its asynchronous product, from which we can easily gain its system NBA. Observe that in this case every configuration has at least a successor, and so no self-loops need to be added.

For \( i \in \{0, 1\} \), let \( NC_i, T_i, C_i \) be the sets of configurations in which process \( i \) is in the non-critical section, is trying to access the critical section, and is in the critical section, respectively, and let \( \Sigma \) stand for the set of all configurations. The finite waiting property for process \( i \) states that if process \( i \) tries to access its critical section, it eventually will. The possible \( \omega \)-executions that violate the property for process \( i \) are represented by the \( \omega \)-regular expression

\[
v_i = \Sigma^* T_i (\Sigma \setminus C_i)^\omega.
\]

We can check this property using the same technique as in Chapter 8. We construct the system NBA \( \omega E \) recognizing the \( \omega \)-executions of the algorithm (the NBA has just two states), and transform the regular expression \( v_i \) into an NBA \( V_i \) using the algorithm of Chapter 11. We then
14.1. AUTOMATA-BASED VERIFICATION OF LIVENESS PROPERTIES

Figure 14.2: Lamport’s algorithm and its asynchronous product.
construct an NBA for $\omega E \cap V_i$ using intersNBA(), and check its emptiness using one of the algorithms of Chapter 13.

Observe that, since all states of $\omega E$ are accepting, we do not need to use the special algorithm for intersection of NBAs, and so we can apply the construction for NFAs.

The result of the check for process 0 yields that the property fails because for instance of the $\omega$-execution

$$[0, 0, nc_0, nc_1] [1, 0, t_0, nc_1] [1, 1, t_0, 1]^\omega$$

In this execution both processes request access to the critical section, but from then on process 1 never makes any further step. Only process 0 continues, but all it does is continuously check that the current value of $b_1$ is 1. Intuitively, this corresponds to process 1 breaking down after requesting access. But we do not expect the finite waiting property to hold if processes may break down while waiting. So, in fact, our definition of the finite waiting property is wrong. We can repair the definition by reformulating the property as follows: in any $\omega$-execution in which both processes execute infinitely many steps, if process 0 tries to access its critical section, then it eventually will. The condition that both processes must move infinitely often is called a fairness assumption.

The simplest way to solve this problem is to enrich the alphabet of the system NBA. Instead of labeling a transition only with the name of the target configuration, we also label it with the number of the process responsible for the move leading to that configuration. For instance, the transition $[0, 0, nc_0, nc_1] \xrightarrow{[1,0,0,nc_1]} [1, 0, t_0, nc_1]$ becomes

$$[0, 0, nc_0, nc_1] \xrightarrow{([1,0,0,nc_1],0)} [1, 0, t_0, nc_1]$$

to reflect the fact that $[1, 0, t_0, nc_1]$ is reached by a move of process 0. So the new alphabet of the NBA is $\Sigma \times \{0, 1\}$. If we denote $M_0 = \Sigma \times \{0\}$ and $M_1 = \Sigma \times \{1\}$ for the ‘moves’ of process 0 and process 1, respectively, then the regular expression

$$\text{inf} = ( (M_0 + M_1)^* M_0 M_1 )^\omega$$

represents all $\omega$-executions in which both processes move infinitely often, and $L(v_i) \cap L(\text{inf})$ (where $v_i$ is suitably rewritten to account for the larger alphabet) is the set of violations of the reformulated finite waiting property. To check if some $\omega$-execution is a violation, we can construct NBAs for $v_i$ and $\text{inf}$, and compute their intersection. For process 0 the check yields that the property indeed holds. For process 1 the property still fails because of, for instance, the sequence

$$( [0, 0, nc_0, nc_1] [0, 1, nc_0, t_1] [1, 1, t_0, t_1] [1, 1, t_0, q_1] [1, 0, t_0, q'_1] [1, 0, c_0, q'_1] [0, 0, nc_0, q'_1] )^\omega$$

in which process 1 repeatedly tries to access its critical section, but always lets process 0 access first.
14.2 Linear Temporal Logic

In Chapter 8 and in the previous section we have formalized properties of systems using regular, or $\omega$-regular expressions, NFAs, or NBAs. This becomes rather difficult for all but the easiest properties. For instance, the NBA or the $\omega$-regular expression for the modified finite waiting property are already quite involved, and it is difficult to be convinced that they correspond to the intended property. In this section we introduce a new language for specifying safety and liveness properties, called Linear Temporal Logic (LTL). LTL is close to natural language, but still has a formal semantics.

Formulas of LTL are constructed from a set $AP$ of atomic propositions. Intuitively, atomic propositions are abstract names for basic properties of configurations, whose meaning is fixed only after a concrete system is considered. Formally, given a system with a set $C$ of configurations, the meaning of the atomic propositions is fixed by a valuation function $V: AP \rightarrow 2^C$ that assigns to each abstract name the set of configurations at which it holds.

Atomic propositions are combined by means of the usual Boolean operators and the temporal operators $X$ ("next") and $U$ ("until"). Formally:

**Definition 14.2** Let $AP$ be a finite set of atomic propositions. The set of LTL formulas over $AP$, denoted by $LTL(AP)$, is the set of expressions generated by the grammar

$$\varphi ::= true \mid p \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid X\varphi_1 \mid \varphi_1 U \varphi_2.$$ 

Formulas are interpreted on sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$, where $\sigma_i \subseteq AP$ for every $i \geq 0$. We call these sequences computations. The set of all computations over $AP$ is denoted by $C(AP)$. The executable computations of a system are the computations $\sigma$ for which there exists an $\omega$-execution $c_0c_1c_2\ldots$ such that for every $i \geq 0$ the set of basic properties satisfied by $c_i$ is exactly $\sigma_i$.

**Definition 14.3** Given a computation $\sigma \in C(AP)$, let $\sigma^j$ denote the suffix $\sigma_j\sigma_{j+1}\sigma_{j+2}\ldots$ of $\sigma$. The satisfaction relation $\sigma \models \varphi$ (read “$\sigma$ satisfies $\varphi$”) is inductively defined as follows:

- $\sigma \models true$.
- $\sigma \models p$ iff $p \in \sigma(0)$.
- $\sigma \models \neg \varphi$ iff $\sigma \not\models \varphi$.
- $\sigma \models \varphi_1 \land \varphi_2$ iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$.
- $\sigma \models X\varphi$ iff $\sigma^1 \models \varphi$.
- $\sigma \models \varphi_1 U \varphi_2$ iff there exists $k \geq 0$ such that $\sigma^k \models \varphi_2$ and $\sigma^i \models \varphi_1$ for every $0 \leq i < k$.

We use the following abbreviations:

- $false$, $\lor$, $\rightarrow$, and $\leftrightarrow$, interpreted in the usual way.
• $F \varphi = \text{true} U \varphi$ (“eventually $\varphi$”). According to the semantics above, $\sigma \models F \varphi$ iff there exists $k \geq 0$ such that $\sigma^k \models \varphi$.

• $G \varphi = \neg F \neg \varphi$ (“always $\varphi$” or “globally $\varphi$”). According to the semantics above, $\sigma \models G \varphi$ iff $\sigma^k \models \varphi$ for every $k \geq 0$.

The set of computations that satisfy a formula $\varphi$ is denoted by $L(\varphi)$. A system satisfies $\varphi$ if all its executable computations satisfy $\varphi$.

**Example 14.4** Consider the little program at the beginning of the chapter. We write some formulas expressing properties of the possible $\omega$-executions of the program. Observe that the system NBA of Figure 14.1 has exactly four $\omega$-executions:

$$e_1 = [1, 0, 0] [5, 0, 0]$$
$$e_2 = ([1, 1, 0] [2, 1, 0] [4, 1, 0])$$
$$e_3 = [1, 0, 1] [5, 0, 1]$$
$$e_4 = [1, 1, 1] [2, 1, 1] [3, 1, 1] [4, 0, 1] [1, 0, 1] [5, 0, 1]$$

Let $C$ be the set of configurations of the program. We choose

$${\mathit{AP}} = \{ {\it{at}}_1, {\it{at}}_2, \ldots, {\it{at}}_5, x=0, x=1, y=0, y=1 \}$$

and define the valuation function $V: {\mathit{AP}} \rightarrow 2^C$ as follows:

• $V({\it{at}}_i) = \{ [\ell, x, y] \in C \mid \ell = i \}$ for every $i \in \{1, \ldots, 5\}$.

• $V(x=0) = \{ [\ell, x, y] \in C \mid x = 0 \}$, and similarly for $x = 1, y = 0, y = 1$.

Under this valuation, $\text{at}_i$ expresses that the program is at line $i$, and $x=j$ expresses that the current value of $x$ is $j$. The executable computations corresponding to the four $\omega$-executions above are

$$\sigma_1 = \{ \text{at}_1, x=0, y=0 \} \{ \text{at}_5, x=0, y=0 \}$$
$$\sigma_2 = ( \{ \text{at}_1, x=1, y=0 \} \{ \text{at}_2, x=1, y=0 \} \{ \text{at}_4, x=1, y=0 \} )$$
$$\sigma_3 = \{ \text{at}_1, x=0, y=1 \} \{ \text{at}_5, x=0, y=1 \}$$
$$\sigma_4 = \{ \text{at}_1, x=1, y=1 \} \{ \text{at}_2, x=1, y=1 \} \{ \text{at}_3, x=1, y=1 \} \{ \text{at}_4, x=0, y=1 \} \{ \text{at}_5, x=0, y=1 \}$$

We give some examples of properties:

• $\phi_0 = x=1 \land X y=0 \land XX \text{at}_4$. In natural language: the value of $x$ in the first configuration of the execution is 1, the value of $y$ in the second configuration is 0, and in the third configuration the program is at location 4. We have $\sigma_2 \models \phi_0$, and $\sigma_1, \sigma_3, \sigma_4 \not\models \phi_0$.

• $\phi_1 = F x=0$. In natural language: $x$ eventually gets the value 0. We have $\sigma_1, \sigma_2, \sigma_4 \models \phi_1$, but $\sigma_3 \not\models \phi_1$. 
• $\phi_2 = x=0 \mathsf{U} at_5$. In natural language: $x$ stays equal to 0 until the execution reaches location 5. Notice however that the natural language description is ambiguous: Do executions that never reach location 5 satisfy the property? Do executions that set $x$ to 1 immediately before reaching location 5 satisfy the property? The formal definition removes the ambiguities: the answer to the first question is ‘no’, to the second ‘yes’. We have $\sigma_1, \sigma_3 \models \phi_2$ and $\sigma_2, \sigma_4 \not\models \phi_2$.

• $\phi_3 = y=1 \land \mathsf{F}(y=0 \land at_5) \land \neg(\mathsf{F}(y=0 \land Xy=1))$. In natural language: the first configuration satisfies $y=1$, the execution terminates in a configuration with $y=0$, and $y$ never decreases during the execution. This is one of the properties we analyzed in Chapter 8, and it is not satisfied by any $\omega$-execution.

Example 14.5 We express several properties of the Lamport-Bruns algorithm (see Chapter 8) using LTL formulas. As system NBA we use the one in which transitions are labeled with the name of the target configuration, and with the number of the process responsible for the move leading to that configuration. We take $AP = \{NC_0, T_0, C_0, NC_1, T_1, C_1, M_0, M_1\}$, with the obvious valuation.

• The mutual exclusion property is expressed by the formula

$$\mathsf{G}(\neg C_0 \lor \neg C_1)$$

The algorithm satisfies the formula.

• The property that process $i$ cannot access the critical section without having requested it first is expressed by

$$\neg(\neg T_i \mathsf{U} C_i)$$

Both processes satisfy this property.

• The na"ive finite waiting property for process $i$ is expressed by

$$\mathsf{G}(T_i \rightarrow \mathsf{FC}_i)$$

The modified version in which both processes must execute infinitely many moves is expressed

$$(\mathsf{GF}M_0 \land \mathsf{GF}M_1) \rightarrow \mathsf{G}(T_i \rightarrow \mathsf{FC}_i)$$

Observe how fairness assumptions can be very elegantly expressed in LTL. The assumption itself is expressed as a formula $\psi$, and the property that $\omega$-executions satisfying the fairness assumption also satisfy $\phi$ is expressed by $\psi \rightarrow \phi$.

None of the processes satisfies the na"ive version of the finite waiting property. Process 0 satisfies the modified version, but process 1 does not.
• The bounded overtaking property for process 0 is expressed by

\[ G(T_0 \rightarrow (\neg C_1 U (C_1 U (\neg C_1 U C_0)))) \]

The formula states that whenever \( T_0 \) holds, the computation continues with a (possibly empty!) interval at which we see \( \neg C_1 \) holds, followed by a (possibly empty!) interval at which \( C_1 \) holds, followed by a point at which \( C_0 \) holds. The property holds.

14.3 From LTL formulas to generalized Büchi automata

We present an algorithm that, given a formula \( \varphi \in LTL(AP) \) returns a NGA \( A_\varphi \) over the alphabet \( 2^AP \) recognizing \( L(\varphi) \), and then derive a fully automatic procedure that, given a system and an LTL formula, decides whether the executable computations of the system satisfy the formula.

14.3.1 Satisfaction sequences and Hintikka sequences

We introduce the notion of satisfaction sequence and Hintikka sequence for a computation \( \sigma \) and a formula \( \phi \).

**Definition 14.6** Given a formula \( \phi \), the negation of \( \phi \) is the formula \( \psi \) if \( \phi = \neg \psi \), and the formula \( \neg \phi \) otherwise. The closure \( cl(\phi) \) of a formula \( \phi \) is the set containing all subformulas of \( \phi \) and their negations. A nonempty set \( \alpha \subseteq cl(\phi) \) is an atom of \( cl(\phi) \) if it satisfies the following properties:

1. \( \text{true} \in cl(\phi) \), then \( \text{true} \in \alpha \).
2. For every \( \phi_1 \land \phi_2 \in cl(\phi) \): \( \phi_1 \land \phi_2 \in \alpha \) if and only if \( \phi_1 \in \alpha \) and \( \phi_2 \in \alpha \).
3. For every \( \neg \phi_1 \in cl(\phi) \): \( \neg \phi_1 \in \alpha \) if and only if \( \phi_1 \notin \alpha \).

The set containing all atoms of \( cl(\phi) \) is denoted by \( at(\phi) \).

Observe that if \( \alpha \) is the set of all formulas of \( cl(\phi) \) satisfied by a computation \( \sigma \), then \( \alpha \) is necessarily an atom. For instance, condition (a1) holds because \( \sigma \) satisfies a conjunction if and only if it satisfies its conjuncts.

**Example 14.7** The closure of the formula \( p \land (p U q) \) is

\[ \{ p, \neg p, q, \neg q, p U q, \neg p U q, p \land (p U q), \neg (p \land (p U q)) \} . \]

The only two atoms containing \( p \land (p U q) \) are

\[ \{ p, q, p U q, p \land (p U q) \} \quad \text{and} \quad \{ p, \neg q, p U q, p \land (p U q) \} . \]
Let us see why. By (a1), every atom \( \alpha \) containing \( p \land (p \lor q) \) must contain \( p \) and \( p \lor q \); by (a2), an atom always contains either a subformula or its negation, but not both, and so \( \alpha \) contains neither \( \neg p \) nor \( \neg(p \lor q) \), and exactly one of \( q \) and \( \neg q \).

**Definition 14.8** The satisfaction sequence for a computation \( \sigma \) and a formula \( \phi \) is the infinite sequence of atoms

\[
sats(\sigma, \phi) = sats(\sigma, \phi, 0) \ sats(\sigma, \phi, 1) \ sats(\sigma, \phi, 2) \ldots
\]

where \( sats(\sigma, \phi, i) \) is the atom containing the formulas of \( \text{cl}(\phi) \) satisfied by \( \sigma^i \).

Intuitively, the satisfaction sequence of a computation \( \sigma \) is obtained by “completing” \( \sigma \): while \( \sigma \) only indicates which atomic propositions hold at each point in time, the satisfaction sequence also indicates which atom holds.

**Example 14.9** Let \( \phi = p \lor q \), \( \sigma_1 = \{p\}^\omega \), and \( \sigma_2 = (\{p\}|q)^\omega \). We have

\[
sats(\sigma_1, \phi) = \{p, \neg q, \neg(p \lor q)\}^\omega
\]
\[
sats(\sigma_2, \phi) = (\{p, \neg q, p \lor q\}|\neg p, q, p \lor q\}^\omega
\]

Observe that \( \sigma \) satisfies \( \phi \) if and only if and only if \( \phi \in sats(\sigma, \phi, 0) \), i.e., if and only if \( \phi \) belongs to the first atom of \( \sigma \).

Satisfaction sequences have a semantic definition: in order to know which atom holds at a point one must know the semantics of LTL. Hintikka sequences provide a syntactic characterization of satisfaction sequences. The definition of a Hintikka sequence does not involve the semantics of LTL, i.e., someone who ignores the semantics can still decide whether a sequence is a Hintikka sequence or not. We prove that a sequence is a satisfaction sequence if and only if it is a Hintikka sequence.

**Definition 14.10** A pre-Hintikka sequence for \( \phi \) is an infinite sequence \( \alpha = \alpha_0 \alpha_1 \alpha_2 \ldots \) of atoms satisfying the following conditions for every \( i \geq 0 \):

1. For every \( X \phi \in \text{cl}(\phi) \): \( X \phi \in \alpha_i \) if and only if \( \phi \in \alpha_{i+1} \).
2. For every \( \phi_1 \lor \phi_2 \in \text{cl}(\phi) \): \( \phi_1 \lor \phi_2 \in \alpha_i \) if and only if \( \phi_2 \in \alpha_i \) or \( \phi_1 \in \alpha_i \) and \( \phi_1 \lor \phi_2 \in \alpha_{i+1} \).

A pre-Hintikka sequence is a Hintikka sequence if it also satisfies

3. For every \( \phi_1 \lor \phi_2 \in \alpha_i \), there exists \( j \geq i \) such that \( \phi_2 \in \alpha_j \).

A pre-Hintikka or Hintikka sequence \( \alpha \) matches a computation \( \sigma \) if for every atomic proposition \( p \in AP, p \in \alpha_i \) if and only if \( p \in \sigma_i \).\footnote{Loosely speaking, \( \alpha_0 \alpha_1 \alpha_2 \ldots \) matches \( \sigma \) if it extends \( \sigma \) with new formulas.}
Observe that conditions (11) and (12) are local; in order to know if \( \alpha \) satisfies them we only need to inspect every pair \( \alpha_i, \alpha_{i+1} \) of consecutive atoms. On the contrary, condition (g) is global, since the the distance between the indices \( i \) and \( j \) can be arbitrarily large.

It follows immediately from the definition above that if \( \alpha = \alpha_0\alpha_1\alpha_2 \ldots \) is a satisfaction sequence, then every pair \( \alpha_i, \alpha_{i+1} \) satisfies (11) and (12), and the sequence \( \alpha \) itself satisfies (g). So every satisfaction sequence is a Hintikka sequence. The following theorem shows that the converse also holds: Every Hintikka sequence if a satisfaction sequence.

**Theorem 14.11** Let \( \sigma \) be a computation and let \( \phi \) be a formula. The unique Hintikka sequence for \( \phi \) matching \( \sigma \) is the satisfaction sequence \( \text{sats}(\sigma, \phi) \).

**Proof:** It follows immediately from the definitions that \( \text{sats}(\sigma, \phi) \) is a Hintikka sequence for \( \phi \) matching \( \sigma \). For the other direction, let \( \alpha = \alpha_0\alpha_1\alpha_2 \ldots \) be a Hintikka sequence for \( \phi \) matching \( \sigma \), and let \( \psi \) be an arbitrary formula of \( c(\phi) \). We prove that for every \( i \geq 0 \): \( \psi \in \alpha_i \) if and only if \( \psi \in \text{sats}(\sigma, \phi, i) \). The proof is by induction on the structure of \( \psi \).

- \( \psi = \text{true} \). Then \( \text{true} \in \text{sats}(\sigma, \phi, i) \) and, since \( \alpha_i \) is an atom, \( \text{true} \in \alpha_i \).
- \( \psi = p \) for an atomic proposition \( p \). Since \( \alpha \) matches \( \sigma \), we have \( p \in \alpha_i \) if and only if \( p \in \sigma_i \).
  By the definition of satisfaction sequence, \( p \in \sigma_i \) if and only if \( p \in \text{sats}(\sigma, \phi, i) \).
- \( \psi = \phi_1 \land \phi_2 \). We have
  \[
  \phi_1 \land \phi_2 \in \alpha_i \\
  \Leftrightarrow \phi_1 \in \alpha_i \text{ and } \phi_2 \in \alpha_i \quad \text{(condition (a1))} \\
  \Leftrightarrow \phi_1 \in \text{sats}(\sigma, \phi, i) \text{ and } \phi_2 \in \text{sats}(\sigma, \phi, i) \quad \text{(induction hypothesis)} \\
  \Leftrightarrow \phi_1 \land \phi_2 \in \text{sats}(\sigma, \phi, i) \quad \text{(definition of sats(\sigma, \phi))}
  \]
- \( \psi \neq \phi_1 \lor \psi = \text{X}\phi_1 \). The proofs are very similar to the last one.
- \( \psi = \phi_1 \cup \phi_2 \). The proof is divided into two parts.
  (a) If \( \phi_1 \cup \phi_2 \in \alpha_i \), then \( \phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i) \). By condition (12) of the definition of Hintikka sequence, we have to consider two cases:
    - \( \phi_2 \in \alpha_i \). By induction hypothesis, \( \phi_2 \in \text{sats}(\sigma, \phi) \), and so \( \phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i) \).
    - \( \phi_1 \in \alpha_i \) and \( \phi_1 \cup \phi_2 \in \alpha_{i+1} \). By condition (g), there is at least one index \( j \geq i \) such that \( \phi_2 \in \alpha_j \). Let \( j_m \) be the smallest of these indices. We prove the result by induction on \( j_m - i \). If \( i = j_m \), then \( \phi_2 \in \alpha_j \), and we proceed as in the case \( \phi_2 \in \alpha_i \). If \( i < j_m \), then since \( \phi_1 \in \alpha_i \), we have \( \phi_1 \in \text{sats}(\sigma, \phi, i) \) (induction on \( \phi \)). Since \( \phi_1 \cup \phi_2 \in \alpha_{i+1} \), we have either \( \phi_2 \in \alpha_{i+1} \) or \( \phi_1 \in \alpha_{i+1} \). In the first case we have \( \phi_2 \in \text{sats}(\sigma, \phi, i + 1) \), and so \( \phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i) \). In the second case, by induction hypothesis (induction on \( j_m - i \)), we have \( \phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i + 1) \), and so \( \phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i) \).
  (b) If \( \phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i) \), then \( \phi_1 \cup \phi_2 \in \alpha_i \). We consider again two cases.
14.3. FROM LTL Formulas to Generalized Büchi Automata

- $\phi_2 \in \text{sats}(\sigma, \phi, i)$. By induction hypothesis, $\phi_2 \in \alpha_i$, and so $\phi_1 \cup \phi_2 \in \alpha_i$.
- $\phi_1 \in \text{sats}(\sigma, \phi, i)$ and $\phi_1 \cup \phi_2 \in \text{sats}(\sigma, \phi, i + 1)$. By the definition of a satisfaction sequence, there is at least one index $j \geq i$ such that $\phi_2 \in \text{sats}(\sigma, \phi, j)$. Proceed now as in case (a).

\[ \square \]

14.3.2 Constructing the NGA

Given a formula $\phi$, we construct a \textit{generalized} Büchi automaton $A_\phi$ recognizing $L(\phi)$. By the definition of a satisfaction sequence, a computation $\sigma$ satisfies $\phi$ if and only if $\phi \in \text{sats}(\sigma, \phi, 0)$ and, by Theorem 14.11, $\text{sats}(\sigma, \phi)$ is the (unique) Hintikka sequence for $\phi$ matching $\sigma$. So $A_\phi$ must recognize the computations $\sigma$ such that the first atom of the Hintikka sequence for $\phi$ matching $\sigma$ contains $\phi$.

To achieve this goal, we construct $A_\phi$ so that its runs are all the sequences

$$ q_0 \xrightarrow{\sigma_0} \alpha_0 \xrightarrow{\sigma_1} \alpha_1 \xrightarrow{\sigma_2} \ldots $$

such that $\sigma = \sigma_0 \sigma_1 \ldots$ is a computation, and $\alpha = \alpha_0 \alpha_1 \ldots$ is a pre-Hintikka sequence of $\phi$ matching $\sigma$. Moreover, the sets of accepting states (recall that $A_\phi$ is a NGA) are chosen so that a run is accepting if and only if the pre-Hintikka sequence is also a Hintikka sequence.

The condition that runs must be pre-Hintikka sequences determines the alphabet, states, transitions, and initial state of $A_\phi$:

- The alphabet of $A_\phi$ is $2^{\text{AP}}$.
- The states of $A_\phi$ (apart from the distinguished initial state $q_0$) are atoms of $\phi$.
- The output transitions of the initial state $q_0$ are the triples $q_0 \xrightarrow{\sigma_0} \alpha$ such that $\sigma_0$ matches $\alpha$ (i.e., for every atomic proposition $p \in \text{AP}$, $p \in \sigma_0$ if and only if $p \in \alpha_0$), and $\phi \in \alpha$.
- The output transitions of any other state $\alpha$ (where $\alpha$ is an atom) are the triples $\alpha \xrightarrow{\sigma} \beta$ such that $\sigma$ matches $\beta$, and the pair $\alpha, \beta$ satisfies conditions (l1) and (l2) (where $\alpha$ and $\beta$ play the roles of $\alpha_i$ resp. $\alpha_{i+1}$).

The sets of accepting states of $A_\phi$ are determined by the condition that accepting runs must exactly correspond to the Hintikka sequences. By the definition of a Hintikka sequence, we must guarantee that in every run $q_0 \xrightarrow{\sigma_0} \alpha_0 \xrightarrow{\sigma_1} \alpha_1 \xrightarrow{\sigma_2} \ldots$, for every $i \geq 0$ such that $\alpha_i$ contains a subformula $\phi_1 \cup \phi_2$, there exists $j \geq i$ such that $\phi_2 \in \alpha_j$. By condition (l2), this amounts to guaranteeing that every run contains infinitely many indices $i$ such that $\phi_2 \in \alpha_i$, or infinitely many indices $j$ such that $\neg(\phi_1 \cup \phi_2) \in \alpha_j$. So we choose the sets of accepting states as follows:

- The accepting condition contains a set $F_{\phi_1 \cup \phi_2}$ of accepting states for each subformula $\phi_1 \cup \phi_2$ of $\phi$. An atom belongs to $F_{\phi_1 \cup \phi_2}$ if it does not contain $\phi_1 \cup \phi_2$ or if it contains $\phi_2$. 
The pseudocode for the translation algorithm is shown below.

\[ \text{LTLtoNGA}(\phi) \]
\[ \text{Input: LTL-Formula } \phi \text{ over } AP \]
\[ \text{Output: NGA } A_\phi = (Q, 2^{AP}, q_0, \delta, F) \text{ with } L(A_\phi) = L(\phi) \]

1. \( Q = \{q_0\}; \delta \leftarrow \emptyset; \)
2. for all \( \alpha \in \text{at}(\phi) \) such that \( \phi \in \alpha \) do
   add \((q_0, \alpha \cap AP, \alpha)\) to \( \delta \); add \( \alpha \) to \( W \)
3. while \( W \neq \emptyset \) do
   pick \( \alpha \) from \( W \)
   for all \( \phi_1 U \phi_2 \in cl(\phi) \) do
     if \( \phi_1 U \phi_2 \notin \alpha \) or \( \phi_2 \in \alpha \) then add \( \alpha \) to \( F_{\phi_1 U \phi_2} \)
   for all \( P \subseteq AP \) do
     for all \( \beta \in \text{at}(\phi) \) matching \( P \) do
       if \( \alpha, \beta \) satisfies (l1) and (l2) then
         add \((\alpha, P, \beta)\) to \( \delta \)
       if \( \beta \notin Q \) then add \( \beta \) to \( W \)
   \( F \leftarrow \emptyset \)
   for all \( \phi_1 U \phi_2 \in cl(\phi) \) do \( F \leftarrow F \cup \{F_{\phi_1 U \phi_2}\} \)
4. return \((Q, 2^{AP}, q_0, \delta, F)\)

**Example 14.12** We construct the automaton \( A_\phi \) for the formula \( \phi = p U q \). The closure \( cl(\phi) \) has eight atoms, corresponding to all the possible ways of choosing between \( p \) and \( \neg p \), \( q \) and \( \neg q \), \( p U q \) and \( \neg(p U q) \). However, we can easily see that the atoms \( \{p, q, \neg(p U q)\}, \{\neg p, q, \neg(p U q)\}, \) and \( \{\neg p, \neg q, p U q\} \) have no output transitions, because they would violate condition (l2). So these states can be eliminated, and we are left with the five atoms shown in Figure 14.3, plus the initial state \( q_0 \). Figure 14.3 uses some conventions to simplify the graphical representation. Observe that every transition of \( A_\phi \) leading to an atom \( \alpha \) is labeled by \( \alpha \cap AP \). Therefore, since the label of a transition can be deduced from its target state, they are omitted in the figure. Moreover, since \( \phi \) only has one subformula of the form \( \phi_1 U \phi_2 \), the NGA is in fact a NBA, and we can represent the accepting states as for NBAs.

Let \( \alpha = \{\neg p, \neg q, \neg(p U q)\} \) and \( \beta = \{p, \neg q, p U q\} \). \( A_\phi \) contains a transition \( \alpha \xrightarrow{p} \beta \) because \( \{p\} \) matches \( \beta \), and \( \alpha, \beta \) satisfy conditions (l1) and (l2). Condition (l1) holds vacuously, because \( \phi \) contains no subformulas of the form \( X \psi \); condition (l2) holds because \( p U q \notin \alpha \) and \( q \notin \beta \) and \( p \notin \alpha \). There is no transition from \( \beta \) to \( \alpha \) because it would violate condition (l2): \( p U q \in \beta \), but neither \( q \in \beta \) nor \( p U q \in \alpha \).
14.3. FROM LTL FORMULAS TO GENERALIZED BüCHI AUTOMATA

from the fact that, loosely speaking, an accepting run is the satisfaction sequence of the computation it accepts, and the satisfaction sequence of a given computation is by definition unique.

14.3.3 Reducing the NGA

The reduction algorithm for NFAs shown in Chapter 3 can be adapted to NBAs and to NGAs. Recall that given an NFA \( A = (Q, \Sigma, \delta, q_0, F) \) the algorithm computes the relation CSR, the coarsest stable refinement of the equivalence relation \( \{ F, Q \setminus F \} \), and returns the quotient \( A/CSR \). The relation CSR partitions the set of states into blocks, and the states of a block are either all final or all nonfinal (because every equivalence class of CSR is included in \( F \) or \( Q \setminus F \)). Moreover, since CSR is stable, for every two states \( q, r \) of a block of CSR and for every \((q, a, r')\) \( \in \delta \), there is a transition \((q, a, r')\) such that \( q' \), \( r' \) belong to the same block. This implies \( L(q) = L(r) \), because every run \( q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_0 \) can be “matched” by a run \( r \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \cdots r_{n-1} \xrightarrow{a_n} r_n \) in such a way that for every \( i \geq 1 \) the states \( q_i \), \( r_i \) belong to the same block, and so, in particular, \( q_n \) is final if and only if \( r_n \) is final, which implies \( a_1 \ldots a_n \in L(q) \) if and only if \( a_1 \ldots a_n \in L(r) \).

Observe that we not only have that \( q_n \) and \( r_n \) are both final or nonfinal: the same holds for every pair \( q_i \), \( r_i \). Moreover, the property also for \( \omega \)-words: every infinite run \( q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \cdots \) is “matched” by a run \( r \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \xrightarrow{a_3} r_3 \cdots \) so that for every \( i \geq 1 \) the states \( q_i \), \( r_i \) are both accepting or non-accepting. This immediately proves \( L_\omega(A) = L_\omega(A/CSR) \), and so the reduction algorithm also works for NBAs.

For the extension to NGA, let \( A = (Q, \Sigma, \delta, q_0, \{ F_1, \ldots, F_n \}) \) ba a NGA. Consider the partition of \( Q \) into blocks given by: two states \( q, r \) belong to the same block if for every \( i \in \{1, \ldots, n\} \) either \( q, r \in F_i \) or \( q, r \notin F_i \). Define CSR’ as the coarsest stable refinement of this new partition. For every two states \( q, r \) belonging to the same block of CSR’, we now have that every infinite run \( q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \cdots \) is “matched” by a run \( r \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \xrightarrow{a_3} r_3 \cdots \) so that for every \( i \geq 1 \)
and for every $j \in \{1, \ldots, n\}$ either $q_1, r_1 \in F_j$ or $q_1, r_1 \notin F_j$. So we get $L_\omega(A) = L_\omega(A/CSR')$.

The algorithm can be applied to the NBA of Figure 14.3. The relation CSR' has three equivalence classes:

$$Q_0 = \{ q_0, (p, \neg q, p \cup q) \}$$
$$Q_1 = \{ (p, q, p \cup q), (\neg p, q, p \cup q), (\neg p, \neg q, \neg(p \cup q)) \}$$
$$Q_2 = \{ (p, \neg q, \neg(p \cup q)) \}$$

which leads to the reduced NBA of Figure 14.4.

![Figure 14.4: Reduced NBA for the formula $p \cup q$.](image)

14.3.4 Size of the NGA

Let $n$ be the length of the formula $\phi$. It is easy to see that the set $cl(\phi)$ has size $O(n)$. Therefore, the NGA $A_\phi$ has at most $O(2^n)$ states. Since $\phi$ contains at most $n$ subformulas of the form $\phi_1 \cup \phi_2$, $A_\phi$ has at most $n$ sets of final states.

We now prove a matching lower bound on the number of states. We exhibit a family of formulas $\{\phi_n\}_{n \geq 1}$ such that $\phi_n$ has length $O(n)$, and every NGA recognizing $L_\omega(\phi_n)$ has at least $2^n$ states.

Consider the family of $\omega$-languages $\{D_n\}_{n \geq 1}$ over the alphabet $\{0, 1, \#\}$ given by $D_n = \{ww^{\omega} | w \in \{0, 1\}^n\}$. We first show that every NGA recognizing $D_n$ has at least $2^n$ states. Assume $A = (Q, \{0, 1, \#\}, \delta, q_0, \{F_1, \ldots, F_k\})$ recognizes $D_n$ and $|Q| < 2^n$. Then for every word $w \in \{0, 1\}^n$ there is a state $q_w$ such that $A$ accepts $w^{\omega}$ from $q_w$. By the pigeon hole principle, $q_{w_1} = q_{w_2}$ for two distinct words $w_1, w_2 \in \{0, 1\}^n$. But then $A$ accepts $w_1w_2^{\omega}$, which does not belong to $D_n$, contradicting the hypothesis.

It now suffices to construct a family of formulas $\{\phi_n\}_{n \geq 1}$ of size $O(n)$ such that $L_\omega(\phi_n) = D_n$. For this, we take $AP = \{0, 1, \#\}$ with the obvious valuation, and construct the following formulas:

- $\phi_{n1} = G((0 \lor 1 \lor \#) \land \neg(0 \land 1) \land \neg(0 \land \#) \land \neg(1 \land \#)).$
  This formula expresses that at every position exactly one proposition of $AP$ holds.

- $\phi_{n2} = \neg \# \land \bigwedge_{i=1}^{2n-1} X^i \neg \# \land X^{2n} G \#.$
  This formula expresses that $\#$ does not hold at any of the first $2n$ positions, and it holds at all later positions.
14.4 Automatic Verification of LTL Formulas

We can now sketch the procedure for the automatic verification of properties expressed by LTL formulas. The input to the procedure is

- a system NBA $A_s$ obtained either directly from the system, or by computing the asynchronous product of a network of automata;
- a formula $\phi$ of LTL over a set of atomic propositions $AP$; and
- a valuation $\nu: AP \rightarrow 2^C$, where $C$ is the set of configurations of $A_s$, describing for each atomic proposition the set of configurations at which the proposition holds.

The procedure follows these steps:

1. Compute a NGA $A_\neg$ for the negation of the formula $\phi$. $A_\neg$ recognizes all the computations that violate $\phi$.

2. Compute a NGA $A_\neg \cap A_s$ recognizing the executable computations of the system that violate the formula.

3. Check emptiness of $A_\neg \cap A_s$.

Steps (1) can be carried out by applying LTLtoNGA, and Step (3) by, say, the two-stack algorithm. For Step (2), observe first that the alphabets of $A_\neg$ and $A_s$ are different: the alphabet of $A_\neg$ is $2^{AP}$, while the alphabet of $A_s$ is the set $C$ of configurations. By applying the valuation $\nu$ we transform $A_\neg$ into an automaton with $C$ as alphabet. Since all the states of system NBAs are accepting, the automaton $A_\neg \cap A_s$ can be computed by interNFA.

It is important to observe that the three steps can be carried out simultaneously. The states of $A_\neg \cap A_s$ are pairs $[\alpha, c]$, where $\alpha$ is an atom of $\phi$, and $c$ is a configuration. The following routine takes a pair $[\alpha, c]$ as input and returns its successors in the NGA $A_\neg \cap A_s$. The routine first computes the successors of $c$ in $A_s$. Then for each successor $c'$, it computes first the set $P$ of atomic propositions satisfying $c'$ according to the valuation, and then the set of atoms $\beta$ such that (a) $\beta$ matches $P$ and (b) the pair $\alpha, \beta$ satisfies conditions (l1) and (l2). The successors of $[\alpha, c]$ are the pairs $[\beta, c']$. 

\[ \phi_{n3} = G((0 \rightarrow X^n(0 \lor \#)) \land (1 \rightarrow X^n(1 \lor \#))) \]

This formula expresses that if the atomic proposition holding at a position is 0 or 1, then $n$ positions later the atomic proposition holding is the same one, or #.

Clearly, $\phi_n = \phi_{n1} \land \phi_{n2} \land \phi_{n3}$ is the formula we are looking for. Observe that $\phi_n$ contains $O(n)$ characters.
Succ([α, c])
1  \( S \leftarrow \emptyset \)
2  for all \( c' \in \delta_s(c) \) do
3    \( P \leftarrow \emptyset \)
4  for all \( p \in AP \) do
5    if \( c' \in v(p) \) then add \( p \) to \( P \)
6  for all \( \beta \in at(\phi) \) matching \( P \) do
7    if \( \alpha, \beta \) satisfies (l1) and (l2) then add \( c' \) to \( S \)
8  return \( S \)

This routine can be inserted in the algorithm for the emptiness check. For instance, if we use TwoStack, then we just replace line 6

6  for all \( r \in \delta(q) \) do

by a call to the routine:

6  for all \([\beta, c'] \in \text{Succ}([\alpha, c])\) do

Exercises

Exercise 100  Let \( AP = \{ p, q \} \) and let \( \Sigma = 2^{AP} \). Give LTL formulas defining the following languages:

\[ \{ p, q \} \emptyset \Sigma^\omega \quad \Sigma^* \{ q \}^\omega \]
\[ \Sigma^* \{ p \} \Sigma^* \{ q \} \Sigma^\omega \quad \{ p \}^\ast \{ q \}^\ast \emptyset \Sigma^\omega \]

Exercise 101  Let \( AP = \{ p, q \} \). Give NBAs accepting the languages defined by the LTL formulas \( XG \neg p \), \( (GFp) \Rightarrow (Fq) \), and \( p \land \neg XFp \).

Exercise 102  Two formulas \( \phi, \psi \) of LTL are equivalent, denoted by \( \phi \equiv \psi \), if they are satisfied by the same computations. Which of the following equivalences hold?

\[
\begin{align*}
X\phi \land X\psi & \equiv X\phi \land \psi & X\phi \lor X\psi & \equiv X\phi \lor \psi \\
F\phi \land F\psi & \equiv F\phi \land \psi & F\phi \lor F\psi & \equiv F\phi \lor \psi \\
G\phi \land G\psi & \equiv G\phi \land \psi & G\phi \lor G\psi & \equiv G\phi \lor \psi \\
FX\phi & \equiv XF\phi & GX\phi & \equiv XG\phi \\
(\psi_1 \lor \psi_2) \lor (\psi_1 \lor \psi_2) & \equiv (\psi_1 \lor \psi_2) \lor (\psi_2 \lor \psi_1) & (\psi_1 \lor \psi_2) \lor (\psi_2 \lor \psi_1) & \equiv (\psi_1 \lor \psi_2) \lor (\psi_2 \lor \psi_1) \\
(\phi \lor \psi_1) & \lor (\phi \lor \psi_2) & \equiv (\phi \lor \psi_1) \lor (\phi \lor \psi_2) & \equiv (\phi \lor \psi_1) \lor (\phi \lor \psi_2) \\
\phi & \lor (\psi_1 \lor \psi_2) & \equiv (\phi \lor (\psi_1 \lor \psi_2) & \equiv (\phi \lor (\psi_1 \lor \psi_2) \\
\phi \lor (\psi_1 \lor \psi_2) & \equiv (\phi \lor (\psi_1 \lor \psi_2) & \equiv (\phi \lor (\psi_1 \lor \psi_2) \\
\psi \lor GF\phi & \equiv GF\phi & \psi \lor GF\phi & \equiv GF\phi \\
\psi \lor FG\phi & \equiv FG\phi & \psi \lor FG\phi & \equiv FG\phi
\end{align*}
\]
Exercise 103  Prove $FGp \equiv VFGp$ and $GFp \equiv VGFp$ for every sequence $V \in \{F, G\}^*$ of the temporal operators $F$ and $G$.

Hint: Observe that, since $FFp \equiv Fp$ and $GGp \equiv Gp$, it suffices to prove $FGp \equiv GFGp$ and $GFp \equiv FGFp$.

Exercise 104  (Santos Laboratory). The weak until operator $W$ has the following semantics:

- $\sigma |_{k} = \phi_1 W \phi_2$ iff there exists $k \geq 0$ such that $\sigma^k \models \phi_2$ and $\sigma^i \models \phi_1$ for all $0 \leq i < k$, or $\sigma^k \models \phi_2$ for every $k \geq 0$.

Prove: $p W q \equiv Gp \lor (p U q) \equiv F \neg p \rightarrow (p U q) \equiv p U (q \lor Gp)$.

Exercise 105  (Santos Laboratory). Let $AP = \{p, q, r\}$. Give formulas that hold for the computations satisfying the following properties. If in doubt about what the property really means, choose an interpretation, and explicitly indicate your choice. Here are two solved examples:

- $p$ is false before $q$: $F(q) \rightarrow (\neg p U q)$.
- $p$ becomes true before $q$: $\neg q W (p \land \neg q)$.

Now it is your turn:

- $p$ is false between $q$ and $r$.
- $p$ precedes $q$ before $r$.
- $p$ precedes $q$ after $r$.
- $p$ and $q$ eventually.

Exercise 106  (Schwoon). Which of the following formulas of LTL are tautologies? (A formula is a tautology if all computations satisfy it.) If the formula is not a tautology, give a computation that does not satisfy it.

- $Gp \rightarrow Fp$
- $G(p \rightarrow q) \rightarrow (Gp \rightarrow Gq)$
- $F(p \land q) \leftrightarrow (Fp \land Fq)$
- $\neg Fp \rightarrow F \neg Fp$
- $(Gp \rightarrow Fq) \leftrightarrow (p U (\neg p \lor q))$
- $(FGp \rightarrow GFq) \leftrightarrow G(p U (\neg p \lor q))$
- $G(p \rightarrow Xp) \rightarrow (p \rightarrow Gp)$. 
Chapter 15

Applications II: Monadic Second-Order Logic and Linear Arithmetic

In Chapter 9 we showed that the languages expressible in monadic second-order logic on finite words were exactly the regular languages, and derived an algorithm that, given a formula, constructs an NFA accepting exactly the set of interpretations of the formula. We show that this result can be easily extended to the case of infinite words: in Section 15.1 we show that the languages expressible in monadic second-order logic on \( \omega \)-words are exactly the \( \omega \)-regular languages.

In Chapter 10 we introduced Presburger Arithmetic, a logical language for expressing properties of the integers, and showed how to construct for a given formula \( \varphi \) of Presburger Arithmetic an NFA \( A_\varphi \) recognizing the solutions of \( \varphi \). In Section 15.2 we extend this result to Linear Arithmetic, a language for describing properties of real numbers with exactly the same syntax as Presburger arithmetic.

15.1 Monadic Second-Order Logic on \( \omega \)-Words

Monadic second-order logic on \( \omega \)-words has the same syntax as and a very similar semantics to its counterpart on finite words.

Definition 15.1 Let \( X_1 = \{x, y, z, \ldots \} \) and \( X_2 = \{X, Y, Z, \ldots \} \) be two infinite sets of first-order and second-order variables. Let \( \Sigma = \{a, b, c, \ldots \} \) be a finite alphabet. The set \( \text{MSO}(\Sigma) \) of monadic second-order formulas over \( \Sigma \) is the set of expressions generated by the grammar:

\[
\varphi ::= Q_\varphi(x) \mid x < y \mid x \in X \mid \neg \varphi \mid \varphi \lor \varphi \mid \exists x \varphi \mid \exists X \varphi
\]

An interpretation of a formula \( \varphi \) is a pair \((w, I)\) where \( w \in \Sigma^\omega \), and \( I \) is a mapping that assigns every free first-order variable \( x \) a position \( I(x) \in \mathbb{N} \) and every free second-order variable \( X \) a set of positions \( I(X) \subseteq \mathbb{N} \). (The mapping may also assign positions to other variables.)

The satisfaction relation \((w, I) \models \varphi\) between a formula \( \varphi \) of \( \text{MSO}(\Sigma) \) and an interpretation \((w, I)\) of \( \varphi \) is defined as follows:
(w, I) ⊨ Q_a(x) \iff w[I(x)] = a
(w, I) ⊨ x < y \iff I(x) < I(y)
(w, I) ⊨ \neg \varphi \iff (w, I) \not\models \varphi
(w, I) ⊨ \varphi_1 \lor \varphi_2 \iff (w, I) \models \varphi_1 \lor (w, I) \models \varphi_2
(w, I) ⊨ \exists x \varphi \iff |w| \geq 1 \text{ and some } i \in \mathbb{N} \text{ satisfies } (w, I[x/x]) \models \varphi
(w, I) ⊨ x \in X \iff I(x) \in I(X)
(w, I) ⊨ \exists X \varphi \iff \text{some } S \subseteq \mathbb{N} \text{ satisfies } (w, I[S/X]) \models \varphi

where \( w[i] \) is the letter of \( w \) at position \( i \), \( I[i/x] \) is the interpretation that assigns \( i \) to \( x \) and otherwise coincides with \( I \), and \( I[S/X] \) is the interpretation that assigns \( S \) to \( X \) and otherwise coincides with \( I \) — whether \( I \) is defined for \( i, X \) or not. If \( (w, I) \models \varphi \) we say that \( (w, I) \) is a model of \( \varphi \). Two formulas are equivalent if they have the same models. The language \( L(\varphi) \) of a sentence \( \varphi \in MSO(\Sigma) \) is the set \( L(\varphi) = \{ w \in \Sigma^\omega \mid w \models \varphi \} \). An \( \omega \)-language \( L \subseteq \Sigma^\omega \) is MSO-definable if \( L = L(\varphi) \) for some formula \( \varphi \in MSO(\Sigma) \).

15.1.1 Expressive power of \( MSO(\Sigma) \) on \( \omega \)-words

We show that the \( \omega \)-languages expressible in monadic second-order logic are exactly the \( \omega \)-regular languages. The proof is very similar to its counterpart for languages of finite words (Proposition 9.12), even a bit simpler.

**Proposition 15.2** If \( L \subseteq \Sigma^\omega \) is regular, then \( L \) is expressible in \( MSO(\Sigma) \).

**Proof:** Let \( A = (Q, \Sigma, \delta, q_0, F) \) be a NBA with \( Q = \{ q_0, \ldots, q_n \} \) and \( L(A) = L \). We construct a formula \( \varphi_A \) such that for every \( w \in \Sigma^\omega \), \( w \models \varphi_A \) iff \( w \in L(A) \).

We start with some notations. Let \( w = a_1 a_2 a_3 \ldots \) be an \( \omega \)-word over \( \Sigma \), and let

\[
P_q = \{ i \in \mathbb{N} \mid q \in \delta(q_0, a_0 \ldots a_i) \}.
\]

In words, \( i \in P_q \) iff \( A \) can be in state \( q \) immediately after reading the letter \( a_i \). Then \( A \) accepts \( w \) iff \( m \in \bigcup_{q \in F} P_q \).

We construct a formula \( \text{Visits}(X_0, \ldots, X_n) \) with free variables \( X_0, \ldots, X_n \) exactly as in Proposition 9.12. The formula has the property that \( I(X_i) = P_{q_i} \) holds for every model \( (w, I) \) and for every \( 0 \leq i \leq n \). In words, \( \text{Visits}(X_0, \ldots, X_n) \) is only true when \( X_i \) takes the value \( P_{q_i} \) for every \( 0 \leq i \leq n \).

So we can take

\[
\varphi_A := \exists X_0 \ldots \exists X_n \, \text{Visits}(X_0, \ldots, X_n) \land \forall x \, \exists y \left( \bigvee_{q \in F} y \in X_i \right)
\]

We proceed to prove that MSO-definable \( \omega \)-languages are regular. Given a sentence \( \varphi \in MSO(\Sigma) \), we encode an interpretation \( (w, I) \) as an \( \omega \)-word. We proceed as for finite words. Consider for instance a formula with first-order variables \( x, y \) and second-order variables \( X, Y \). Consider the interpretation
15.2 Linear Arithmetic

Linear Arithmetic is a language for describing properties of real numbers. It has the same syntax as Presburger arithmetic (see Chapter 10), but formulas are interpreted over the reals, instead of over the naturals or the integers. Given a formula $\varphi$ or real Presburger Arithmetic, we show how to construct a NBA $A_\varphi$ recognizing the solutions of $\varphi$. Section 15.2.1 discusses how to encode real numbers as strings, and Section 15.3 constructs the NBA.

15.2.1 Encoding Real Numbers

We encode real numbers as infinite strings in two steps. First we encode reals as pairs of numbers, and then these pairs as strings.
We encode a real number \( x \in \mathbb{R} \) as a pair \((x_I, x_F)\), where \( x_I \in \mathbb{Z} \), \( x_F \in [0, 1] \) and \( x = x_I + x_F \). We call \( x_I \) and \( x_F \) the integer and fractional parts of \( x \). So, for instance, \((1, 1/3)\) encodes \(4/3\), and \((-1, 2/3)\) encodes \(-1/3\) (not \(-5/3\)). Every integer is encoded by two different pairs, e.g., \(2\) is encoded by \((1, 1)\) and \((2, 0)\). We are not bothered by this. (In the standard decimal representation of real numbers integers also have two representations, for example \(2\) is represented by both \(2\) and \(0.2\).)

We encode pairs \((x_I, x_F)\) as infinite strings \(w_I \star w_F\). The string \(w_i\) is a 2-complement encoding of \(x_I\) (see Chapter 10). However, unlike Chapter 10, we use the \(msbf\) instead of the \(lsbf\) encoding (this is not essential, it leads to a more elegant construction.) So \(w_I\) is any word \(w_I = a_n a_{n-1} \ldots a_0 \ldots \in \{0, 1\}^\omega\) satisfying

\[
x_I = \sum_{i=0}^{n-1} a_i \cdot 2^i - a_0 \cdot 2^n
\]  

(15.1)

The string \(w_F\) is any \(\omega\)-word \(b_1 b_2 b_3 \ldots \in \{0, 1\}^\omega\) satisfying

\[
x_F = \sum_{i=1}^{\infty} b_i \cdot 2^{-i}
\]  

(15.2)

The only word \(b_1 b_2 b_3 \ldots\) for which we have \(x_F = 1\) is \(1^\omega\). So, in particular, the encodings of the integer \(1\) are \(0^\omega \star 0^\omega\) and \(0^\omega \star 1^\omega\). Equation 15.2 also has two solutions for fractions of the form \(2^{-k}\). For instance, the encodings of \(1/2\) are \(0^\omega \star 10^\omega\) and \(0^\omega \star 01^\omega\). Other fractions have a unique encoding: \(0^\omega \star (01)^\omega\) is the unique encoding of \(1/3\).

**Example 15.4** The encodings of \(3\) are \(0^\omega \star 011\star 0^\omega\) and \(0^\omega \star 010 \star 1^\omega\).

The encodings of \(3.\bar{3}\) are \(0^\omega \star 011 \star (01)^\omega\).

The encodings of \(–3.75\) are \(1^\omega \star 100 \star 010^\omega\) and \(1^\omega \star 100 \star 001^\omega\).

Tuples of reals are encoded using padding to make the \(\star\)-symbols fall on the same column. For instance, the encodings of the triple \((–6.75, 12.\bar{3}, 3)\) are

\[
\begin{pmatrix}
[0] & [0] & [0] & [1] & [0] & [0] & [0]
\end{pmatrix}^\omega
\]

**15.3 Constructing an NBA for the Real Solutions**

Given a real Presburger formula \(\varphi\), we construct a NBA \(A_\varphi\) accepting the encodings of the solutions of \(\varphi\). If \(\varphi\) is a negation, disjunction, or existential quantification, we proceed as in Chapter 10, replacing the operations on NFAs and transducers by operations on NBAs.

Consider now an atomic formula \(\varphi = a \cdot x \leq b\). The NBA \(A_\varphi\) must accept the encodings of all the tuples \(c \in \mathbb{R}^n\) satisfying \(a \cdot c \leq b\). We decompose the problem into two subproblems for
15.3. CONSTRUCTING AN NBA FOR THE REAL SOLUTIONS

integer and fractional parts. Given \( c \in \mathbb{R}^n \), let \( c_I \) and \( c_F \) be the integer and fractional part of \( c \) for some encoding of \( c \). For instance, if \( c = (2.3, -2.75, 1) \), then we can have \( c_I = (2, -3, 1) \) and \( c_F = (0.3, 0.25, 0) \), corresponding to the encoding, say,

\[
[010 \star (01)^\omega, 111 \star 010^\omega, 01 \star 0^\omega]
\]

or \( c_I = (2, -3, 0) \) and \( c_F = (0.3, 0.25, 1) \), corresponding to, say,

\[
[010 \star (01)^\omega, 11111 \star 001^\omega, 0 \star 1^\omega].
\]

Let \( \alpha^+ (\alpha^-) \) be the sum of the positive (negative) components of \( a \); for instance, if \( a = (1, -2, 0, 3, -1) \) then \( \alpha^+ = 4 \) and \( \alpha^- = -3 \). Since \( c_F \in [0, 1]^n \), we have

\[
\alpha^- \leq a \cdot c_F \leq \alpha^+
\]  

and therefore, if \( c \) is a solution of \( \varphi \), then

\[
a \cdot c_I + a \cdot c_F \leq b \tag{15.4}
\]

\[
a \cdot c_I \leq b - \alpha^- \tag{15.5}
\]

Putting together 15.3-15.5, we get that \( c_I + c_F \) is a solution of \( \varphi \) iff:

- \( a \cdot c_I \leq b - \alpha^+ \); or
- \( a \cdot c_I \leq \beta \) for some integer \( \beta \in [b - \alpha^+ + 1, b - \alpha^-] \) and \( a \cdot c_F \leq b - \beta \).

If we denote \( \beta^+ = b - \alpha^+ + 1 \) and \( \beta^- = b - \alpha^- \), then we can decompose the solution space of \( \varphi \) as follows:

\[
\text{Sol}(\varphi) = \{ c_I + c_F \mid a \cdot c_I < \beta^+ \}
\]

\[
\cup \bigcup_{\beta^+ \leq \beta \leq \beta^-} \{ c_I + c_F \mid a \cdot c_I = \beta \text{ and } a \cdot c_F \leq b - \beta \}
\]

**Example 15.5** We use \( \varphi = 2x - y \leq 0 \) as running example. We have

\[
\begin{align*}
\alpha^+ &= 2 & \alpha^- &= 0 \\
\beta^+ &= -1 & \beta^- &= 0
\end{align*}
\]

So \( x, y \in \mathbb{R} \) is a solution of \( \varphi \) iff:

- \( 2x_I - y_I \leq -2 \); or
- \( 2x_I - y_I = -1 \) and \( 2x_F - y_F \leq 1 \); or
- \( 2x_I - y_I = 0 \) and \( 2x_F - y_F \leq 0 \).
The solutions of \( a \cdot c_I < \beta^+ \) and \( a \cdot c_I = \beta \) can be computed using algorithms IneqZNFA and EqZNFA of Section 10.3. Recall that both algorithms use the \( lsbf \) encoding, but it is easy to transform their output into NFAs for the \( msbf \) encoding: since the algorithms deliver NFAs with exactly one final state, it suffices to reverse the transitions of the NFA, and exchange the initial and final states: the new automaton recognizes a word \( w \) iff the old one recognizes its reverse \( w^{-1} \), and so it recognizes exactly the \( msbf \)-encodings.

**Example 15.6** Figure 15.1 shows NFAs for the solutions of \( 2x_I - y_I \leq -2 \) in \( lsbf \) (left) and \( msbf \) encoding (right). The NFA on the right is obtained by reversing the transition, and exchanging the initial and final state. Figure 15.2 shows NFAs for the solutions of \( 2x_I - y_I = -1 \), also in \( lsbf \) and \( msbf \) encoding.

![Figure 15.1: NFAs for the solutions of 2x−y ≤ −2 over Z with lsbf (left) and msbf (right) encodings.](image-url)

### 15.3.1 A NBA for the Solutions of \( a \cdot x_F \leq \beta \)

We construct a DBA recognizing the solutions of \( a \cdot x_F \leq \beta \). The algorithm is similar to \( AFtoNFA \) in Section 10.2. The states of the DBA are integers. We choose transitions and final states so that the following property holds:

State \( q \in \mathbb{Z} \) recognizes the encodings of the tuples \( c_F \in [0, 1]^n \) such that \( a \cdot c_F \leq q \). \hspace{1cm} (15.6)

However, recall that \( \alpha^- \leq a \cdot c_F \leq \alpha^+ \) for every \( c_F \in [0, 1]^n \), and therefore:
15.3. CONSTRUCTING AN NBA FOR THE REAL SOLUTIONS

Figure 15.2: NFAs for the solutions of $2x - y = -1$ over $\mathbb{Z}$ with $lbsf$ (left) and $msbf$ (right) encodings.

- all states $q \geq \alpha^+$ accept all tuples of reals in $[0, 1]^n$, and can be merged with the state $\alpha^+$;
- all states $q < \alpha^-$ accept no tuples in $[0, 1]^n$, and can be merged with the state $\alpha^- - 1$.

Calling these two merged states all and none, the possible states of the DBA (not all of them may be reachable from the initial state) are

$$\text{all}, \text{ none}, \text{ and } \{q \in \mathbb{Z} \mid \alpha^- \leq q \leq \alpha^+ - 1\}.$$  \hspace{1cm}

All these states but none are final, and the initial state is $\beta$. Let us now define the set of transitions. Given a state $q$ and a letter $\zeta \in \{0, 1\}^n$, let us determine the target state $q'$ of the unique transition $q \xrightarrow{\zeta} q'$. Clearly, if $q = \text{all}$, then $q' = \text{all}$, and if $q = \text{none}$, then $q' = \text{none}$. If $q \in \mathbb{Z}$, we compute the value $v$ that $q'$ must have in order to satisfy property 15.6, and then: If $v \in [\alpha^-, \alpha^+ - 1]$, we set $q' = v$; if $v < \alpha^-$, we set $q' = \text{none}$, and if $q > \alpha^+ - 1$, we set $q' = \text{all}$. To compute $v$, recall that a word $w \in (\{0, 1\}^n)^*$ is accepted from $q'$ iff the word $\zeta w$ is accepted from $q$. So the tuple $c' \in \mathbb{R}^n$ encoded by $w$ and the tuple $c \in \mathbb{R}^n$ of real numbers encoded by $\zeta w$ are linked by the equation

$$c = \frac{1}{2} \zeta + \frac{1}{2} c'.$$  \hspace{1cm} (15.7)

Since $c'$ is accepted from $q'$ iff $c$ is accepted by $q$, to fulfil property 15.6 we must choose $v$ so that $a \cdot (\frac{1}{2} \zeta + \frac{1}{2} c') \leq q$ holds iff $a \cdot c' \leq v$ holds. We get $v = q - a \cdot \zeta$, and so we define the transition function of the DBA as follows:

$$\delta(q, \zeta) = \begin{cases} 
\text{none} & \text{if } q = \text{none} \text{ or } q - a \cdot \zeta < \alpha^- \\
q - a \cdot \zeta & \text{if } \alpha^- \leq q - a \cdot \zeta \leq \alpha^+ - 1 \\
\text{all} & \text{if } q = \text{all} \text{ or } \alpha^+ - 1 < q - a \cdot \zeta
\end{cases}$$

Example 15.7 Figure 15.3 shows the DBA for the solutions of $2x_F - y_F \leq 1$ (the state none has been omitted). Since $\alpha^+ = 2$ and $\alpha^- = -1$, the possible states of the DBA are all, none, and
The initial state is 1. Let us determine the target state of the transitions leaving state 1. We instantiate the definition of \( \delta(q, \zeta) \) with \( q = 1, \alpha^+ = 2 \) and \( \alpha^- = -1 \), and get
\[
\delta(1, \zeta) = \begin{cases} 
\text{none} & \text{if } 2\zeta_x + \zeta_y < -2 \\
1 - 2\zeta_x + \zeta_y & \text{if } -2 \leq -2\zeta_x + \zeta_y \leq 0 \\
\text{all} & \text{if } 0 < -2\zeta_x + \zeta_y 
\end{cases}
\]
which leads to
\[
\delta(1, \zeta) = \begin{cases} 
1 & \text{if } \zeta_x = 0 \text{ and } \zeta_y = 0 \\
\text{all} & \text{if } \zeta_x = 0 \text{ and } \zeta_y = 1 \\
-1 & \text{if } \zeta_x = 1 \text{ and } \zeta_y = 0 \\
0 & \text{if } \zeta_x = 1 \text{ and } \zeta_y = 1 
\end{cases}
\]
Recall that, by property 15.6, a state \( q \in \mathbb{Z} \) accepts the encodings of the pairs \((x_F, y_F) \in [0, 1]^n\) such that \( 2x_F - y_F \leq q \). This allows us to immediately derive the DBAs for \( 2x_F - y_F \leq 0 \) or \( 2x_F - y_F \leq -1 \): it is the DBA of Figure 15.3, with 0 or -1 as initial state, respectively.

![DBA for the solutions of 2x – y ≤ 1 over [0, 1] × [0, 1].](image)

**Example 15.8** Consider again \( \varphi = 2x - y \leq 0 \). Recall that \((x, y) \in \mathbb{R}^2\) is a solution of \( \varphi \) iff:

(i) \( 2x_I - y_I \leq -2 \); or

(ii) \( 2x_I - y_I = -1 \) and \( 2x_F - y_F \leq 1 \); or

(iii) \( 2x_I - y_I = 0 \) and \( 2x_F - y_F \leq 0 \).

Figure 15.4 shows at the top a DBA for the \( x, y \) satisfying (i). It is easily obtained from the NFA for the solutions of \( 2x_I - y_I \leq -2 \) shown on the right of Figure 15.1.
The DBA at the bottom of Figure 15.4 recognizes the $x, y \in \mathbb{R}$ satisfying (ii) or (iii). To construct it, we “concatenate” the DFA on the right of Figure 15.2, and the DBA of Figure 15.3. The DFA recognizes the integer solutions of $2x_I - y_I = -1$, which is adequate for (ii), but changing the final state to 0 we get a DFA for the integer solutions of $2x_I - y_I = 0$, adequate for (iii). Similarly with the DBA, and so it suffices to link state $-1$ of the DFA to state 1 of the DBA, and state 0 of the DFA to state 0 of the DBA.

Figure 15.4: DBA for the real solutions of $2x - y \leq 0$ satisfying (i) (top) and (ii) or (iii) (bottom).